# Implementation of Time Delay Control to Magnetic Bearings

by

**John Wetzel**

B.S.. Mechanical Engineering (1984)
State University of New York at Buffalo
M.S.. Mechanical Engineering (1992)
Massachusetts Institute of Technology

Submitted to the Department of
Mechanical Engineering in Partial
Fulfillment of the Requirements for the
Engineer's Degree in Mechanical Engineering

at the

Massachusetts Institute of Technology

June 1997



Signature of Author_____
Department of Mechanical Engineering
June 1997

Certified by_____
Professor Kamal Youcef-Toumi
Thesis Supervisor

Accepted by_____
Professor Ain A. Sonin
Chairman, Department Committee

# Implementation of Time Delay Control to Magnetic Bearings

by

John Wetzel

Submitted to the Department of
Mechanical Engineering in Partial
Fulfillment of the Requirements for the
Engineer's Degree in Mechanical Engineering

ABSTRACT

Magnetic bearings possess characteristics that make them ideal for solving certain classes of engineering problems. One characteristic is their ability to achieve and maintain very high rotational speeds due to low friction. Another characteristic is their inability to contaminate the working fluid due to lack of lubrication and absence of metal to metal contact. However, magnetic bearings pose many interesting challenges to the engineer. The inherent characteristics of magnetic bearings are such that they are open loop unstable and therefore require feedback control. This instability also requires that the controller be extremely robust. The application used as a test platform is a high speed turbopump. The electromagnetic fields generated by each of the five axes of the turbopump are inherently nonlinear as are the equations of motion of the rotor. The system is also subject to disturbances caused by rotor gyroscopic forces, rotor imbalance, and rotor bending modes.

This control algorithm chosen to control this turbopump application is Time Delay Control (TDC). TDC was chosen because it uses information from previous sampling interval(s) to estimate unmodeled system dynamics. This estimation is then used to augment the ideal control signal produced by a desired dynamics reference model and thereby produce the final control action. In this way, TDC becomes a powerful technique for controlling nonlinear systems and systems subject to a disturbance rich environment. However, TDC is a relatively recent control algorithm in which design has thus far been primarily a trial and error process. This thesis will document one attempt to move the design process beyond trial and error.

The three main themes of the thesis are the modeling, design, and implementation of TDC to magnetic bearings. To measure the success of the modeling, design, and implementation processes, the theoretical, digital, and analog controller closed loop frequency responses and disturbance rejection responses are compared in the case where the rotor is not spinning. Also the disturbance rejection responses are compared for both the analog and digital controllers when the rotor is spun at low and medium speeds. The results show that the digital controller has a significantly higher bandwidth, higher maximum closed loop gain, and significantly less compliance overall then the analog controller when the rotor is not spinning. The digital controller also exhibits better stiffness at low rotor speeds but begins to lose stability at medium rotor speeds.

**Acknowledgment**

Special thanks to Professor Kamal Youcef-Toumi for his guidance and infinite patience.

I would also like to thank Tom Allen, the best boss I could ever hope for. He has only one concern -- keep the sponsor happy. This allowed me to rearrange my schedule on certain weeks so that I could spend more hours of my thesis.

I would also like to thank John Maglio. While he had nothing to do with my thesis, his friendship was a major contributor to my ability to remain sane.

Thanks also to Ting-Jen Yeh my partner in crime in the lab. He represents the high degree of talent that MIT is capable of turning out. His dedication was an inspiration and his keen mind inspired me to greater heights. Without his help and encouragement, I don't believe this thesis would have been possible. I learned more from him in the lab then in all my studies at MIT.

Finally I would like to the Coca Cola Bottling Co. and the Mars Candy Co. (the manufacturer of Snickers candy bars). Together they comprise the breakfast of champions.

I've made some mistakes now baby
but I did the best I could,
it takes what it takes and sometimes
it takes longer then it should.
-Patty Smythe

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation and Background

Magnetic bearings may not be as common as their conventional counterparts but they do possess specific characteristics that make them ideal for solving certain classes of engineering problems. One such characteristic is their ability to achieve and maintain very high rotational speeds. This makes magnetic bearings ideal for such applications as flywheels. The California Zero Emissions Vehicle program is forcing auto makers to take a hard look at flywheels as energy storage elements in electric vehicles [4, 21, 5, 15]. Electric vehicles use and produce large, short term bursts of electric power. Conventional batteries are unable to meet these power demands but flywheels have no such limitation. Flywheels also have a higher energy density then conventional batteries and weight is an important consideration in vehicle design. Also flywheels have the possibility of outlasting conventional batteries which must be replaced after three years of normal use. Electric utilities are also researching the use of flywheels to meet peak energy demands. Instead of constructing costly additional powerplants, some utilities are proposing constructing flywheel substations which can be powered up during off peak night time hours for use during the day [10].

Another characteristic of magnetic bearings is that they require no lubrication which might contaminate the working fluid. They have therefore found application in turbopumps used in microprocessor production facility clean rooms. They are also used in cryogenic turbopumps where the heat produced by conventional bearings pollutes the working fluid [11]. Magnetic bearings have also found application in extremely high temperature surroundings where normal lubricants cake or burn off.

Magnet bearings are not without their own unique problems. Unlike contact bearings, magnetic bearings are active devices. This necessitates that magnetic bearings have their own power supply and controller. Also in critical applications, emergency power supplies may also be necessary. Even if emergency power is provided for, magnetic bearings usually have conventional backup bearings in case the bearings should fail or the external loading capacity is exceeded. Finally magnetic bearings are much more expensive than conventional contact bearings.

The foundation of magnetic bearings goes back to 1842 when Earnshaw demonstrated that magnetic suspension could be achieved if at least one axis was actively controlled [9]. This led directly to the two degree of freedom magnetic suspension or the semi-passive suspension. The

load capacity, stiffness, and damping of these first magnetic bearings was very poor but they did see service in some light duty applications.

It wasn't until 1957 that technology had advanced enough to allow the development of the first active magnetic suspension. By the mid 1960s, many research teams were involved in developing actively controlled magnetic suspension systems. Currently active magnetic bearings see service in high-speed centrifuges, compressors, rocket motor turbopumps, and flywheels.

As the use of magnetic bearings became more widespread, the problems of control become more apparent. Classical control theory requires that the mathematical model of the system be completely known. In practical applications involving magnetic bearings, the model parameters may be time varying or poorly known. Also magnetic bearings are expected to operate in environments where unforeseen disturbances exist. In such cases, fixed gain controllers may not provide satisfactory performance.

This thesis will be dealing with the control technique of Time Delay Control (TDC) proposed by Youcef-Toumi and Ito in 1986 [24]. TDC depends upon the direct estimation of uncertainties through time delay. TDC uses past observations of the state variables and control signal, as well as the current error signal to estimate the unmodeled dynamics of the system. This estimate is then used to augment the ideal control signal produced using a desired dynamics reference model. The use of past observations to estimate unmodeled dynamics makes this algorithm extremely powerful when applied to nonlinear plants having unknown dynamics and subject to unpredictable disturbances. Up to this point, there has been a lack of documented examples of the design process used to implement a Time Delay controller. Previous research has been conducted and guidelines have been formulated for designing a Time Delay controller [17]. However, these guidelines are vague and there is no evidence that they were ever directly used to design a controller. This paper will document the entire design process regardless of whether an adequate controller is produced or not. This will allow later researchers to benefit from the successes or failures described herein. There is however reason for optimism that a successful Time Delay controller will be designed. Previous research conducted using TDC on one axis of this same application has produced encouraging results [17, 23].

## 1.2 Scope and Content of Thesis

The main themes of this thesis deal with the modeling, design, and implementation of a digital controller using the TDC algorithm to a magnetic levitated turbopump. The modeling portion describes the procedure used to determine the theoretical model transfer function. One method used to determine this transfer function is based upon the derivation of the equations of motion for the rotor and comparing their theoretical response to the actual system response. Another method is to use the general form of the transfer function derived from the equations of motion and recursively changing the coefficients in the numerator and denominator to obtain the best fit response to the actual system response. From the results obtained by using both of these methods, the most accurate theoretical transfer function is obtained.

The design portion examines how variations in the parameters of the digital controller effect closed loop system stability in an effort to determine the optimal sampling rate for the controller. The remaining parameters of the digital controller are determined by their effect on

several important system performance measures. These measures include compliance, bandwidth, and maximum closed loop gain. These measures are of particular importance due to assumptions made during the modeling process and the operating environment to which this application is subjected.

The implementation portion addresses issues that arise from the process of mating a theoretical controller to actual hardware. These issues include control loop algorithm efficiency, integrator windup, noisy sensors, and proper filtering. This application also relied on a custom designed controller board which has its own nonstandard interface implementation and exhibits design flaws as most customized designed boards do. The effect that these anomalies have upon the controller implementation is also documented.

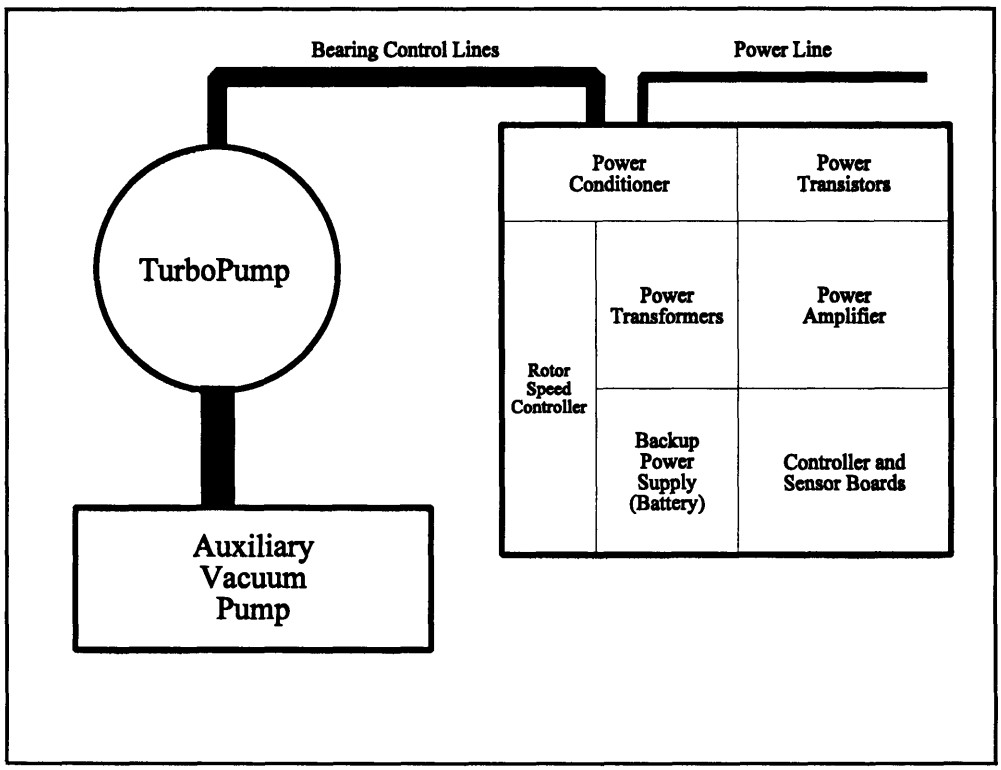
This thesis is organized into six chapters and eleven appendices. Chapter 2 describes the physical components that make up the turbopump and its subsystems. This chapter also determines the theoretical open loop transfer function from the actual system response. Chapter 3 briefly describes the Time Delay Control Law and the simplifications used to increase the computational efficiency of the algorithm. Increased efficiency decreases computation time which allows higher sampling rates. Chapter 4 describes the physical components that comprised the digital controller. This chapter also presents the controller design criteria and issues specific to implementing a digital controller using this particular hardware. Chapter 5 evaluates digital controller performance under both nonspinning and spinning conditions. Chapter 6 summarizes the findings of this thesis and presents recommendations for further research. The appendices are included to fill in the details. They are not required to understand the thesis but may be of interest to researchers duplicating or verifying this work.

# Chapter 2

# System Description and Analysis

In this chapter, a physical description of the turbopump and all pertinent subsystems is provided. The theoretical open loop response based upon the linearized equations of motion of the rotor is then compared to the actual system response of the axial bearing axis and one radial bearing axis. Next the recursive best fit open loop response is determined from the actual system response. Finally, the choice is made as to whether to use the theoretical response or the best fit response to model the open loop system.

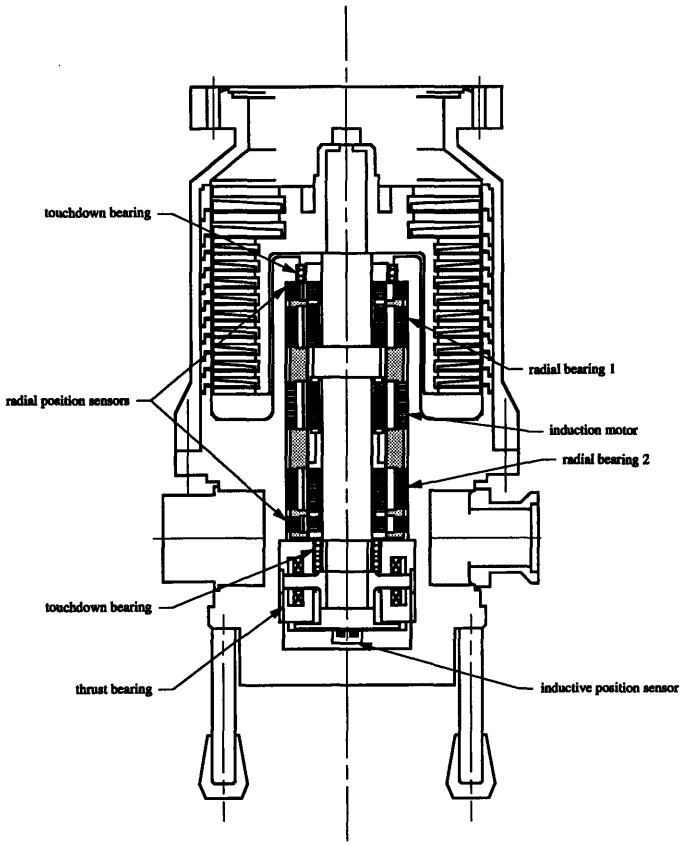
## 2.1 Physical Description




**2-1** Physical System Layout

The system is composed of three major components: 1) the turbopump, 2) the controller, and 3) the auxiliary vacuum pump (see Figure 2.1). The turbopump is connected to the controller through an umbilical cord which contains the bearing control lines, sensor carrier wave lines, sensor position signal lines, turbopump motor control lines, and turbopump motor speed sensor lines. The turbopump is also connected to the auxiliary vacuum pump through a flexible metal tube. The maximum operating speed of this turbopump is 45,000 rpm. Its normal operating speed is 30,000 rpm.

### 2.1.1 Turbopump

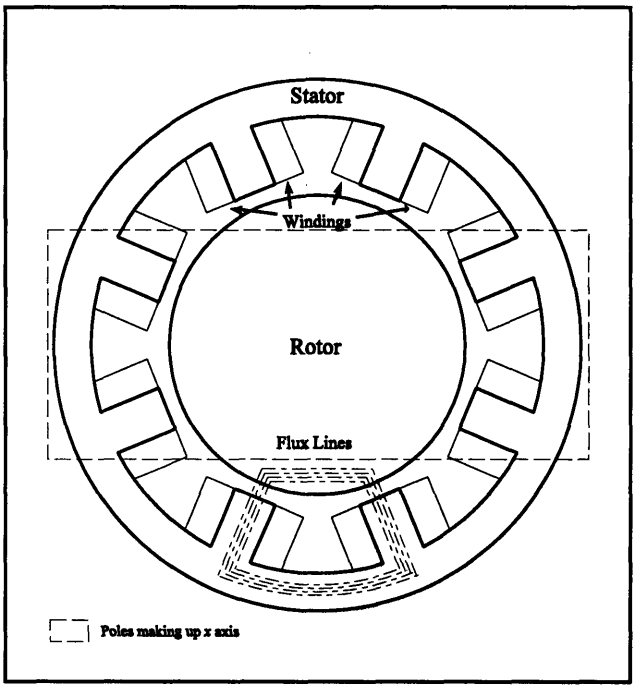



**2-2** Turbopump Cutaway

As shown in Figure 2.2, the turbopump is a multi-vane rotor suspended by two radial magnetic bearings and one axial magnetic bearing. Each magnetic bearing has an accompanying position sensor and touchdown bearing. The touchdown bearings are provided in case of controller failure and are of the conventional ball bearing variety. The rotational velocity of the rotor is governed

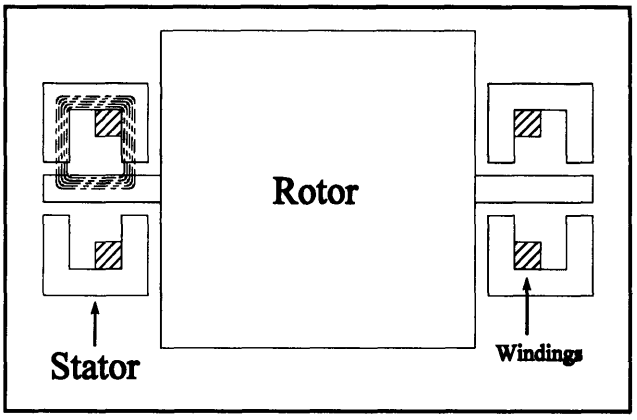
by an induction motor located midway along the rotor. The upper radial bearing is designated as radial bearing 1 and the lower radial bearing is designated radial bearing 2.

Each radial bearing is composed of a ring of laminated, ferromagnetic material having eight poles. Each pole is wound with $N$ turns of wire. Each radial bearing has an $x$ and $y$ axis composed of two opposing pole pairs. The lines of magnetic flux flow from one pole, through the rotor, back through the opposing pole, and into the ring. When the centerline of the rotor is positioned at the centerline of the radial bearing, the clearance between rotor and the poles of the radial bearing is approximately 250 μm. The clearance between the rotor and the touchdown bearings at this same position is approximately 200 μm.



**2-3** Radial Bearing Diagram

The axial bearing is composed of an upper and lower ring of ferromagnetic material having a U-shaped cross-sectional area. The inner leg of the U is wound with $N$ turns of wire. The lines of magnetic flux flow from one leg of the U, through the disk which is attached to the rotor, back through the opposite leg of the U, and into the ring. When the disk is positioned equidistant from the upper and lower axial bearing rings, the clearance is approximately 400 μm.



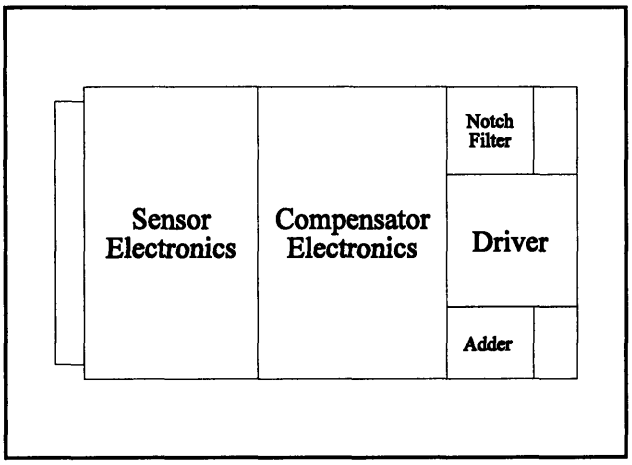
**2-4** Axial Bearing Diagram

## 2.1.2 Controller

The controller is made up of a number of subsystems (see Figure 2.1). Most of these subsystems are devoted to meeting the power demands of the turbopump. The power conditioner is responsible for converting the AC power supplied to the controller to the 24 VDC used by the controller. The power transformers convert the 24 VDC power to 15 VDC for use by the power amplifier and 12 VDC used by the controller/sensor integrated circuit boards. The backup power supply is a battery with sufficient energy to allow the system to shutdown gracefully in the event of a power outage. The rotor speed controller is comprised of two integrated circuit boards and is responsible for monitoring the rotor speed, controlling the acceleration and deceleration rates during startup and shutdown, and providing power to the rotor induction motor during normal operation. One of these boards was modified

by replacing two resistors with variable resistors. This modification allows the final operating speed of the rotor to be adjusted through trial and error. The sensor integrated circuit board is responsible for generating a 50 KHz carrier wave for use by the position sensors.

The analog compensator is located on the controller integrated circuit board. There is a controller board for each bearing of the turbopump. The radial bearing controller board is responsible for both the $x$ and $y$ axis whereas the axial bearing is responsible for only the $z$ axis and is therefore not a fully populated circuit board. The sensor electronics portion of the board (see Figure 2.5) is responsible for comparing the sensor board carrier signal to the signal returned from the position sensor. Using these two signals, the board derives the proper position signal for each axis. This position signal is then sent to a test point, and the adder. The test point allows the signal to be viewed on an oscilloscope and is also used for input into the digital controller. The adder is a modification performed by the turbopump manufacturer to allow disturbances to be injected into the position signal ahead of the compensator. From the adder, the signal leads to the compensator electronics portion of the board. After the proper control signal is determined, the compensator sends the signal through a gain before being sent to the driver. The board was modified just before the gain by adding a switch. This switch determines whether the control signal from the analog compensator or the digital controller will be used. Also after the switch, a test point was added to allow disturbances to be injected into the control signal before the driver. The gain following the switch is adjusted by a variable resistor on the edge of the board. The driver is responsible for dividing the control signal between the two opposing pole pairs that comprise an axis of a radial bearing or between the upper and lower ring of the axial bearing. The driver performs this function based upon control signal magnitude and whether the signal is negative or positive. The final element on the controller board is the notch filter. The notch filter cancels that portion of the control signal that might excite the second bending mode. It operates by isolating the necessary frequency components from the control signal, inverting these components, and adding them back into the control signal and thereby canceling their effect. This was added by the manufacturer for use by the analog compensator. However, the modification applies the cancellation signal after the analog/digital controller switch. Therefore, the board was modified further by adding another switch to cancel the contribution of this filter when an axis is under digital control.



**2-5** Magnetic Bearing Controller Board

The two signals produced by the driver are then sent to the power amplifier. The power amplifier in turn uses the power transistors to produce the final bearing coil current. The power transistors are located at the rear of the controller directly in front of the cooling fan. Each power transistor is also attached to a large heat sink.

### 2.1.3 Auxiliary Vacuum Pump

The auxiliary vacuum pump was provided by the manufacturer to ensure that the system would behave correctly when the rotor is spinning. The turbopump exhaust is blocked by a clear plastic disk that allows viewing the rotor during all phases of operation. Blocking the exhaust meant that the system would not be operating within its normal environment. To help compensate for this, a vacuum pump was attached and run prior to and during all tests in which the rotor was spinning. The auxiliary vacuum pump is not needed when the rotor is not spinning.

## 2.2 System Analysis

The determination of the transfer function for the driver and the turbopump relied on the theoretically derived rotor equations of motion and recursive, brute force techniques. The goal was to derive the open loop transfer function that would accurately represent the actual system response provided by an HP 3562A Dynamic System Analyzer. In the case of the driver, the theoretical equations representing the electrical components of the subsystem would be too difficult to derive and therefore only a recursive technique was used to derive the transfer function. The turbopump however provided the opportunity to apply both a recursive and a theoretical technique to obtain the best transfer function.

The recursive technique is based upon the researcher estimating the general form of the transfer function and recursively trying different values for the numerator and denominator until a suitably accurate transfer function was obtained. This technique can be prohibitively time consuming when applied to many high-order systems but proved relatively fast for this particular system. During each pass of the algorithm, the bode plot of the transfer function guess was compared against that of the actual system. First the estimated magnitude data was multiplied by a gain derived from the difference between DC gains of the two plots. Then at each data point, the square of the difference between the estimated and actual magnitudes was calculated and summated. The same was done for the estimated and actual phase plots as well. The summation of the total error of magnitude and phase plots produced the total error of the guess. The estimate that produced the smallest error was deemed the best fit transfer function.

The only data presented in the remainder of this chapter corresponds to the axial bearing and radial bearing 2X which is representative of the other radial bearing axes. A more detailed presentation of the system analysis methodology and data is presented in Appendix E.

### 2.2.1 Driver Best Fit System Analysis

The actual driver response was obtained by inputting a swept sine disturbance at the analog/digital controller switch test point and monitoring the driver output at a test point between the driver and the power amplifier. All tests were performed on the turbopump when the rotor was not spinning. The form of the transfer function used in the recursive analysis was derived by analyzing the actual driver magnitude and phase responses. The best fit transfer function form

is,

$$\frac{X(s)}{U(s)} = \frac{A_1}{s + A_2}$$

The values derived from the best fit recursive analysis for $A_1$ and $A_2$ are,

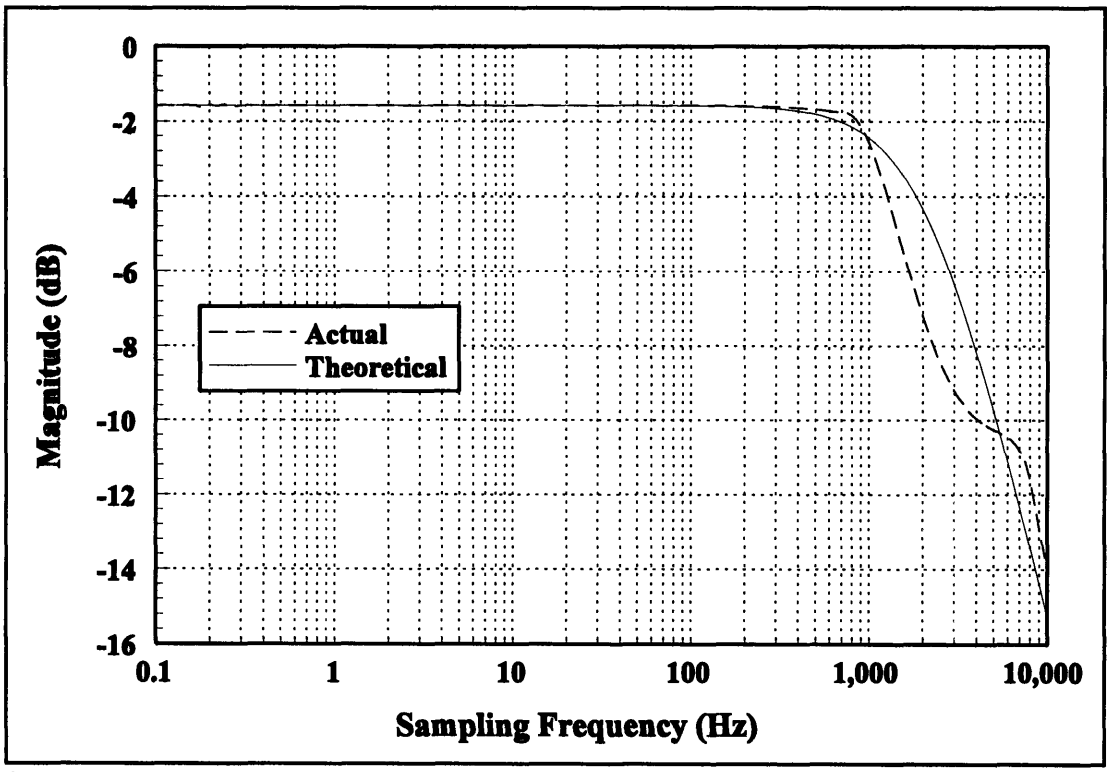
| Parameter | Axial | Rad2X |
|---|---|---|
| $A_1$ | 14707.770 | 11112.759 |
| $A_2$ | 13310.000 | 11140.000 |

In order to directly compare the response of the best fit transfer function to that of the actual system, the units of the transfer function must be comparable. Therefore the best fit transfer function uses the following conversion factors,

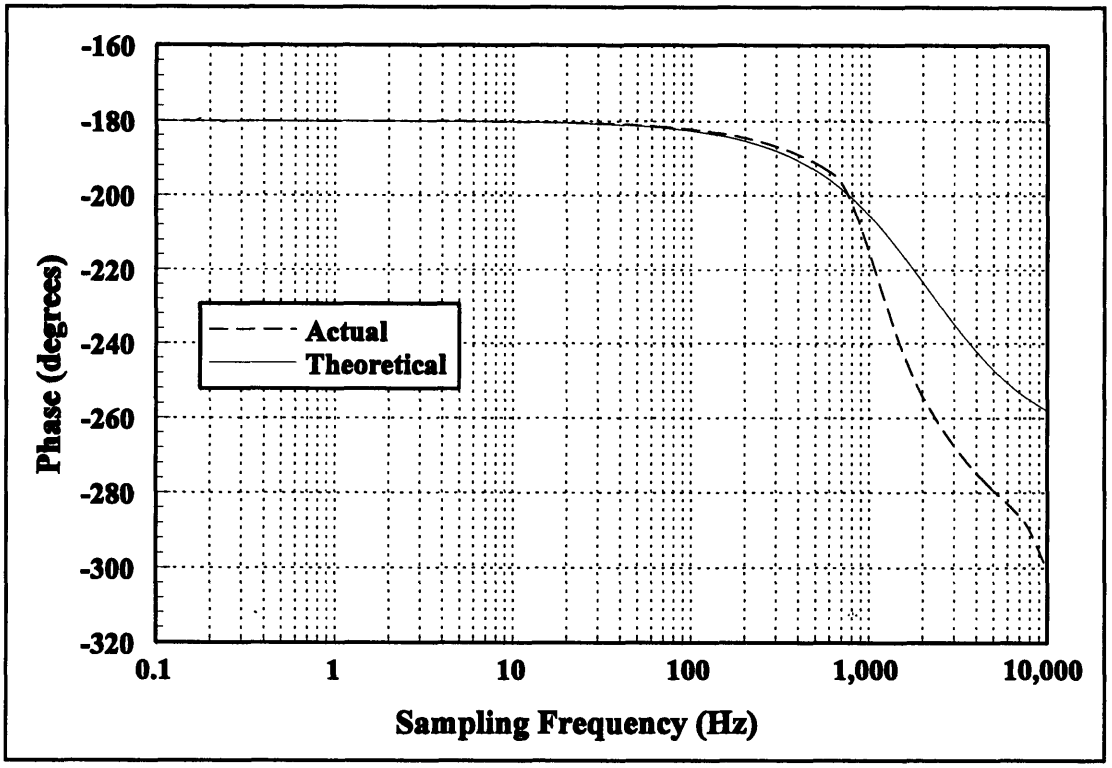
| Conversion Factor | Axial | Rad2X |
|---|---|---|
| Control Signal (A/V) | 2.776 | 1.159 |
| Driver Signal (V/A) | 0.3 | 0.3 |

Finally, mention should be made of the phase plot for the axial bearing. Normally when the bearing is at its equilibrium position, the driver produces relatively equal control currents for each of the opposing sides of the bearing. The major difference between the opposing control signals is generally that one is negative and one is positive. The test point chosen for all of the best fit analyses was purposely chosen to be positive control current test point. In this way, the phase would not have to be adjusted. However, the axial bearing must support the weight of the rotor. Due to the particular implementation of the driver, the control signal required is large enough such that the lower bearing is unpowered. Hence there is only one possible test point which can be used for the axial bearing and its value happens to be negative. Therefore the phase is shifted appropriately.

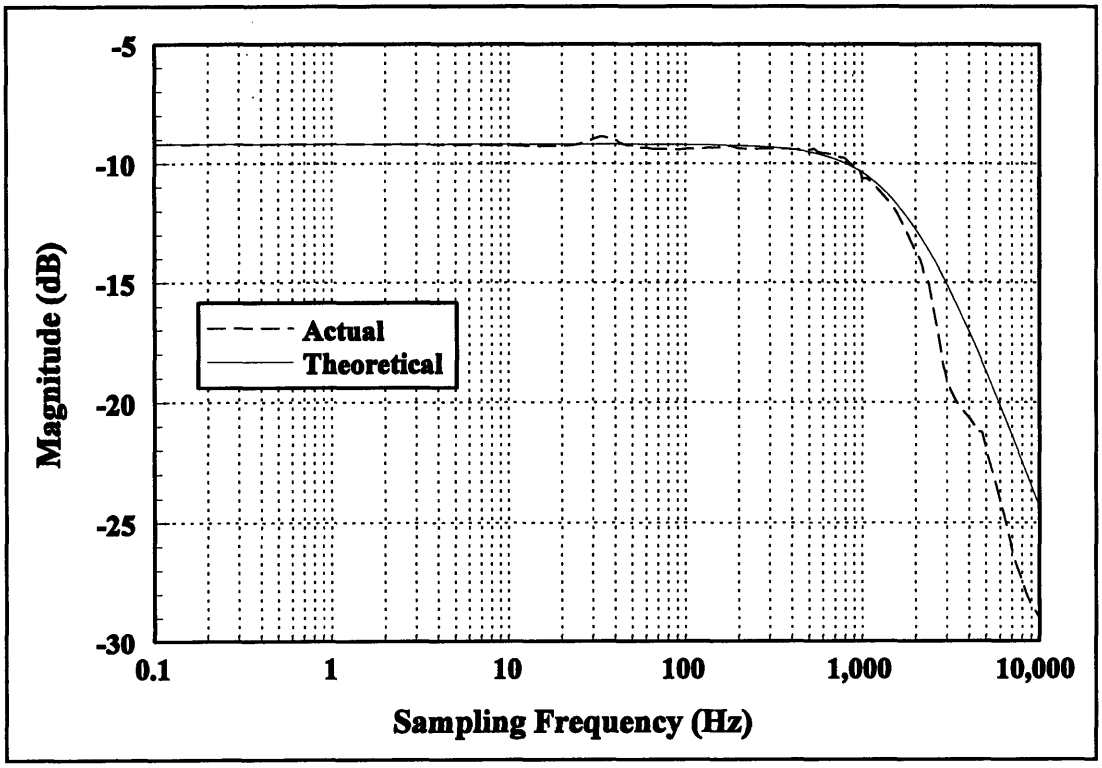
Figures 2.6, 2.7, 2.8, and 2.9 represent the best fit recursive analysis of the driver transfer function magnitude and phase plots for both the axial and 2X radial bearing.



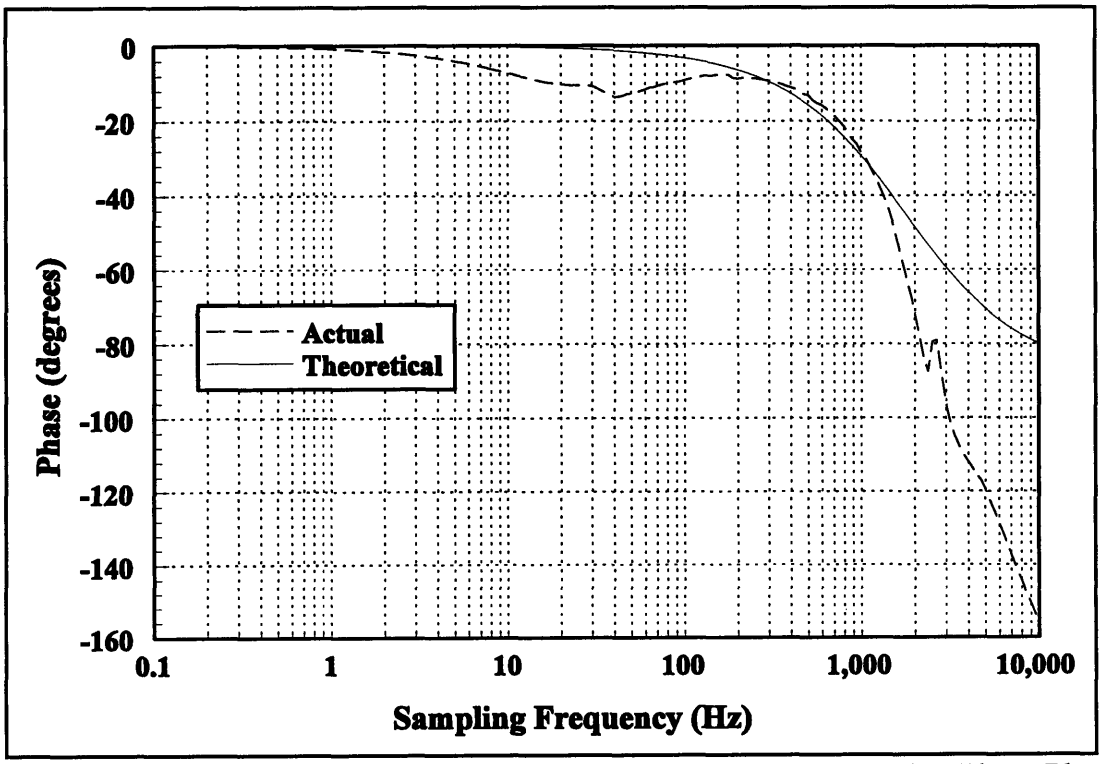

**2-6** Axial Bearing Best Fit versus Actual Driver Transfer Function Magnitude Plot




**2-7** Axial Bearing Best Fit versus Actual Driver Transfer Function Phase Plot




**2-8** Radial Bearing 2X Best Fit versus Actual Driver Transfer Function Magnitude Plot




**2-9** Radial Bearing 2X Best Fit versus Actual Driver Transfer Function Phase Plot

## 2.2.2 Theoretical Turbopump System Analysis

The actual turbopump response was obtained by inputting a swept sine disturbance at the test point between the driver and the power amplifier and monitoring the position signal at the appropriate test point. All tests were performed on the turbopump when the rotor was not spinning. The theoretical transfer function was derived from the linearized rotor equations of motion. For an in depth analysis of the rotor equations of motion, consult Appendices A, B, and C. The gyroscopic force terms of the linearized equations of motion can be ignored because the rotor was not spinning. Also the coupling forces generated by the same axis of the opposing bearing will be treated as disturbances in an effort to simplify the model and therefore they are ignored. These two assumptions create a theoretical transfer function having the following form,

$$\frac{X(s)}{U(s)} = \frac{P}{s^2 - Q}$$

The values of the variables $P$ and $Q$ were derived from the physical characteristics of the magnetic bearings,

| Axial | Radial |
|---|---|
| $P = \frac{\mu_0 N^2 A u_0}{2 m h_0^2}$ | $P = \frac{2\mu_0 N^2 A \cos\beta \left(\frac{1}{m} + \frac{b^2}{I_{rr}}\right) I_0}{h_0^2}$ |
| $Q = \frac{\mu_0 N^2 A u_0^2}{2 m h_0^3}$ | $Q = \frac{4\mu_0 N^2 A \cos\beta \left(\frac{1}{m} + \frac{b^2}{I_{rr}}\right) I_0^2}{h_0^3}$ |

Where: $\mu_0$ = air permeability
N = number of wire turns
A = magnetic flux area
$\beta$ = angle between pole and centerline of pole pair
m = rotor mass
b = distance from rotor center of gravity to bearing
$I_{rr}$ = moment of inertia, radial direction
$I_0$ = bias current
$u_0$ = axial bearing equilibrium current
$h_0$ = nominal air gap between bearing and rotor

The values of the parameters that define the physical characteristics of the each magnetic bearing were provided by the manufacturer. They are,

| Parameter | | Axial | Radial |
|---|---|---|---|
| $\mu_0$ | $(N/A^2)$ | $1.26 \times 10^{-6}$ | $1.26 \times 10^{-6}$ |
| N | | 133 | 100 |
| A | $(m^2)$ | $7.0 \times 10^{-4}$ | $9.75 \times 10^{-5}$ |
| $\beta$ | (degrees) | N/A | 22.5 |
| m | (Kg) | 2.2 | 2.2 |
| B | (m) | N/A | 0.0691 |
| $I_{rr}$ | $(Kg \cdot m^2)$ | N/A | $8.285 \times 10^{-3}$ |
| $I_0$ | (A) | N/A | 0.5 |
| $u_0$ | (A) | 1.007 | N/A |
| $h_0$ | (m) | $4.0 \times 10^{-4}$ | $2.5 \times 10^{-4}$ |

Substituting the appropriate values for the bearing characteristic parameters into the appropriate bearing equations yields,

| Parameter | Axial | Rad2X |
|---|---|---|
| P | 22.162 | 18.720 |
| Q | 55403.757 | 74881.133 |

Due to the location of available test points, the actual transfer function encompasses not only the magnetic bearings but also the power amplifier. However, the theoretical transfer function is derived from the linearized rotor equations of motion and therefore ignores the power amplifier. The contribution of the power amplifier was determined by comparing the DC gains of the actual and theoretical transfer functions. The power amplifier is assumed to be constant gain amplifier having the form $A_1/A_2$. The values obtained for the power amplifier are,

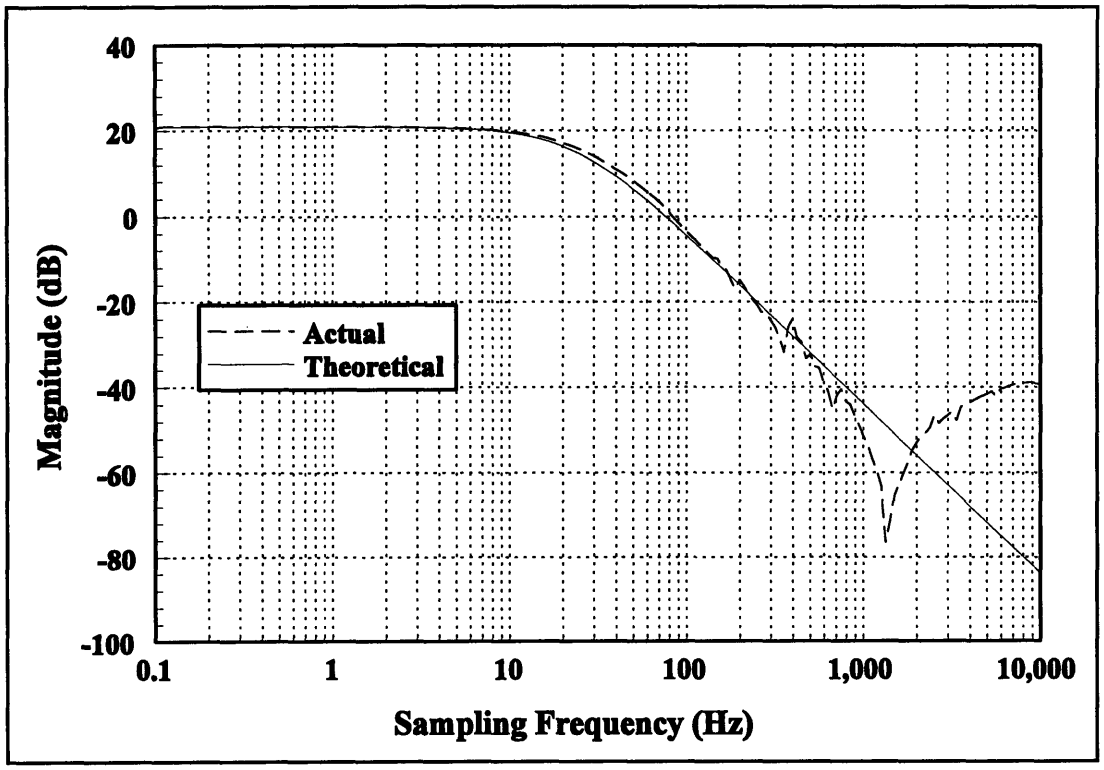
| Parameter | Axial | Rad2X |
|---|---|---|
| $A_1$ | 1.000 | 1.906 |
| $A_2$ | 1.138 | 1.000 |

Finally, in order to directly compare the response of the theoretical transfer function to that of the actual system, the units of the transfer function must be comparable. Therefore the theoretical transfer function uses the following conversion factors,

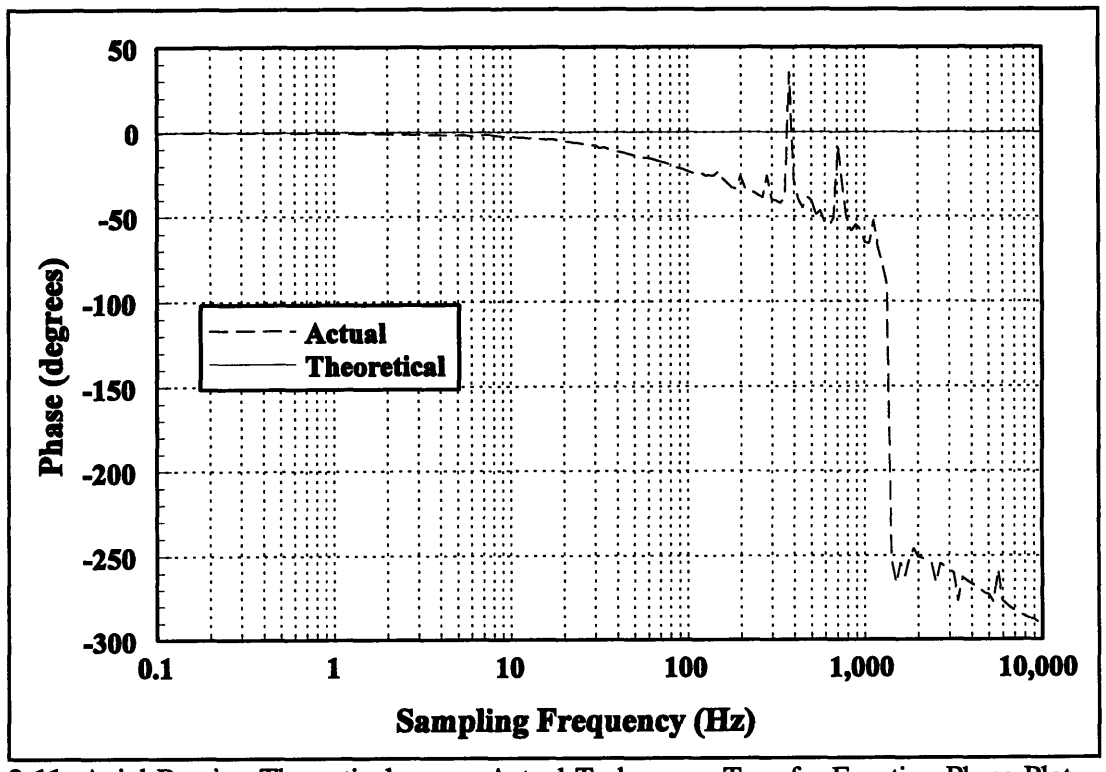
| Conversion Factor | Axial | Rad2X |
|---|---|---|
| Position Signal (V/m) | 9450.0 | 25000.0 |
| Driver Signal (V/A) | 0.3 | 0.3 |

Again, mention should be made of the phase plot for the axial bearing. Normally when the bearing is at its equilibrium position, the driver produces relatively equal control currents for each of the opposing sides of the bearing. The major difference between the opposing control signals is generally that one is negative and one is positive. The test point chosen for all of the best fit analyses was purposely chosen to be positive control current test point. In this way, the phase would not have to be adjusted. However, the axial bearing must support the weight of the rotor. Due to the particular implementation of the driver, the control signal required is large enough such that the lower bearing is unpowered. Hence there is only one possible test point which can be used for the axial bearing and its value happens to be negative. Therefore the phase is shifted appropriately.

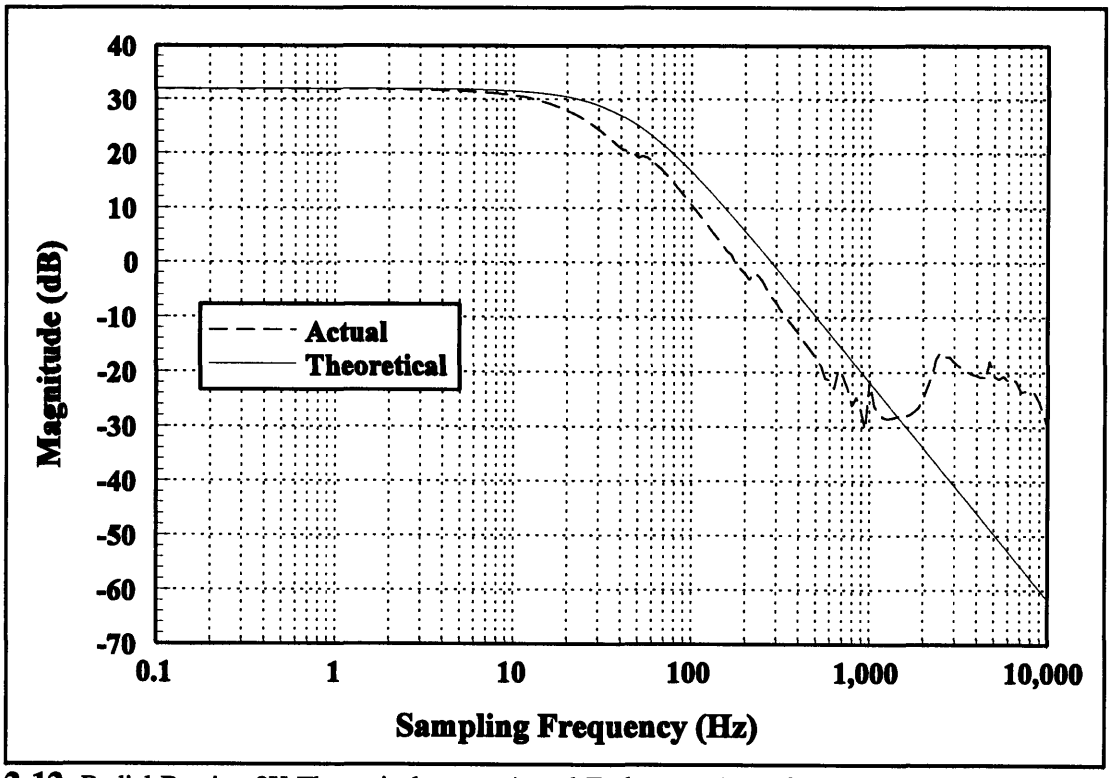
Figures 2.10, 2.11, 2.12, and 2.13 represent the theoretical analysis of the turbopump transfer function magnitude and phase plots for both the axial and 2X radial bearing.



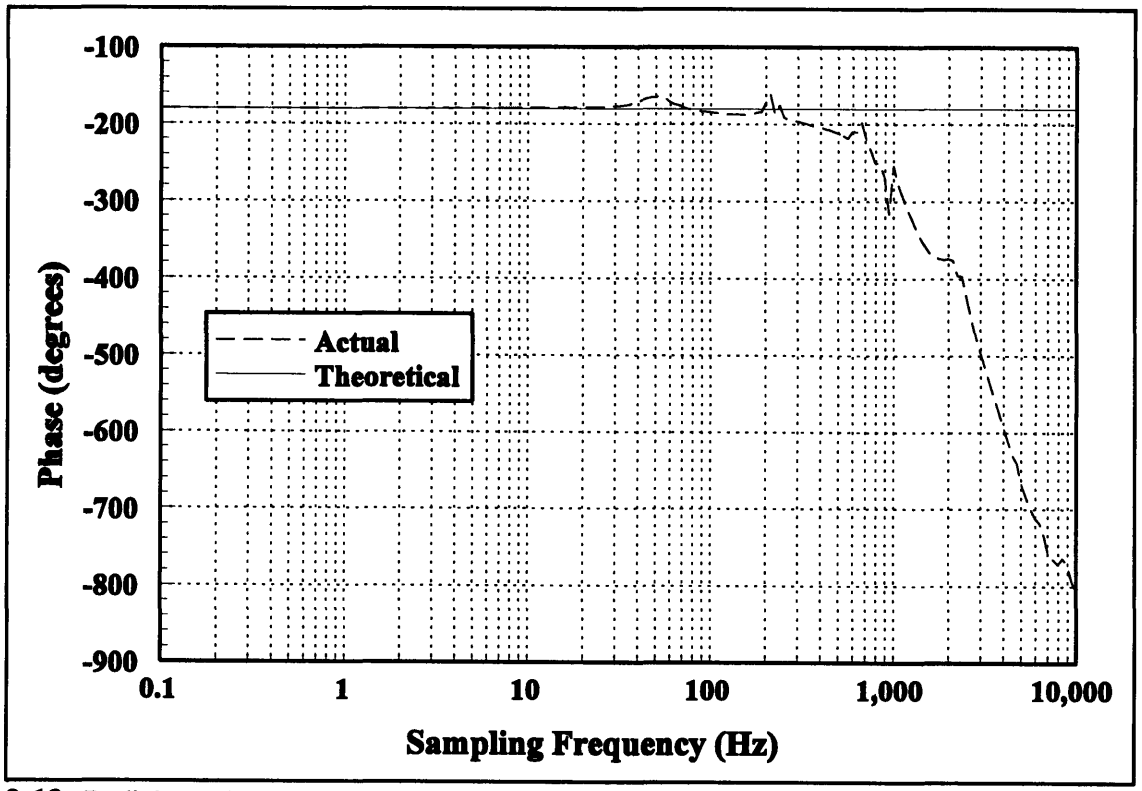

**2-10** Axial Bearing Theoretical versus Actual Turbopump Transfer Function Magnitude Plot




**2-11** Axial Bearing Theoretical versus Actual Turbopump Transfer Function Phase Plot




**2-12** Radial Bearing 2X Theoretical versus Actual Turbopump Transfer Function Magnitude Plot




**2-13** Radial Bearing 2X Theoretical versus Actual Turbopump Transfer Function Phase Plot

### 2.2.3 Best Fit Turbopump System Analysis

The previous section showed that the theoretical transfer function provides a fairly accurate picture of the turbopump frequency response. However, the theoretical response becomes less accurate as the frequency increases. It is important that the transfer function used to design the digital controller be as accurate as possible within the probable bandwidth of the controller. Therefore the theoretical analysis was not used in the controller design. Instead, a best fit recursive analysis was performed on the actual frequency response using a transfer function form derived from the theoretical analysis. The assumption was also made that the digital controller bandwidth would comfortably fall below 1000 Hz. Therefore, the best fit recursive analysis was optimized to produce a transfer function using the data points between 0.1 and 1000 Hz. However, this limits the validity of the model to 1000 Hz and therefore any controller designed using this model must provide adequate signal attenuation beyond this frequency.

As stated earlier, the actual turbopump response was obtained by inputting a swept sine disturbance at the test point between the driver and the power amplifier and monitoring the position signal at the appropriate test point. All tests were performed on the turbopump when the rotor was not spinning. The best fit transfer function was derived from the theoretical transfer function and therefore has the following form,

$$\frac{X(s)}{U(s)} = \frac{P}{s^2 - Q}$$

The values derived from the best fit recursive analysis for $P$ and $Q$ are,

| Parameter | Axial | Rad2X |
|---|---|---|
| P | 7.990 | 16.926 |
| Q | 22739.568 | 35530.574 |

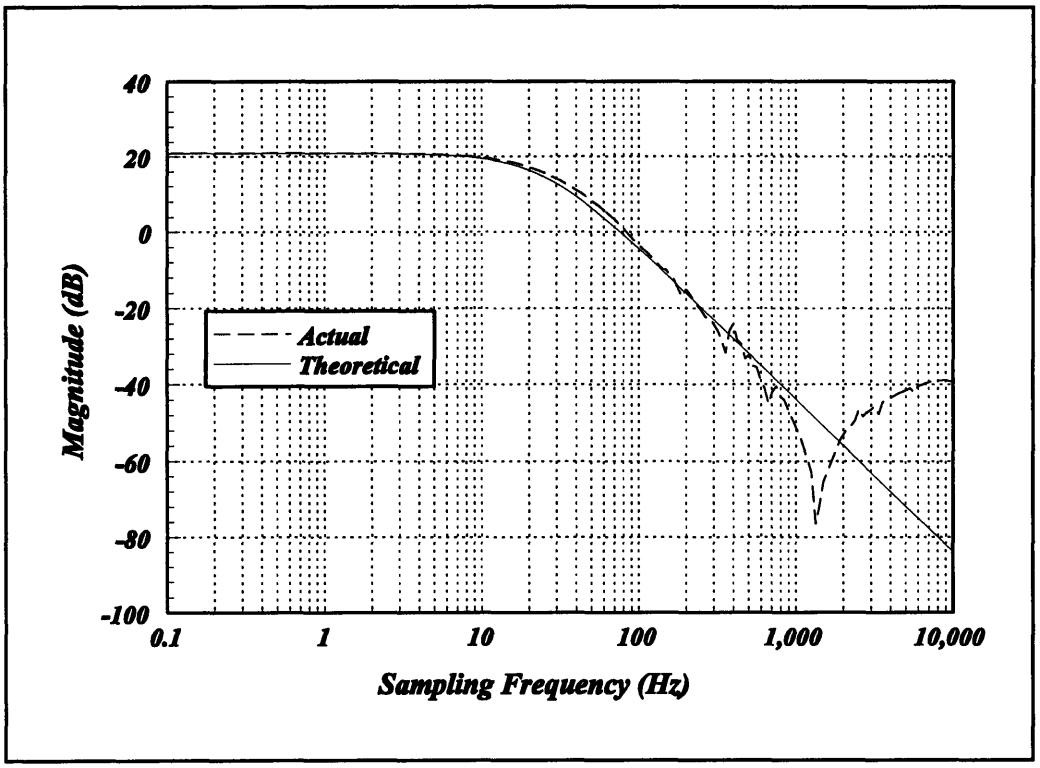
Again, in order to directly compare the response of the best fit transfer function to that of the actual system, the units of the transfer function must be comparable. Therefore the best fit transfer function uses the following conversion factors,

| Conversion Factor | Axial | Rad2X |
|---|---|---|
| Position Signal (V/m) | 9450.0 | 25000.0 |
| Driver Signal (V/A) | 0.3 | 0.3 |

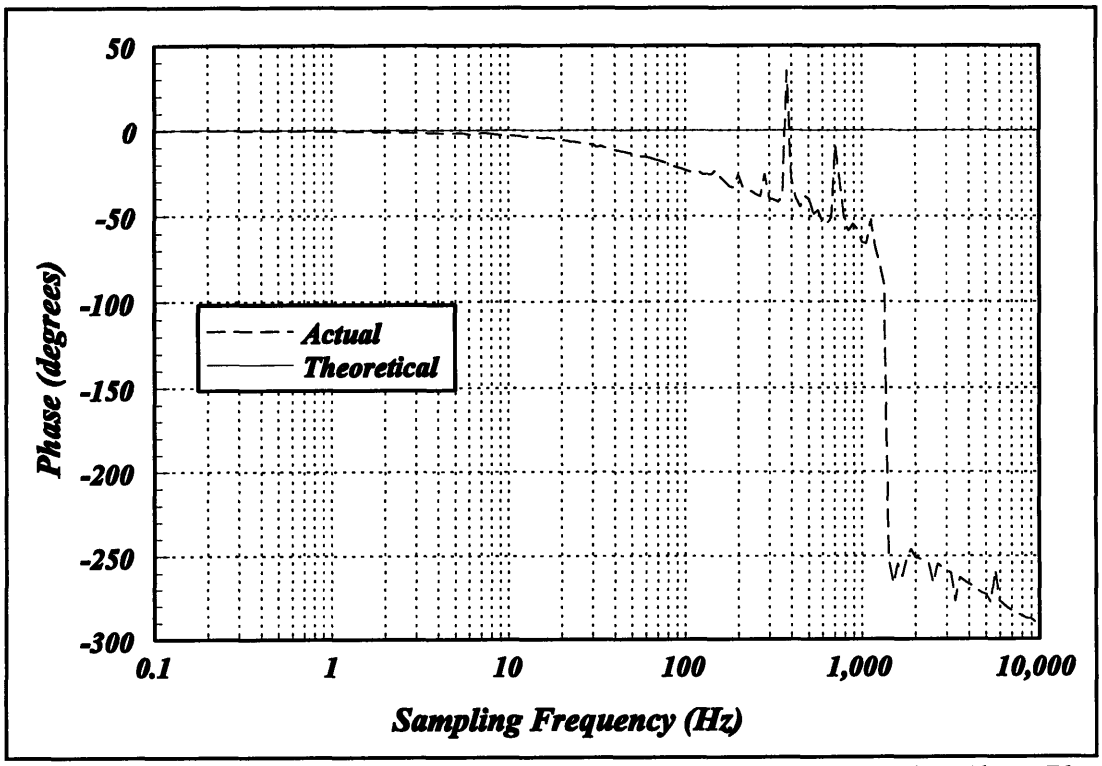
Finally, mention is again made of the phase plot for the axial bearing. The driver normally produces comparatively equal control currents for each half of the bearing. The major difference

is generally that one control current is negative and one is positive. The test point chosen for all of the best fit analyses was purposely chosen to be positive control current test point. In this way, the phase would not have to be adjusted. However, the axial bearing must support the weight of the rotor. Due to the particular implementation of the driver, the control signal required is large enough such that the lower bearing is unpowered. Hence there is only one possible test point which can be used for the axial bearing and its value happens to be negative. Therefore the phase is shifted appropriately.

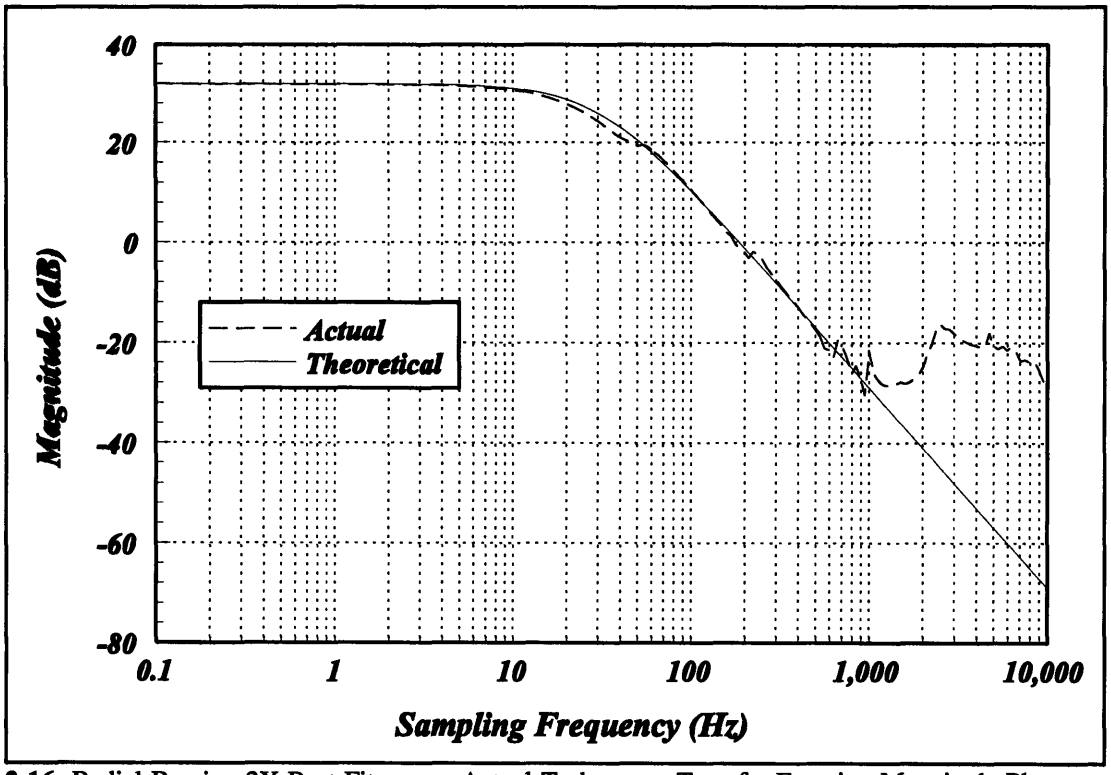
Figures 2.14, 2.15, 2.16, and 2.17 represent the best fit recursive analysis of the turbopump transfer function magnitude and phase plots for both the axial and 2X radial bearing.



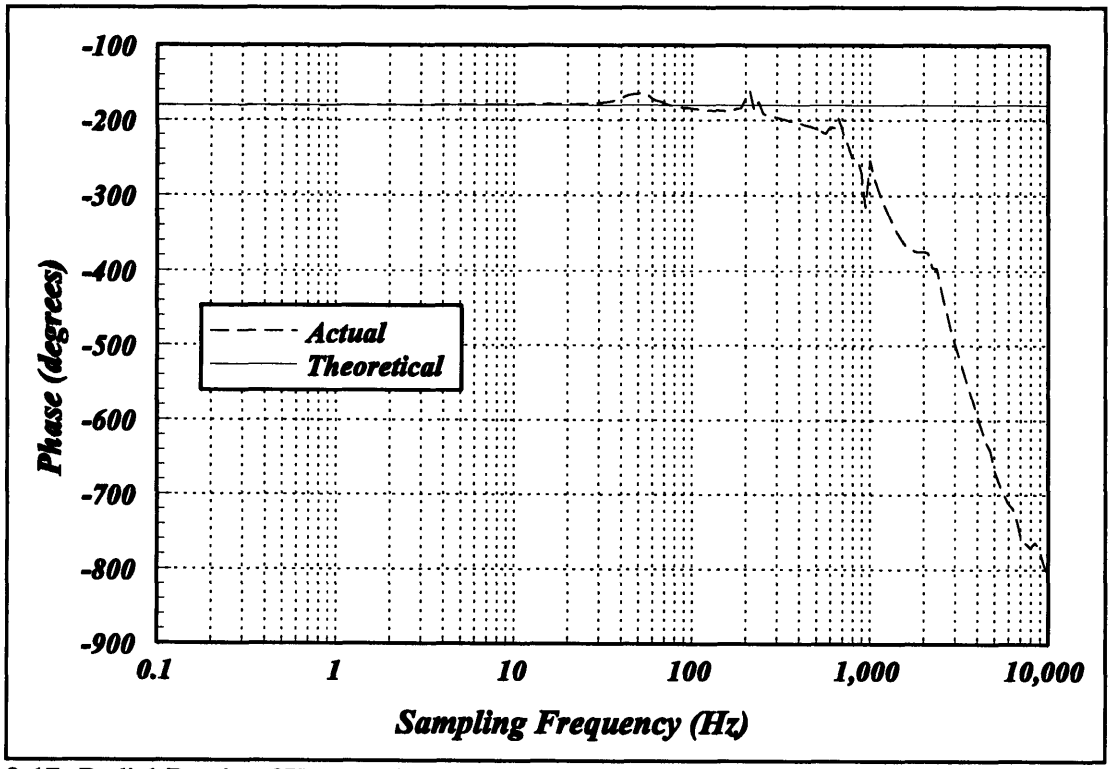

**2-14** Axial Bearing Best Fit versus Actual Turbopump Transfer Function Magnitude Plot




**2-15** Axial Bearing Best Fit versus Actual Turbopump Transfer Function Phase Plot




**2-16** Radial Bearing 2X Best Fit versus Actual Turbopump Transfer Function Magnitude Plot




**2-17** Radial Bearing 2X Best Fit versus Actual Turbopump Transfer Function Phase Plot

## 2.2.4 Best Fit Open Loop System Analysis

The best fit recursive analysis has been done for both the driver and turbopump subsystems. As a verification of this analysis, the combined transfer function of these two subsystems is compared to the actual system response. The actual driver response was obtained by inputting a swept sine disturbance at the analog/digital controller switch and monitoring the position signal at the appropriate test point. All tests were performed on the turbopump when the rotor was not spinning. The best fit open loop transfer function form is,

$$\frac{X(s)}{U(s)} = \left(\frac{A_1}{s+A_2}\right)\left(\frac{P}{s^2-Q}\right)$$

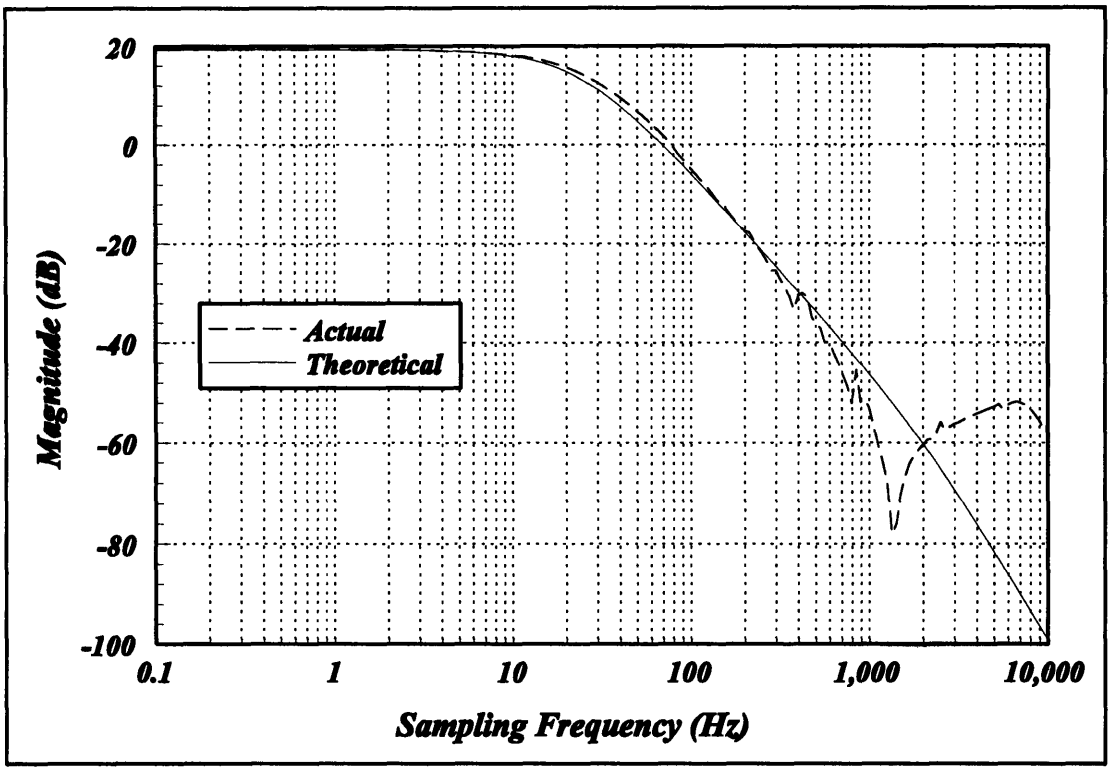
The values derived from the previous best fit analyses are,

| Parameter | Axial | Rad2X |
|---|---|---|
| $A_1$ | 14707.770 | 11112.759 |
| $A_2$ | 13310.000 | 11140.000 |
| P | 7.990 | 16.926 |
| Q | 22739.568 | 35530.574 |

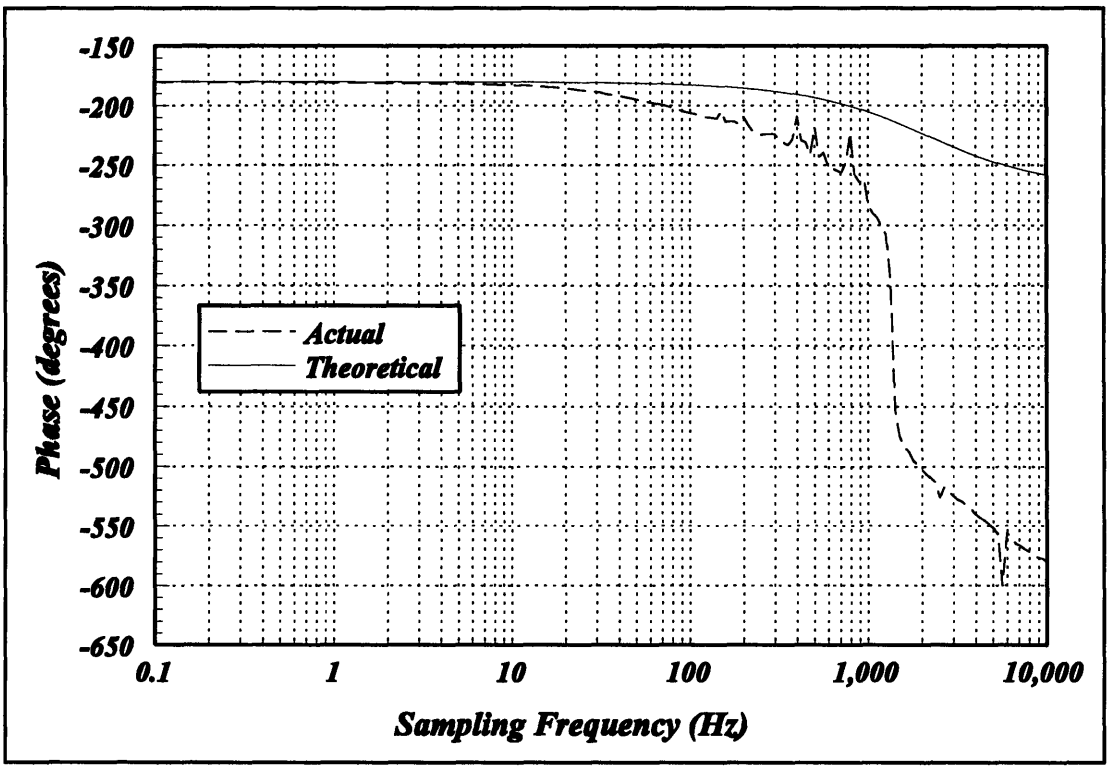
In order to directly compare the response of the best fit transfer function to that of the actual system, the units of the transfer function must be comparable. Therefore the best fit transfer function uses the following conversion factors,

| Conversion Factor | Axial | Rad2X |
|---|---|---|
| Control Signal (A/V) | 2.776 | 1.159 |
| Position Signal (V/m) | 9450.0 | 25000.0 |

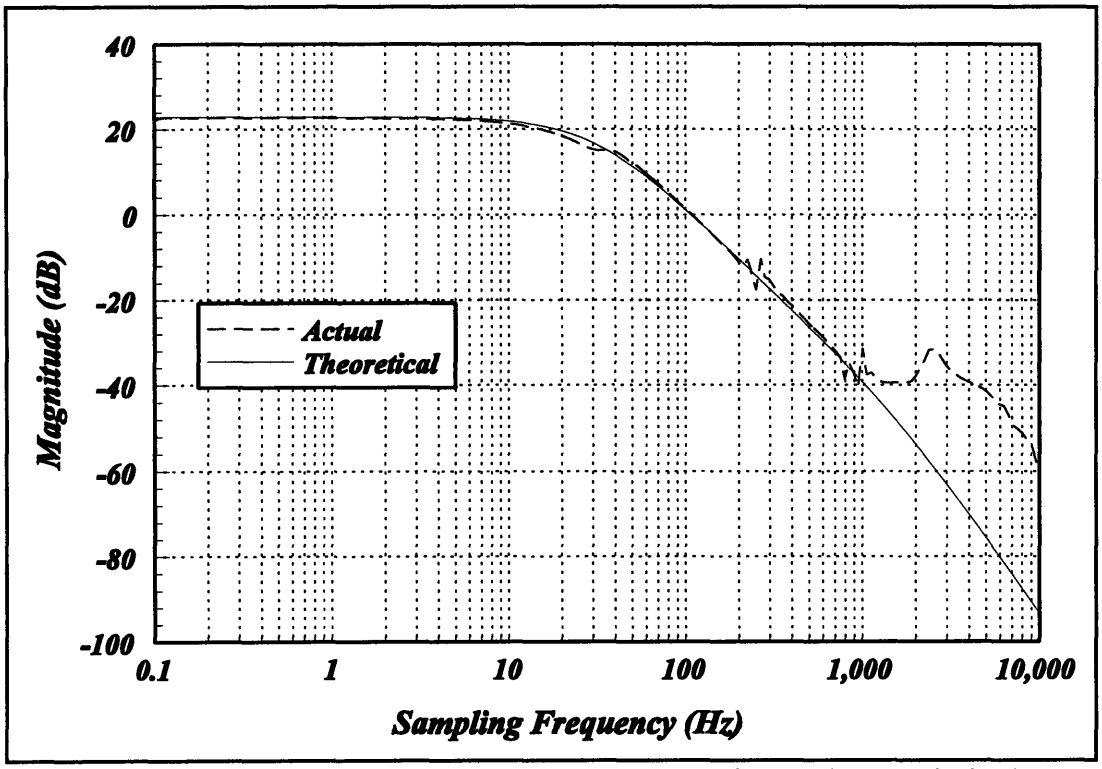
Figures 2.18, 2.19, 2.20, and 2.21 represent the best fit recursive analysis of the open loop transfer function magnitude and phase plots for both the axial and 2X radial bearing.



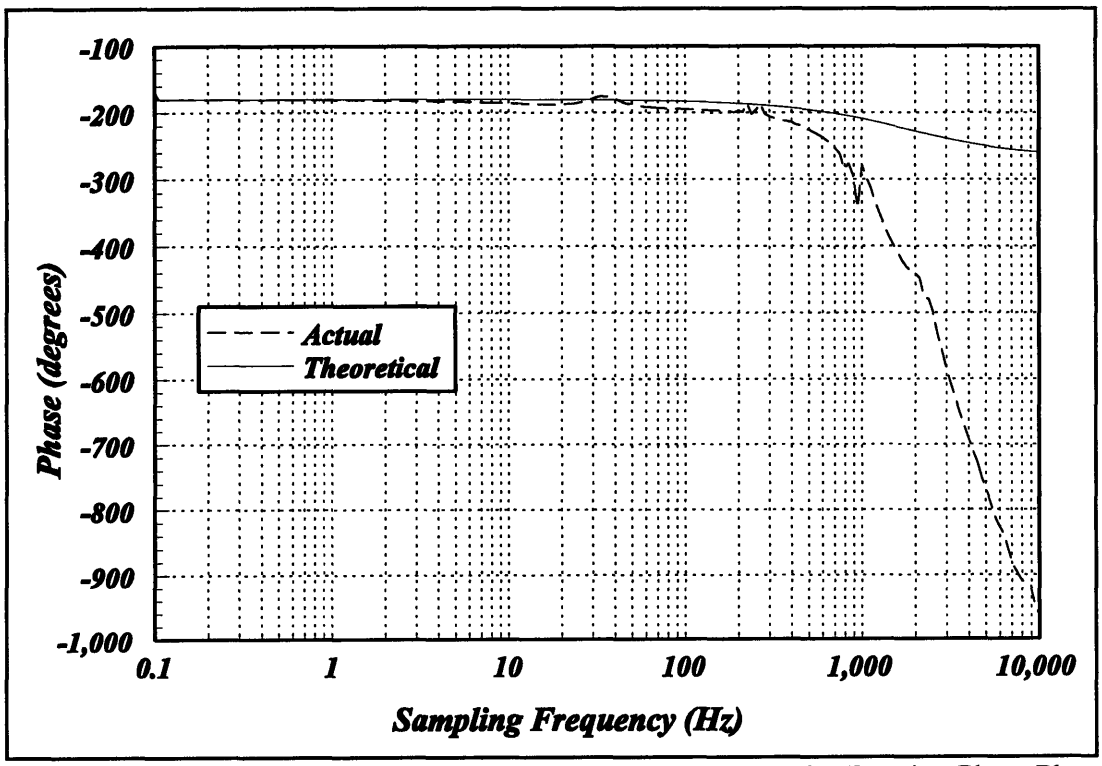

**2-18** Axial Bearing Best Fit versus Actual Open Loop Transfer Function Magnitude Plot




**2-19** Axial Bearing Best Fit versus Actual Open Loop Transfer Function Phase Plot




**2-20** Radial Bearing 2X Best Fit versus Actual Open Loop Transfer Function Magnitude Plot




**2-21** Radial Bearing 2X Best Fit versus Actual Open Loop Transfer Function Phase Plot

## 2.3 Summary and Remarks

This chapter described the process used to determine the open loop transfer function that would be employed in designing the digital controller. The turbopump transfer function was analyzed using a theoretical transfer function based upon the rotor equations of motion and using a best fit recursive analysis. The theoretical analysis showed fairly accurate agreement with the actual system response. However, the theoretical response grew progressively worst with increasing frequency. This conclusion becomes more apparent when all the bearing responses are reviewed as in Appendix E. Therefore, the results of the best fit recursive analysis will be used to design the digital controller. The best fit recursive analysis was also optimized to return the best possible correlation between the range of 0.1 and 1000 Hz. The upper bound was chosen to be 1000 Hz because this was deemed sufficiently greater then the bandwidth of the controller and because the actual system response became exceedingly less well behaved beyond that frequency for all of the bearing axes. Any theoretical analysis of the driver circuit was deemed too complicated and therefore only a best fit recursive analysis was performed on the driver subsystem. Finally, both the driver and turbopump subsystem transfer functions were combined and compared with the actual open loop system response. The good correlation between the plots offered verification that best fit recursive analysis had yielded a highly accurate transfer function below 1000 Hz. However, using the best fit recursive analysis effectively removes any correlation between the physics of the turbopump and the open loop transfer function.

# Chapter 3

# Time Delay Control Algorithm

---

This chapter is a review of the Time Delay Control law. This chapter also describes modifications made to the Control law in the actual controller to decrease processing time. First, the Time Delay Control law equations are derived based upon systems having an unknown distribution matrix. Finally, the specific simplifications that were made to the Time Delay Control law to increase its efficiency and the potential ramifications these simplifications have on the controller effectiveness are described. Most of these simplifications are applicable to the Time Delay Control law in general but a few are specific to the hardware on which it will be implemented.

## 3.1 Time Delay Control Law

The dynamic equations governing any system may be described by the following [22]:

$$\frac{d\mathbf{X}(t)}{dt} = \mathbf{F}(\mathbf{X},t) + \mathbf{H}(\mathbf{X},t) + \mathbf{B}(\mathbf{X},t)\,\mathbf{U}(t) + \mathbf{J}(\mathbf{X},\mathbf{U},t) + \mathbf{D}(t) \tag{3.1}$$

where: $\mathbf{X}(t)$ = (n×1) plant state vector,
$\mathbf{U}(t)$ = (r×1) control vector,
$\mathbf{F}(\mathbf{X},t)$ = (n×1) nonlinear vector representing known part of system dynamics,
$\mathbf{H}(\mathbf{X},t)$ = (n×1) nonlinear vector representing unknown part of system dynamics,
$\mathbf{B}(\mathbf{X},t)$ = (n×r) arbitrary estimate of the control distribution matrix,
$\mathbf{J}(\mathbf{X},\mathbf{U},t)$ = (n×1) distribution matrix error estimation vector,
$\mathbf{D}(t)$ = (n×1) unknown disturbance vector,
$t$ = time.

The distribution matrix error estimation vector $\mathbf{J}(\mathbf{X},\mathbf{U},t)$ represents the difference between the actual unknown distribution matrix and our estimate of that matrix or,

$$\mathbf{J}(\mathbf{X},\mathbf{U},t) = \mathbf{G}(\mathbf{X},\mathbf{U},t) - \mathbf{B}(\mathbf{X},t)\,\mathbf{U}(t) \tag{3.2}$$

where: $\mathbf{G(X,U,t)}$ = (n×1) actual unknown distribution matrix,
$\mathbf{B(X,t)}$ = (n×r) arbitrary estimate of the control distribution matrix,
$\mathbf{U(t)}$ = (r×1) control vector,

Let the desired dynamics reference model that we would ideally wish the actual plant to imitate be a linear time-invariant system given by:

$$\frac{d\mathbf{X}_m(t)}{dt} = \mathbf{A}_m\mathbf{X}_m(t) + \mathbf{B}_m\mathbf{R}(t) \tag{3.3}$$

where: $\mathbf{X}_m(t)$ = (n×1) model state vector,
$\mathbf{A}_m$ = (n×n) constant, stable matrix vector,
$\mathbf{B}_m$ = (n×r) constant command matrix,
$\mathbf{R}(t)$ = (r×1) command vector.

We now define the error vector as the difference between the actual plant state vectors and the desired dynamics reference model state vectors,

$$\mathbf{E}(t) = \mathbf{X}_m(t) - \mathbf{X}(t) \tag{3.4}$$

The objective is to force the error $\mathbf{E}(t)$ to vanish with a desired dynamics:

$$\frac{d\mathbf{E}(t)}{dt} = \mathbf{A}_e\mathbf{E}(t) \tag{3.5}$$

Combining Eq. (3.1), Eq. (3.3), Eq. (3.4), and Eq. (3.5) yields the following equation governing the error dynamics:

$$\begin{aligned}
\frac{d\mathbf{E}}{dt} &= \frac{d\mathbf{X}_m}{dt} - \frac{d\mathbf{X}}{dt} \\
&= \mathbf{A}_m\mathbf{X}_m + \mathbf{B}_m\mathbf{R} - \mathbf{F} - \mathbf{H} - \mathbf{J} - \mathbf{D} - \mathbf{BU} \\
&= \mathbf{A}_m(\mathbf{E} + \mathbf{X}) + \mathbf{B}_m\mathbf{R} - \mathbf{F} - \mathbf{H} - \mathbf{J} - \mathbf{D} - \mathbf{BU} \\
&= \mathbf{A}_m\mathbf{E} + \left[-\mathbf{F} - \mathbf{H} - \mathbf{J} - \mathbf{D} + \mathbf{A}_m\mathbf{X} + \mathbf{B}_m\mathbf{R} - \mathbf{BU}\right]
\end{aligned} \tag{3.6}$$

Now assume that there is a control $\mathbf{U}$ that satisfies the following requirement,

$$-\mathbf{F} - \mathbf{H} - \mathbf{J} - \mathbf{D} + \mathbf{A}_m\mathbf{X} + \mathbf{B}_m\mathbf{R} - \mathbf{BU} = \mathbf{KE} \tag{3.7}$$

where **K** is an (n×n) error feedback matrix. The substitution of Eq. (3.7) into Eq. (3.6) yields,

$$\frac{dE}{dt} = \left(A_m + K\right)E = A_e E \tag{3.8}$$

As the Eq. (3.8) indicates, any desired error system matrix $\mathbf{A_e}$ can be obtained from the proper choice of the error feedback gain matrix **K**. The control action **U** must be selected so that it satisfies the requirements of Eq. (3.6). The best approximation of this solution is given by,

$$U = B^{+}\left[-F - H - J - D + A_m X + B_m R - KE\right] \tag{3.9}$$

where: $\mathbf{B^+}$ is a pseudo-inverse matrix defined as $\left(\left(B^T B\right)^{-1} B^T\right)$

The constraint of Eq. (3.7) can always be met for systems that can be expressed in canonical form. However, such a constraint would severely limit the usefulness of TDC. Therefore, the only method of obtaining the desired error dynamics is by ensuring that the control action satisfies the above equation. Since the terms **H**(**X**,t), **J**(**X**,**U**,t), and **D**(t) are all unknown in the above equation, we must obtain an estimate for these terms. If the controller is sampling at a fast enough rate, these terms can be estimated from their values at the previous sampling instant. In other words,

$$H(X,t) + J(X,U,t) + D(t) \cong H(X,t-T) + J(X,U,t-T) + D(t-T) \tag{3.10}$$

where: T = small time delay.

Rewriting Eq. (3.1) for the previous sampling interval,

$$\dot{X}(t-T) = F(X,t-T) + H(X,t-T) + B(X,t-T)U(t-T) + J(X,U,t-T) + D(t-T) \tag{3.11}$$

Rearranging,

$$H(X,t-T) + J(X,U,t-T) + D(t-T) = \dot{X}(t-T) - F(X,t-T) - B(X,t-T)U(t-T) \tag{3.12}$$

Substituting Eq. (3.12) into Eq. (3.10),

$$H(X,t) + J(X,U,t) + D(t) \cong \dot{X}(t-T) - F(X,t-T) - B(X,t-T)U(t-T) \tag{3.13}$$

The time delay control law is obtained by substituting Eq. (3.13) into Eq. (3.9) and is given by,

$$U(t) = B^{+}(X,t)\left[-F(X,t) - \dot{X}(t-T) + F(X,t-T) + B(X,t-T)U(t-T) + A_m X(t) + B_m R(t) - KE(t)\right] \tag{3.14}$$

The time delay control law can be easily understood when it is divided into its four constituent parts:

1. A pseudo-inverse matrix, $\mathbf{B}^{+}(\mathbf{X},t)$, which nullifies the control matrix $\mathbf{B}(\mathbf{X},t)$ found in the constraining equation $-\mathbf{F}-\mathbf{H}-\mathbf{D}+\mathbf{A}_m\mathbf{X}+\mathbf{B}_m\mathbf{R}-\mathbf{B}\mathbf{U} = \mathbf{K}\mathbf{E}$
2. A undesired dynamics term, $-F(X,t)-\dot{X}(t-T)+F(X,t-T)+B(X,t-T)U(t-T)$, which attempts to cancel the undesired known dynamics $\mathbf{F}(\mathbf{X},t)$, the unknown nonlinear dynamics $\mathbf{H}(\mathbf{X},t)$, the error in estimating the distribution matrix $\mathbf{J}(\mathbf{X},\mathbf{U},t)$, and the unexpected disturbance $\mathbf{D}(t)$. These last three terms are estimated from the previous sampling interval and are represented by $\dot{X}(t-T)-F(X,t-T)-B(X,t-T)U(t-T)$.
3. A desired dynamics reference model term, $\mathbf{A}_m\mathbf{X}(t)+\mathbf{B}_m\mathbf{R}(t)$, which inserts the ideal system response.
4. An error feedback term, $\mathbf{K}\mathbf{E}(t)$, which adjusts the error dynamics to compensate for the difference between the ideal error response and the actual reference model's response.

In essence, the time delay control algorithm first computes the ideal control action based on the current state of the system and the desired dynamics reference model. It then estimates the current unknown system dynamics using the state of the system at the previous sampling interval(s), the final control action at the previous sampling interval(s), and the current error. Using this estimate, the ideal control action is augmented to compensate for the unknown system dynamics to produce the final control action. Because the time delay control algorithm uses information from the previous sampling interval(s) to determine the unknown system dynamics, it is critical for stability that the unknown system dynamics not change significantly between sampling intervals. Therefore, a sufficiently small sampling interval is a necessary condition for stability. Also, in the above derivation, the control distribution matrix $\mathbf{B}$ is assumed to be known and linear.

## 3.2 Implementation Simplifications

As previously stated, a necessary condition for stability when using time delay control is that the sampling interval must be sufficiently small. Sufficiently small however is a rather vague term and differs from system to system. One of the variables that determine the maximum theoretical sampling rate of a controller is control algorithm efficiency. In order to make the algorithm as efficient as possible, certain simplifications were made to the algorithm to decrease the computation time required to produce the final control action. This section will examine the simplifications made to the time delay control algorithm and their implications on system performance. The form of the time control algorithm used in this section is a slightly different version of the one described by Eq. (3.14),

$$\begin{aligned} U(t) = B^{+}(X,t)\big[ &-F(X,t)-\dot{X}(t-T)+F(X,t-T)+B(X,t-T)U(t-T) \\ &+A_m X(t)+B_m R(t)-(A_e-A_m)E(t)\big] \end{aligned} \tag{3.15}$$

### 3.2.1 Constant Control Distribution Matrix, B(X,t)

If the control distribution matrix is assumed to be time and position invariant, then the time delay control algorithm becomes,

$$U(t) = U(t-T) + \frac{1}{B}\left[ -F(X,t) - \dot{X}(t-T) + F(X,t-T) + A_m X(t) + B_m R(t) - (A_e - A_m)E(t) \right] \qquad (3.16)$$

This simplification eliminates the recalculation of both the control matrix and the pseudo inverse matrix at each sampling interval. This is particularly important for multi-input systems but may be of a lesser concern for single-input systems. This simplification also eliminates all division operations from the algorithm because the reciprocal of the distribution matrix can be computed previously and stored for use during input signal processing. This is particularly important for efficient signal processing because most (if not all) digital signal processors require significantly more cycles to divide than to multiply. For instance, the processor used by this controller requires one cycle to multiply two floating point numbers and seven cycles to divide two such numbers. The disadvantage of this simplification is that the control is less refined. The term $\mathbf{B}^{+}(\mathbf{X},t)$ is essentially the feedback gain of the controller. In this turbopump application, a gain calculated at each sampling interval would allow the controller to apply higher gains depending upon the rotor distance from the nominal center of the bearings. This could lead to better system response and disturbance rejection properties. It would also lead to a greater lag between position signal acquisition and control action application due to longer calculation times. Also the distribution matrix is a function of time which could allow for smaller feedback gains during the startup phase of the turbopump which would give the magnetic bearings and position sensors time to settle before control was applied.

### 3.2.2 Error Dynamics Matrix Equals Reference Model Matrix ($\mathbf{A_e = A_m}$)

If the error dynamics matrix ($\mathbf{A_e}$) is assumed to be equal to the desired dynamics reference model matrix ($\mathbf{A_m}$), then the time delay control algorithm becomes,

$$U(t) = U(t-T) + \frac{1}{B}\left[ -F(X,t) - \dot{X}(t-T) + F(X,t-T) + A_m X(t) + B_m R(t) \right]$$

or

$$U(t) = U(t-T) + \frac{1}{B}\left[ -F(X,t) - \dot{X}(t-T) + F(X,t-T) + A_e X(t) + B_m R(t) \right] \qquad (3.17)$$

This simplification can be rewritten in an equivalent manner as $\mathbf{K = 0}$. The advantage of this simplification is the elimination of a multiplication step. For multi-input systems this would become a matrix multiplication elimination. The physical interpretation of this simplification requires some clarification. The Time Delay Control algorithm contains a desired dynamics

reference model and an error dynamics model. The desired dynamics reference model represents the ideal trajectory that we would like the system to follow. This is the command following portion of the algorithm. The error dynamics model represents the response we would like the system to take in response to any error between the actual state variables and the desired dynamics reference model state variables. This is the disturbance rejection portion of the algorithm. Setting the desired dynamics reference model matrix equal to the error dynamics matrix effectively eliminates the contribution that the command following portion of the algorithm has upon the system response. We are not interested in the trajectory of the system but merely in the difference between the actual state variables and the reference variables. The ramification of ignoring the command following portion of the algorithm for this particular system is that the response during the startup phase will be less then optimal particularly for the axial magnetic bearing. At startup, the rotor is generally the farthest it will be from the equilibrium position. The use of the command following portion of the algorithm would provide optimal response during this phase. Without it, the system is subject to large overshoots and longer settling times. However the startup phase is such a brief portion of the time that the controller spends controlling the turbopump that the added algorithm efficiency derived from such a simplification is justified.

### 3.2.3 Final System Specific Simplifications

The final simplifications to the Time Delay Control Algorithm are related specifically to the system we are working on and therefore may not be generally applicable. The first simplification is specifying that all the dynamics of the system are unknown or $\mathbf{F}(\mathbf{X},t) = \mathbf{0}$. Therefore,

$$U(t) = U(t-T) + \frac{1}{B}\left[-\dot{X}(t-T) + A_e X(t) + B_m R(t)\right] \tag{3.18}$$

This places a greater reliance on the ability of the Time Delay Control algorithm to correctly compensate for the unmodeled dynamics. This in turn places greater emphasis on the proper choice of the controller sampling rate. This simplification increases the efficiency of the algorithm but it also most likely requires a higher sampling rate which may actually negate the efficiency gains.

The final simplification pertains to the reference signal. For this particular system that reference signal is zero and therefore,

$$U(t) = U(t-T) - \frac{1}{B}\left[\dot{X}(t-T) - A_e X(t)\right] \tag{3.19}$$

The above equation represents the limit to how efficient the Time Delay Control algorithm can become.

## 3.3 Summary and Remarks

This chapter introduced the Time Delay Control Algorithm and demonstrated how it uses information from the previous sampling interval(s) to predict the uncertain system dynamics. This characteristic is the compelling reason for using Time Delay Control with this magnetic bearing turbopump application. The attractive magnet force applied by each magnetic bearing is proportional to the square of the coil current and inversely proportional to the square to the gap between the rotor and bearing. Therefore, the equations of motion of the rotor are inherently nonlinear. Also the system is subject to disturbances characteristic of rotating machinery such as rotor gyroscopic effects, rotor imbalance, and rotor flexibility. It is impractical to design a controller using a model which incorporates all these properties and therefore the model must be simplified. For this application, this simplification entails linearizing the equations of motion and ignoring most of the rotor dynamic effects. It is hoped that the inherent strengths of Time Delay Control with compensate for the deficiencies in a controller designed using such a simplified model.

Time Delay Control is not without its own problems. It is a necessary condition of Time Delay Control that the sampling interval be sufficiently small. There is currently, however, no method for quantifying sufficiently small nor for determining the optimum sampling interval. Therefore this researcher simplified the control law if an effort to make it more efficient. The increased efficiency of the algorithm would allow for faster sampling rates thereby increasing the range of possible sampling intervals from which to determine the optimum sampling interval. These simplifications are also necessary because, due to cost considerations, all five axes will be controlled using only one processor. This requires that the control algorithm must be processed five times per sampling interval and therefore algorithm efficiency is of paramount importance. These simplifications however come at a price. Most of these simplifications primarily effect the startup phase of the control process and will probably lead to less than ideal initial time responses. However, the exclusion of all known system dynamics from the control algorithm places more importance on determining the optimal sampling interval.

# Chapter 4

# Controller Description and Design

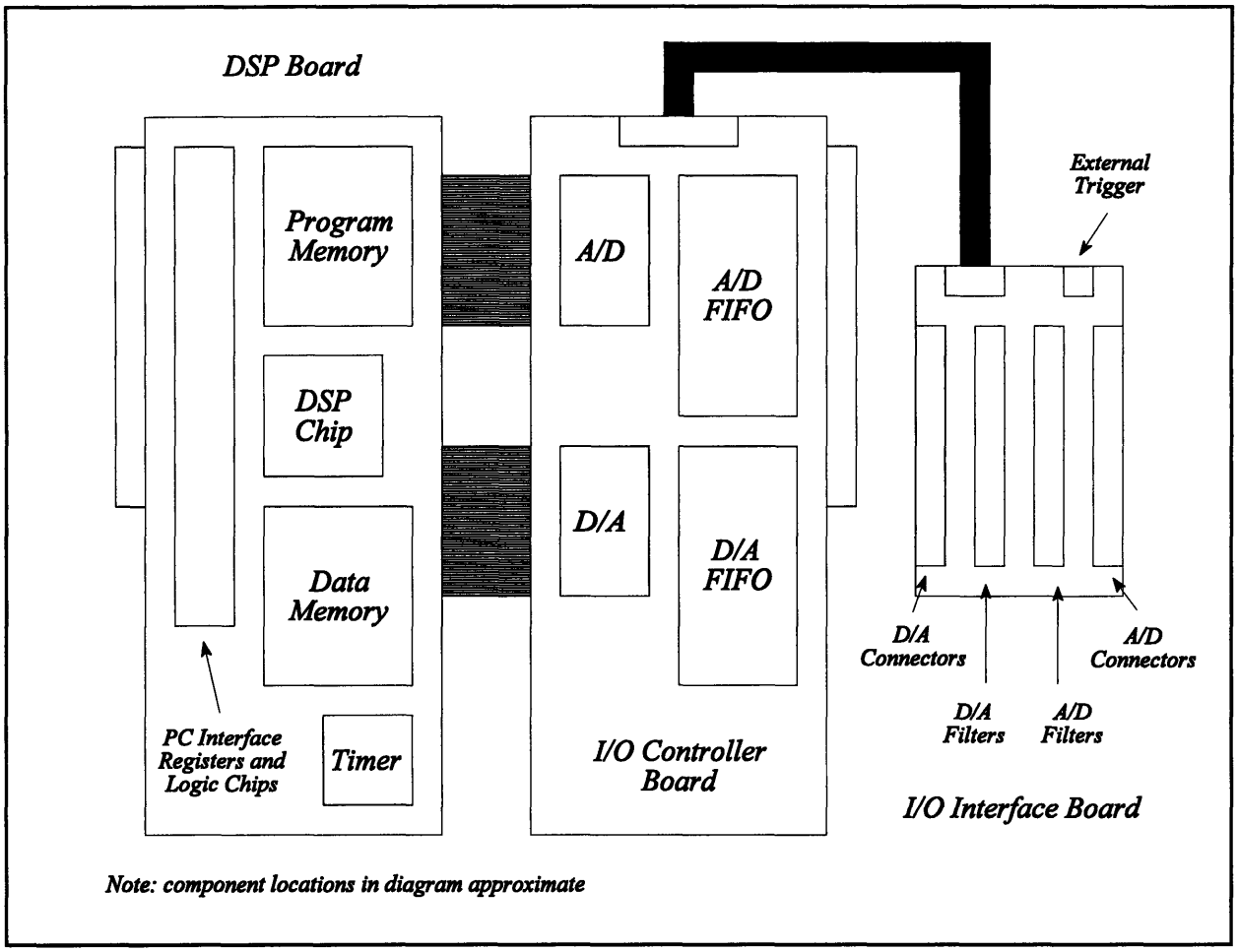
This chapter describes the hardware that constitutes the controller and the design process used to arrive at the final controller design. A detailed account of the process to determine the sampling rate and the other controller parameters required by the Time Delay Control (TDC) algorithm is also provided. Next the actual structure of the program is examined in an effort to explain how system dynamics and hardware limitations actually effect the controller implementation. Then the system characteristics that were not accounted for in the theoretical model but which must be addressed during the implementation process are examined. Finally, the design methods used to produce the filters necessary for proper operation are provided and the manual tuning process is outlined.

## 4.1 Physical Description

The digital controller hardware has the following three major components: 1) the Digital Signal Processor (DSP) board, 2) the Input/Output (I/O) Controller board, and 3) the I/O Interface board. Both the DSP board and the I/O Controller board occupy slots on a Personal Computer (PC) bus and draw power through that bus. Only the DSP board is addressable using ports allocated from the PC's free port address space. The address of the port is set through jumpers on the DSP board. The DSP board and the I/O Controller board are physically connected by two ribbon cables located at the top edge of each board. The I/O Controller board and the I/O Interface board communicate via a ribbon cable which attaches to external connectors located on each board. The I/O Interface board is a circuit board having a leg at each corner to allow it to be free standing. This is the only board situated outside the PC. Figure 4-1 displays each board and the components that comprise their circuitry.

### 4.1.1 DSP Board

The primary components of the DSP board are the DSP chip, the memory, the timer chip, and the PC interface logic chips. The DSP chip is an Analog Devices ADSP-21020. The chip natively supports integer and floating point operations, has sixteen 48-bit general purpose registers, an accumulator, a multiplier, and a barrel shifter. The chip supports interrupts and the register set can be split to support nested interrupts. The chip can address up to 16 Megawords




**4-1** Controller Hardware

of 48-bit program memory and 4 Gigawords of 48-bit data memory on separate buses. The data memory can be partitioned into program and data subspaces so that programs can execute code residing in data memory. The DSP chip has a clock speed of 33 MHZ and each instruction executes in one clock tick. The chip is capable of executing two instructions per clock tick provided that one instruction uses only the general registers and the accumulator, multiplier, or shifter and the other instruction uses the general registers and a program or data memory operation.

The board is populated with 32 Kilowords of 48-bit program and data memory. The first 256 words of program memory are reserved for the interrupt vector table. The program memory address space at 800000h is mapped to two A/D registers on the I/O Controller board which allows values to be passed between the two boards. The data memory address space at 20000000h is mapped to other registers on the DSP board. These registers allow communication between other components on the board such as the timer chip and the PC Interface logic chips. The data memory address space at 40000000h is also mapped to three registers on the I/O Controller board. Two of these three registers maps the D/A registers and the other one controls

the number of A/Ds and D/As that are to be used.

The timer chip has a 33 MHZ clock speed and is addressed through two registers. One register holds the current count and the other is the reload register. At each clock tick, the current count register is decremented. When the register reaches zero, an interrupt is generated and the value in the reload register is placed in the current counter register. The value in the reload register is the number of clock ticks between interrupts and therefore corresponds directly to the required sampling rate.

The PC Interface logic chips are responsible for handling all communication between the DSP and the PC. These chips contain registers that allow the uploading or downloading of data to and from program and data memory. They also provide a status register that can be used to quickly send one word of data to the PC. The PC Interface logic chips are incapable of generating interrupts on the PC bus and therefore polling the status register is the only way of obtaining real time information from the DSP board.

### 4.1.2 I/O Controller Board

The I/O Controller board has the following four components: 1) the Analog-to-Digital converter (A/D), 2) the A/D First In, First Out queue (FIFO), 3) the Digital-to-Analog converter (D/A), and 4) the D/A FIFO. Both FIFOs map a register into the DSP board address space so that the DSP chip can transmit values to the D/A FIFO and received values from the A/D FIFO. The number values sent to or received from the FIFOs must be equal to the value specified in the channel register. This register is also mapped to the DSP address space and allows for up to seven A/Ds and D/As to be chosen. If the values sent or received is not equal to the channel register value, garbage values will be returned or the FIFOs will fill up depending upon circumstances. Two other registers are also mapped into the DSP address space and these are the A/D and D/A status registers. They show when a conversion is taking place and whether the FIFO is full or empty.

The A/D is a 14-bit design having a range of ±5V. The A/D data is a left-shifted twos complement binary number. The D/A is a 16-bit design also having a range of ±5V. The D/A data is essentially a twos-complement value with the sign bit negated. Normally A/Ds and D/As produced by the same manufacturer and populating the same board have the same bitness. It was probably cost constraints that prevented the A/Ds from also being 16-bit. Also, A/Ds and D/As produced by the same manufacturer and populating the same board usually have the same method of representing data values for reasons of consistency. However, on this board, the D/A data values have a somewhat awkward representation.

Finally, the relationship between the number in the channel register and what A/Ds or D/As are addressed must be elaborated on. If the programmer specifies one A/D for instance in the channel register, the first A/D is accessed. If two A/Ds are specified, the first two A/Ds are accessed. If three A/Ds are specified, the first three A/Ds are accessed and so on. The ramifications of this design decision is that the A/Ds are not individually addressable. Therefore, if the programmer only wants the value from the fifth A/D, five A/Ds must be specified in the channel register and the program must wait for five conversion processes to take place. The program then must retrieve four values from the A/D FIFO and throw them away in order to get

to the fifth value. This same process also applies to the D/A except five valid values must be loaded into the FIFO before conversion. Also there is no way to specify that only one conversion process should take place -- either A/D or D/A. A program can only specify that a full conversion process should take place. When a conversion process is triggered by a program. First the number of D/A data values specified in the channel register are converted and then the number of A/D data values specified in the channel register are converted. Depending upon the implementation, this could lead to wasted time doing unnecessary conversions.

### 4.1.3 I/O Interface Board

The I/O Interface board consists of seven A/D connectors, five D/A connectors, and their associated filters. The connectors mimic those found on most oscilloscopes and test equipment. The filters are second order Butterworth designs implemented using an RC circuit. The cutoff frequency can be altered by replacing the resistor packs on the board. However, changing the resistor packs also changes the DC offset of the associated A/Ds and D/As. Generally, the DC offset increases as the cutoff frequency decreases.

## 4.2 Controller Design

A controller is only as good as the theoretical model it was designed to control. The degree to which the theoretical model mimics the actual system determines how well the controller performs initially and how much manual tuning will be necessary to obtain satisfactory performance. The process of designing a robust controller requires that the engineer have some feel for what performance measures are important. The importance of a particular performance measure is in part dictated by how the application is to be used. For instance, a robot arm would require a strong emphasis on command following whereas an aircraft autopilot would require a strong emphasis on disturbance rejection.

In this turbopump, as with all turbomachinery applications, disturbance rejection is the critical performance measure. Command following does play a minor role during the startup phase when the magnetic bearings are just being powered up and the rotor is at an unknown position. The startup phase however lasts on the order of seconds while the operating phase lasts on the order of hours or even days. With turbomachinery, the bearings are subject to disturbances brought about by gyroscopic forces, rotor imbalance, and rotor flexibility. This turbopump also operates at high rotational speeds thus magnifying these effects.

Disturbance rejection takes on a greater importance when the simplifications to the system model are reviewed. The original linearized equations of motion of the rotor have the following form,

Radial Bearing 1X

$$\ddot{x}_1 = \frac{4C_1kI_0^2}{h_0^3}x_1 + \frac{4C_3kI_0^2}{h_0^3}x_2 - C_4\dot{y}_1 + C_4\dot{y}_2 + \frac{2C_1kI_0}{h_0^2}u_{x_1} + \frac{2C_3kI_0}{h_0^2}u_{x_2}$$

Axial Bearing

$$\ddot{z} = \frac{4kI_0^2}{mh_0^3}z + \frac{2kI_0}{mh_0^2}u_z$$

The axial bearing equation of motion lends itself well to classical design techniques but further simplifications must be performed on the radial bearing equation of motion. The first simplification is that the controller will be designed for the case when the rotor is not spinning. This simplification means that all gyroscopic effects will be ignored in the design process. Therefore manual tuning on the actual system will almost assuredly be required. This manual tuning however can be minimized if the gyroscopic effects are treated as a disturbance thereby placing further emphasis on disturbance rejection performance. With this simplification, the radial bearing equation of motion becomes,

$$\ddot{x}_1 = \frac{4C_1kI_0^2}{h_0^3}x_1 + \frac{4C_3kI_0^2}{h_0^3}x_2 + \frac{2C_1kI_0}{h_0^2}u_{x_1} + \frac{2C_3kI_0}{h_0^2}u_{x_2}$$

The final assumption is that the contribution to the radial bearing equations of motion attributable to the coupling between the radial bearings can be treated as a disturbance. The radial bearing equation of motion then takes on the form of the axial bearing equation of motion,

$$\ddot{x}_1 = \frac{4C_1kI_0^2}{h_0^3}x_1 + \frac{2C_1kI_0}{h_0^2}u_{x_1}$$

Now the radial bearing equation of motion is also in a form that makes it possible to utilize classical design techniques. However the net effect of all these assumptions is an increased reliance on good disturbance rejection and manual tuning.

These assumptions were carried out in the Chapter 2 during the system identification phase of the design process. During that phase, it became apparent that the most accurate transfer function was obtained by using a best fit recursive technique. However, this technique was optimized to produce the most accurate fit up to 1000 Hz. The region above 1000 Hz is dominated by high frequency dynamics. Including this region in the best fit recursive analysis tends to create a derived transfer function which is less accurate within the 100 to 1000 Hz region. The 100 to 1000 Hz region lies within the probable bandwidth of our controller while the region above 1000 Hz does not. Therefore, the best fit recursive analysis ignores high frequency region and any controller designed using this transfer function must provide adequate attenuation of the control signal after 1000 Hz to prevent excitation of unmodeled system dynamics at higher frequencies.

Now the choice of Time Delay Control (TDC) as the control algorithm becomes apparent. TDC utilizes information from the previous sampling interval(s) as well as the current error signal to determine the unmodeled dynamics at the previous interval. Provided that the sampling interval is sufficiently small, the unmodeled dynamics will not change significantly during this

interval. Therefore the control algorithm has an estimate of the present unmodeled dynamics and augments the control signal to compensate. It is hoped that this augmentation will be sufficient to compensate for the coupling effects, gyroscopic effects, and any other disturbances the system may encounter.

This may seem like too much to ask considering the number and the nature of the simplifications made but previous research suggests otherwise. TDC has been applied separately to both the axial bearing and a radial bearing with encouraging results. However, what is missing in all previous work is a rigorous systematic procedure for designing a controller using TDC. This thesis will not attempt to define a definitive design procedure but merely to document one attempt at rigorous design so that others learn and perhaps carry the process further.

### 4.2.1 Sampling Rate Determination

A necessary condition for stability of the Time Delay Control (TDC) algorithm is that the sampling interval be sufficiently small such that the unmodeled system dynamics do not change appreciably between sampling intervals. To date, previous research has not addressed how to determine the appropriate sampling rate. Generally, the sampling rate has been determined by engineering experience or hardware limitations. These methods are fine for research purposes but quite unsatisfactory given real world economic constraints. Sampling at a higher than required rate may not adversely effect performance but it can lead to the use of higher priced hardware then otherwise would be required.

The use of TDC as the controller algorithm also places additional emphasis on proper sampling rate determination. The simplified TDC law derived in Chapter 3 was defined as,

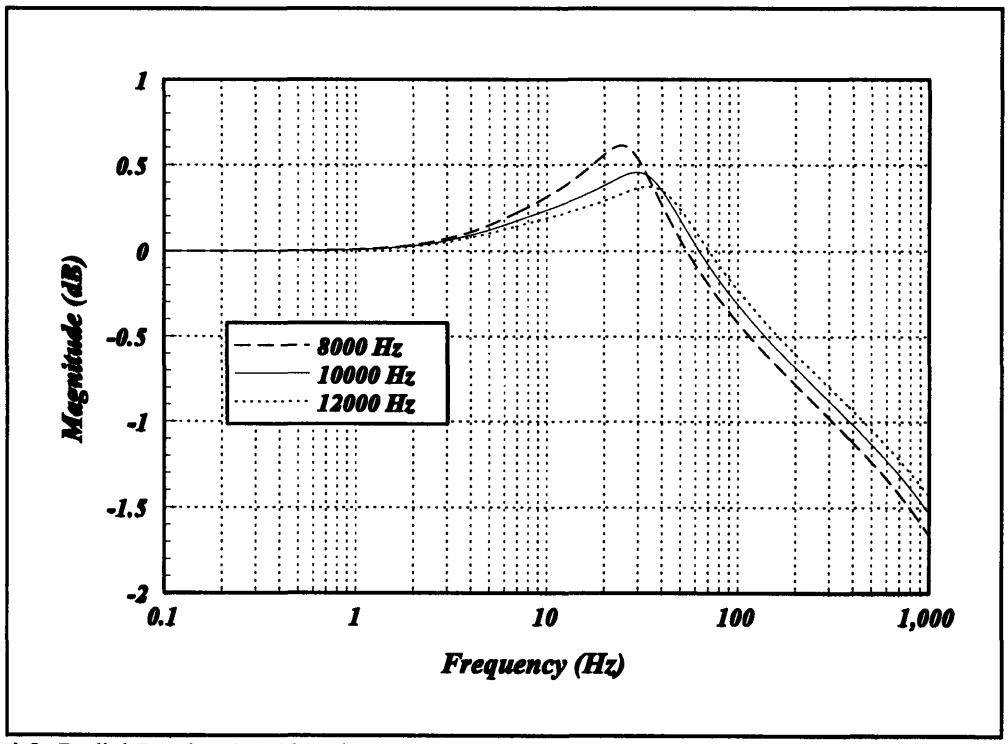
$$U(t) = U(t-T) - \frac{1}{B}\left[\dot{X}(t-T) - A_e X(t)\right]$$

Taking the Laplace transform and computing the transfer function yields,

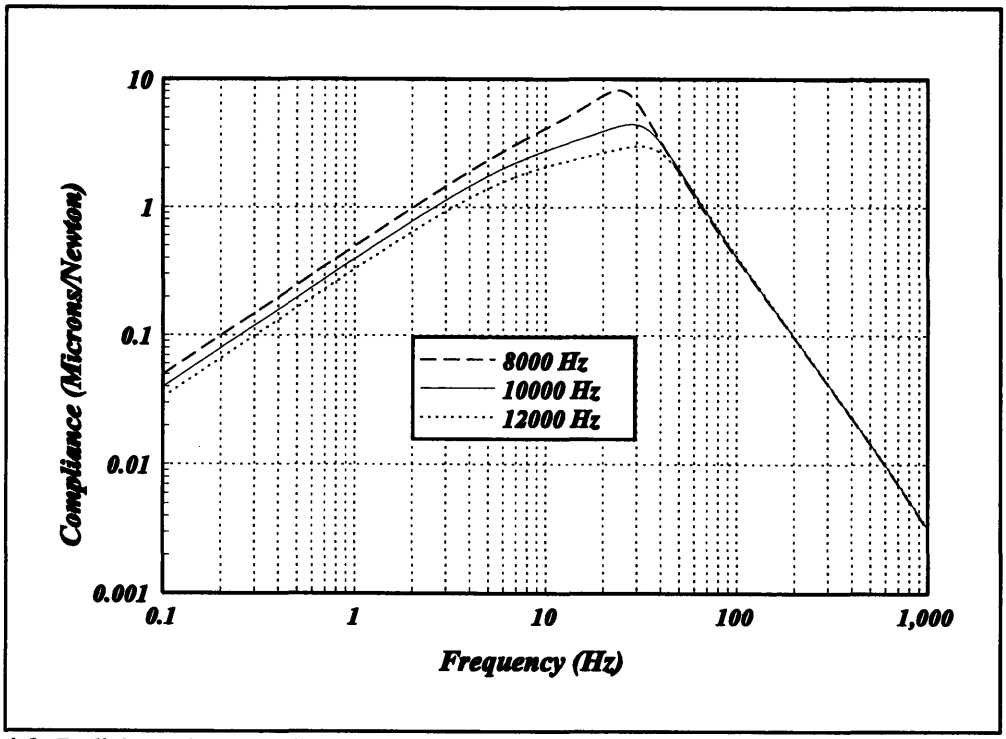
$$\frac{U(s)}{X(s)} = -\frac{e^{-Ts}s - A_e}{B(1-e^{-Ts})}$$

This transfer function demonstrates that TDC can be thought of as a nonlinear form of PID control. It also demonstrates that the sampling rate effects both the derivative and integral portions of this PID controller. When a continuous time control algorithm is implemented digitally, the sampling rate normally just effects the stability of the system and is a function of the Nyquist frequency. With TDC, the sampling rate not only effects system stability but also system performance. Figures 4-2 and 4-3 show how the sampling rate effects the performance of radial bearing 2X when the remaining controller variables are held constant.

While determination of the proper sampling is crucial in producing a stable system, it is not the only factor. There are other controller parameters that also effect the stability of the system and these parameters cannot be ignored in determining the proper sampling rate. Thus




**4-2** Radial Bearing 2X Closed Loop Frequency Response as a Function of Sampling Rate




**4-3** Radial Bearing 2X Disturbance Rejection Response as a Function of Sampling Rate

the effect that sampling rate has on the stability of the system must be viewed in concert with the effects the other parameters also have upon system stability.

The equations of motion of the rotor define the acceleration in terms of control current and rotor position. Therefore the state variables of the controller must mimic the physics of the system and hence,

$$\begin{bmatrix} \ddot{x} \\ \dot{x} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} \dot{x} \\ x \end{bmatrix}$$

Obviously $a_{21} = 1$ and $a_{22} = 0$. The acceleration component has the following form,

$$\ddot{x} = a_{11}\dot{x} + a_{12}x \qquad \text{or} \qquad \ddot{x} - a_{11}\dot{x} - a_{12}x = 0$$

The Laplace transform of the above takes on a familiar form,

$$\left(s^2 - a_{11}s - a_{12}\right) X(s)$$

Thus,

$$-a_{11} = 2\zeta\omega_n \qquad \text{and} \qquad -a_{12} = \omega_n^2$$

where: $\zeta$ = damping ratio
$\omega_n$ = natural frequency

The simplified TDC law now becomes,

$$U(t) = U(t-T) - \frac{1}{B}\begin{bmatrix} \ddot{x}(t-T) + 2\zeta\omega_n\dot{x}(t) + \omega_n^2 x(t) \\ \dot{x}(t-T) - \dot{x}(t) \end{bmatrix}$$

Since a necessary condition for stability is that the unknown system dynamics do not change appreciably between sampling intervals, this implies,

$$\dot{x}(t-T) \approx \dot{x}(t)$$

Finally, the simplified TDC law becomes,

$$U(t) = U(t-T) - \frac{1}{B}\left[\ddot{x}(t-T) + 2\zeta\omega_n\dot{x}(t-T) + \omega_n^2 x(t)\right]$$

The reason for the substitution of the velocity terms in the above equation will become apparent later. The TDC law requires both the acceleration and the velocity of the rotor whereas the system only provides the position signal. Therefore the derivative must be calculated and for the time delayed values, a backward difference representation is used,

$$\dot{x}(t-T) = \frac{x(t)-x(t-2T)}{2T} + O(T)^2$$

The velocity is derived by the position signal supplied by the position sensor. The acceleration however is derived using the previously derived velocity. This introduces a problem because a characteristic of differentiators is that they amplify signal noise. Therefore, it is quite possible that the actual acceleration derived from the velocity approximation may be overwhelmed by the signal noise especially in applications where the sensor is particularly noisy.

Finally, the parameter $B$ is the conversion factor between acceleration and control current. This conversion factor is defined by the physical characteristics of the magnetic bearings. If this conversion factor is denoted as $b$, then the value of parameter $B$ is defined as,

$$\frac{1}{B} = \frac{K}{b}$$

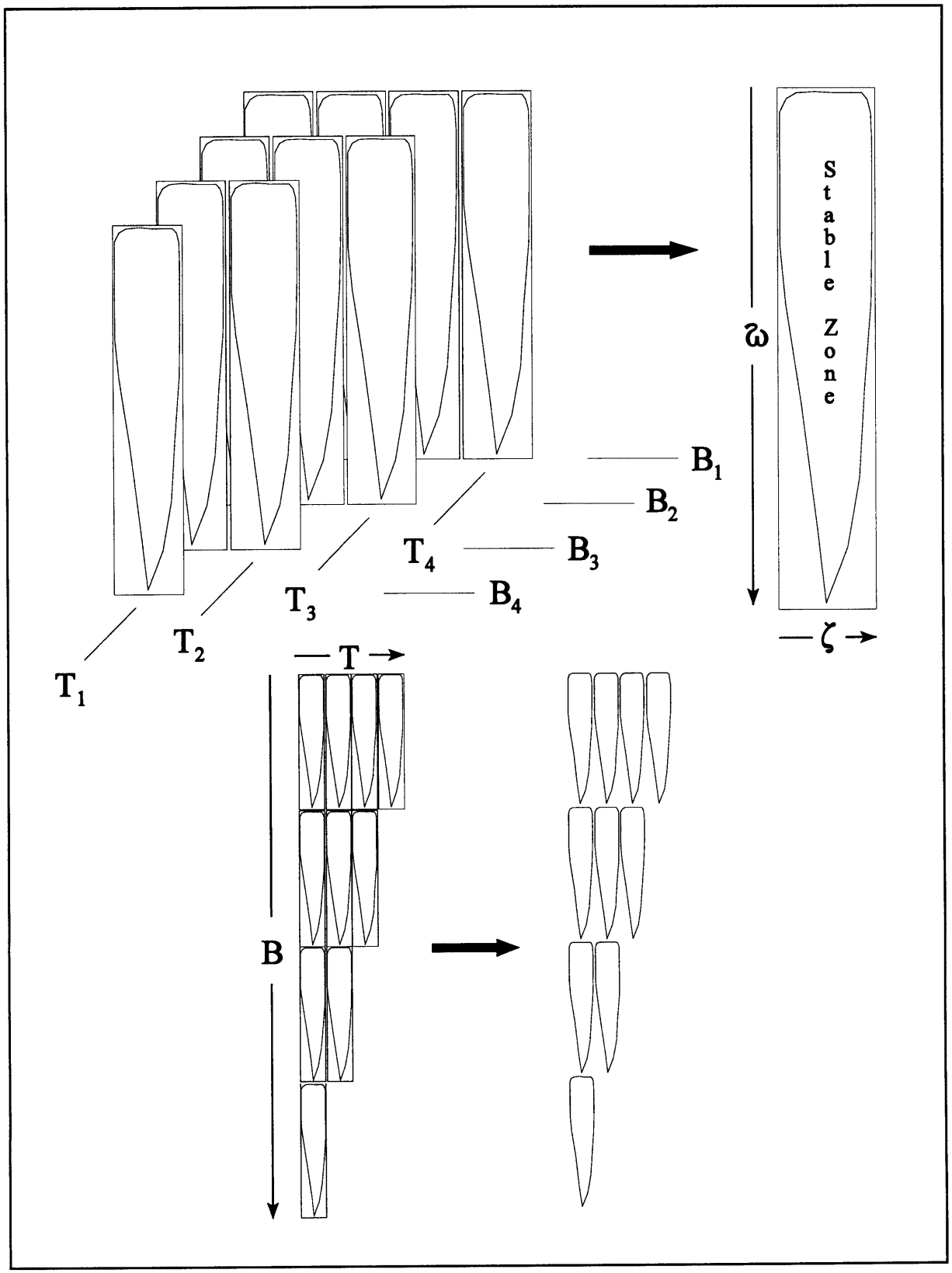
where: $K$ = feedback gain

Throughout the remainder of this thesis, $B$ will be referred to as the feedback gain and not by its separate components. As such it is not a true gain but does contain the conversion factor and has the appropriate units.

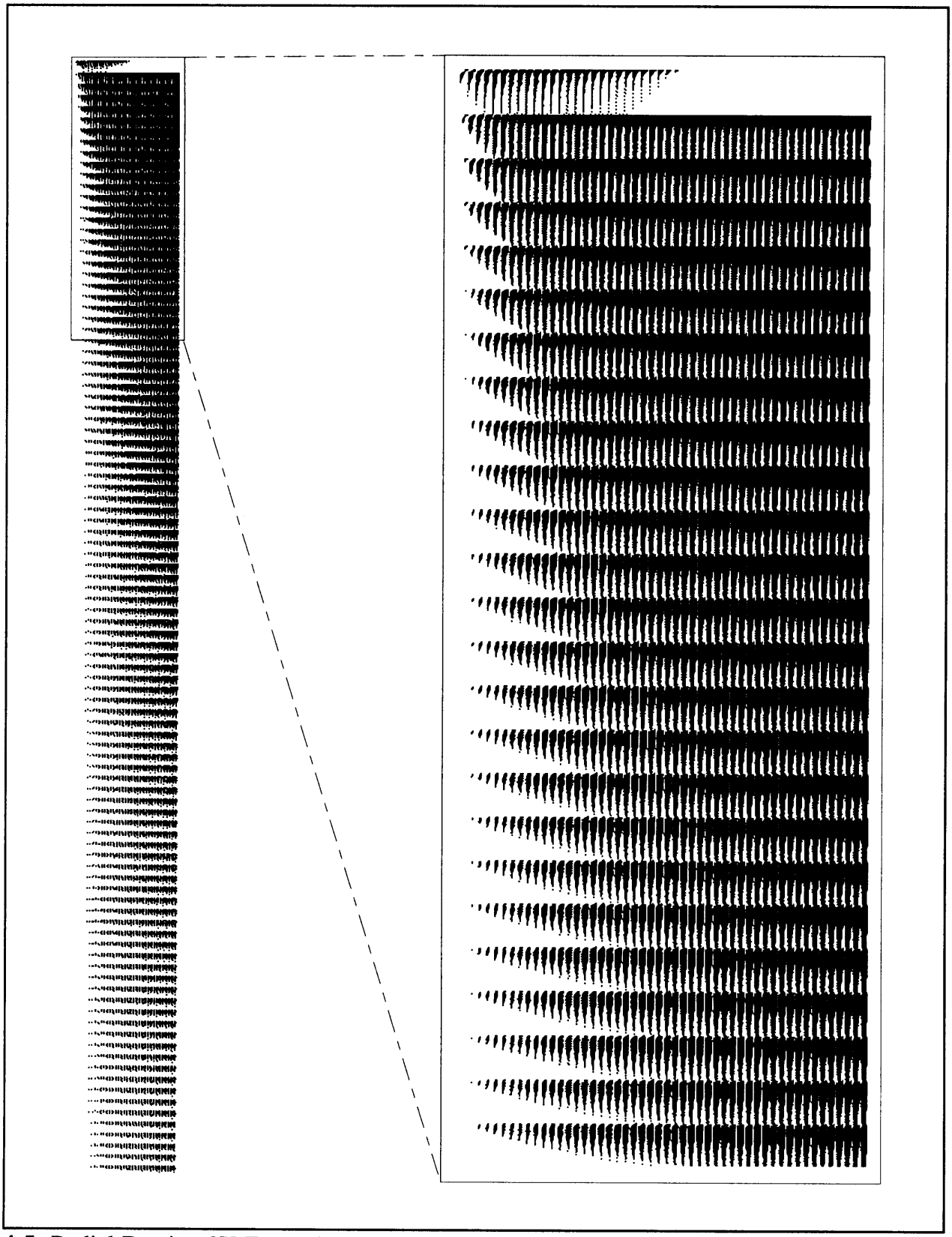
The parameters of the controller have thus been defined as,

$T$ = sampling interval
$B$ = feedback gain
$\zeta$ = damping ratio
$\omega_n$ = natural frequency

The objective is to determine how variations of the four controller parameters effect the stability of the closed loop system. In an effort to visualize this four dimensional variable space, a two dimensional plot was created in which each element of that plot was itself a two dimensional plot. Figure 4-4 is provided as an aid in understanding this visualization technique. The overall plot has a horizontal axis which charts the sampling interval and a vertical axis which charts the feedback gain. Each elemental plot has a horizontal axis which charts the damping ratio and a vertical axis which charts the natural frequency. All of these plots have their origin at the top left corner with the horizontal variable increasing from left to right and the vertical variable




**4-4** Four Dimensional Parameter Space Mapping



**4-5** Radial Bearing 2X Four Dimensional Parameter Space Stability Plot

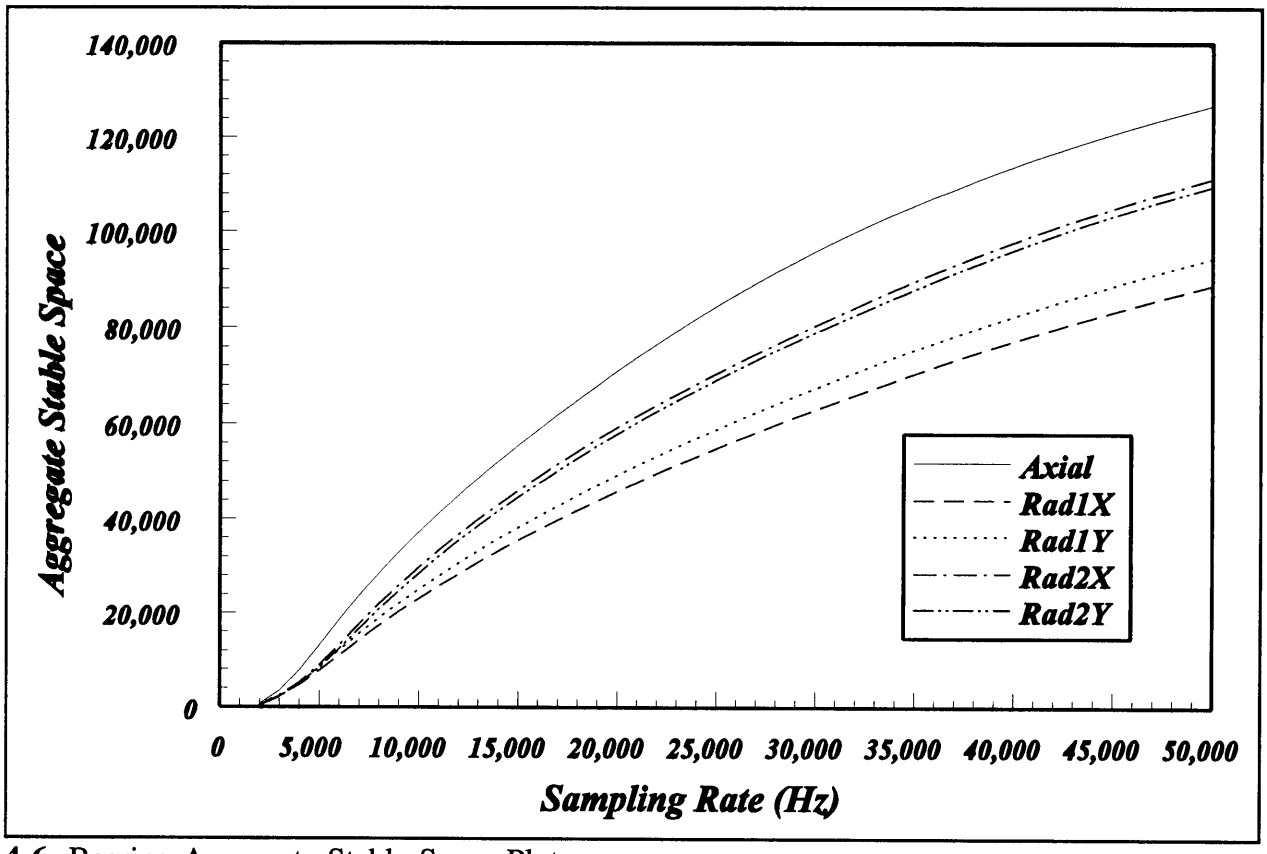
increasing from top to bottom. Within each elemental plot, a stable system was denoted by a plotted point. This four dimensional space was converted to a two dimensional plot by laying the elemental plots end to end. The axis lines delineating the elemental plots were then removed so as not to be confused with the plotted data.

Figure 4-5 represents just such a plot for radial bearing 2X. The parameter ranges over which the data is plotted are,

| Parameter | Range |
|---|---|
| Sampling Interval (1/Hz) | 1/1000 -1/ 50000 |
| Feedback Gain | 10 - 1000 |
| Damping Ratio | 0.1 - 10 |
| Natural Frequency (rads/sec) | 100 - 1000 |

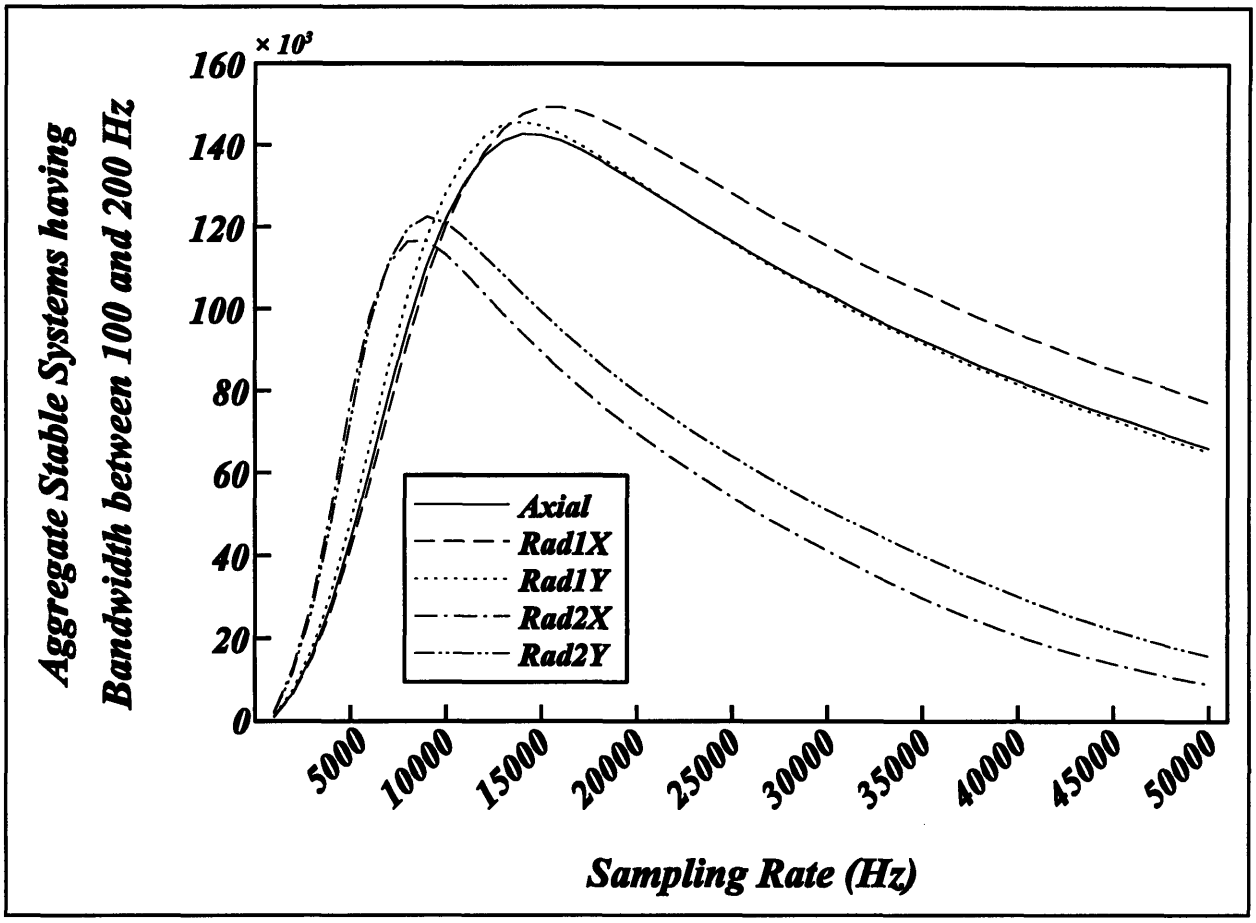
One notable trend is that the stability data is clustered into contiguous regions and not scattered throughout each elemental plot. The damping ratio and natural frequency parameters exhibit trends that are somewhat different from the remaining parameters. Both the damping ratio and natural frequency have increasingly larger stability zones as their values increase. Eventually these stability zones peak and then begin to decrease as the values continue to increase. On the other hand, the sampling interval exhibits increasingly larger stability zones as its value increases whereas the feedback gain parameters exhibit the opposite trend. However, the emphasis is to determine the optimal sampling rate and stability alone is not sufficient to allow such a determination.

The next step is to define just what constitutes the optimal sampling interval. The assumption is made that the optimal sampling interval is the interval which produces the largest aggregate stability region. As the aggregate stability region increases, the upper and lower bounds of the remaining parameters that produce stable systems also increases. As the parameter ranges increase, the number of possible stable controllers increases. As the number of possible controllers increase, the likelihood of finding the optimal controller within that region also increases. Simply put, the larger the aggregate stable space, the greater the probability of finding the optimal controller within that stable space. The aggregate stable space is merely the sum of the plotted stable systems having a common sampling interval. This assumption by itself does not alter the conclusions from the current analysis because the plot clearly shows that as the sampling interval decreases, the aggregate stability space increases. Figure 4-4 displays the aggregate stable space as a function of decreasing sampling interval for all five axes.




**4-6** Bearing Aggregate Stable Space Plot

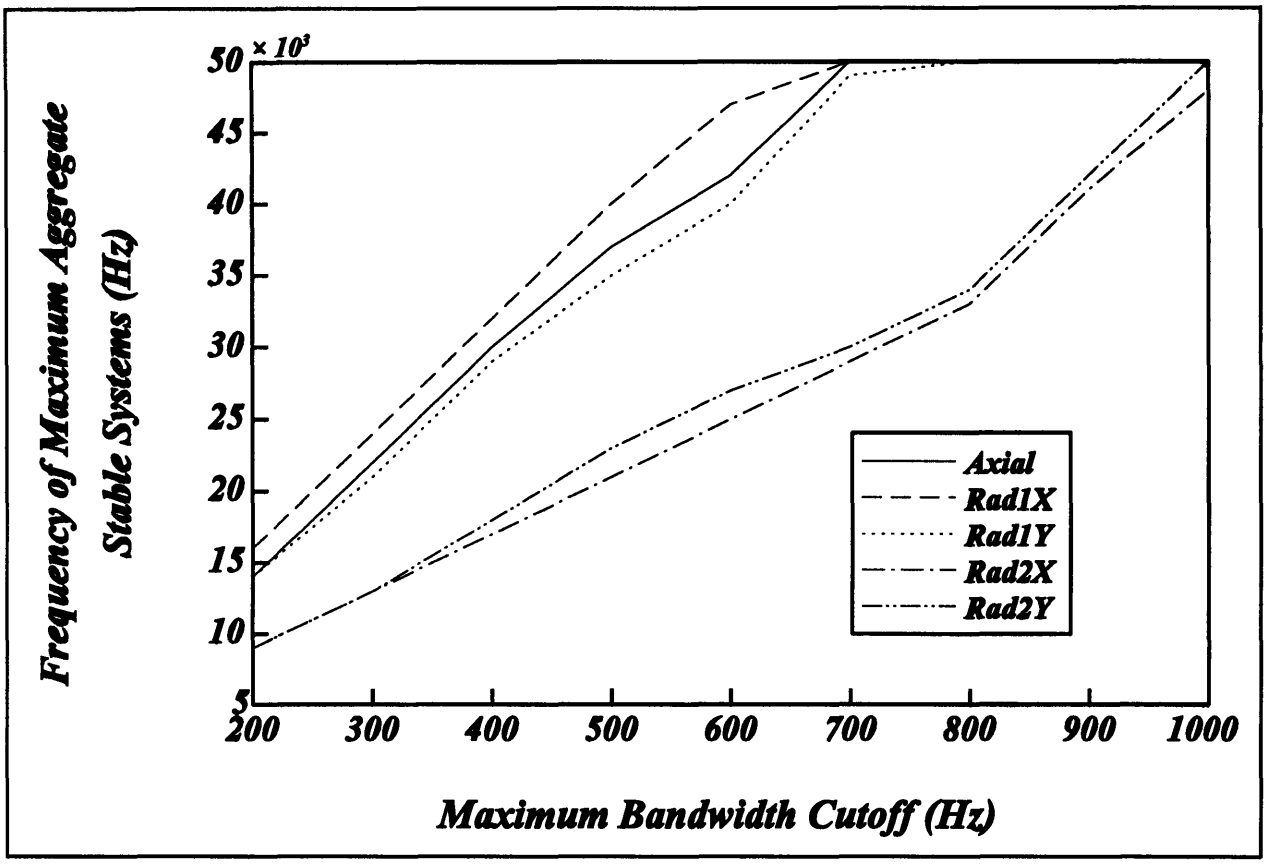
Using the aggregate stability space assumption, a further constraint is placed upon the analysis. Given the set of points defined by the four controller parameters that produce stable closed loop systems, how many also produce bandwidths that fall within the probable bandwidth of the optimal controller. The upper bound of the bandwidth of the controller is defined by the use of the best fit transfer function as the model of the open loop system. This model is only valid up to 1000 Hz and therefore the bandwidth upper bound must also conform to that limit. However there is the added constraint that the control signal be adequately attenuated at the model limit of 1000 Hz. The theoretical closed loop system has six zeroes and seven poles. Therefore the slope at high frequencies will be -20 dB/decade. Assuming a maximum gain of -20 dB at 1000 Hz and a slope of -20 dB/decade, the maximum bandwidth upper bound of approximately 200 Hz. The determination of the lower bound however is still a matter of engineering experience which later must be verified by the actual system performance. Ideally, the bandwidth of the system should be as close to the maximum allowable as possible. Therefore the lower bound was chosen as 100 Hz.




**4-7** Aggregate Stable Systems having Bandwidth between 100 and 200 Hz

Figure 4-5 shows the results of this added constraint. Not only is there a maximum aggregate stable space for each axis but that maximum falls within the attainable minimum sampling interval set by the combined factors of DSP chip speed, I/O hardware conversion rates, and algorithm efficiency. This figure also highlights a problem with using TDC as the control algorithm for this particular application. TDC requires that the sampling interval be sufficiently small for stability to be assured. However, given the current assumptions in the present analysis, the optimum sampling interval is different for each axis. Yet, if only one DSP chip is to be used, then only one sampling interval can be chosen. Hence, there is a high probability that the theoretically optimum controller will never be realized if TDC is the control algorithm.

This also raises the question of which sampling interval to choose. For this particular application, the failure of one axis causes the entire system to fail. Therefore the system sampling interval should be optimal sampling interval of the axis that is most likely to fail first. Again, engineering experience dictated that either the 2X or 2Y radial bearing would be the first to fail due to their larger distance from the rotor center of gravity. Later testing on the actual system would prove this assumption correct. Given all of these factors, the sampling rate of the system was set at 10000 Hz.




**4-8** Optimal Sampling Rate versus Bandwidth Constraint Upper Bound

There are also problems associated with this method of determining the optimal sampling rate. The sampling rate at which the maximum aggregate stable systems are produced varies with substantially with the upper bound of the allowable bandwidth as shown in Figure 4-6. Ideally, the bandwidth upper bound should be approximated initially and modified once the bandwidth of the actual closed loop system is obtained experimentally. This in turn would lead to a new optimal sampling rate which would cause the bandwidth of the actual closed loop system to change again. Hence the optimal sampling rate analysis becomes a recursive process which halts when there is little change in either the sampling rate or the actual closed loop system bandwidth. This recursive method was not used by this researcher due to the lack of easy access to a system analyzer.

The purpose of the sampling interval analysis was to eliminate the use of engineering experience as the criteria for its determination. In that regard it has not been wholly successful. However, it is the contention of this researcher that using engineering experience to estimate the upper and lower bounds of the probable bandwidth of the closed loop system is a far better proposition then basing its determination on experience alone.

### 4.2.2 Velocity Derivation

Previously, the simplified TDC law was shown to be,

$$U(t) = U(t-T) - \frac{1}{B}\begin{bmatrix} \ddot{x}(t-T) + 2\zeta\omega_n\dot{x}(t) + \omega_n^2 x(t) \\ \dot{x}(t-T) - \dot{x}(t) \end{bmatrix}$$

A necessary condition for stability is that the unknown system dynamics not change appreciably between sampling intervals. This implies,

$$\dot{x}(t-T) \approx \dot{x}(t)$$

This provides an opportunity to choose which term should be used to represent the velocity in this implementation of the TDC law. The proper choice is related to the method by which each term is determined when implemented in digital form. The choice then becomes one between the backward difference representation or the central difference representation,

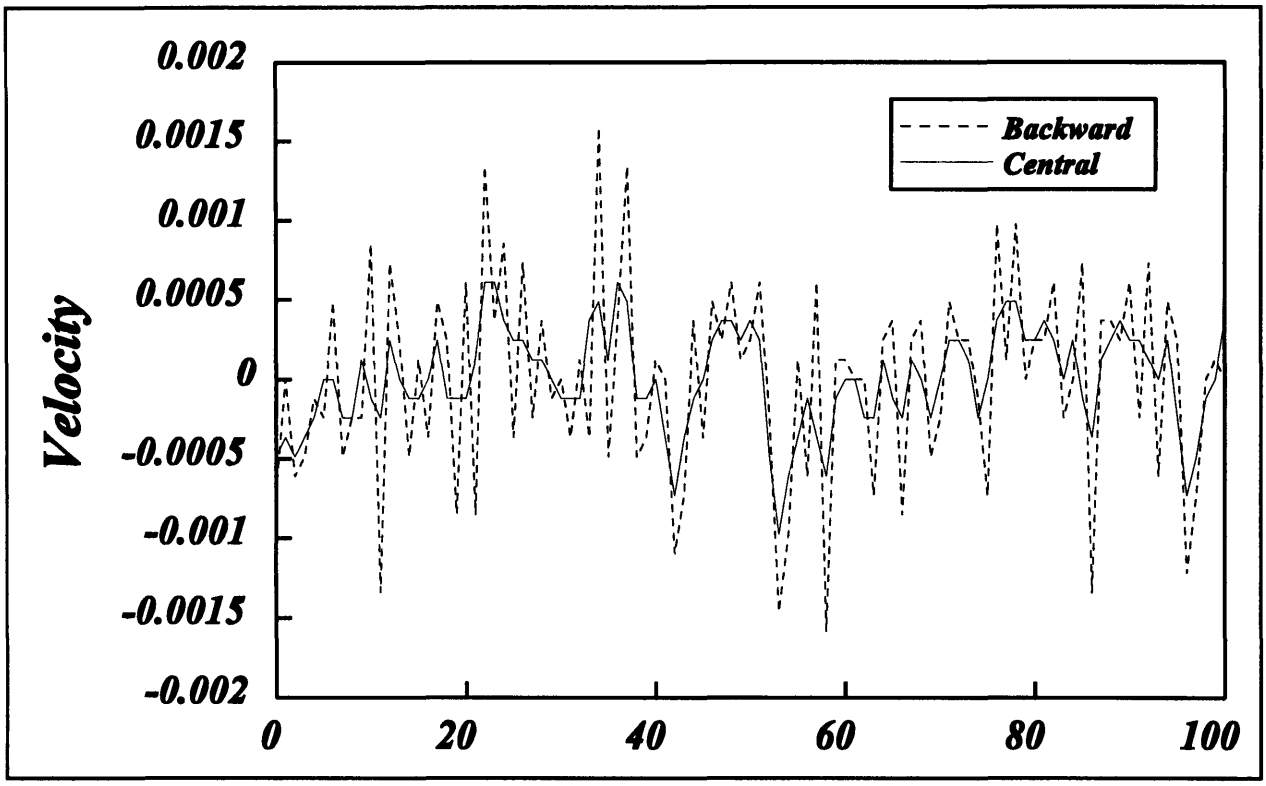
$$\dot{x}(t) = \frac{3x(t) - 4x(t-T) + x(t-2T)}{2T} + O(T)^2$$

$$\dot{x}(t-T) = \frac{x(t) - x(t-2T)}{2T} + O(T)^2$$

The best representation was determined by calculating the velocity returned by each difference formula from a sample of the noise generated by the position sensors. This noise sample was obtained by sampling the position signal of each axis when the magnetic bearings of all the axes were unpowered. Figure 4-7 shows a portion of the results from such an analysis when performed on the 2X radial bearing. Clearly, the central difference representation is less responsive to the sensor noise and therefore $\dot{x}(t-T)$ was chosen as the term to represent the velocity in the TDC law.

### 4.2.3 Optimal Controller Determination

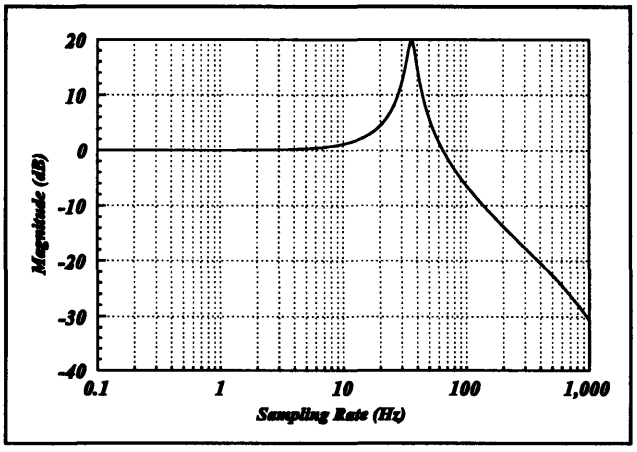
With the value of the sampling interval resolved, there still remains the question as to how the values of the remaining controller parameters will be determined. Many assumptions were made in the radial bearing equations of motion to reduce their complexity enough to allow the use of classical design techniques. Due to these assumptions, the major requirement of the controller is that it be as stiff as possible. Therefore the maximum value of the disturbance rejection plot is of paramount importance. Due to the use of the best fit transfer function as the open loop model of the system, there must be adequate attenuation of closed loop system at and above 1000 Hz. Another design criteria is to maximize the bandwidth of the closed loop system. Higher bandwidth allows the system to react to higher frequency signals. This leads to faster rise times and generally faster system response.




**4-9** Backward Difference and Central Difference Velocity Comparisons

Finally, the maximum closed loop gain must not exceed 10 dB. This design criteria requires some explanation. The response of the closed loop is dominated by two underdamped poles. If the closed loop system is viewed as a purely analog system (i.e. the digital controller can be accurately represented by Laplace transforms), these poles lie in the left half plane symmetrical to the real axis. As the parameters of the controller change, these poles move within the left half plane region. However, as the poles move closer to the right half plane, the system becomes more oscillatory. The movement of these poles closer to the right half plane also produces a bode plot with a maximum magnitude exhibiting a spiked appearance as shown in Figure 4-8. Therefore, the final criteria is that the maximum closed loop gain shall not exceed 10 dB. This upper bound effectively limits how oscillatory the closed loop system will become.

The performance criteria outlined above provide guidelines for designing the controller. However, the majority of these criteria are vague and provide no concrete numerical values on which to base calculations. The question is then how does one design a controller using these criteria.




**4-10** Peak From Underdamped Poles

The remainder of this section with describe one method of designing a controller using the above criteria. It is by no means the only way nor is it the best way to approach this design problem.

The vagueness of the performance criteria and the magnitude of the aggregate stable space necessitated a brute force approach. From the data used the map the four dimensional stable space, a subset of the stable systems having a sampling interval equal to the optimum sampling interval was removed. Using this subset, a histogram of each of the parameters $B$, $\zeta$, and $\omega_n$ was produced. Using the mean and standard deviation of each histogram, the outliers of each parameter were removed thus compressing the upper bound and lower bound of each parameter. A program was written that recursed through the allowable range of each parameter and calculated the following,

1. System stability
2. Maximum compliance
3. Maximum closed loop gain
4. Bandwidth
5. Closed loop gain at 1000 Hz

From this data, all unstable systems and systems with maximum closed gains in excess of 10 dB were removed. The remainder was sorted by increasing maximum compliance, bandwidth, increasing maximum closed loop gain, and gain at 1000 Hz. This data would be the basis on which the final controller would be designed. This analysis was performed on each individual axis.

This produced a large number of possible controllers but considering the assumptions made to simplify the model, this number would most likely drop substantially when tested on the actual hardware. This was especially true when examining the beginning of the sorted controller list. The controllers having small maximum compliances had small values for the feedback gain parameter $B$. Experience with the turbopump system suggested that controllers having such small feedback gains would not produce stable systems. Due to the number of simplifications made to the theoretical model, the controller list was not viewed as a strictly quantitative solution. However the data could be used qualitatively, thus providing a starting point and also enumerating trends in how parameter variations effected different performance criteria.

## 4.3 Controller Implementation

Considerable engineering went into designing the digital controller in the form of analysis and simulations. However, designing the controller is only half the battle. Careful engineering must also go into the implementation of such a controller on the given hardware. Such aspects as DSP chip architecture and speed, memory sizes, and I/O board conversion rates are all taken into account during the implementation process. The implementation process took on added significance given the inflexible design of the I/O Controller board. This section attempts to explain the design decisions made during the implement process and shed some light on the somewhat complex structure of the final program.

### 4.3.1 Parallel versus Serial Processing

Perhaps the most critical design decision concerned the control signal processing paradigm. Parallel processing is one way in which the control signals could be processed and this was method that the I/O Controller board designer assumed would be used. With parallel processing, a conversion process is triggered which causes all D/As to output the control signal values calculated during the previous sampling interval. After all output values have been converted, the I/O Controller turns its attention to the A/Ds and converts all incoming values for processing by the DSP chip. The digital controller would then sequentially examine each input value form the A/D FIFO, calculate the proper control signal, and place the control signal value in the D/A FIFO. The controller would then wait till the next sampling interval to trigger a conversion process which would again send out the values in the D/A FIFO and replenish the A/D FIFO with new converted values.

The advantage of this technique is reducing the overhead associated with the A/D and D/A conversion process. The disadvantage is that the control signal calculated from information gathered during the current sampling interval is not applied until the beginning of the next sampling interval. This time lag means that the control signal may be outdated and therefore could actually degrade system performance. Of course, all digital controllers are subject to computational lags but good engineering should keep these lags to the bare minimum.

There is a method that the digital controller can employ that will reduce the time delay to a minimum. When the timer triggers the digital controller, the controller can notify the I/O Controller board that there are no D/As before triggering the conversion process. Then after determining the updated control signals, the controller can place the values in the D/A FIFO, notify the I/O Controller board that there are no A/Ds, and trigger a conversion process. This technique eliminates the lag spent waiting for the next timer event to occur. There is however a lag induced by that requirement that all control signals for all the D/As must be calculated prior to triggering the conversion event. This lag becomes worse as the number of axis under digital control increases. There is also a slight increase in computation time incurred from wait states imposed by the I/O Controller board while updating the number of A/Ds and D/As in use during each conversion process.

The alternative to parallel processing is serial processing. However, due to the design of the I/O Controller board, this form of processing incurs substantial overhead. With serial processing, the controller is only processing the values for one axis. Since the A/Ds and D/As are not individually addressable, the conversion process must take place for all axes as in the parallel processing method. Only one of the values however is required and therefore the controller incurs the overhead of converting unnecessary values. For this particular system, the A/D conversion time was approximately 20 ms per axis. In fact, the overhead incurred through unnecessary conversions was longer then the control algorithm computation time.

Another disadvantage of serial processing is that the computation time for each axis must be significantly less than the sampling interval. With parallel processing the average computation time per axis is merely the sampling interval divided by the number of axes. If the value of the input causes one axis to take a little longer to compute, the extra time could be regained by an easy computation on another axis. With serial processing, the computation and conversion time must be one-fifth the sampling interval or interrupts will be lost. With the added overhead of

the conversion process inherent to serial processing, the algorithm efficiency becomes of paramount importance.

Given the disadvantages outlined above, parallel processing would seem to be the obvious choice. However, serial processing has one important advantage when used with a system of this type. Parallel processing outputs all of the values simultaneously and inputs all the values simultaneously. In a system where coupling between one or more of the axes is expected, this method enhances the coupling effect. Serial processing by its very nature allows some control of the coupling effect through proper choice of axis computation order. For instance, if radial bearing 1X and radial bearing 2X are assumed to be coupled, the order of computation might be 1X, then 1Y, then 2X, then 2Y. The time in between processing 1X and 2X would allow some of the coupling effects to appear in the other bearing. Therefore the digital controller would have the chance to compensate for the changes induced by the control signal imparted on another axis. This advantage was deemed important enough to offset all previously cited disadvantages and serial processing was chosen as the computational method.

### 4.3.2 Digital Controller Program Structure

Algorithm efficiency is particularly important with this application and the structure of the controller program reflects that. Any variable that might possibly require modification is defined as a constant at the top of the program. The vast majority of the code calculates the variables needed in the control algorithm from these constants during the initialization phase. For instance, the values of $\zeta$ and $\omega_n$ are defined as constants for each axis at the top of the program. The initialization code then calculates the value of $2\zeta\omega_n$ for use by the control algorithm and stored as a variable in data memory. If the algorithm requires division by a constant, the reciprocal is calculated during the initialization process and stored in a data variable. This is required because the division process takes seven cycles while the multiplication process only requires one.

Data variables stored in data memory can be accessed directly by name or indirectly by using pointers. The DSP chip can process two instructions per cycle if one instruction involves on chip calculation and the other involves memory access. However that memory access must be through pointers and not through direct addressing. Therefore pointers are used throughout the control algorithm. The data variables are organized in memory in the order in which they will be used in the program. There is no overhead incurred when data memory is addressed sequentially therefore careful organization of data variables improves processing speed.

The maximum amperage that can be drawn by all five axes is ten amps. Therefore the controller limits the maximum control signal to two amps. The control signal is checked before being sent to the D/A FIFO and truncated to ± 2 amps if required. The TDC algorithm utilizes the value of the previous control signal to compute the current control signal. If for some reason the system is unresponsive, the value increases until the system reacts. For high gain controllers, the control signal can build quickly but because of the amperage limit, the system will saturate. Thus the system exhibits the same characteristics normally attributed to integrator windup. In order to prevent this, if the control signal exceeds ± 2 amps, not only is the output control signal truncated but the value of the control signal stored in data memory is also truncated.

The initialization phase is also responsible for calculating conversion factors, initializing the timer chip, and setting the DSP chip into interrupt mode. After completing all these tasks, the program runs in an endless loop waiting for a timer interrupt. When a timer interrupt is generated, the processor jumps to the interrupt table which in turn jumps to the digital controller routine. The routine queries the axis number data variable and immediately triggers a conversion event. Since this will take some time, the algorithm initializes all the pointers based upon the axis number. The routine then polls the I/O Controller board waiting for the conversion to complete. Upon completion, the conversion factor is applied to the proper input value and the control signal is calculated. After testing for saturation, the routine notifies the I/O Control board that there are five D/As and no A/Ds. It then stuffs five values into the D/A FIFO and triggers a conversion event. The routine polls the I/O Controller board waiting for the conversion to complete and then updates the data memory variables for the next interrupt, notifies the I/O Controller board that there are five A/Ds and no D/As, and increments the axis number data variable.

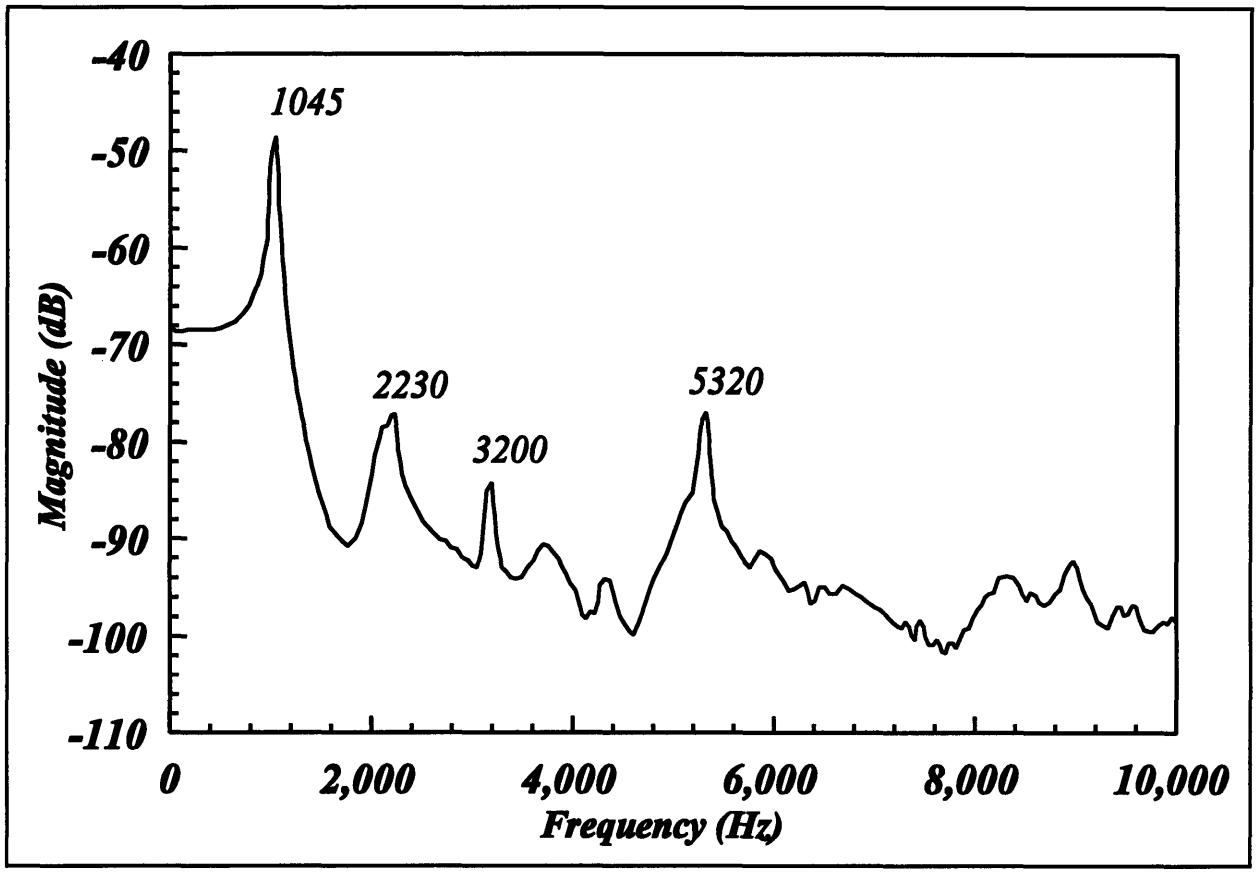
## 4.4 Other Implementation Issues

The objective thus far has been to design a robust digital controller employing the TDC law using classical design techniques. This could only be achieved by making specific simplifications to the model. Therefore, certain characteristics of the system and the hardware were ignored during the modeling process. However, these and other characteristics cannot be ignored during the implementation process because of their detrimental effect on the robustness of the controller. This section briefly describes certain system characteristics that were deemed to have a detrimental effect on overall stability and the methods used to remove their harmful effects.

### 4.4.1 Bending Modes

A significant problem inherent to high speed turbomachinery is rotor flexibility. Precautions must be taken to ensure that the control signal does not excite the major bending modes of the rotor which could lead to instability. Figure 4-9 is a frequency response plot provided by the manufacturer which displays the bending modes of a typical rotor assembly. Of particular concern are the first two bending modes at 1045 and 2230 Hz. One of the criteria in the design of the digital controller was that there be adequate control signal attenuation at 1000 Hz due to the limitations of the theoretical model. However, this attenuation may not be sufficient to guarantee that the first or second bending mode will not be excited. Therefore, as an added measure of insurance, notch filters were added to the system at the first two bending mode frequencies. Both filters were implemented in hardware using op amps and breadboards. These filters were incorporated in a filter box which lies underneath the I/O Interface board. All signals coming to the A/Ds and from the D/As must first pass through this filter box. A switch was also installed that would bypass the first bending mode notch filter. No provision was made

to bypass the second bending mode.

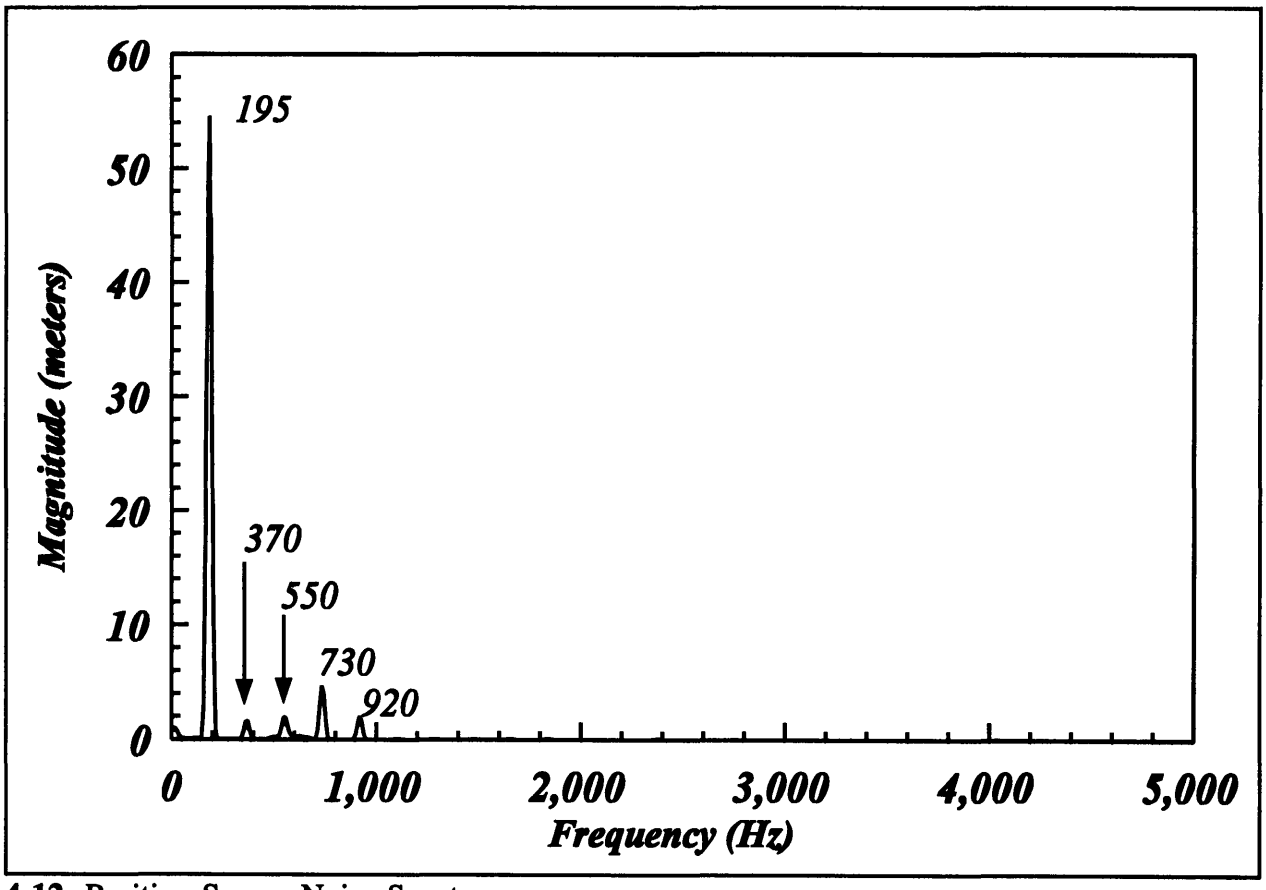



**4-11** Rotor Assembly Bending Modes

## 4.4.2 Sensor Noise

An important consideration in controller implementation is sensor noise. The sensor noise was obtained by capturing the sensor signal while the magnetic bearings were unpowered. In this mode, the rotor would be resting on the axial touchdown bearing and therefore essentially motionless. This captured data was then corrected to remove any offset since the rotor is not guaranteed to be in the center of the magnetic bearings when they are unpowered. A power spectrum analysis was carried out on this corrected data and the results plotted. Figure 4-10 shows that results of this power spectrum analysis of the 2X radial bearing. The results in this figure are representative of the data obtained from the other sensors. As shown, this system is particularly noisy at approximately the 200 Hz region which lies within the probable bandwidth of the controller. To minimize the impact of this noisy sensor, a notch filter was implemented in software to attenuate signals within this region. A software implementation was chosen so that the notch position and notch width could be easily changed to tailor each filter to the needs of each axis. Obviously, this would increase the control algorithm computation time but the added

versatility of the software implementation was important enough to warrant the delay. Special emphasis was placed on implementing a very efficient software notch filter in order to minimize the extra computation time needed.

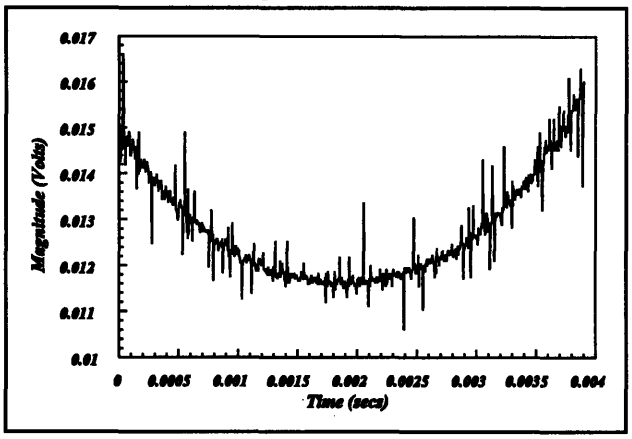



**4-12** Position Sensor Noise Spectrum

## 4.4.3 Anti-Aliasing Filter

There were two problems associated with the design of the filters used by A/Ds and D/As on the I/O Interface board. The first problem stems from the choice of filters used. The I/O Interface board was designed for very fast sampling rates. Therefore, the filters would also have to be very fast so that the actual signal level would be obtained quickly. In the instance of the D/A, a slower filter may not reach the required signal level before the next sampling interval changes the level. Fast filters require fast rise times and herein lies the problem. The choice of filter was a second-order Butterworth filter. As with all second-order systems, very fast rise times lead to very high overshoots which in turn introduces noise into the signal. These overshoots were very noticeable when viewed on an oscilloscope. An experiment was conducted in which a function generator was connected to an A/D and the HP System Analyzer was connected to a D/A. A small program was written which at each sampling interval, grabbed a

value from the A/D and output it directly to the D/A. Using this program, the HP System Analyzer captured a time series output of the D/A while inputting a simple sine wave via the function generator. The results of this experiment are shown in Figure 4-11.




**4-13** D/A Sine Wave Time Series

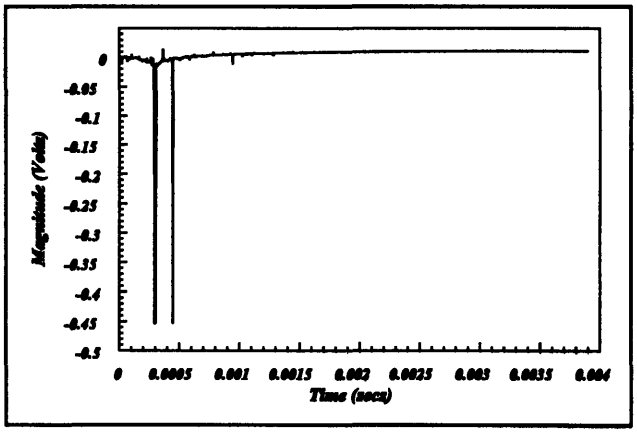
To alleviate this filter induced noise, the filter box contains low-pass filters for the D/As having a cutoff frequency of 15 KHz. These were implemented in hardware and proved sufficient to remove most of the energy of the spike. The same filters are also used for the A/Ds but there is no way to place a filter between the filters and the A/D FIFO. There is probably very little danger of the filters corrupting the A/D signal because the A/D settling time is most likely far slower then the filter settling time.

The second problem again arises due to the filters. Whereas the first problem primarily concerned the D/As, this particular problem effects the A/Ds. The filters are designed so that the cutoff frequency can be adjusted by replacing resistor packs on the I/O Interface board. A characteristic of the design of this board is that changing the cutoff frequency of the filters changes the DC offset of the A/Ds. To determine the DC offset, a number of values are read from the A/Ds that have been grounded. The average is taken and this value is used in the digital controller to determine the actual position signal. The table below lists the different offsets calculated for different cutoff frequencies.

| Filter Cutoff | 10261 Hz | 3386 Hz | 2822 Hz | 940 Hz |
|---|---|---|---|---|
| Offset (Volts) | 1.028460 | 0.258709 | 0.031324 | -2.700709 |

The design of the digital controller was based upon a theoretical model which was valid up till 1000 Hz. Since the performance of the controller could not be accurately predicted for higher frequency signals, the A/D anti-aliasing filter was set to a cutoff frequency of 1000 Hz. This resulted in a decrease in the effective dynamic range of the A/D from ±5.0 volts to ±2.3 volts. For the radial bearings, this would limit the position signal to ±92 μm or 36.8 percent of the bearing gap. Since it was impossible to guarantee that the controller would or could limit the amount of deviation of the rotor to ±92 μm of center, the filter on the I/O Interface board was set to a cutoff frequency of 2 KHz and a 1 KHz lowpass filter was implemented in software. This would of course require extra cycles in the control algorithm but the A/D dynamic range was needed.

### 4.4.4 D/A Glitch



4-14 D/A Zero Crossing Anomaly

The D/A also exhibited one other characteristic which to a large extent was alleviated by the 15 KHz lowpass filter. Figure 4-12 shows the results of a time series capture of the D/A output using the HP System Analyzer. As in the previous experiment, the function generator inputted a simple sine wave into the A/D and that value was outputted to the D/A. When the value output gets very small, the D/A value tends to spike to a very large magnitude. This occurs quite regularly and can be seen readily using an oscilloscope. It is unclear just what is caused this glitch but it has been speculated that the D/As do not handle the transition from negative to positive values correctly or that the transition to a value of zero is handled badly.

## 4.5 Filter Design

The previous section outlined how filters were used to eliminate destabilizing system dynamics and compensate for hardware design deficiencies. Whenever possible, these filters were implemented in hardware for speed reasons and to minimize computation time. However, certain filters required software implementation and the design of these filters is the subject of this section. With all filter design, a compromise must be made between filter characteristics and processor resources [16, 20].

### 4.5.1 Low Pass Filter Design

A fourth-order Butterworth analog filter was chosen as the basis for the low pass filter design. This filter provides good attenuation in over a reasonably small frequency range and the computation time requirements were not unrealistic. Designing a Butterworth filter is a cookbook process [14]. The order of the filter determines the limits of the filter characteristics. Given the following,

$\omega_1$ = cutoff frequency
$K_1$ = cutoff frequency attenuation
$\omega_2$ = target frequency
$K_2$ = target frequency attenuation

Normally $\omega_1$ is known and $K_1$ is -3 dB. Of the remaining parameters, $\omega_2$ and $K_2$, defining one also defines the other since the order of filter has already been defined. For instance, the low pass filters used with each bearing axis had the design parameters of $\omega_1$ = 1 KHz, $K_1$ = -3 dB, and $K_2$ is -30 dB. The target frequency was determined using the following,

$$\Omega_1 = \frac{2}{T}\tan\frac{2\pi\omega_1}{T} \qquad \text{and} \qquad \Omega_2 = \frac{2}{T}\tan\frac{2\pi\omega_2}{T}$$

$$n = \frac{\log_{10}\left[\frac{10^{-K_1/10}-1}{10^{-K_2/10}-1}\right]}{2\log_{10}\left(\frac{\Omega_1}{\Omega_2}\right)}$$

Using these equations, $\omega_2$ = 2090 Hz which should provide adequate attenuation. What remains is the conversion of the analog filter to a digital format. A fourth-order Butterworth filter has the following form.

$$\frac{Y(s)}{X(s)} = \frac{1}{s^4 + 2.613s^3 + 3.414s^2 + 2.613s + 1}$$

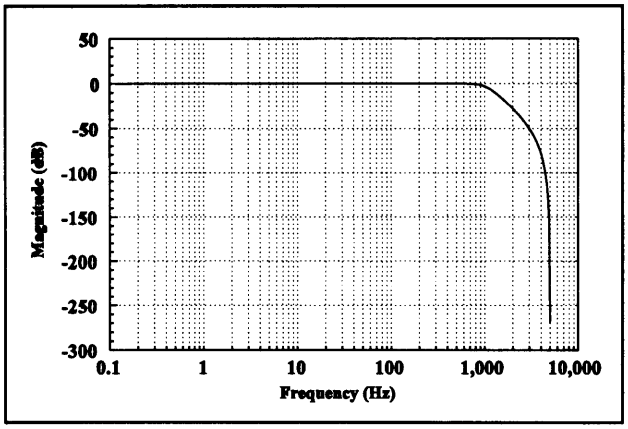
The conversion process begins by applying an order correction factor, $s = s/n$, where $n$ is the order of the filter. Then the filter is transformed using the Tustin approximation,

$$s = \frac{2}{T}\frac{z-1}{z+1}$$

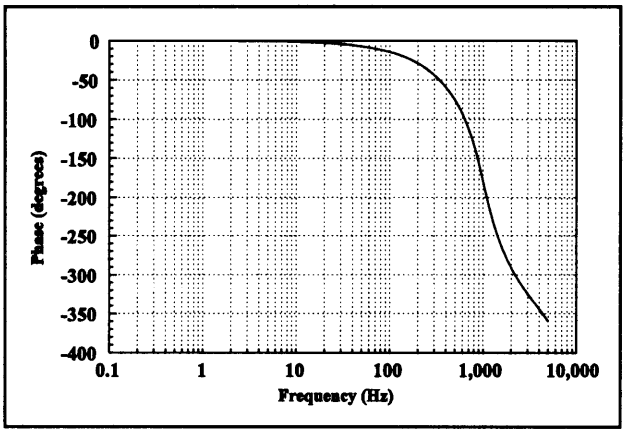
The final form then becomes,

$$\frac{Y(z)}{X(z)} = \frac{1 + 4z^{-1} + 6z^{-2} + 4z^{-3} + z^{-4}}{206.8088 - 489.8232z^{-1} + 478.2204z^{-2} - 217.9162z^{-3} + 38.7102z^{-4}}$$

The response of this filter is shown in Figures 4-13 and 4-14. This filter was implemented at a cost of 18 additional cycles.




**4-15** Low Pass Filter Frequency Response



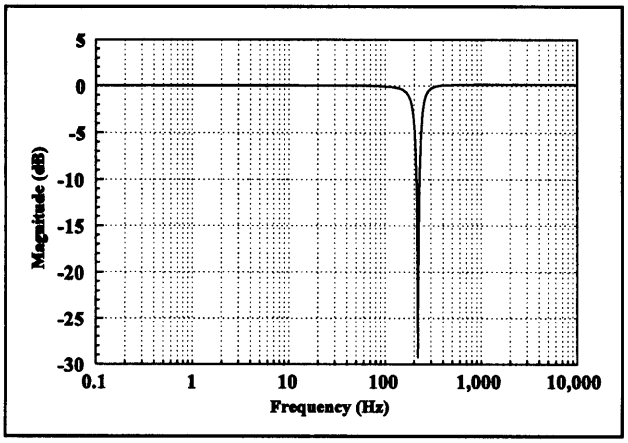

**4-16** Low Pass Filter Phase Response

## 4.5.2 Notch Filter Design

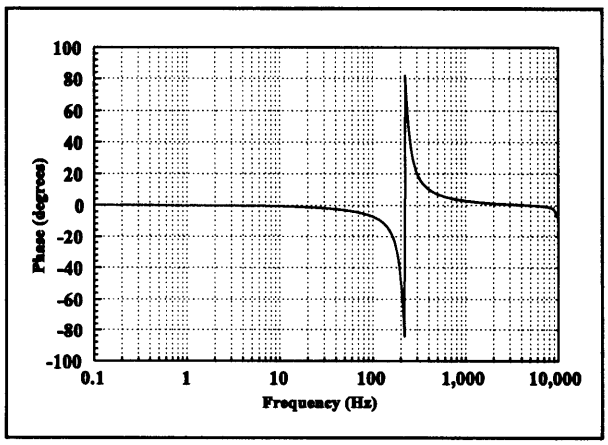
The notch filter is necessary to remove sensor noise from the position signal. This filter is a purely digital design having the following form,

$$\frac{y(t)}{x(t)} = \frac{x(t) - 2\cos\omega_0 T x(t-T) + x(t-2T)}{x(t) - 2\left(1 - \frac{\alpha L A_0^2}{4}\right)\cos\omega_0 T x(t-T) + \left(1 - \frac{\alpha L A_0^2}{4}\right)^2 x(t-2T)}$$

The parameter $\omega_0$ is the notch frequency and the parameters $\alpha$, $L$, and $A_0$ effect the notch depth and width. Typical notch filter values are $\omega_0$ = 220 Hz, $\alpha$ = $L$ = 0.25, and $A_0$ = 1.0. The response of this filter is shown in Figures 4-15 and 4-16. This filter was implemented at a cost of 9 cycles.




**4-17** Notch Filter Frequency Response




**4-18** Notch Filter Phase Response

## 4.6 Manual Tuning

With all of the simplifications that were necessary to allow classical design techniques to be used in designing the digital controller, there would be no way to avoid manual tuning. Not only would the controller require manual tuning but also the notch filter which removed sensor noise must be tuned to the appropriate frequency and notch width. The tuning process would rely on the sorted stable controller lists that were created for each bearing.

The initial step is to determine the sensor noise notch filter's notch frequency. This required sampling the sensor output while the bearings were unpowered. A power spectrum analysis was then carried out on this data to determine the notch frequency. Next, reasonable values for the parameters $\zeta$ and $\omega_n$ were set so that the proper feedback gain $B$ could be obtained. Experimentation had shown that of the remaining parameters, the feedback gain had the strongest effect on system stability. The values chosen for the parameters $\zeta$ and $\omega_n$ were 1.0 and 100.0 respectively. It is important that the trial and error determination of the feedback gain must be carried out on all five bearings simultaneously. Any attempt to tune each axis individually by using the analog controller for the remaining axes would lead to an unstable system when all five axes were under digital control. After the feedback gain for each axis has been determined, then the parameters $\zeta$ and $\omega_n$ can be tuned to optimize system performance.

Ideally the fine tuning of the parameters would be accomplished using a system analyzer to look at the frequency response and disturbance rejection curves generated by each axis. In the absence of a system analyzer, a reasonable analysis can be performed using a function generator and an oscilloscope. The function generator is used to supplied a low amplitude disturbance at the appropriate test point and the oscilloscope will be used to monitor the position signal. Using this technique, the maximum closed loop gain and bandwidth of each axis can be determined as well as maximum compliance of the system. When the performance of the system seems satisfactory, the rotor of the turbopump can be spun to a suitable speed and stability can be ascertained. When testing with the rotor of the turbopump spinning, make sure that the first test speed is the highest speed the digital controller must meet. If the digital controller remains stable at the highest speed, it will remain stable at the slower speeds.

## 4.7 Summary and Remarks

In this chapter, the controller design process has been outlined. A major requirement for stability of the TDC law is that the sampling interval be sufficiently small such that the system dynamics do not change significantly between sampling intervals. An analysis of the effect of the controller design parameters showed that there is no obvious optimal sampling rate. Therefore, statistical methods were employed to determine the sampling rate. The remaining controller design parameters were chosen based upon the performance requirements of the system. Using these requirements, the controller should have the smallest compliance possible, the maximum closed loop gain should be below 10 dB, there should be proper signal attenuation at 1000 Hz, and the bandwidth to the controller should be maximized. Possible controllers

meeting these requirements were ranked and used during the manual tuning phase.

The controller program structure was examined and design decisions defended. Other implementation issues specific to this application were presented and techniques for minimizing their detrimental effects on system stability were outlined. Some of these implementation issues were related to characteristics of the system and others were related to the hardware used to implement the controller. The techniques used to minimize stability problems caused by these characteristics consisted primarily of implementing filters and the filter design methods were presented. Finally, a brief account of the manual tuning methodology was presented.

The obvious conclusion from this chapter is that the problem of designing a controller for this particular system just got considerably harder. The design decision to use TDC and the limitation of having only one sampling rate for all axes means that an optimal controller will never be realized. Add to that the bending modes, noisy sensor, and hardware bugs and this problem becomes worse. Also, the simplifications necessary to allow classical design techniques to be used calls into question the validity of the stability analysis used to determine the sampling rate and the other controller parameters. The controller will only be as good as the theoretical model it was designed to control and therefore everything hinges on the how well the theoretical system response matches the actual system response.

# Chapter 5

# Controller Evaluation

This chapter compares the results of the performance of the digital, analog, and wherever possible the theoretical controllers. The first section describes the manual tuning process. The ease or difficulty associated with the manual tuning process is a good indicator of the validity of the theoretical model. The second section presents the system performance when the rotor is not spinning. Throughout this chapter, the radial 2X bearing was chosen to represent the response of a typical radial bearing. Both the closed loop frequency response and the disturbance rejection response are presented during static testing. The third section presents the system performance when the rotor is spinning at 15000 and 28000 RPM. Only the disturbance rejection response is presented because the closed loop frequency response conveys little information since command following is of minor importance in this particular application. The final section summarizes the results.

## 5.1 The Manual Tuning Process

There were problems with the manual tuning process almost immediately. Due to the assumptions made during the modeling process, the most important system performance criteria was the maximum compliance. The dominant characteristic of systems having a low maximum compliance as predicted by the theoretical model was high feedback gain. However, it became apparent that such high feedback gains predicted by the theoretical model would lead to an unstable system. Another critical simplification of the theoretical model was that coupling could be treated as a disturbance. This simplification implies that any coupling between bearings is small when compared to forces exerted by the bearings themselves. The validity of this assumption was tested by manually tuning each radial bearing while the appropriate coupled bearing was under analog control. If coupling was indeed small between these two bearings, little or no adjustment would have to be made when both coupled bearings were under digital control. This was not the case however. In fact all four radial bearing controllers had to be manually tuned simultaneously.

Simultaneously tuning four radial bearings requires a systematic approach. This researcher was able to obtain a stable system in a timely fashion by setting the damping ratio and natural frequency to approximate values (i.e. $\zeta = 1.0$, $\bar{\omega} = 100$) and varying the value of the feedback gains. Once an approximate value of the feedback gain was obtained for each bearing, the values of the damping ratio and natural frequency could be tuned to improve system

performance. System performance was determined by using a function generator and an oscilloscope to monitor the disturbance rejection response of each bearing. As demonstrated by the theoretical model, the stability region was a contiguous area and is not a random scattering of stable systems distributed throughout the entire variable range. Therefore it was only a matter of finding the right parameters that yielded a stable controller and, through trial and error, probing for the limits of the stability region. Using this technique, this researcher was able to develop a stable controller able to yield good performance up to 15000 RPM in a relatively short time.

However, developing a controller that remained stable up to 28000 RPM was another matter. Simply varying the parameters of the controller was not yielding a controller stable enough to pass the 25000 RPM range. Time and again it was the lower bearing which failed first and caused the analog control box to shut down the system. Stability above this 25000 RPM range only came after this researcher replaced the hardware implementation of the first bending mode notch filter with a software implementation. This allowed the notch filter cutoff frequency, notch depth, and notch width to be varied individually for each bearing. Altering the characteristics of the first bending mode notch filter also influenced the performance of the system at lower frequencies. It would turn out that the cutoff frequency of the notch filter would effect system performance enough to allow a greater operating speed. By varying the parameters of the notch filter as well as the parameters of the controller, this researcher was able to develop a stable controller capable of 28000 RPM after much trial and error. The notch filter implemented in software was of the following form,

$$\frac{y(z)}{x(z)} = \frac{z^2 - 2\cos\omega_0 Tz + 1}{z^2 - 2\left(1 - \frac{\alpha L A_0^2}{4}\right)\cos\omega_0 Tz + \left(1 - \frac{\alpha L A_0^2}{4}\right)^2 z}$$

Where: $T$ = sampling interval
$\omega_0$ = notch frequency
$\alpha$, $L$, $A_0$ = parameters effecting notch depth and width

The final values of the notch filter parameters are the following,

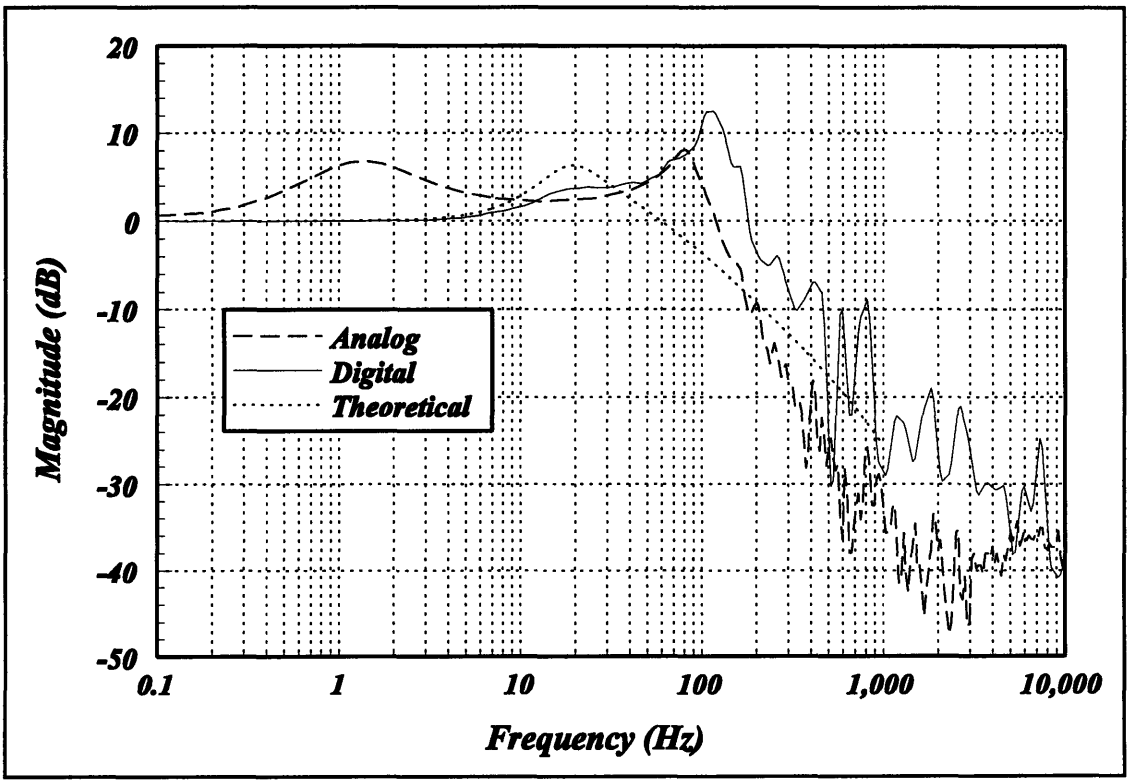
| Parameter | Axial | Rad1X | Rad1Y | Rad2X | Rad2Y |
|---|---|---|---|---|---|
| $\omega_0$ (Hz) | 950 | 1000 | 960 | 1100 | 1100 |
| $\alpha$ | 0.20 | 0.20 | 0.20 | 0.20 | 0.20 |
| L | 0.20 | 0.20 | 0.20 | 0.20 | 0.20 |
| $A_0$ | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |

Note that the above table presents the values for the notch filter used by the axial bearing. The axial bearing would not be effected by the first bending mode but the code used to control each bearing axis is common to all bearings. Including a notch filter in the axial bearing controller allowed streamlined code development and did not compromise axial bearing stability. Obviously the first bending mode cannot be different for each bearing axis. What moving the notch filter does provide is better controller performance and stability at high frequencies.

## 5.2 Static Test Results

This section displays the results of the closed loop frequency response and disturbance rejection response of the axial bearing and one axis of a particular radial bearing when the rotor is not spinning. The 2X radial bearing was chosen to represent the response of a typical radial bearing axis because this axis was the most likely to fail during testing and therefore represents the worst case response. In all cases, the theoretical response is the predicted response of the digital controller using the theoretical plant model. Both the analog and digital responses were determined using an HP System Analyzer.

### 5.2.1 Axial Bearing Test Results




**5-1** Axial Bearing Static Closed Loop Frequency Response

Figure 5-1 displays the closed loop frequency response of the axial bearing under digital and analog control as well as the predicted theoretical response. The following table shows the important performance values used as design criteria for the digital controller. These performance values are derived from the data points used to display the graphs in Figure 5-1.

| Parameter | Analog | Digital | Theoretical |
|---|---|---|---|
| Peak Gain (dB) | 8.06 | 12.50 | 6.33 |
| Bandwidth @ -3dB (Hz) | 134.01 | 195.09 | 91.64 |
| Gain @ 1000Hz (DB) | -32.82 | -18.53 | -25.55 |

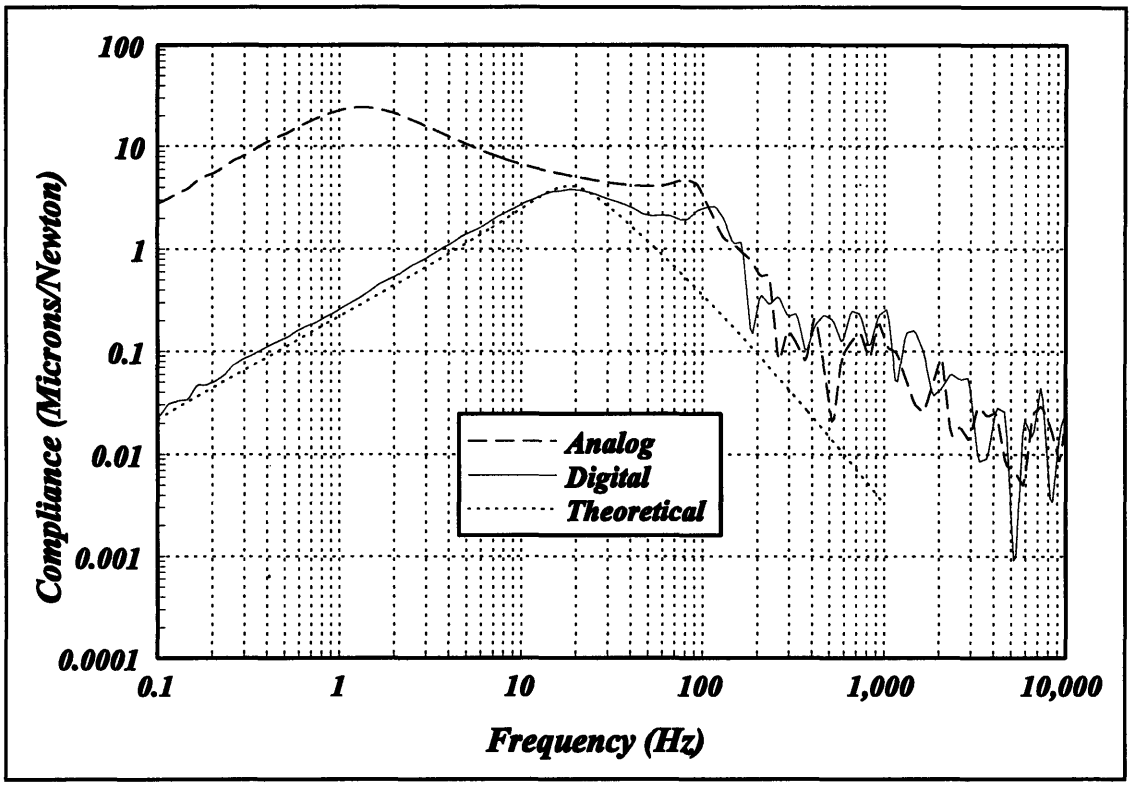
Analyzing this data leads to the following conclusions,

1. The theoretical response is not an accurate indicator of actual digital controller performance because of the very simple model used. Both the bandwidth and the peak gain of the digital controller are twice that of the theoretical model.
2. The digital controller has a bandwidth which is approximately fifty percent greater then the analog controller but it also has a peak gain which is approximately 1.7 times greater then the analog controller.
3. At frequencies beyond the bandwidth of the controller, the slopes of both the analog and digital controllers are approximately equal.

Figure 5-1 displays the most important findings in this thesis. Remember that the axial bearing is not subject to coupling effects and therefore the simplification of treating coupling effects as disturbances was not needed. Also the axial bearing is not subject to gyroscopic effects even if the rotor was spinning, which it isn't in this case, and therefore the simplification of treating gyroscopic effects as a disturbance was also not needed. The only simplification made in the case of the axial bearing was linearizing the inherently nonlinear equations of motion. Therefore, the linearization process must account for the discrepancy. The use of a very simple model is believed to be the cause of this anomaly.

Figure 5-2 displays the disturbance rejection response of the axial bearing under digital and analog control as well as the predicted theoretical response. The following table shows the important performance values used as design criteria for the digital controller. These performance values are derived from the data points used to display the graphs in Figure 5-2.

| Parameter | Analog | Digital | Theoretical |
|---|---|---|---|
| Peak Compliance (Microns/Newton) | 24.14 | 3.80 | 4.11 |
| Compliance @ 1000Hz (Microns/Newton) | 0.135 | 0.247 | 0.003 |



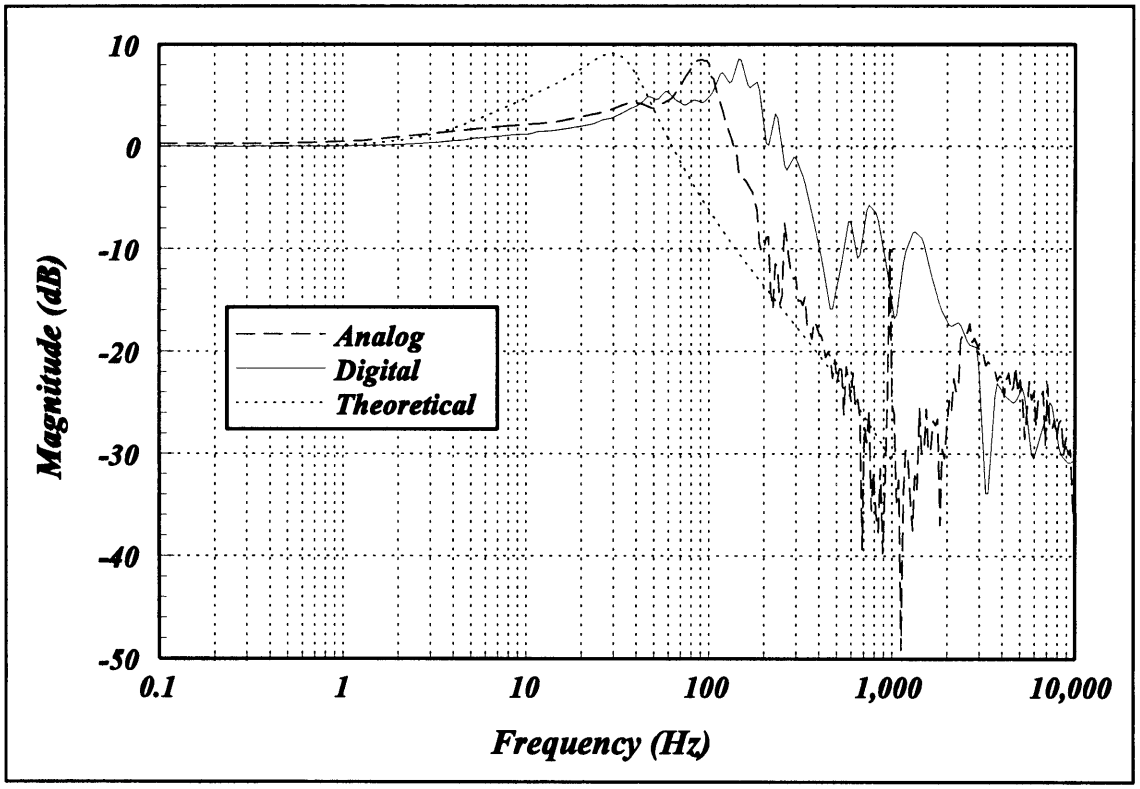

**5-2** Axial Bearing Static Disturbance Rejection Response

Analyzing this data leads to the following conclusions,

1. The theoretical response accurately predicts actual digital controller response at and below 20 Hz. However, the theoretical response becomes progressively less accurate as frequency increases.
2. The digital controller has significantly better disturbance rejection properties at and below 10 Hz. However, the fact that the analog controller has significantly worse disturbance rejection properties in this region and yet produces very stable operation leads one to believe that the low frequency disturbance rejection is of low importance as a controller design criteria for this bearing axis.
3. At frequencies beyond 100 Hz, the disturbance rejection properties of both the analog and digital controllers are approximately equal.

Disturbance rejection was considered the most important design criteria and the accurate prediction of the low frequency disturbance rejection response by the theoretical model lends credibility to the use of the theoretical model as a design tool when disturbance rejection is the most important performance characteristic. From the standpoint of disturbance rejection, the digital controller is significantly better then the analog controller when the rotor is not spinning.

## 5.2.2 Radial Bearing Test Results




**5-3** Radial Bearing 2X Static Closed Loop Frequency Response

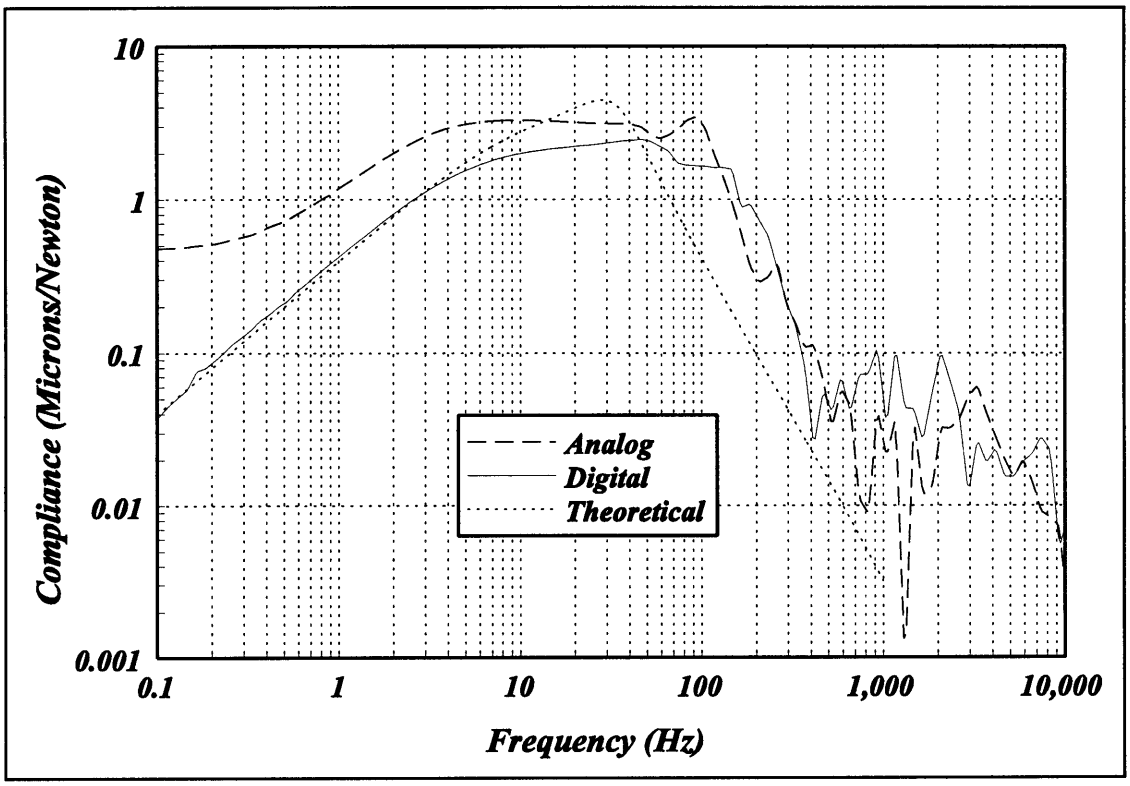
Figure 5-3 displays the closed loop frequency response of radial bearing 2X under digital and analog control as well as the predicted theoretical response. The following table shows the important performance values used as design criteria for the digital controller. These performance values are derived from the data points used to display the graphs in Figure 5-3.

| Parameter | Analog | Digital | Theoretical |
|---|---|---|---|
| Peak Gain (dB) | 8.51 | 8.59 | 9.13 |
| Bandwidth @ -3dB (Hz) | 152.46 | 326.15 | 77.35 |
| Gain @ 1000Hz (DB) | -25.19 | -15.27 | -30.68 |

Analyzing this data leads to the following conclusions,

1. The theoretical response is not an accurate indicator of actual digital controller performance. The peak gain of the theoretical model is approximately equal to that of the actual digital controller but the bandwidth of the digital controller is four times greater then that of the theoretical model.
2. The digital controller has a bandwidth which is over twice that of the analog controller though their peak gains are approximately equal.
3. At frequencies beyond the bandwidth of the controller, the slope of the digital controller is less then that of the analog controller.

As with the axial bearing, the bandwidth predicted by the theoretical model is not an accurate indicator of the actual digital controller bandwidth. The theoretical model however does seem to accurately predict the peak gain of the digital controller but this is true only for the lower bearing (axes 2X and 2Y). In the case of the upper bearing (axes 1X and 1Y), the peak gain predicted by the theoretical model is significantly higher then the actual digital controller response. The digital controller has significantly better bandwidth then the analog controller but the slope of the digital controller is significantly less then that of the analog controller in the high frequency regions. This high frequency region is where the major bending modes of the rotor are located which could cause stability problems when the rotor is spinning.




**5-4** Radial Bearing 2X Static Disturbance Rejection Response

Figure 5-4 displays the disturbance rejection response of radial bearing 2X under digital and analog control as well as the predicted theoretical response. The following table shows the

important performance values used as design criteria for the digital controller. These performance values are derived from the data points used to display the graphs in Figure 5-4.

| Parameter | Analog | Digital | Theoretical |
|---|---|---|---|
| Peak Compliance (Microns/Newton) | 3.30 | 2.47 | 4.47 |
| Compliance @ 1000Hz (Microns/Newton) | 0.027 | 0.051 | 0.003 |

Analyzing this data leads to the following conclusions,

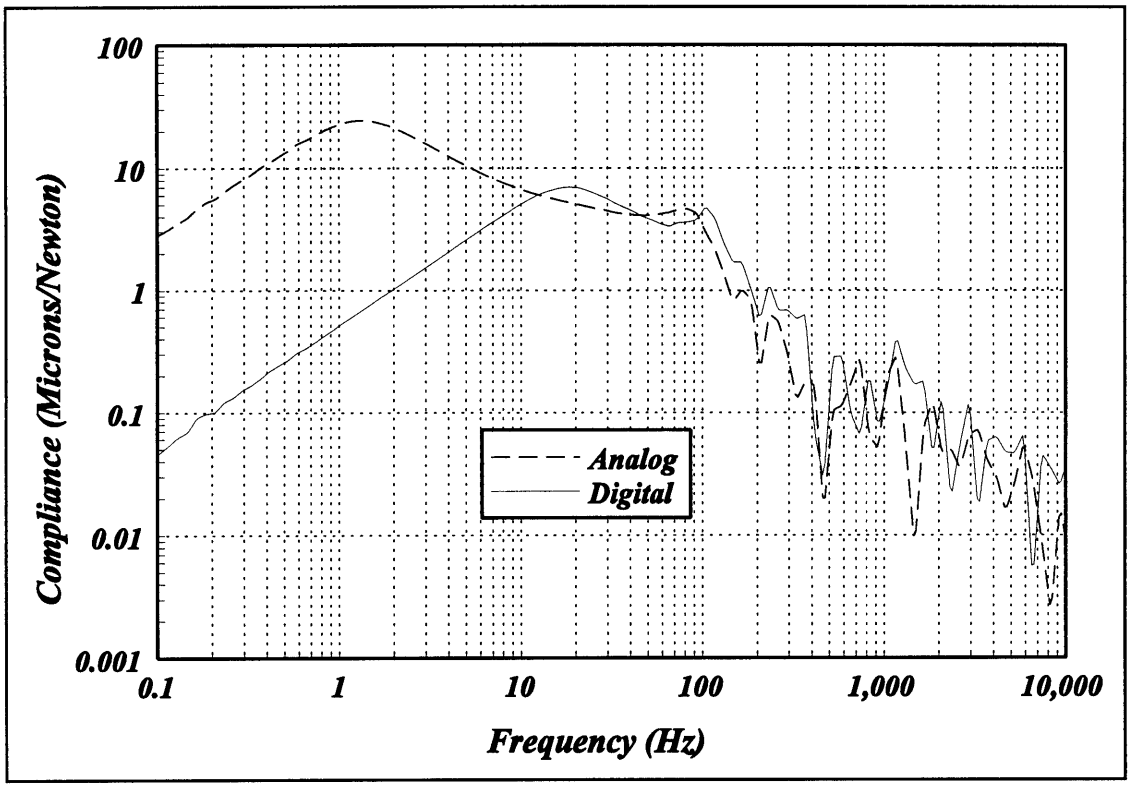
1. The theoretical response accurately predicts actual digital controller response at and below 5 Hz. The theoretical response becomes progressively less accurate as frequency increases. The difference between the theoretical maximum compliance and the actual digital controller maximum compliance is significant.
2. The digital controller has significantly better disturbance rejection properties at and below 10 Hz. As with the axial bearing, the fact that the analog controller has significantly worse disturbance rejection properties in this region and yet produces very stable operation leads one to believe that the low frequency disturbance rejection is of low importance as a controller design criteria for the radial bearings.
3. At frequencies beyond 100 Hz, the disturbance rejection properties of both the analog and digital controllers are approximately equal.

Whereas the theoretical model is a good predictor of actual digital controller response up to 5 Hz for the lower bearing, that characteristic frequency is 10 Hz for the upper bearing. However the difference between the predicted maximum compliance and the actual maximum compliance is significantly greater for the upper bearing then for the lower bearing. The disparity between maximum compliance of the theoretical and digital controllers was not a characteristic of the axial bearings. This points toward bearing coupling as the most likely cause of this increased stiffness.

## 5.3 Dynamic Test Results

This section displays the results of the disturbance rejection response of the axial bearing and one axis of a particular radial bearing when the rotor is spinning at 15000 and 28000 RPM. The closed loop frequency response is absent from the dynamic tests because the command following response of the system is of little importance in the case where the rotor is spinning. The 2X radial bearing was chosen to represent the response of a typical radial bearing axis because this axis was the most likely to fail during testing and therefore represents the worst case response. Both the analog and digital responses were determined using an HP System Analyzer.

### 5.3.1 Axial Bearing Test Results



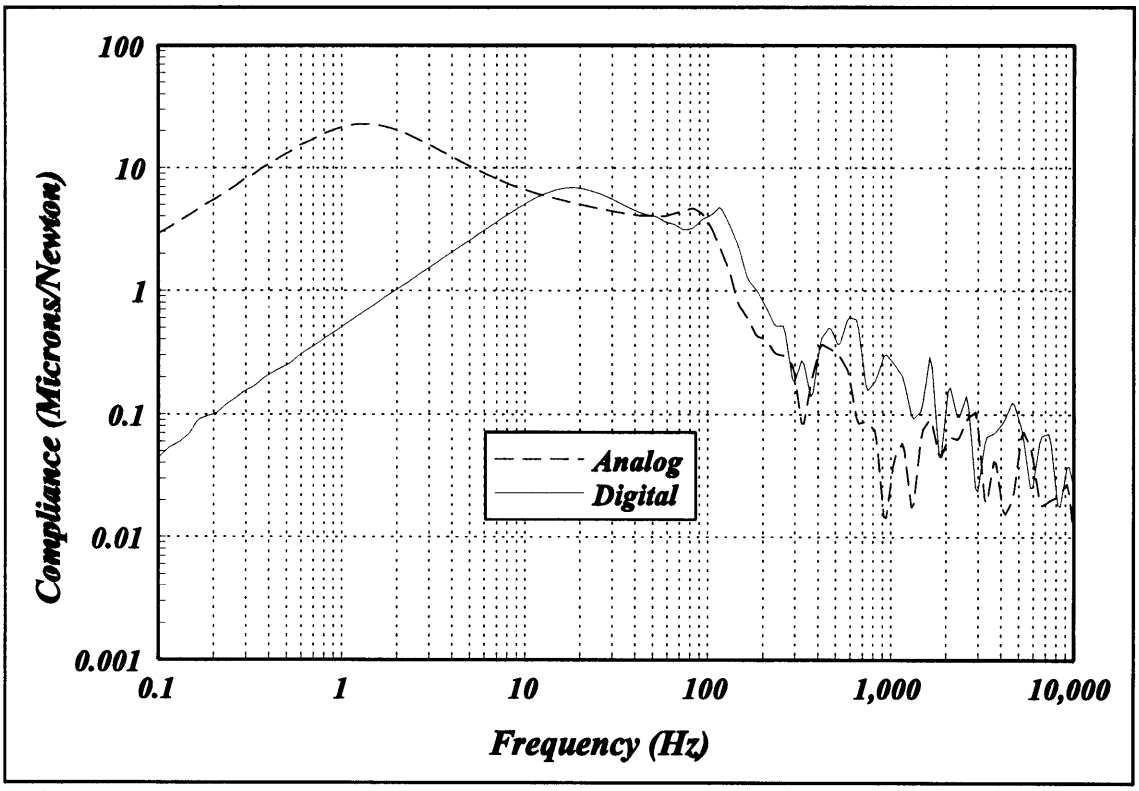

**5-5** Axial Bearing Dynamic Disturbance Rejection Response - 15000 RPM

Figure 5-5 displays the disturbance rejection response of the axial bearing under digital and analog control while the rotor is spinning at 15000 RPM. The following table shows the important performance values used as design criteria for the digital controller. These performance values are derived from the data points used to display the graphs in Figure 5-5.

| Parameter | Analog | Digital |
|---|---|---|
| Peak Compliance (Microns/Newton) | 24.45 | 7.00 |
| Compliance @ 1000Hz (Microns/Newton) | 0.121 | 0.121 |

Analyzing this data leads to the following conclusions,

1. The digital controller has significantly better disturbance rejection properties at and below 10 Hz.
2. For the majority of the frequencies beyond 10 Hz, the disturbance rejection properties of the digital controller is somewhat worse then that of the analog controller.



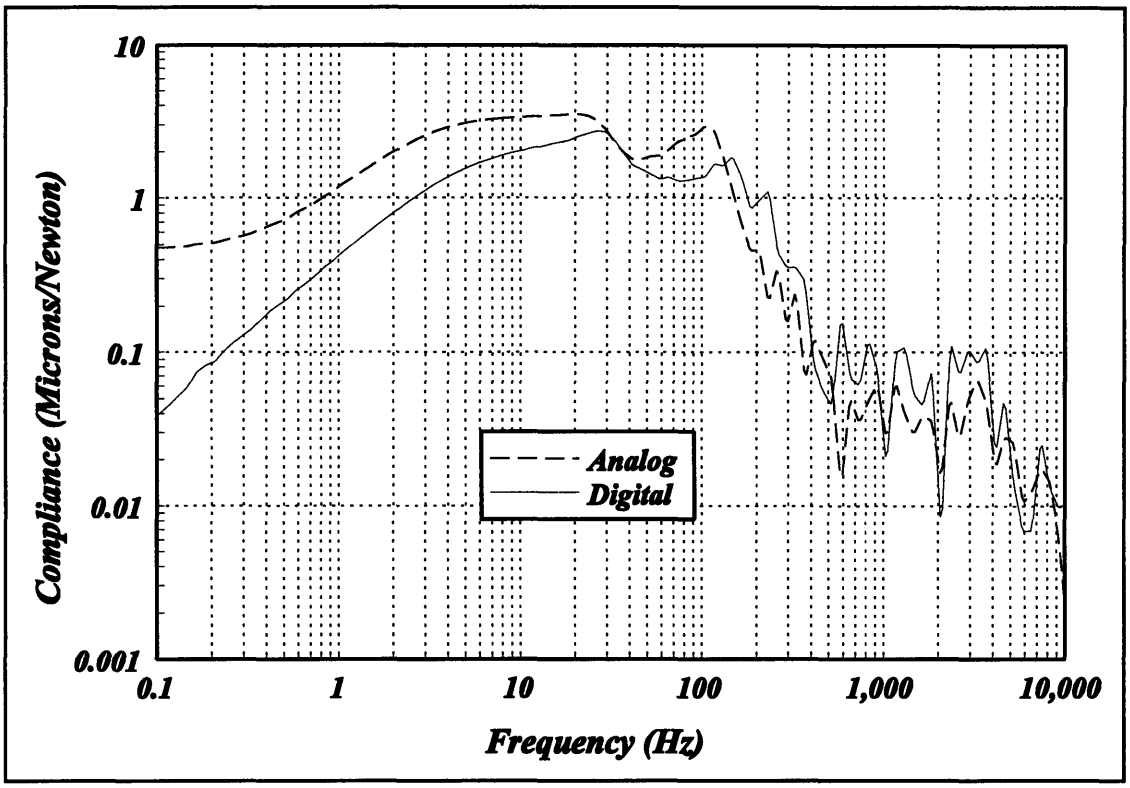

**5-6** Axial Bearing Dynamic Disturbance Rejection Response - 28000 RPM

Figure 5-6 displays the disturbance rejection response of the axial bearing under digital and analog control while the rotor is spinning at 28000 RPM. The following table shows the important performance values used as design criteria for the digital controller. These performance values are derived from the data points used to display the graphs in Figure 5-6.

| Parameter | Analog | Digital |
|---|---|---|
| Peak Compliance (Microns/Newton) | 22.73 | 6.94 |
| Compliance @ 1000Hz (Microns/Newton) | 0.027 | 0.277 |

Analyzing this data leads to the same conclusions as in the previous case. The disturbance rejection performance of the axial bearing is virtually the same for each controller regardless of the rotor speed as one would expect. The digital controller has superior performance at lower frequencies but is slightly worse at higher frequencies. After 400 Hz, the magnitudes of the compliance are sufficiently small such that any difference between the analog and digital controllers is negligible. There is a uniform increase in compliance between the static case and the dynamic cases. This is probably caused by the axial forces generated by the rotating vanes of the turbopump.

### 5.3.2 Radial Bearing Test Results




**5-7** Radial Bearing 2X Dynamic Disturbance Rejection Response - 15000 RPM

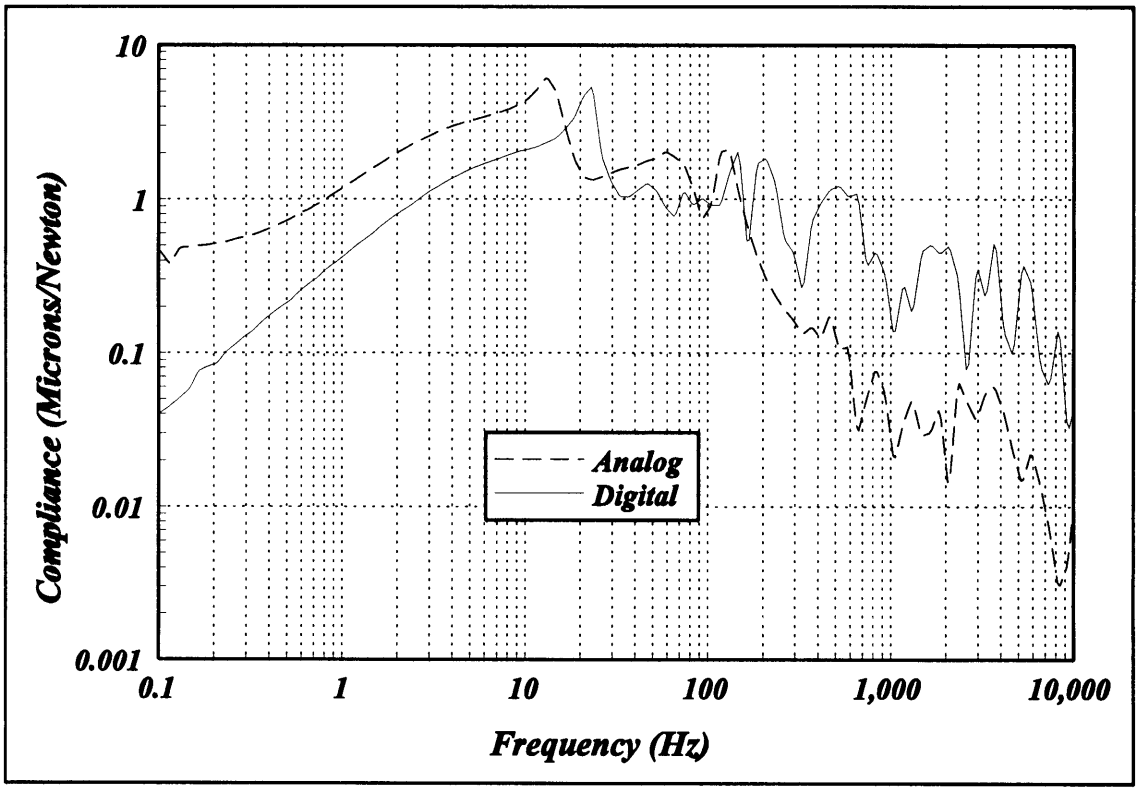
Figure 5-7 displays the disturbance rejection response of radial bearing 2X under digital and analog control while the rotor is spinning at 15000 RPM. The following table shows the important performance values used as design criteria for the digital controller. These performance values are derived from the data points used to display the graphs in Figure 5-7.

| Parameter | Analog | Digital |
|---|---|---|
| Peak Compliance (Microns/Newton) | 3.51 | 2.72 |
| Compliance @ 1000Hz (Microns/Newton) | 0.037 | 0.030 |

Analyzing this data leads to the following conclusions,

1. The digital controller has significantly better disturbance rejection properties at and below 10 Hz.
2. Between 10 and 100 Hz, the digital controller displays disturbance rejection properties that are either better then or equal to that of the analog controller.
3. For the majority of the frequencies beyond 100 Hz, the disturbance rejection properties of the digital controller is somewhat worse then that of the analog controller.

The digital controller has slightly worse disturbance rejection properties then the analog controller above 100 Hz but above 300 Hz, the magnitudes of the compliance are sufficiently small such that any difference between the two controllers is negligible. Overall, at this particular speed, the digital controller is a significantly better controller then the analog one.




**5-8** Radial Bearing 2X Dynamic Disturbance Rejection Response - 28000 RPM

Figure 5-8 displays the disturbance rejection response of radial bearing 2X under digital and analog control while the rotor is spinning at 28000 RPM. The following table shows the important performance values used as design criteria for the digital controller. These performance values are derived from the data points used to display the graphs in Figure 5-8.

| Parameter | Analog | Digital |
|---|---|---|
| Peak Compliance (Microns/Newton) | 6.13 | 5.36 |
| Compliance @ 1000Hz (Microns/Newton) | 0.028 | 0.175 |

Analyzing this data leads to the following conclusions,

1. The digital controller has significantly better disturbance rejection properties at and below 10 Hz.
2. Between 10 and 100 Hz, the digital controller displays disturbance rejection properties that are either better then or equal to that of the analog controller.
3 For the majority of the frequencies beyond 100 Hz, the disturbance rejection properties of the analog controller are on average approximately 2.2 times better then that of the digital controller.

The inability of the digital controller to exhibit better stiffness in the high frequency region would lead to stability problems at this rotor speed. The spike exhibited by both the analog and digital controllers in the 11 Hz region is due to coupling between the upper and lower bearing. Under normal operation, the position signals of the upper and lower bearings are sinusoidal waves having different amplitudes and frequencies. However at the frequency of the spike, the position signals of both the upper and lower bearings had the same amplitude and the same frequency. This spike is a characteristic exhibited by all the bearings at this speed. Though it is not apparent from Figure 5-8, the digital controller is on the edge of instability at this speed.

## 5.4 Summary and Remarks

During the digital controller design stage, it became apparent that the theoretical model was much worse then expected. The theoretical model returned values of the feedback gain that were so high that they would never yield a stable system. The inaccuracy of the theoretical model was verified by the data from the static closed loop frequency response of the radial and axial bearings. The use of a very simple model to represent the response of the open loop system accounts for the large discrepancies between the theoretical and actual system response.

The theoretical model was much more accurate at predicting the actual disturbance rejection response of the digital controller. However the theoretical model grew less accurate as frequency increased. In the case of the radial bearings, the theoretical model in some ways predicted worst response then the actual digital controller exhibited. During static testing and when the rotor was spinning at 15000 RPM, the shapes of the analog and digital controllers are similar though their magnitudes vary somewhat. The digital controller exhibit significantly

greater compliance in the high frequency region when rotor speed was increased to 28000 RPM. This high frequency region is where the major bending modes lie and therefore system stability was compromised in the case of the digital controller. The shape of the disturbance rejection response remained relatively constant for the axial bearing during static and dynamic testing. However the digital controller did show a uniform increase in compliance during dynamic testing probably due to axial forces generated by the rotating vanes of the turbopump.

# Chapter 6

# Conclusions and Recommendations

---

The goal of this researcher's work was to model, design, and implement a digital controller which used the Time Delay Control (TDC) algorithm to control the five axes of a magnetically levitated turbopump. The final success or failure of the resulting digital controller is directly related to how well each of these three tasks were performed. Therefore, the success of each stage of the process with be analyzed in the light of the final digital controller performance.

The modeling process is based upon the proper determination of the theoretical open loop transfer function which imitates the characteristics of the actual open loop transfer function. The actual open loop transfer function was determined for each axis using a system analyzer while the turbopump was under analog control. At first, the actual transfer function was compared with the theoretical transfer function derived from the physics governing the operation of each axis of a magnetic bearing. Superior results were obtained by recursively testing various values in the numerator and denominator of a transfer function having the same form as the transfer function derived from the physics of the magnetic bearing. The response of each guess was compared against the actual system response using a least squares error estimation to determine the best fit transfer function. This yielded consistently better results across all bearings. The recursive analysis was limited to frequencies below 1000 Hz because the high frequency dynamics of the system tended to make the best fit transfer function less accurate within the probable bandwidth of the controller. This same recursive analysis was applied to the analog controller driver to produce the final open loop transfer function.

The modeling process would proved to be flawed. For both the axial and radial bearings, the theoretical analysis predicted stable systems which were in fact unstable when applied to the actual system. The closed loop frequency response of the axial bearing was significantly different from the actual system response. The radial bearing closed loop frequency response of the theoretical analysis was expected to differ from the actual system response because coupling was ignored in the theoretical model. However the radial bearing theoretical frequency response was far worse then expected. This researcher theorizes that the poor performance of the theoretical model is in part due to the determination of the actual open loop system frequency response. The inherent characteristics of magnetic bearings ensures that the system is open loop unstable. This requires that the system be under active control during testing. With such a system, the equilibrium position is determined by the controller and may vary from controller to controller. The system analyzer will in turn linearize the response about this equilibrium point.

In a highly nonlinear system, the characteristics at different equilibrium points may be significantly different and therefore their linearized responses may also vary significantly. Thus the open loop frequency response could be controller dependent.

The next task was the design of the digital controller. The problem was that no one had attempted or at least documented a rigorous design procedure for TDC. The primary reason for simplifying the radial equations of motion was so that classical design techniques could be employed. Classical design techniques would allow a design procedure to be tested and evaluated. It would have been extremely difficult to use multi-input multi-output design techniques using a control algorithm in which single input single output design techniques amounted to merely trial and error. Of particular importance was evaluation of the proper sampling rate. After analyzing how system stability was effected by changes in the parameters of the controller, this researcher decided to use statistics to deduce the proper sampling rate. However, this technique required that the theoretical model could accurately determine whether the system was stable and its bandwidth. The theoretical model has since proved significantly inaccurate and therefore compromises the results obtained from this analysis. It does not however compromise the validity of the design technique in general.

It also became apparent that the drawbacks of using TDC outweighed its benefits for this particular application. TDC uses information from previous sampling interval(s) to estimate unmodeled system characteristics. For this particular system, these unmodeled characteristics are coupling and gyroscopic effects. However, the most important parameter effecting TDC's performance and stability is the sampling rate. If TDC is represented as a PID controller, the sampling rate also determines the integrator value. Ideally, since the characteristics of each magnetic bearing axis can differ significantly, each bearing axis should be controlled by a controller with a sampling rate tailored to needs of each axis. This would not be feasible using one DSP chip and it would be economically unfeasible to use more DSPs. Therefore the integration value was constant across all axes which led to a less flexible design process. This also meant that the controller should be optimized to provide the best performance on the bearing axis that was most likely to fail first. However, it was impossible to determine the bearing most likely to fail first with any certainty until a working controller was in place. One sampling rate for all bearing axes also meant that creating an optimal controller for this application was very unlikely.

The final task was implementing the digital controller. This also was not problem free. One of the state variables required by the digital controller was the velocity of the rotor at each sampling interval. The turbopump did not have a velocity sensor however so the position was differentiated to provide the current velocity. The TDC law required differentiation of the state variables which meant that the acceleration was calculated from the second derivative of the position signal. Differentiation by its very nature amplifies the noise inherent in the signal. Therefore, it is very likely that noise component of the position signal overwhelmed the signal component in the calculated acceleration value. The I/O Interface board also produced high amplitude spikes when crossing zero and the anti-aliasing filters exhibited large overshoots both of which added high frequency noise to the control signal. Steps were taken to remove this high frequency noise through an auxiliary low pass filter having a cutoff frequency of approximately 15000 Hz. However, it was the inability of the controller to reduce compliance in the region between 1000 Hz and 10000 Hz which eventually limited the rotor speed to 28000 RPM. This

may have been exasperated by noise produced by the I/O Interface board.

This researcher's recommendations consists of two parts. The first part concerns recommendations pertaining to the turbopump application. The second part concerns recommendations pertaining to the TDC law.

Further research was be performed in an effort to enhance the accuracy of the theoretical model. The use of a very simple model provided inaccurate theoretical responses which controller design difficult. Different methods for incorporating bending modes and gyroscopic effects into the theoretical model must be investigated and verified against the actual system response. Until a more accurate model is developed, all further controller design will have to rely on trail and error to a very large extent.

As far as TDC law is concerned, there is only one recommendation. More basic research should be performed to find a procedure for designing Time Delay controllers with particular emphasis on determination of the proper sampling rate. With other digital control algorithms, the sampling rate is determined by the Nyquist frequency and the anti-aliasing filter cutoff frequency. This usually leaves the designer with plenty of choices with no penalty for sampling at a greater then necessary rate. However with TDC, the sampling rate also effects system performance and therefore sampling rate determination takes on an added importance. This importance may also vary with the type of application and the number of simplifications imposed to the TDC law. This particular application is a very challenging one and this Time Delay controller performed as well as or better then other digital controllers designed for this same application.

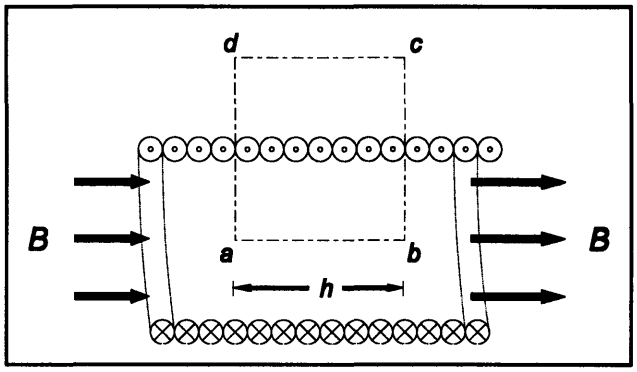
# Appendix A

# Magnetic Circuit Analysis

## A.1 Ampere's Law

The discovery that currents produce magnetic effects was made by Hans Christian Oersted in 1820 [18]. Today the quantitative relationship between current $I$ and the magnetic field $\vec{B}$ is written as Ampere's Law,

$$\oint \vec{B} \cdot d\vec{l} = \mu_0 i \qquad \text{(A.1)}$$



A-1 Ideal Solenoid

Now Ampere's Law is applied to an ideal solenoid. An ideal solenoid is one in which its length is much greater than its diameter. The field outside the coil is assumed to be essentially zero and that the field inside the coil is assumed to be essentially uniform. Applying Ampere's Law to the rectangular path *abcd* in Figure A-1 yields,

$$\oint \vec{B} \cdot d\vec{l} = \int_a^b \vec{B} \cdot d\vec{l} + \int_b^c \vec{B} \cdot d\vec{l} + \int_c^d \vec{B} \cdot d\vec{l} + \int_d^a \vec{B} \cdot d\vec{l} \qquad \text{(A.2)}$$

The first integral on the right is $Bh$. The second and fourth integrals are zero because $\vec{B} \cdot d\vec{l} = 0$ $\left(\vec{B} \perp d\vec{l}\right)$. The third integral is zero because by definition, the magnetic field is zero for all points outside the coil of an ideal solenoid. Thus,

$$\oint \vec{B} \cdot d\vec{l} = Bh = \mu_0 iN \qquad \text{(A.3)}$$

This relation holds quite well for actual solenoids at interior points near the center of the solenoid. It shows that $\vec{B}$ is independent of the diameter and length of the solenoid and that $\vec{B}$

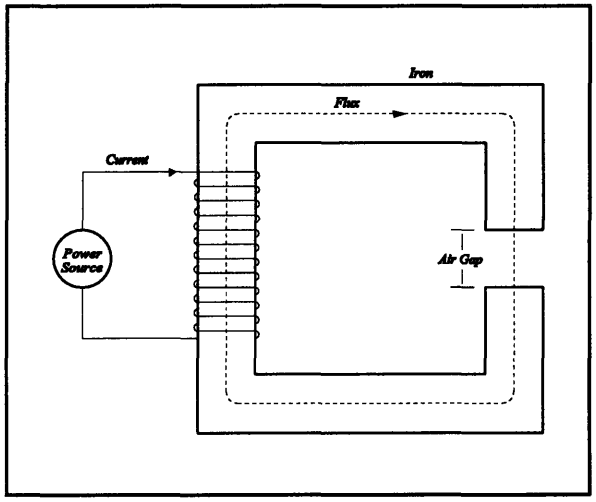
is constant over the solenoid cross-sectional area.

The term magnetic flux $\vec{\Phi}$ is now introduced. Flux is a property of all vector fields and is defined for both open and closed surfaces as,

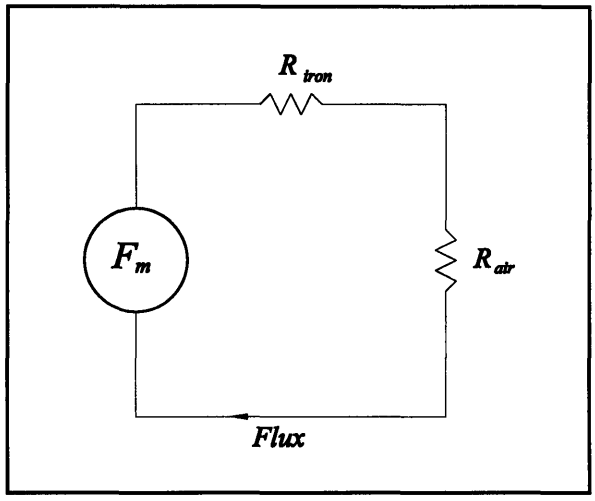
$$d\vec{\Phi} = \vec{B} \cdot d\vec{A} \tag{A.4}$$

# A.2 The Magnetic Circuit

In problems involving magnetic devices, the engineer is concerned with the quantities of magnetic flux $\vec{\Phi}$ and magnetic field $\vec{B}$ that occupy three-dimensional space. Fortunately in most instances the only space of interest is occupied by ferromagnetic materials except for small air gaps. These ferromagnetic materials with their high permeabilities effectively confine the magnetic flux to themselves just as copper wires confine the electrical current or as pipes confine a fluid. The net effect on this confinement is that our three dimensional field problem becomes essentially a one dimensional circuit problem [7]. Consider the magnetic circuit shown below,




**A-2** Typical Magnetic Circuit




**A-3** Magnetic Circuit Diagram

The magnetomotive force of the coil produces flux which is confined to the iron and to part of the air in the air gap having the same cross-sectional area as the iron. This is analogous to a magnetomotive force source driving flux through two series-connected reluctances (reluctance is to flux as resistance is to electrical current). This analogy between the magnetic circuit and the electrical circuit carries through in many other respects. This magnetic circuit analogy will be used to model the magnetic bearings of our turbopump.

Before the magnetic circuit model of the turbopump's magnetic bearings is analyzed, the magnetic circuit terms analogous to their electrical circuit counterparts need to be established. The electrical terms of most importance are voltage source, current, and resistance. In magnetic

circuits, voltage is analogous to magnetomotive force which is defined from Ampere's Law,

$$F_m = \oint \frac{\vec{B}}{\mu} \cdot d\vec{l} \tag{A.5}$$

the magnetic field is divided by the material permeability to allow the magnetomotive force to be independent of the material conducting the magnetic field. Ampere's Law for a solenoid has been previously derived therefore the magnetomotive force of a solenoid is.

$$F_m = \oint \frac{\vec{B}}{\mu} \cdot d\vec{l} = \frac{\mu IN}{\mu} = IN \tag{A.6}$$

The magnetic circuit equivalent of electrical current is referred to as magnetic flux which was previously defined as

$$\Phi = \int_S \vec{B} \cdot d\vec{A} \tag{A.7}$$

The magnetic circuit equivalent of electrical resistance referred to as reluctance now needs attention. From Ohm's Law

$$R = \frac{V}{i} \tag{A.8}$$

Where: R = electrical resistance
V = voltage drop across resistance
I = electrical current through resistance

For an analogous magnetic circuit,

$$R_m = \frac{F_m}{\Phi} \tag{A.9}$$

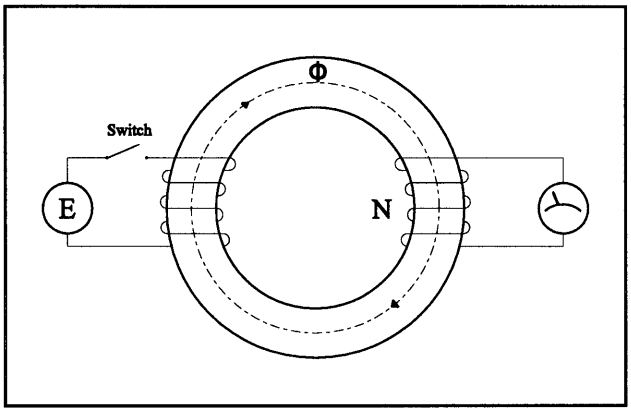
Where: $R_m$ = magnetic reluctance
$F_m$ = magnetomotive force
$\Phi$ = magnetic flux

To derive reluctance, a magnetic circuit is assumed to exist having a constant magnetic flux density $B$ over a known constant area $A$ and that the circuit has a total path length $L$

$$R = \frac{F_m}{\Phi} = \frac{\frac{BL}{\mu}}{BA} = \frac{L}{\mu A} \tag{A.10}$$

Thus the impediment to flow of magnetic flux is directly proportional to path length and inversely proportional to the material cross-sectional area and permeability.

Finally a formula for the amount of energy stored in the magnetic field of a magnetic circuit is derived. This in turn requires the introduction of Faraday's Law. Faraday, using the apparatus in Figure A-4, conducted experiments into the relationship between electricity and magnetism. The formula bearing his name that describes this relationship as it relates to a solenoid of $N$ turns is




**A-4** Faraday's Experimental Apparatus

$$e = -N\frac{d\Phi}{dt} \tag{A.11}$$

$e$ is the electromotive force or voltage that resists the change in flux. The power used by the electrical coils of the solenoid to produce the magnetic field is defined as

$$P = iv \tag{A.12}$$

The energy stored in the magnetic field must then be

$$E = \int iv\, dt \tag{A.13}$$

Using Faraday's Law and changing the sign to denote the EMF transferred as opposed to resisting the change

$$E = \int NI\frac{d\Phi}{dt}\, dt \tag{A.14}$$

Magnetic flux $\Phi$ was previously defined as

$$\Phi = \int \vec{B}\cdot d\vec{A} \tag{A.15}$$

Since the cross-sectional area of our magnetic bearing is essentially constant and at all times parallel to the magnetic flux,

$$\Phi = BA \quad \Rightarrow \quad d\Phi = A\,dB \tag{A.16}$$

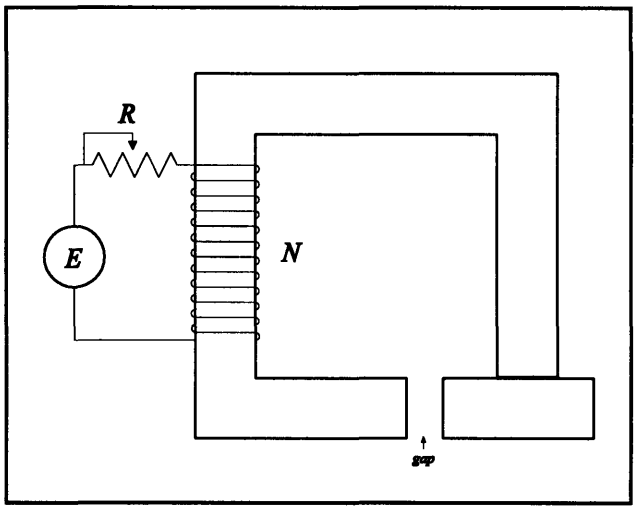
Remember, the magnetic field of a solenoid was previously defined as

$$B = \frac{\mu IN}{l} \quad \Rightarrow \quad IN = \frac{Bl}{\mu} \tag{A.17}$$

Substituting Eq. (A.16) and Eq. (A.17) into Eq. (A.14) yields

$$E = \int \frac{BlA}{\mu}\, dB \quad \Rightarrow \quad E = \frac{1}{2}\frac{lA}{\mu}B^2 \tag{A.18}$$

Now a formula that defines the energy stored in a magnetic field of a magnetic circuit by a solenoid present in the same circuit has been derived. It is now possible to show that mechanical work can be done by extracting energy stored in the magnetic field. To describe how this is feasible, consider the sample magnetic circuit in Figure A-5. This circuit is comprised of an exciting coil placed on a ferromagnetic core equipped with a movable element called a relay armature. Currently the exciting coil is energized from a constant voltage source and the gap is at its equilibrium position. Suppose $R$ is adjusted such that the magnetic flux induced in the circuit increases while the relay armature is held fast. The net effect of the additional energy supplied by the source is an increase in the stored magnetic energy. Now suppose that $R$ was adjusted as previously described but this time the relay armature is allowed to move. In this instance the armature does move implying that some of the magnetic energy stored in the field has been converted to mechanical work. The nature of this conversion of magnetical energy to mechanical work manifests itself in a change in the air gap and therefore a change in the reluctance of the magnetic circuit.



**A-5** Magnetic Energy to Mechanical Work Sample Circuit

Before continuing, the equation describing the energy stored in a magnetic circuit will be defined using terms representative of our magnetic circuit analogy. Previously, stored magnetic energy was defined as

$$E = \frac{1}{2}\frac{lA}{\mu}B^2 \tag{A.19}$$

Recall that for a magnetic circuit, magnetomotive force is defined as,

$$F_m = IN = \frac{Bl}{\mu} \tag{A.20}$$

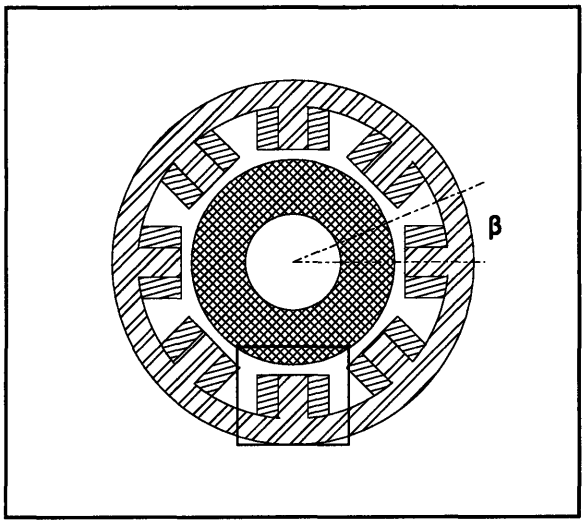
Also, magnetic flux was defined as,

$$\Phi = BA \tag{A.21}$$

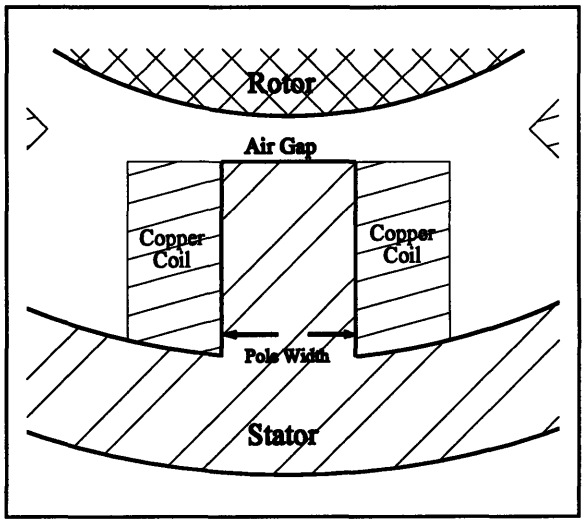
Hence

$$E = \frac{1}{2}F_m\Phi \tag{A.22}$$

# A.3 Radial Bearing Magnetic Circuit

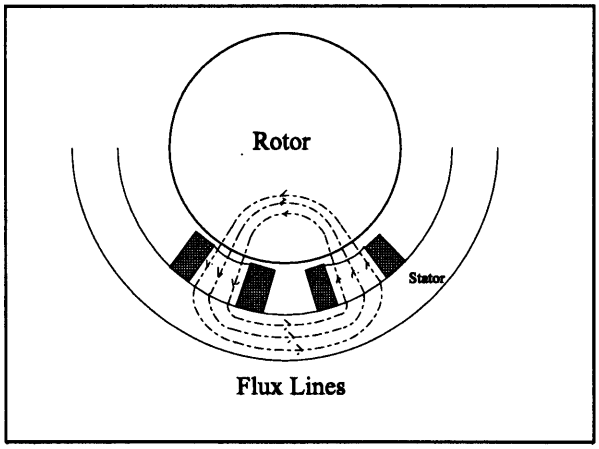



**A-6** Turbopump Radial Bearing

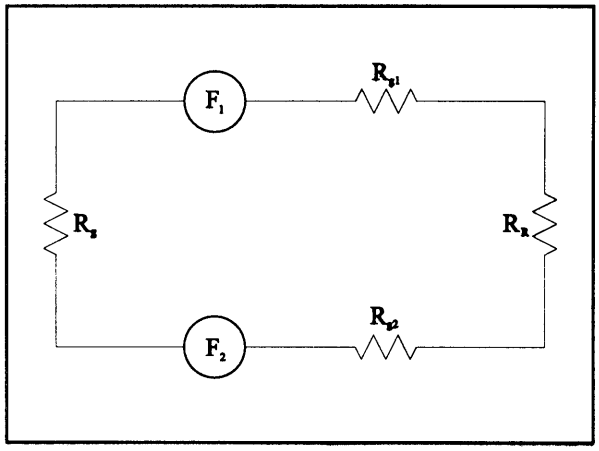



**A-7** Radial Bearing Magnetic Pole

The physical structure of the magnetic radial bearings used by this particular turbopump is shown in Figures A-6 and A-7. The radial bearings consist of a ring of laminated ferromagnetic material having 8 poles. Each pole is wound with $N$ turns of wire. The lines of magnetic flux generated from this bearing structure is represented by the Figure A-8. The equivalent magnetic circuit of our magnetic bearing is represented in Figure A-9.




**A-8** Radial Bearing Flux Lines




**A-9** Radial Bearing Magnetic Circuit

Assuming that the rotor of the turbopump is equidistant from both poles at any given time and that both poles can be modeled as solenoids having an equal number of turns,

$$F_1 = F_2 \quad \text{and} \quad R_{g1} = R_{g2} \tag{A.23}$$

The force exerted by the bearings on the rotor shaft is defined as

$$F = \frac{dE}{dx} = \frac{d}{dx}\left(\frac{1}{2}F_m\Phi\right)$$

$$\frac{dE}{dx} = \frac{dE}{d\Phi}\frac{d\Phi}{dx} = \frac{1}{2}F_m\frac{d\Phi}{dx} \tag{A.24}$$

Previously the magnetic flux was defined as

$$\Phi = \frac{F_m}{R_m} \tag{A.25}$$

Therefore,

$$F = \frac{1}{2}F_m^2\frac{d}{dx}\left(\frac{1}{R_m^T}\right) \tag{A.26}$$

where $R_m^T$ defines the total reluctance of the magnetic circuit. The magnetomotive force is supplied by two coils in series each having $N$ turns therefore,

$$F_m = F_1 + F_2 = 2NI \tag{A.27}$$

hence

$$F = 2N^2I^2\frac{d}{dx}\left(\frac{1}{R_m^T}\right) \tag{A.28}$$

The combined reluctance of the circuit is comprised of the reluctance of the bearing stator, the two air gaps, and the rotor shaft in series.

$$R_m^T = R_S + R_{g1} + R_{g2} + R_R \tag{A.29}$$

Recall that reluctance was defined as the following

$$R_m = \frac{L}{\mu A} \tag{A.30}$$

Therefore

$$R_m^T = \frac{L_S}{\mu_S A} + \frac{2x}{\mu_0 A} + \frac{L_R}{\mu_R A} \tag{A.31}$$

Where: $L_S$ = stator flux path length
$\mu_S$ = stator material permeability
x = air gap
$\mu_0$ = air permeability
$L_R$ = rotor flux path length
$\mu_R$ = rotor material permeability

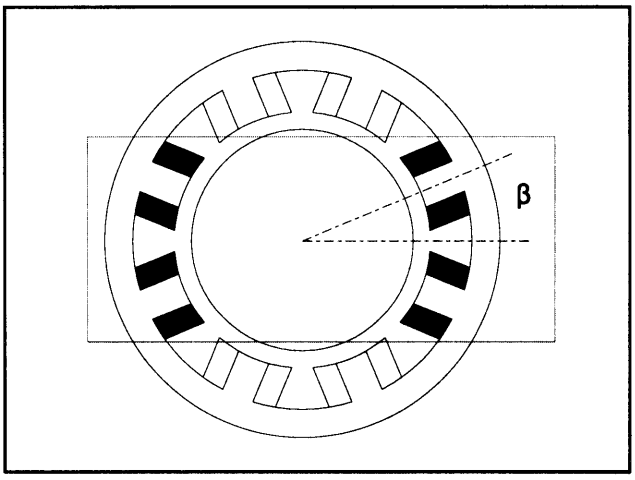
Since the stator and the rotor shaft are composed of ferromagnetic materials, their reluctances are considerably smaller than that of the air gaps. This is primarily due to the great difference in the permeabilities of ferromagnetic materials (between 1000 and 10000) and air ($4\pi \times 10^{-7}$). Thus the reluctance of the circuit simplifies to

$$R_m^T \simeq \frac{2x}{\mu_0 A} \quad \text{or} \quad \frac{1}{R_m^T} \simeq \frac{\mu_0 A}{2x} \tag{A.32}$$

Substituting Eq. (A.32) into Eq. (A.28) yields,

$$F = 2N^2I^2\frac{d}{dx}\left(\frac{\mu_0 A}{2x}\right) = -\frac{\mu_0 N^2 I^2 A}{x^2} \tag{A.33}$$

The sign denotes the that the force exerted by the bearings on the rotor is an attractive force. The radial bearing is physically comprised of one ring having 8 poles. However, the radial bearing is viewed as being comprised of two bearing axes composed of two opposing pole pairs - two poles adjacent to one another with a mirror image on the opposing side. The calculated attractive force $F$ represents the contributions of a pole pair on one side of the bearing axis. The geometry of these adjacent poles is such that the attractive force is not applied along the centerline of the pole pair but at an angle. Taking into account pole geometry, the equation for the force exerted by a virtual bearing on the rotor is defined as,

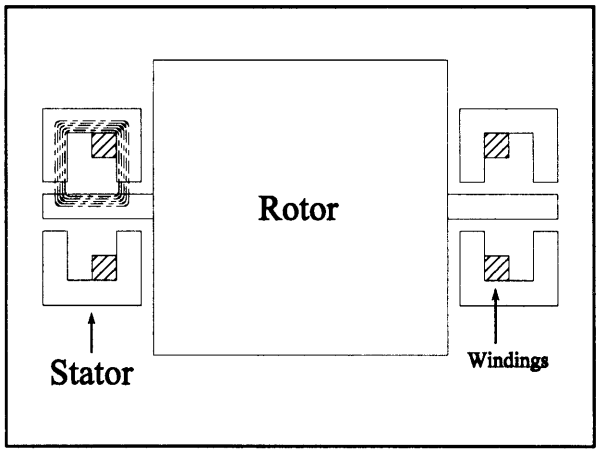



**A-10** Radial Bearing Pole Composition

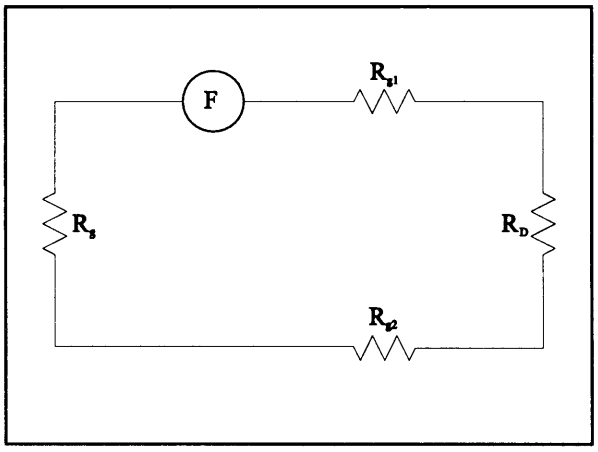
$$F = \frac{\mu_0 N^2 I_r^2 A \cos\beta}{x_r^2} - \frac{\mu_0 N^2 I_l^2 A \cos\beta}{x_l^2} \qquad \textbf{(A.34)}$$

where the subscripts $r$ and $l$ denote right and left respectively and $\beta$ is the angle of the pole pair centerline to the pole centerline. In this instance, $\beta = 22.5°$.

# A.4 Axial Bearing Magnetic Circuit




**A-11** Axial Bearing Configuration




**A-12** Axial Bearing Magnetic Circuit

The physical structure on our particular turbopump magnetic axial bearings is shown in

Figure A-11. The axial bearings consist of an upper and lower ring of ferromagnetic material having a U-shaped cross-section. The inner leg of this U is wound with $N$ turns of wire. The lines of magnetic flux generated by the windings flow along the U and into a disk that is press fit to the rotor shaft. The equivalent magnetic circuit of our magnetic bearing is represented in Figure A-12.

As with the radial bearing , the rotor disk is assumed to be equidistant from both poles at any given time and therefore,

$$R_{g1} = R_{g2} \tag{A.35}$$

The force exerted by the bearings on the rotor disk is defined as

$$F = \frac{1}{2} F_m^2 \frac{d}{dx}\left(\frac{1}{R_m^T}\right) \tag{A.36}$$

where $R_m^T$ defines the total reluctance of the magnetic circuit. The magnetomotive force is supplied by one coil having $N$ turns therefore,

$$F_m = NI \tag{A.37}$$

hence

$$F = \frac{N^2 I^2}{2} \frac{d}{dx}\left(\frac{1}{R_m^T}\right) \tag{A.38}$$

The combined reluctance of the circuit is comprised of the reluctance of the bearing stator, the two air gaps, and the rotor disk in series.

$$R_m^T = R_S + R_{g1} + R_{g2} + R_d \tag{A.39}$$

Therefore

$$R_m^T = \frac{L_S}{\mu_S A} + \frac{2x}{\mu_0 A} + \frac{L_d}{\mu_d A} \tag{A.40}$$

Where: $L_S$ = stator flux path length
$\mu_S$ = stator material permeability
x = air gap
$\mu_0$ = air permeability
$L_d$ = rotor disk flux path length
$\mu_d$ = rotor disk material permeability

Since the stator and the rotor disk are composed of ferromagnetic materials, their reluctances are considerably smaller than that of the air gaps. This is primarily due to the great difference in the permeabilities of ferromagnetic materials (between 1000 and 10000) and air ($4\pi \times 10^{-7}$). Thus the reluctance of the circuit simplifies to

$$R_m^T \simeq \frac{2x}{\mu_0 A} \quad \text{or} \quad \frac{1}{R_m^T} \simeq \frac{\mu_0 A}{2x} \tag{A.41}$$

Substituting Eq. (A.41) into Eq. (A.38) yields

$$F = \frac{N^2 I^2}{2} \frac{d}{dx}\left(\frac{\mu_0 A}{2x}\right) = -\frac{\mu_0 N^2 I^2 A}{4x^2} \tag{A.42}$$

The sign denotes the that the force exerted by the bearings on the rotor is an attractive force. Note that due to the shape of the axial bearing, the area of the inner pole is less than that of the outer pole. Therefore the area $A$ is the average area of both the inner and outer poles. As with the radial bearings, the attractive force $F$ represents the contributions of both legs on one side of the rotor disk. The total force exerted by both the upper and lower legs on the rotor disk is,

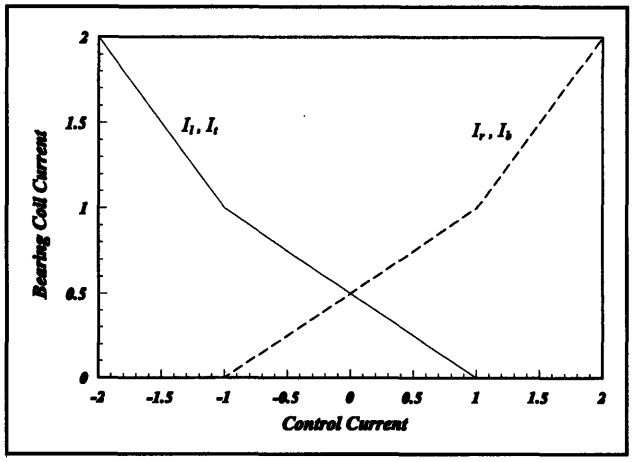
$$F = \frac{\mu_0 N^2 I_t^2 A}{4x_t^2} - \frac{\mu_0 N^2 I_b^2 A}{4x_b^2} \tag{A.43}$$

where the subscripts $t$ and $b$ denote top and bottom respectively.

## A.5 Magnetic Bearing Driver

Each radial bearing is composed of pole pairs on opposing sides of the rotor. The attractive force exerted by each pole pair is proportional to the current supplied. However, the controller supplies only one current to control both opposing pole pairs. It is the task of the driver to divide this single control current to each of the individual pole pairs.

Another characteristic of magnetic bearings is their startup latency. There is a time lag from when the actual control current is applied to an unpowered magnetic bearing till when the rotor is subject to the attractive magnetic force. This is of particular concern for the radial bearings because the rotor's normal movement about the equilibrium point would constantly cause the controller to turn opposing sides of the bearing off and on. There is also the problem that small corrections (and therefore small control currents) applied to any one side of a radial bearing would be consumed by losses inherent to magnetic bearings. In order to overcome these limitations, the driver is also responsible for implementing a bias current whenever the control current falls below a certain value signifying that the rotor is close to the equilibrium point. When within this range, the driver adds or subtracts a certain proportion of the control current from the bias current and sends the resulting current to the respective sides of the radial bearing. In this way, the coil currents are rising and falling with respect to the control current and not constantly being turned off and on.




**A-13** Coil Current to Control Signal Relationship

The graphical representation of the relationship between the bearing coil current and the control current for this particular turbopump application is shown in Figure A-13. This driver mechanism produces three distinct operating zones which alter the form of the attractive magnetic force equation derived earlier. For purposes of analysis, $u$ will denote the control current and $u_0$ will denote the maximum current at which the driver uses a bias current to derive the magnetic bearing coil currents. Therefore, the driver relies on a bias current when $-u_0 < u < u_0$. For this particular turbopump, $u_0 = 1$A. In this analysis, the $x$ axis alone is chosen to represent the radial bearings but the $y$ axis could have just as easily been chosen. Also, the values of $x_l$ , $x_r$ , $x_t$ , and $x_b$ are replaced with an equivalent representation that uses the output from the position sensor and the nominal distance from the poles to the bearing center $h_0$. The three different forms of the magnetic force equation are as follows:

Case 1: $-u_0 > u$

Radial Bearing

$$F = -\frac{\mu_0 N^2 A \cos\beta u^2}{(h_0 + x)^2}$$

Axial Bearing

$$F = -\frac{\mu_0 N^2 A u^2}{4(h_0 + x)^2}$$

Case 2: $-u_0 < u < u_0$

Radial Bearing

$$F = \frac{\mu_0 N^2 A \cos\beta (I_0 + 0.5u)^2}{(h_0 - x)^2} - \frac{\mu_0 N^2 A \cos\beta (I_0 - 0.5u)^2}{(h_0 + x)^2}$$

Axial Bearing

$$F = \frac{\mu_0 N^2 A (I_0 + 0.5u)^2}{4(h_0 - x)^2} - \frac{\mu_0 N^2 A (I_0 - 0.5u)^2}{4(h_0 + x)^2}$$

Case 3: $u_0 < u$

Radial Bearing

$$F = \frac{\mu_0 N^2 A \cos\beta u^2}{(h_0 - x)^2}$$

Axial Bearing

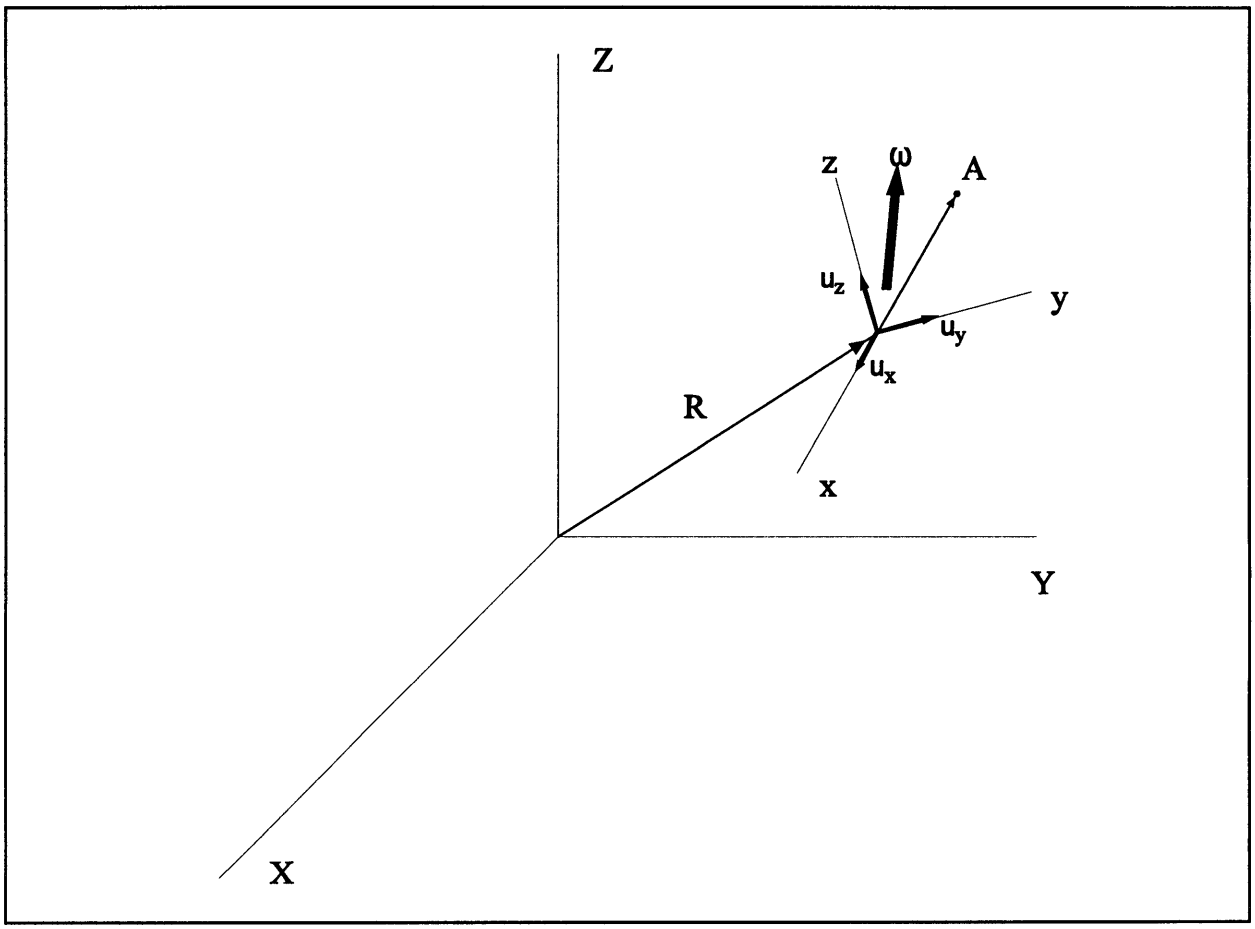
$$F = \frac{\mu_0 N^2 A u^2}{4(h_0 - x)^2}$$

The importance of understanding these three cases becomes apparent during the linearization process. The radial bearings will obviously fall within the criteria of case 2 but the axial bearing, which must compensate for the weight of the rotor assembly, it not as easy to categorize.

# Appendix B

# Rotor Mechanics

## B.1 Time Derivatives With Respect to an Intermediate Reference Frame




**B-1** Fixed and Intermediate Reference Frames

Let $\vec{R}$ be the position vector of the origin of the intermediate frame with respect to the

origin of the fixed reference frame, and let $\vec{\omega}$ denote the angular velocity vector of the intermediate frame with respect to the fixed reference frame. Now suppose that the position vector $\vec{A}$ is fixed with respect to the intermediate reference frame.

The vector $\vec{A}$ can be represented in the following form with respect to the intermediate reference frame $o_{xyz}$ [19],

$$\vec{A} = A_x\hat{u}_x + A_y\hat{u}_y + A_z\hat{u}_z \tag{B.1}$$

In general, the scalar components of this vector will vary with time and the unit vectors will vary in orientation when viewed from $O_{XYZ}$. Hence, the time rate of change of vector $\vec{A}$ with respect to the fixed reference frame is,

$$\left(\frac{d\vec{A}}{dt}\right)_{O_{XYZ}} = \dot{A}_x\hat{u}_x + \dot{A}_y\hat{u}_y + \dot{A}_z\hat{u}_z + A_x\frac{d\hat{u}_x}{dt} + A_y\frac{d\hat{u}_y}{dt} + A_z\frac{d\hat{u}_z}{dt} \tag{B.2}$$

The first three terms represent the rate of change of $\vec{A}$ with respect to the intermediate reference frame. The last three terms represent the contribution due to rotation of the intermediate reference frame with respect to the fixed reference frame. The first three terms can be written in the equivalent form,

$$\left(\frac{d\vec{A}}{dt}\right)_{o_{xyz}} = \dot{A}_x\hat{u}_x + \dot{A}_y\hat{u}_y + \dot{A}_z\hat{u}_z \tag{B.3}$$

As stated earlier, the last three terms are due to the rotation of the intermediate reference frame. Translational motion is ignored because this motion does not alter the direction of $\vec{A}$ as seen from $O_{XYZ}$. Also the magnitude of $\vec{A}$ is fixed for these terms and thus the vector cannot change as a result of this motion. The line of action of $\vec{A}$ however, will change as seen from the fixed reference frame $O_{XYZ}$ but a change of line of action does not signify a change in the vector. Therefore,

$$\begin{aligned} A_x\frac{d\hat{u}_x}{dt} + A_y\frac{d\hat{u}_y}{dt} + A_z\frac{d\hat{u}_z}{dt} &= A_x\vec{\omega}\times\hat{u}_x + A_y\vec{\omega}\times\hat{u}_y + A_z\vec{\omega}\times\hat{u}_z \\ &= \vec{\omega}\times\left(A_x\hat{u}_x + A_y\hat{u}_y + A_z\hat{u}_z\right) \\ &= \vec{\omega}\times\vec{A} \end{aligned} \tag{B.4}$$

Consequently,

$$\left(\frac{d\vec{A}}{dt}\right)_{O_{XYZ}} = \left(\frac{d\vec{A}}{dt}\right)_{o_{xyz}} + \vec{\omega}\times\vec{A} \tag{B.5}$$

Since $\vec{A}$ is an arbitrary vector, the derivative rule for vectors using intermediate reference frame is

$$\frac{d}{dt} = \left(\frac{d}{dt}\right)_{rel} + \vec{\omega}\times \tag{B.6}$$

## B.2 Forces

Newton's law states that linear momentum and force are related in the following way,

$$\vec{F} = \frac{d}{dt}(m\vec{v}) = m\frac{d}{dt}(\vec{v}) \tag{B.7}$$

Applying the derivative rule for vectors using intermediate reference frame to the velocity vector yields,

$$\vec{v} = \frac{d}{dt}(\vec{x}) = \frac{d}{dt}(\vec{R} + \vec{r}) = \frac{d\vec{R}}{dt} + \frac{d\vec{r}}{dt} + \vec{\omega}\times\vec{r} \tag{B.8}$$

$$\vec{F} = m\frac{d}{dt}(\vec{R} + \vec{r} + \vec{\omega}\times\vec{r}) \tag{B.9}$$

$$\vec{F} = m[\ddot{\vec{R}} + \ddot{\vec{r}} + \dot{\vec{\omega}}\times\vec{r} + \vec{\omega}\times\dot{\vec{r}} + \dot{\vec{\omega}}\times\vec{r} + \vec{\omega}\times\dot{\vec{r}} + \vec{\omega}\times(\vec{\omega}\times\vec{r})] \tag{B.10}$$

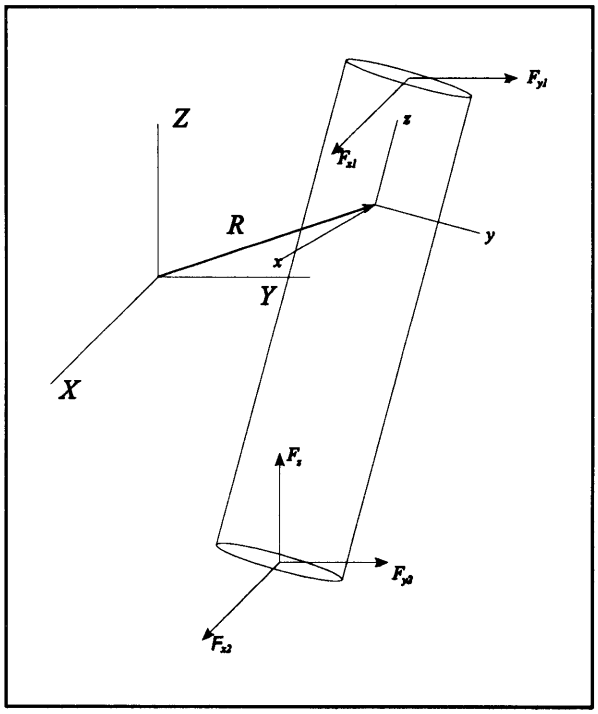
All points of interest in lie within the rigid body of the rotor hence $\ddot{r} = \dot{r} = 0$

$$\vec{F} = m[\ddot{\vec{R}} + \dot{\vec{\omega}}\times\vec{r} + \vec{\omega}\times(\vec{\omega}\times\vec{r})] \tag{B.11}$$

The steady state operation of the turbopump is of primary interest. At this stage, the angular acceleration $\dot{\vec{\omega}}$ is assumed to be negligible. Therefore,

$$\vec{F} = m[\ddot{\vec{R}} + \vec{\omega}\times(\vec{\omega}\times\vec{r})] \tag{B.12}$$

The external forces $\vec{F}$ must be measured relative to the fixed reference frame. In the model of the turbopump, these forces originate from the magnetic bearings present in the pump housing. Since the pump housing remains stationary with respect to the fixed reference frame, no further analysis is required with regards to external forces.



**B-2** Rotor Forces and Reference Frames

The acceleration term $\ddot{\vec{R}}$ is merely the acceleration of the intermediate reference frame with respect to the fixed reference frame. The intermediate reference frame has been purposely chosen to lie at the center of mass of the rotor. Therefore $\ddot{\vec{R}}$ is the acceleration of the center of mass of the rotor as viewed from the fixed reference frame. The last term is the angular velocity of the intermediate reference frame with respect to the fixed reference frame $\vec{\omega}$ and the distance from the intermediate reference frame to the point where the force is applied $\vec{r}$. It is assumed that the external forces applied by the magnetic bearings act at the center of the cross-sectional area of the rotor hence

$$\vec{r} = r\hat{u}_z \tag{B.13}$$

The angular velocity of the intermediate frame $\vec{\omega}$ will be more difficult to determine. The rotor of the turbopump has an angular velocity of

$$\vec{\omega} = \omega_x \hat{u}_x + \omega_y \hat{u}_y + \Omega \hat{u}_z \tag{B.14}$$

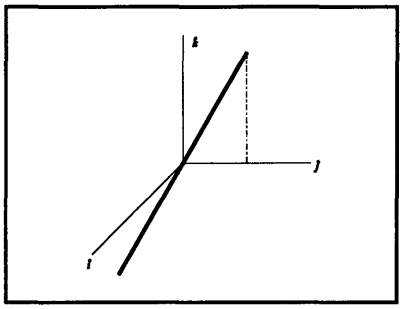
The $z$ component of the angular velocity is by far the largest component but the other components cannot be ignored until further analysis is performed to determine the magnitudes of the angular velocities in both the $\hat{u}_x$ and $\hat{u}_y$ direction that are allowable due to the clearances between the rotor and the magnetic bearing in both these directions [1].

Using the Euler angles there are three possible worst cases [6]:

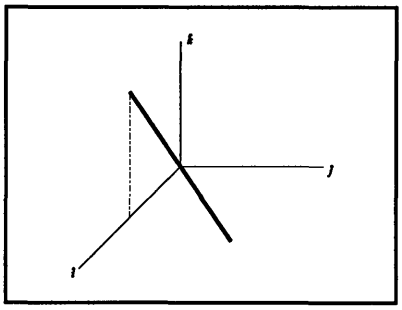
1. The rotor is inclined such that opposing ends of the rotor contact the bearings in the $\hat{j}$ direction.
2. The rotor is inclined such that opposing ends of the rotor contact the

bearings in the $\hat{i}$ direction.

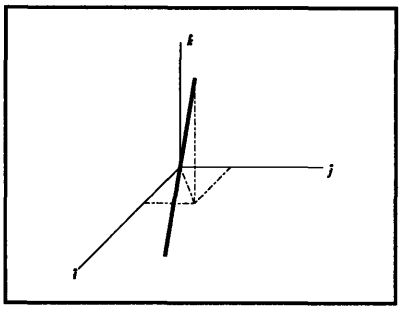
3. The rotor is inclined such that opposing ends of the rotor contact the bearings $\pi/4$ radians from the principle axes in the xy plane.



**B-3** Rotor Inclination 1



**B-4** Rotor Inclination 2



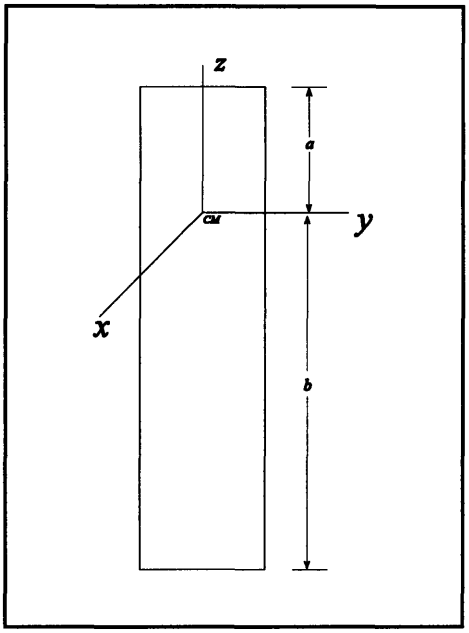
**B-5** Rotor Inclination 3

Before examining each of these cases, the relative magnitudes of the tolerances of the radial bearings will be examined. Assuming that the XYZ axes lie on the center line of the radial bearing, the clearance between the rotor and the radial bearings is shown at the right.

| | |
|---|---|
| $\hat{i}$ direction | ± 0.000125 m |
| $\hat{j}$ direction | ± 0.000125 m |
| $\hat{k}$ direction | ± 0.0002 m |

If the rotor is positioned such that its principle axes lie along the centerline of the radial bearings and midway between the axial bearings, then the lengths that correspond to those shown in Figure B-6 are:

a = 0.0238 m, b = 0.0691 m




**B-6** Rotor Center of Gravity

Now each of the three worst cases is examined.

Case 1: Rotor contacting radial bearings in $\hat{j}$ direction. Since the radial bearings lie in the housing, the force is always applied a distance of $a-z$ from the origin where $z$ is the distance the axial bearing varies from nominal center. Since $a \gg z$, this difference will be ignored in this and the remaining cases. Therefore

$$r\hat{u}_z = 0.000125\hat{j} + 0.0238\hat{k} \qquad \textbf{(B.15)}$$

$$r\hat{u}_z \simeq 0.0238\hat{k} = a\hat{k} \qquad \textbf{(B.16)}$$

Case 2: Rotor contacting radial bearings in $\hat{i}$ direction.
This case is very similar to Case 1

$$r\hat{u}_z = 0.000125\hat{i} + 0.0238\hat{k} \tag{B.17}$$

$$r\hat{u}_z \simeq 0.0238\hat{k} = a\hat{k} \tag{B.18}$$

Case 3: Rotor contacting the radial bearings midway in both the $\hat{i}$ and $\hat{j}$ directions.

$$r\hat{u}_z = 0.000125\cos\tfrac{\pi}{4}\hat{i} + 0.000125\sin\tfrac{\pi}{4}\hat{j} + 0.0238\hat{k} \tag{B.19}$$

$$r\hat{u}_z \simeq 0.0238\hat{k} = a\hat{k} \tag{B.20}$$

For all intensive purposes, the intermediate reference varies so little from the basic reference frame that their unit vectors are equivalent.

$$\begin{aligned} \hat{u}_k &= \hat{k} \\ \hat{u}_j &= \hat{j} \\ \hat{u}_i &= \hat{i} \end{aligned} \tag{B.21}$$

This same analysis is applied to the angular velocity vector. It is assumed that the angular velocity terms in the $\hat{u}_x$ and $\hat{u}_y$ directions are negligible in comparison to the angular velocity in the $\hat{u}_z$ direction. For this particular turbopump, the angular velocity in the $\hat{u}_z$ direction ranges from 12000-45000 rpm or 1250-4710 rad/sec. Such large magnitudes could translate into substantial angular velocities in the $\hat{u}_x$ and $\hat{u}_y$ directions if the rotor varies from true vertical relative to the fixed reference frame. Now the angular velocity is examined in the same manner as the position vector using the three worst cases. In this analysis, an average angular velocity of 3240 rad/sec will be used.

Case 1: Rotor contacting radial bearings in $\hat{j}$ direction.

$$\begin{aligned} \vec{\omega} &= \Omega\left[\sin\left(\arctan\frac{0.000125}{0.0238}\right)\hat{j} + \cos\left(\arctan\frac{0.000125}{0.0238}\right)\hat{k}\right] \\ &= \Omega\left[\sin(0.00525)\hat{j} + \cos(0.00525)\hat{k}\right] \end{aligned} \tag{B.22}$$

$$\begin{aligned} \sin(0.00525) &\simeq 0.00525 \\ \cos(0.00525) &\simeq 1.0 \end{aligned} \tag{B.23}$$

$$\vec{\omega} = \Omega\left[0.00525\hat{j} + \hat{k}\right] = 17\hat{j} + 3240\hat{k} \tag{B.24}$$

$$\begin{aligned}\vec{\omega}\times\vec{r} &= \left(17\hat{j} + 3240\hat{k}\right) \times \left(0.000125\hat{j} + 0.0238\hat{k}\right) \\ &= (0.4046 - 0.405)\hat{i} \\ &\simeq \vec{0}\end{aligned} \tag{B.25}$$

Case 2: Rotor contacting radial bearing in the $\hat{i}$ direction.
Again, this case is very similar to Case 1 hence

$$\vec{\omega} = \Omega\left[0.00525\hat{i} + \hat{k}\right] = 17\hat{i} + 3240\hat{k} \tag{B.26}$$

$$\begin{aligned}\vec{\omega}\times\vec{r} &= \left(17\hat{i} + 3240\hat{k}\right) \times \left(0.000125\hat{i} + 0.0238\hat{k}\right) \\ &= (-0.4046 + 0.405)\hat{j} \\ &\simeq \vec{0}\end{aligned} \tag{B.27}$$

Case 3: Rotor contacting the radial bearings midway in both the $\hat{i}$ and $\hat{j}$ directions.

$$\begin{aligned}\vec{\omega} &= \Omega\left[\sin\left(\arctan\frac{0.000125}{0.0238}\right)\cos\frac{\pi}{4}\hat{i} + \sin\left(\arctan\frac{0.000125}{0.0238}\right)\sin\frac{\pi}{4}\hat{j} + \cos\left(\arctan\frac{0.000125}{0.0238}\right)\hat{k}\right] \\ &= \Omega\left[\sin(0.00525)\cos\frac{\pi}{4}\hat{i} + \sin(0.00525)\sin\frac{\pi}{4}\hat{j} + \cos(0.00525)\hat{k}\right] \\ &= \Omega\left[0.00371\hat{i} + 0.00371\hat{j} + \hat{k}\right] = 12\hat{i} + 12\hat{j} + 3240\hat{k}\end{aligned} \tag{B.28}$$

$$\begin{aligned}\vec{\omega}\times\vec{r} &= \left(12\hat{i} + 12\hat{j} + 3240\hat{k}\right) \times \left(0.0000884\hat{i} + 0.0000884\hat{j} + 0.0238\hat{k}\right) \\ &= (0.2856 - 0.2864)\hat{i} + (-0.2856 + 0.2864)\hat{j} + (0.00106 - 0.00106)\hat{k} \\ &\simeq \vec{0}\end{aligned} \tag{B.29}$$

Therefore, the last term of Eq. (B-12) becomes

$$\vec{\omega}\times(\vec{\omega}\times\vec{r}) = \vec{\omega}\times\vec{0} = 0 \tag{B.30}$$

Finally,

$$\vec{F} = m\ddot{\vec{R}} \tag{B.31}$$

or

$$\begin{aligned} F_{x1} + F_{x2} &= m\ddot{x}_C \\ F_{y1} + F_{y2} &= m\ddot{y}_C \\ F_z &= m(\ddot{z}_C - g) \end{aligned} \tag{B.32}$$

# B.3 Moments

The moment $M$ and the moment of momentum $H$ are related in the following way,

$$\vec{M} = \frac{d\vec{H}}{dt} = \frac{d}{dt}(\vec{H})$$

using the vector derivative formula [8]

$$\vec{M} = \left(\frac{d\vec{H}}{dt}\right)_{o_{xyz}} + \vec{\omega}\times\vec{H} \tag{B.33}$$

The moment of momentum for a rigid body is

$$\vec{H} = I\vec{\omega} \tag{B.34}$$

Where **I** is the inertia tensor and $\vec{\omega}$ is the angular velocity vector of the rigid body. Also for a rigid body, the inertia tensor does not vary with time hence,

$$\vec{M} = I_C\dot{\vec{\omega}} + \vec{\omega}\times I_C\vec{\omega} \tag{B.35}$$

Since the intermediate reference has been purposely chosen to lie along the principle axes of the rotor

$$I_C = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} = \begin{bmatrix} I_{rr} & 0 & 0 \\ 0 & I_{rr} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{B.36}$$

Now the term $\vec{\omega}\times I_C\vec{\omega}$ is examined. Previously $\vec{\omega}$ was shown to have an average magnitude of

$$\vec{\omega} = 12\hat{i} + 12\hat{j} + 3240\hat{k} \tag{B.37}$$

Since $\omega_x = \omega_y \ll \omega_z$

$$\vec{\omega} \simeq 3240\hat{k} \tag{B.38}$$

and

$$I_C\vec{\omega} \simeq I_{zz}\omega_z\hat{k} \tag{B.39}$$

Therefore

$$\vec{M} = I_C\dot{\vec{\omega}} + \left(\vec{\omega}\times I_C\vec{\omega}\right) \tag{B.40}$$

$$\begin{aligned} M_x &= I_{rr}\dot{\omega}_x + I_{zz}\omega_y\omega_z = -aF_{y1} + bF_{y2} \\ M_y &= I_{rr}\dot{\omega}_y - I_{zz}\omega_x\omega_z = aF_{x1} - bF_{x2} \\ M_z &= I_{zz}\dot{\omega}_z = \text{Rotor Torque} \end{aligned} \tag{B.41}$$

Moments in the $z$ direction are merely the torque supplied by the motor to maintain a constant angular velocity and are therefore of little interest to us and shall be ignored. Finally,

$$\begin{aligned} I_{rr}\dot{\vec{\omega}}_x + I_{zz}\omega_y\omega_z &= -aF_{y1} + bF_{y2} \\ I_{rr}\dot{\vec{\omega}}_y - I_{zz}\omega_x\omega_z &= aF_{x1} - bF_{x2} \end{aligned} \tag{B.42}$$

# Appendix C

# Turbopump Equations of Motion

---

This appendix builds on fundamentals outlined in the previous two appendices and derives the turbopump equations of motion. The first section begins with the generalized nonlinear equations of motion and simplifies these equations given the geometric limitations of the turbopump system. Next the linearization formula is introduced. Then the linearized equations of the motion for the radial bearings are derived while the next section does likewise for the axial bearing. Finally, the final form of the linearized turbopump equations of motion are presented.

## C.1 Nonlinear Equations of Motion

The equations governing the motion of the rotor are the following,

$$\begin{aligned} F_{x1} + F_{x2} &= m\ddot{x}_C \\ F_{y1} + F_{y2} &= m\ddot{y}_C \\ F_z &= m(\ddot{z}_C - g) \\ I_{rr}\ddot{\Theta}_x + I_{zz}\dot{\Theta}_y\dot{\Theta}_z &= -aF_{y1} + bF_{y2} \\ I_{rr}\ddot{\Theta}_y + I_{zz}\dot{\Theta}_x\dot{\Theta}_z &= aF_{x1} - bF_{x2} \end{aligned} \tag{C.1}$$

Rearranging,

$$\begin{aligned} \ddot{x}_C &= \frac{F_{x1}}{m} + \frac{F_{x2}}{m} \\ \ddot{y}_C &= \frac{F_{y1}}{m} + \frac{F_{y2}}{m} \\ \ddot{z}_C &= \frac{F_z}{m} + g \\ \ddot{\Theta}_x &= \frac{-aF_{y1}}{I_{rr}} + \frac{bF_{y2}}{I_{rr}} - \frac{I_{zz}\dot{\Theta}_y\dot{\Theta}_z}{I_{rr}} \\ \ddot{\Theta}_y &= \frac{aF_{x1}}{I_{rr}} - \frac{bF_{x2}}{I_{rr}} + \frac{I_{zz}\dot{\Theta}_x\dot{\Theta}_z}{I_{rr}} \end{aligned} \tag{C.2}$$

In the following analysis, the subscript *1* denotes the upper bearing location, the subscript *2* denotes the lower bearing location, and the subscript *c* denotes the rotor center of gravity. The distance to where the forces $F_{x1}$ and $F_{y1}$ are applied to the rotor are

$$\begin{aligned} x_1 &= x_C + a\sin\Theta_y \\ y_1 &= y_C - a\sin\Theta_x \\ a &= z_1 - z_C \end{aligned} \tag{C.3}$$

The distance to where the $F_{x2}$ and $F_{y2}$ are applied to the rotor is

$$\begin{aligned} x_2 &= x_C - b\sin\Theta_y \\ y_2 &= y_C + b\sin\Theta_x \\ b &= z_C - z_2 \end{aligned} \tag{C.4}$$

Differentiating to obtain acceleration,

$$\begin{aligned} \dot{x}_1 &= \dot{x}_C + \dot{a}\sin\Theta_y + a\cos\Theta_y\dot{\Theta}_y \\ \dot{y}_1 &= \dot{y}_C - \dot{a}\sin\Theta_x - a\cos\Theta_x\dot{\Theta}_x \\ \dot{a} &= \dot{z}_1 - \dot{z}_C \end{aligned} \tag{C.5}$$

$$\begin{aligned} \ddot{x}_1 &= \ddot{x}_C + \ddot{a}\sin\Theta_y + \dot{a}\cos\Theta_y\dot{\Theta}_y + \dot{a}\cos\Theta_y\dot{\Theta}_y - a\sin\Theta_y\dot{\Theta}_y^2 + a\cos\Theta_y\ddot{\Theta}_y \\ \ddot{y}_1 &= \ddot{y}_C - \ddot{a}\sin\Theta_x - \dot{a}\cos\Theta_x\dot{\Theta}_x - \dot{a}\cos\Theta_x\dot{\Theta}_x + a\sin\Theta_x\dot{\Theta}_x^2 - a\cos\Theta_x\ddot{\Theta}_x \\ \ddot{a} &= \ddot{z}_1 - \ddot{z}_C \end{aligned} \tag{C.6}$$

First movement in the $z$ direction is examined. At the equilibrium conditions, when the center of mass of the rotor is positioned at the origin of the inertial reference frame, $a$ = 0.0238 m. The maximum possible deviation of the center of mass from equilibrium conditions is the maximum travel allowed by the magnetic bearings. This distance is equal to ± 0.00025. The maximum possible error when assuming $a$ is equal to its steady state value is,

$$\text{ERROR} = \frac{\pm 0.00025}{0.0238} = 0.0105 = 1.05\% \tag{C.7}$$

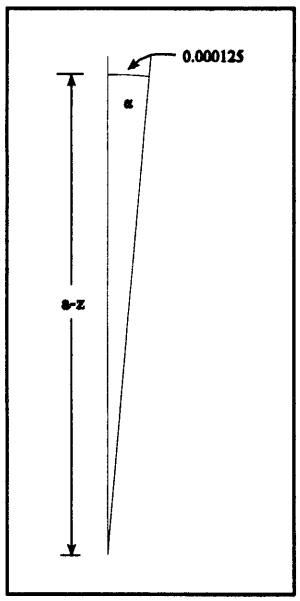
Therefore a very small error is incurred by assuming the following

$$a = a_{ss} \qquad \dot{a} = \ddot{a} = 0 \tag{C.8}$$

The equations for acceleration now simplify to

$$\begin{aligned} \ddot{x}_1 &= \ddot{x}_C - a_{ss}\sin\Theta_y\dot{\Theta}_y^2 + a_{ss}\cos\Theta_y\ddot{\Theta}_y \\ \ddot{y}_1 &= \ddot{y}_C + a_{ss}\sin\Theta_x\dot{\Theta}_x^2 - a_{ss}\cos\Theta_x\ddot{\Theta}_x \end{aligned} \tag{C.9}$$

Now the maximum contributions made by the angles $\theta_x$ and $\theta_y$ are examined. Due to symmetry of the rotor, our analysis of the contribution of the angle $\theta_x$ also applies equally to that of $\theta_y$. At the equilibrium point, $a$ = 0.0238 m. The maximum allowable rotation about any axis is related to the clearance between the rotor and the each individual magnetic bearing when both of their centerlines are equal. For the radial bearings, the clearance is equal to 0.000125 m. Therefore




$$\Theta_{MAX} = \arctan\left(\frac{0.000125}{0.0238}\right) \simeq 0.00525 \tag{C.10}$$

$$\begin{aligned} \sin\Theta_{MAX} &\simeq \Theta_{MAX} \\ \cos\Theta_{MAX} &\simeq 1 \end{aligned} \tag{C.11}$$

Substituting Eq. (C-11) into Eq. (C-9) yields

$$\begin{aligned} \ddot{x}_1 &= \ddot{x}_C - a_{ss}\Theta_y\dot{\Theta}_y^2 + a_{ss}\ddot{\Theta}_y \\ \ddot{y}_1 &= \ddot{y}_C + a_{ss}\Theta_x\dot{\Theta}_x^2 - a_{ss}\ddot{\Theta}_x \end{aligned} \tag{C.12}$$

Since $\theta_x$ and $\theta_y$ are likely to be extremely small, the final simplified equation for the acceleration at the point where the forces $F_{x1}$ and $F_{y1}$ are applied is,

$$\begin{aligned} \ddot{x}_1 &= \ddot{x}_C + a_{ss}\ddot{\Theta}_y \\ \ddot{y}_1 &= \ddot{y}_C - a_{ss}\ddot{\Theta}_x \end{aligned} \tag{C.13}$$

Likewise, this same analysis can be performed for the acceleration at the point where the forces $F_{x2}$ and $F_{y2}$ are applied and the ensuing simplified equation would be

$$\begin{aligned} \ddot{x}_2 &= \ddot{x}_C - b_{ss}\ddot{\Theta}_y \\ \ddot{y}_2 &= \ddot{y}_C + b_{ss}\ddot{\Theta}_x \end{aligned} \tag{C.14}$$

Substituting Eq. (C-2) into Eq. (C-13) and Eq. (C-14) yields,

$$\begin{aligned}
\ddot{x}_1 &= \frac{F_{x1}}{m} + \frac{F_{x2}}{m} + \frac{a^2 F_{x1}}{I_{rr}} - \frac{abF_{x1}}{I_{rr}} + \frac{aI_{zz}\dot{\Theta}_x\dot{\Theta}_z}{I_{rr}} \\
\ddot{x}_2 &= \frac{F_{x1}}{m} + \frac{F_{x2}}{m} - \frac{abF_{x1}}{I_{rr}} + \frac{b^2 F_{x1}}{I_{rr}} - \frac{bI_{zz}\dot{\Theta}_x\dot{\Theta}_z}{I_{rr}} \\
\ddot{y}_1 &= \frac{F_{y1}}{m} + \frac{F_{y2}}{m} + \frac{a^2 F_{y1}}{I_{rr}} - \frac{abF_{y1}}{I_{rr}} + \frac{aI_{zz}\dot{\Theta}_y\dot{\Theta}_z}{I_{rr}} \\
\ddot{y}_2 &= \frac{F_{y1}}{m} + \frac{F_{y2}}{m} - \frac{abF_{y1}}{I_{rr}} + \frac{b^2 F_{y1}}{I_{rr}} - \frac{bI_{zz}\dot{\Theta}_y\dot{\Theta}_z}{I_{rr}}
\end{aligned} \tag{C.15}$$

Rearranging,

$$\begin{aligned}
\ddot{x}_1 &= \left(\frac{1}{m} + \frac{a^2}{I_{rr}}\right) F_{x1} + \left(\frac{1}{m} - \frac{ab}{I_{rr}}\right) F_{x2} + \left(\frac{aI_{zz}}{I_{rr}}\right) \dot{\Theta}_x\dot{\Theta}_z \\
\ddot{x}_2 &= \left(\frac{1}{m} - \frac{ab}{I_{rr}}\right) F_{x1} + \left(\frac{1}{m} + \frac{b^2}{I_{rr}}\right) F_{x2} - \left(\frac{bI_{zz}}{I_{rr}}\right) \dot{\Theta}_x\dot{\Theta}_z \\
\ddot{y}_1 &= \left(\frac{1}{m} + \frac{a^2}{I_{rr}}\right) F_{y1} + \left(\frac{1}{m} - \frac{ab}{I_{rr}}\right) F_{y2} + \left(\frac{aI_{zz}}{I_{rr}}\right) \dot{\Theta}_y\dot{\Theta}_z \\
\ddot{y}_2 &= \left(\frac{1}{m} - \frac{ab}{I_{rr}}\right) F_{y1} + \left(\frac{1}{m} + \frac{b^2}{I_{rr}}\right) F_{y2} - \left(\frac{bI_{zz}}{I_{rr}}\right) \dot{\Theta}_y\dot{\Theta}_z
\end{aligned} \tag{C.16}$$

All of the above variables are known except for the angular velocities in the $x$ and $y$ direction. To determine these unknowns, Chasle's Theorem is applied. Chasle's Theorem states any motion of a rigid body can be decomposed into a translation of a point on the rigid body and a rotation of the rigid body about the same point. This theorem when applied to the derivatives of translation yields,

$$\dot{x}_1 = \dot{x}_C + \omega_C r_1 \tag{C.17}$$

Applying this equation to our rotor yields

$$\begin{aligned} \dot{x}_1 &= \dot{x}_C + a\dot{\Theta}_y \\ \dot{x}_2 &= \dot{x}_C - b\dot{\Theta}_y \end{aligned} \tag{C.18}$$

Solving for $x_c$ and equating both equations yields,

$$\dot{x}_1 - a\dot{\Theta}_y = \dot{x}_2 + b\dot{\Theta}_y \tag{C.19}$$

Solving for $\dot{\Theta}_y$

$$\dot{\Theta}_y = \frac{\dot{x}_1 - \dot{x}_2}{a + b} \tag{C.20}$$

Likewise, solving for $\dot{\Theta}_x$

$$\dot{\Theta}_x = \frac{\dot{y}_2 - \dot{y}_1}{a + b} \tag{C.21}$$

Substituting Eq. (C-20) and Eq. (C-21) into Eq. (C-16) produces the final simplified equations of motion

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{y}_1 \\ \dot{y}_2 \\ \dot{z} \\ \dot{v}_{x1} \\ \dot{v}_{x2} \\ \dot{v}_{y1} \\ \dot{v}_{y2} \\ \dot{v}_z \end{bmatrix} = \begin{bmatrix} v_{x1} \\ v_{x2} \\ v_{y1} \\ v_{y2} \\ v_z \\ \left(\frac{1}{m} + \frac{a^2}{I_{rr}}\right) F_{x1} + \left(\frac{1}{m} - \frac{ab}{I_{rr}}\right) F_{x2} - \left(\frac{aI_{zz}\dot{\Theta}_z}{(a+b)I_{rr}}\right) \dot{y}_1 + \left(\frac{aI_{zz}\dot{\Theta}_z}{(a+b)I_{rr}}\right) \dot{y}_2 \\ \left(\frac{1}{m} - \frac{ab}{I_{rr}}\right) F_{x1} + \left(\frac{1}{m} + \frac{a^2}{I_{rr}}\right) F_{x2} + \left(\frac{bI_{zz}\dot{\Theta}_z}{(a+b)I_{rr}}\right) \dot{y}_1 - \left(\frac{bI_{zz}\dot{\Theta}_z}{(a+b)I_{rr}}\right) \dot{y}_2 \\ \left(\frac{1}{m} + \frac{a^2}{I_{rr}}\right) F_{y1} + \left(\frac{1}{m} - \frac{ab}{I_{rr}}\right) F_{y2} + \left(\frac{aI_{zz}\dot{\Theta}_z}{(a+b)I_{rr}}\right) \dot{x}_1 - \left(\frac{aI_{zz}\dot{\Theta}_z}{(a+b)I_{rr}}\right) \dot{x}_2 \\ \left(\frac{1}{m} - \frac{ab}{I_{rr}}\right) F_{y1} + \left(\frac{1}{m} + \frac{a^2}{I_{rr}}\right) F_{y2} - \left(\frac{bI_{zz}\dot{\Theta}_z}{(a+b)I_{rr}}\right) \dot{x}_1 + \left(\frac{bI_{zz}\dot{\Theta}_z}{(a+b)I_{rr}}\right) \dot{x}_2 \\ \frac{F_z}{m} - g \end{bmatrix} \tag{C.22}$$

# C.2 Linearization of the Magnetic Force Equations

It can be seen that since the force exerted by the magnetic bearings is nonlinear, that the equations of motion of the rotor are nonlinear. Therefore, linearizing the magnetic bearing force

equation about the equilibrium position of the rotor is the obvious next step. Given a system of nonlinear equations having the following form [12],

$$\frac{d\vec{x}(t)}{dt} = \vec{f}[\vec{x}(t), \vec{u}(t)] \tag{C.23}$$

The linearization process utilizing Taylor series expansion and discarding all terms higher than first order yields

$$\Delta\dot{x}_i = \sum_{j=1}^{n} \left.\frac{\partial \vec{f}_i(\vec{x},\vec{u})}{\partial \vec{x}_j}\right|_{\vec{x}_0,\vec{u}_0} \Delta x_j + \sum_{j=1}^{p} \left.\frac{\partial \vec{f}_i(\vec{x},\vec{u})}{\partial \vec{u}_j}\right|_{\vec{x}_0,\vec{u}_0} \Delta u_j \tag{C.24}$$

$x_0$ denotes the nominal operating point corresponding to the nominal input $u_0$. In vector matrix form, the above equation becomes

$$\Delta\vec{x}' = \vec{A}^*\Delta\vec{x} + \vec{B}^*\Delta\vec{u} \tag{C.25}$$

where

$$A^* = \left[\begin{array}{cccc} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{array}\right]_{x_0,u_0} \qquad B^* = \left[\begin{array}{cccc} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} & \cdots & \frac{\partial f_1}{\partial u_n} \\ \frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} & \cdots & \frac{\partial f_2}{\partial u_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial u_1} & \frac{\partial f_n}{\partial u_2} & \cdots & \frac{\partial f_n}{\partial u_n} \end{array}\right]_{x_0,u_0} \tag{C.26}$$

# C.3 Radial Bearing Linearization

The state equations representing the attractive magnetic force applied to the rotor of our turbopump by one axis of each radial bearing are

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\frac{\mu_0 N^2 A \cos\beta (I_0 - 0.5u)^2}{m(h_0 + x_1)^2} + \frac{\mu_0 N^2 A \cos\beta (I_0 + 0.5u)^2}{m(h_0 - x_1)^2} \end{aligned} \tag{C.27}$$

where: $\mu_0$ = air permeability constant
$N$ = number of solenoid wire turns
$A$ = solenoid cross-sectional area
$I_0$ = magnetic solenoid bias current
$m$ = rotor assembly mass
$h_0$ = rotor equilibrium state position
$u$ = control current
$x_1$ = bearing position sensor value

The above equations assume that the control current will be small enough so that the driver is applying a bias current to each side of the magnetic bearing axis. This is a reasonable assumption because a rotor that is not spinning is not subjected to any forces in the $x$ and $y$ directions. Differentiating each term of the linearization matrix yields,

$$\begin{aligned}
&\frac{\partial f_1}{\partial x_1} = 0 \quad \frac{\partial f_1}{\partial x_2} = 1 \quad \frac{\partial f_2}{\partial x_2} = 0 \quad \frac{\partial f_1}{\partial r} = 0 \\
&\frac{\partial f_2}{\partial x_1} = \frac{2\mu_0 N^2 A \cos\beta (I_0 - 0.5u)^2}{m(h_0 + x_1)^3} + \frac{2\mu_0 N^2 A \cos\beta (I_0 + 0.5u)^2}{m(h_0 - x_1)^3} \\
&\frac{\partial f_2}{\partial r} = \frac{\mu_0 N^2 A \cos\beta (I_0 - 0.5u)}{m(h_0 + x_1)^2} + \frac{\mu_0 N^2 A \cos\beta (I_0 + 0.5u)}{m(h_0 - x_1)^2}
\end{aligned} \tag{C.28}$$

If the inertia reference frame is chosen such that it coincides with rotor center of mass, then the equilibrium point becomes $y_0 = 0$, $x_0 = 0$, and $u_0 = 0$. Evaluating the terms of the linearization matrix about the equilibrium point yields

$$\begin{aligned}
&\frac{\partial f_1}{\partial x_1} = 0 \quad \frac{\partial f_1}{\partial x_2} = 1 \quad \frac{\partial f_2}{\partial x_2} = 0 \quad \frac{\partial f_1}{\partial r} = 0 \\
&\frac{\partial f_2}{\partial x_1} = \frac{2\mu_0 N^2 A \cos\beta I_0^2}{m h_0^3} + \frac{2\mu_0 N^2 A \cos\beta I_0^2}{m h_0^3} = \frac{4\mu_0 N^2 A \cos\beta I_0^2}{m h_0^3} \\
&\frac{\partial f_2}{\partial r} = \frac{\mu_0 N^2 A \cos\beta I_0}{m h_0^2} + \frac{\mu_0 N^2 A \cos\beta I_0}{m h_0^2} = \frac{2\mu_0 N^2 A \cos\beta I_0}{m h_0^2}
\end{aligned} \tag{C.29}$$

Therefore, the linear magnetic force is

$$\begin{bmatrix} \Delta\dot{x}_1 \\ \Delta\dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{4\mu_0 N^2 A I_0^2}{m h_0^3} & 0 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{2\mu_0 N^2 A I_0}{m h_0^2} \end{bmatrix} u \tag{C.30}$$

or,

$$F_{(x,y)(1,2)} = \frac{4\mu_0 N^2 A \cos\beta I_0^2}{h_0^2}(x,y)_{(1,2)} + \frac{2\mu_0 N^2 A \cos\beta I_0}{h_0^2} u_{(x,y)(1,2)} \tag{C.31}$$

# C.4 Axial Bearing Linearization

The state equations representing the attractive magnetic force applied to the disk attached to the rotor of our turbopump by the axial bearing are

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\frac{\mu_0 N^2 A (I_0 - 0.5u)^2}{4m(h_0 + x_1)^2} + \frac{\mu_0 N^2 A (I_0 + 0.5u)^2}{4m(h_0 - x_1)^2} \end{aligned} \tag{C.32}$$

where: $\mu_0$ = air permeability constant
$N$ = number of solenoid wire turns
$A$ = solenoid cross-sectional area
$I_0$ = magnetic solenoid bias current
$m$ = rotor assembly mass
$h_0$ = rotor equilibrium state position
$u$ = control current
$x_1$ = bearing position sensor value

The above equations assume that the control current will be small enough so that the driver is applying a bias current to each side of the magnetic bearing axis. This assumption will have to be verified later to determine its acceptability because the rotor is constantly being subjected to the force of gravity in the axial direction. Differentiating each term of the linearization matrix yields,

$$\begin{aligned} &\frac{\partial f_1}{\partial x_1} = 0 \quad \frac{\partial f_1}{\partial x_2} = 1 \quad \frac{\partial f_2}{\partial x_2} = 0 \quad \frac{\partial f_1}{\partial r} = 0 \\ &\frac{\partial f_2}{\partial x_1} = \frac{\mu_0 N^2 A (I_0 - 0.5u)^2}{2m(h_0 + x_1)^3} + \frac{\mu_0 N^2 A (I_0 + 0.5u)^2}{2m(h_0 - x_1)^3} \\ &\frac{\partial f_2}{\partial r} = \frac{\mu_0 N^2 A (I_0 - 0.5u)}{4m(h_0 + x_1)^2} + \frac{\mu_0 N^2 A (I_0 + 0.5u)}{4m(h_0 - x_1)^2} \end{aligned} \tag{C.33}$$

If the inertia reference frame is chosen such that it coincides with rotor center of mass, then the equilibrium point becomes $x_0 = 0$. However, the value of $u_0$ is not obvious because unlike the

radial bearing, the attractive force most compensate for the weight of the rotor. In order to determine $u_0$, Eq. (C-32) is evaluated with $x_1 = x_0 = 0$ and $\dot{x} = g$

$$\begin{aligned} g &= -\frac{\mu_0 N^2 A (I_0 - 0.5u)^2}{4mh_0^2} + \frac{\mu_0 N^2 A (I_0 + 0.5u)^2}{4mh_0^2} \\ &= \frac{\mu_0 N^2 A I_0 u}{2mh_0^2} \end{aligned} \tag{C.34}$$

Rearranging,

$$u = \frac{2mgh_0^2}{\mu_0 N^2 A I_0} \tag{C.35}$$

| Parameter | Axial | Radial |
|---|---|---|
| Air Permeability, $\mu_0$ ($N/A^2$) | $1.26 \times 10^{-6}$ | $1.26 \times 10^{-6}$ |
| Mass, m (Kg) | 2.2 | 2.2 |
| Number of Wire Turns, N | 133 | 100 |
| Magnetic Flux Area, A ($m^2$) | $7.0 \times 10^{-4}$ | $9.75 \times 10^{-5}$ |
| Bias Current (A) | 0.5 | 0.5 |
| Centerline Distance, $h_0$ (m) | $4.0 \times 10^{-4}$ | $2.5 \times 10^{-4}$ |

Substituting the values supplied by the manufacturer into Eq. (C-35) yields $u = 0.885$ which corresponds to the case $u_0 > u > -u_0$. Therefore the previously assumed driver equation is correct and re-evaluating Eq. (C-33) with $z_0 = 0$, $u_z = 0.885$ yields,

$$\begin{aligned} \frac{\partial f_2}{\partial x_1} &= \frac{\mu_0 N^2 A (I_0 - 0.443)^2}{2mh_0^3} + \frac{\mu_0 N^2 A (I_0 + 0.443)^2}{2mh_0^3} = \frac{\mu_0 N^2 A}{mh_0^3}\left(I_0^2 + 0.196\right) \\ \frac{\partial f_2}{\partial r} &= \frac{\mu_0 N^2 A (I_0 - 0.443)}{4mh_0^2} + \frac{\mu_0 N^2 A (I_0 + 0.443)}{4mh_0^2} = \frac{\mu_0 N^2 A I_0}{2mh_0^2} \end{aligned} \tag{C.36}$$

Therefore, the linear magnetic force is

$$\begin{bmatrix} \Delta\dot{x}_1 \\ \Delta\dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{\mu_0 N^2 A\left(I_0^2+0.196\right)}{m h_0^3} & 0 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{\mu_0 N^2 A I_0}{2m h_0^2} \end{bmatrix} u \tag{C.37}$$

or,

$$F_z = \frac{\mu_0 N^2 A\left(I_0^2+0.196\right)}{h_0^3} z + \frac{\mu_0 N^2 A I_0}{2h_0^2} u_z \tag{C.38}$$

The theoretical analysis thus far has shown that the control signal necessary to maintain the rotor in the equilibrium position lies within the range $u_0 > u > -u_0$. In actuality, by monitoring the driver test points of the axial bearing, the control signal necessary to maintain the rotor in the equilibrium position lies within the range $u > u_0$. The reason for this discrepancy can be explained by the omission of the power amplifier in the theoretical analysis of the system. The analysis thus far assumed that the output signal from the driver is applied directly to the magnetic bearings. In actuality, the driver output signal first passes through a power amplifier before being sent to the magnetic bearings. This does not effect the analysis of the radial bearings because omission of the power amplifier would only decrease the signal required from the driver and therefore the equilibrium position would remain in the range $u_0 > u > -u_0$. In Appendix E, the power amplifier is assumed to be a constant having a form of $A_1/A_2$. The values for each component of this gain are determined by comparing the DC gains of the actual system response between the driver output test port and the position signal test port with the theoretical system response from the linearized equations of motion. This analysis shows that for the axial bearing alone, the power amplifier actually attenuates the driver signal and that the values are $A_1 = 1.0$, $A_2 = 1.138$. Using this information, Eq. (C-34) becomes,

$$\begin{aligned} g &= -\frac{\mu_0 N^2 A\left(I_0-0.5u\right)^2 A_1}{4m h_0^2 A_2} + \frac{\mu_0 N^2 A\left(I_0+0.5u\right)^2 A_1}{4m h_0^2 A_2} \\ &= \frac{\mu_0 N^2 A I_0 u A_1}{2m h_0^2 A_2} \end{aligned} \tag{C.39}$$

Rearranging,

$$u = \frac{2mgh_0^2 A_2}{\mu_0 N^2 A I_0 A_1} \tag{C.40}$$

Substituting the values supplied by the manufacturer into Eq. (C-40) yields $u = 1.007$ which corresponds to the case $u > u_0$. Therefore actual state equations representing the attractive

magnetic force applied to the disk attached to the rotor of our turbopump by the axial bearing are now

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= \frac{\mu_0 N^2 A u^2}{4m(h_0 - x_1)^2} \end{aligned} \tag{C.41}$$

Differentiating each term of the linearization matrix yields,

$$\begin{aligned} &\frac{\partial f_1}{\partial x_1} = 0 \quad \frac{\partial f_1}{\partial x_2} = 1 \quad \frac{\partial f_2}{\partial x_2} = 0 \quad \frac{\partial f_1}{\partial r} = 0 \\ &\frac{\partial f_2}{\partial x_1} = \frac{\mu_0 N^2 A u^2}{2m(h_0 - x_1)^3} \\ &\frac{\partial f_2}{\partial r} = \frac{\mu_0 N^2 A u}{2m(h_0 - x_1)^2} \end{aligned} \tag{C.42}$$

If the inertia reference frame is chosen such that it coincides with rotor center of mass, then the equilibrium point becomes $y_0 = 0$, $x_0 = 0$, and $u_0 \approx 1$. Evaluating the terms of the linearization matrix about the equilibrium point yields

$$\begin{aligned} &\frac{\partial f_1}{\partial x_1} = 0 \quad \frac{\partial f_1}{\partial x_2} = 1 \quad \frac{\partial f_2}{\partial x_2} = 0 \quad \frac{\partial f_1}{\partial r} = 0 \\ &\frac{\partial f_2}{\partial x_1} = \frac{\mu_0 N^2 A u_0^2}{2m h_0^3} \\ &\frac{\partial f_2}{\partial r} = \frac{\mu_0 N^2 A u_0}{2m h_0^2} \end{aligned} \tag{C.43}$$

The numerical value of $u_0$ could have been substituted into Eq. (C-43) but this would probably lead to confusion later because the dimensions of the remaining variables would not produce a suitable answer. Therefore, the linear magnetic force is

$$\begin{bmatrix} \Delta\dot{x}_1 \\ \Delta\dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{\mu_0 N^2 A u_0^2}{2m h_0^3} & 0 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{\mu_0 N^2 A u_0}{2m h_0^2} \end{bmatrix} u \tag{C.44}$$

or,

$$F_z = \frac{\mu_0 N^2 A u_0^2}{2h_0^3} z + \frac{\mu_0 N^2 A u_0}{2h_0^2} u_z \tag{C.45}$$

## C.5 Linearized Equations of Motion

Given the following constants,

$$k = \mu_0 N^2 A, \quad C_1 = \left(\frac{1}{m} + \frac{a^2}{I_{rr}}\right), \quad C_2 = \left(\frac{1}{m} + \frac{b^2}{I_{rr}}\right), \quad C_3 = \left(\frac{1}{m} - \frac{ab}{I_{rr}}\right),$$
$$C_4 = \left(\frac{aI_{zz}\dot{\Theta}_z}{(a+b)I_{rr}}\right), \quad C_5 = \left(\frac{bI_{zz}\dot{\Theta}_z}{(a+b)I_{rr}}\right) \tag{C.46}$$

Substituting Eq. (C.31) and Eq. (C.45) into Eq. (C.22), the final linearized equations of motion then become,

$$
\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{y}_1 \\ \dot{y}_2 \\ \dot{z} \\ \dot{v}_{x1} \\ \dot{v}_{x2} \\ \dot{v}_{y1} \\ \dot{v}_{y2} \\ \dot{v}_z \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \frac{4C_1 k\cos\beta I_0^2}{h_0^3} & \frac{4C_3 k\cos\beta I_0^2}{h_0^3} & 0 & 0 & 0 & 0 & 0 & -C_4 & C_4 & 0 \\ \frac{4C_3 k\cos\beta I_0^2}{h_0^3} & \frac{4C_2 k\cos\beta I_0^2}{h_0^3} & 0 & 0 & 0 & 0 & 0 & C_5 & -C_5 & 0 \\ 0 & 0 & \frac{4C_1 k\cos\beta I_0^2}{h_0^3} & \frac{4C_3 k\cos\beta I_0^2}{h_0^3} & 0 & C_4 & -C_4 & 0 & 0 & 0 \\ 0 & 0 & \frac{4C_3 k\cos\beta I_0^2}{h_0^3} & \frac{4C_2 k\cos\beta I_0^2}{h_0^3} & 0 & -C_5 & C_5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{ku_0^2}{2mh_0^3} & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ y_1 \\ y_2 \\ z \\ v_{x1} \\ v_{x2} \\ v_{y1} \\ v_{y2} \\ v_z \end{bmatrix}
$$

$$
+ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \frac{2C_1 k\cos\beta I_0}{h_0^2} & \frac{2C_3 k\cos\beta I_0}{h_0^2} & 0 & 0 & 0 \\ \frac{2C_3 k\cos\beta I_0}{h_0^2} & \frac{2C_2 k\cos\beta I_0}{h_0^2} & 0 & 0 & 0 \\ 0 & 0 & \frac{2C_1 k\cos\beta I_0}{h_0^2} & \frac{2C_3 k\cos\beta I_0}{h_0^2} & 0 \\ 0 & 0 & \frac{2C_3 k\cos\beta I_0}{h_0^2} & \frac{2C_2 k\cos\beta I_0}{h_0^2} & 0 \\ 0 & 0 & 0 & 0 & \frac{ku_0}{2mh_0^2} \end{bmatrix} \begin{bmatrix} u_{x1} \\ u_{x2} \\ u_{y1} \\ u_{y2} \\ u_z \end{bmatrix} \qquad \textbf{(C.47)}
$$

# Appendix D

# Hybrid System Modeling

In this appendix, different approaches are used to model the hybrid system. First, the digital controller is modeled as a continuous time system thereby allowing the entire system to analyzed using continuous time design methods. Then the plant is modeled as a discrete time system thereby allowing the entire system to analyzed using discrete time design methods. Finally, the effects of using derivatives in the discrete time controller are analyzed.

## D.1 Purely Continuous System

All calculations were applied to the radial bearing equations. This same analysis also applies to the axial bearings but the results are not shown in the interest of brevity.

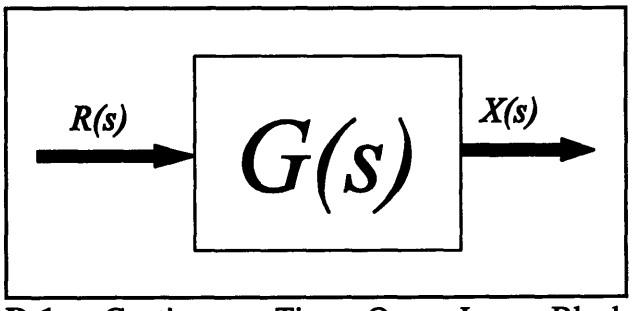
### D.1.1 Open Loop System

The block diagram of the open loop system is shown in Figure D-1. The linearized radial bearing equation represented in the block diagram by **G(s)** is,

$$\dot{v}_{xI} = \frac{4C_1 kI_0^2}{h_0^3}x_1 + \frac{4C_3 kI_0^2}{h_0^3}x_2 - C_4\dot{y}_1 + C_4\dot{y}_2 + \frac{2C_1 kI_0}{h_0^2}u_{xI} + \frac{2C_3 kI_0}{h_0^2}u_{x2} \qquad \textbf{(D.1)}$$

Assuming that the coupling between bearings can be treated as a disturbance ($x_2$ and $u_{x2}$ have little or no effect on $x_1$) and that the rotor is not rotating ($C_4 = 0$) [12],

$$\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{4C_1 kI_0^2}{h_0^3} & 0 \end{bmatrix}\begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{2C_1 kI_0}{h_0^2} \end{bmatrix} u \qquad \textbf{(D.2)}$$




**D-1** Continuous Time Open Loop Block Diagram

$$[y] = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + [0]u \tag{D.3}$$

Taking the Laplace transform,

$$\begin{aligned} sX(s) &= AX(s) + BU(s) \\ Y(s) &= DX(s) + EU(s) \end{aligned} \tag{D.4}$$

$$\frac{X(s)}{R(s)} = D[sI - A]^{-1}B + E \tag{D.5}$$

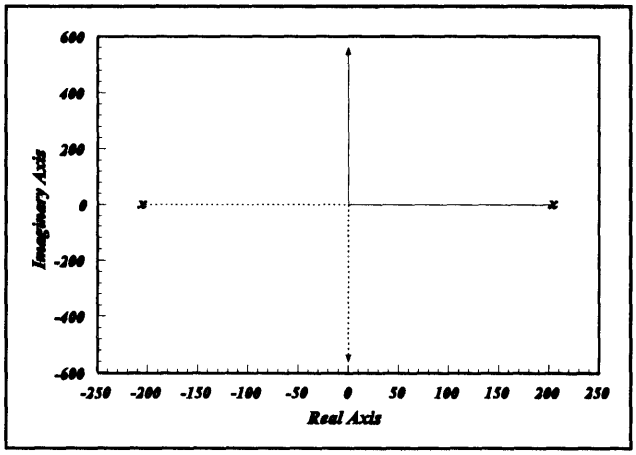
$$\frac{X(s)}{R(s)} = \begin{bmatrix} 1 & 0 \end{bmatrix} \left\{ \frac{1}{s^2 - \frac{4C_1 kI_0^2}{h_0^3}} \begin{bmatrix} s & 1 \\ \frac{4C_1 kI_0^2}{h_0^3} & s \end{bmatrix} \right\} \begin{bmatrix} 0 \\ \frac{2C_1 kI_0}{h_0^2} \end{bmatrix} + [0] \tag{D.6}$$

$$\frac{X(s)}{R(s)} = \frac{\frac{2C_1 kI_0}{h_0^2}}{s^2 - \frac{4C_1 kI_0^2}{h_0^3}} \tag{D.7}$$

Recall that the constant $C_1$ was defined as

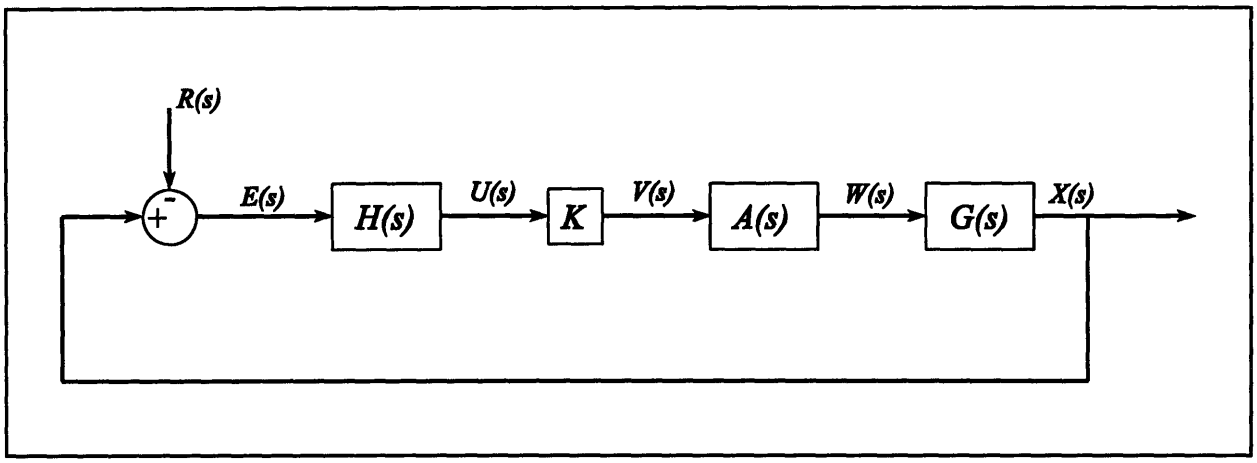
$$C_1 = \left( \frac{1}{m} + \frac{a^2}{I_{rr}} \right) \tag{D.8}$$

Figure D-2 displays the root locus diagram of the transfer function represented by Eq. (D.7). Since the constant $C_1$ is always positive, a pole of the characteristic equation lies in the right half plane. Therefore the magnetic radial bearing is unstable in the absence of control and at best marginally stable under proportional control.




**D-2** Continuous Time Open Loop Root Locus Diagram

## D.1.2 Closed Loop System



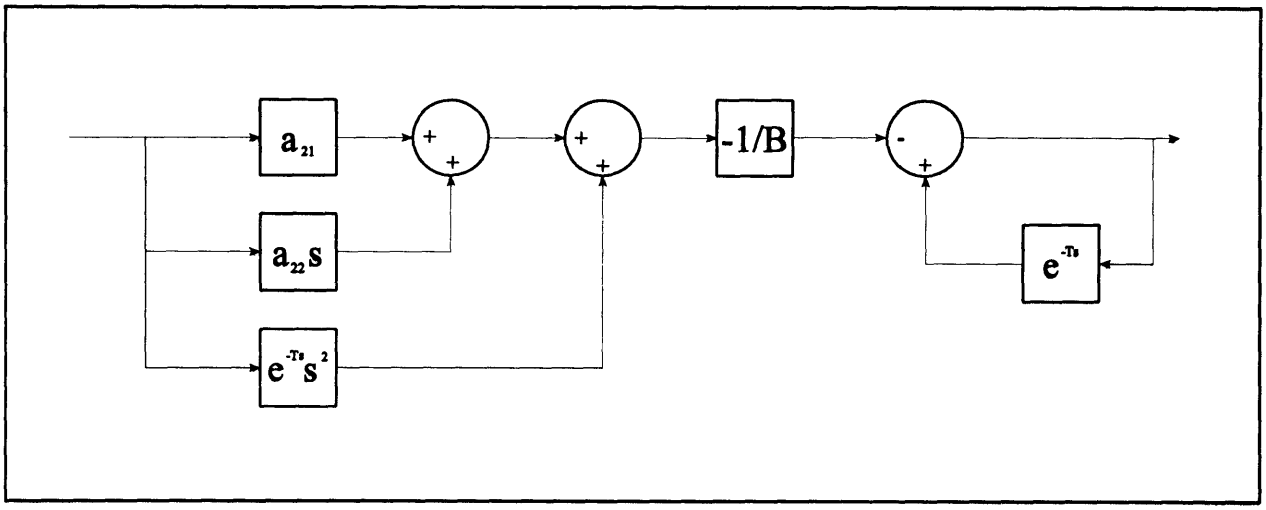

**D-3** Continuous Time Closed Loop System Block Diagram

The block diagram of the closed loop system is shown in Figure D-3. **H(s)** represents the digital controller, **K** represents the control gain, **A(s)** represents the control current driver which is responsible for generating the bias current and dividing the control current between both sides of the bearing axis, and **G(s)** represents the turbopump. From the block diagram, the individual block equations are,

$$\begin{aligned} E(s) &= X(s) - R(s) \\ U(s) &= H(s)E(s) \\ V(s) &= KU(s) \\ W(s) &= A(s)V(s) \\ X(s) &= G(s)W(s) \end{aligned} \tag{D.9}$$

Rearranging yields

$$\frac{X(s)}{R(s)} = \frac{KA(s)G(s)H(s)}{KA(s)G(s)H(s) - 1} \tag{D.10}$$




**D-4** Continuous Time Controller Block Diagram

Figure D-4 shows the block diagram representation of the digital controller in a continuous time form. The continuous time digital controller approximation is,

$$H(s) = \frac{-\left(e^{-Ts}s^2 + a_{22}s + a_{21}\right)}{B\left(1 - e^{-Ts}\right)} \tag{D.11}$$

The continuous time turbopump approximation is,

$$G(s) = \frac{\frac{2C_1kI_0}{h_0^2}}{s^2 - \frac{4C_1kI_0^2}{h_0^3}} = \frac{P}{s^2 - Q} \tag{D.12}$$

The continuous time control current driver approximation is,

$$A(s) = \frac{A_1}{s + A_2} \tag{D.13}$$

Substituting Eq. (D.11), Eq. (D.12), and Eq. (D.13) into Eq. (D.10) yields,

$$\frac{X(s)}{R(s)} = \frac{-K\left(\frac{A_1}{s+A_2}\right)\left(\frac{P}{s^2-Q}\right)\left(\frac{e^{-Ts}s^2 + a_{22}s + a_{21}}{B\left(1-e^{-Ts}\right)}\right)}{-K\left(\frac{A_1}{s+A_2}\right)\left(\frac{P}{s^2-Q}\right)\left(\frac{e^{-Ts}s^2 + a_{22}s + a_{21}}{B\left(1-e^{-Ts}\right)}\right) - 1} \tag{D.14}$$

or,

$$\frac{X(s)}{R(s)} = \frac{A_1KP\left(e^{-Ts}s^2 + a_{22}s + a_{21}\right)}{B\left(1-e^{-Ts}\right)s^3 + \left[A_2B\left(1-e^{-Ts}\right) + A_1KPe^{-Ts}\right]s^2 + \left[a_{22}A_1KP - BQ\left(1-e^{-Ts}\right)\right]s + \left[a_{21}A_1KP - A_2BQ\left(1-e^{-Ts}\right)\right]} \quad \textbf{(D.15)}$$

## D.1.3 Poles of the Closed Loop Characteristic Equation

The closed loop characteristic equation can be written as

$$B\left(1-e^{-Ts}\right)s^3 + \left[A_2B + \left(A_1KP - A_2B\right)e^{-Ts}\right]s^2 + \left[\left(a_{22}A_1KP - BQ\right) + BQe^{-Ts}\right]s + \left[\left(a_{21}A_1KP - A_2BQ\right) + A_2BQe^{-Ts}\right]$$

If order to determine whether the system is stable, the roots of the characteristic equation must be found. This implies that the continuous time representation of time delay $e^{-Ts}$ must be represented in some other analogous form. One such analogous form is the Taylor series approximation [12],

$$e^{-Ts} = 1 - Ts + \frac{T^2s^2}{2!} - \frac{T^3s^3}{3!} + \ldots \quad \textbf{(D.16)}$$

The Taylor series expansion is an infinite series. The accuracy of this approximation is directly related to the number of terms used and this in turn is directly related to the time delay $T$. As the time delay decreases, the number of terms necessary for an accurate approximation also decreases.

Another analogous form is to approximate the time delay as a polynomial function,

$$e^{-Ts} \cong \frac{1}{\left[1 + (Ts/n)\right]^n} \quad \textbf{(D.17)}$$

In this form, the accuracy improves as the value of $n$ approaches infinity. However, as $n$ decreases, the slope of the associated polynomial decreases and therefore the function reaches its final value at an ever increasing time. Obviously, the final value must be reached before the next sampling interval so once again the value of $n$ is tied to the time delay or sampling rate.

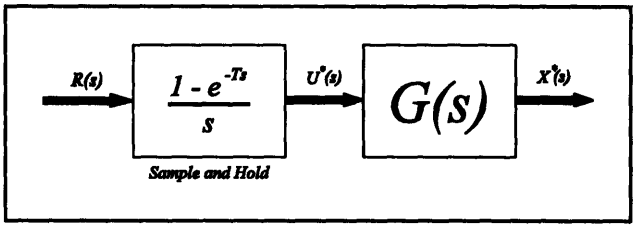
Both approximations are therefore tied to the sampling rate. However, the sampling rate of the digital controller is a controller variable which forces the approximation to be a variable. This would make the stability analysis unnecessarily complicated and therefore modeling the closed loop system as a purely continuous time system was dropped from consideration.

# D.2 Purely Digital System

All calculations were applied to the radial bearing equations. This same analysis also applies to the axial bearings but the results are not shown in the interest of brevity.

## D.2.1 Open Loop System

The block diagram of the discrete time open loop system is shown in Figure D-5. The linearized radial bearing equation represented in the block diagram by G(s) is,




**D-1** Discrete Time Open Loop Block Diagram

$$\dot{v}_{x1} = \frac{4C_1kI_0^2}{h_0^3}x_1 + \frac{4C_3kI_0^2}{h_0^3}x_2 - C_4\dot{y}_1 + C_4\dot{y}_2 + \frac{2C_1kI_0}{h_0^2}u_{x1} + \frac{2C_3kI_0}{h_0^2}u_{x2} \qquad \textbf{(D.18)}$$

Again assuming that the coupling between bearings can be treated as a disturbance ($x_2$ and $u_{x2}$ have little or no effect on $x_1$) and that the rotor is not rotating ($C_4 = 0$) [2, 13, 20]

$$\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{4C_1kI_0^2}{h_0^3} & 0 \end{bmatrix}\begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{2C_1kI_0}{h_0^2} \end{bmatrix}u \qquad \textbf{(D.19)}$$

$$[y] = \begin{bmatrix} 1 & 0 \end{bmatrix}\begin{bmatrix} x \\ v \end{bmatrix} + [0]u \qquad \textbf{(D.20)}$$

Taking the LaPlace transform,

$$\begin{aligned} sX^*(s) &= AX^*(s) + BU^*(s) \\ Y^*(s) &= DX^*(s) + EU^*(s) \end{aligned} \qquad \textbf{(D.21)}$$

$$\frac{X^*(s)}{T^*(s)} = D[sI - A]^{-1}B + E \qquad \textbf{(D.22)}$$

$$\frac{X^*(s)}{T^*(s)} = [1 \quad 0]\left\{\frac{1}{s^2 - \frac{4C_1kI_0^2}{h_0^3}}\begin{bmatrix} s & 1 \\ \frac{4C_1kI_0^2}{h_0^3} & s \end{bmatrix}\right\}\begin{bmatrix} 0 \\ \frac{2C_1kI_0}{h_0^2} \end{bmatrix} + [0] \tag{D.23}$$

$$\frac{X^*(s)}{T^*(s)} = \frac{\frac{2C_1kI_0}{h_0^2}}{s^2 - \frac{4C_1kI_0^2}{h_0^3}} = \frac{P}{s^2 - Q} \tag{D.24}$$

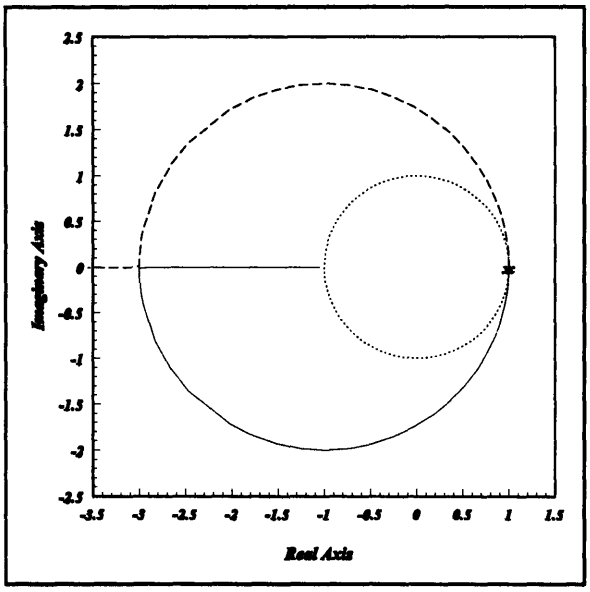
The z-transform of the open loop system is,

$$\begin{aligned}\left[\left(\frac{1-e^{-Ts}}{s}\right)G(s)\right]_z &= \left[\frac{(s)}{s}\right]_z - \left[\frac{e^{-Ts}G(s)}{s}\right]_z \\ &= \left[\frac{G(s)}{s}\right]_z - \left[e^{-Ts}\right]_z\left[\frac{G(s)}{s}\right]_z \\ &= \left[\frac{G(s)}{s}\right]_z - z^{-1}\left[\frac{G(s)}{s}\right]_z \\ &= \left(1-z^{-1}\right)\left[\frac{G(s)}{s}\right]_z\end{aligned}$$

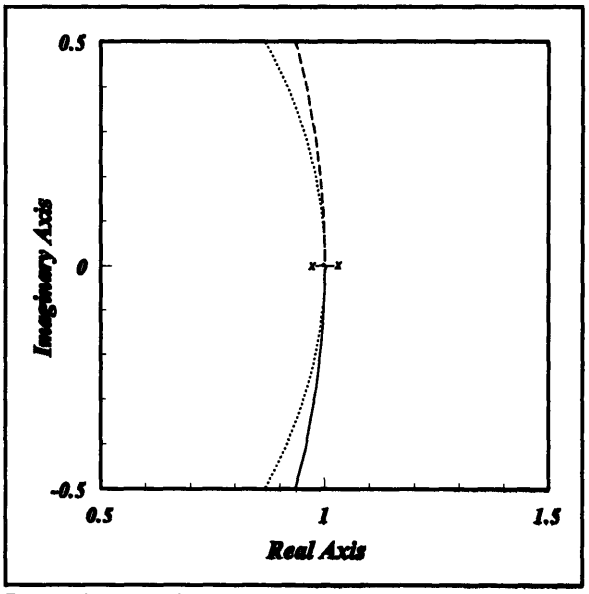
Using a backward difference variable transformation such that $s = \frac{(z-1)}{zT}$ yields,

$$\begin{aligned}\left(1-z^{-1}\right)\left[\frac{G(s)}{s}\right]_z &= \left(1-z^{-1}\right)\left[\frac{P}{s(s^2-Q)}\right]_z \\ &= \left(\frac{z-1}{z}\right)\left[\frac{K_1}{s} + \frac{K_2}{s+\sqrt{Q}} + \frac{K_3}{s-\sqrt{Q}}\right]_z \\ &= \left(\frac{z-1}{z}\right)\left[\frac{K_1z}{z-1} + \frac{K_2z}{z-e^{-\sqrt{Q}T}} + \frac{K_3z}{z-e^{\sqrt{Q}T}}\right] \\ &= \frac{P}{Q}\left[\frac{-2\left(z-e^{-\sqrt{Q}T}\right)\left(z-e^{\sqrt{Q}T}\right)+(z-1)\left(z-e^{\sqrt{Q}T}\right)+(z-1)\left(z-e^{-\sqrt{Q}T}\right)}{2\left(z-e^{\sqrt{Q}T}\right)\left(z-e^{\sqrt{Q}T}\right)}\right]\end{aligned} \tag{D.25}$$

For discrete time systems, stability is assured if poles of the characteristic equation lie within the unit circle. Figure D-6 shows that for a particular proportional gain, the system becomes marginally stable when both poles equal 1. However, the system is unstable in the abscence of any control because one pole lies outside the unit circle and proportional control can only briefly bring the system to marginal stability.

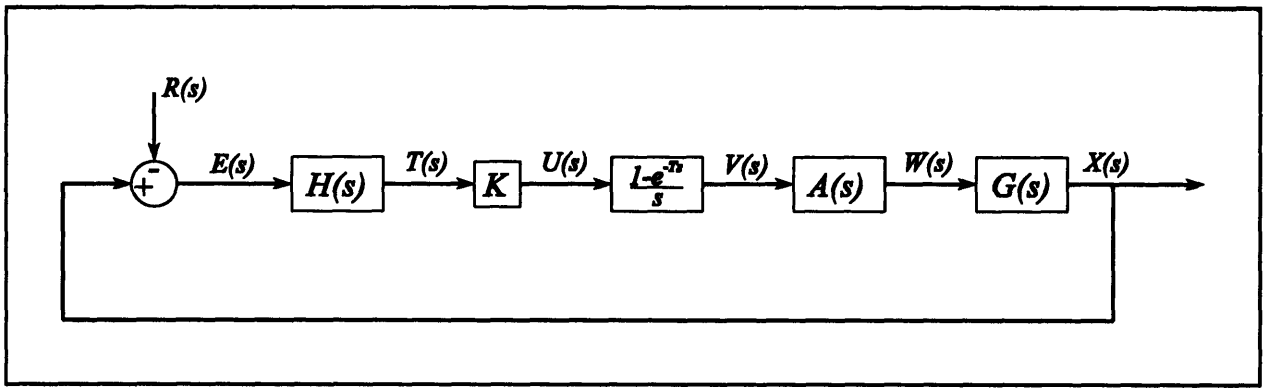



D-6 Discrete Time Open Loop Root Locus Diagram




D-7 Discrete Time Open Loop Root Locus Diagram

## D.2.2 Closed Loop System




D-8 Discrete Time Closed Loop Block Diagram

The block diagram of our closed loop system is shown in Figure D-8 above. **H(s)** represents the digital controller, **K** represents the control gain, $\frac{(1-e^{-Ts})}{s}$ represents an idealized sample and hold element, **A(s)** represents the control current driver which is responsible for generating the bias current and dividing the control current between both sides of the bearing axis, and **G(s)** represents the turbopump. To further simplify the analysis, the sample and hold, amplifier, and the turbopump are combined,

$$F(s) = \frac{(1-e^{-Ts})A_1 P}{s(s+A_2)(s^2-Q)} \qquad \textbf{(D.26)}$$

The z-transform of the above is calculated using the same process employed in our open loop equation analysis.

$$
\begin{aligned}
F(z) &= (1-z^{-1})\left[\frac{A_1 P}{s(s+A_2)(s^2-Q)}\right]_z \\
&= \left(\frac{z-1}{z}\right)\left[\frac{K_1}{s}+\frac{K_2}{s+A_2}+\frac{K_3}{s+\sqrt{Q}}+\frac{K_4}{s-\sqrt{Q}}\right]_z \\
&= \left(\frac{z-1}{z}\right)\left[\frac{K_1 z}{z-1}+\frac{K_2 z}{z-e^{-A_2 T}}+\frac{K_3 z}{z-e^{-\sqrt{Q}T}}+\frac{K_4 z}{z-e^{\sqrt{Q}T}}\right] \\
&= \frac{\begin{array}{l}K_1(z-e^{-A_2T})(z-e^{-\sqrt{Q}T})(z-e^{\sqrt{Q}T})+K_2(z-1)(z-e^{-\sqrt{Q}T})(z-e^{\sqrt{Q}T}) \\ +K_3(z-1)(z-e^{-A_2T})(z-e^{\sqrt{Q}T})+K_4(z-1)(z-e^{-A_2T})(z-e^{-\sqrt{Q}T})\end{array}}{(z-e^{-A_2T})(z-e^{-\sqrt{Q}T})(z-e^{\sqrt{Q}T})}
\end{aligned}
\qquad \textbf{(D.27)}
$$

From the block diagram, the individual block equations are,

$$
\begin{aligned}
E(z) &= X(z) - R(z) \\
T(z) &= H(z)E(z) \\
U(z) &= KT(z) \\
X(z) &= F(z)U(z)
\end{aligned}
\qquad \textbf{(D.28)}
$$

Rearranging yields

$$
\frac{X(z)}{R(z)} = \frac{KH(z)F(z)}{KH(z)F(z)-1} \qquad \textbf{(D.29)}
$$

The digital controller receives only position input and therefore must approximate both the velocity $v(t)$, $v(t\text{-}T)$, and the acceleration $a(t\text{-}T)$. The backward difference velocity approximation is,

$$
v(t) = \frac{3x(t)-4x(t-T)+x(t-2T)}{2T}+O(T)^2 \qquad \textbf{(D.30)}
$$

The central difference velocity approximation is,

$$v(t-T) = \frac{x(t)-x(t-2T)}{2T} + O(T)^2 \tag{D.31}$$

The central difference acceleration approximation is,

$$\begin{aligned} a(t-T) &= \frac{v(t)-v(t-2T)}{2T} + O(T)^2 \\ &= \frac{2x(t)-4x(t-T)-2x(t-2T)+4x(t-3T)-x(t-4T)}{4T^2} + O(T)^2 \end{aligned} \tag{D.32}$$

Using Eq. (D.31) and Eq. (D.32), the digital controller transfer function becomes,

$$\begin{aligned} H(z) &= \frac{\frac{-1}{4T^2}\left[2\left(3a_{22}T+2a_{21}T^2+1\right)z^4-4\left(1+2a_{22}T\right)z^3+2\left(a_{22}T-1\right)z^2+4z-1\right]}{Bz^3(z-1)} \\ &= \frac{-\left(M_1z^4-M_2z^3+M_3z^2+M_4z-M_5\right)}{Bz^3(z-1)} \end{aligned} \tag{D.33}$$

Substituting Eq. (D.27) and Eq. (D.33) into Eq. (D.29) yields,

$$\frac{X(z)}{R(z)} = \frac{-K\left(\frac{A_1PT^4z^3}{\left[(1+A_2T)z-1\right]\left[(1-QT^2)z^2-2z+1\right]}\right)\left(\frac{M_1z^4-M_2z^3+M_3z^2+M_4z-M_5}{Bz^3(z-1)}\right)}{-K\left(\frac{A_1PT^4z^3}{\left[(1+A_2T)z-1\right]\left[(1-QT^2)z^2-2z+1\right]}\right)\left(\frac{M_1z^4-M_2z^3+M_3z^2+M_4z-M_5}{Bz^3(z-1)}\right)-1} \tag{D.34}$$

or,

$$\begin{aligned} \frac{X(z)}{R(z)} = \frac{A_1KPT^4\left(M_1z^4-M_2z^3+M_3z^2+M_4z-M_5\right)}{\left[A_1KPT^4M_1-B\left(1+A_2T\right)\right]z^4-\left[A_1KPT^4M_2+3B\left(1+A_2T\right)+B\left(1-QT^2\right)\right]z^3} \\ +\left[A_1KPT^4M_3+3B\left(4+3A_2T\right)-B\left(1-QT^2\right)\right]z^2+\left(A_1KPT^4M_4+A_2BT+4B\right)z-\left(A_1KPT^4M_5+B\right) \end{aligned} \tag{D.35}$$

# D.3 Derivatives and Noise

In conventional calculus, differentiation of a function is a well-defined formal procedure that is highly dependent on the form of the function. Many different rules and techniques are employed for different functions. Digital computers however can only use the simple instructions of addition, subtraction, multiplication, and division along with some logical operations to determine

the derivative of a function. Therefore a technique which employs only these simple instructions to calculate function derivatives is needed. Such a technique is known as finite difference calculus and is employed by the control algorithm of our particular digital controller. There are however drawbacks to using derivatives in a control algorithm [3]. Differentiators are noise amplifiers. For instance, assume that a position signal that has a noise component is being differentiated,

$$x(t) = \sin t + 10^{-3}\sin 10^3 t$$

In this particular example, the high frequency noise component has an amplitude that is one thousand times smaller then the actual position signal. Now taking the derivative,

$$v(t) = \cos t + \cos 10^3 t$$

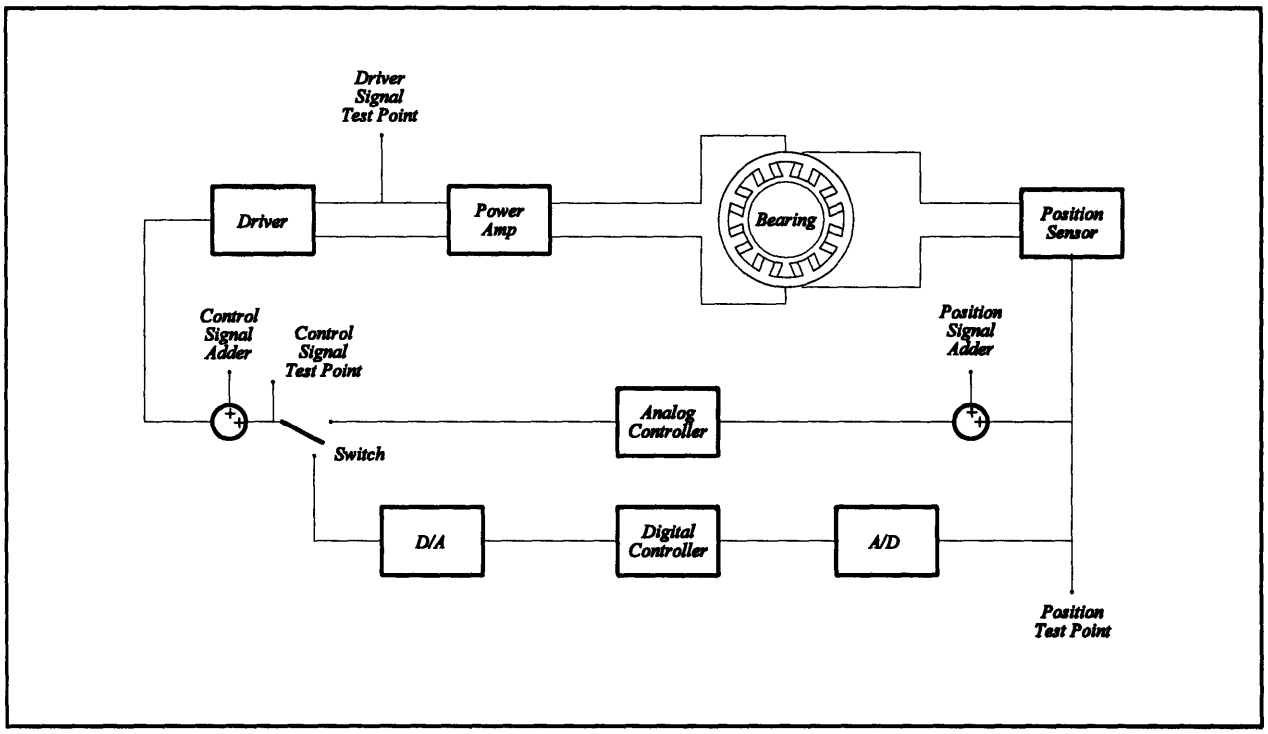
Differentiating the signal has magnified the amplitude of the noise component a thousand times. Now taking the second derivative,

$$a(t) = -\sin t - 10^3 \sin 10^3 t$$

Now the high frequency noise component has overwhelmed the actual signal. This exact scenario is happening in the digital controller used by this application. The state variables of the controller are position and velocity but the physical system is only capable of providing the position signal. Therefore the controller computes the missing velocity signal by differentiating the position signal using finite difference calculus. This velocity term is in turn differentiated to obtain the acceleration which is needed to determine the control signal. It is extremely likely that this acceleration term is primarily noise and therefore severely compromises the robustness of this controller.

# Appendix E

# System Analysis Data




**E-1** System Block Diagram and Test Points

This appendix presents graphically the results of the system analysis of different components of the turbopump. The actual system analysis data was obtained using the Hewlett Packard HP 3562A Dynamic System Analyzer across the appropriate test points. Refer to Figure E-1 for the location of the test points in relation to the system components. Also included are the source listings of the programs used to obtain the numerical values associated with each transfer function numerator and denominator.

# E.1 Theoretical versus Actual Pump Transfer Function

The theoretical pump transfer function is obtained from the linearized equations of motion. These equations are further simplified by assuming that any coupling between axes of each bearing is negligible and therefore can be treated as a disturbance, and that the rotor is not spinning and therefore all gyroscopic terms are zero. Using these assumptions, the transfer functions for each respective type of bearing is,

Axial

$$\frac{X(s)}{U(s)} = \frac{\frac{\mu_0 N^2 A}{2mh_0^2}}{s^2 - \frac{\mu_0 N^2 A}{2mh_0^3}}$$

Radial

$$\frac{X(s)}{U(s)} = \frac{\frac{2\mu_0 N^2 A C_1 \cos\beta I_0}{h_0^2}}{s^2 - \frac{4\mu_0 N^2 A C_1 \cos\beta I_0^2}{h_0^3}}$$

The table below represents the important magnetic bearing variable values as provided by the turbopump manufacturer.

| Parameter | Axial | Radial |
|---|---|---|
| Air Permeability, $\mu_0$ ($N/A^2$) | $1.26 \times 10^{-6}$ | $1.26 \times 10^{-6}$ |
| Mass, m (Kg) | 2.2 | 2.2 |
| Number of Wire Turns, N | 133 | 100 |
| Magnetic Flux Area, A ($m^2$) | $7.0 \times 10^{-4}$ | $9.75 \times 10^{-5}$ |
| Bias Current (A) | 0.5 | 0.5 |
| Centerline Distance, $h_0$ (m) | $4.0 \times 10^{-4}$ | $2.5 \times 10^{-4}$ |

To simplify the analysis, the transfer functions representing linearized equations of motion of all the magnetic bearings will share the same format,

$$\frac{X(s)}{U(s)} = \frac{P}{s^2 - Q}$$

The representative values of $P$ and $Q$ are,

| Parameter | Axial | Rad1X | Rad1Y | Rad2X | Rad2Y |
|---|---|---|---|---|---|
| P | 22.162 | 9.496 | 9.496 | 18.720 | 18.720 |
| Q | 55403.757 | 37984.062 | 37984.062 | 74881.133 | 74881.133 |

The actual transfer function is obtained using the Dynamic System Analyzer. The input was a swept sine wave having a range of 0.1 Hz to 10 KHz. The sine wave is incremented linearly and has an amplitude of 0.1 Volts. The input was applied to the driver signal test point and the output was obtained from the position test point (see Figure E-1). The actual transfer function encompasses not only the magnetic bearings but also the power amplifier. However, the theoretical transfer function is derived from the linearized rotor equations of motion and therefore ignores the power amplifier. The contribution of the power amplifier was determined by comparing the DC gains of the actual and theoretical transfer functions. The power amplifier is assumed to be constant gain amplifier having the form $A_1/A_2$. The values obtained for the power amplifier are,

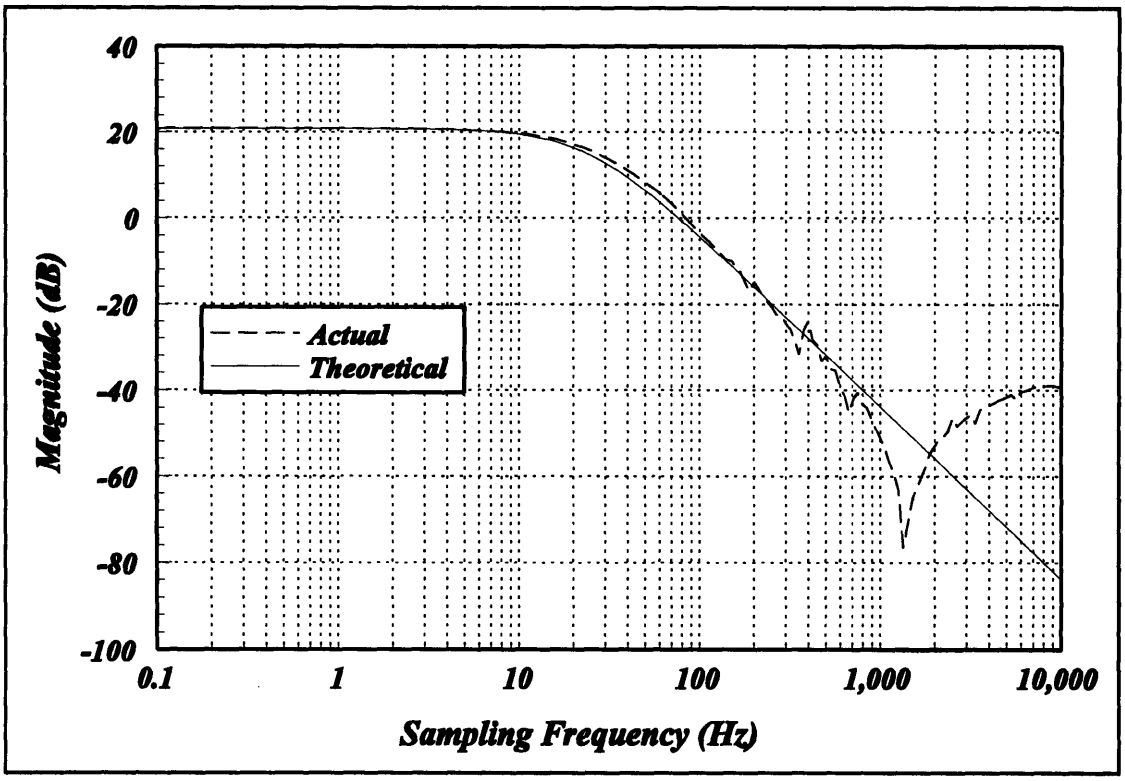
| Parameter | Axial | Rad1X | Rad1Y | Rad2X | Rad2Y |
|---|---|---|---|---|---|
| $A_1$ | 1.000 | 3.682 | 3.736 | 1.906 | 1.821 |
| $A_2$ | 1.138 | 1.000 | 1.000 | 1.000 | 1.000 |

Finally, either the actual transfer function data or the theoretical transfer function data must be converted to compatible units for comparison purposes. The actual transfer function has units of V/V and the theoretical transfer function data has units of m/A. The conversion factors are,

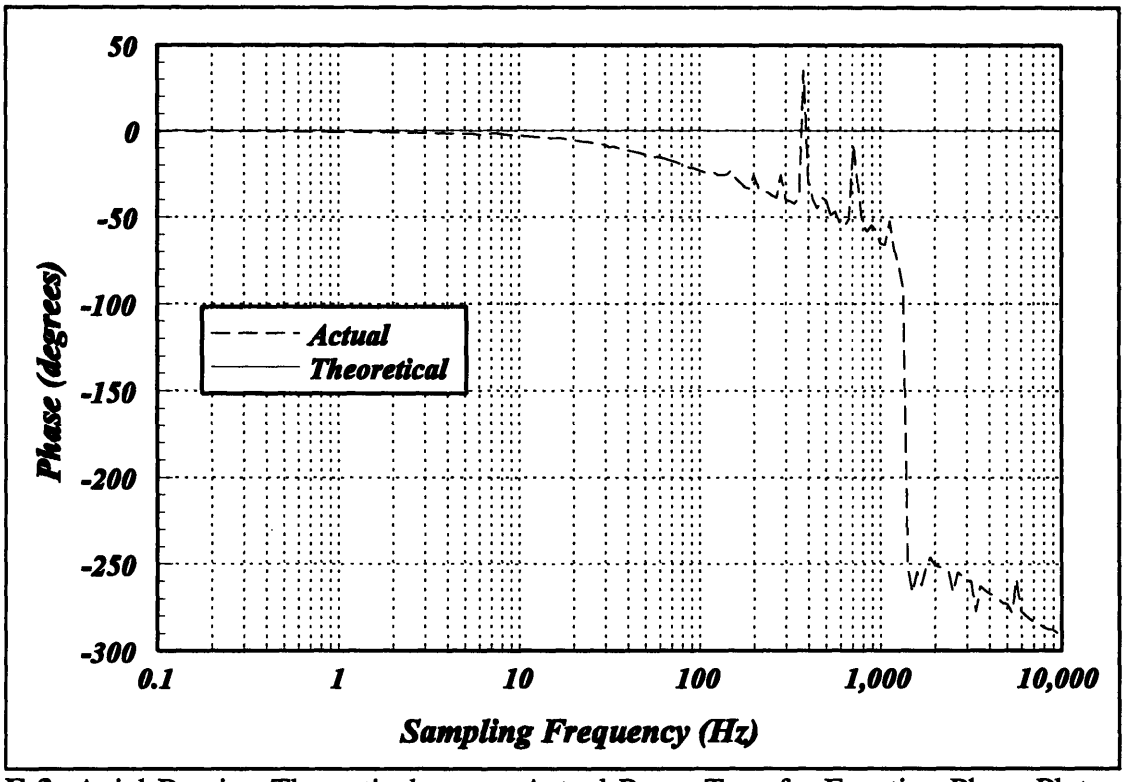
| Conversion Factor | Axial | Rad1X | Rad1Y | Rad2X | Rad2Y |
|---|---|---|---|---|---|
| Position Signal (V/m) | 9450.0 | 25000.0 | 25000.0 | 25000.0 | 25000.0 |
| Driver Signal (V/A) | 0.3 | 0.3 | 0.3 | 0.3 | 0.3 |

The actual transfer function versus theoretical transfer function plots begin on the next page. Note that the axial bearing phase plots seem to disagree with the value expected from the transfer function. This discrepancy is due to the normal operation of the driver. In the case of the radial bearings, the control signal at the equilibrium point is such that the driver produces magnetic coil currents that are both positive and negative in magnitude. The driver test point that corresponded to the positive coil current was used to obtain the actual pump transfer function plots so that the

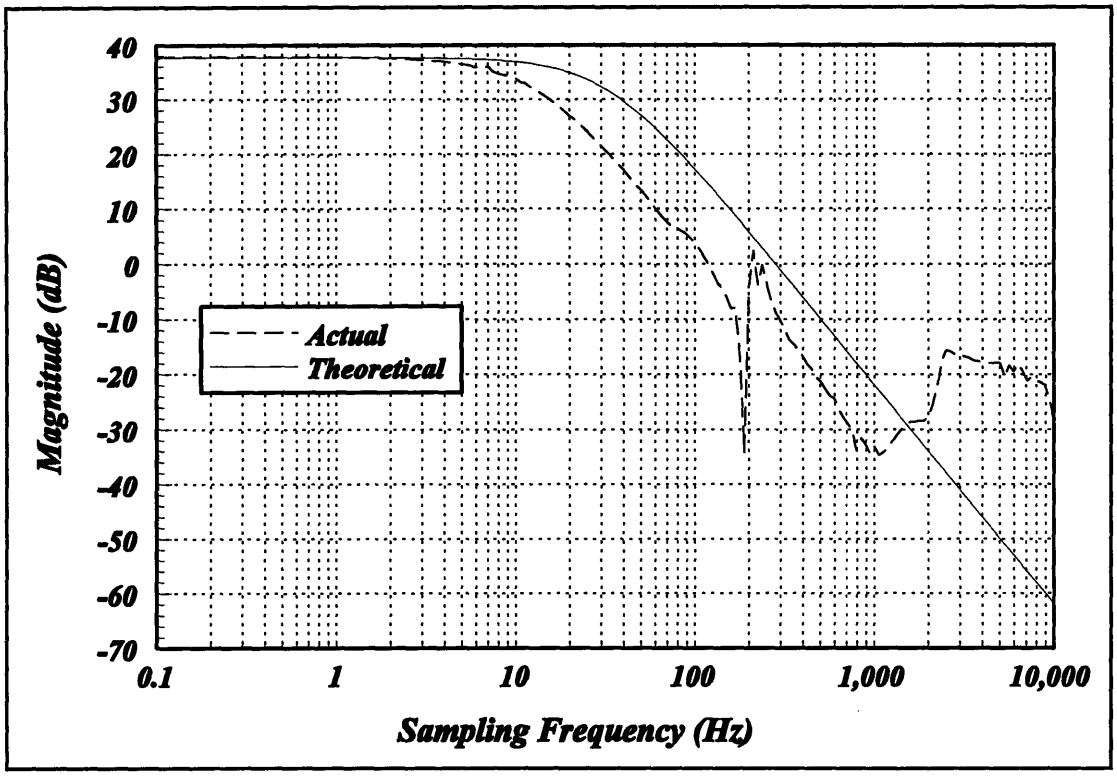
phase plots would not require correction. However, the axial bearing driver only produces a negative magnetic coil current when the bearing is in its equilibrium position. Therefore, the numerator of the theoretical transfer function was inverted to obtain the proper phase.



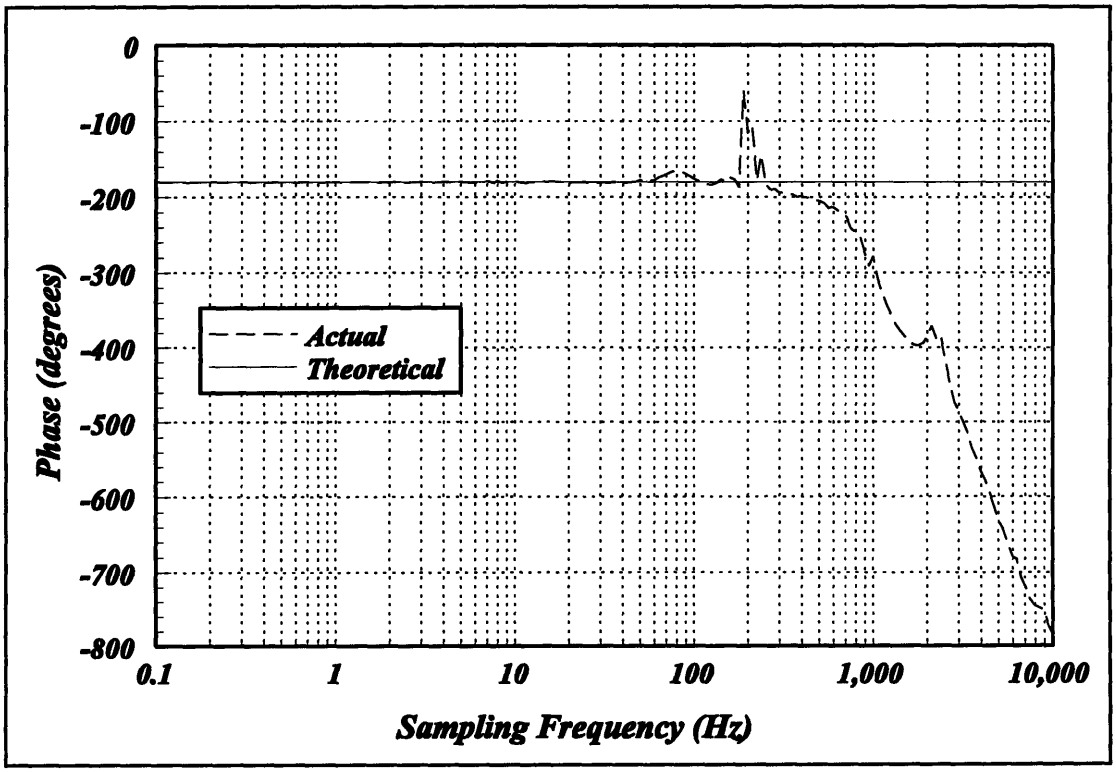

**E-2** Axial Bearing Theoretical versus Actual Pump Transfer Function Magnitude Plot



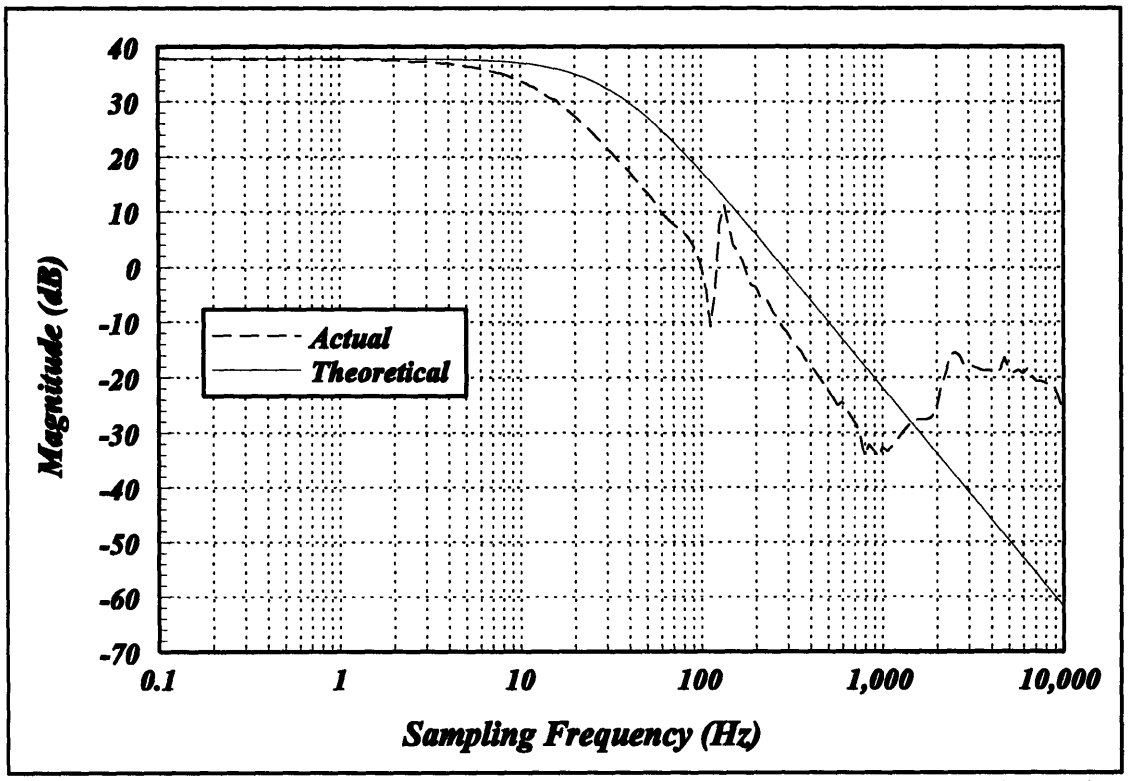

**E-3** Axial Bearing Theoretical versus Actual Pump Transfer Function Phase Plot



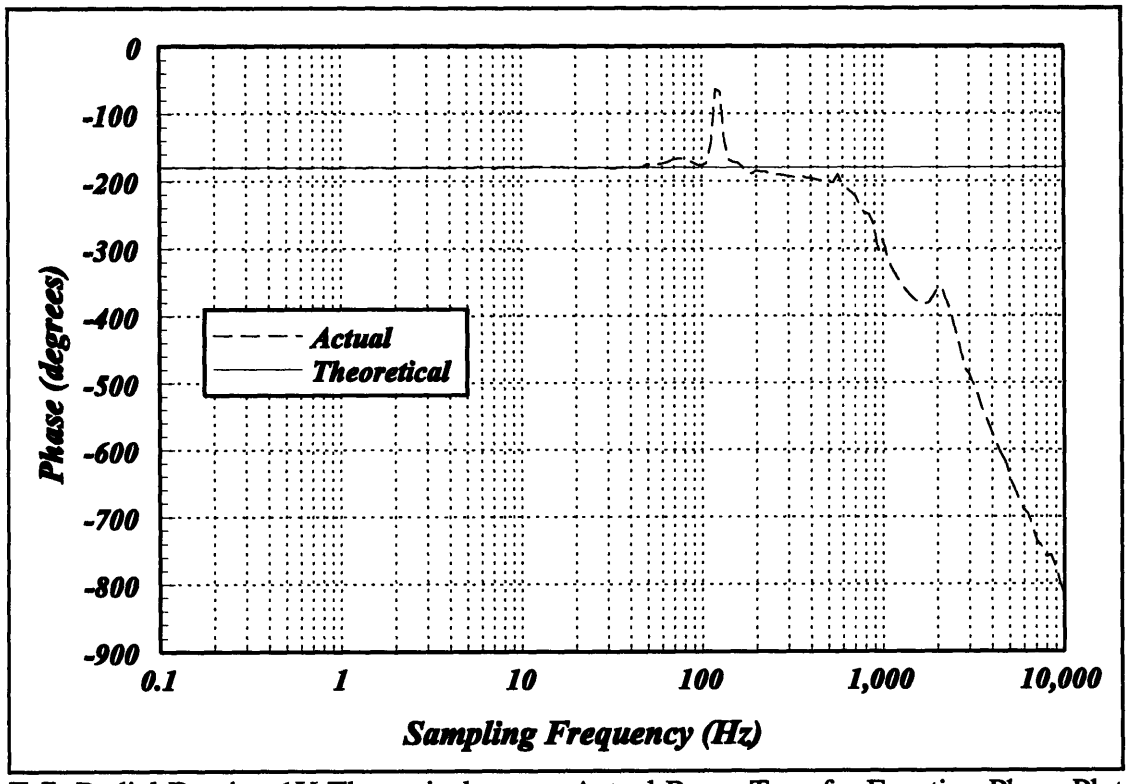

**E-4** Radial Bearing 1X Theoretical versus Actual Pump Transfer Function Magnitude Plot



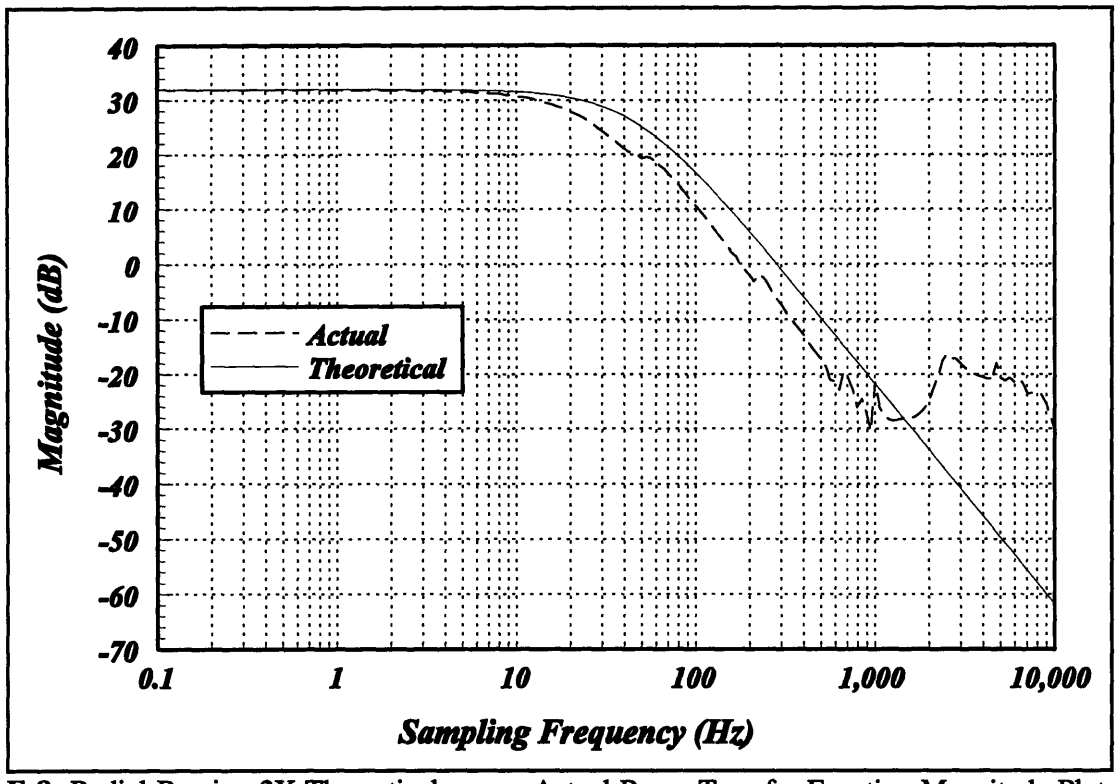

**E-5** Radial Bearing 1X Theoretical versus Actual Pump Transfer Function Phase Plot



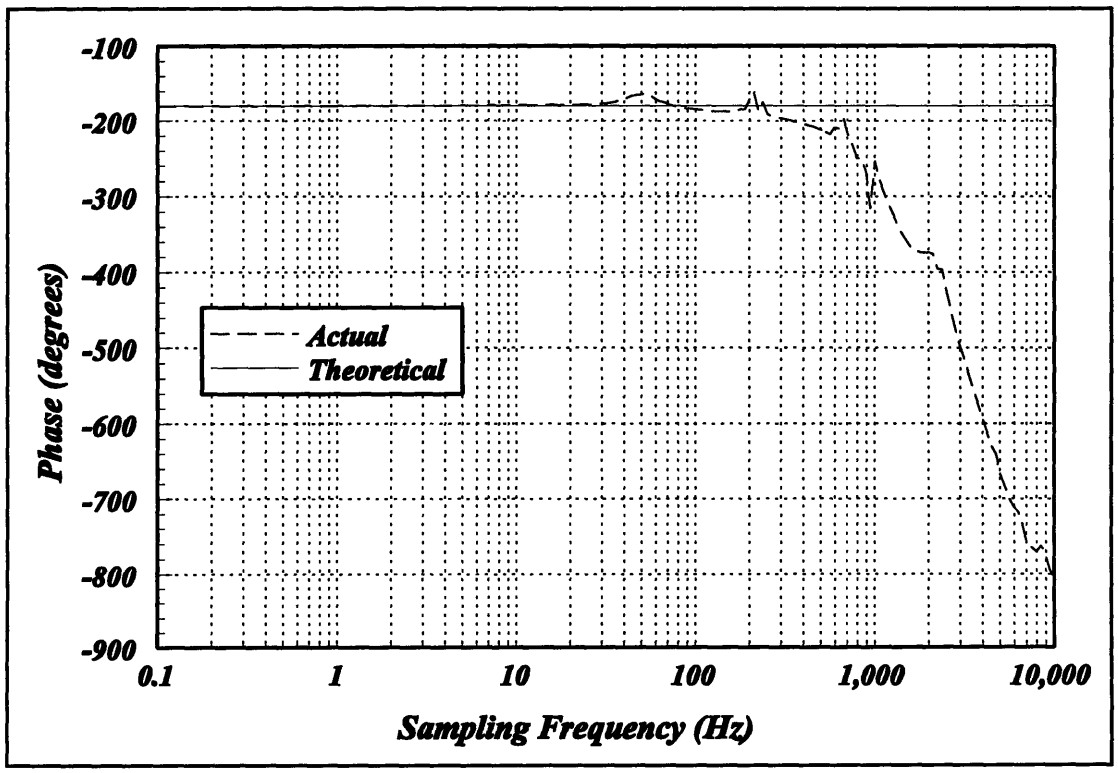

**E-6** Radial Bearing 1Y Theoretical versus Actual Pump Transfer Function Magnitude Plot



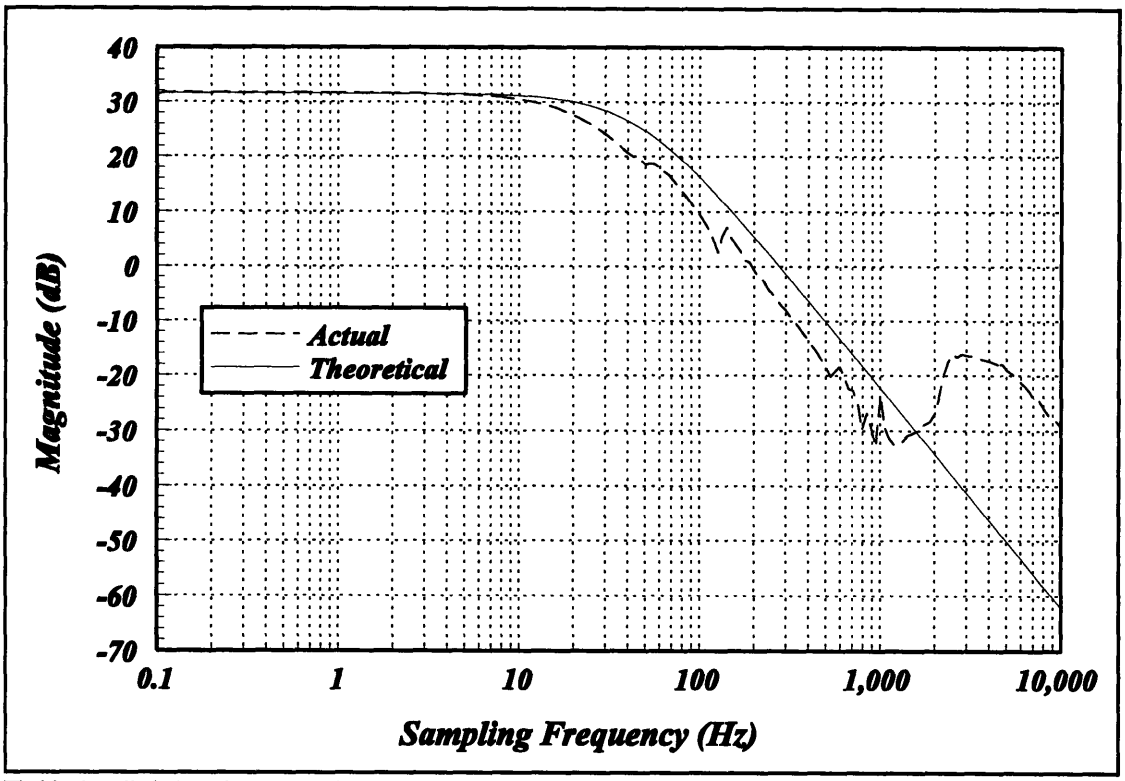

**E-7** Radial Bearing 1Y Theoretical versus Actual Pump Transfer Function Phase Plot



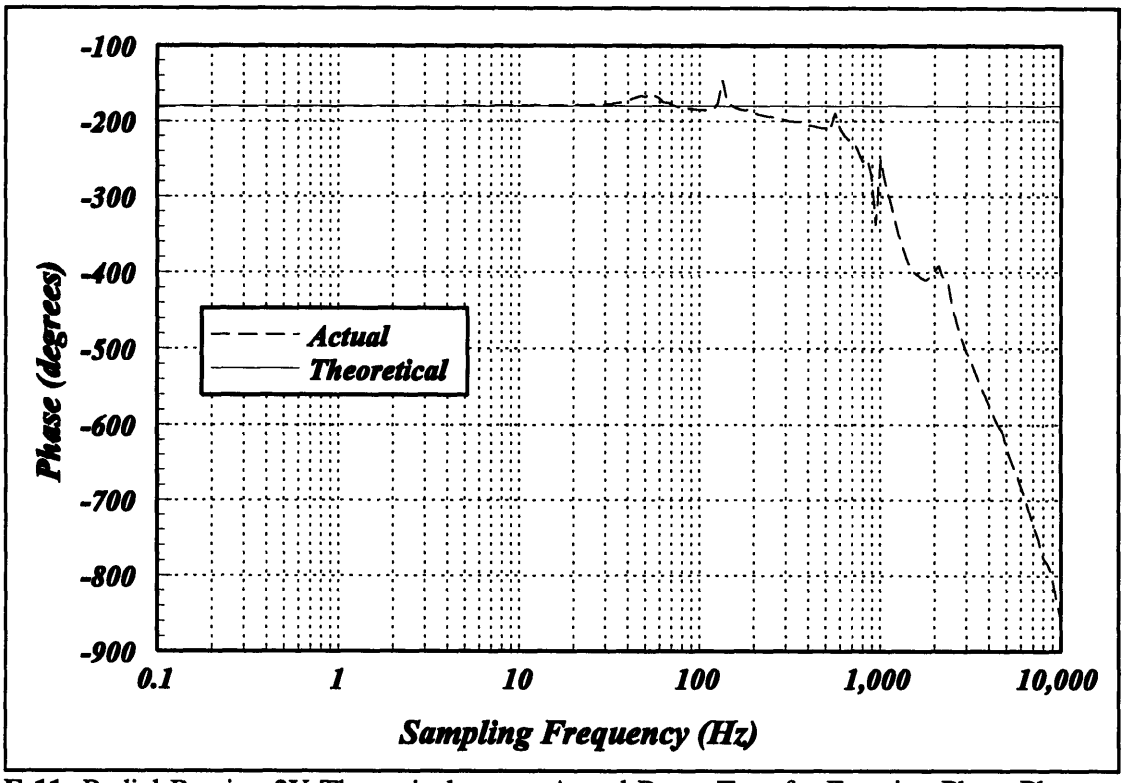

**E-8** Radial Bearing 2X Theoretical versus Actual Pump Transfer Function Magnitude Plot




**E-9** Radial Bearing 2X Theoretical versus Actual Pump Transfer Function Phase Plot




**E-10** Radial Bearing 2Y Theoretical versus Actual Pump Transfer Function Magnitude Plot




**E-11** Radial Bearing 2Y Theoretical versus Actual Pump Transfer Function Phase Plot

## E.2 Best Fit versus Actual Pump Transfer Function

The best fit pump transfer function is derived by using the general form of the theoretical pump transfer function and recursively trying different values for the numerator and the denominator. The general form is,

$$\frac{X(s)}{U(s)} = \frac{P}{s^2 - Q}$$

For each recursive attempt, initially $P = 1.0$ and a different value of $Q$ is used. The bode plot of this guess is calculated and $P$ is updated so that the DC gains of this guess and the actual bode plot are equal. The data points of the bode plot of each guess and the data points of the actual bode plot are taken at the same frequency values. The square of the difference between the current guess data points and the actual data points is calculated over the desired frequency range of the best fit model. These error values are summated for both the magnitude and the phase plots to produce the cumulative error for each guess. The guess that produces the least cumulative error is deemed the best fit pump transfer function.

The validity of the best fit analysis depends upon frequency range, guess value range, and iteration granularity. The frequency range of this particular best fit analysis is between 0.1 and 1000 Hz. The range was limited to 1000 Hz because the actual pump bode plot displays large phase and magnitude changes beyond this frequency indicative of a higher then second order system. Calculating the best fit transfer function over the entire frequency range would cause the transfer function to be less accurate at lower frequencies. The analysis was limited to 1000 Hz in order to provide the most accurate best fit transfer function within the probable bandwidth of the digital controller. By limiting the analysis to 1000 Hz, a digital controller derived using this transfer function must be constrained to provide adequate attenuation of the control signal after 1000 Hz.

Guess value range was based upon the form of the theoretical pump transfer function and the data from the actual pump transfer function plots. Using this information, the theoretical transfer function exhibits the following form,

$$\frac{X(s)}{U(s)} = \frac{K}{s^2 - \omega_n^2}$$

Therefore the guess value range is from 0.1 Hz to 10 KHz. Each value taken from this range is converted to rads/sec and squared to provide the next possible guess to be evaluated. By analyzing the actual pump transfer function plots, the range could have been reduced further but a conservative approach was taken.

Iteration granularity refers to the numerical difference between guesses. The finer the iteration granularity, the more possible guesses are evaluated and therefore the more accurate the best fit transfer function returned. However finer granularities increase computation time

significantly. This best fit analysis used variable iteration granularities based upon the guess value range. The iteration granularities are,

| | 0.1 - 1 Hz | 1 - 10 Hz | 10 - 100 Hz | 100 - 1000 Hz | 1000 - 10000 Hz |
|---|---|---|---|---|---|
| Iteration Granularity | 0.01 | 0.1 | 1.0 | 10.0 | 10.0 |

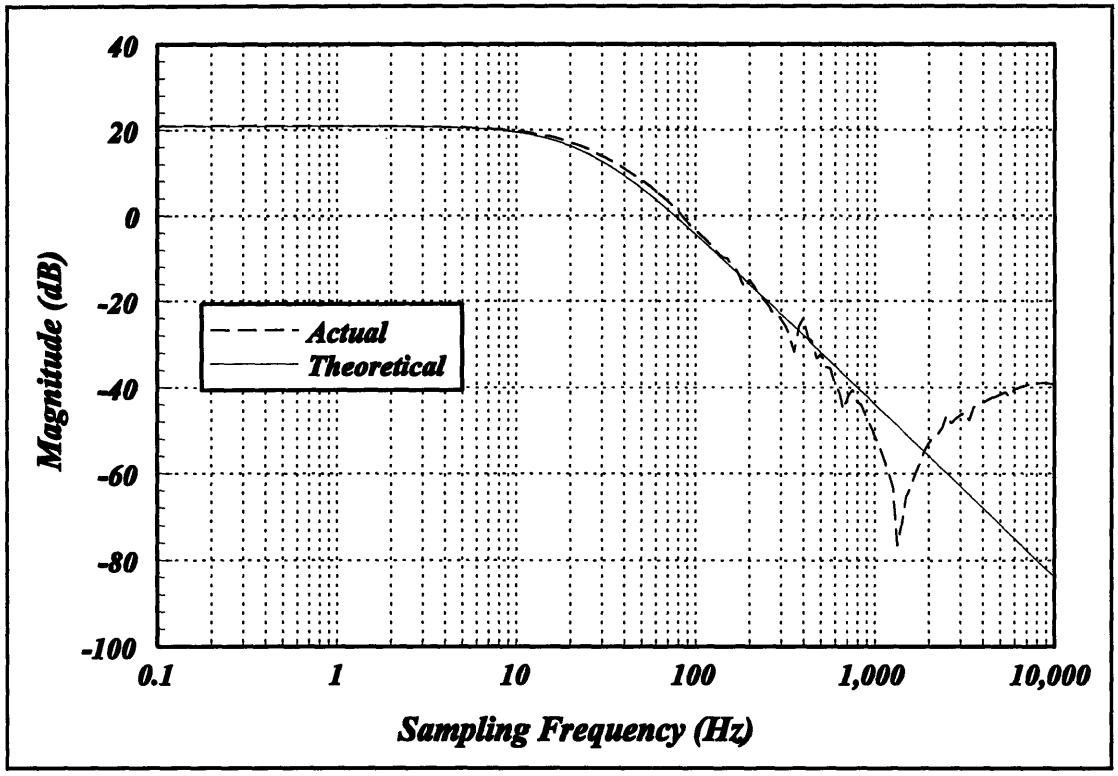
Using the process described previously, the representative values of $P$ and $Q$ are,

| Parameter | Axial | Rad1X | Rad1Y | Rad2X | Rad2Y |
|---|---|---|---|---|---|
| P | 7.990 | 7.123 | 8.296 | 16.926 | 15.113 |
| Q | 22739.568 | 7737.770 | 8882.644 | 35530.574 | 32201.348 |

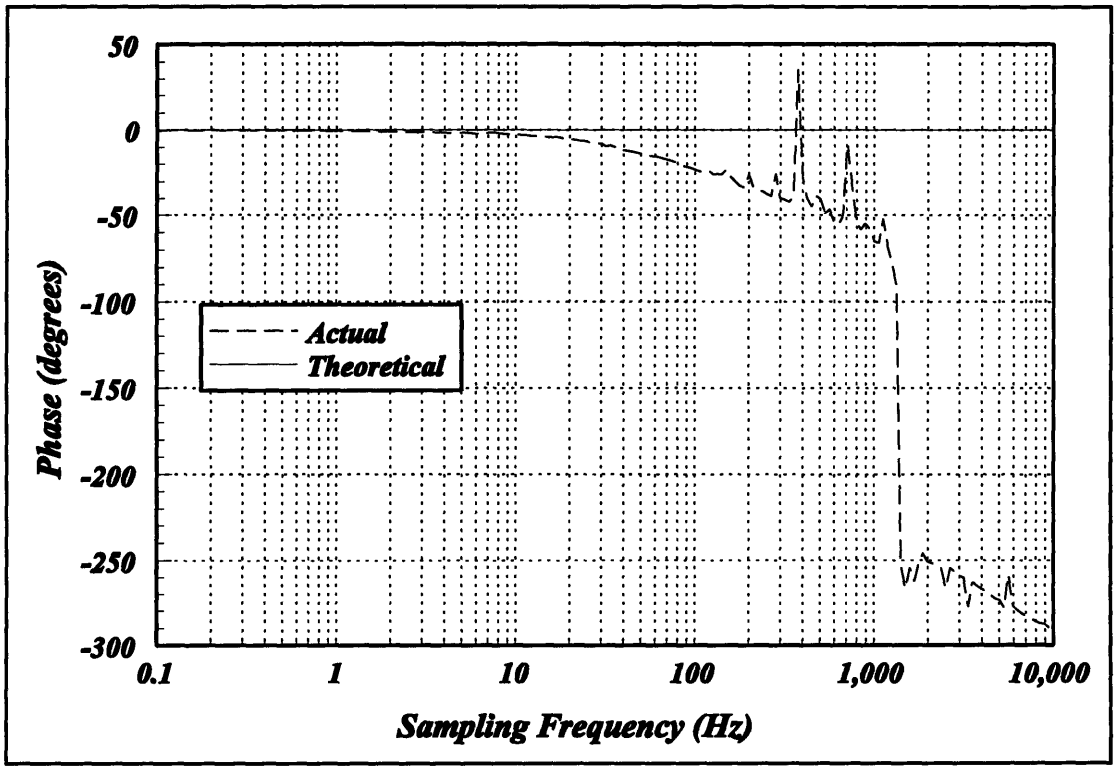
Finally, either the actual transfer function data or the best fit transfer function data must be converted to compatible units for comparison purposes. The actual transfer function has units of V/V and the best fit transfer function data has units of m/A. The conversion factors are,

| Conversion Factor | Axial | Rad1X | Rad1Y | Rad2X | Rad2Y |
|---|---|---|---|---|---|
| Position Signal (V/m) | 9450.0 | 25000.0 | 25000.0 | 25000.0 | 25000.0 |
| Driver Signal (V/A) | 0.3 | 0.3 | 0.3 | 0.3 | 0.3 |

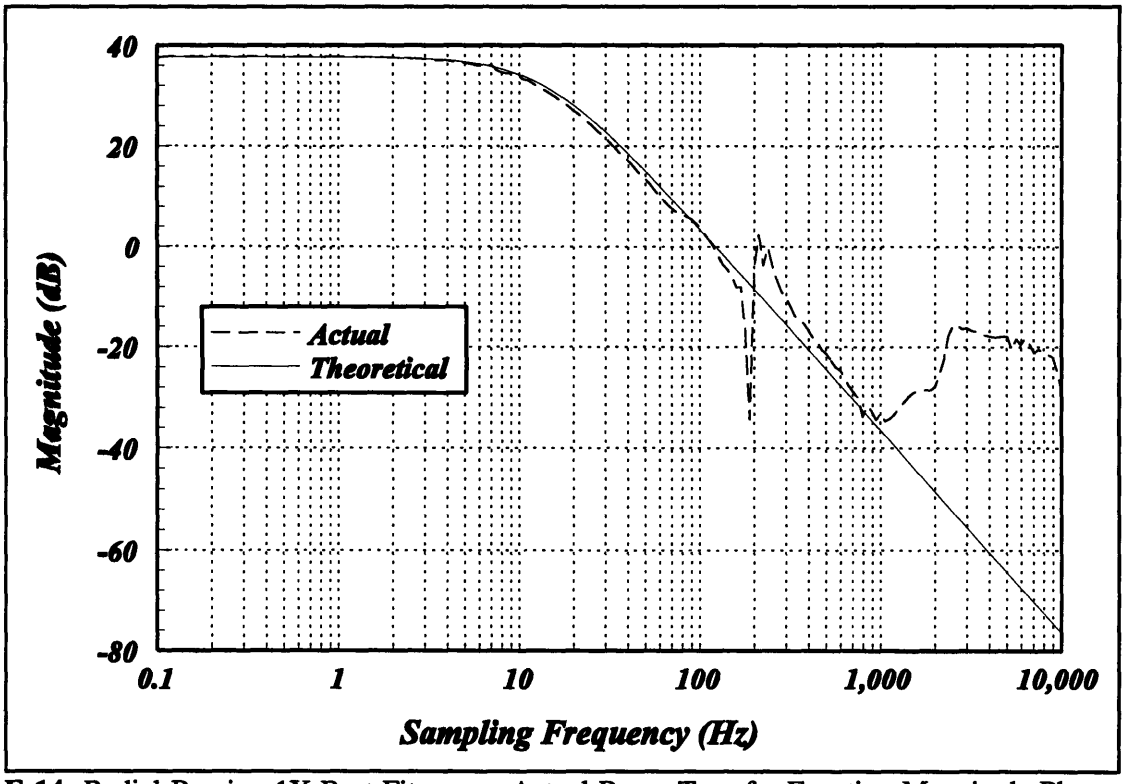
The actual transfer function versus best fit transfer function plots begin on the next page. Note that the axial bearing phase plots seem to disagree with the value expected from the transfer function. This discrepancy is due to the normal operation of the driver. In the case of the radial bearings, the control signal at the equilibrium point is such that the driver produces magnetic coil currents that are both positive and negative in magnitude. The driver test point that corresponded to the positive coil current was used to obtain the actual pump transfer function plots so that the phase plots would not require correction. However, the axial bearing driver only produces a negative magnetic coil current when the bearing is in its equilibrium position. Therefore, the numerator of the best fit transfer function was inverted to obtain the proper phase.



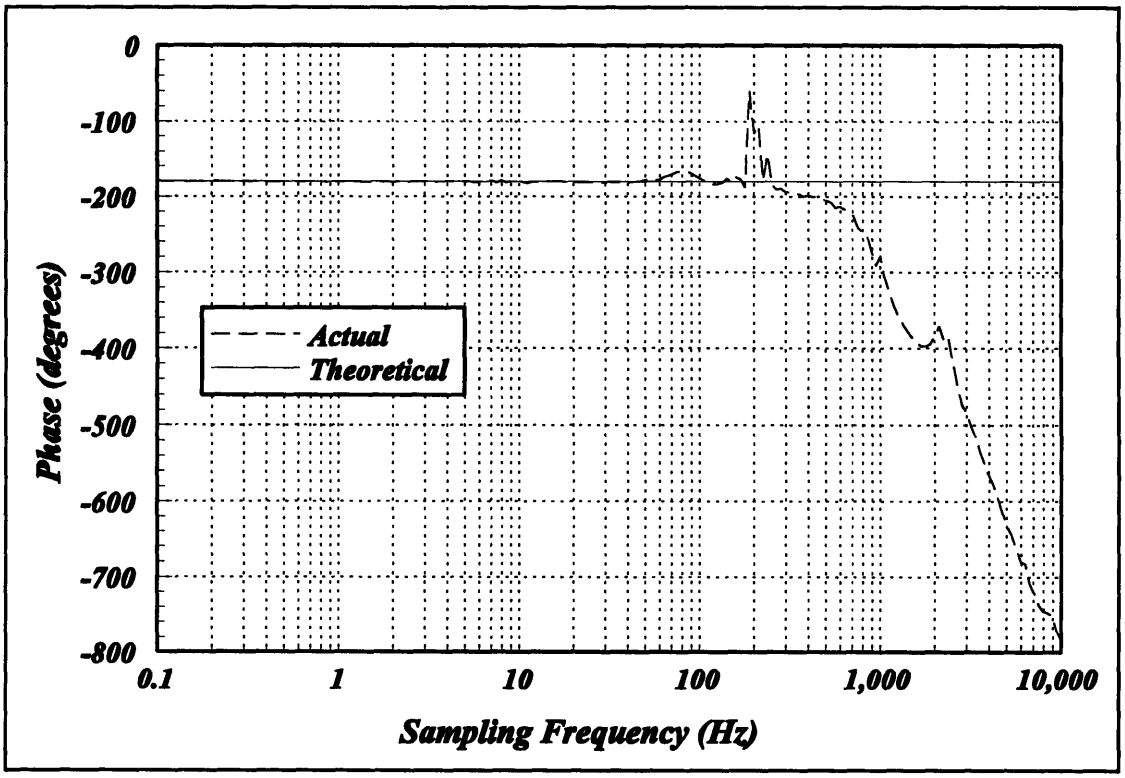

**E-12** Axial Bearing Best Fit versus Actual Pump Transfer Function Magnitude Plot



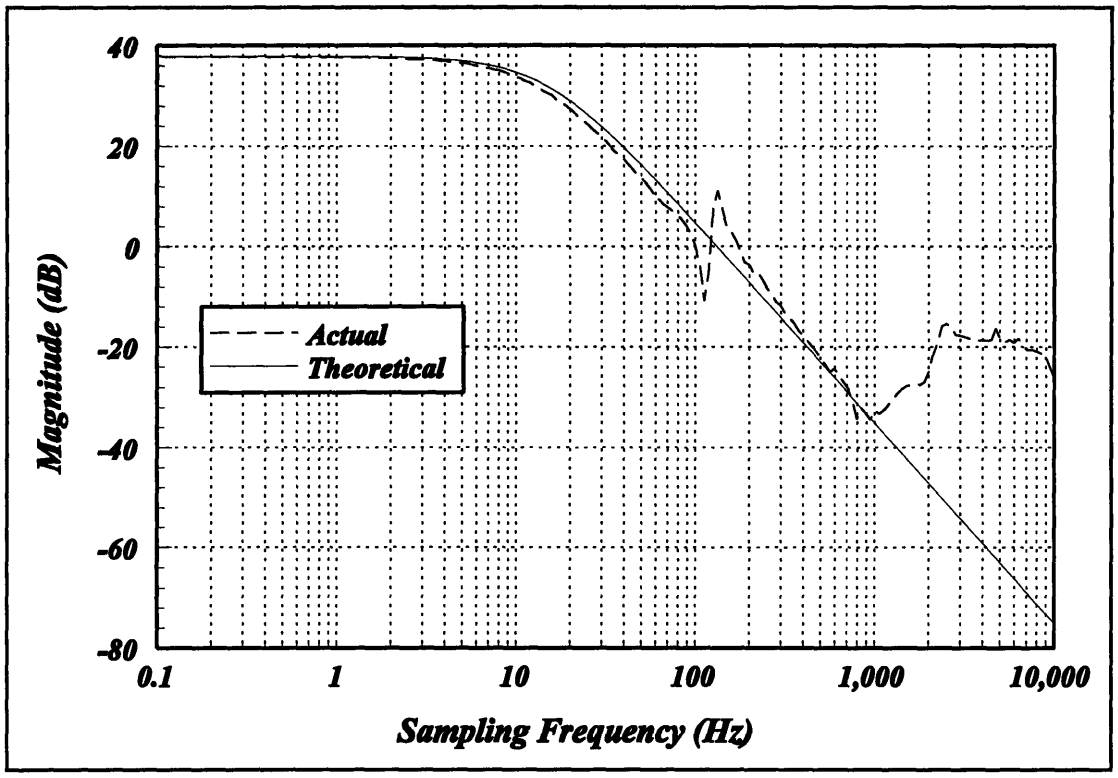

**E-13** Axial Bearing Best Fit versus Actual Pump Transfer Function Phase Plot



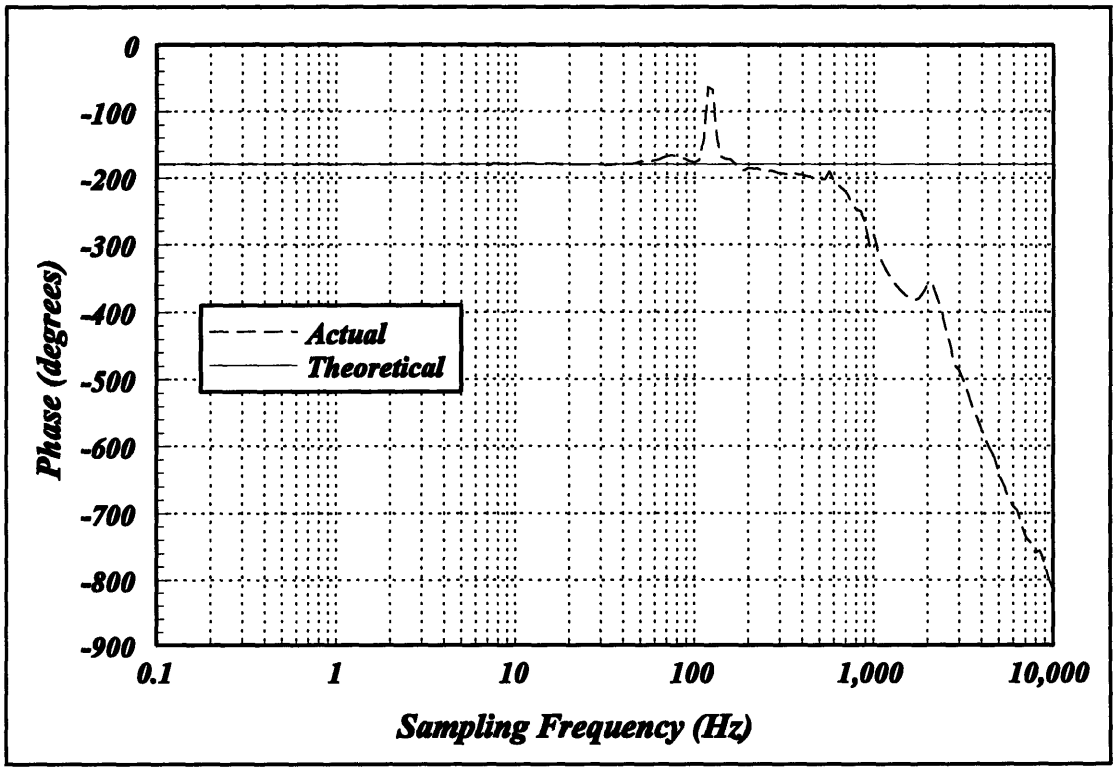

**E-14** Radial Bearing 1X Best Fit versus Actual Pump Transfer Function Magnitude Plot



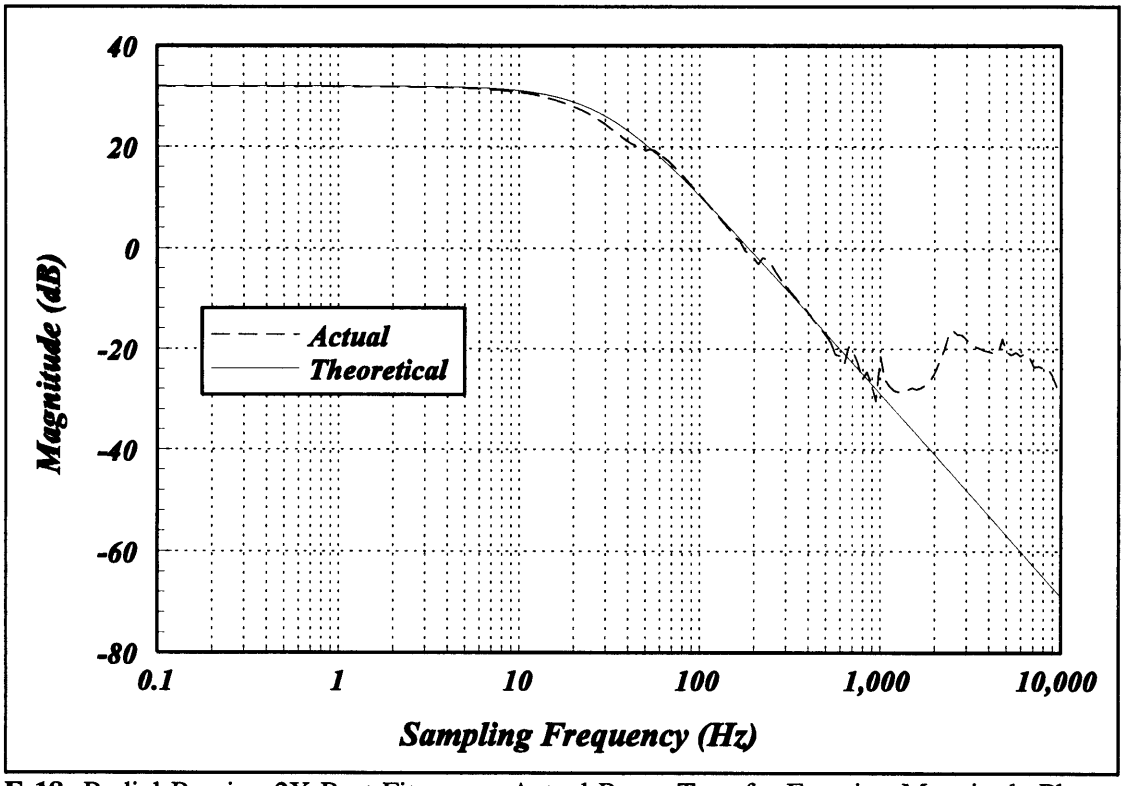

**E-15** Radial Bearing 1X Best Fit versus Actual Transfer Function Phase Plot



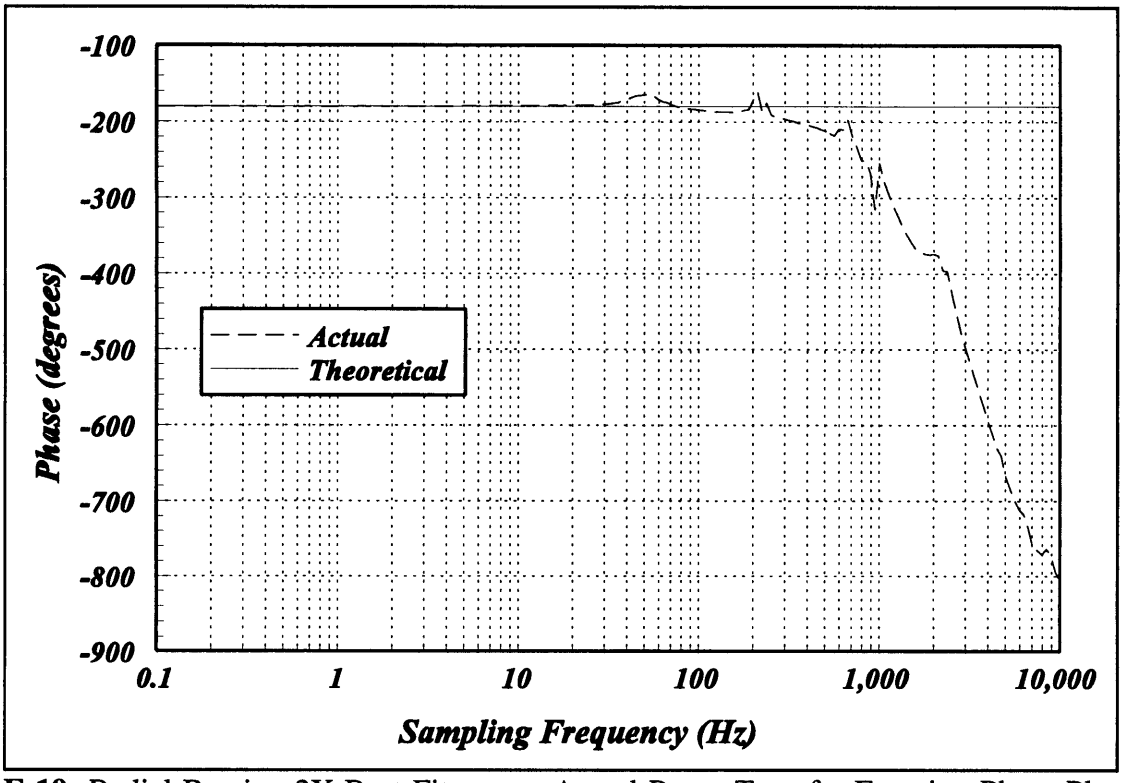

**E-16** Radial Bearing 1Y Best Fit versus Actual Pump Transfer Function Magnitude Plot



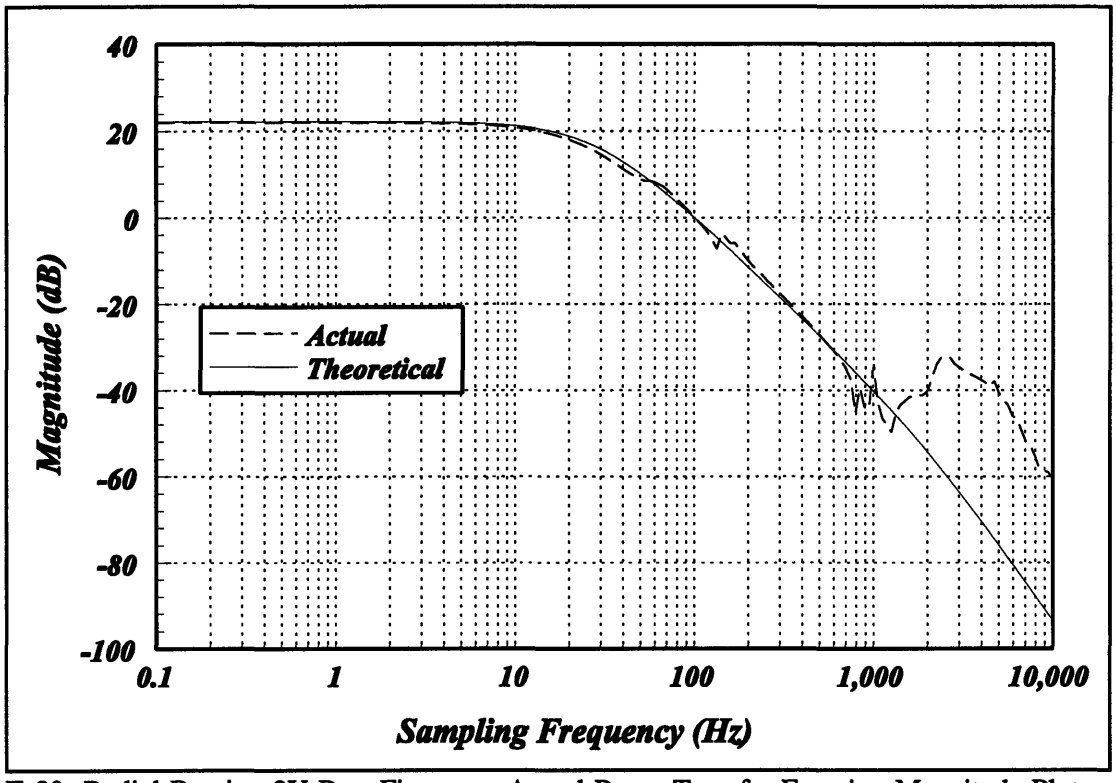

**E-17** Radial Bearing 1Y Best Fit versus Actual Pump Transfer Function Phase Plot



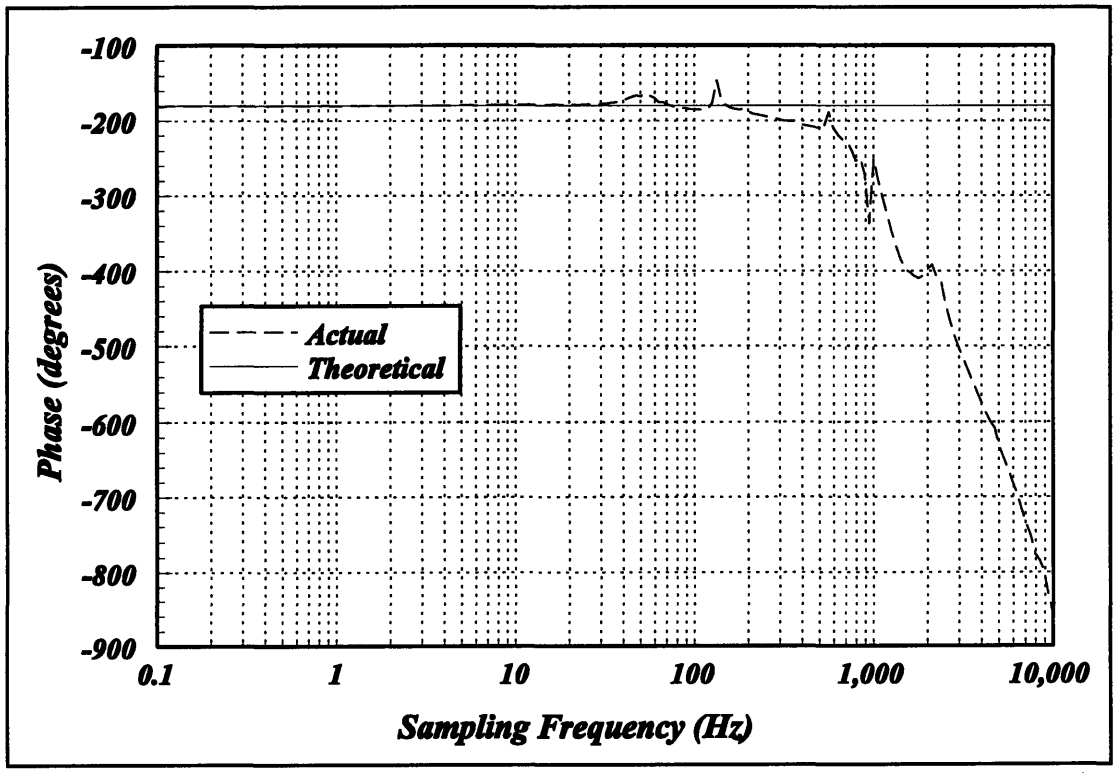

**E-18** Radial Bearing 2X Best Fit versus Actual Pump Transfer Function Magnitude Plot




**E-19** Radial Bearing 2X Best Fit versus Actual Pump Transfer Function Phase Plot




**E-20** Radial Bearing 2Y Best Fit versus Actual Pump Transfer Function Magnitude Plot




**E-21** Radial Bearing 2Y Best Fit versus Actual Pump Transfer Function Phase Plot

# E.3 Best Fit versus Actual Driver Transfer Function

By examining the magnitude and phase plots of the actual driver transfer function, the general form of the best fit driver transfer function is determined to be,

$$\frac{X(s)}{U(s)} = \frac{A_1}{s + A_2}$$

A brute force recursive method was used to determine the actual values of the numerator and denominator of the best fit transfer function. For each recursive attempt, initially $A_1 = 1.0$ and a different value of $A_2$ is used. The bode plot of this guess is calculated and $A_1$ is updated so that the DC gains of this guess and the actual bode plot are equal. The data points of the bode plot of each guess and the data points of the actual bode plot are taken at the same frequency values. The square of the difference between the current guess data points and the actual data points is calculated over the desired frequency range of the best fit model. These error values are summated for both the magnitude and the phase plots to produce the cumulative error for each guess. The guess that produces the least cumulative error is deemed the best fit driver transfer function.

The validity of the best fit analysis depends upon frequency range, guess value range, and iteration granularity. The frequency range of this particular best fit analysis is between 0.1 and 1000 Hz. The range was limited to 1000 Hz because the actual driver bode plot displays large phase and magnitude changes beyond this frequency indicative of a higher then second order system. Calculating the best fit transfer function over the entire frequency range would cause the transfer function to be less accurate at lower frequencies. The analysis was limited to 1000 Hz in order to provide the most accurate best fit transfer function within the probable bandwidth of the digital controller. By limiting the analysis to 1000 Hz, a digital controller derived using this transfer function must be constrained to provide adequate attenuation of the control signal after 1000 Hz.

The guess value range is from 0.1 Hz to 100 KHz. Each value taken from this range is converted to rads/sec and squared to provide the next possible guess to be evaluated. By analyzing the actual driver transfer function plots, the range could have been reduced further but a conservative approach was taken.

Iteration granularity refers to the numerical difference between guesses. The finer the iteration granularity, the more possible guesses are evaluated and therefore the more accurate the best fit transfer function returned. However finer granularities increase computation time significantly. This best fit analysis used variable iteration granularities based upon the guess value range. The iteration granularities are,

| | 1 - 10 Hz | 10 - 100 Hz | 0.1 - 1 KHz | 1 - 10 KHz | 10 - 100 KHz |
|---|---|---|---|---|---|
| Iteration Granularity | 0.1 | 1.0 | 10.0 | 10.0 | 10.0 |

Using the process described previously, the representative values of $A_1$ and $A_2$ are,

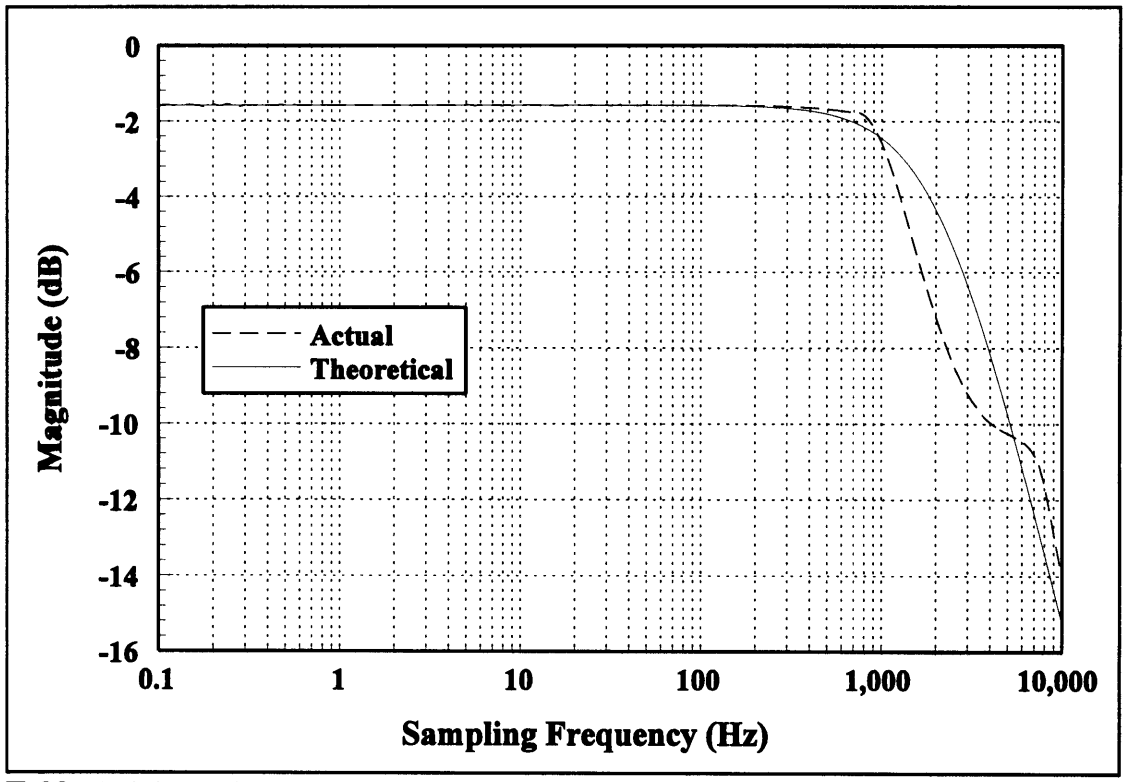
| Parameter | Axial | Rad1X | Rad1Y | Rad2X | Rad2Y |
|---|---|---|---|---|---|
| $A_1$ | 14707.770 | 13105.209 | 13043.913 | 11112.759 | 12273.613 |
| $A_2$ | 13310.000 | 13130.000 | 13080.000 | 11140.000 | 12290.000 |

Finally, either the actual transfer function data or the best fit transfer function data must be converted to compatible units for comparison purposes. The actual transfer function has units of V/V and the best fit transfer function data has units of A/A. The conversion factors are,

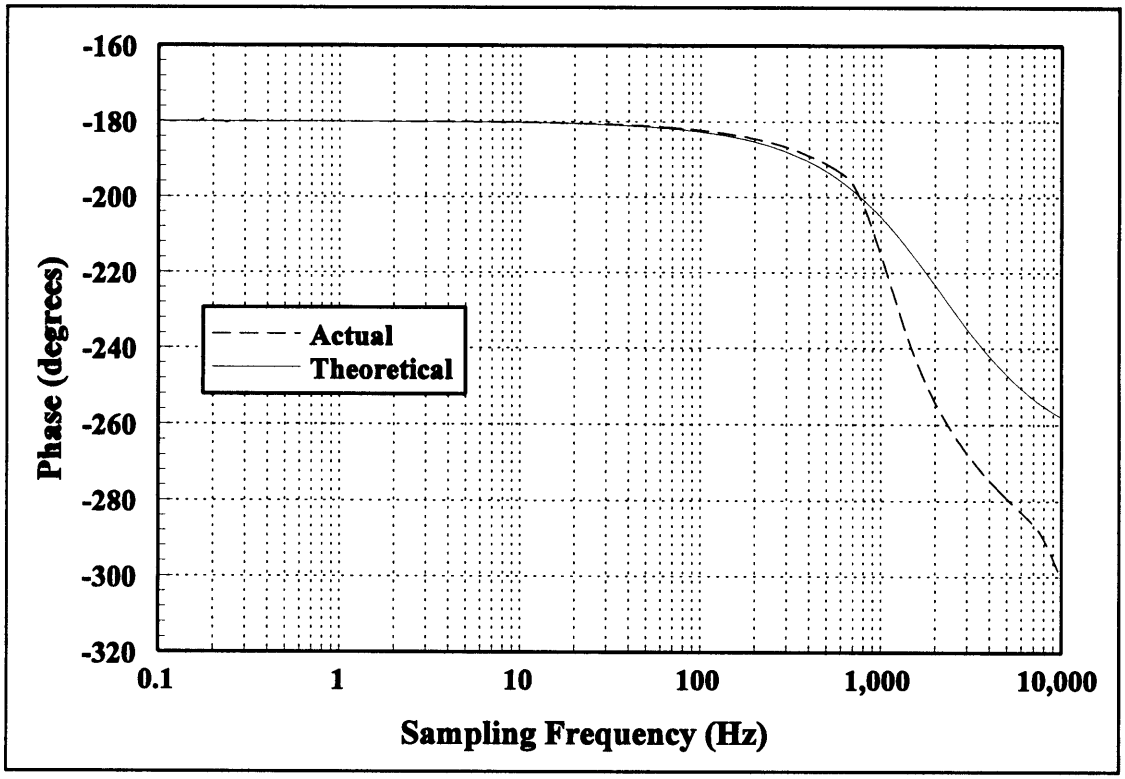
| Conversion Factor | Axial | Rad1X | Rad1Y | Rad2X | Rad2Y |
|---|---|---|---|---|---|
| Control Signal (A/V) | 2.776 | 1.175 | 1.165 | 1.159 | 1.144 |
| Driver Signal (V/A) | 0.3 | 0.3 | 0.3 | 0.3 | 0.3 |

The control signal conversion factors require some explanation. This conversion factor is used by the digital controller to convert the calculated control current to an appropriate value for the D/A converters which are voltage devices. Therefore, these conversion factors are merely the DC gains of the driver transfer functions. In the case of the radial bearings, the conversion factors are the averages of the DC gains from the two coil current control signals produced by the driver.

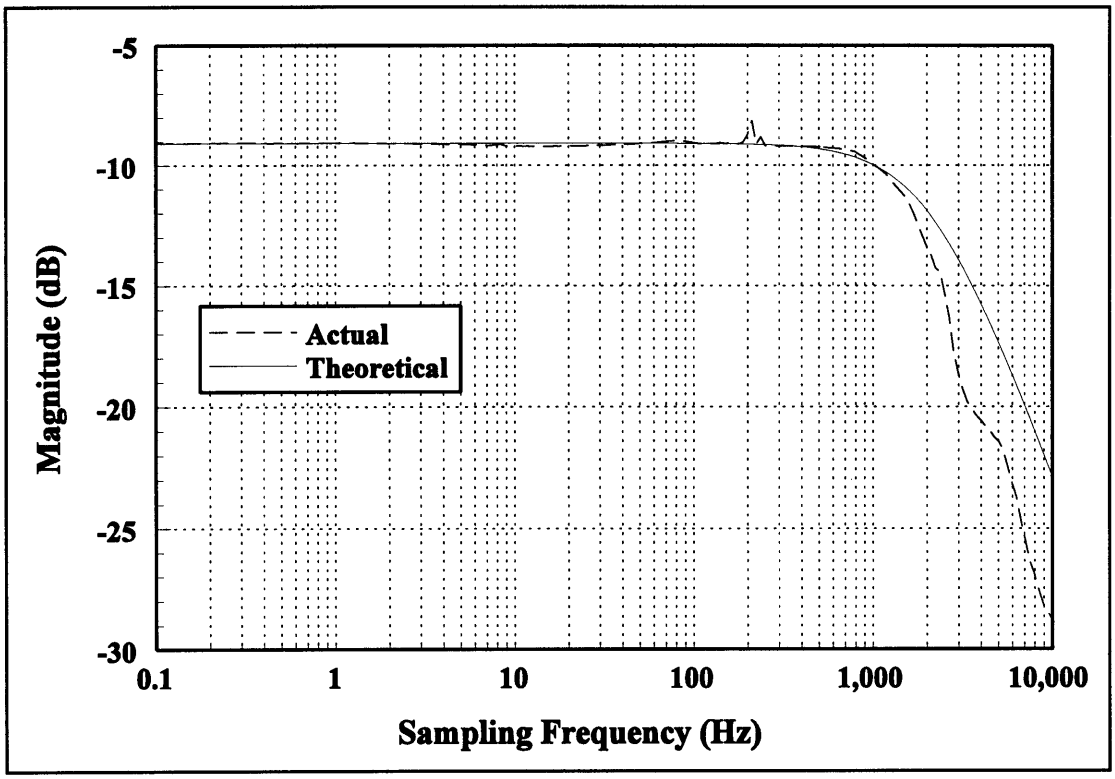
The actual transfer function versus best fit transfer function plots begin on the next page. Note that the axial bearing phase plots seem to disagree with the value expected from the transfer function. This discrepancy is due to the normal operation of the driver. In the case of the radial bearings, the control signal at the equilibrium point is such that the driver produces magnetic coil currents that are both positive and negative in magnitude. The driver test point that corresponded to the positive coil current was used to obtain the actual driver transfer function plots so that the phase plots would not require correction. However, the axial bearing driver only produces a negative magnetic coil current when the bearing is in its equilibrium position. Therefore, the numerator of the best fit transfer function was inverted to obtain the proper phase.



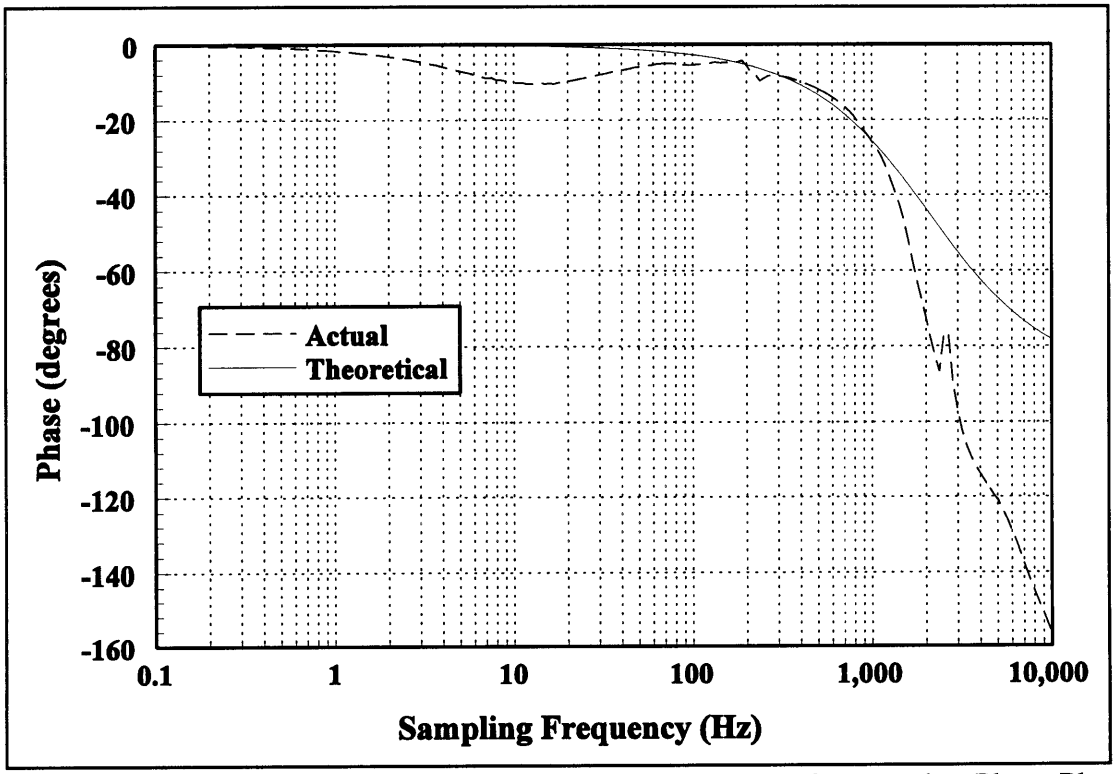

**E-22** Axial Bearing Best Fit versus Actual Driver Transfer Function Magnitude Plot



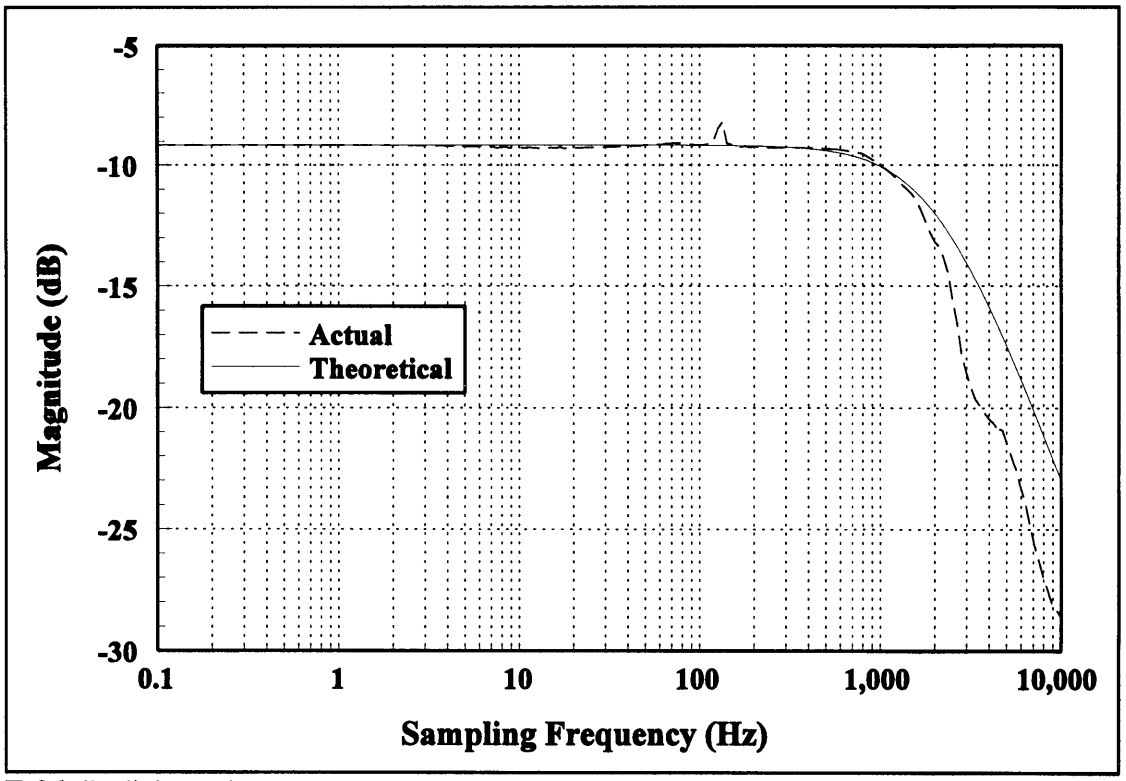

**E-23** Axial Bearing Best Fit versus Actual Driver Transfer Function Phase Plot



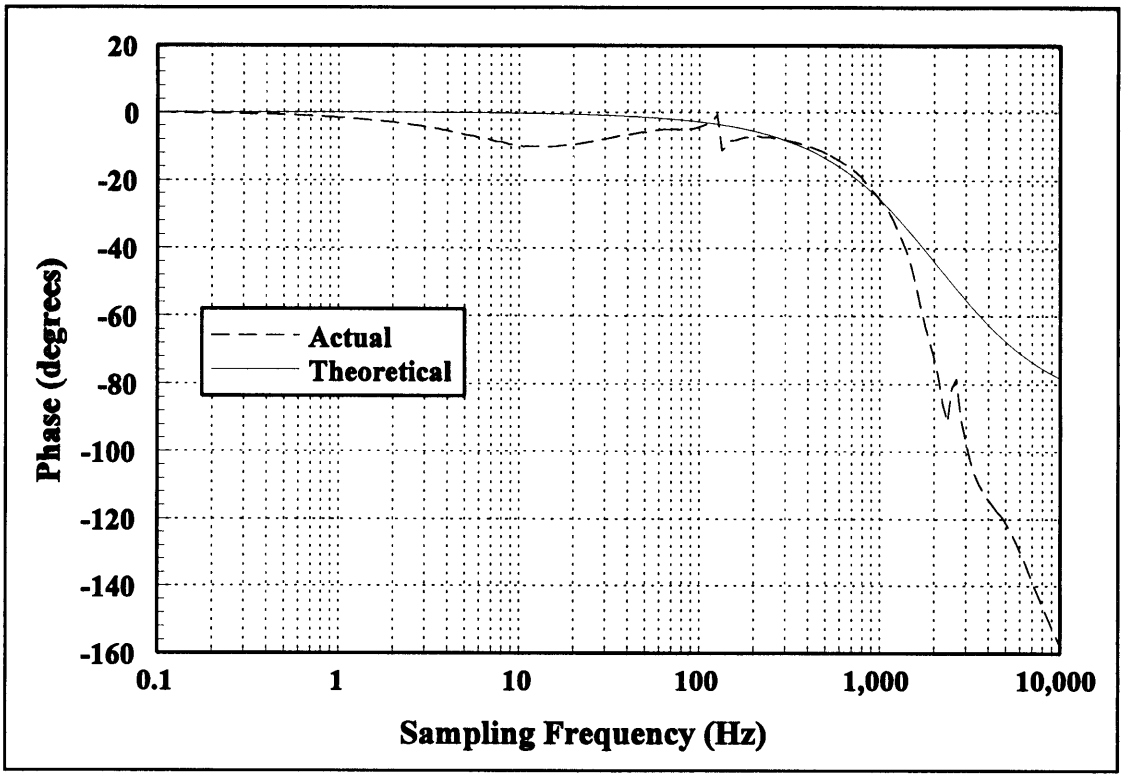

**E-24** Radial Bearing 1X Best Fit versus Actual Driver Transfer Function Magnitude Plot



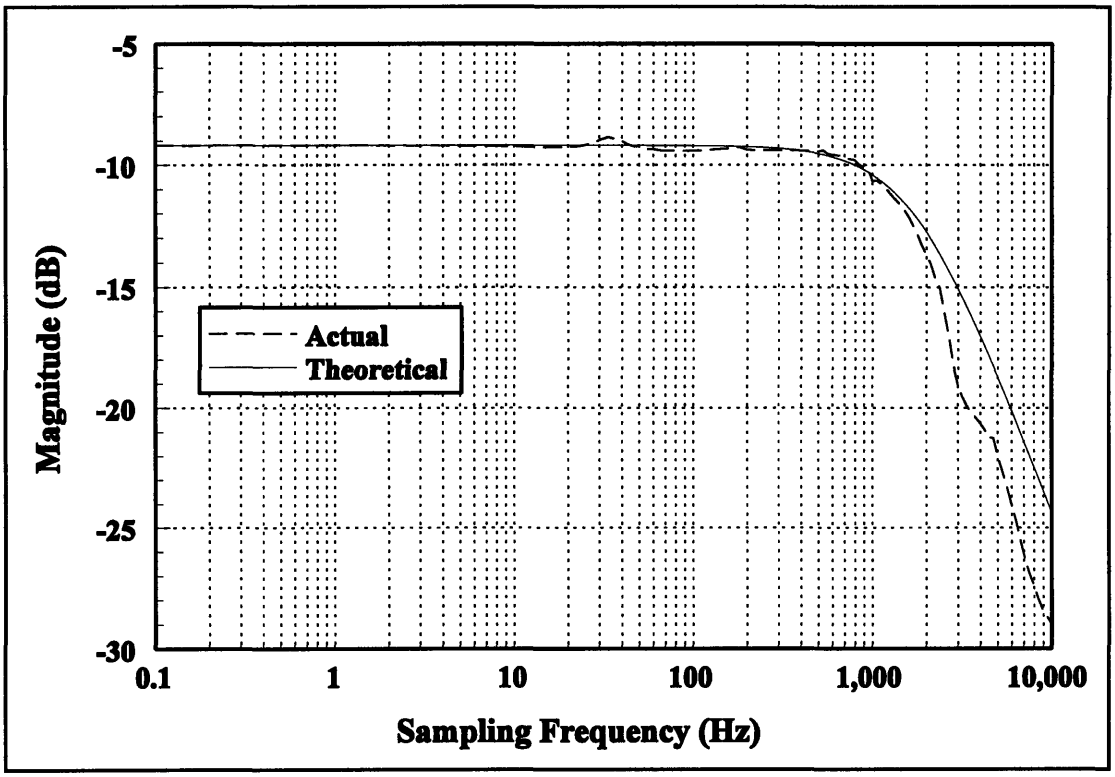

**E-25** Radial Bearing 1X Best Fit versus Actual Driver Transfer Function Phase Plot



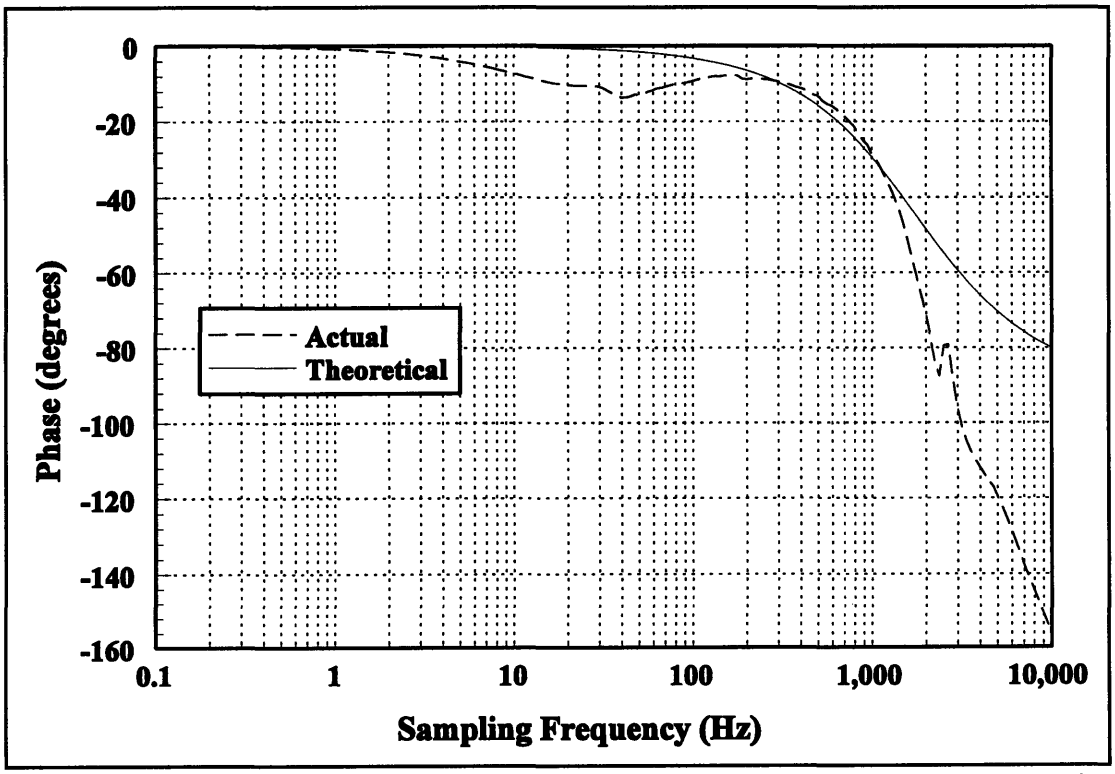

**E-26** Radial Bearing 1Y Best Fit versus Actual Driver Transfer Function Magnitude Plot



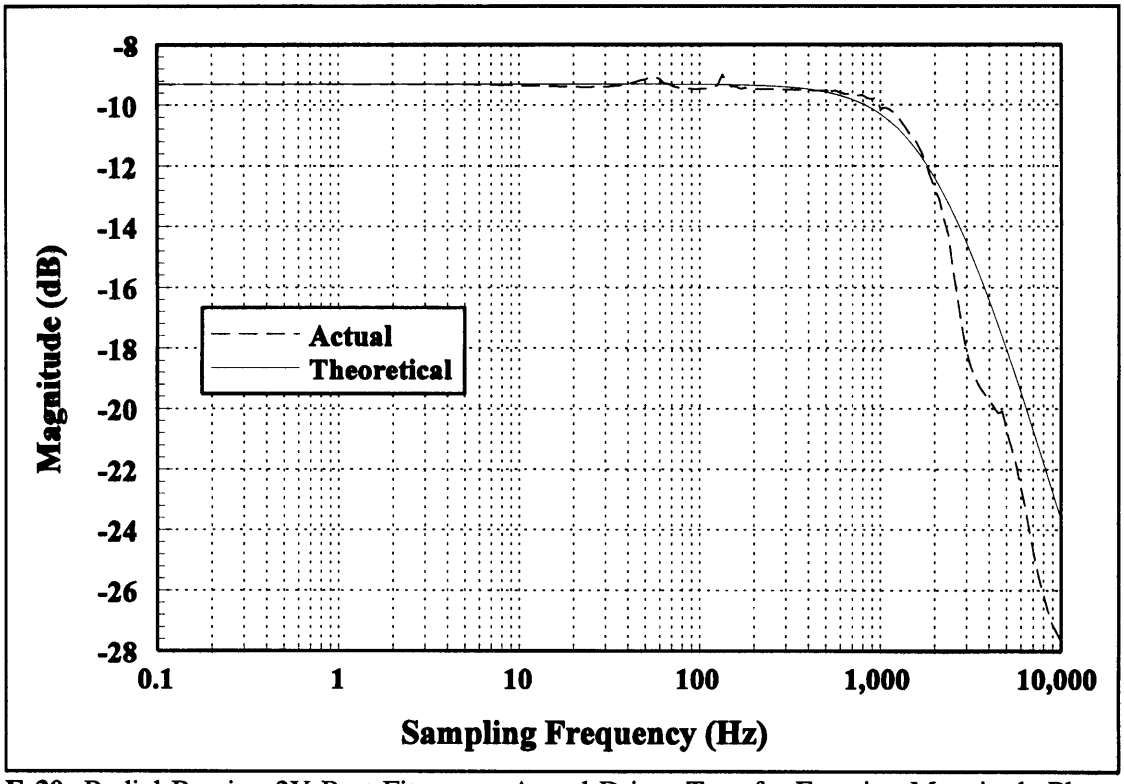

**E-27** Radial Bearing 1Y Best Fit versus Actual Driver Transfer Function Phase Plot



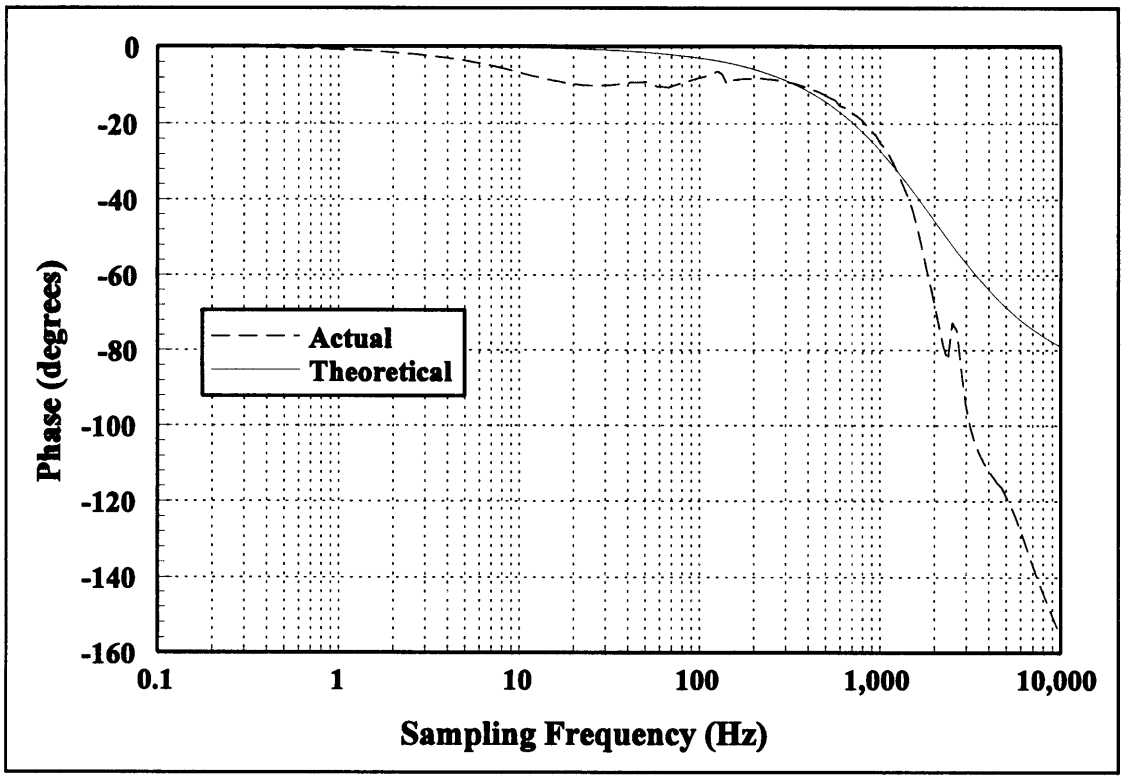

**E-28** Radial Bearing 2X Best Fit versus Actual Driver Transfer Function Magnitude Plot




**E-29** Radial Bearing 2X Best Fit versus Actual Driver Transfer Function Phase Plot




**E-30** Radial Bearing 2Y Best Fit versus Actual Driver Transfer Function Magnitude Plot




**E-31** Radial Bearing 2Y Best Fit versus Actual Driver Transfer Function Phase Plot

# E.4 Best Fit versus Actual Plant Transfer Function

The best fit transfer functions for both the pump and the driver have been determined individually but the summation of these components must be compared to the actual plant transfer function. The general form of the best fit plant transfer function is,

$$\frac{X(s)}{U(s)} = \left(\frac{A_1}{s+A_2}\right)\left(\frac{P}{s^2-Q}\right)$$

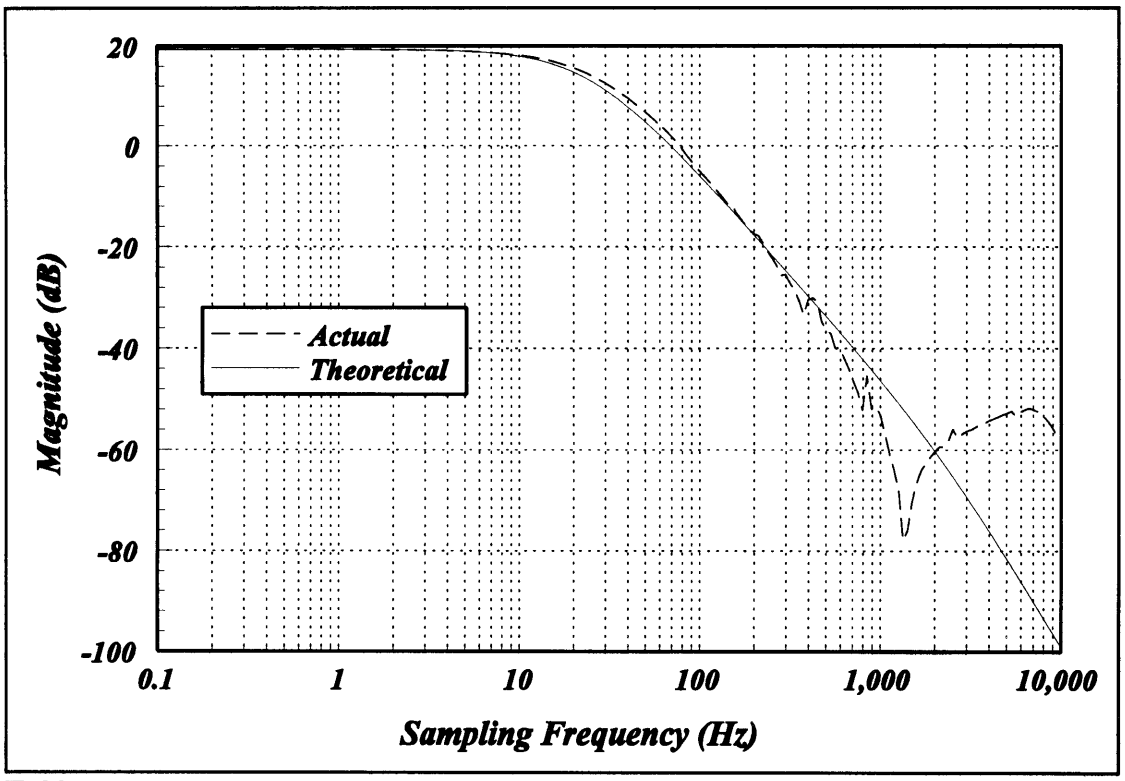
the representative values of $P$, $Q$, $A_1$ and $A_2$ are,

| Parameter | Axial | Rad1X | Rad1Y | Rad2X | Rad2Y |
|---|---|---|---|---|---|
| P | 7.990 | 7.123 | 8.296 | 16.926 | 15.113 |
| Q | 22739.568 | 7737.770 | 8882.644 | 35530.574 | 33201.348 |
| $A_1$ | 14707.770 | 13105.209 | 13043.913 | 11112.759 | 12273.613 |
| $A_2$ | 13310.000 | 13130.000 | 13080.000 | 11140.000 | 12290.000 |

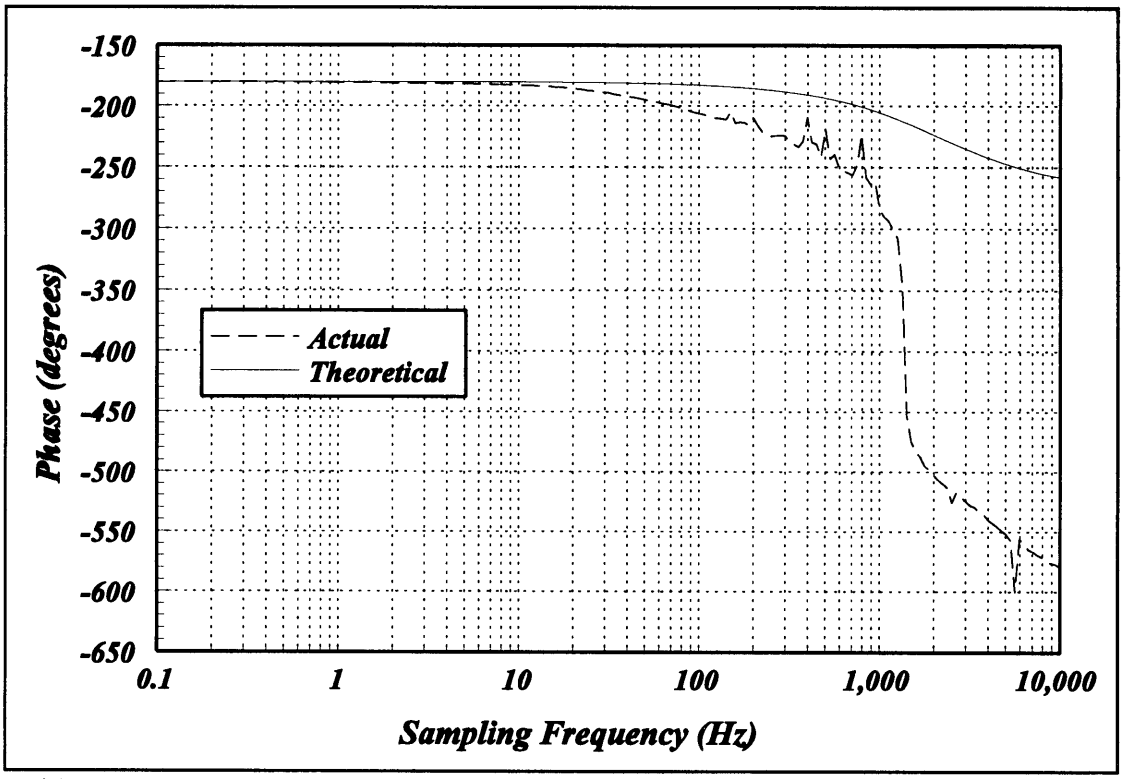
Finally, either the actual transfer function data or the best fit transfer function data must be converted to compatible units for comparison purposes. The actual transfer function has units of V/V and the best fit transfer function data has units of m/A. The conversion factors are,

| Conversion Factor | Axial | Rad1X | Rad1Y | Rad2X | Rad2Y |
|---|---|---|---|---|---|
| Control Signal (A/V) | 2.776 | 1.175 | 1.165 | 1.159 | 1.144 |
| Position Signal (V/m) | 9540.0 | 25000.0 | 25000.0 | 25000.0 | 25000.0 |

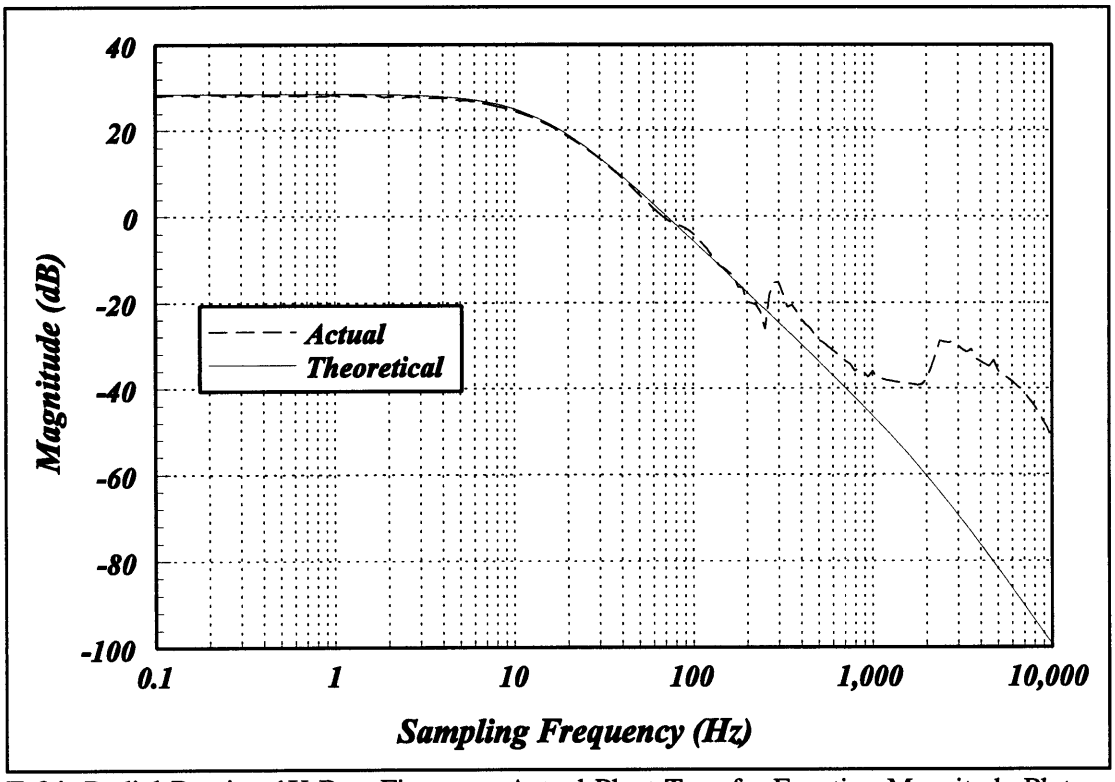
The actual transfer function versus best fit transfer function plots begin on the next page.



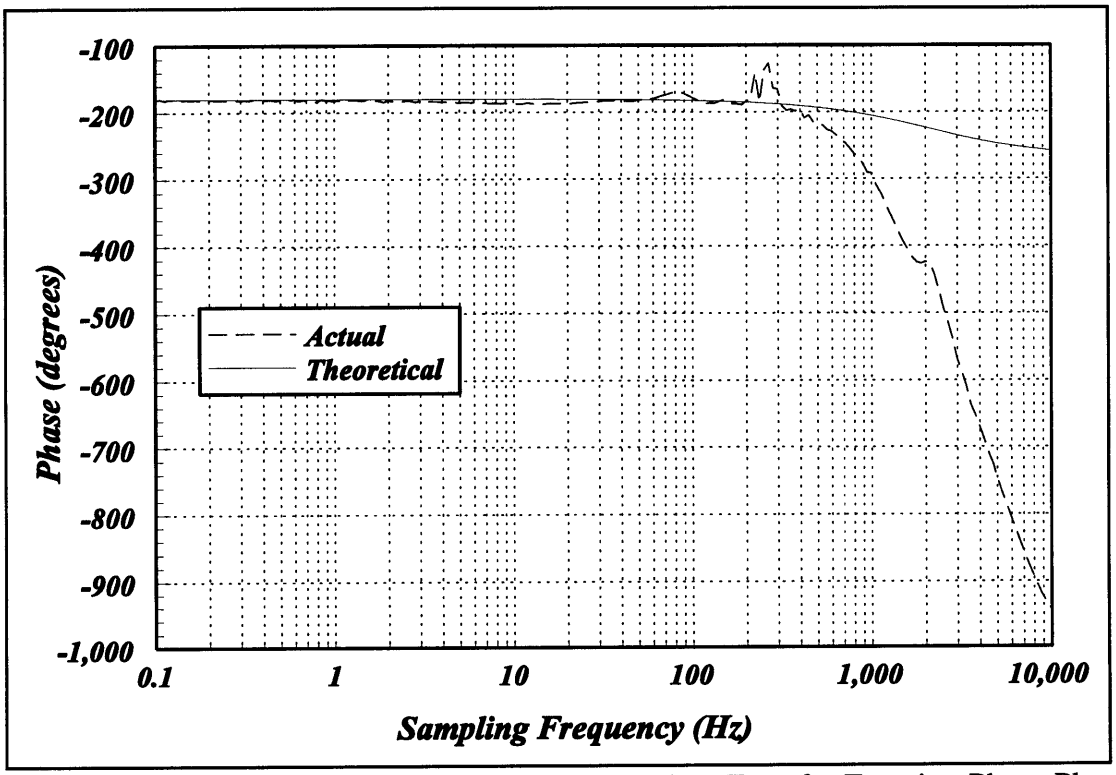

**E-32** Axial Bearing Best Fit versus Actual Plant Transfer Function Magnitude Plot



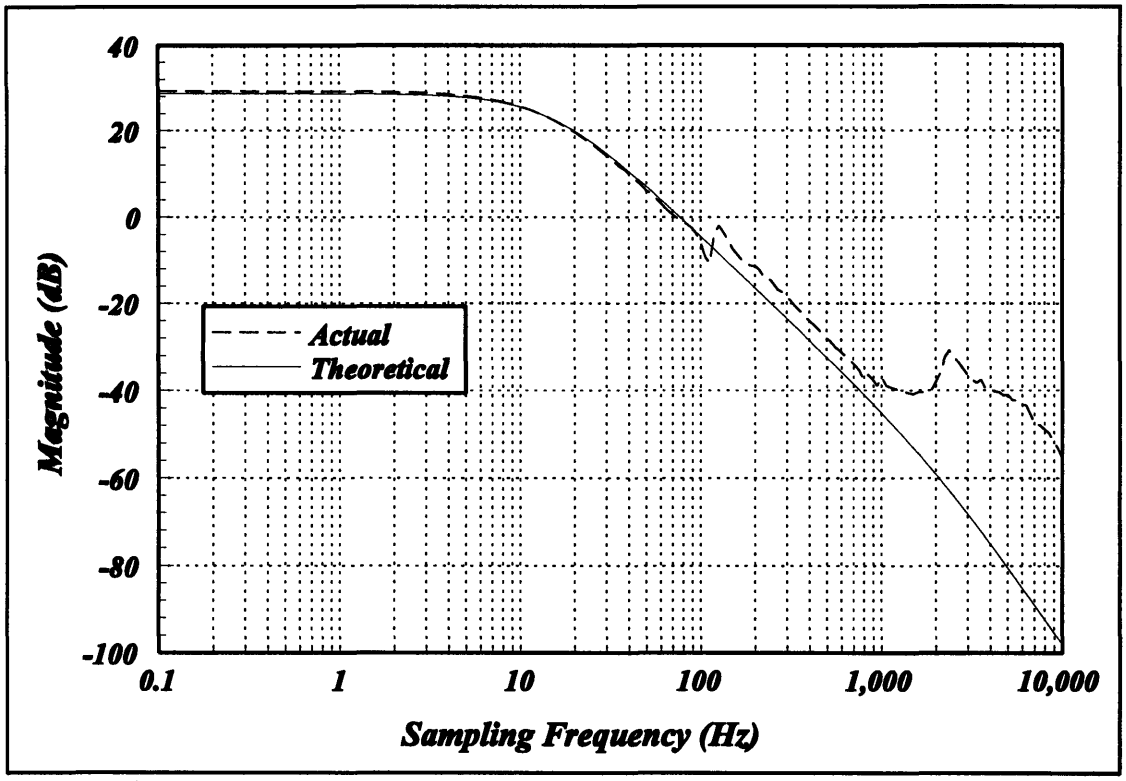

**E-33** Axial Bearing Best Fit versus Actual Plant Transfer Function Phase Plot



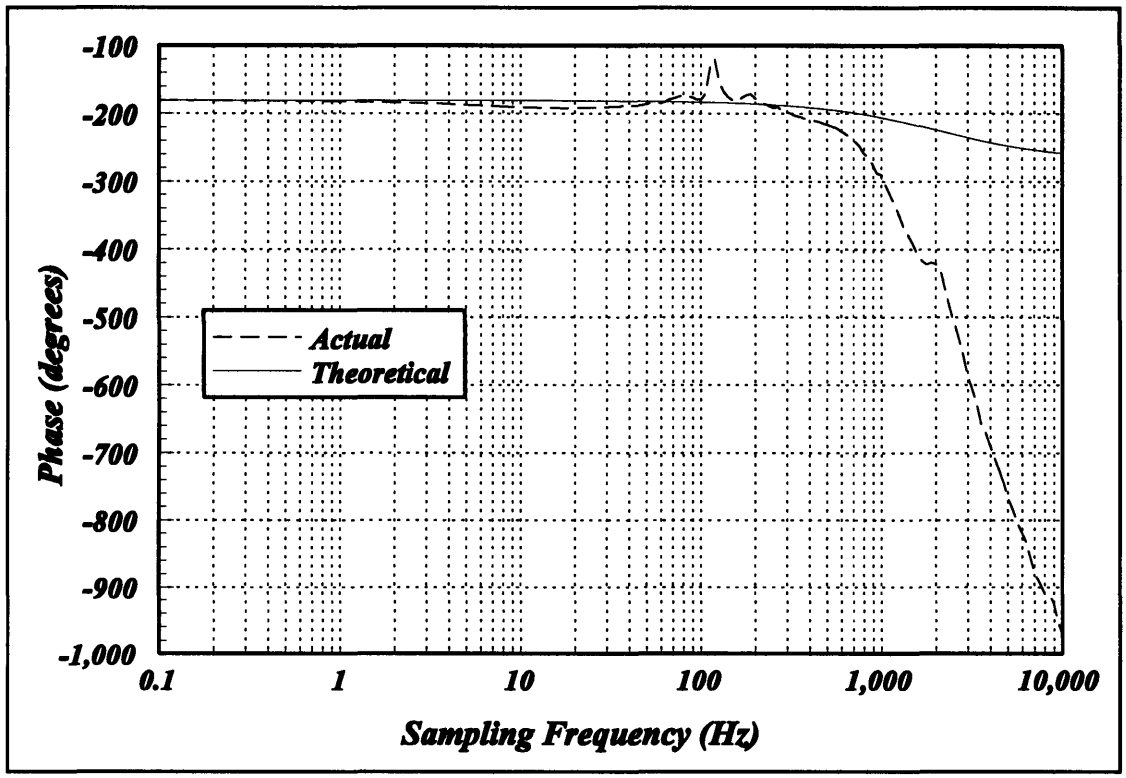

**E-34** Radial Bearing 1X Best Fit versus Actual Plant Transfer Function Magnitude Plot



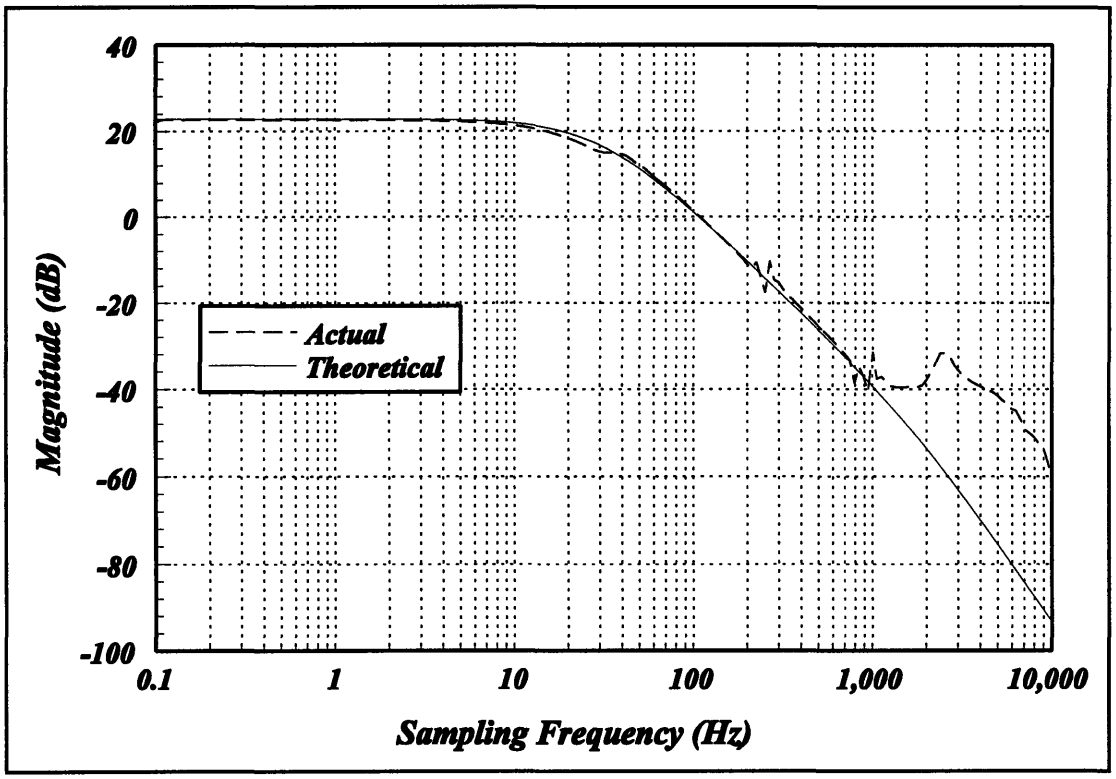

**E-35** Radial Bearing 1X Best Fit versus Actual Plant Transfer Function Phase Plot



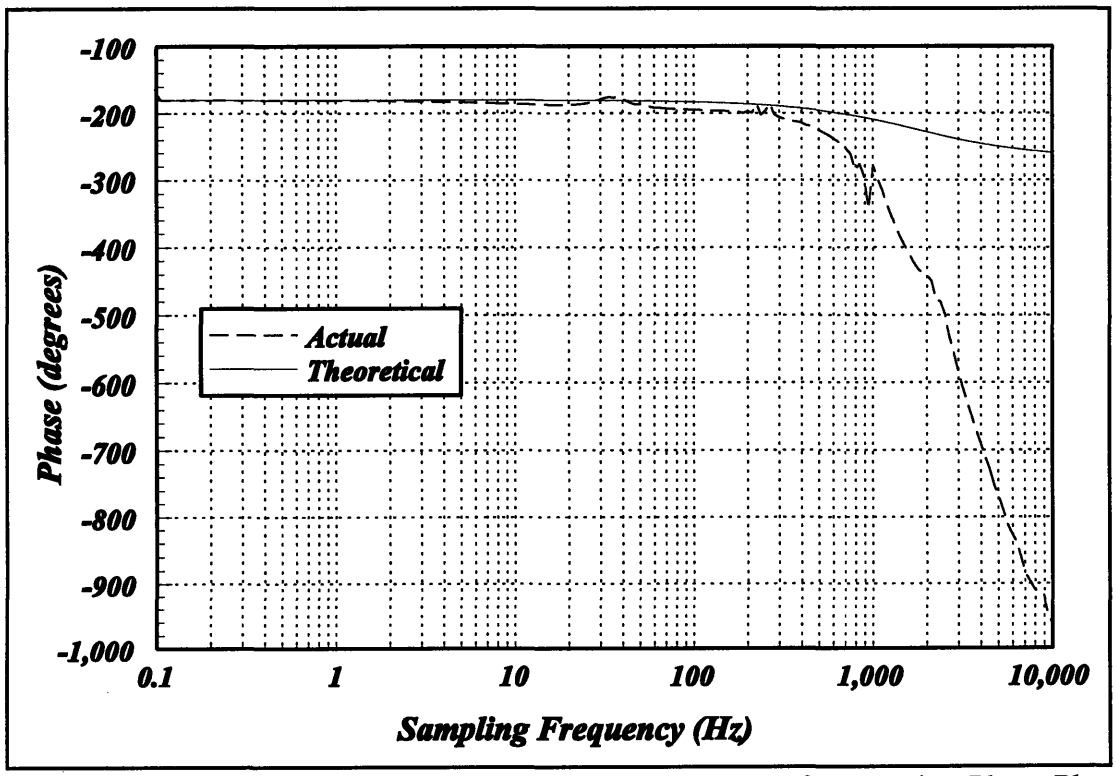

**E-36** Radial Bearing 1Y Best Fit versus Actual Plant Transfer Function Magnitude Plot



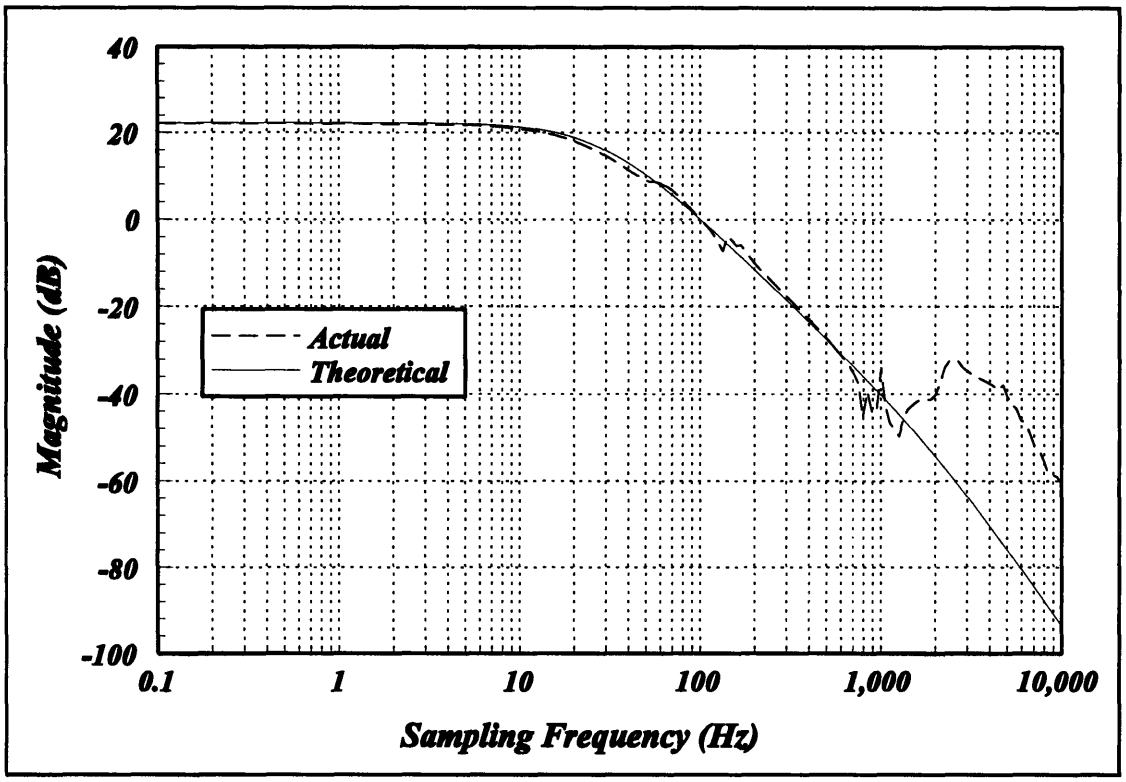

**E-37** Radial Bearing 1Y Best Fit versus Actual Plant Transfer Function Phase Plot



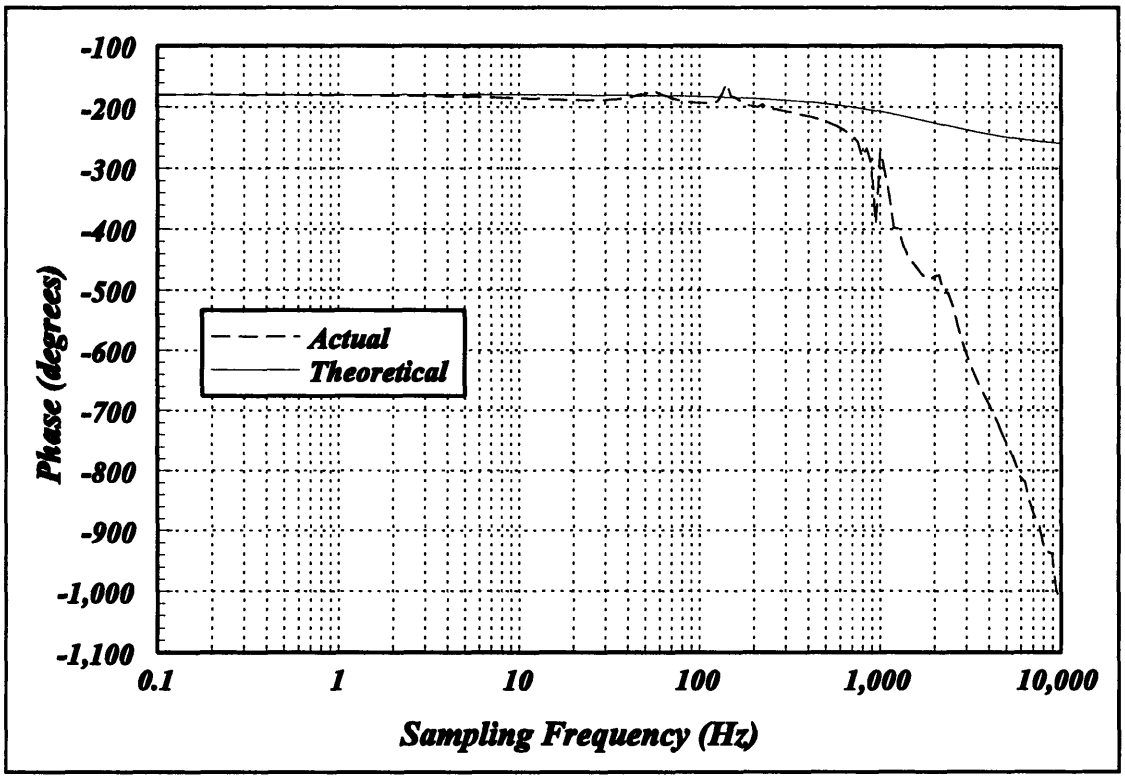

**E-38** Radial Bearing 2X Best Fit versus Actual Plant Transfer Function Magnitude Plot




**E-39** Radial Bearing 2X Best Fit versus Actual Plant Transfer Function Phase Plot




**E-40** Radial Bearing 2Y Best Fit versus Actual Plant Transfer Function Magnitude Plot




**E-41** Radial Bearing 2Y Best Fit versus Actual Plant Transfer Function Phase Plot

# E.5 Best Fit Pump Transfer Function Program Listings

The following section contains two listings of programs used to determine the best fit transfer function for the turbopump magnetic bearing subsystem. The first listing is a Matlab script which determines and graphically displays the best fit transfer function. The second listing is a C source listing which performs the same function as the Matlab script but has no graphing capabilities. It is however substantially faster then the Matlab script.

## E.5.1 Matlab Script

```
function [] = FindOpenFunc(bearing)
% This attempts to find the corresponding transfer function of the open
% loop system from data returned by the system analyzer

if(nargin ~= 1)
  disp('Syntax error');
  disp('FindOpenFunc(bearing)');
  disp('where:');
  disp('        bearing - vacuum pump bearing number');
  disp('                  0 = axial bearing');
  disp('                  1 = radial 1X bearing');
  disp('                  2 = radial 1Y bearing');
  disp('                  3 = radial 2X bearing');
  disp('                  4 = radial 2Y bearing');
  error;
end

if(bearing == 0)
  magname = '/usr/tmp/sysanal/axt7psmg.dat';
  phsname = '/usr/tmp/sysanal/axt7psph.dat';
  bearname = 'Axial';
  sengain = 9450.0;   % Volts/meter
elseif(bearing == 1)
  magname = '/usr/tmp/sysanal/1xt6psmg.dat';
  phsname = '/usr/tmp/sysanal/1xt6psph.dat';
  bearname = 'Rad1X';
  sengain = 25000.0;   % Volts/meter
elseif(bearing == 2)
  magname = '/usr/tmp/sysanal/1yt8psmg.dat';
  phsname = '/usr/tmp/sysanal/1yt8psph.dat';
  bearname = 'Rad1Y';
  sengain = 25000.0;   % Volts/meter
elseif(bearing == 3)
  magname = '/usr/tmp/sysanal/2xt6psmg.dat';
  phsname = '/usr/tmp/sysanal/2xt6psph.dat';
  bearname = 'Rad2X';
  sengain = 25000.0;   % Volts/meter
elseif(bearing == 4)
  magname = '/usr/tmp/sysanal/2yt8psmg.dat';
  phsname = '/usr/tmp/sysanal/2yt8psph.dat';
  bearname = 'Rad2Y';
  sengain = 25000.0;   % Volts/meter
else
  disp(['bearing number out of range:  ' num2str(bearing)]);
  error;
end

% Some constants
ampgain = 1.0/0.3;  % Amps/Volt
dcavg = 10;         % number of values to average to get DC gain
```

```
mytitle = [bearname ' Bearing Open Loop Transfer Function Best Fit ' ...
        '(Range = 0.1 - '];

fid = fopen(magname);
if(fid < 3)
  disp(['unable to open magnitude data file:  ' magname]);
  error;
end
[mag, count] = fscanf(fid, '%f %f', [2 inf]);
mag = mag';
fclose(fid);

fid = fopen(phsname);
if(fid < 3)
  disp(['unable to open phase data file:  ' phsname]);
  error;
end
[phs, count] = fscanf(fid, '%f %f', [2 inf]);
phs = phs';
fclose(fid);

% Smooth phase anomalies
for ii = 2:size(phs,1);
  if(abs(phs(ii-1,2)-phs(ii,2)) > 180.0)
    if(phs(ii-1,2) >= 0.0)
      phs(ii,2) = phs(ii,2)+360.0;
    else
      phs(ii,2) = phs(ii,2)-360.0;
    end
  end
end

% Limit frequency range because the data from the system analyzer is very
% dirty after 1000 Hz
limit = 1;

if(limit == 1)
  ii = max(find(mag(:,1) < 1000.0));
  mag = mag(1:ii,:);
  phs = phs(1:ii,:);
  mytitle = [mytitle '1000 Hz)'];
else
  mytitle = [mytitle '10000 Hz)'];
end

% Assume that the transfer function of the open loop system has the same form
% as that derived from the dynamic equations of motion namely P^2/(s^2 - Q^2)
% Set up range of numbers to test as Q
begin = 1.0;
endd = 10000.0;

ind = begin;
range = [begin];
while(ind < endd)
  inc = ind/10.0;
  if(inc > 10.0)
    inc = 10.0;
  end
  range = [range [ind+inc:inc:ind*10.0]];
  ind = ind * 10.0;
end

resid = zeros(length(range),3);

vartype = 'Var Type = Ideal';

nw = mag(:,1)*2*pi;

inc = 1;
```

```
if(bearing == 0)
  opnum = -(sengain * ampgain);    % convert to Volts/Volts
else
  opnum = sengain * ampgain;       % convert to Volts/Volts
end

more off;
for ii = 1:length(range)
  opden = [1 0 -((2*pi*range(ii))^2)];
  [tmag, tphs, w] = bode(opnum, opden, nw);
  if(bearing == 3)
    tphs = tphs + 360.0;
  end
  tmag = 20.0*log10(tmag);
  gain = 0;   % find average gain by checking DC gain --> gain is numerator
  for jj = 1:dcavg
    gain = gain + (mag(jj,2)-tmag(jj));
  end
  gain = gain/dcavg;
  tmag = gain+tmag;
  resid(ii,:) = [0 0 0];
  for jj = 1:length(mag)
    resid(ii,1) = resid(ii,1) + ((mag(jj,2) - tmag(jj))^2);
    resid(ii,2) = resid(ii,2) + ((phs(jj,2) - tphs(jj))^2);
  end
  resid(ii,1) = resid(ii,1)/length(mag);
  resid(ii,2) = resid(ii,2)/length(mag);
  resid(ii,3) = resid(ii,1) + resid(ii,2);
  if(fix(rem(ii,100)) == 0)
    disp([num2str(ii) ' of ' num2str(length(range))]);
  end
end

[x, ii] = min(resid(:,3));
x = x(1);
ii = ii(1);
opden = [1 0 -((2*pi*range(ii))^2)];
[tmag, tphs, w] = bode(opnum, opden, nw);
if(bearing == 3)
  tphs = tphs + 360.0;
end
tmag = 20.0*log10(tmag);
gain = 0;   % find average gain by checking DC gain --> gain is numerator
for jj = 1:dcavg
  gain = gain + (mag(jj,2)-tmag(jj));
end
gain = gain/dcavg;
tmag = tmag + gain;
opnum = opnum * (10^(gain / 20.0));

msg = sprintf('%.3f', abs(opnum/(ampgain * sengain)));
opnum_title = ['P = ', msg];
msg = sprintf('%.3f', abs(opden(length(opden))));
opden_title = ['Q = ', msg];

msg = ['Plotting best residual magnitude  (R = ' num2str(x) '  val = ' ...
       num2str(range(ii)) '  Gain = ' num2str(10^(gain/20.0)) ')'];
disp(msg);

weight = 'phase weighted';

clf;
subplot(2,1,1);

semilogx(mag(:,1), mag(:,2), '-');
xlabel('Frequency (Hz)');
ylabel('Magnitude (dB)');
title(mytitle);
```

```
hold on;
subplot(2,1,2);
semilogx(phs(:,1), phs(:,2), '-');
xlabel('Frequency (Hz)');
ylabel('Phase (degrees)');

hold on;

subplot(2,1,1);
semilogx(mag(:,1), tmag, '--');
hold off;

subplot(2,1,2);
semilogx(phs(:,1), tphs, '--');

title([vartype ', ' opnum_title ', ' opden_title ', ' weight]);

hold off;
more on;
```

## E.5.2 C Source

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define  STATUS_OK    0
#define  STATUS_NOK   1
#define  PI           3.14159265358979323846

#define  VOLT_CONV    0.3        /* Ohms, V = IR */

int ConBode(float *num, float *den, int numsize, int densize, float *freq,
            int fsize, float **mag, float **phase);

float sen_gain[5] = { 9450.0, 25000.0, 25000.0, 25000.0, 25000.0 };

int main(int argc, char **argv)
{
  int i, k, off, dcavg;
  int bearing, size, psize;
  char *magname, *phsname, *bearname;
  char *buffer, *ptr;
  float sengain, ampgain, t1, t2, t3, t4;
  float *freq, *mag, *phase;
  float *pltval, *pltres, *tmag, *tphs;
  float num[1], den[3];
  FILE *ifp;

  if(argc != 2)
    {
      fprintf(stderr, "Syntax error\n");
      fprintf(stderr, "FindOpenFunc(bearing)\n");
      fprintf(stderr, "where:\n");
      fprintf(stderr, "        bearing - vacuum pump bearing number\n");
      fprintf(stderr, "                  0 = axial bearing\n");
      fprintf(stderr, "                  1 = radial 1X bearing\n");
      fprintf(stderr, "                  2 = radial 1Y bearing\n");
      fprintf(stderr, "                  3 = radial 2X bearing\n");
      fprintf(stderr, "                  4 = radial 2Y bearing\n");
      return(STATUS_NOK);
    }

  bearing = strtol(argv[1], NULL, 10);
```

```
switch(bearing)
  {
  case 0:
    magname = "/usr/sysanal/axt7psmg.dat";
    phsname = "/usr/sysanal/axt7psph.dat";
    bearname = "Axial";
    break;
  case 1:
    magname = "/usr/sysanal/1xt6psmg.dat";
    phsname = "/usr/sysanal/1xt6psph.dat";
    bearname = "Rad1X";
    break;
  case 2:
    magname = "/usr/sysanal/1yt8psmg.dat";
    phsname = "/usr/sysanal/1yt8psph.dat";
    bearname = "Rad1Y";
    break;
  case 3:
    magname = "/usr/sysanal/2xt6psmg.dat";
    phsname = "/usr/sysanal/2xt6psph.dat";
    bearname = "Rad2X";
    break;
  case 4:
    magname = "/usr/sysanal/2yt8psmg.dat";
    phsname = "/usr/sysanal/2yt8psph.dat";
    bearname = "Rad2Y";
    break;
  default:
    fprintf(stderr, "bearing number out of range:  %d\n", bearing);
    return(STATUS_NOK);
    break;
  }

ampgain = VOLT_CONV / sen_gain[bearing];
dcavg = 10;

if((buffer = (char *) calloc(80, sizeof(char))) == NULL)
  {
    fprintf(stderr, "Out of memory error (%d)\n", __LINE__);
    return(STATUS_NOK);
  }

if((freq = (float *) calloc(3000, sizeof(float))) == NULL)
  {
    fprintf(stderr, "Out of memory error (%d)\n", __LINE__);
    free(buffer);
    return(STATUS_NOK);
  }
mag = (float *) (((unsigned) freq) + (1000 * sizeof(float)));
phase = (float *) (((unsigned) mag) + (1000 * sizeof(float)));

if((ifp = fopen(magname, "r")) == NULL)
  {
    fprintf(stderr, "Unable to open analog closed loop magnitude file:  "
          "%s\n", magname);
    return(STATUS_NOK);
  }

k = 0;
while(fgets(buffer, 80, ifp))
  {
    if((i = strlen(buffer)) == 79)
      fprintf(stderr, "Possible input buffer overflow\n");
    if(buffer[i-1] == '\n')
      buffer[--i] = '\0';
    freq[k] = (float) strtod(buffer, &ptr);
    mag[k++] = (float) strtod(ptr, NULL);
  }
```

```
if(!feof(ifp))
  {
    fprintf(stderr, "Error while reading magnitude file\n");
    free(buffer);
    free(freq);
    return(STATUS_NOK);
  }
fclose(ifp);

if((ifp = fopen(phsname, "r")) == NULL)
  {
    fprintf(stderr, "Unable to open analog closed loop phase file:  "
            "%s\n", phsname);
    return(STATUS_NOK);
  }

k = 0;
while(fgets(buffer, 80, ifp))
  {
    if((i = strlen(buffer)) == 79)
      fprintf(stderr, "Possible input buffer overflow\n");
    if(buffer[i-1] == '\n')
      buffer[--i] = '\0';
    freq[k] = (float) strtod(buffer, &ptr);
    phase[k++] = (float) strtod(ptr, NULL);
  }

if(!feof(ifp))
  {
    fprintf(stderr, "Error while reading phase file\n");
    free(buffer);
    free(freq);
    return(STATUS_NOK);
  }
fclose(ifp);

size = k;

/* smooth phase anomalies */
while(phase[0] > 170.0)
  phase[0] -= 360.0;
for(i=1; i<size; i++)
  {
    if(fabs(phase[i-1]-phase[i]) > 170.0)
      {
        if(phase[i-1] >= 0.0)
          phase[i] += 360.0;
        else
          phase[i] -= 360.0;
      }
    if(freq[i] <= 1000.0)
      k = i;
  }

size = k + 1;

/*
 * assume that the denominator of the transfer function is third order
 * (physics of magnetic bearing is second order, power amplifier is
 * first order)
 */

psize = 100;
if((pltval = (float *) calloc(psize, sizeof(float))) == NULL)
  {
    fprintf(stderr, "Out of memory error (%d)\n", __LINE__);
    free(buffer);
    free(freq);
    return(STATUS_NOK);
```

```
  }

pltval[0] = t1 = 1.0;
t2 = 10000.0;
k = 1;

while(t1 < t2)
  {
    t3 = t1 / 10.0;
    if(t3 > 10.0)
      t3 = 10.0;
    t4 = t1 * 10.0;
    for(t1+=t3; t1<t4; t1+=t3)
      {
        pltval[k++] = t1;
        if(k == psize)
          {
            psize += 100;
            if((pltval = (float *) realloc(pltval, psize*sizeof(float)))
               == NULL)
              {
                fprintf(stderr, "Out of memory error (%d)\n", __LINE__);
                free(buffer);
                free(freq);
                return(STATUS_NOK);
              }
          }
      }
    pltval[k++] = t1 = t4;
  }

psize = k;

if((pltres = (float *) calloc(3*psize, sizeof(float))) == NULL)
  {
    fprintf(stderr, "Out of memory error (%d)\n", __LINE__);
    free(buffer);
    free(freq);
    free(pltval);
    return(STATUS_NOK);
  }

den[0] = 1.0;
den[1] = 0.0;
t4 = pltval[0];
if(bearing)
  num[0] = 1.0;
else
  num[0] = -1.0;
for(i=0; i<psize; i++)
  {
    den[2] = 2.0*PI*pltval[i];
    den[2] *= -(den[2]);
    if(ConBode(num, den, 1, 3, freq, size, &tmag, &tphs))
      {
        fprintf(stderr, "Error encountered in ConBode (%d)\n", __LINE__);
        free(buffer);
        free(freq);
        free(pltval);
        free(pltres);
        return(STATUS_NOK);
      }
    while(tphs[0] > 170.0)
      tphs[0] -= 360.0;
    for(k=1; k<size; k++)
      {
        while(fabs(tphs[k-1]-tphs[k]) > 170.0)
          {
            if(tphs[k-1] >= 0.0)
```

```
                tphs[k] += 360.0;
              else
                tphs[k] -= 360.0;
            }
        }
      t1 = 0.0;
      for(k=0; k<dcavg; k++)
        t1 += (mag[k]-tmag[k]);
      t1 /= ((float) dcavg);
      off = i * 3;
      pltres[off] = pltres[off+1] = pltres[off+2] = 0.0;
      for(k=0; k<size; k++)
        {
          t2 = mag[k] - tmag[k] - t1;
          t3 = phase[k] - tphs[k];
          pltres[off] += (t2 * t2);
          pltres[off+1] += (t3 * t3);
        }
      pltres[off] /= ((float) size);
      pltres[off+1] /= ((float) size);
      pltres[off+2] = pltres[off] + pltres[off+1];
      free(tmag);
      free(tphs);
#if 1
      if(pltval[i] >= t4)
        {
          printf("%8.1f\n", t4);
          t4 *= 10.0;
        }
#else
      printf("%8.1f\n", pltval[i]);
#endif
    }

  k = 0;
  for(i=1; i<psize; i++)
    if(pltres[(i*3)+2] < pltres[(k*3)+2])
      k = i;
  den[2] = 2.0*PI*pltval[k];
  den[2] *= -(den[2]);
  if(ConBode(num, den, 1, 3, freq, size, &tmag, &tphs))
    {
      fprintf(stderr, "Error encountered in ConBode (%d)\n", __LINE__);
      free(buffer);
      free(freq);
      free(pltval);
      free(pltres);
      return(STATUS_NOK);
    }
  t1 = 0.0;
  for(k=0; k<dcavg; k++)
    t1 += (mag[k]-tmag[k]);
  t1 /= ((float) dcavg);
  t2 = (float) pow(10.0, (double) (t1/20.0));
  printf("%s:  P = %.3f, Q = %.3f\n", bearname, t2 * ampgain, fabs(den[2]));
  free(buffer);
  free(freq);
  free(pltval);
  free(pltres);
  free(tmag);
  free(tphs);

  return(STATUS_OK);

}


int ConBode(float *num, float *den, int numsize, int densize, float *freq,
            int fsize, float **mag, float **phase)
```

```
{
  int i, j, k;
  float *mg, *ph, inc;
  float frq, ansr, ansi;

  if((mg = (float *) calloc(fsize, sizeof(float))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigBode.\n");
      return(1);
    }

  if((ph = (float *) calloc(fsize, sizeof(float))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigBode.\n");
      free(mg);
      return(1);
    }

  for(i=0; i<fsize; i++)
    {
      frq = 2.0 * PI * freq[i];
      mg[i] = ph[i] = ansi = 0.0;
      ansr = num[numsize-1];
      for(j=0; j<numsize-1; j++)
        {
          if(num[j] != 0.0)
            {
              k = numsize - 1 - j;
              if((k/2)%2 == 1)
                inc = -1.0;
              else
                inc = 1.0;
              if(k%2 == 1)
                ansi += (inc * num[j] * ((float) pow(frq, (double) k)));
              else
                ansr += (inc * num[j] * ((float) pow(frq, (double) k)));
            }
        }
      mg[i] += (20.0 * log10(sqrt(pow(ansr,2.0) + pow(ansi,2.0))));
      ph[i] += (180.0 * atan2(ansi, ansr) / PI);
      ansi = 0.0;
      ansr = den[densize-1];
      for(j=0; j<densize-1; j++)
        {
          if(den[j] != 0.0)
             {
               k = densize - 1 - j;
               if((k/2)%2 == 1)
                 inc = -1.0;
               else
                 inc = 1.0;
               if(k%2 == 1)
                 ansi += (inc * den[j] * ((float) pow(frq, (double) k)));
               else
                 ansr += (inc * den[j] * ((float) pow(frq, (double) k)));
             }
        }
      mg[i] -= (20.0 * log10(sqrt(pow(ansr,2.0) + pow(ansi,2.0))));
      ph[i] -= (180.0 * ((float) atan2(ansi, ansr)) / PI);
    }

  if(mag)
    (*mag) = mg;
  if(phase)
    {
      for(i=0; i<fsize; i++)
        {
          while(fabs(ph[i-1]-ph[i]) > 180.0)
            {
```

```
                frq = ph[i] - 360.0;
                if(fabs(ph[i-1]-ph[i]) > fabs(ph[i-1]-frq))
                  ph[i] = frq;
                else
                  ph[i] += 360.0;
              }
          }
        (*phase) = ph;
      }

  return(0);

}
```

# E.6 Best Fit Driver Transfer Function Program Listings

The following section contains two listings of programs used to determine the best fit transfer function for the turbopump driver subsystem. The first listing is a Matlab script which determines and graphically displays the best fit transfer function. The second listing is a C source listing which performs the same function as the Matlab script but has no graphing capabilities. It is however substantially faster then the Matlab script.

## E.6.1 Matlab Script

```
function [] = FindAmpFunc(bearing)
% This attempts to find the corresponding transfer function of the amp/driver
% from data returned by the system analyzer

if(nargin ~= 1)
  disp('Syntax error');
  disp('FindAmpFunc(bearing)');
  disp('where:');
  disp('        bearing - vacuum pump bearing number');
  disp('                  0 = axial bearing');
  disp('                  1 = radial 1X bearing');
  disp('                  2 = radial 1Y bearing');
  disp('                  3 = radial 2X bearing');
  disp('                  4 = radial 2Y bearing');
  error;
end

if(bearing == 0)
  magname = '/usr/tmp/sysanal/axcnt7mg.dat';
  phsname = '/usr/tmp/sysanal/axcnt7ph.dat';
  bearname = 'Axial';
elseif(bearing == 1)
  magname = '/usr/tmp/sysanal/1xcnt6mg.dat';
  phsname = '/usr/tmp/sysanal/1xcnt6ph.dat';
  bearname = 'Rad1X';
elseif(bearing == 2)
  magname = '/usr/tmp/sysanal/1ycnt8mg.dat';
  phsname = '/usr/tmp/sysanal/1ycnt8ph.dat';
  bearname = 'Rad1Y';
elseif(bearing == 3)
  magname = '/usr/tmp/sysanal/2xcnt6mg.dat';
  phsname = '/usr/tmp/sysanal/2xcnt6ph.dat';
  bearname = 'Rad2X';
elseif(bearing == 4)
  magname = '/usr/tmp/sysanal/2ycnt8mg.dat';
  phsname = '/usr/tmp/sysanal/2ycnt8ph.dat';
```

```
  bearname = 'Rad2Y';
else
  disp(['bearing number out of range:  ' num2str(bearing)]);
  error;
end

% Some constants
ampgain = [0.3   0.3   0.3   0.3   0.3];      % Volts/Amp
dcgain =  [2.776 1.175 1.165 1.159 1.144];    % Amps/Volt
convfact = 1.0 / (ampgain(bearing+1) * dcgain(bearing+1));
dcavg = 10;          % number of values to average to get DC gain

mytitle = [bearname ' Bearing Amp/Driver Transfer Function Best Fit ' ...
        '(Range = 0.1 - '];

fid = fopen(magname);
if(fid < 3)
  disp(['unable to open magnitude data file:  ' magname]);
  error;
end
[mag, count] = fscanf(fid, '%f %f', [2 inf]);
fclose(fid);

fid = fopen(phsname);
if(fid < 3)
  disp(['unable to open phase data file:  ' phsname]);
  error;
end
[phs, count] = fscanf(fid, '%f %f', [2 inf]);
fclose(fid);

% Smooth phase anomalies
[phs(2,:)] = PhaseFix(bearing, phs(2,:));

% Limit frequency range because the data from the system analyzer is very
% dirty after 1000 Hz
limit = 1;

if(limit == 1)
  ii = max(find(mag(1,:) < 1000.0));
  mag = mag(:,1:ii);
  phs = phs(:,1:ii);
  mytitle = [mytitle '1000 Hz)'];
else
  mytitle = [mytitle '10000 Hz)'];
end

% Assume that the transfer function of the open loop system has the same form
% as that derived from the dynamic equations of motion namely A1/(s + A2)
% Set up range of numbers to test as A2
begin = 100.0;
endd = 100000.0;

ind = begin;
range = [begin];
while(ind < endd)
  inc = ind/10.0;
  if(inc > 10.0)
    inc = 10.0;
  end
  range = [range [ind+inc:inc:ind*10.0]];
  ind = ind * 10.0;
end

resid = zeros(length(range),3);
vartype = 'Var Type = Ideal';
nw = mag(1,:)*2*pi;
inc = 1;
if(bearing == 0)
```

```
  opnum = -1.0;
else
  opnum = 1.0;
end

more off;
for ii = 1:length(range)
  opden = [1 range(ii)];
  [tmag, tphs, w] = bode(opnum, opden, nw);
  if(bearing == 0)
    while(tphs(1) > 90.0)
      tphs(1) = tphs(1) - 360.0;
    end
  end
  [tphs] = PhaseFix(bearing, tphs);
  tmag = 20.0*log10(tmag);
  gain = 0;   % find average gain by checking DC gain --> gain is numerator
  for jj = 1:dcavg
    gain = gain + (mag(2,jj)-tmag(jj));
  end
  gain = gain/dcavg;
  tmag = gain+tmag;
  resid(ii,:) = [0 0 0];
  for jj = 1:length(mag)
    resid(ii,1) = resid(ii,1) + ((mag(2,jj) - tmag(jj))^2);
    resid(ii,2) = resid(ii,2) + ((phs(2,jj) - tphs(jj))^2);
  end
  resid(ii,1) = resid(ii,1)/length(mag);
  resid(ii,2) = resid(ii,2)/length(mag);
  resid(ii,3) = resid(ii,1) + resid(ii,2);
  if(fix(rem(ii,100)) == 0)
    disp([num2str(ii) ' of ' num2str(length(range))]);
  end
end

[x, ii] = min(resid(:,3));
x = x(1);
ii = ii(1);
opden = [1 range(ii)];
[tmag, tphs, w] = bode(opnum, opden, nw);
if(bearing == 0)
  while(tphs(1) > 90.0)
    tphs(1) = tphs(1) - 360.0;
  end
end
[tphs] = PhaseFix(bearing, tphs);
tmag = 20.0*log10(tmag);
gain = 0;   % find average gain by checking DC gain --> gain is numerator
for jj = 1:dcavg
  gain = gain + (mag(2,jj)-tmag(jj));
end
gain = gain/dcavg;
tmag = tmag + gain;
opnum = opnum * (10^(gain / 20.0));

msg = sprintf('%.3f', abs(opnum)*convfact);
opnum_title = ['A1 = ', msg];
msg = sprintf('%.3f', range(ii));
opden_title = ['A2 = ', msg];

msg = ['Plotting best residual magnitude  (R = ' num2str(x) '  val = ' ...
       num2str(range(ii)) '  Gain = ' num2str(10^(gain/20.0)) ')'];
disp(msg);

weight = 'phase weighted';

clf;
subplot(2,1,1);
```

```
semilogx(mag(1,:), mag(2,:), '-');
xlabel('Frequency (Hz)');
ylabel('Magnitude (dB)');
title(mytitle);

hold on;

subplot(2,1,2);

semilogx(phs(1,:), phs(2,:), '-');
xlabel('Frequency (Hz)');
ylabel('Phase (degrees)');

hold on;

subplot(2,1,1);
semilogx(mag(1,:), tmag, '--');
hold off;

subplot(2,1,2);
semilogx(phs(1,:), tphs, '--');

title([vartype ', ' opnum_title ', ' opden_title ', ' weight]);

hold off;
more on;
```

## E.6.2 C Source

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define  STATUS_OK    0
#define  STATUS_NOK   1
#define  PI           3.14159265358979323846

#define  VOLT_CONV    0.3        /* Ohms, V = IR */

float dig_gain[5]  =  { 2.776   ,  1.175   ,  1.165   ,  1.159   ,  1.144};

int ConBode(float *num, float *den, int numsize, int densize, float *freq,
            int fsize, float **mag, float **phase);

int main(int argc, char **argv)
{
  int i, k, off, dcavg;
  int bearing, size, asize;
  char *magname, *phsname, *bearname;
  char *buffer, *ptr;
  float voltconv, ampgain, t1, t2, t3, t4;
  float *freq, *mag, *phase;
  float *ampval, *ampres, *tmag, *tphs;
  float num[1], den[2];
  FILE *ifp;

  if(argc != 2)
    {
      fprintf(stderr, "Syntax error\n");
      fprintf(stderr, "FindAmpFunc(bearing)\n");
      fprintf(stderr, "where:\n");
      fprintf(stderr, "        bearing - vacuum pump bearing number\n");
      fprintf(stderr, "                  0 = axial bearing\n");
      fprintf(stderr, "                  1 = radial 1X bearing\n");
      fprintf(stderr, "                  2 = radial 1Y bearing\n");
```

```
    fprintf(stderr, "                    3 = radial 2X bearing\n");
    fprintf(stderr, "                    4 = radial 2Y bearing\n");
    return(STATUS_NOK);
  }

bearing = strtol(argv[1], NULL, 10);

switch(bearing)
  {
  case 0:
    magname = "/usr/tmp/sysanal/axcnt7mg.dat";
    phsname = "/usr/tmp/sysanal/axcnt7ph.dat";
    bearname = "Axial";
    break;
  case 1:
    magname = "/usr/tmp/sysanal/1xcnt6mg.dat";
    phsname = "/usr/tmp/sysanal/1xcnt6ph.dat";
    bearname = "Rad1X";
    break;
  case 2:
    magname = "/usr/tmp/sysanal/1ycnt8mg.dat";
    phsname = "/usr/tmp/sysanal/1ycnt8ph.dat";
    bearname = "Rad1Y";
    break;
  case 3:
    magname = "/usr/tmp/sysanal/2xcnt6mg.dat";
    phsname = "/usr/tmp/sysanal/2xcnt6ph.dat";
    bearname = "Rad2X";
    break;
  case 4:
    magname = "/usr/tmp/sysanal/2ycnt8mg.dat";
    phsname = "/usr/tmp/sysanal/2ycnt8ph.dat";
    bearname = "Rad2Y";
    break;
  default:
    fprintf(stderr, "bearing number out of range:  %d\n", bearing);
    return(STATUS_NOK);
    break;
  }

dcavg = 10;
ampgain = 1.0 / (VOLT_CONV * dig_gain[bearing]);

if((buffer = (char *) calloc(80, sizeof(char))) == NULL)
  {
    fprintf(stderr, "Out of memory error (%d)\n", __LINE__);
    return(STATUS_NOK);
  }

if((freq = (float *) calloc(3000, sizeof(float))) == NULL)
  {
    fprintf(stderr, "Out of memory error (%d)\n", __LINE__);
    free(buffer);
    return(STATUS_NOK);
  }
mag = (float *) (((unsigned) freq) + (1000 * sizeof(float)));
phase = (float *) (((unsigned) mag) + (1000 * sizeof(float)));

if((ifp = fopen(magname, "r")) == NULL)
  {
    fprintf(stderr, "Unable to open analog closed loop magnitude file:  "
            "%s\n", magname);
    return(STATUS_NOK);
  }

k = 0;
while(fgets(buffer, 80, ifp))
  {
    if((i = strlen(buffer)) == 79)
```

```
      fprintf(stderr, "Possible input buffer overflow\n");
    if(buffer[i-1] == '\n')
      buffer[--i] = '\0';
    freq[k] = (float) strtod(buffer, &ptr);
    mag[k++] = (float) strtod(ptr, NULL);
  }

if(!feof(ifp))
  {
    fprintf(stderr, "Error while reading magnitude file\n");
    free(buffer);
    free(freq);
    return(STATUS_NOK);
  }
fclose(ifp);

if((ifp = fopen(phsname, "r")) == NULL)
  {
    fprintf(stderr, "Unable to open analog closed loop phase file:  "
            "%s\n", phsname);
    return(STATUS_NOK);
  }

k = 0;
while(fgets(buffer, 80, ifp))
  {
    if((i = strlen(buffer)) == 79)
      fprintf(stderr, "Possible input buffer overflow\n");
    if(buffer[i-1] == '\n')
      buffer[--i] = '\0';
    freq[k] = (float) strtod(buffer, &ptr);
    phase[k++] = (float) strtod(ptr, NULL);
  }

if(!feof(ifp))
  {
    fprintf(stderr, "Error while reading phase file\n");
    free(buffer);
    free(freq);
    return(STATUS_NOK);
  }
fclose(ifp);

size = k;

/* smooth phase anomalies */
while(phase[0] > 170.0)
  phase[0] -= 360.0;
for(i=1; i<size; i++)
  {
    if(fabs(phase[i-1]-phase[i]) > 170.0)
      {
        if(phase[i-1] >= 0.0)
          phase[i] += 360.0;
        else
          phase[i] -= 360.0;
      }
    if(freq[i] <= 1000.0)
      k = i;
  }

size = k + 1;

/*
 * assume that the denominator of the transfer function is third order
 * (physics of magnetic bearing is second order, power amplifier is
 * first order)
 */
```

```
asize = 100;
if((ampval = (float *) calloc(asize, sizeof(float))) == NULL)
  {
    fprintf(stderr, "Out of memory error (%d)\n", __LINE__);
    free(buffer);
    free(freq);
    return(STATUS_NOK);
  }

ampval[0] = t1 = 1.0;
t2 = 100000.0;
k = 1;

while(t1 < t2)
  {
    t3 = t1 / 10.0;
    if(t3 > 10.0)
      t3 = 10.0;
    t4 = t1 * 10.0;
    for(t1+=t3; t1<t4; t1+=t3)
      {
        ampval[k++] = t1;
        if(k == asize)
          {
            asize += 100;
            if((ampval = (float *) realloc(ampval, asize*sizeof(float)))
               == NULL)
              {
                fprintf(stderr, "Out of memory error (%d)\n", __LINE__);
                free(buffer);
                free(freq);
                return(STATUS_NOK);
              }
          }
      }
    ampval[k++] = t1 = t4;
  }

asize = k;

if((ampres = (float *) calloc(3*asize, sizeof(float))) == NULL)
  {
    fprintf(stderr, "Out of memory error (%d)\n", __LINE__);
    free(buffer);
    free(freq);
    free(ampval);
    return(STATUS_NOK);
  }

den[0] = 1.0;
t4 = ampval[0];
if(!(bearing))
  num[0] = -1.0;
else
  num[0] = 1.0;
for(i=0; i<asize; i++)
  {
    den[1] = ampval[i];
    if(ConBode(num, den, 1, 2, freq, size, &tmag, &tphs))
      {
        fprintf(stderr, "Error encountered in ConBode (%d)\n", __LINE__);
        free(buffer);
        free(freq);
        free(ampval);
        free(ampres);
        return(STATUS_NOK);
      }
    if(!bearing)
      while(tphs[0] > 90.0)
```

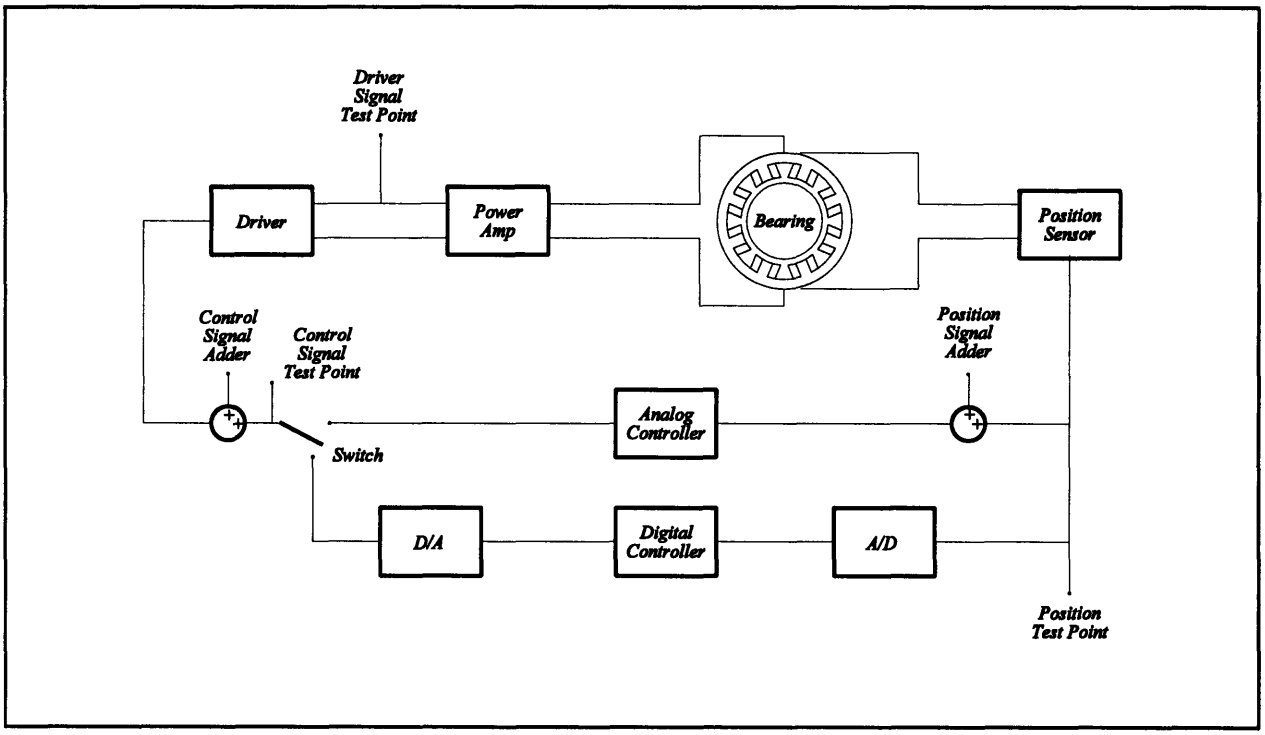
```
          tphs[0] -= 360.0;
        while(tphs[0] > 170.0)
          tphs[0] -= 360.0;
        for(k=1; k<size; k++)
          {
            while(fabs(tphs[k-1]-tphs[k]) > 170.0)
              {
                if(tphs[k-1] >= 0.0)
                  tphs[k] += 360.0;
                else
                  tphs[k] -= 360.0;
              }
          }
        t1 = 0.0;
        for(k=0; k<dcavg; k++)
          t1 += (mag[k]-tmag[k]);
        t1 /= ((float) dcavg);
        off = i * 3;
        ampres[off] = ampres[off+1] = ampres[off+2] = 0.0;
        for(k=0; k<size; k++)
          {
            t2 = mag[k] - tmag[k] - t1;
            t3 = phase[k] - tphs[k];
            ampres[off] += (t2 * t2);
            ampres[off+1] += (t3 * t3);
          }
        ampres[off] /= ((float) size);
        ampres[off+1] /= ((float) size);
        ampres[off+2] = ampres[off] + ampres[off+1];
        free(tmag);
        free(tphs);
#if 1
        if(ampval[i] >= t4)
          {
            printf("%8.1f\n", t4);
            t4 *= 10.0;
          }
#else
        printf("%8.1f\n", ampval[i]);
#endif
      }

    k = 0;
    for(i=1; i<asize; i++)
      if(ampres[(i*3)+2] < ampres[(k*3)+2])
        k = i;
    den[1] = ampval[k];
    if(ConBode(num, den, 1, 2, freq, size, &tmag, &tphs))
      {
        fprintf(stderr, "Error encountered in ConBode (%d)\n", __LINE__);
        free(buffer);
        free(freq);
        free(ampval);
        free(ampres);
        return(STATUS_NOK);
      }
    t1 = 0.0;
    for(k=0; k<dcavg; k++)
      t1 += (mag[k]-tmag[k]);
    t1 /= ((float) dcavg);
    t2 = (float) pow(10.0, (double) (t1/20.0));
    printf("%s:  A1 = %.3f, A2 = %.3f\n", bearname, t2*ampgain, den[1]);

    free(buffer);
    free(freq);
    free(ampval);
    free(ampres);
    free(tmag);
    free(tphs);
```

```
  return(STATUS_OK);

}


int ConBode(float *num, float *den, int numsize, int densize, float *freq,
            int fsize, float **mag, float **phase)
{
  int i, j, k;
  float *mg, *ph, inc;
  float frq, ansr, ansi;

  if((mg = (float *) calloc(fsize, sizeof(float))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigBode.\n");
      return(1);
    }

  if((ph = (float *) calloc(fsize, sizeof(float))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigBode.\n");
      free(mg);
      return(1);
    }

  for(i=0; i<fsize; i++)
    {
      frq = 2.0 * PI * freq[i];
      mg[i] = ph[i] = ansi = 0.0;
      ansr = num[numsize-1];
      for(j=0; j<numsize-1; j++)
        {
          if(num[j] != 0.0)
            {
              k = numsize - 1 - j;
              if((k/2)%2 == 1)
                inc = -1.0;
              else
                inc = 1.0;
              if(k%2 == 1)
                ansi += (inc * num[j] * ((float) pow(frq, (double) k)));
              else
                ansr += (inc * num[j] * ((float) pow(frq, (double) k)));
            }
        }
      mg[i] += (20.0 * log10(sqrt(pow(ansr,2.0) + pow(ansi,2.0))));
      ph[i] += (180.0 * atan2(ansi, ansr) / PI);
      ansi = 0.0;
      ansr = den[densize-1];
      for(j=0; j<densize-1; j++)
        {
          if(den[j] != 0.0)
             {
              k = densize - 1 - j;
              if((k/2)%2 == 1)
                inc = -1.0;
              else
                inc = 1.0;
              if(k%2 == 1)
                ansi += (inc * den[j] * ((float) pow(frq, (double) k)));
              else
                ansr += (inc * den[j] * ((float) pow(frq, (double) k)));
             }
        }
      mg[i] -= (20.0 * log10(sqrt(pow(ansr,2.0) + pow(ansi,2.0))));
      ph[i] -= (180.0 * ((float) atan2(ansi, ansr)) / PI);
    }
  if(mag)
    (*mag) = mg;
```

```
  if(phase)
    {
      for(i=0; i<fsize; i++)
        {
          while(fabs(ph[i-1]-ph[i]) > 180.0)
            {
              frq = ph[i] - 360.0;
              if(fabs(ph[i-1]-ph[i]) > fabs(ph[i-1]-frq))
                ph[i] = frq;
              else
                ph[i] += 360.0;
            }
        }
      (*phase) = ph;
    }
  return(0);
}
```

# Appendix F

# Static Experimental Plots




**F-1** System Block Diagram and Test Points

This appendix presents graphically the closed loop response and the disturbance rejection plots of each axis when the turbopump rotor is not spinning. The first section describes the specifics of how the plots were obtained. The next two sections present these graphs for the analog and the digital controller respectively. The fourth section presents the analog, digital, and theoretical responses superimposed to aid in comparing the various controllers. The various plots were obtained using the Hewlett Packard HP 3562A Dynamic System Analyzer using the appropriate test points. Refer to Figure F-1 for the location of the test points in relation to the system components. The final two sections present important performance measures obtained from the plots in this appendix.

# F.1 Graph Production Details

## F.1.1 Closed Loop Frequency Response Details

The closed loop frequency graphs were obtained by inputting a swept sine wave having an amplitude of 0.01 Volts and ranging from 0.1 to 10000 Hz. For the analog controller, the swept sine wave was applied at the position signal adder. For the digital controller, the swept sine wave was applied to an auxiliary adder which was placed between the position signal test point and the A/D. The output for both controllers was obtained by monitoring the position signal test point.

## F.1.2 Disturbance Rejection Plot Details

The disturbance rejection plots were obtained by inputting a swept sine wave having an amplitude of 0.03 Volts and ranging from 0.1 to 10000 Hz. For both the analog and digital controller, the swept sine wave was applied at the control signal adder and the output was obtained by monitoring the position signal test point. The magnitude graph obtained using the system analyzer has units of V/V which must be converted using the following conversion factor,

Axial

$$\left(\frac{2h_0^2}{\mu_0 N^2 A}\right)\bullet\left(\frac{1\times10^6\,\mu m}{m}\right)\bullet\left(\begin{array}{c}Sensor\\Conversion\\Factor\end{array}\right)$$

Radial

$$\left(\frac{m h_0^2}{2\mu_0 N^2 A C_1 \cos\beta I_0}\right)\bullet\left(\frac{1\times10^6\,\mu m}{m}\right)\bullet\left(\begin{array}{c}Sensor\\Conversion\\Factor\end{array}\right)$$

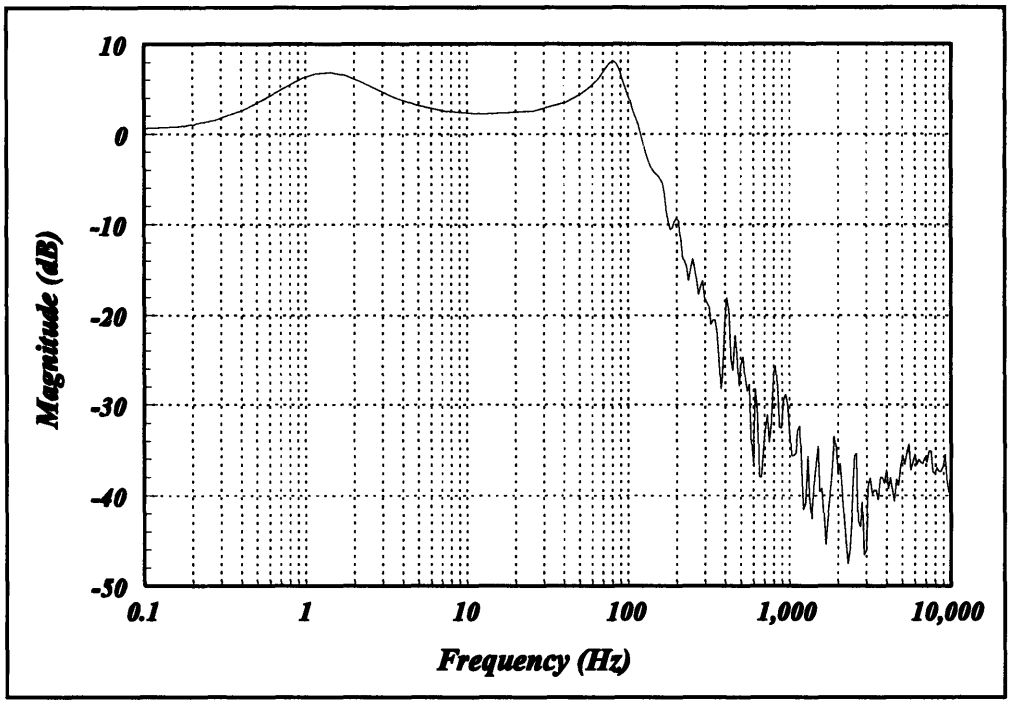
The values of the conversion factor for each axis is shown below.

| | Axial | Rad1X | Rad1Y | Rad2X | Rad2Y |
|---|---|---|---|---|---|
| Conversion Factor | 2.1704 | 1.9147 | 1.9147 | 0.9712 | 0.9712 |

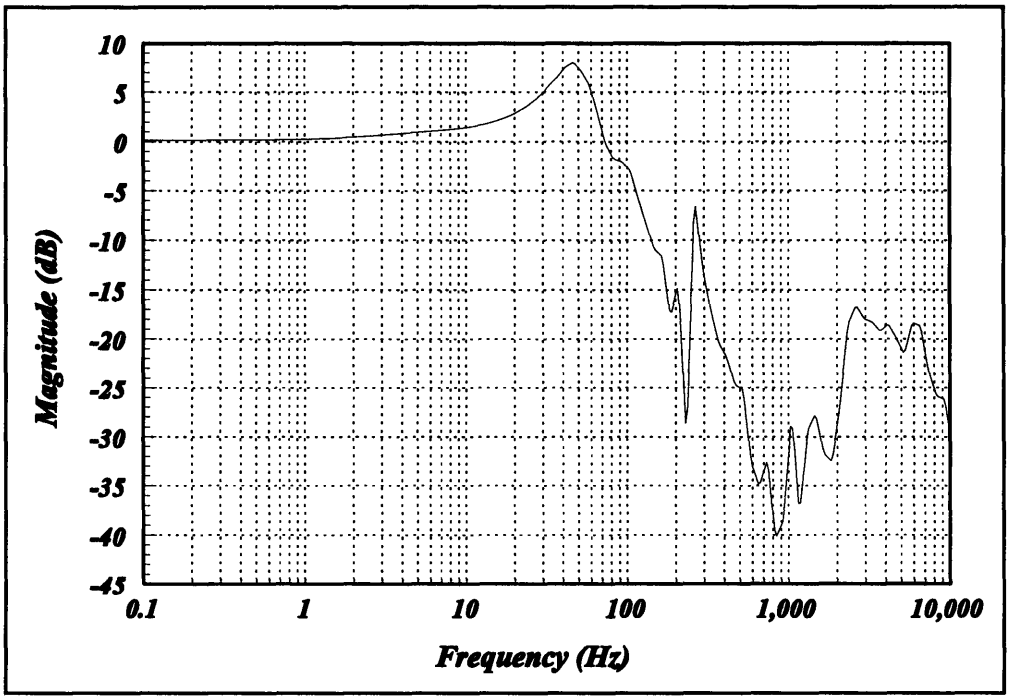
Only the magnitude plots will be presented in the remainder of this appendix as the phase plots convey little useful information.

# F.2 Closed Loop Frequency Response

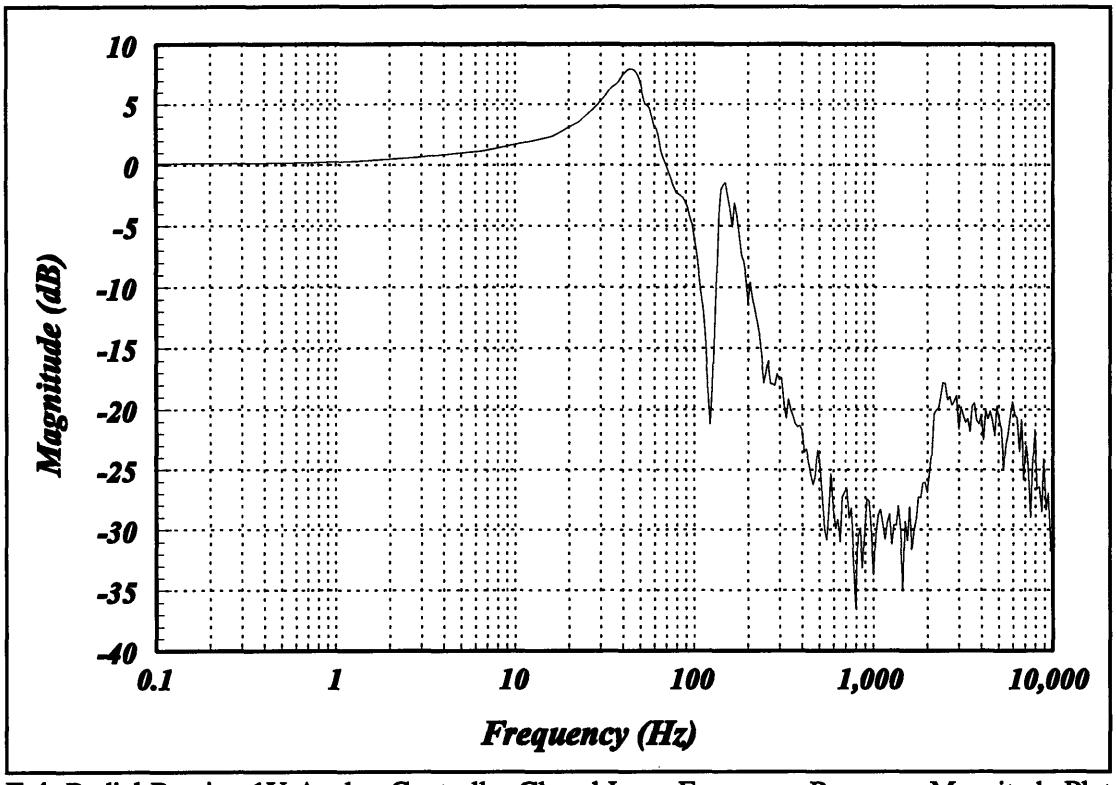
## F.2.1 Analog Controller



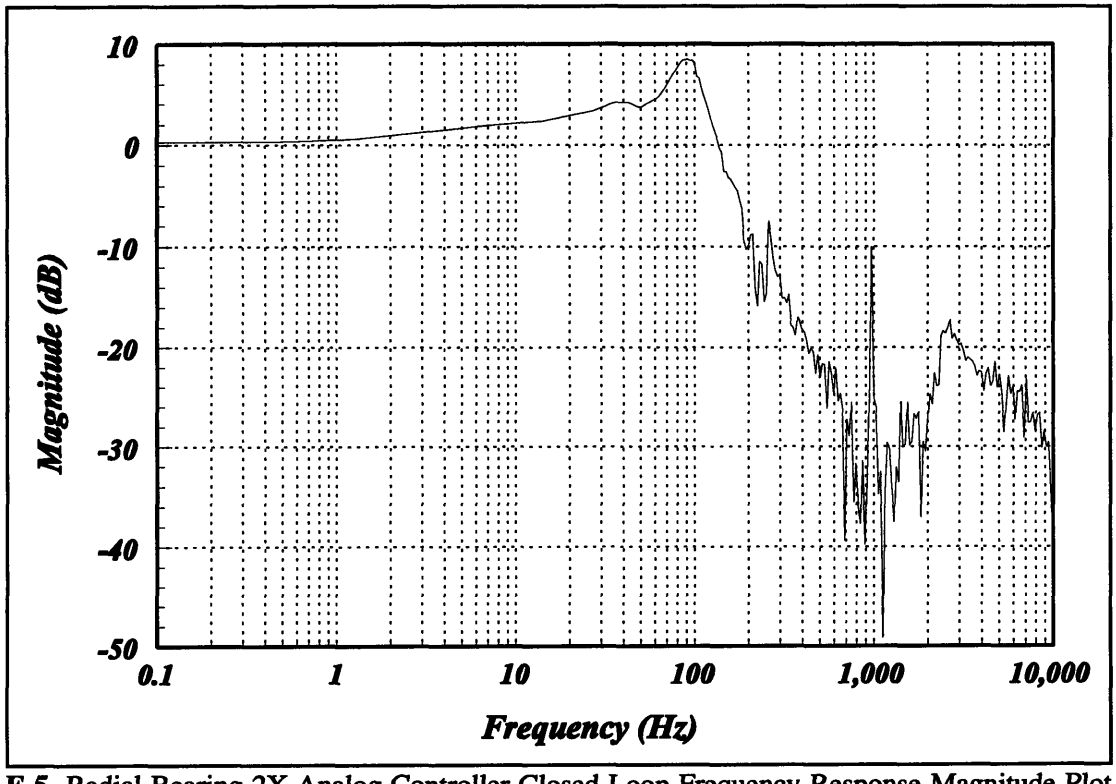

F-2 Axial Bearing Analog Controller Closed Loop Frequency Response Magnitude Plot



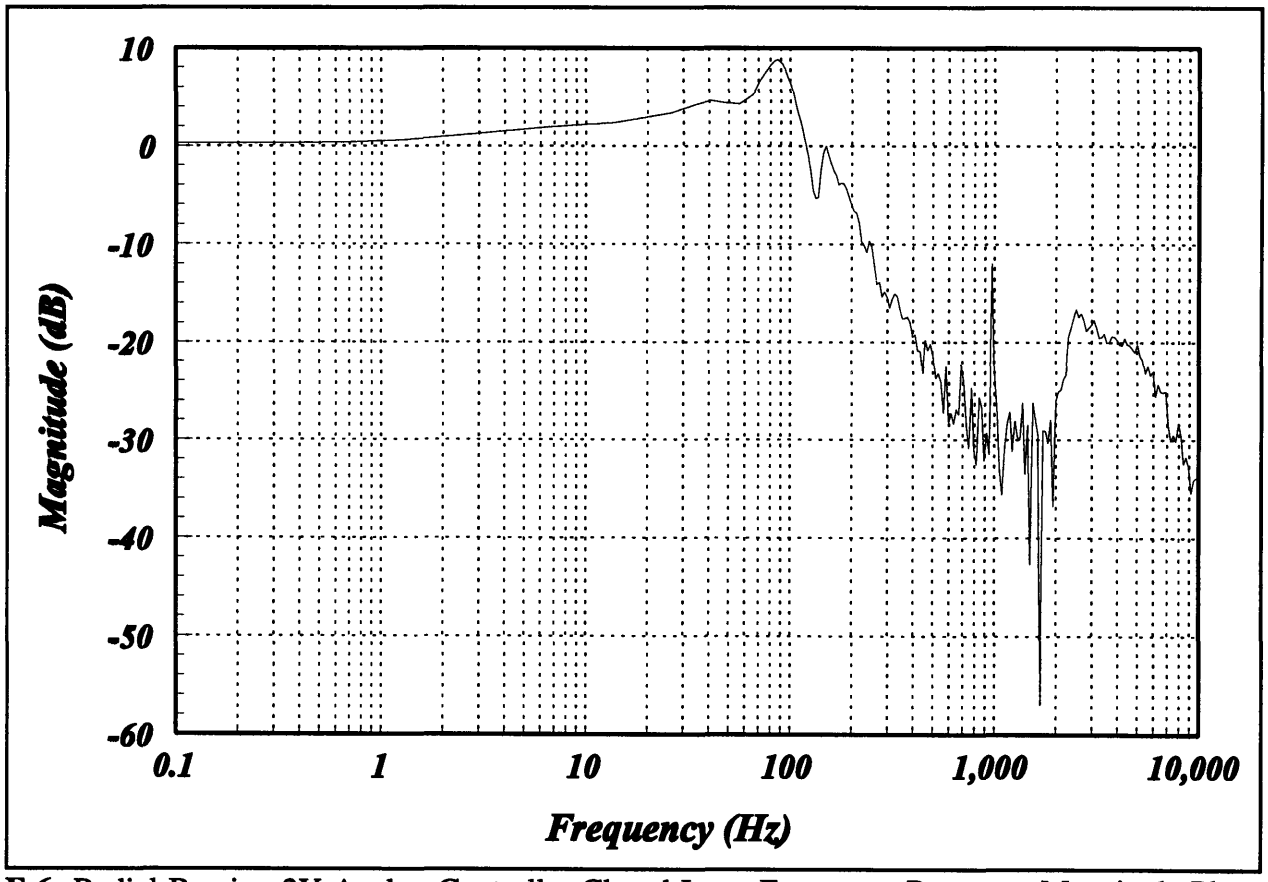

F-3 Radial Bearing 1X Analog Controller Closed Frequency Response Magnitude Plot




**F-4** Radial Bearing 1Y Analog Controller Closed Loop Frequency Response Magnitude Plot

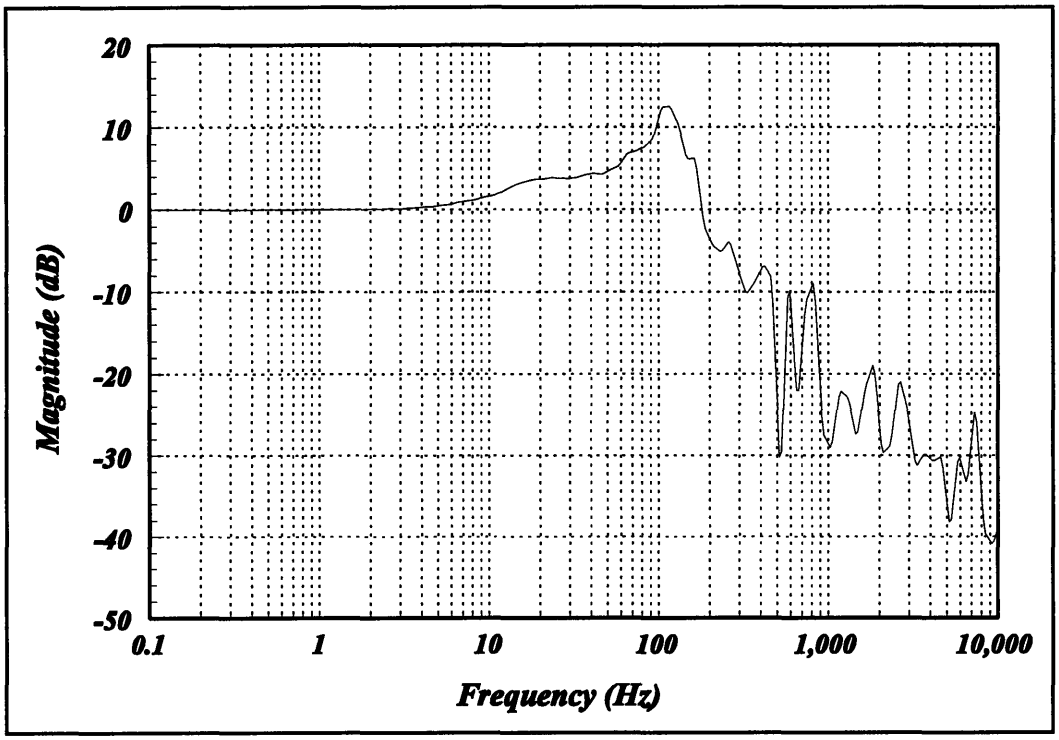



**F-5** Radial Bearing 2X Analog Controller Closed Loop Frequency Response Magnitude Plot

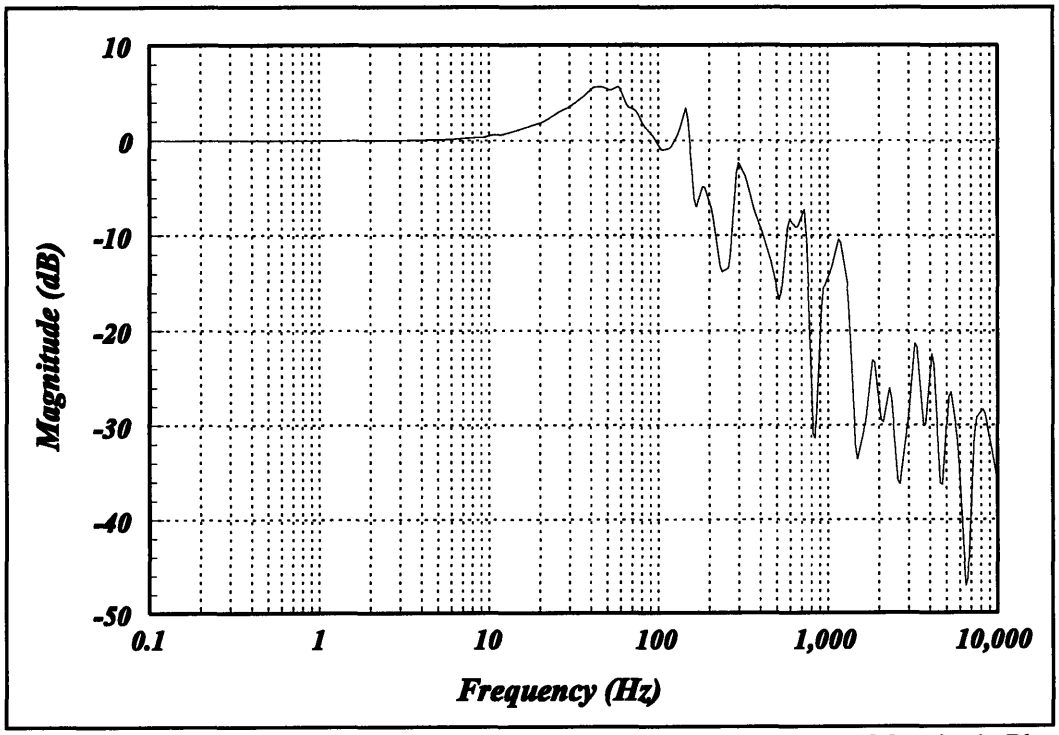



**F-6** Radial Bearing 2Y Analog Controller Closed Loop Frequency Response Magnitude Plot

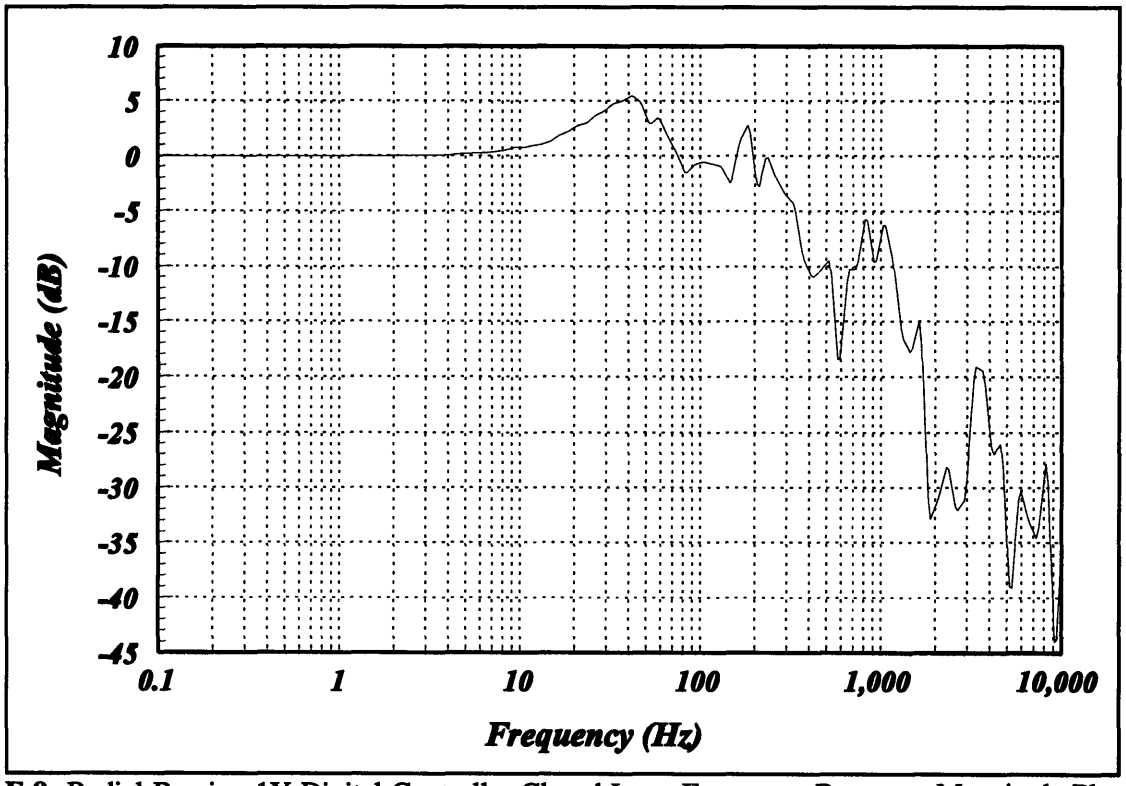
## F.2.2 Digital Controller



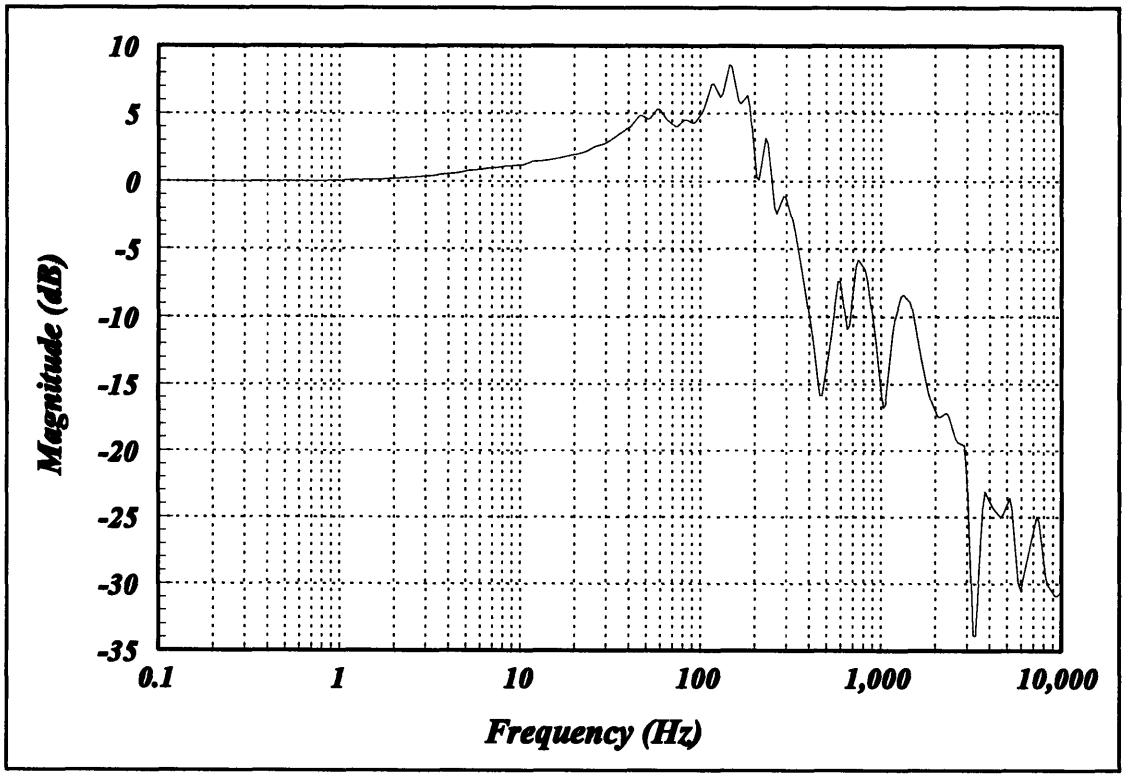

**F-7** Axial Bearing Digital Controller Closed Loop Frequency Response Magnitude Plot



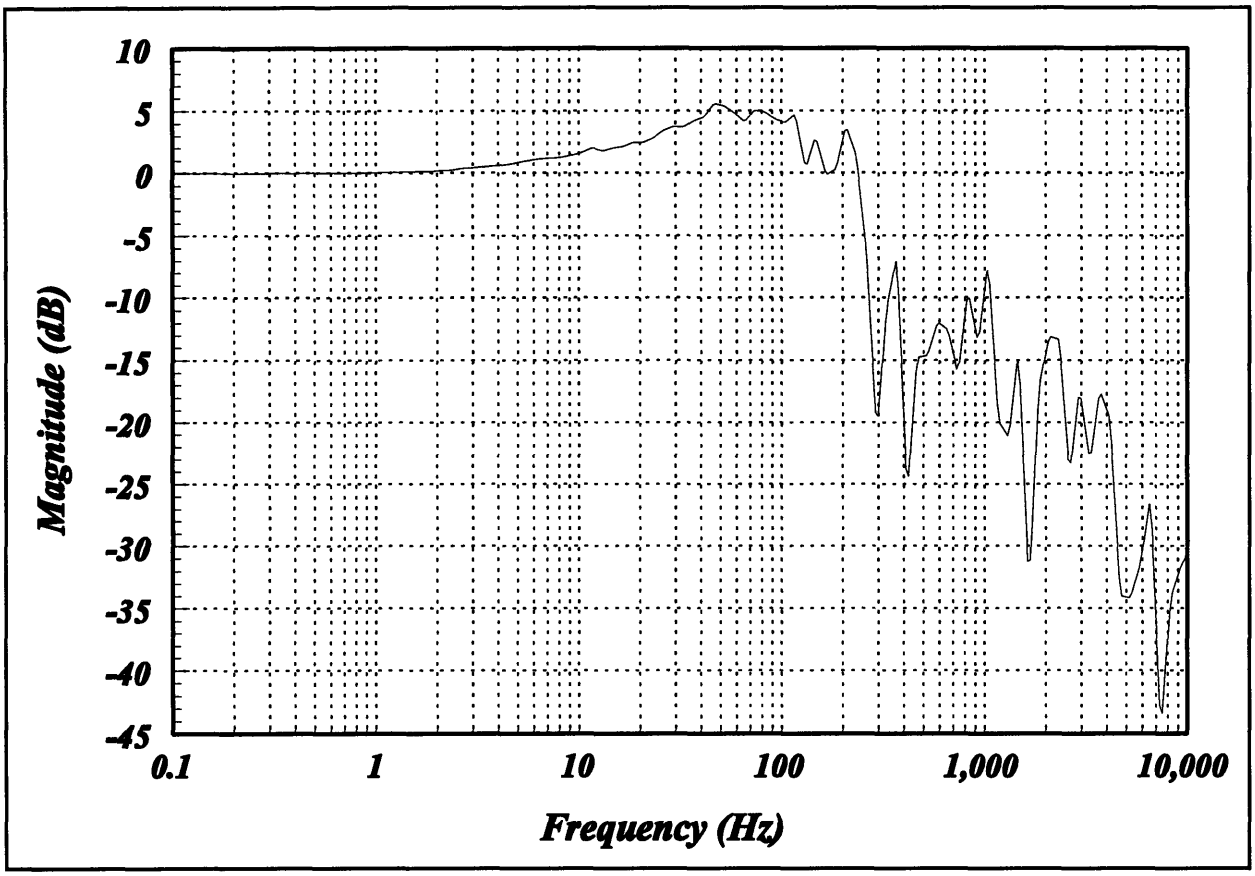

**F-8** Radial Bearing 1X Digital Controller Closed Frequency Response Magnitude Plot




**F-9** Radial Bearing 1Y Digital Controller Closed Loop Frequency Response Magnitude Plot



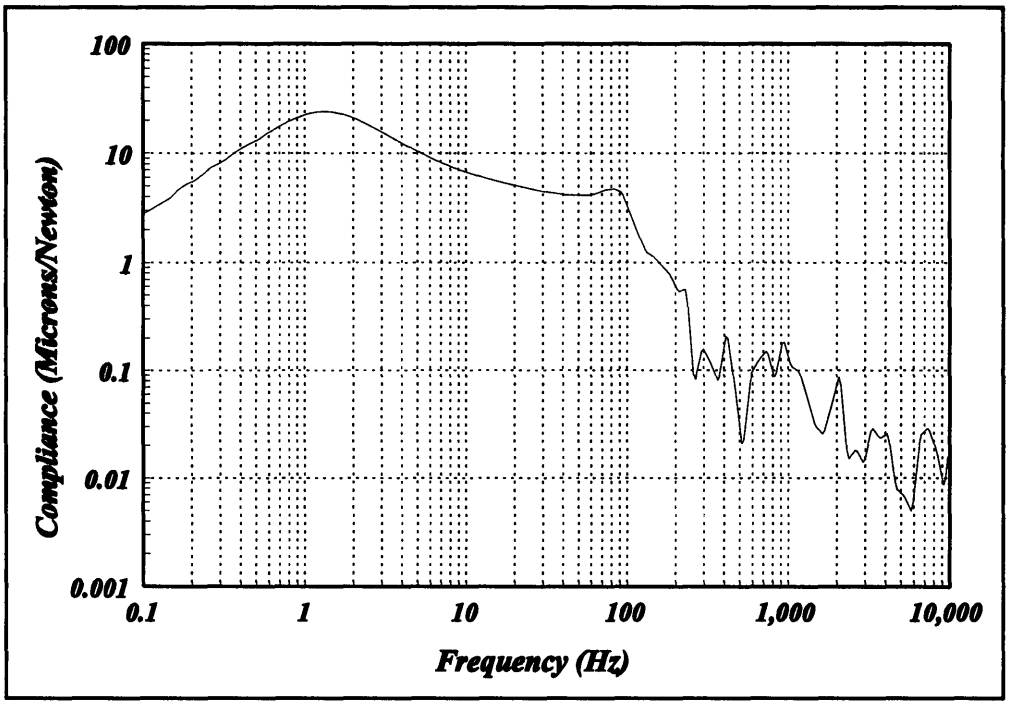

**F-10** Radial Bearing 2X Digital Controller Closed Loop Frequency Response Magnitude Plot



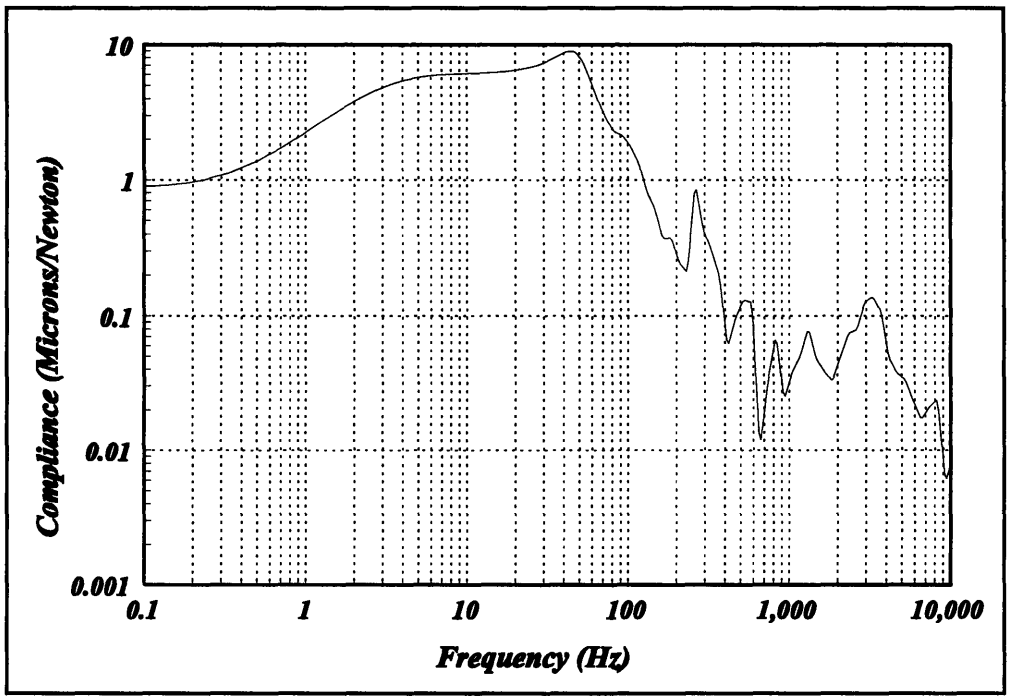

**F-11** Radial Bearing 2Y Digital Controller Closed Loop Frequency Response Magnitude Plot

# F.3 Disturbance Rejection

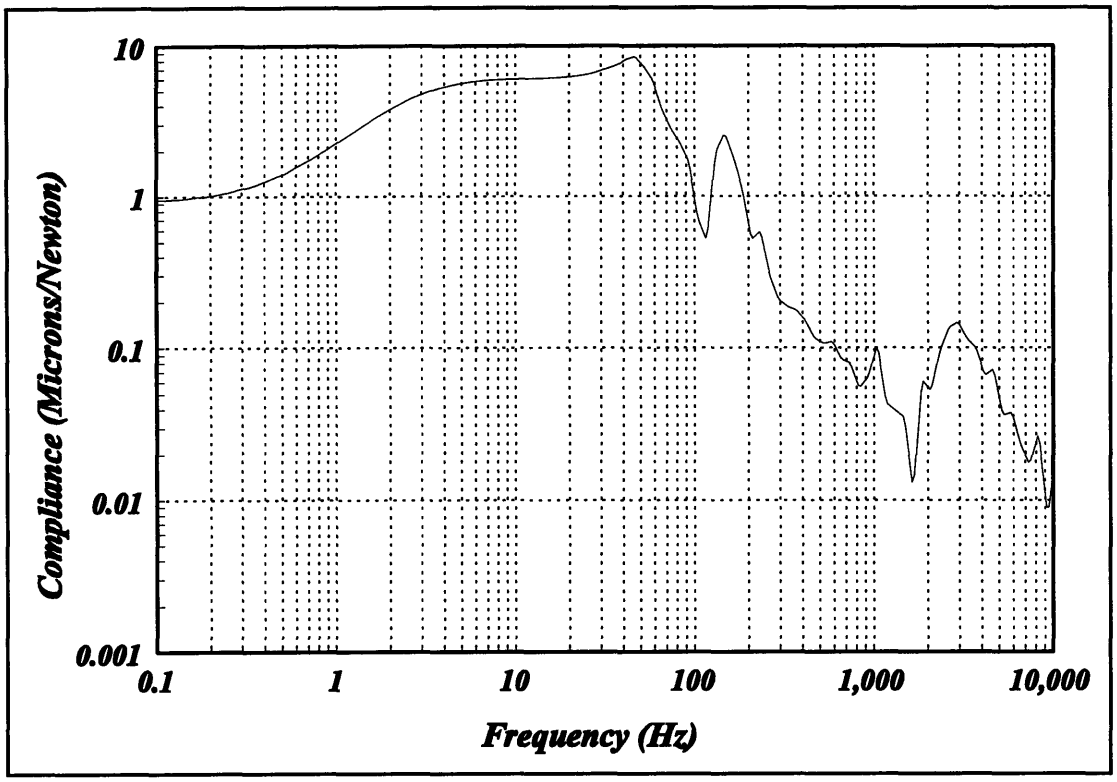
## F.3.1 Analog Controller



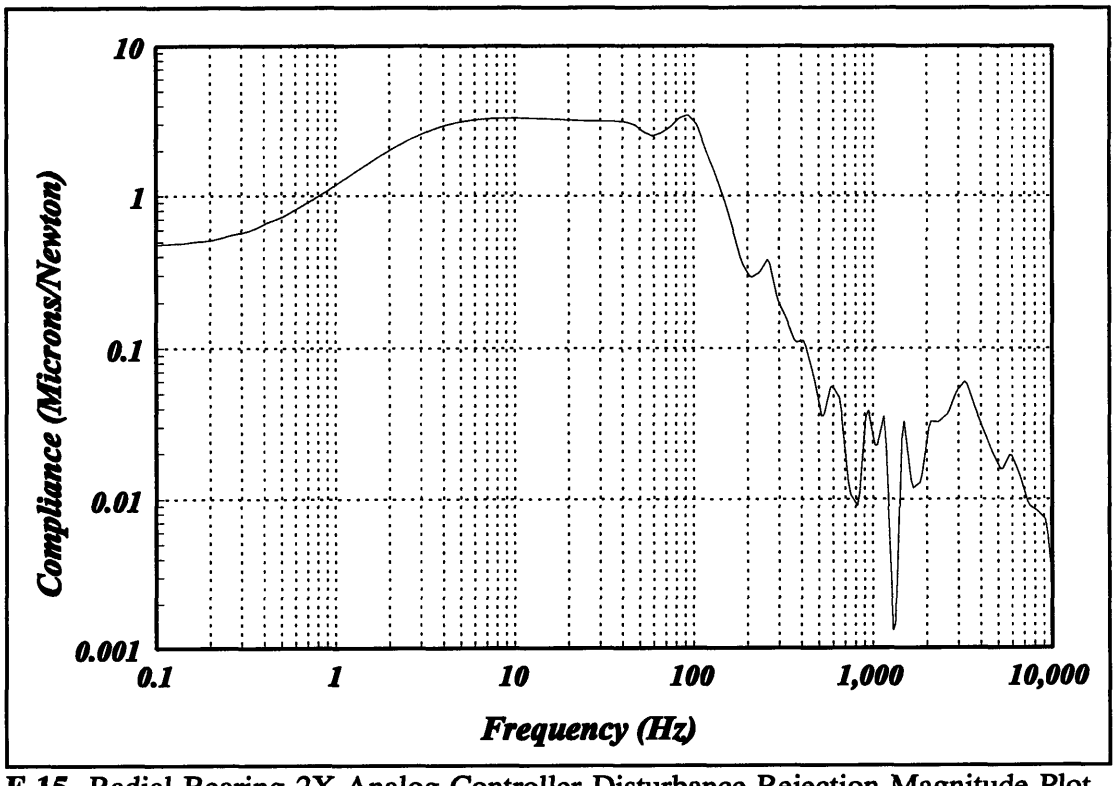

**F-12** Axial Bearing Analog Controller Disturbance Rejection Magnitude Plot



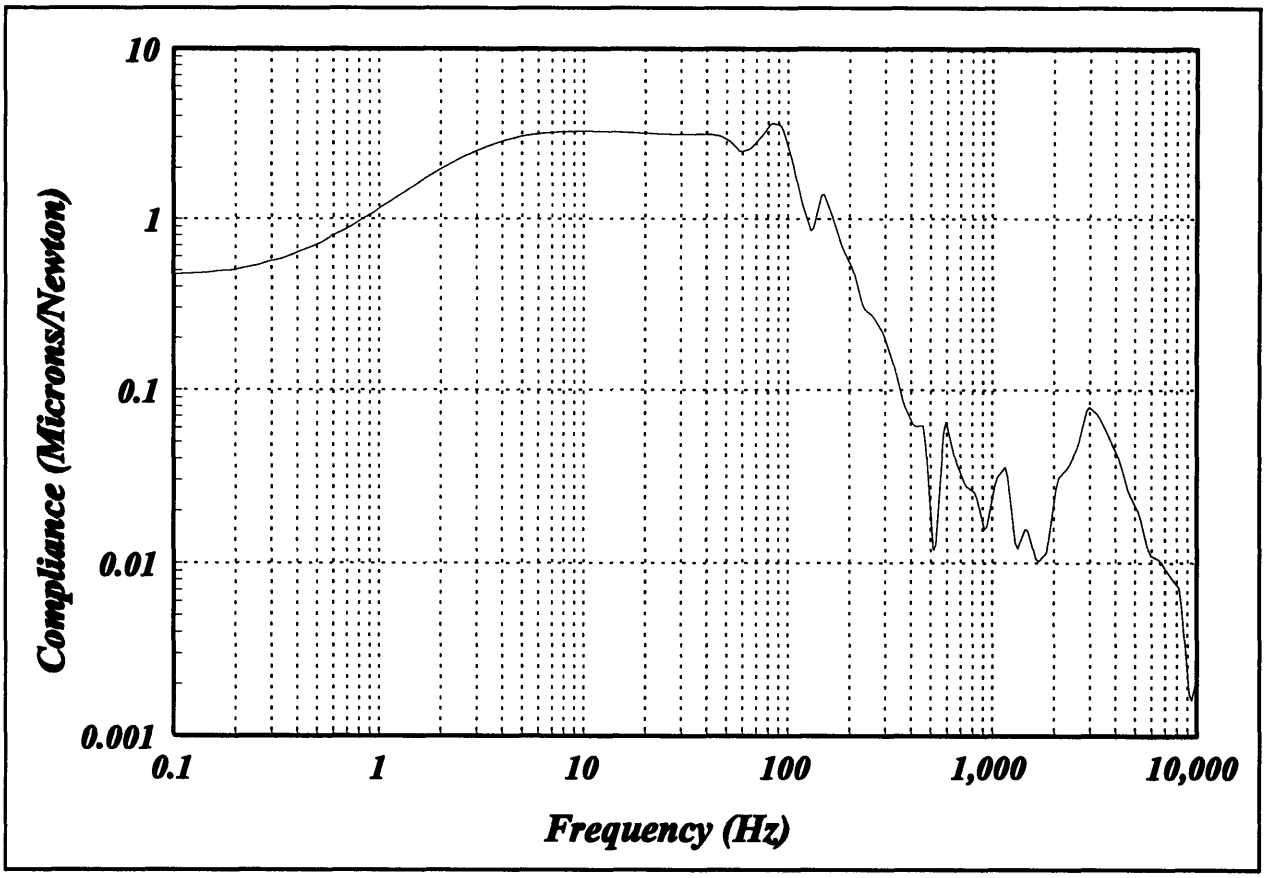

**F-13** Radial Bearing 1X Analog Controller Disturbance Rejection Magnitude Plot




**F-14** Radial Bearing 1Y Analog Controller Disturbance Rejection Magnitude Plot

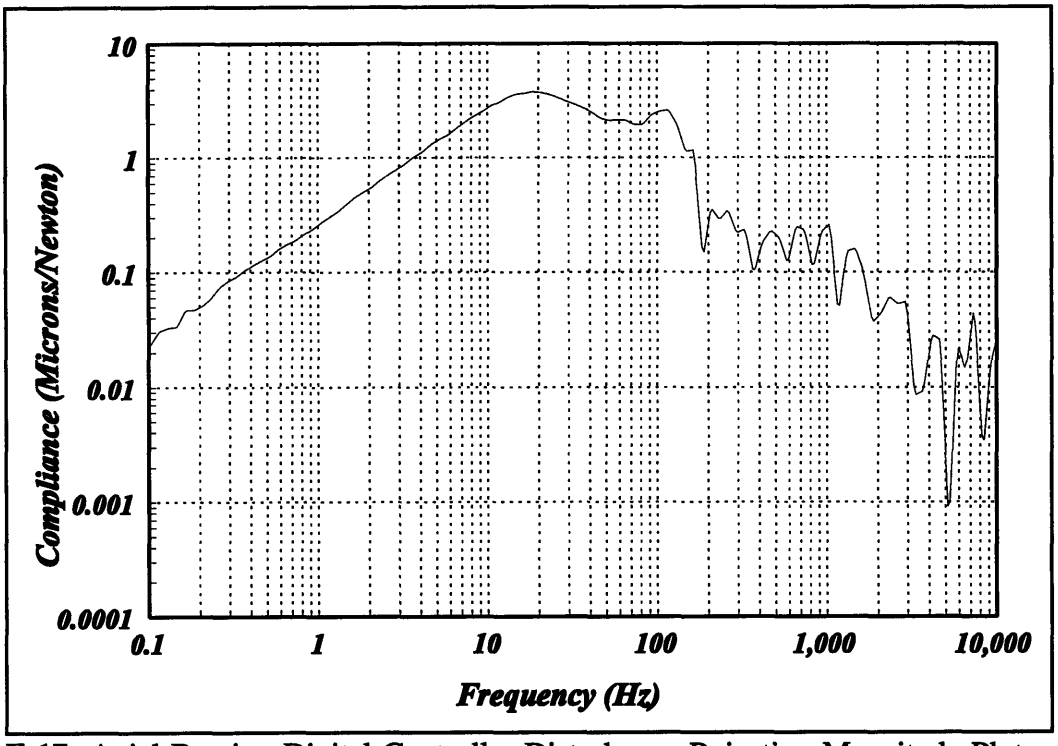



**F-15** Radial Bearing 2X Analog Controller Disturbance Rejection Magnitude Plot

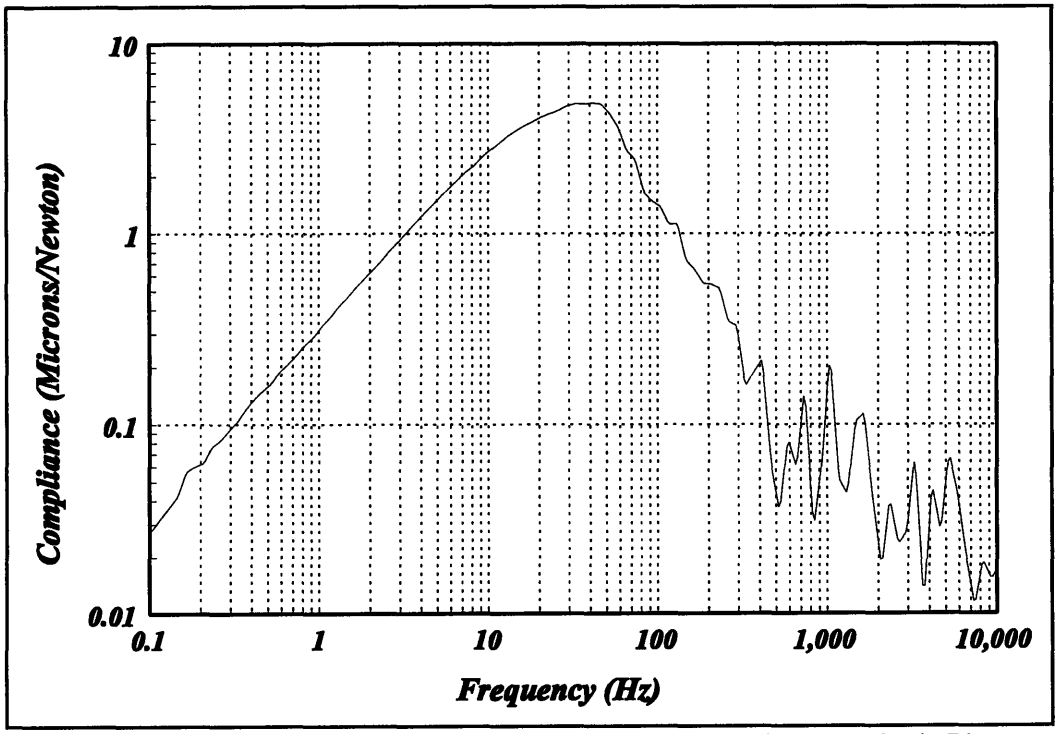



**F-16** Radial Bearing 2Y Analog Controller Disturbance Rejection Magnitude Plot

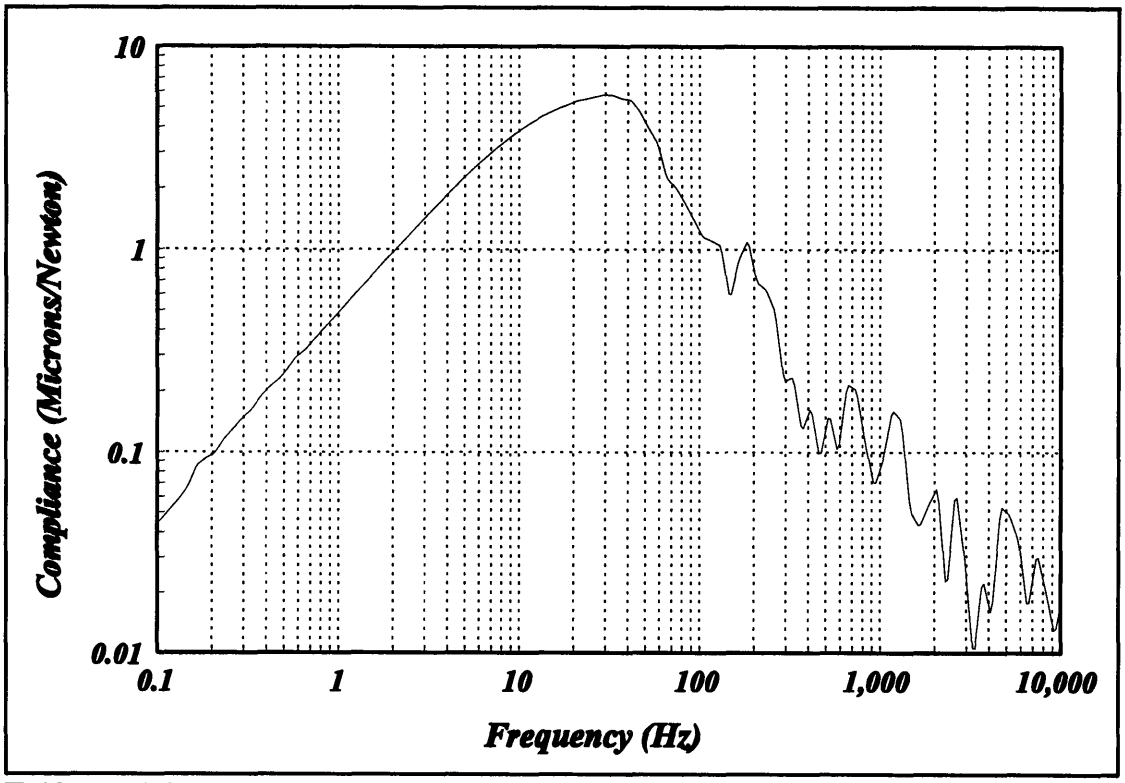
## F.3.2 Digital Controller



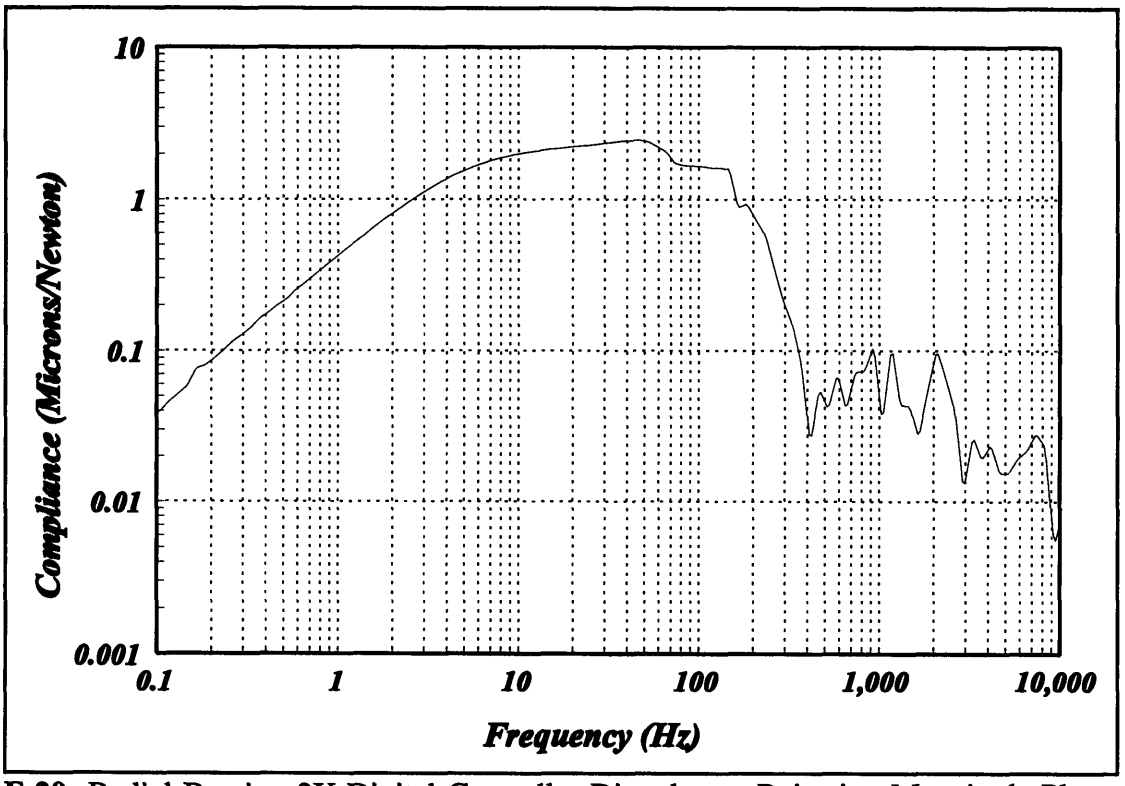

**F-17** Axial Bearing Digital Controller Disturbance Rejection Magnitude Plot



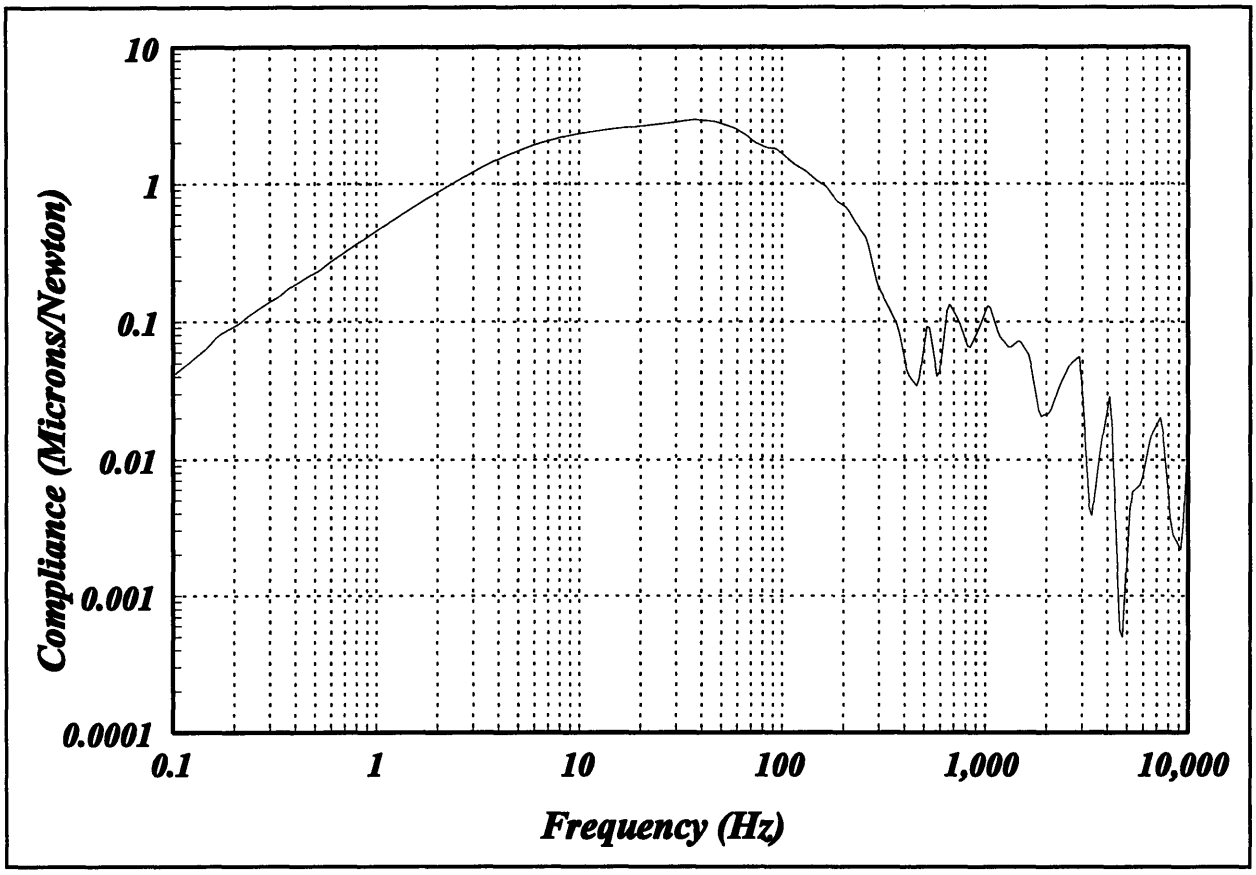

**F-18** Radial Bearing 1X Digital Controller Disturbance Rejection Magnitude Plot




**F-19** Radial Bearing 1Y Digital Controller Disturbance Rejection Magnitude Plot

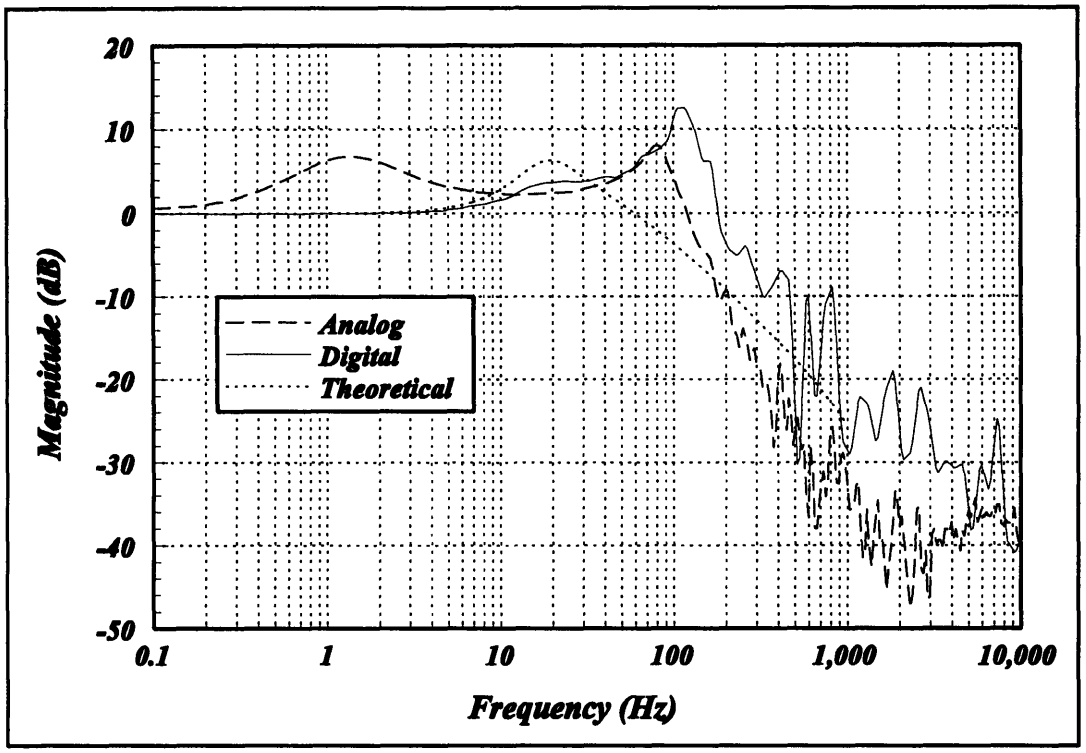



**F-20** Radial Bearing 2X Digital Controller Disturbance Rejection Magnitude Plot

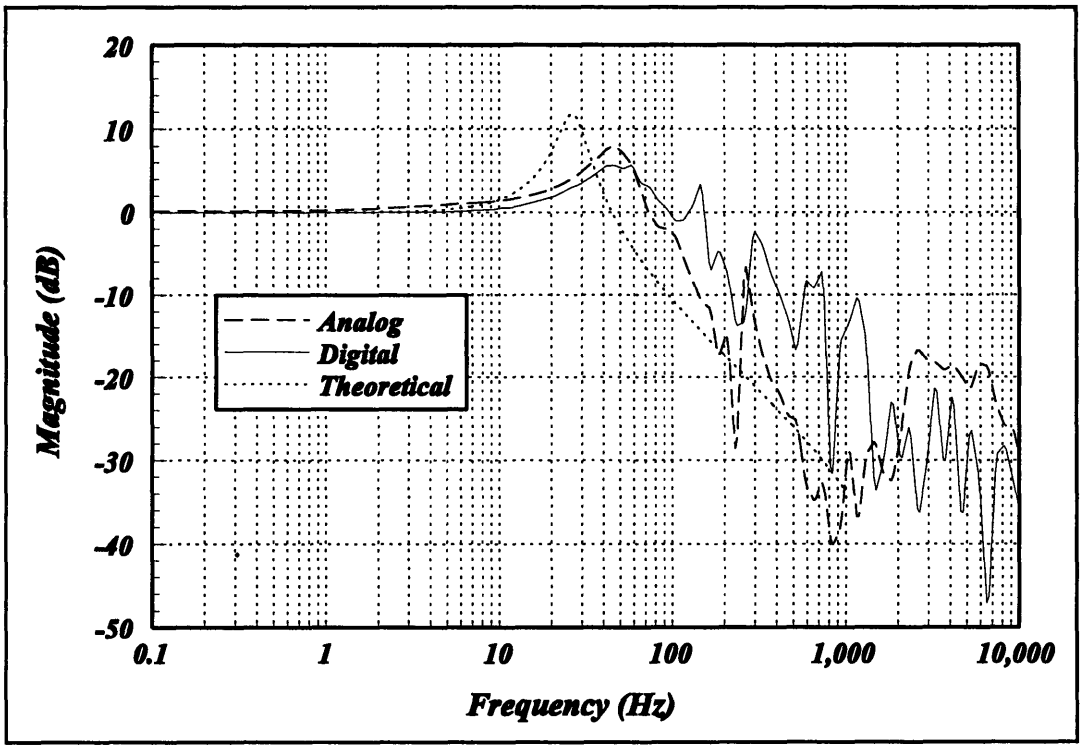



**F-21** Radial Bearing 2Y Digital Controller Disturbance Rejection Magnitude Plot

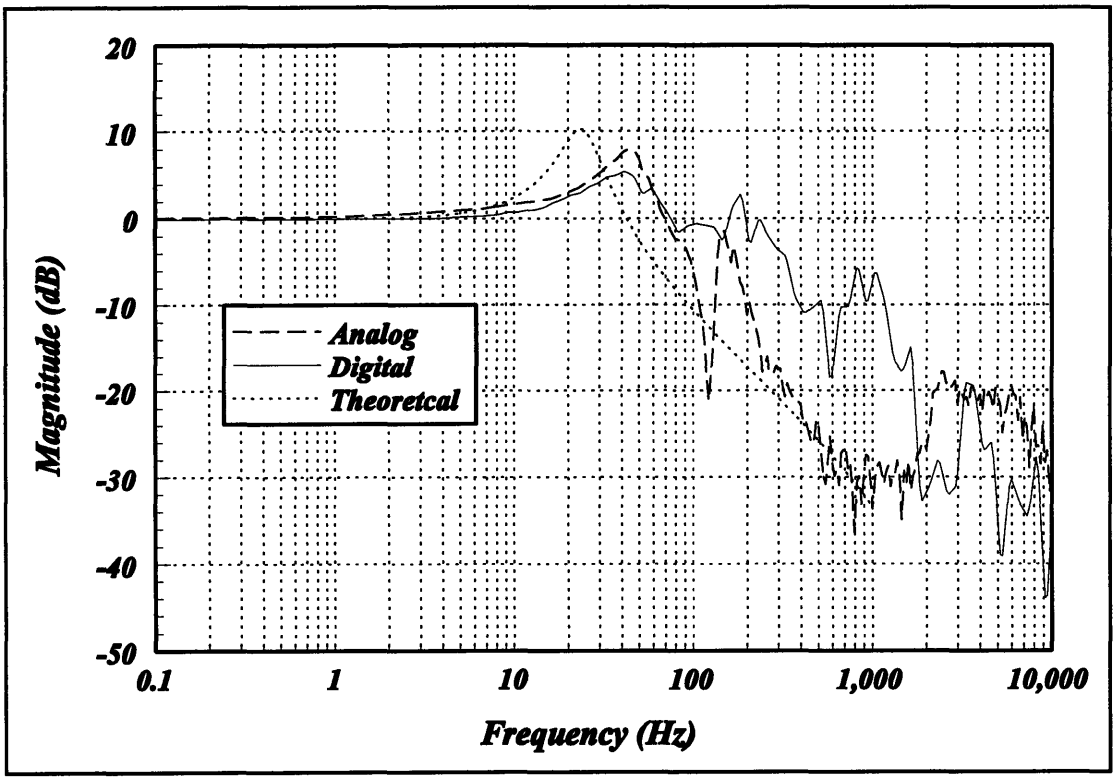
# F.4 Closed Loop Frequency Response Comparison



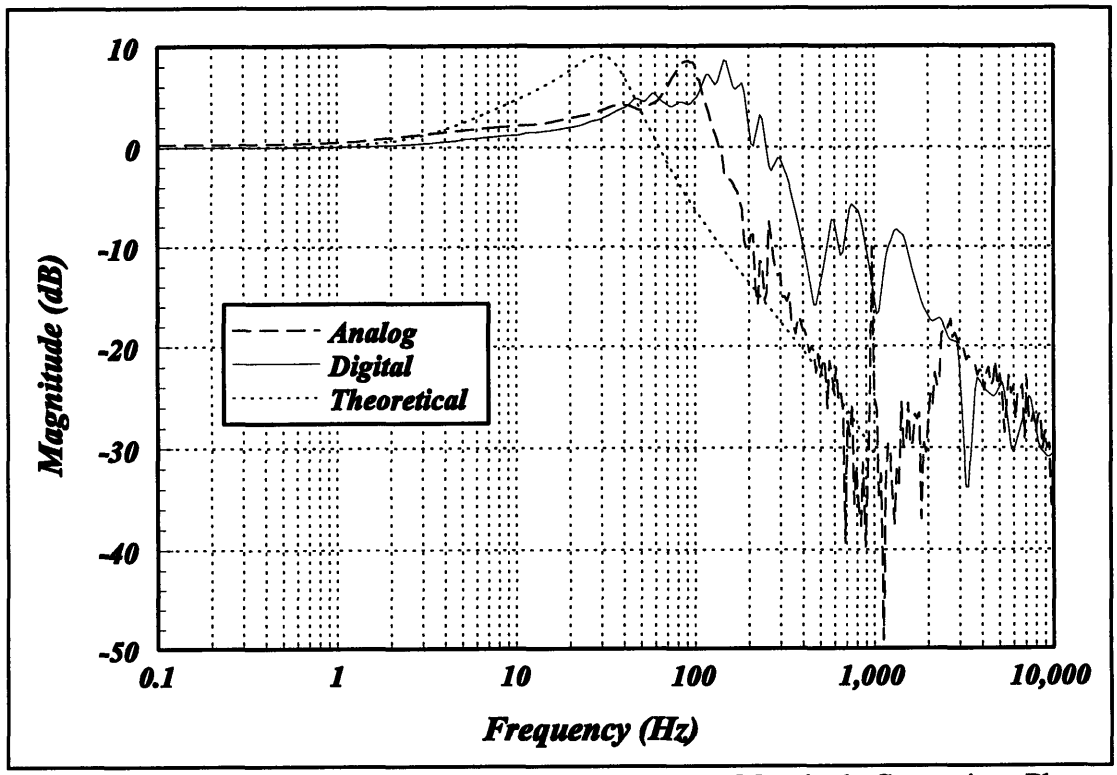

F-22 Axial Bearing Closed Loop Frequency Response Magnitude Comparison Plot



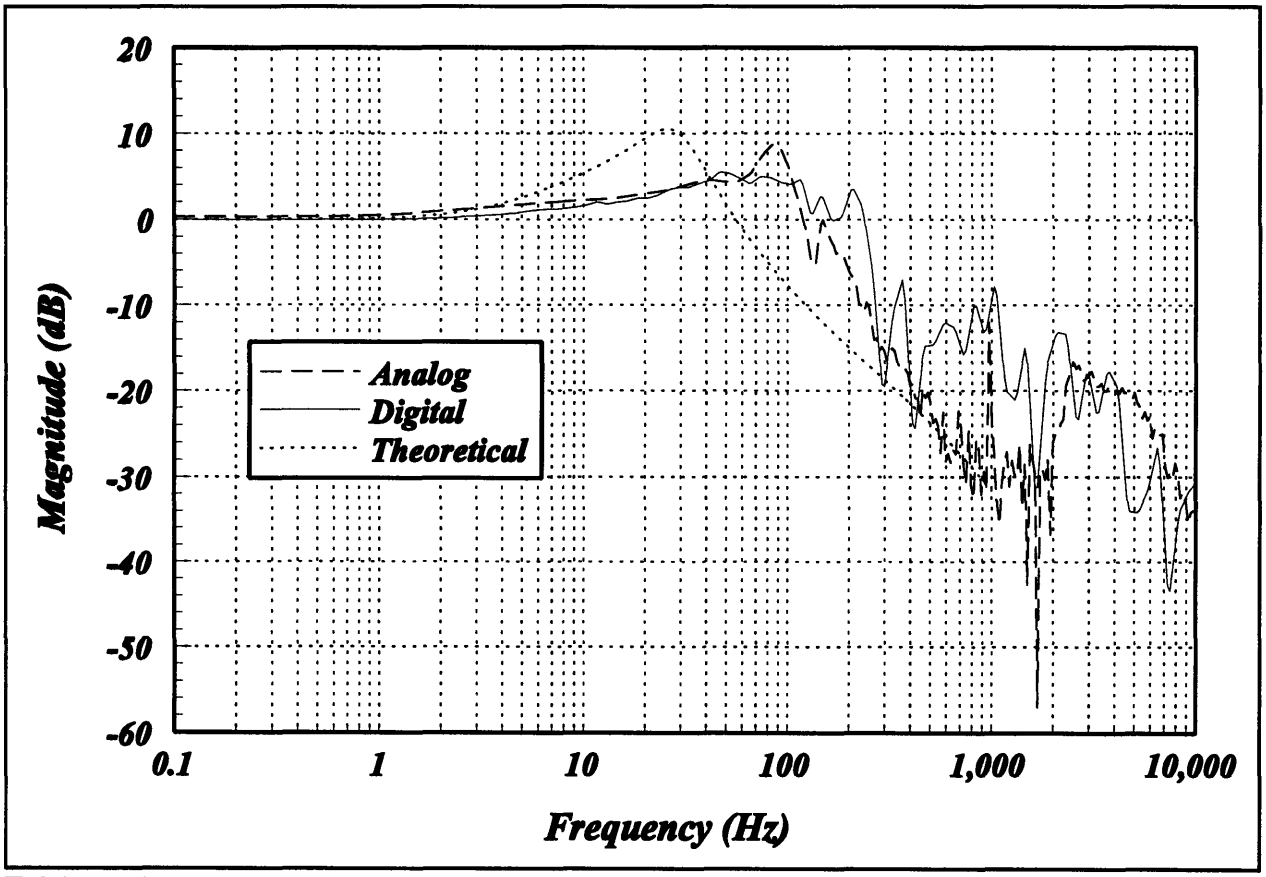

F-23 Radial Bearing 1X Closed Loop Frequency Response Magnitude Comparison Plot




**F-24** Radial Bearing 1Y Closed Loop Frequency Response Magnitude Comparison Plot

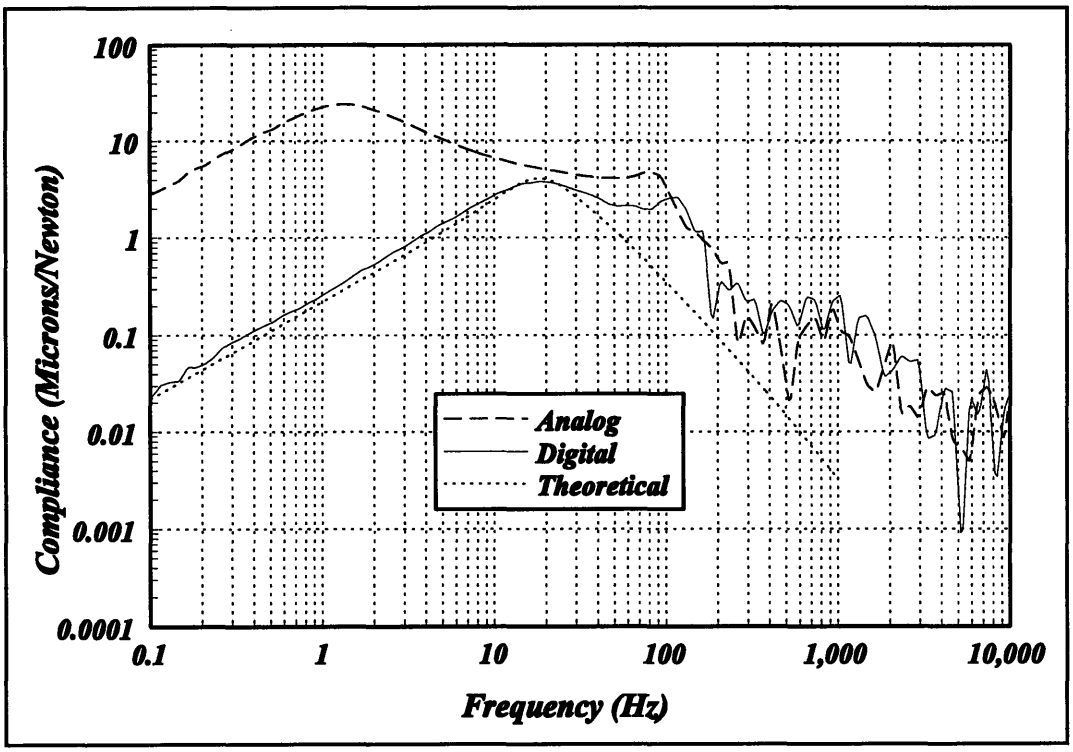



**F-25** Radial Bearing 2X Closed Loop Frequency Response Magnitude Comparison Plot

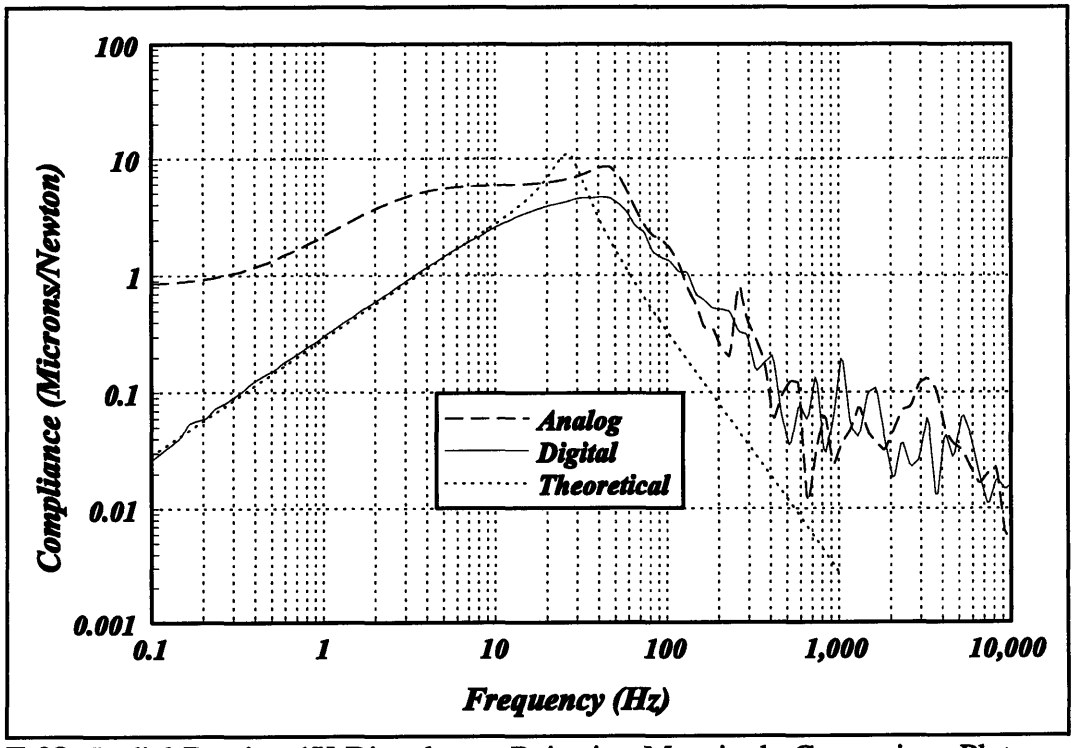



**F-26** Radial Bearing 2Y Closed Loop Frequency Response Magnitude Comparison Plot

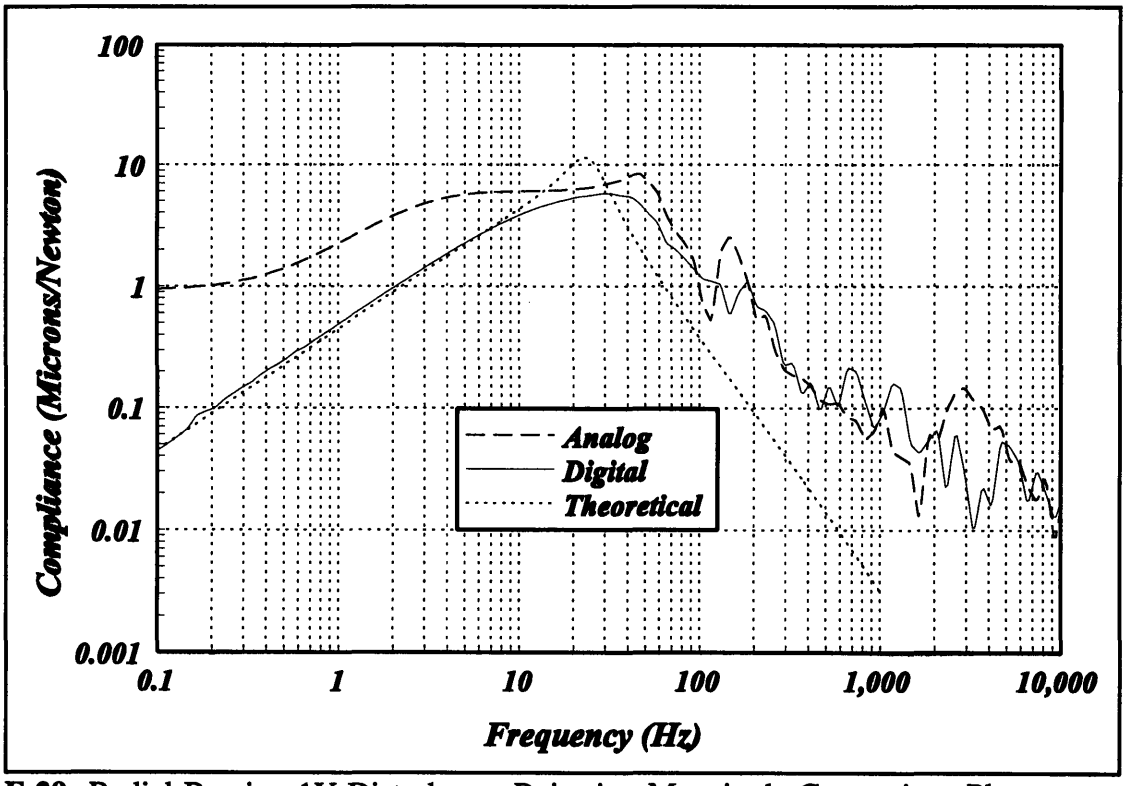
# F.5 Disturbance Rejection Comparison



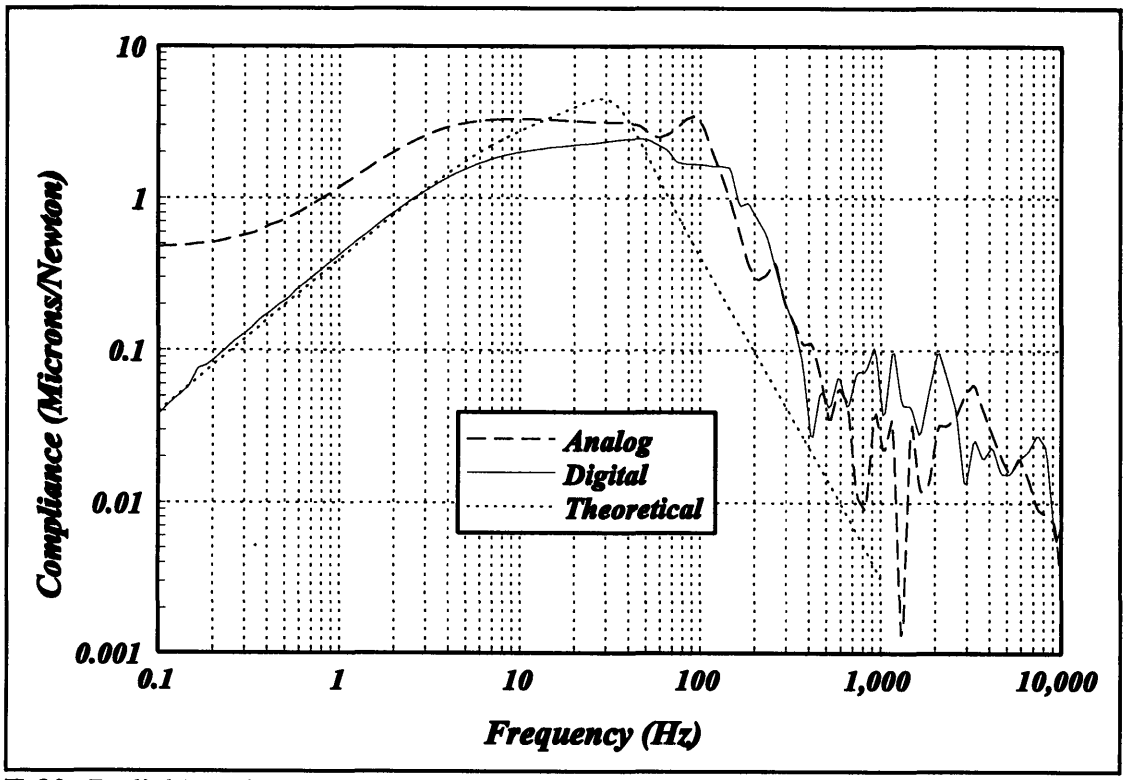

**F-27** Axial Bearing Disturbance Rejection Magnitude Comparison Plot



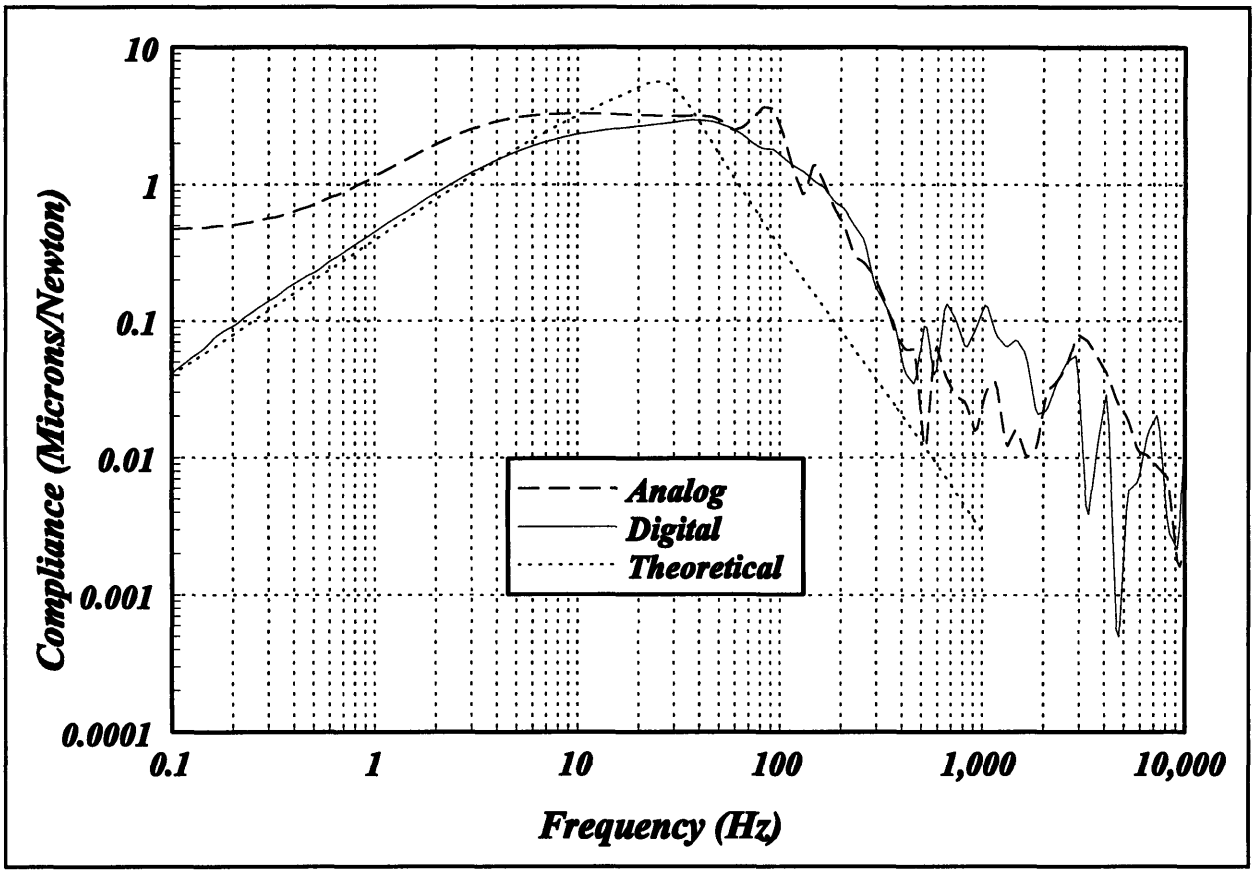

**F-28** Radial Bearing 1X Disturbance Rejection Magnitude Comparison Plot




**F-29** Radial Bearing 1Y Disturbance Rejection Magnitude Comparison Plot




**F-30** Radial Bearing 2X Disturbance Rejection Magnitude Comparison Plot




**F-32** Radial Bearing 2Y Disturbance Rejection Magnitude Comparison Plot

# F.6 Closed Loop Frequency Response Performance Values

| Parameter | Rad1X | | | Rad1Y | | |
|---|---|---|---|---|---|---|
| | Analog | Digital | Theoretical | Analog | Digital | Theoretical |
| Peak Gain (DB) | 8.00 | 5.68 | 11.79 | 7.94 | 5.41 | 7.49 |
| Bandwidth @ -3dB (Hz) | 104.06 | 157.93 | 53.85 | 88.96 | 283.09 | 51.82 |
| Gain @ 1000Hz (dB) | -31.61 | -14.35 | -33.72 | -33.73 | -7.38 | -33.53 |

| Parameter | Rad2X | | | Rad2Y | | |
|---|---|---|---|---|---|---|
| | Analog | Digital | Theoretical | Analog | Digital | Theoretical |
| Peak Gain (DB) | 4.19 | 4.84 | 9.13 | 8.78 | 5.53 | 10.51 |
| Bandwidth @ -3dB (Hz) | 152.46 | 326.15 | 77.35 | 167.88 | 249.09 | 68.63 |
| Gain @ 1000Hz (dB) | -25.19 | -15.27 | -30.68 | -23.41 | -9.47 | -31.61 |

| Parameter | Axial | | |
|---|---|---|---|
| | Analog | Digital | Theoretical |
| Peak Gain (DB) | 8.06 | 12.50 | 6.33 |
| Bandwidth @ -3dB (Hz) | 134.01 | 195.09 | 91.64 |
| Gain @ 1000Hz (dB) | -32.82 | -18.53 | -25.55 |

# F.7 Disturbance Rejection Performance Values

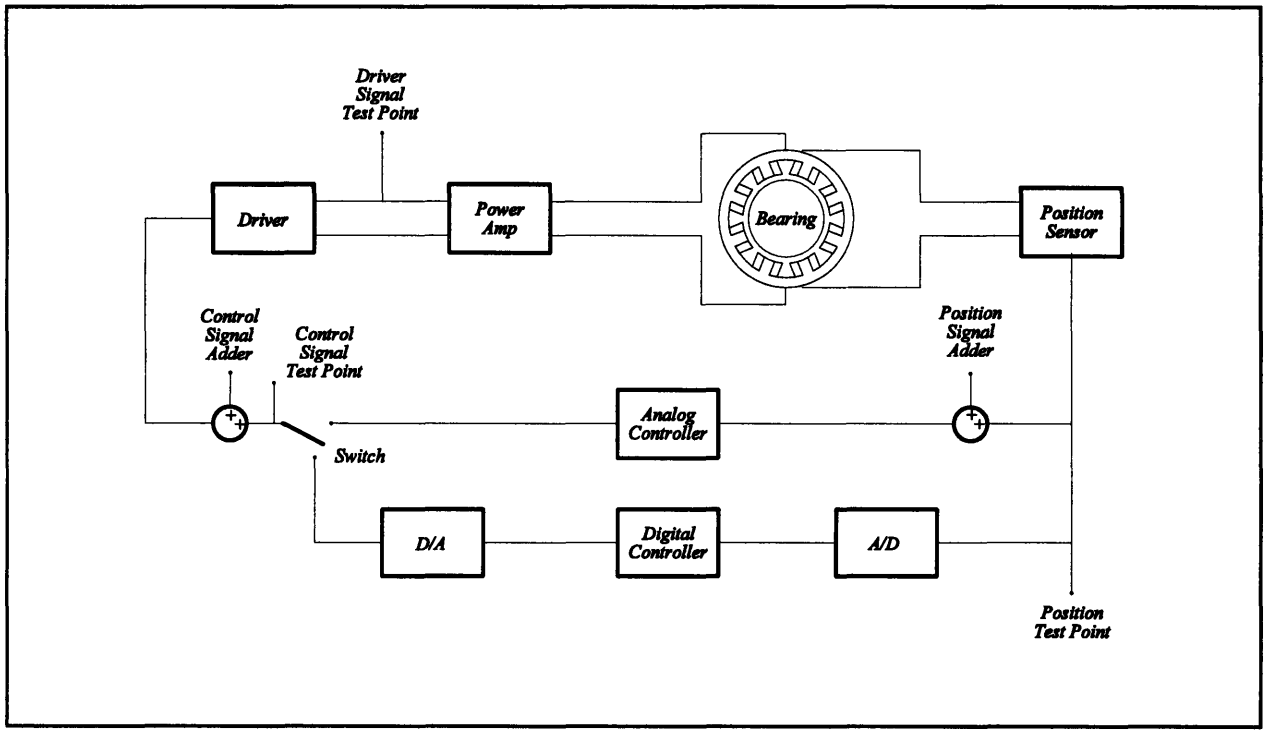
| Parameter | Rad1X | | | Rad1Y | | |
|---|---|---|---|---|---|---|
| | Analog | Digital | Theoretical | Analog | Digital | Theoretical |
| Peak Compliance (Microns/Newton) | 8.52 | 4.59 | 10.87 | 8.47 | 5.75 | 11.57 |
| Compliance @ 1000Hz (Microns/Newton) | 0.031 | 0.139 | 0.003 | 0.088 | 0.082 | 0.003 |

| Parameter | Rad2X | | | Rad2Y | | |
|---|---|---|---|---|---|---|
| | Analog | Digital | Theoretical | Analog | Digital | Theoretical |
| Peak Compliance (Microns/Newton) | 3.30 | 2.47 | 4.47 | 3.27 | 2.93 | 5.63 |
| Compliance @ 1000Hz (Microns/Newton) | 0.027 | 0.051 | 0.003 | 0.023 | 0.117 | 0.003 |

| Parameter | Axial | | |
|---|---|---|---|
| | Analog | Digital | Theoretical |
| Peak Compliance (Microns/Newton) | 24.14 | 3.80 | 4.11 |
| Compliance @ 1000Hz (Microns/Newton) | 0.135 | 0.247 | 0.003 |

# Appendix G

# Dynamic Experimental Plots




**G-1** System Block Diagram and Test Points

This appendix presents graphically the disturbance rejection plots of each axis when the turbopump rotor is spinning at 15000 and 28000 RPM. The first section describes the specifics of how the plots were obtained. The next two sections present the disturbance rejection graphs for the analog and the digital controller respectively. The fourth section presents the analog, and digital responses superimposed to aid in comparing the various controllers. The various plots were obtained using the Hewlett Packard HP 3562A Dynamic System Analyzer using the appropriate test points. Refer to Figure G-1 for the location of the test points in relation to the system components. The final two sections present important performance measures obtained from the plots in this appendix.

# G.1 Graph Production Details

The disturbance rejection plots were obtained by inputting a swept sine wave having an amplitude of 0.03 Volts and ranging from 0.1 to 10000 Hz. For both the analog and digital controller, the swept sine wave was applied at the control signal adder and the output was obtained by monitoring the position signal test point. The magnitude graph obtained using the system analyzer has units of V/V which must be converted using the following conversion factor,

Axial

$$\left(\frac{2h_0^2}{\mu_0 N^2 A}\right)\bullet\left(\frac{1\times10^6\,\mu m}{m}\right)\bullet\left(\begin{array}{c}Sensor\\Conversion\\Factor\end{array}\right)$$

Radial

$$\left(\frac{m h_0^2}{2\mu_0 N^2 A C_1 \cos\beta I_0}\right)\bullet\left(\frac{1\times10^6\,\mu m}{m}\right)\bullet\left(\begin{array}{c}Sensor\\Conversion\\Factor\end{array}\right)$$

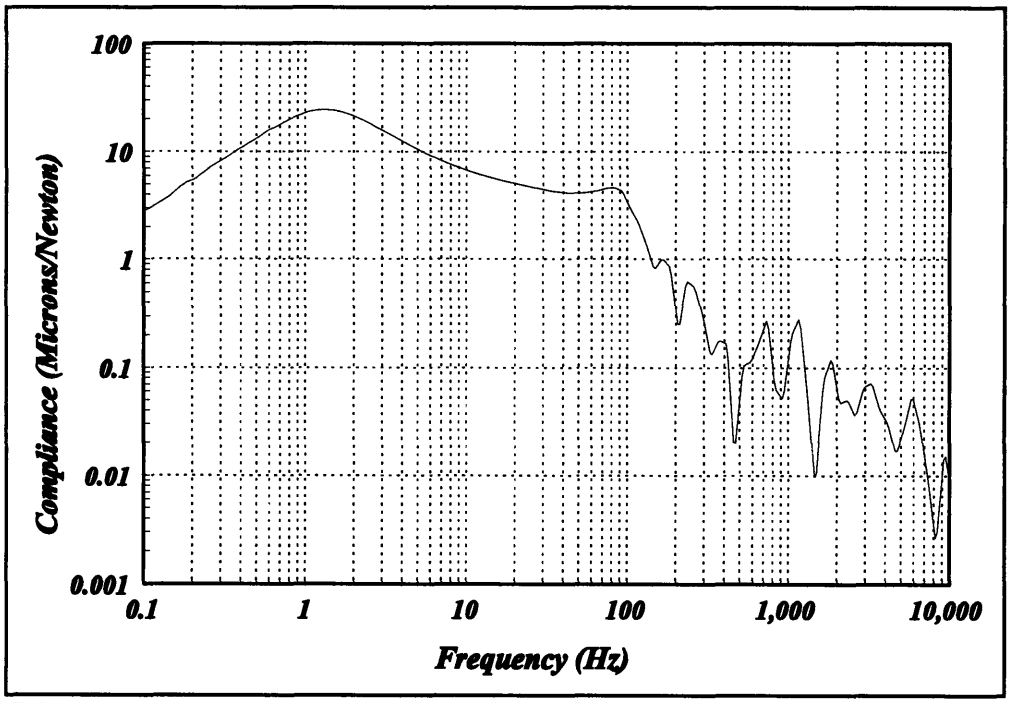
The values of the conversion factor for each axis is shown below.

| | Axial | Rad1X | Rad1Y | Rad2X | Rad2Y |
|---|---|---|---|---|---|
| Conversion Factor | 2.1704 | 1.9147 | 1.9147 | 0.9712 | 0.9712 |

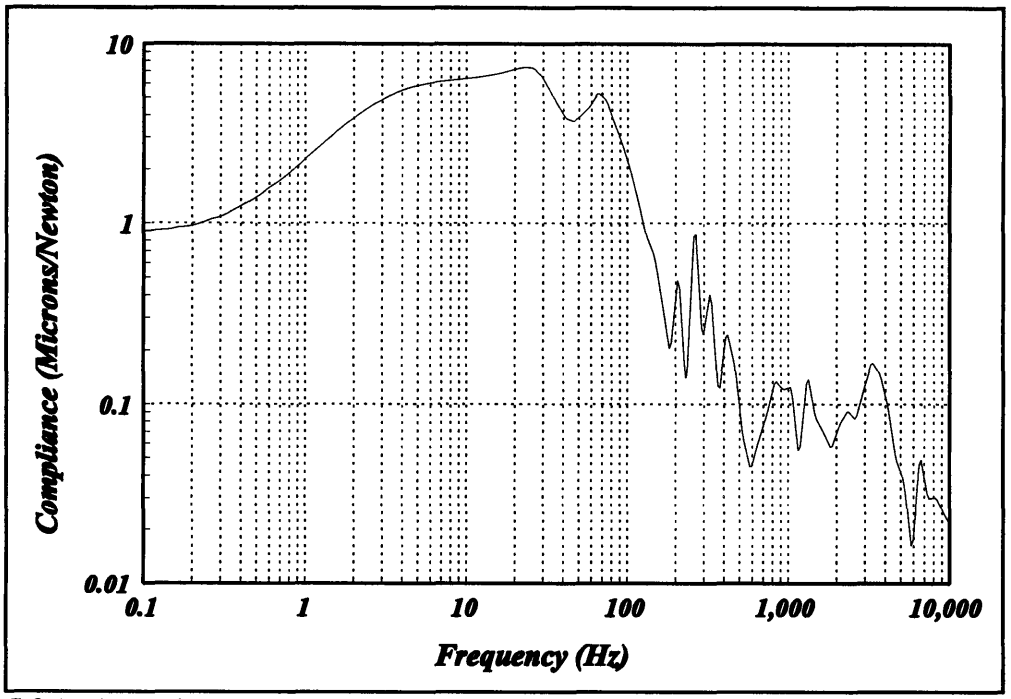
Only the magnitude plots will be presented in the remainder of this appendix as the phase plots convey little useful information. Of note are the spikes exhibited by both the analog and digital controllers in the 11 Hz region when the rotor was spun at 28000 RPM. These spikes are due to coupling between the upper and lower bearing. Under normal operation, the position signals of the upper and lower bearings are sinusoidal waves having different amplitudes and frequencies. However at the frequency of the spike, the position signals of both the upper and lower bearings had the same amplitude and the same frequency.

# G.2 Disturbance Rejection at 15000 RPM

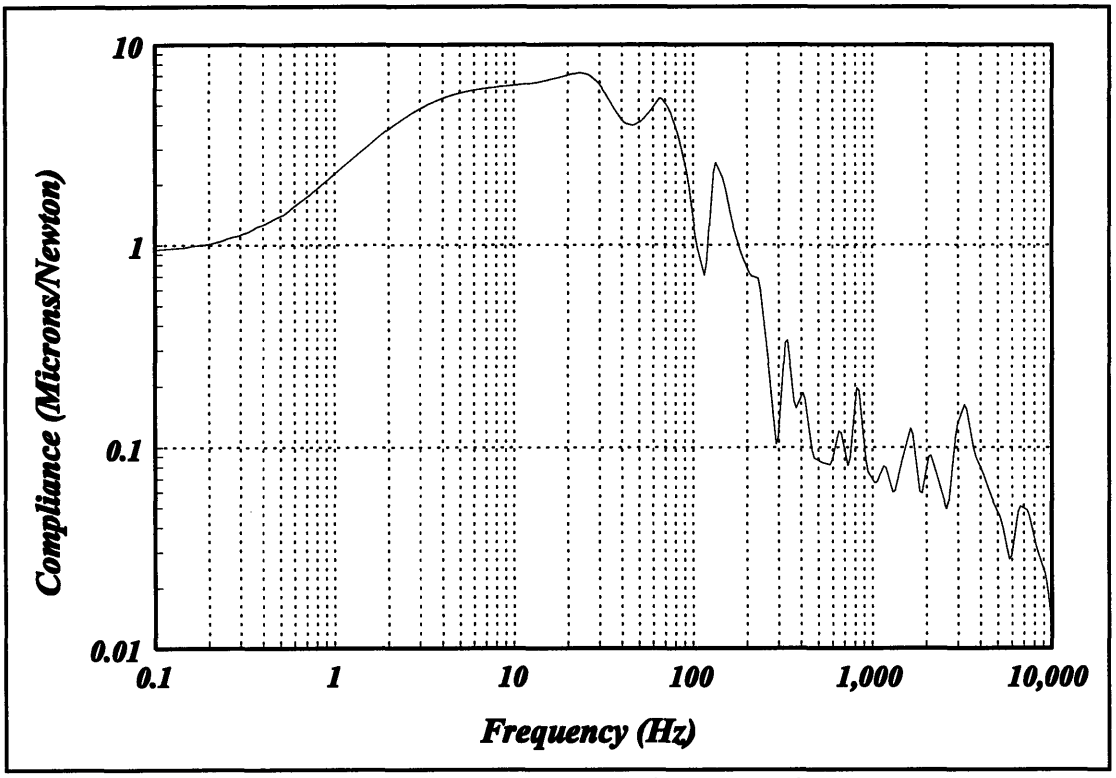
## G.2.1 Analog Controller



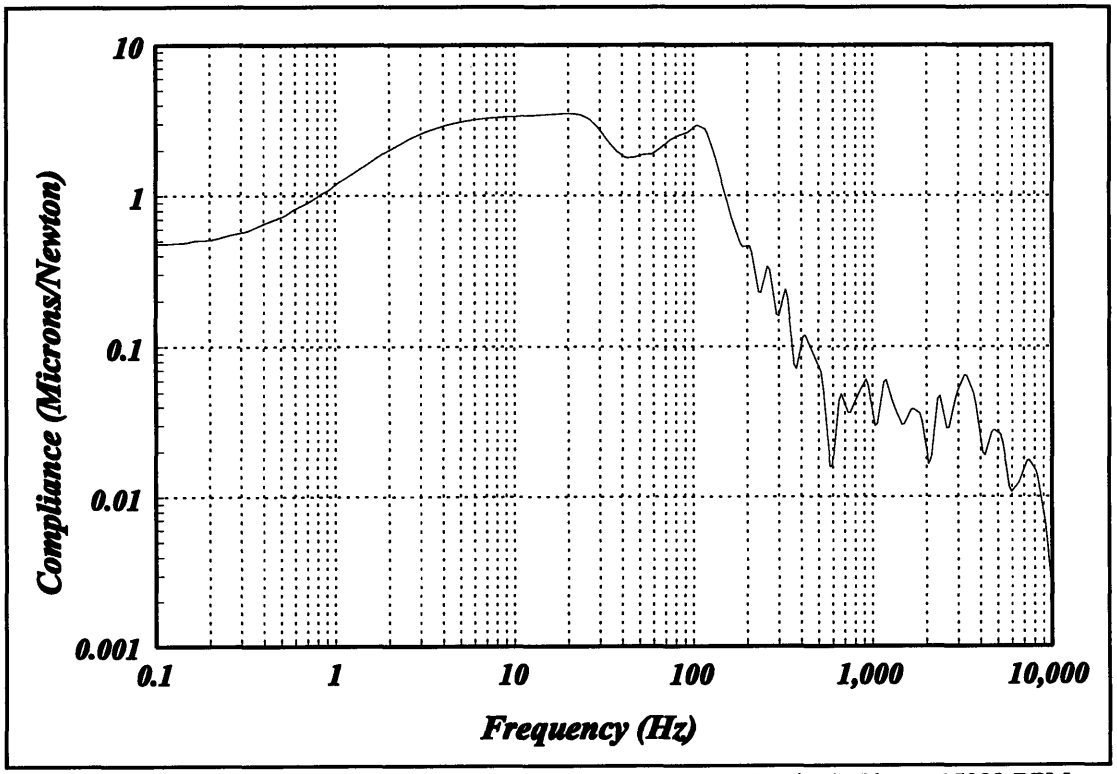

G-2 Axial Bearing Analog Controller Disturbance Rejection Magnitude Plot at 15000 RPM



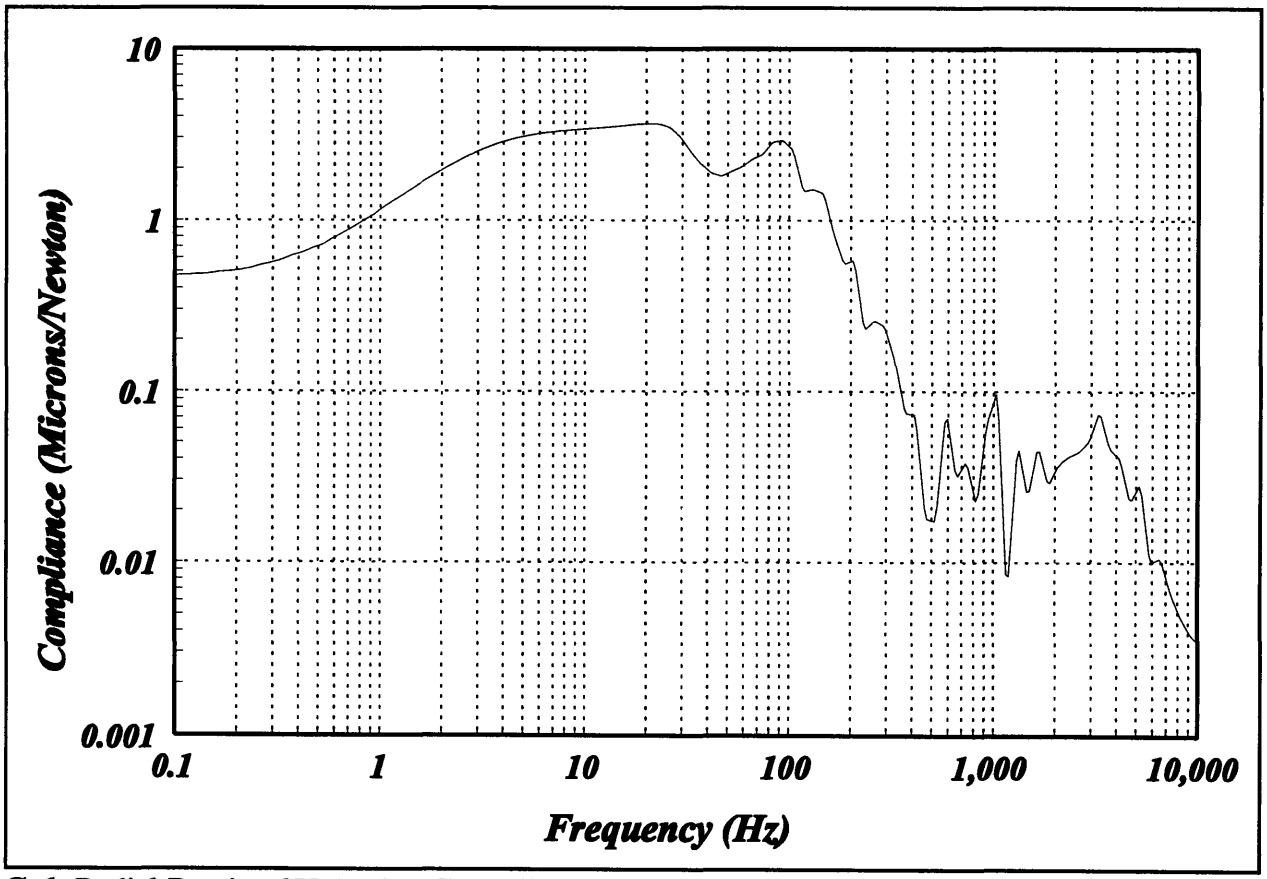

G-3 Radial Bearing 1X Analog Controller Disturbance Rejection Magnitude Plot at 15000 RPM




**G-4** Radial Bearing 1Y Analog Controller Disturbance Rejection Magnitude Plot at 15000 RPM

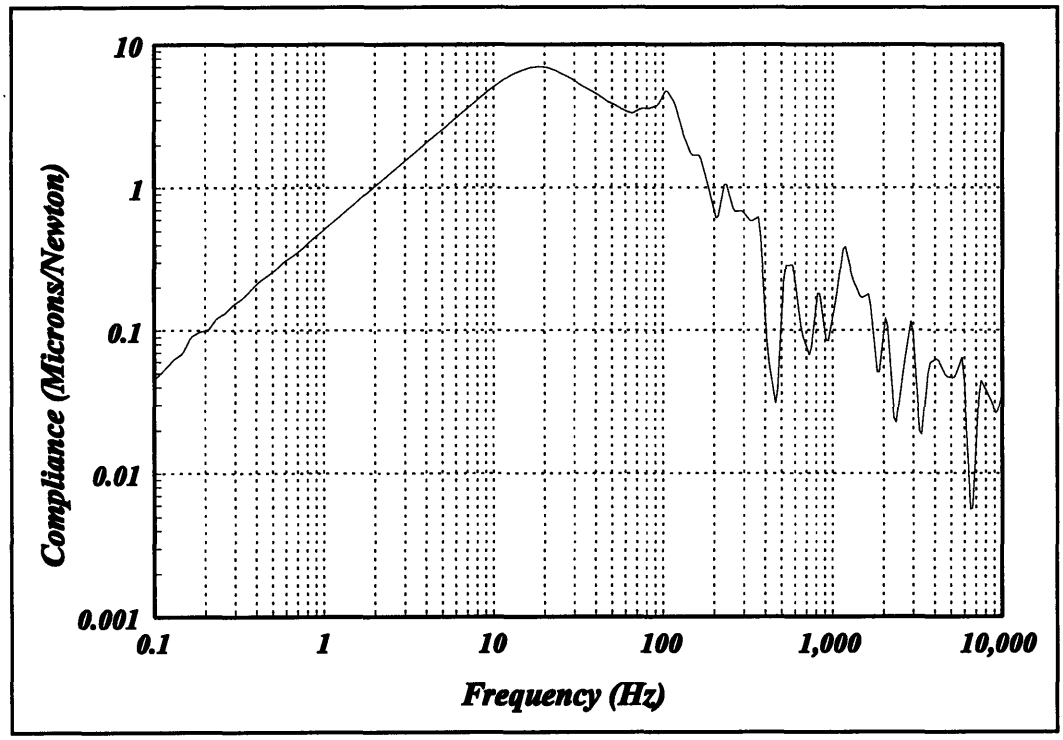



**G-5** Radial Bearing 2X Analog Controller Disturbance Rejection Magnitude Plot at 15000 RPM

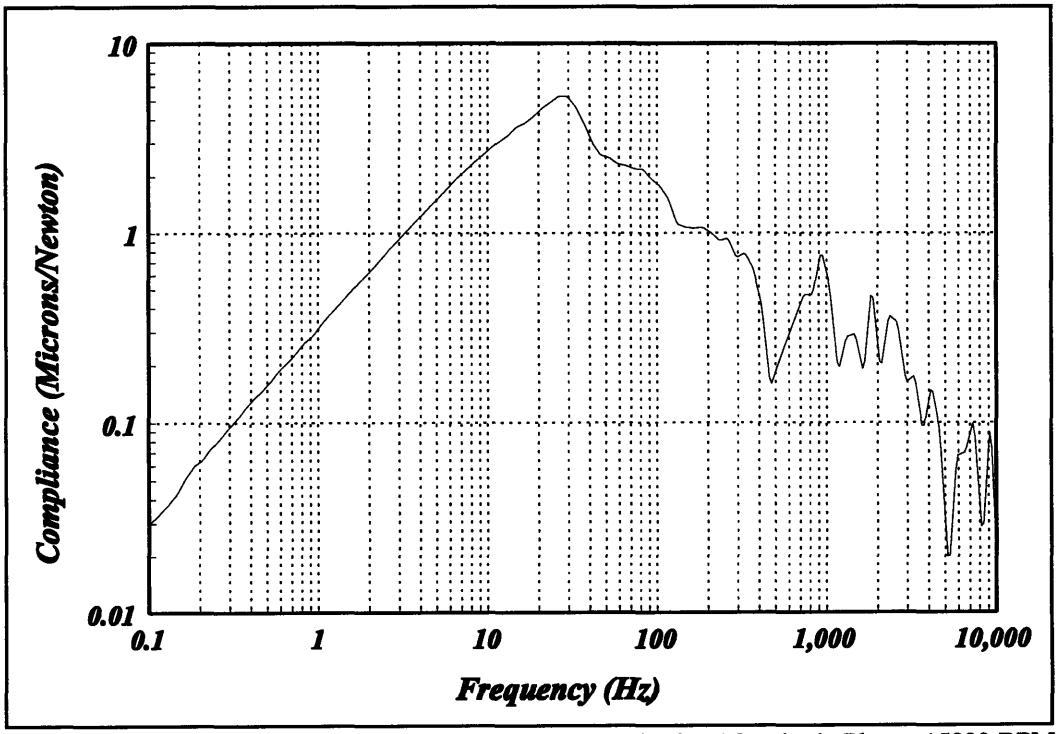



**G-6** Radial Bearing 2Y Analog Controller Disturbance Rejection Magnitude Plot at 15000 RPM

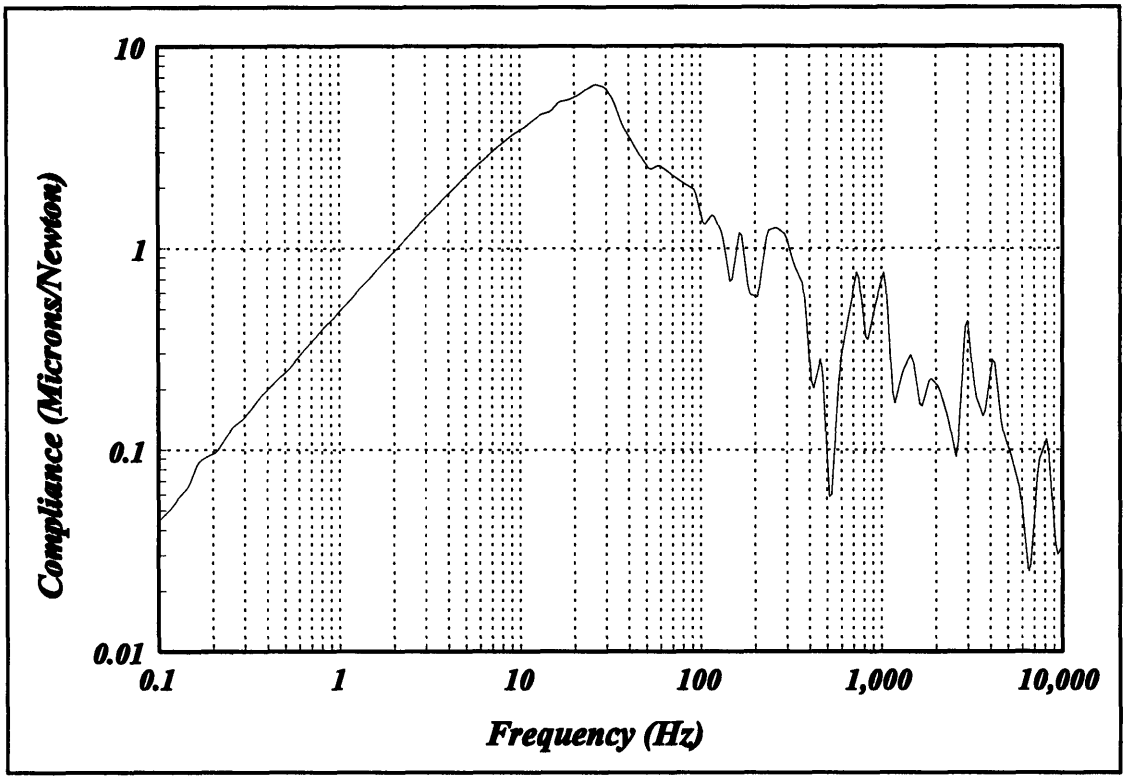
## G.2.2 Digital Controller



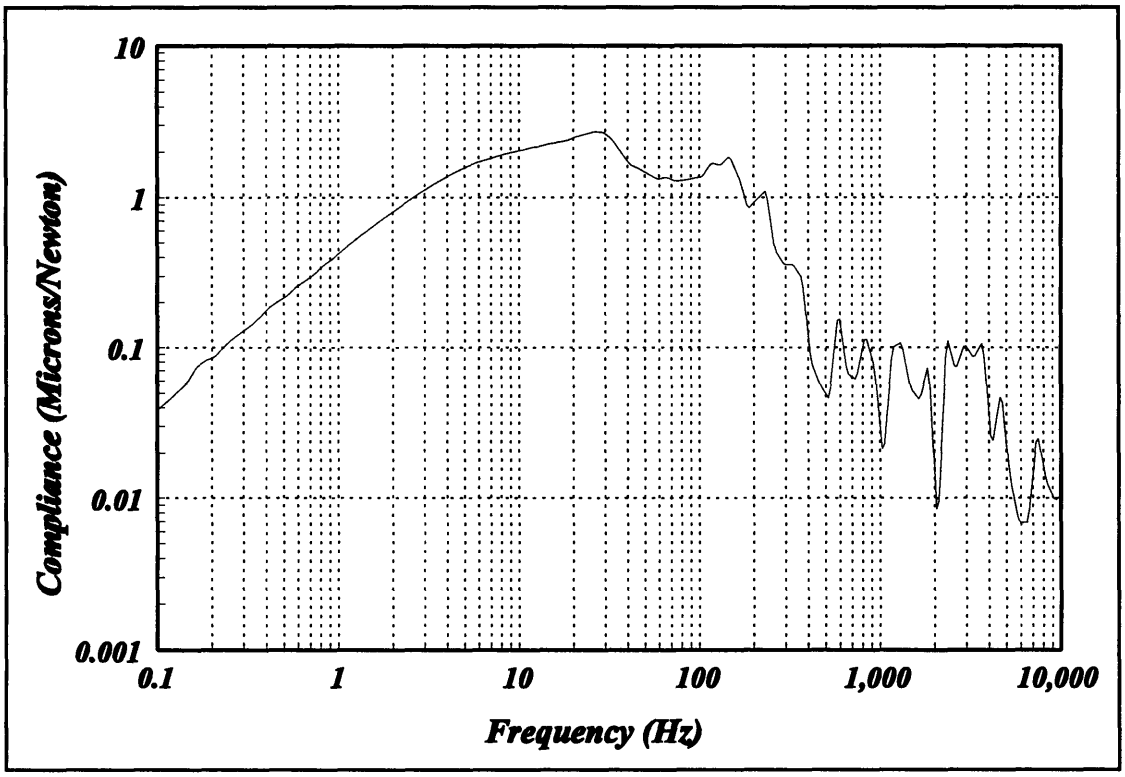

G-7 Axial Bearing Digital Controller Disturbance Rejection Magnitude Plot at 15000 RPM



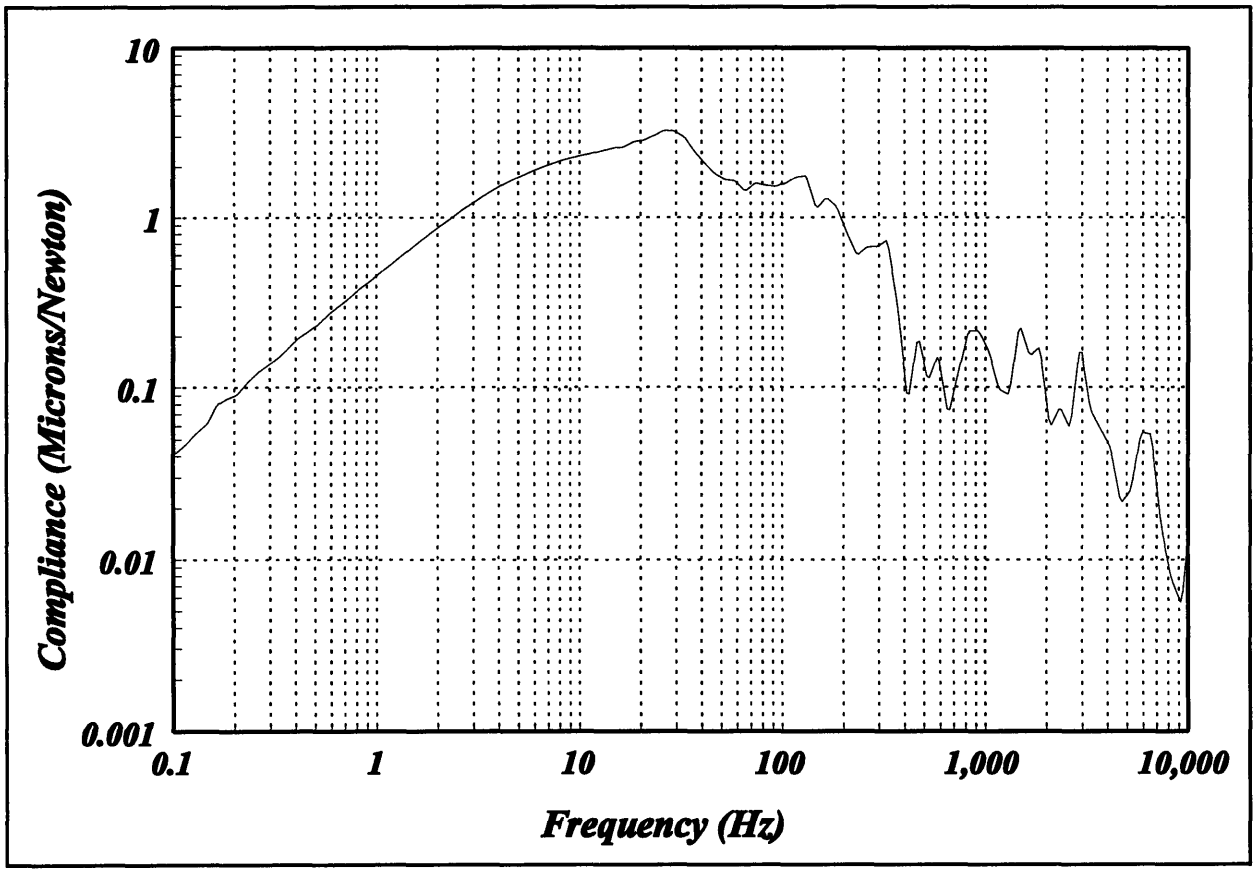

G-8 Radial Bearing 1X Digital Controller Disturbance Rejection Magnitude Plot at 15000 RPM




G-9 Radial Bearing 1Y Digital Controller Disturbance Rejection Magnitude Plot at 15000 RPM



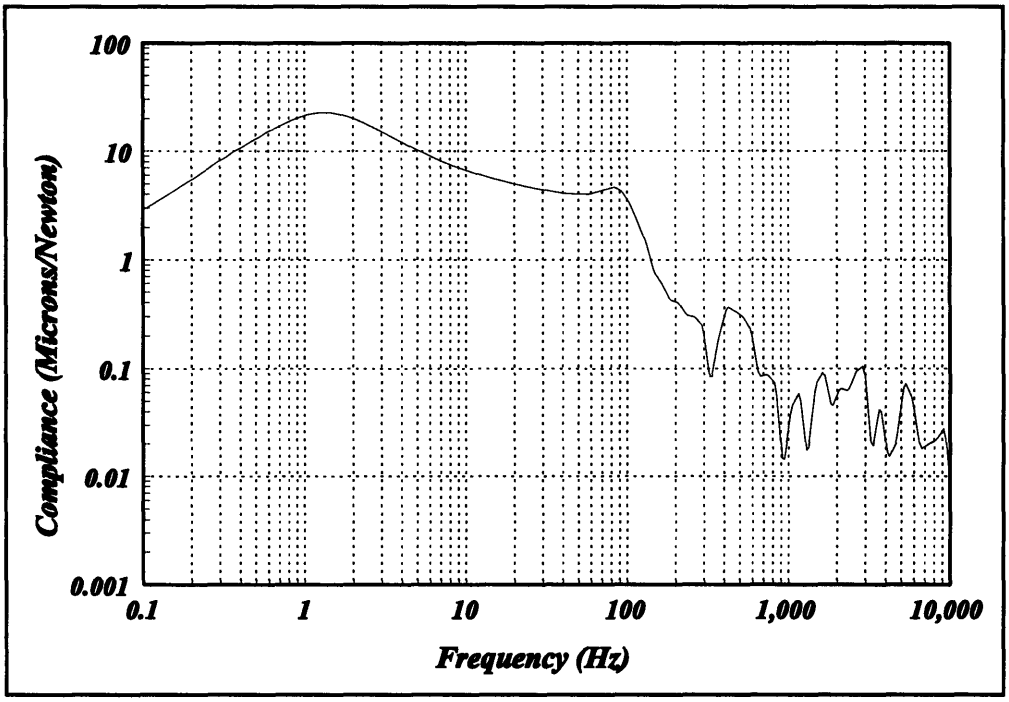

G-10 Radial Bearing 2X Digital Controller Disturbance Rejection Magnitude Plot at 15000 RPM



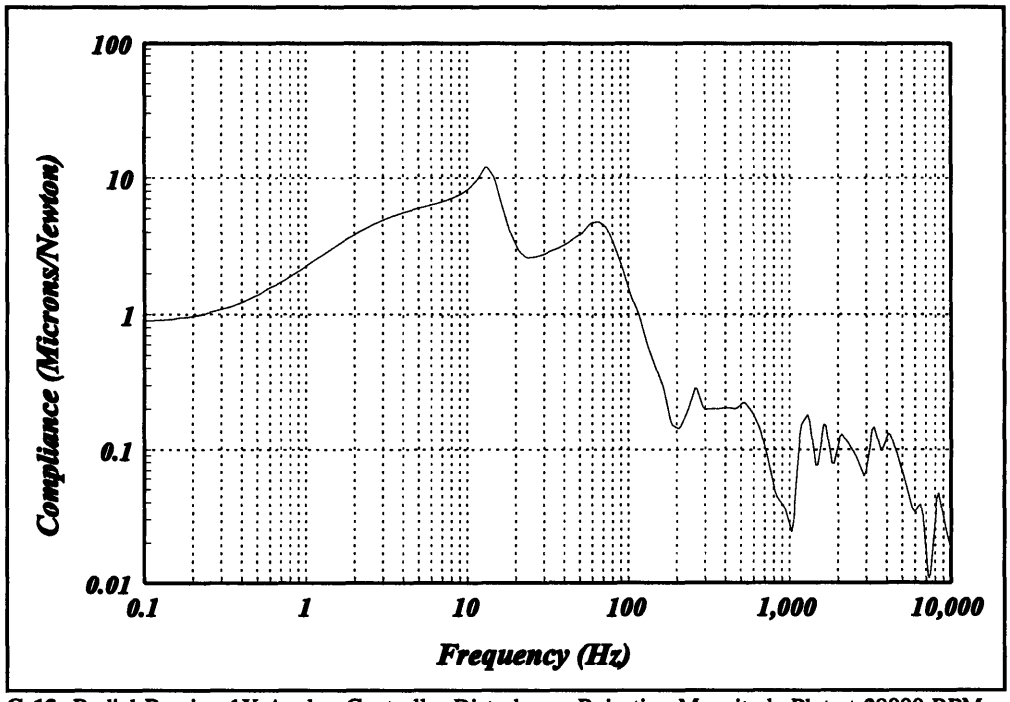

**G-11** Radial Bearing 2Y Digital Controller Disturbance Rejection Magnitude Plot at 15000 RPM

# G.3 Disturbance Rejection at 28000 RPM

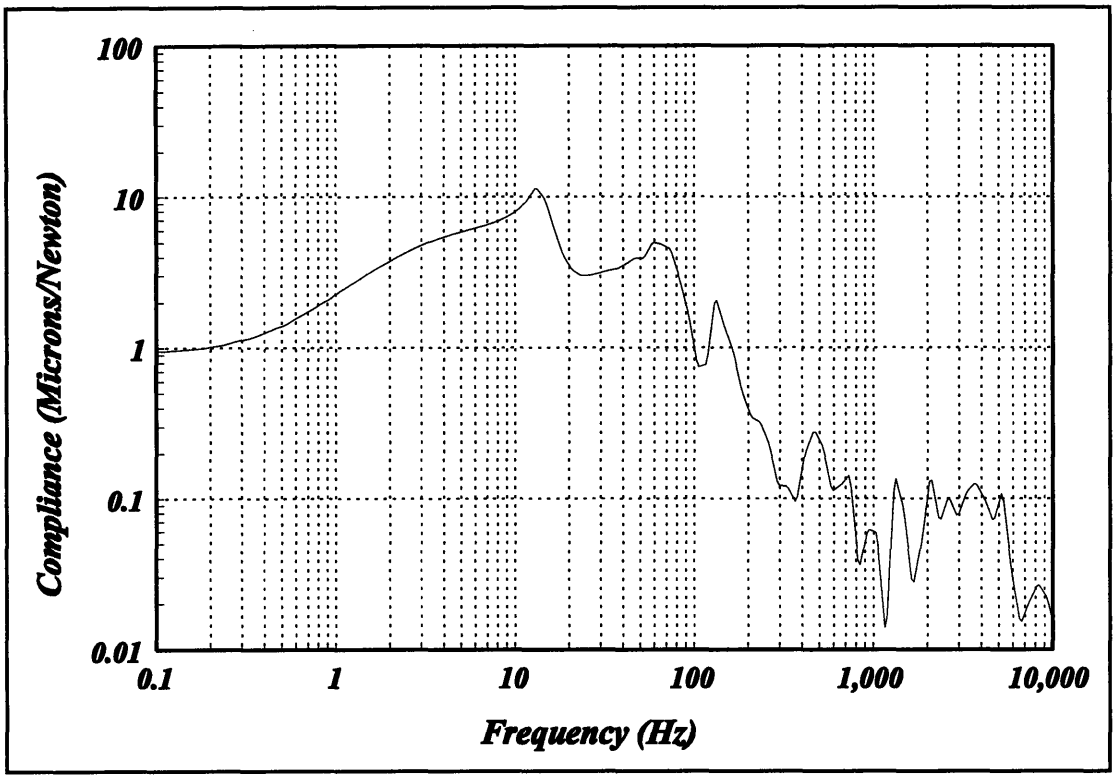
## G.3.1 Analog Controller



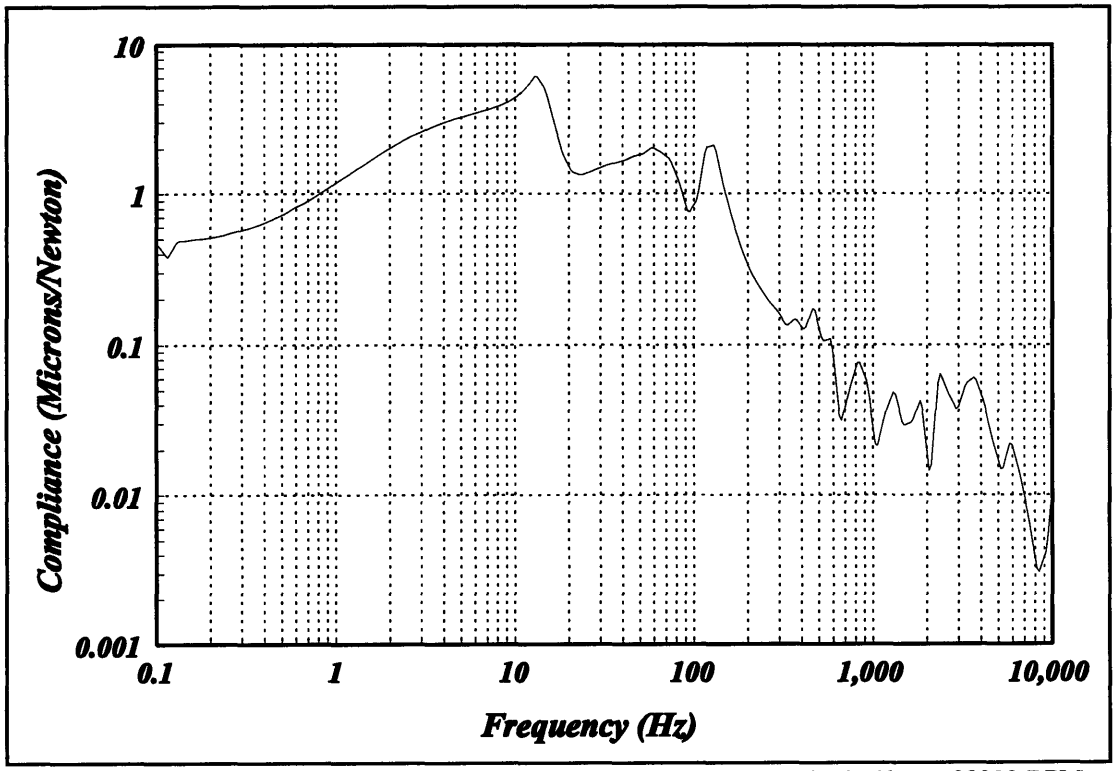

G-12 Axial Bearing Analog Controller Disturbance Rejection Magnitude Plot at 28000 RPM



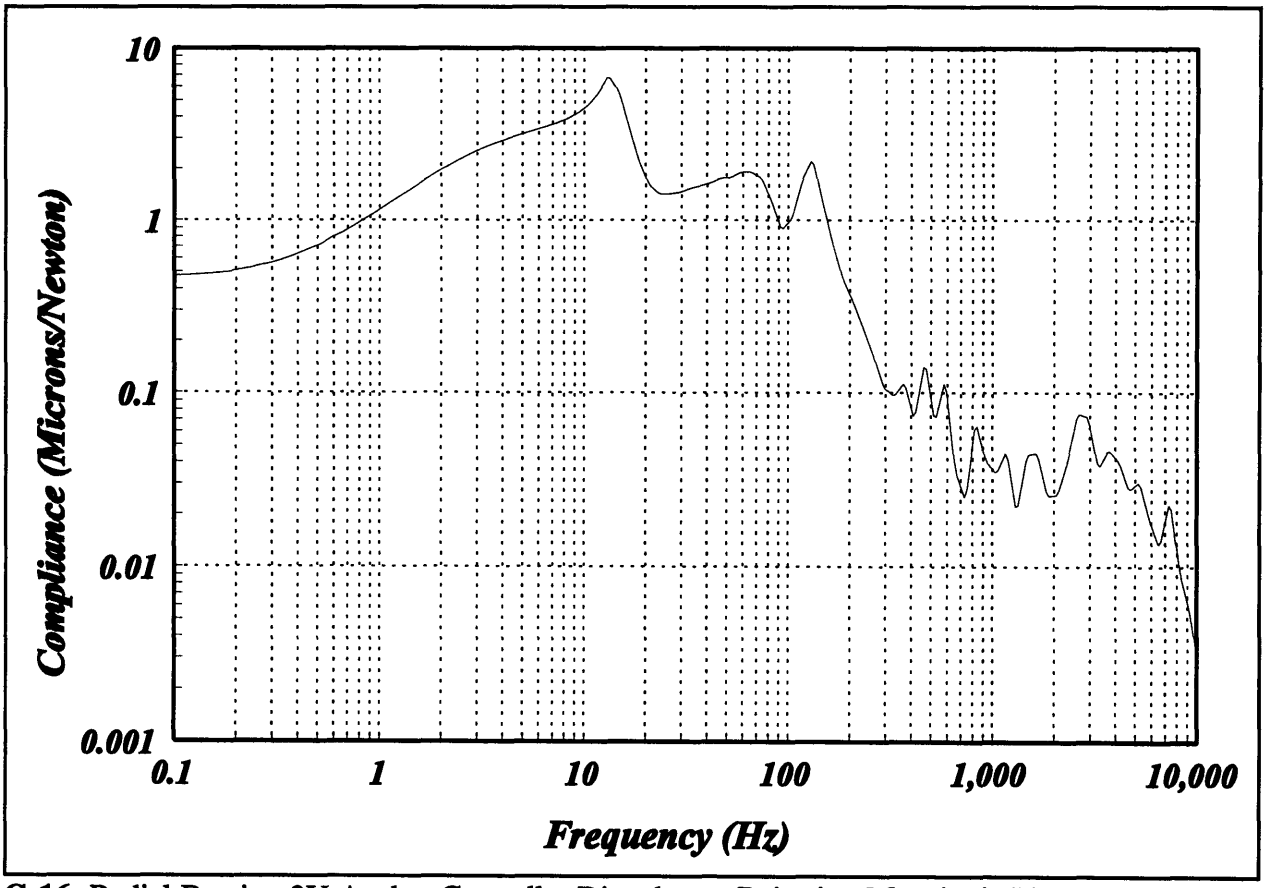

G-13 Radial Bearing 1X Analog Controller Disturbance Rejection Magnitude Plot at 28000 RPM




G-14 Radial Bearing 1Y Analog Controller Disturbance Rejection Magnitude Plot at 28000 RPM

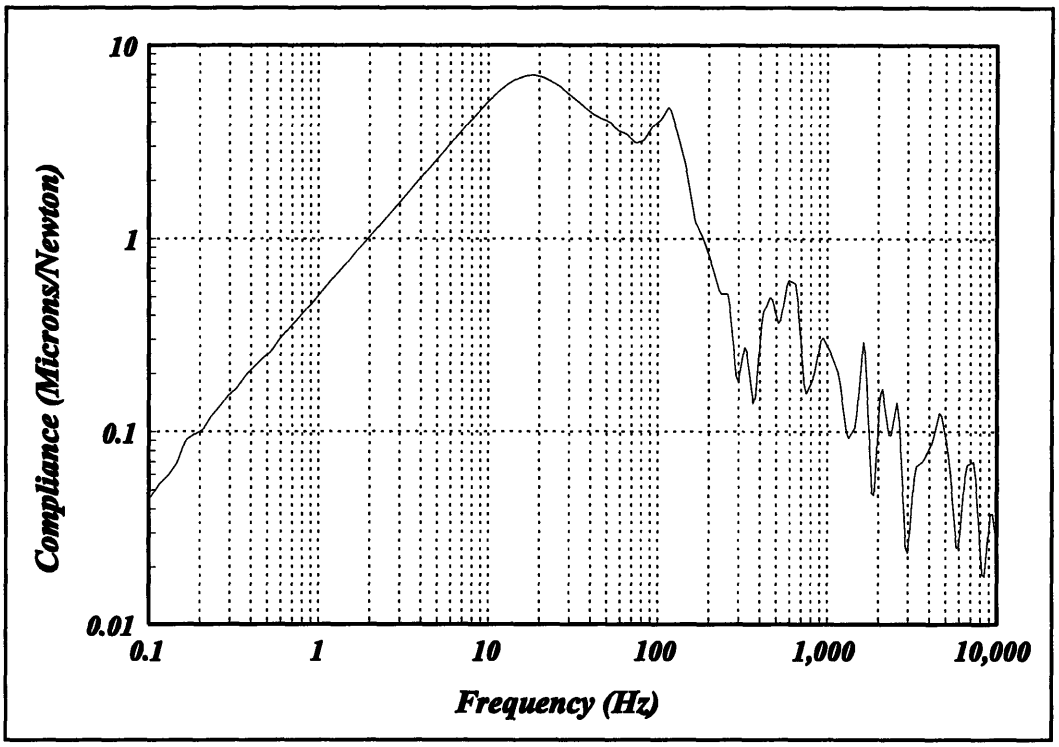



G-15 Radial Bearing 2X Analog Controller Disturbance Rejection Magnitude Plot at 28000 RPM

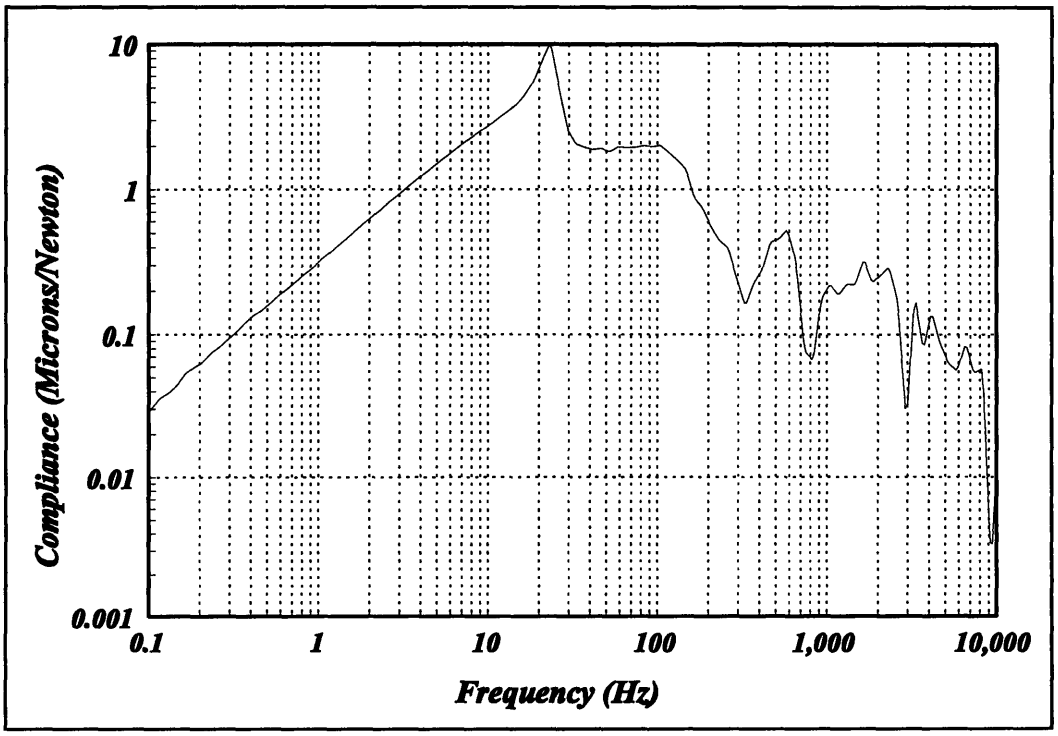



**G-16** Radial Bearing 2Y Analog Controller Disturbance Rejection Magnitude Plot at 28000 RPM

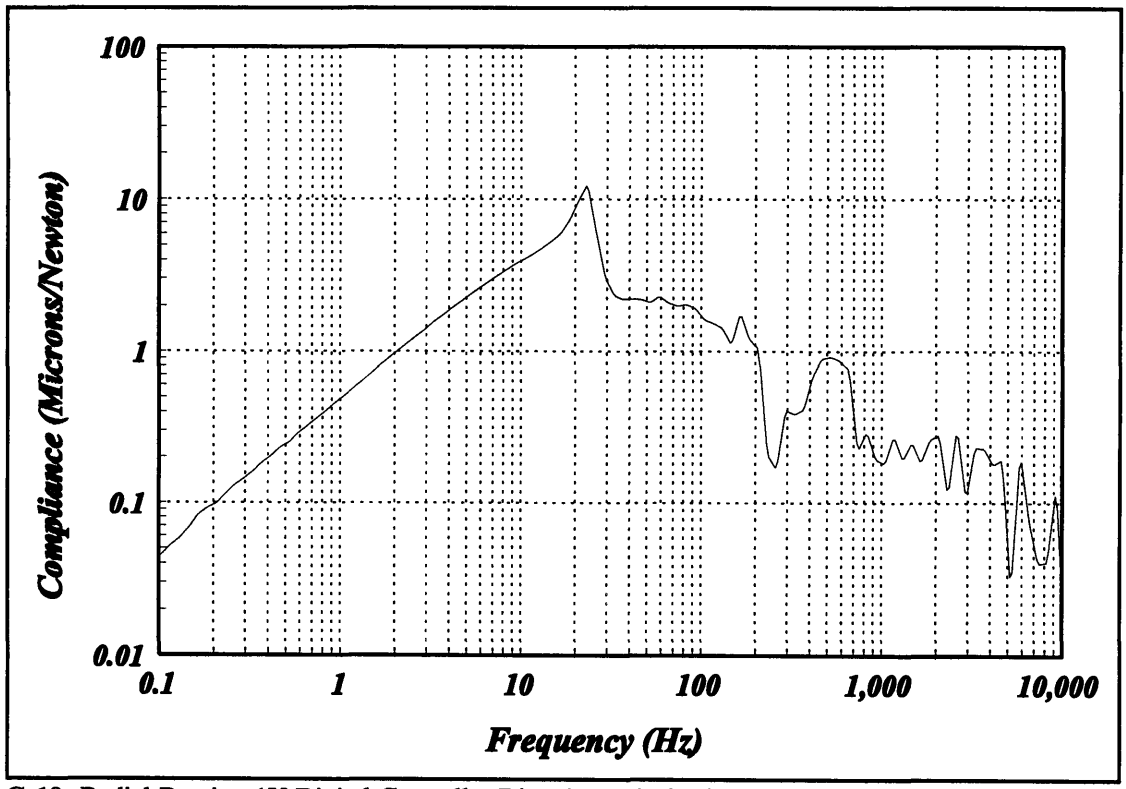
## G.3.2 Digital Controller



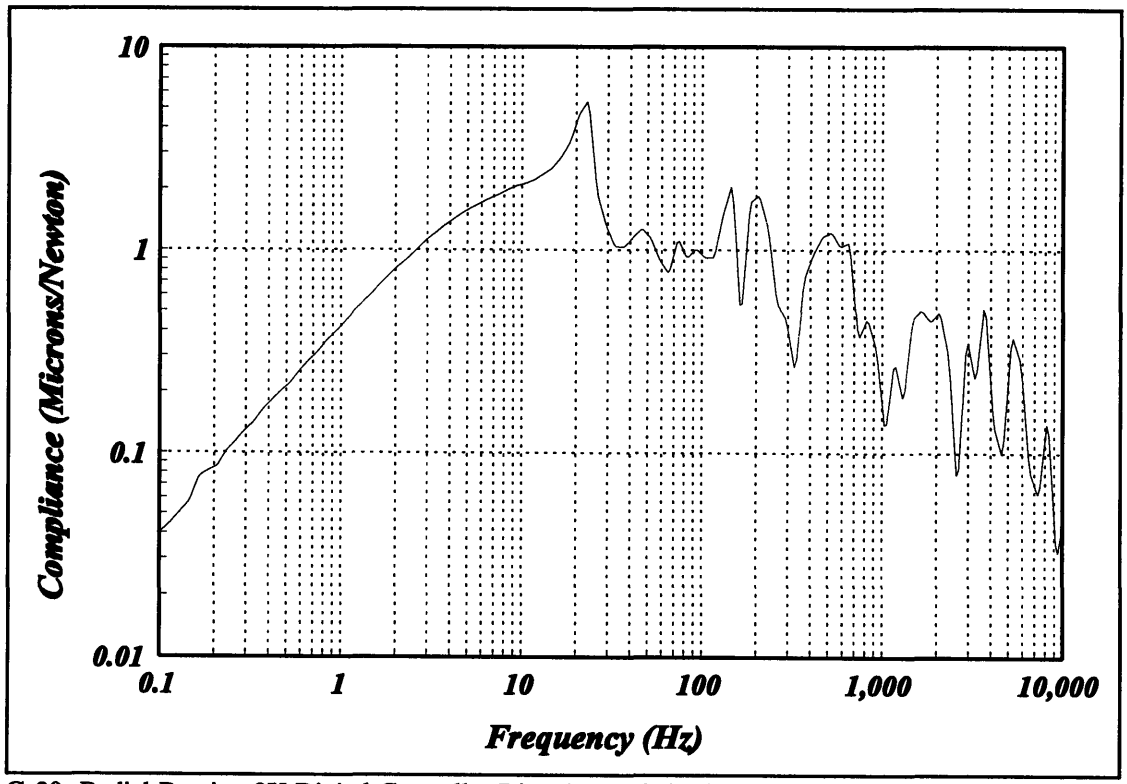

G-17 Axial Bearing Digital Controller Disturbance Rejection Magnitude Plot at 28000 RPM



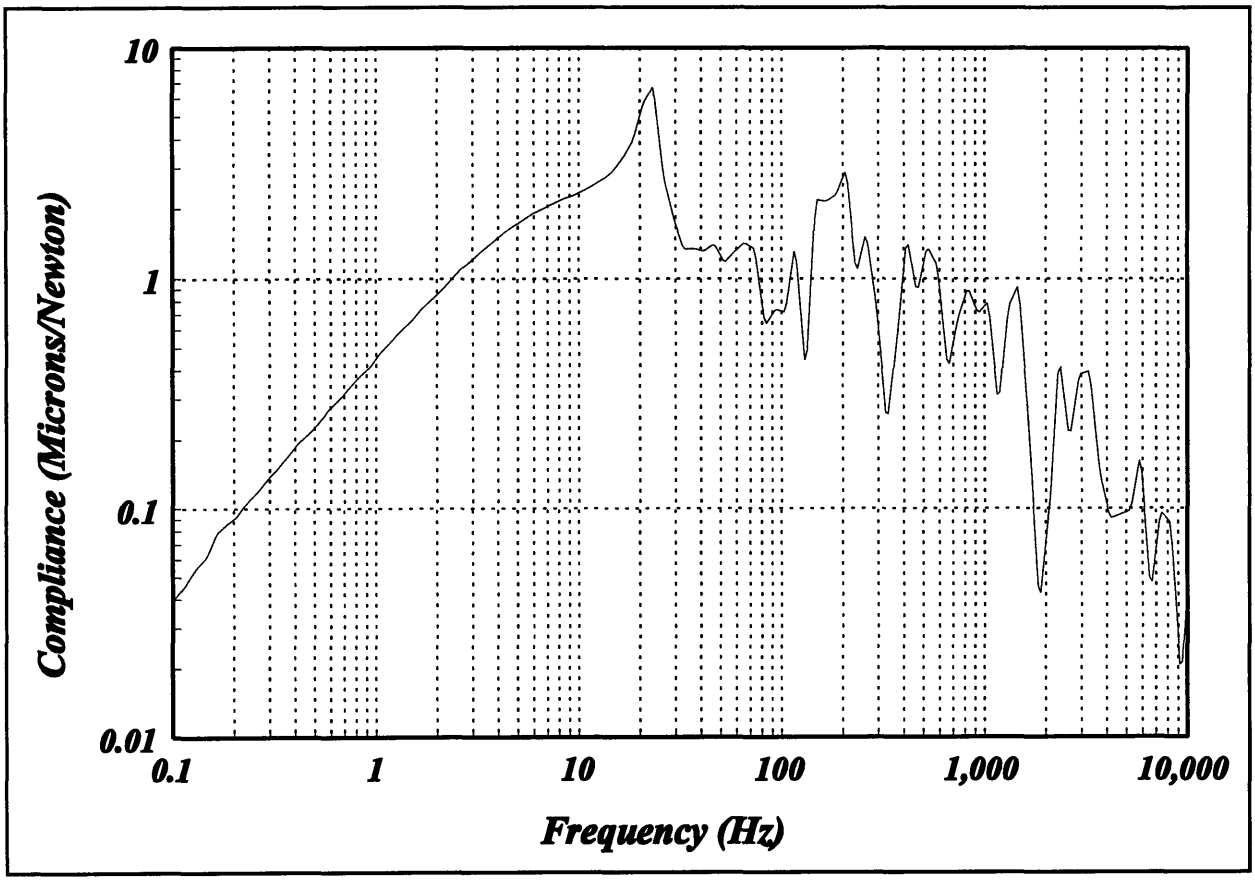

G-18 Radial Bearing 1X Digital Controller Disturbance Rejection Magnitude Plot at 28000 RPM




**G-19** Radial Bearing 1Y Digital Controller Disturbance Rejection Magnitude Plot at 28000 RPM

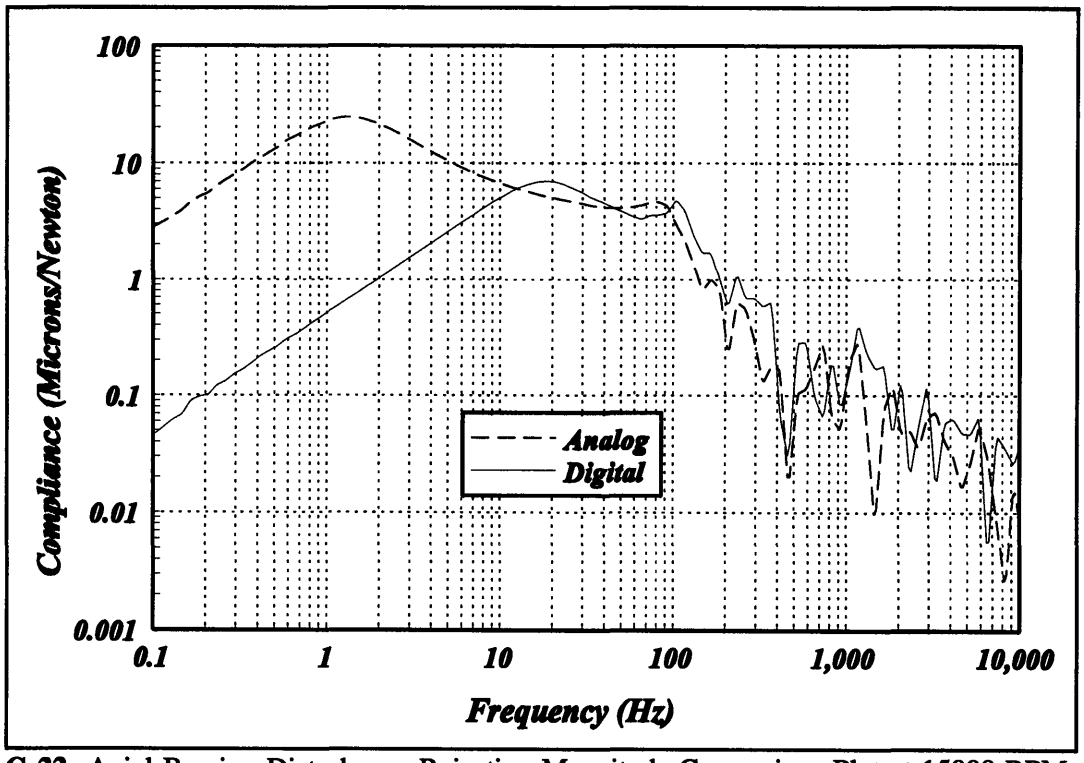



**G-20** Radial Bearing 2X Digital Controller Disturbance Rejection Magnitude Plot at 28000 RPM

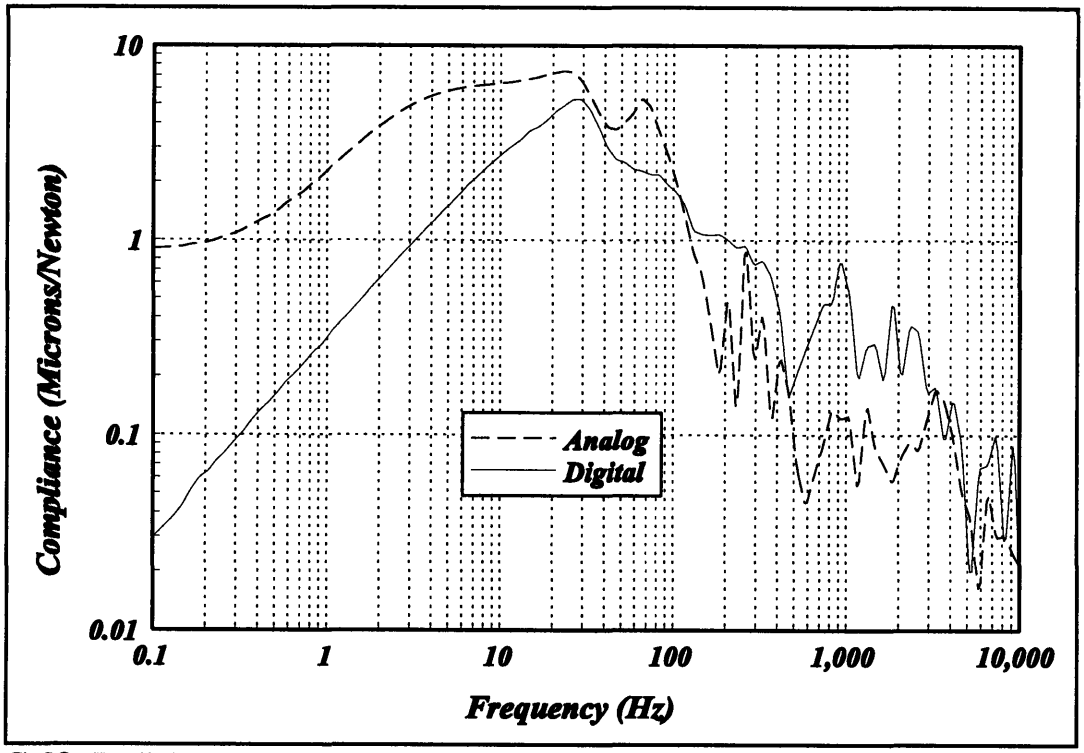



**G-21** Radial Bearing 2Y Digital Controller Disturbance Rejection Magnitude Plot at 28000 RPM

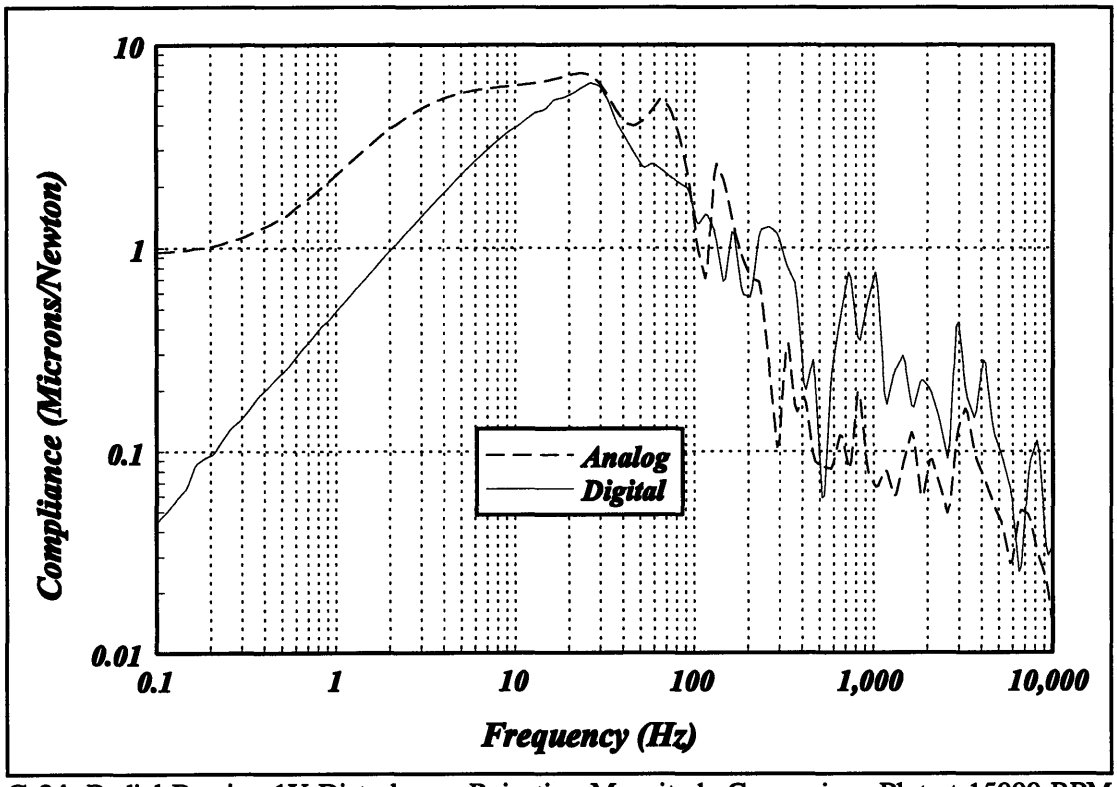
# G.4 Disturbance Rejection Comparison at 15000 RPM



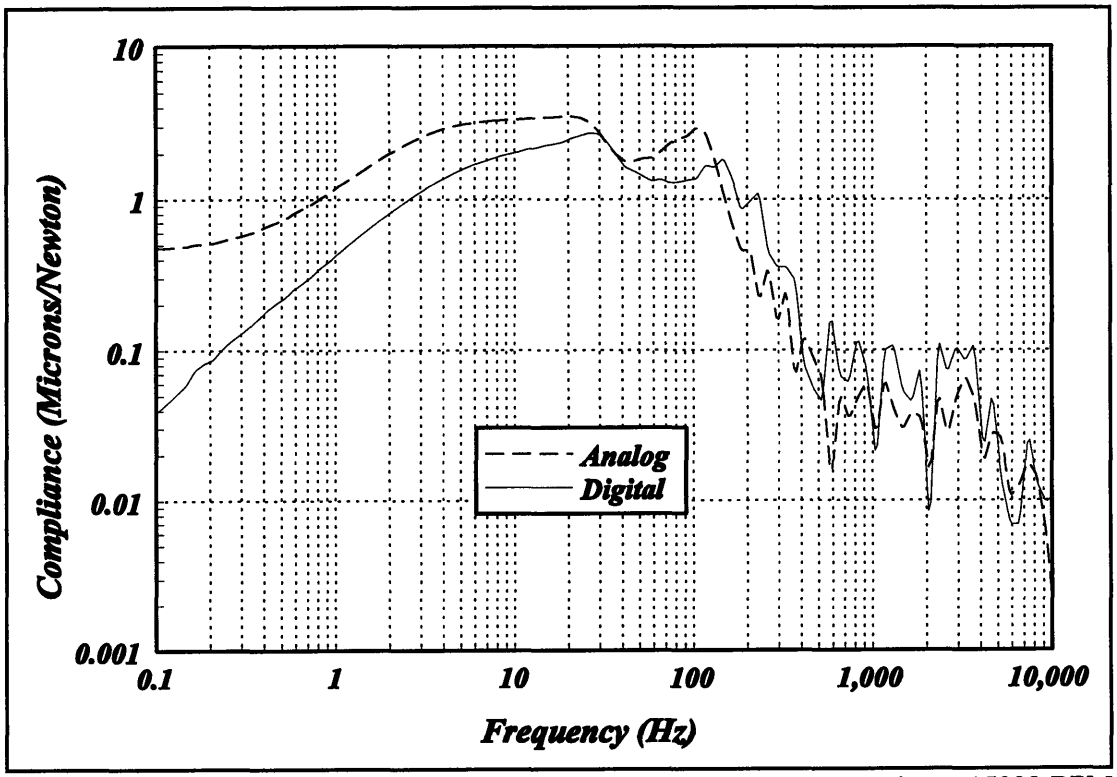

G-22 Axial Bearing Disturbance Rejection Magnitude Comparison Plot at 15000 RPM



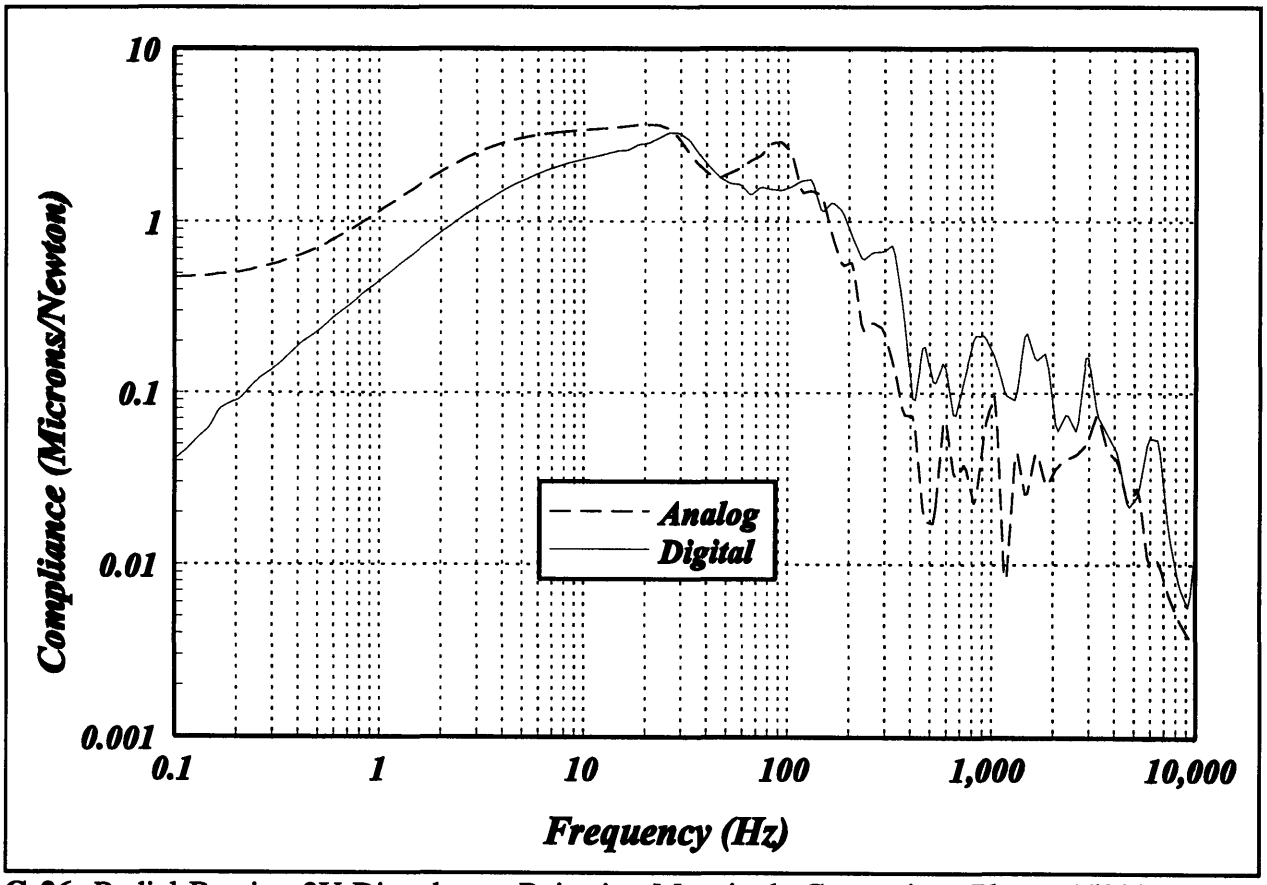

G-23 Radial Bearing 1X Disturbance Rejection Comparison Plot at 15000 RPM




**G-24** Radial Bearing 1Y Disturbance Rejection Magnitude Comparison Plot at 15000 RPM

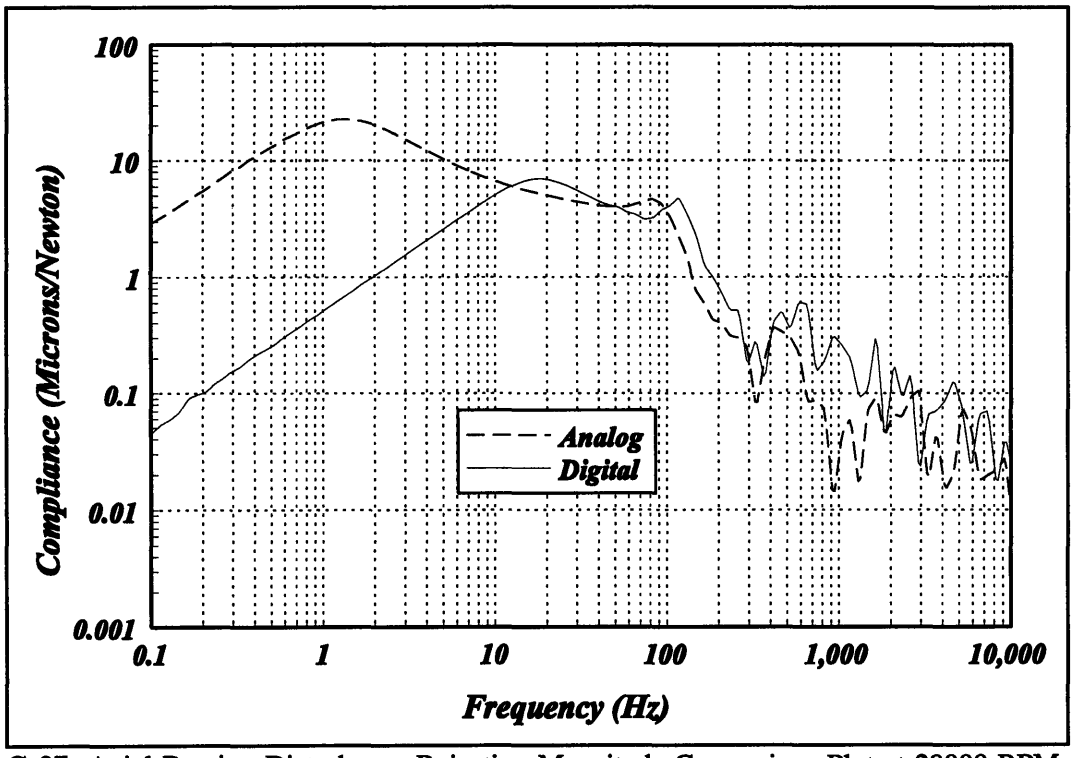



**G-25** Radial Bearing 2X Disturbance Rejection Magnitude Comparison Plot at 15000 RPM

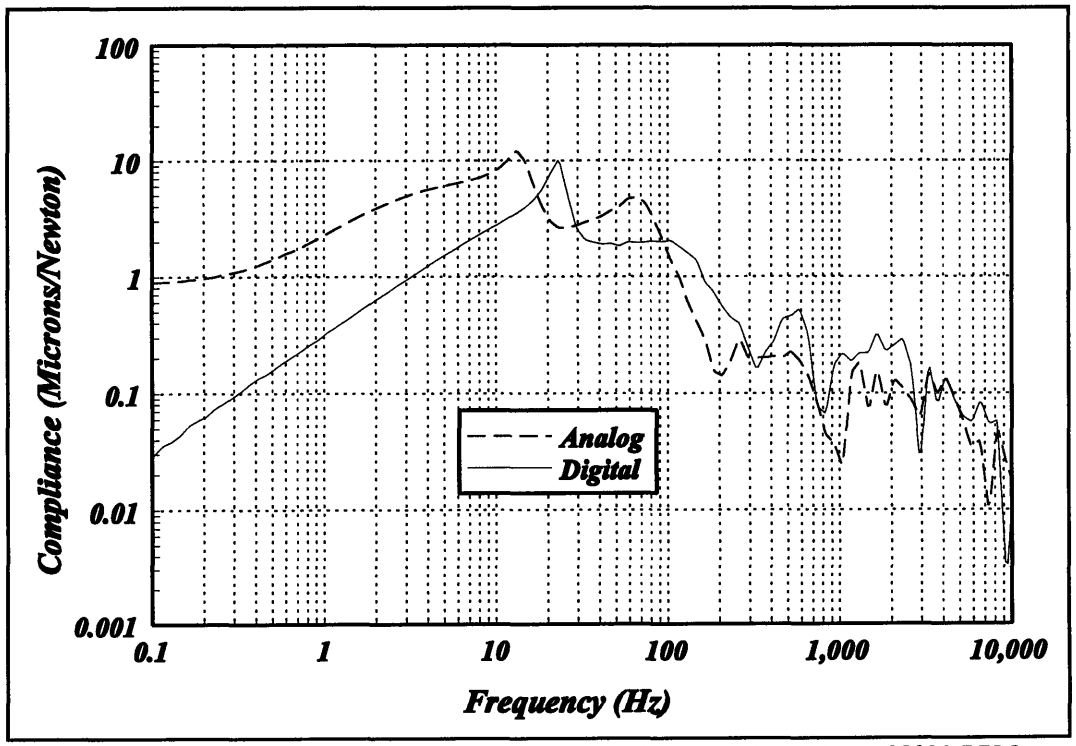



**G-26** Radial Bearing 2Y Disturbance Rejection Magnitude Comparison Plot at 15000 RPM

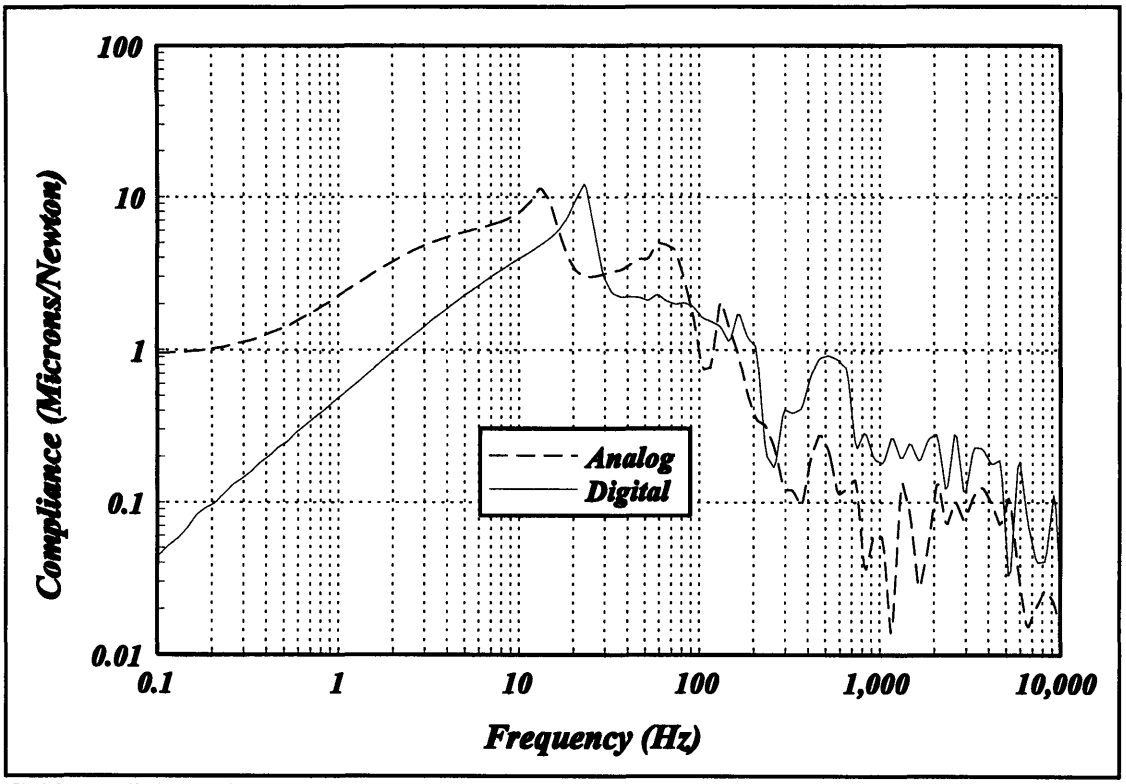
# G.5 Disturbance Rejection Comparison at 28000 RPM



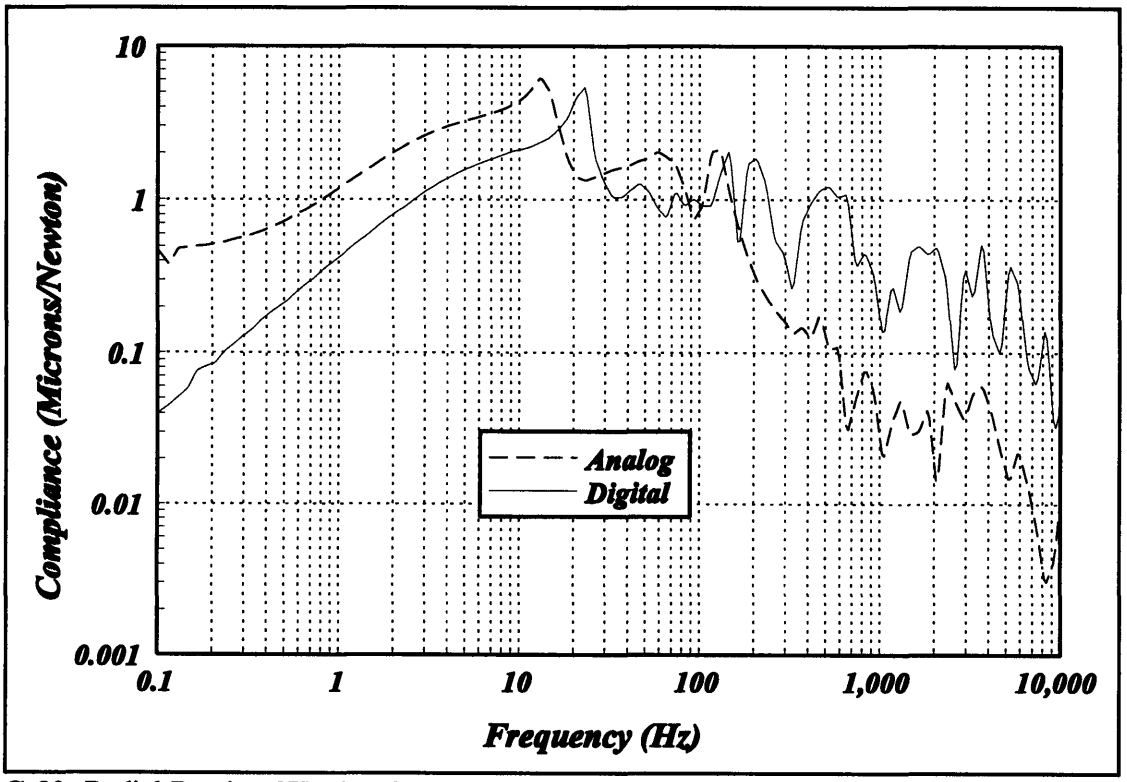

G-27 Axial Bearing Disturbance Rejection Magnitude Comparison Plot at 28000 RPM



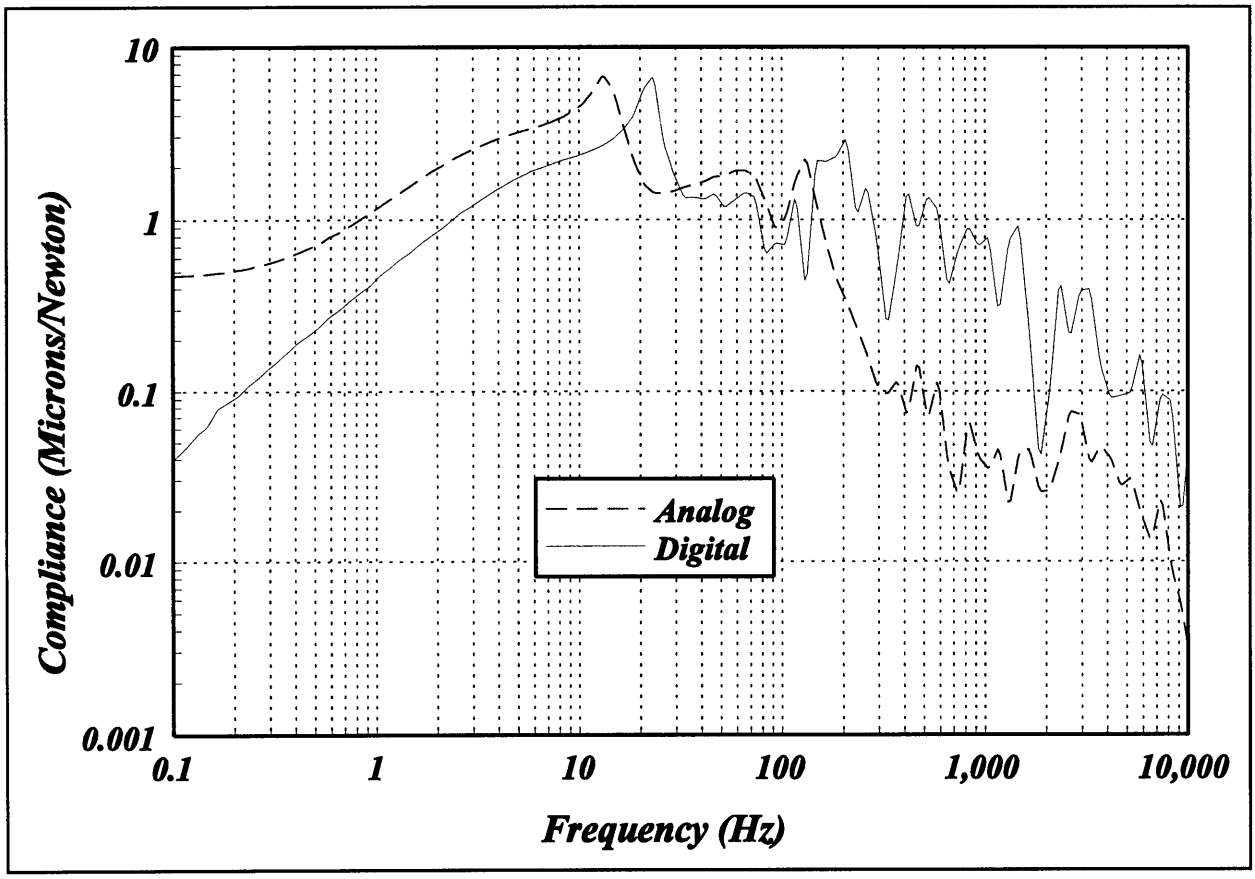

G-28 Radial Bearing 1X Disturbance Rejection Magnitude Comparison Plot at 28000 RPM




**G-29** Radial Bearing 1Y Disturbance Rejection Magnitude Comparison Plot at 28000 RPM




**G-30** Radial Bearing 2X Disturbance Rejection Magnitude Comparison Plot at 28000 RPM




**G-31** Radial Bearing 2Y Disturbance Rejection Magnitude Comparison Plot at 28000 RPM

# G.6 Disturbance Rejection Performance Values at 15000 RPM

| Parameter | Rad1X | | Rad1Y | |
|---|---|---|---|---|
| | Analog | Digital | Analog | Digital |
| Peak Compliance (Microns/Newton) | 7.37 | 5.29 | 7.22 | 6.43 |
| Compliance @ 1000Hz (Microns/Newton) | 0.123 | 0.615 | 0.069 | 0.689 |

| Parameter | Rad2X | | Rad2Y | |
|---|---|---|---|---|
| | Analog | Digital | Analog | Digital |
| Peak Compliance (Microns/Newton) | 3.51 | 2.72 | 3.63 | 3.26 |
| Compliance @ 1000Hz (Microns/Newton) | 0.037 | 0.030 | 0.086 | 0.179 |

| Parameter | Axial | |
|---|---|---|
| | Analog | Digital |
| Peak Compliance (Microns/Newton) | 24.45 | 7.00 |
| Compliance @ 1000Hz (Microns/Newton) | 0.121 | 0.121 |

# G.7 Disturbance Rejection Performance Values at 28000 RPM

| Parameter | Rad1X | | Rad1Y | |
|---|---|---|---|---|
| | Analog | Digital | Analog | Digital |
| Peak Compliance (Microns/Newton) | 12.02 | 9.95 | 11.34 | 12.18 |
| Compliance @ 1000Hz (Microns/Newton) | 0.027 | 0.203 | 0.060 | 0.184 |

| Parameter | Rad2X | | Rad2Y | |
|---|---|---|---|---|
| | Analog | Digital | Analog | Digital |
| Peak Compliance (Microns/Newton) | 6.13 | 5.36 | 6.71 | 6.67 |
| Compliance @ 1000Hz (Microns/Newton) | 0.028 | 0.0175 | 0.036 | 0.759 |

| Parameter | Axial | |
|---|---|---|
| | Analog | Digital |
| Peak Compliance (Microns/Newton) | 22.73 | 6.94 |
| Compliance @ 1000Hz (Microns/Newton) | 0.027 | 0.277 |

# Appendix H

# Assorted Program Listings

---

The appendix provides the listings of the programs used to determine the values of the controller parameters and to model the theoretical system. Where speed was essential and graphical presentation of the data was unnecessary, the programs were written in C. Otherwise the programs were written as Matlab scripts. This appendix is comprised of four sections. The first section lists programs used to determine the optimal sampling rate. This section also lists programs that provided important system response values such as maximum compliance, controller bandwidth, and maximum closed loop gain over a range of values for damping ratio, natural frequency, and feedback gain. The second section lists programs used to display the actual system responses obtained from the system analyzer and the theoretical responses predicted by the model. The third section lists programs that return different aspects of the components of the theoretical model. The last section lists certain miscellaneous programs which were used in one form or another during the course of designing or analyzing the controller and its response.

## H.1 Controller Parameter Determination Programs

This section lists programs used to determine the optimal sampling rate. Also listed here are the programs that provided the large file of system response values such as bandwidth, maximum compliance, and maximum closed loop gain given the controller design parameters of damping ratio, natural frequency, and feedback gain. These programs were run in a specific order with one program building on the results of the previous ones. The listings below are presented in the order in which they were initially run.

### H.1.1 limits.c

The program recursively computes the eigenvalues of the closed loop system for a range of values of sampling rate, damping ratio, natural frequency, and feedback gain. From the eigenvalue data, the stability of the system is determined and the results are saved in a binary character array in an effort to conserve disk space.

```
#include <stdlib.h>
```

```
#include <stdio.h>
#include <string.h>
#include <math.h>

#ifdef NMALLOC
#include "nmalloc.h"
#endif

#define RADIX 2.0
#define NR_END 1
#define MAXM 8

#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))
#define MYERR       1e-8
#define OVERSHOOT   0.1
#define MAXSETTLE   0.6
#define BHAT_SIZE   100
#define DAMP_SIZE    19
#define FREQ_SIZE   100
#define SAMP_SIZE    50

void DigStabFunc(double A1, double A2, double P, double Q, double newt,
                 double a22, double a21, double *J1, double *J2, double *J3,
                 double *J4, double *J5, double *M1, double *M2, double *M3,
                 double *M4, double *N1, double *N2, double *N3, double *N4);
int DigReference(double bfreq, double efreq, double dfreq, double bT,
                 double eT, double dT, double initx, double initu, double maxt,
                 double **settle, int *n);
int DigStep(int bearing, double b, double damp, double freq, double T,
            double initx, double initu, double maxt, double P, double Q,
            double A1, double A2, double **y, double **u, double **t, int *n);
int zrhqr(double a[], int m, double rtr[], double rti[]);
void balanc(double **a, int n);
int hqr(double **a, int n, double wr[], double wi[]);
double **matrix(long nrl, long nrh, long ncl, long nch);
void free_matrix(double **m, long nrl, long nrh, long ncl, long nch);

void main(int argc, char **argv)
{
  int i, j, k, l, m, begin;
  int curcols, currows, ebara, axis;
  int ibeg, jbeg;
  char buffer[2000], *filename;
  FILE *fp;
  double P, Q, A1, A2;
  double bhat, freq, damp, newt;
  double a21, a22;
#ifdef STEP
  int setsize;
  double *settle;
  double *y, *u, *t, initx, initu, over, set;
#endif
  double subtot[9], rroots[9], iroots[9];
  double J1, J2, J3, J4, J5, M1, M2, M3, M4, N1, N2, N3, N4;

  ebara = 0;
  begin = axis = 0;

  if(argc < 2 || argc > 6)
    {
      fprintf(stderr, "Syntax error\n");
      fprintf(stderr, "Syntax:  limits filename [-a n] [-b m]\n");
      fprintf(stderr, "         where:  filename - binary array filename\n");
      fprintf(stderr, "                 -b n -> start integer [0-10000]\n");
      fprintf(stderr, "                 -a m -> bearing axis [0-4]\n");
      exit(1);
    }

  for(i=1; i<argc; i++)
```

```
  {
    if(i == 1)
      filename = argv[1];
    else if(argv[i][0] == '-')
      {
        if(argv[i][1] == 'b')
          begin = atoi(argv[i+1]);
        else if(argv[i][1] == 'a')
          axis = atoi(argv[i+1]);
      }
  }

curcols = (SAMP_SIZE * DAMP_SIZE);
currows = 0;

if(begin)
  {
    if((fp = fopen(filename, "r+")) == NULL)
      {
        fprintf(stderr, "Unable to open file \"%s\" (r+)\n", filename);
        exit(1);
      }
    currows = begin;
  }
else
  {
    if((fp = fopen(filename, "w+")) == NULL)
      {
        fprintf(stderr, "Unable to open file \"%s\" (w+)\n", filename);
        exit(1);
      }
  }

fseek(fp, 0L, SEEK_SET);

if(!fwrite((void *) &currows, sizeof(int), 1, fp))
  {
    fprintf(stderr, "row data write failed\n");
    fclose(fp);
    exit(1);
  }
if(!fwrite((void *) &curcols, sizeof(int), 1, fp))
  {
    fprintf(stderr, "columns data write failed\n");
    fclose(fp);
    exit(1);
  }

if(ebara)
  {
    switch(axis)
      {
      default:
      case 0:
        P = 11.081;
        Q = 27701.879;
        A1 = 1666.622;
        A2 = 1700.000;
        break;
      case 1:
        P = 10.278;
        Q = 41113.653;
        A1 = 14222.428;
        A2 = 4370.000;
        break;
      case 2:
        P = 10.278;
        Q = 41113.653;
        A1 = 14575.775;
```

```
          A2 = 4030.000;
          break;
        case 3:
          P = 20.263;
          Q = 81050.754;
          A1 = 4027.457;
          A2 = 2450.000;
          break;
        case 4:
          P = 20.263;
          Q = 81050.754;
          A1 = 4108.477;
          A2 = 2600.000;
          break;
        }
    }
  else
    {
      switch(axis)
        {
        default:
        case 0:
          P = 7.990;
          Q = 22739.569;
          A1 = 11097.006;
          A2 = 13310.0;
          break;
        case 1:
          P = 7.123;
          Q = 7737.770;
          A1 = 4619.584;
          A2 = 13130.0;
          break;
        case 2:
          P = 8.296;
          Q = 8882.644;
          A1 = 4558.848;
          A2 = 13080.0;
          break;
        case 3:
          P = 16.926;
          Q = 35530.576;
          A1 = 3863.909;
          A2 = 11140.0;
          break;
        case 4:
          P = 15.113;
          Q = 33201.349;
          A1 = 4212.3;
          A2 = 12290.0;
          break;
        }
    }

#ifdef STEP
  switch(axis)
    {
    default:
    case 0:
      initx = -0.0002;
      initu = 1.167;
      break;
    case 1:
    case 2:
    case 3:
    case 4:
      initx = 0.0001;
      initu = 0.0;
      break;
```

```
    }
#endif

  /*
   * newt = [1000:1000:50000];
   * damp = [[0.1:0.1:0.9] [1:1:10]];
   * freq = [10:10:1000];
   * bhat = [10:10:1000];
   */

  subtot[8] = 1.0;

#ifdef STEP
  /* get maximum settling times for given frequencies */
  if(DigReference(10.0, 1000.0, 10.0, 1000.0, 50000.0, 1000.0,
                  0.0, 1.0, MAXSETTLE, &settle, &setsize))
    {
      fprintf(stderr, "Error encountered in DigReference - limits.\n");
      exit(1);
    }
#endif

  if(begin)
    {
      fseek(fp, (currows * curcols * sizeof(char)) + (2 * sizeof(int)),
            SEEK_SET);
      ibeg = (int) (currows / 100);
      jbeg = (int) (currows % 100);
    }
  else
    ibeg = jbeg = 0;

  begin = 0;
  for(i=ibeg; i<BHAT_SIZE; i++)    /* bhat loop */
    {
      bhat = (10.0 + (i * 10.0));
      for(j=jbeg; j<FREQ_SIZE; j++)    /* freq loop */
        {
          freq = (10.0 + (j * 10.0));
          a21 = freq * freq;
          for(k=0; k<SAMP_SIZE; k++)    /* time delay loop */
            {
              newt = 1000.0 + (k * (1000.0));
              for(l=0; l<DAMP_SIZE; l++)    /* damp loop */
                {
                  if(l < 10)
                    damp = ((l + 1) * 0.1);
                  else
                    damp = (l - 9.0);
                  a22 = 2.0 * damp * freq;
                  DigStabFunc(A1, A2, P, Q, newt, a22, a21, &J1, &J2, &J3,
                              &J4, &J5, &M1, &M2, &M3, &M4, &N1, &N2, &N3,
                              &N4);
                  subtot[7] = ((J1*M2)/bhat)+(N2-1.0);
                  subtot[6] = (((J1*M3)+(J2*M2))/bhat)+(N3-N2);
                  subtot[5] = (((J1*M4)+(J2*M3)+(J3*M2))/bhat)+(N4-N3);
                  subtot[4] = (((J2*M4)+(J3*M3)+(J4*M2))/bhat)-N4;
                  subtot[3] = ((J3*M4)+(J4*M3)+(J5*M2))/bhat;
                  subtot[2] = ((J4*M4)+(J5*M3))/bhat;
                  subtot[1] = (J5*M4)/bhat;

                  if(zrhqr(subtot, 8, rroots, iroots))
                    {
                      fprintf(stderr, "Eigenvalue calculation problem ");
                      fprintf(stderr, "(i = %d  j = %d  k = %d  l = %d)\n",
                              i, j, k, l);
                      for(m=1; m<=8; m++)
                        {
                          iroots[m] = 0.0;
```

```
                        rroots[m] = 2.0;
                      }
                  }

                for(m=1; m<=8; m++)
                  if((rroots[m]*rroots[m])+(iroots[m]*iroots[m]) > 1.0)
                    break;

                if(m == 9)
                  {
#ifdef STEP
                    if(DigStep(axis, bhat, damp, freq, 1.0/newt, initx,
                               initu, 1.1*settle[(k*FREQ_SIZE)+j], P, Q, A1,
                               A2, &y, &u, &t, &setsize))
                      {
                        fprintf(stderr, "DigStep calculation problem ");
                        fprintf(stderr, "(i = %d  j = %d  k = %d  l = %d)\n",
                                i, j, k, l);
                        buffer[(k*DAMP_SIZE)+l] = 2;
                      }
                    over = initx;
                    set = 1.0;
                    for(m=0; m<setsize; m++)
                      {
                        if(initx > 0.0 && y[m] < over)
                          over = y[m];
                        else if(initx < 0.0 && y[m] > over)
                          over = y[m];
                        if(fabs(y[m]) > fabs(0.05 * initx))
                          set = 1.0;
                        else if(set == 1.0)
                          set = t[m];
                      }
#if 0
                    fprintf(stderr, "%f %f\n", over, set);
#endif
                    if(fabs(over) < fabs(OVERSHOOT * initx) && set != 1.0)
                      {
                        buffer[(k*DAMP_SIZE)+l] = 1;
                        begin++;
                      }
                    else
                      buffer[(k*DAMP_SIZE)+l] = 2;
                    free(y);
                    free(u);
                    free(t);
#else
                    buffer[(k*DAMP_SIZE)+l] = 1;
                    begin++;
#endif
                  }
                else
                  buffer[(k*DAMP_SIZE)+l] = 2;
              }
          }
        if(fwrite((void *) buffer, sizeof(char), curcols, fp) != curcols)
          {
            fprintf(stderr, "data write failed\n");
            fclose(fp);
            exit(1);
          }
        currows++;

        fseek(fp, 0L, SEEK_SET);
        if(!fwrite((void *) &currows, sizeof(int), 1, fp))
          {
            fprintf(stderr, "row data write failed\n");
            fclose(fp);
            exit(1);
```

```
          }
        if(!fwrite((void *) &curcols, sizeof(int), 1, fp))
          {
            fprintf(stderr, "columns data write failed\n");
            fclose(fp);
            exit(1);
          }
        fflush(fp);
        fseek(fp, (currows * curcols * sizeof(char)) + (2 * sizeof(int)),
              SEEK_SET);
      }
    if(jbeg)
      jbeg = 0;
    printf("%3d of %3d (%d)\n", i+1, BHAT_SIZE, begin);
  }

  fclose(fp);
  exit(0);
}


void DigStabFunc(double A1, double A2, double P, double Q, double T,
                 double a22, double a21, double *J1, double *J2, double *J3,
                 double *J4, double *J5, double *M1, double *M2, double *M3,
                 double *M4, double *N1, double *N2, double *N3, double *N4)
{
  double K1, K2, K3, K4, L1, L2, L3, L4, T1, T2, sq;

  T1 = 1/T;
  T2 = T1*T1;
  sq = sqrt(Q);

#ifdef BACKWARD
  /* central difference acceleration and backward difference velocity */
  *J1 = (3.0/(4.0*(T2)))+((3.0*a22)/(2.0*T1))+(a21);
  *J2 = (-4.0/(4.0*(T2)))-((4.0*a22)/(2.0*T1));
  *J3 = (-2.0/(4.0*(T2)))+(a22/(2.0*T1));
  *J4 = 1.0/(T2);
  *J5 = -1.0/(4.0*(T2));
#else
  /* central difference acceleration and  velocity */
  *J1 = (1.0/(4.0*(T2)))+(a22/(2.0*T1))+(a21);
  *J2 = 0.0;
  *J3 = (-2.0/(4.0*(T2)))-(a22/(2.0*T1));
  *J4 = 0.0;
  *J5 = 1.0/(4.0*(T2));
#endif

  K1 = (-A1*P)/(A2*Q);
  K2 = (-A1*P)/(A2*((A2*A2)-Q));
  K3 = (A1*P)/(2.0*Q*(A2-sq));
  K4 = (A1*P)/(2.0*Q*(A2+sq));

  L1 = exp(-A2*T1);
  L2 = exp(-sq*T1);
  L3 = exp(sq*T1);
  L4 = 1;

  *M1 = 0;
  *M2 = (-K1*(L1+L2+L3))-(K2*(L2+L3+L4))-(K3*(L1+L3+L4))-(K4*(L1+L2+L4));
  *M3 = (K1*((L1*L2)+(L3*(L1+L2))))+(K2*((L2*L4)+(L3*(L2+L4))))+
    (K3*((L1*L4)+(L3*(L1+L4))))+(K4*((L1*L4)+(L2*(L1+L4))));
  *M4 = (-K1*L1*L2*L3)-(K2*L2*L3*L4)-(K3*L1*L3*L4)-(K4*L1*L2*L4);

  *N1 = 1.0;
  *N2 = -L1-L2-L3;
  *N3 = (L1*L2)+(L1*L3)+(L2*L3);
  *N4 = -(L1*L2*L3);
```

```
}

int zrhqr(double a[], int m, double rtr[], double rti[])
{
  int j, k;
  double **hess, xr, xi;

  hess = matrix(1,MAXM,1,MAXM);
  if(m > MAXM || a[m] == 0.0)
    {
      fprintf(stderr, "bad args in zrhqr\n");
      free_matrix(hess,1,MAXM,1,MAXM);
      return(1);
    }
  for(k=1;k<=m;k++)
    {
      hess[1][k] = -a[m-k]/a[m];
      for(j=2;j<=m;j++)
        hess[j][k] = 0.0;
      if(k != m)
        hess[k+1][k] = 1.0;
    }
  balanc(hess, m);
  if(hqr(hess,m,rtr,rti))
    {
      free_matrix(hess,1,MAXM,1,MAXM);
      return(1);
    }
  for(j=2; j<=m; j++)
    {
      xr = rtr[j];
      xi = rti[j];
      for(k=j-1; k>=1; k--)
        {
          if(rtr[k] <= xr)
            break;
          rtr[k+1] = rtr[k];
          rti[k+1] = rti[k];
        }
      rtr[k+1] = xr;
      rti[k+1] = xi;
    }
  free_matrix(hess,1,MAXM,1,MAXM);
  return(0);
}


void balanc(double **a, int n)
{
  int last,j,i;
  double s,r,g,f,c,sqrdx;

  sqrdx = RADIX*RADIX;
  last = 0;
  while(last == 0)
    {
      last = 1;
      for(i=1; i<=n; i++)
        {
          r = c = 0.0;
          for(j=1; j<=n; j++)
            if(j != i)
              {
                c += fabs(a[j][i]);
                r += fabs(a[i][j]);
              }
          if(c && r)
            {
              g = r/RADIX;
```

```
                f = 1.0;
                s = c+r;
                while(c<g)
                  {
                    f *= RADIX;
                    c *= sqrdx;
                  }
                g = r*RADIX;
                while(c > g)
                  {
                    f /= RADIX;
                    c /= sqrdx;
                  }
                if((c+r)/f < 0.95*s)
                  {
                    last = 0;
                    g = 1.0/f;
                    for(j=1; j<=n; j++)
                      a[i][j] *= g;
                    for(j=1; j<=n; j++)
                      a[j][i] *= f;
                  }
              }
          }
      }
}


int hqr(double **a, int n, double wr[], double wi[])
{
  int nn, m, l, k, j, its, i, mmin;
  double z, y, x, w, v, u, t, s, r, q, p, anorm;

  anorm = fabs(a[1][1]);
  for(i=2; i<=n; i++)
    for(j=(i-1); j<=n; j++)
      anorm += fabs(a[i][j]);
  nn=n;
  t=0.0;
  while(nn >= 1)
    {
      its = 0;
      do {
        for(l=nn; l>=2; l--)
          {
            s = fabs(a[l-1][l-1])+fabs(a[l][l]);
            if(s == 0.0)
              s=anorm;
            if((double)(fabs(a[l][l-1]) + s) == s)
              break;
          }
        x = a[nn][nn];
        if(l == nn)
          {
            wr[nn] = x+t;
            wi[nn--] = 0.0;
          }
        else
          {
            y = a[nn-1][nn-1];
            w = a[nn][nn-1]*a[nn-1][nn];
            if(l == (nn-1))
              {
                p = 0.5*(y-x);
                q = p*p+w;
                z = sqrt(fabs(q));
                x += t;
                if(q >= 0.0)
                  {
```

```
            z = p+SIGN(z,p);
            wr[nn-1] = wr[nn] = x+z;
            if(z)
              wr[nn] = x-w/z;
            wi[nn-1] = wi[nn] = 0.0;
          }
        else
          {
            wr[nn-1] = wr[nn] = x+p;
            wi[nn-1] = -(wi[nn] = z);
          }
        nn -= 2;
      }
    else
      {
        if(its == 30)
          {
            fprintf(stderr, "Too many iterations in hqr\n");
            return(1);
          }
        if(its == 10 || its == 20)
          {
            t += x;
            for(i=1; i<=nn; i++)
              a[i][i] -= x;
            s = fabs(a[nn][nn-1])+fabs(a[nn-1][nn-2]);
            y = x = 0.75*s;
            w = -0.4375*s*s;
          }
        ++its;
        for(m=(nn-2); m>=l; m--)
          {
            z = a[m][m];
            r = x-z;
            s = y-z;
            p = (r*s-w)/a[m+1][m]+a[m][m+1];
            q = a[m+1][m+1]-z-r-s;
            r = a[m+2][m+1];
            s = fabs(p)+fabs(q)+fabs(r);
            p /= s;
            q /= s;
            r /= s;
            if(m == l)
              break;
            u = fabs(a[m][m-1])*(fabs(q)+fabs(r));
            v = fabs(p)*(fabs(a[m-1][m-1])+fabs(z)+fabs(a[m+1][m+1]));
            if((double)(u+v) == v)
              break;
          }
        for(i=m+2; i<=nn; i++)
          {
            a[i][i-2] = 0.0;
            if(i != (m+2))
              a[i][i-3] = 0.0;
          }
        for(k=m; k<=nn-1; k++)
          {
            if(k != m)
              {
                p = a[k][k-1];
                q = a[k+1][k-1];
                r = 0.0;
                if(k != (nn-1))
                  r = a[k+2][k-1];
                if((x = fabs(p)+fabs(q)+fabs(r)) != 0.0)
                  {
                    p /= x;
                    q /= x;
                    r /= x;
```

```
                        }
                    }
                  if((s = SIGN(sqrt(p*p+q*q+r*r),p)) != 0.0)
                    {
                      if(k == m)
                        {
                          if(l != m)
                            a[k][k-1] = -a[k][k-1];
                        }
                      else
                        a[k][k-1] = -s*x;
                      p += s;
                      x = p/s;
                      y = q/s;
                      z = r/s;
                      q /= p;
                      r /= p;
                      for(j=k; j<=nn; j++)
                        {
                          p = a[k][j]+q*a[k+1][j];
                          if(k != (nn-1))
                            {
                              p += r*a[k+2][j];
                              a[k+2][j] -= p*z;
                            }
                          a[k+1][j] -= p*y;
                          a[k][j] -= p*x;
                        }
                      mmin = nn<k+3 ? nn : k+3;
                      for(i=l; i<=mmin; i++)
                        {
                          p = x*a[i][k]+y*a[i][k+1];
                          if(k != (nn-1))
                            {
                              p += z*a[i][k+2];
                              a[i][k+2] -= p*r;
                            }
                          a[i][k+1] -= p*q;
                          a[i][k] -= p;
                        }
                    }
                }
            }
        }
      } while(l < nn-1);
    }
  return(0);
}

double **matrix(long nrl, long nrh, long ncl, long nch)
{
  long i, nrow = nrh-nrl+1, ncol = nch-ncl+1;
  double **m;

  /* allocate pointers to rows */
  if((m = (double **) malloc((size_t) ((nrow+NR_END) * sizeof(double *))))
     == NULL)
    {
      fprintf(stderr, "allocation failure 1 in matrix()\n");
      exit(1);
    }
  m += NR_END;
  m -= nrl;

  /* allocate rows and set pointers to them */
  if((m[nrl] = (double *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(double))))
     == NULL)
    {
      fprintf(stderr, "allocation failure 2 in matrix()\n");
```

```
      exit(1);
    }
  m[nrl] += NR_END;
  m[nrl] -= ncl;

  for(i=nrl+1; i<=nrh; i++)
    m[i]=m[i-1]+ncol;

  return(m);

}


void free_matrix(double **m, long nrl, long nrh, long ncl, long nch)
{
  free((char *) (m[nrl]+ncl-NR_END));
  free((char *) (m+nrl-NR_END));
}


int DigStep(int bearing, double b, double damp, double freq, double T,
            double initx, double initu, double maxt, double P, double Q,
            double A1, double A2, double **y, double **u, double **t, int *n)
{
  int i, j, k, ttsize, uptsize;
  double a22, a21, g, divs, rT, maxx;
  double *tt, *upt, *ty, *tu, cx[5];
  double px[2], pxdot[2], am[5], vm[3], xm, cuT, cu;
  double ampx, ampu, accel, vel, ampxdot, maxu;
  double rkt[2][4], rk[4];
  /*
   * [y, u, t] = DigStep(bearing, b, damp, freq, T, initx, initu, maxt)
   *     bearing - vacuum pump bearing number
   *               0 = axial bearing
   *               1 = radial 1X bearing
   *               2 = radial 1Y bearing
   *               3 = radial 2X bearing
   *               4 = radial 2Y bearing
   *     b       - controller gain (b hat)
   *     damp    - controller damping ratio
   *     freq    - controller natural frequency
   *     T       - controller sampling interval
   *     initx   - initial position
   *     initu   - initial control
   *     maxt    - maximum response time
   */

  a22 = 2.0 * damp * freq;
  a21 = freq * freq;
  g = 9.807;
  divs = 10.0;
  rT = T / divs;
  (*u) = (*y) = (*t) = NULL;
  (*n) = 0;

  ttsize = ((int) ((maxt / T) + 0.5)) + 1;

  if((tt = (double *) calloc(ttsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigStep.\n");
      return(1);
    }

  uptsize = ((int) ((maxt / rT) + 0.5)) + 1;

  if((upt = (double *) calloc(uptsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigStep.\n");
      free(tt);
```

```
      return(1);
    }

  if((ty = (double *) calloc(uptsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigStep.\n");
      free(tt);
      free(upt);
      return(1);
    }

  if((tu = (double *) calloc(uptsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigStep.\n");
      free(tt);
      free(upt);
      free(ty);
      return(1);
    }

  for(i=1; i<uptsize; i++)
    upt[i] = upt[i-1] + rT;

  for(j=0,i=10; i<uptsize; i+=10)
    tt[j++] = upt[i];

  for(i=0; i<5; i++)
    cx[i] = initx;

  if(!bearing)
    {
      maxx = 0.0002;
      maxu = 3.0;
    }
  else
    {
      maxx = 0.0001;
      maxu = 2.0;
    }

  px[1] = pxdot[0] = pxdot[1] = 0.0;
  px[0] = initx;

#ifdef BACKWARD
  am[0] = 3.0; am[1] = -4.0; am[2] = -2.0; am[3] = 4.0; am[4] = -1.0;
#else
  am[0] = 1.0; am[1] = 0.0; am[2] = -2.0; am[3] = 0.0; am[4] = 1.0;
#endif
  for(i=0; i<5; i++)
    am[i] /= (4.0 * (T * T));

#ifdef BACKWARD
  vm[0] = 3.0 * a22; vm[1] = -4.0 * a22; vm[2] = a22;
#else
  vm[0] = a22; vm[1] = 0.0; vm[2] = -a22;
#endif
  for(i=0; i<3; i++)
    vm[i] /= (2.0 * T);
  xm = a21;
  cuT = cu = initu;

  ampx = ampu = initu;

  rk[0] = rk[3] = 1.0/6.0;
  rk[1] = rk[2] = 1.0/3.0;

  for(j=i=0; i<uptsize; i++)
    {
      if(upt[i] == tt[j])
```

```
  {
    /* update controller position variables */
    for(k=4; k>0; k--)
      cx[k] = cx[k-1];
    cx[0] = px[0];
    /* determine control signal */
    for(accel=0.0,k=0; k<5; k++)
      accel += (am[k] * cx[k]);
    for(vel=0.0,k=0; k<3; k++)
      vel += (vm[k] * cx[k]);
    cu = cuT - ((accel + vel + (xm*cx[0]))/b);
    if(cu > maxu)
      cu = maxu;
    else if(cu < -maxu)
      cu = -maxu;
    cuT = cu;
    j++;
  }

/* determine amplified control */
rkt[0][0] = (-A2*ampx) + (A1*cu);          /* f{y(j),t(j)}              */
ampxdot = ampx + (rkt[0][0]*(rT/2));       /* y*(j+1/2)                 */
rkt[0][1] = (-A2*ampxdot) + (A1*cu);       /* f{y*(j+1/2),t(j+1/2)}     */
ampxdot = ampx + (rkt[0][1]*(rT/2));       /* y**(j+1/2)                */
rkt[0][2] = (-A2*ampxdot) + (A1*cu);       /* f{y**(j+1/2),t(j+1/2)}    */
ampxdot = ampx + (rkt[0][2]*rT);           /* y*(j+1)                   */
rkt[0][3] = (-A2*ampxdot) + (A1*cu);       /* f{y*(j+1),t(j+1)}         */
for(ampxdot=0.0,k=0; k<4; k++)
  ampxdot += (rkt[0][k]*rk[k]);
ampx += (ampxdot * rT);
ampu = ampx;

/* determine new position */
rkt[0][0] = px[1];                         /* f{y(j),t(j)}              */
rkt[1][0] = (Q*px[0]) + (P*ampu);          /* f{y(j),t(j)}              */
pxdot[0] = px[0] + (rkt[0][0]*(rT/2));     /* y*(j+1/2)                 */
pxdot[1] = px[1] + (rkt[1][0]*(rT/2));     /* y*(j+1/2)                 */
rkt[0][1] = pxdot[1];                      /* f{y*(j+1/2),t(j+1/2)}     */
rkt[1][1] = (Q*pxdot[0]) + (P*ampu);       /* f{y*(j+1/2),t(j+1/2)}     */
pxdot[0] = px[0] + (rkt[0][1]*(rT/2));     /* y**(j+1/2)                */
pxdot[1] = px[1] + (rkt[1][1]*(rT/2));     /* y**(j+1/2)                */
rkt[0][2] = pxdot[1];                      /* f{y**(j+1/2),t(j+1/2)}    */
rkt[1][2] = (Q*pxdot[0]) + (P*ampu);       /* f{y**(j+1/2),t(j+1/2)}    */
pxdot[0] = px[0] + (rkt[0][2]*rT);         /* y*(j+1)                   */
pxdot[1] = px[1] + (rkt[1][2]*rT);         /* y*(j+1)                   */
rkt[0][3] = pxdot[1];                      /* f{y*(j+1),t(j+1)}         */
rkt[1][3] = (Q*pxdot[0]) + (P*ampu);       /* f{y*(j+1),t(j+1)}         */
pxdot[0] = pxdot[1] = 0.0;
for(k=0; k<4; k++)
  {
    pxdot[0] += (rkt[0][k]*rk[k]);
    pxdot[1] += (rkt[1][k]*rk[k]);
  }
if(bearing == 0)
  pxdot[1] -= g;

/* save values for plotting */
ty[i] = px[0];
tu[i] = cu;

/* update variables */
px[0] += (pxdot[0] * rT);
px[1] += (pxdot[1] * rT);

if(px[0] > maxx)
  px[0] = maxx;
else if(px[0] < -maxx)
  px[0] = -maxx;
```

```
    }

  free(tt);

  (*y) = ty;
  (*u) = tu;
  (*t) = upt;
  (*n) = uptsize;

  return(0);

}


int DigReference(double bfreq, double efreq, double dfreq, double bT,
                 double eT, double dT, double initx, double initu, double maxt,
                 double **settle, int *n)
{
  int i, j, k, fsize, tsize, setsize, ee, bb;
  double *time, upt, *freq, output, *set;

  (*settle) = NULL;
  (*n) = 0;

  tsize = ((int) (((eT - bT) / dT) + 0.5)) + 1;
  fsize = ((int) ((efreq - bfreq) / dfreq)) + 1;
  setsize = fsize * tsize;

  if((time = (double *) calloc(tsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigReference.\n");
      return(1);
    }

  if((freq = (double *) calloc(fsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigReference.\n");
      free(time);
      return(1);
    }

  if((set = (double *) calloc(setsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigReference.\n");
      free(time);
      free(freq);
      return(1);
    }

  for(time[0]=bT,i=1; i<tsize; i++)
    time[i] = time[i-1] + dT;

  for(freq[0]=bfreq,i=1; i<fsize; i++)
    freq[i] = freq[i-1] + dfreq;

  for(i=0; i<tsize; i++)
    {
      for(j=0; j<fsize; j++)
        {
          ee = (int) (maxt*time[i]);
          bb = 1;
          k = ee >> 1;
          while(k != 0)
            {
              upt = k / time[i];
              /*
               * This equation is the solution of the inverse Laplace transform
               * of a second order system having a damping ratio of 1.0 and
               * determining the closest sampling time to that point
```

```
                */
               output = 1.0 - ((1.0 + (freq[j]*upt)) * exp(-freq[j] * upt));
               if(output < 0.95)
                 bb = k;
               else
                 ee = k;
               k = bb + ((ee - bb) >> 1);
               if(k == ee || k == bb)
                 {
                   set[(i*fsize)+j] = k / time[i];
                   k = 0;
                 }
             }
         }
    }

  free(time);
  free(freq);

  (*settle) = set;
  (*n) = setsize;

  return(0);

}
```

## H.1.2 mat2text.c

This program examines the binary character matrix produced by limits.c and creates a text file listing the values of sampling rate, damping ratio, natural frequency, and feedback gain that produced stable systems. This program made it possible to create and delete the considerably larger text file from the smaller binary file at any time.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MEMERR(x) \
  fprintf(stderr, "Out of memory error - %s (%d)\n", x, __LINE__);
#define FUNCERR(x,y) \
  fprintf(stderr, "Error encountered in %s - %s (%d)\n", x, y, __LINE__);

#define STATUS_OK  0
#define STATUS_NOK 1


void main(int argc, char **argv)
{
  int i,j;
  int begrow, begcol, rows, cols, mag;
  int matrows, matcols;
  char *infile, *outfile, *buffer;
  FILE *ifp, *ofp;
  float freq, time, damp, bhat;

  begrow = begcol = rows = cols = mag = 0;

  if(argc < 3)
    {
      fprintf(stderr, "syntax error ...\n\n");
      fprintf(stderr, "Syntax:  mat2text infile outfile [-r n] [-c m]");
      fprintf(stderr, " [-nr l] [-nc k]\n");
```

```
    fprintf(stderr, "         where:  infile  -> matrix pathname\n");
    fprintf(stderr, "                 outfile -> text file pathname\n");
    fprintf(stderr, "                 -r  n   -> start at nth row\n");
    fprintf(stderr, "                 -c  m   -> start at mth column\n");
    fprintf(stderr, "                 -nr l   -> convert l rows\n");
    fprintf(stderr, "                 -nc k   -> convert k columns\n");
    exit(1);
  }

for(i=1; i<argc; i++)
  {
    if(i == 1)
      infile = argv[i];
    else if(i == 2)
      outfile = argv[i];
    else if(argv[i][0] == '-')
      {
        if(argv[i][1] == 'r')
          begrow = atoi(argv[i+1]);
        else if(argv[i][1] == 'c')
          begcol = atoi(argv[i+1]);
        else if(argv[i][1] == 'n')
          {
            if(argv[i][2] == 'r')
              rows = atoi(argv[i+1]);
            else if(argv[i][2] == 'c')
              cols = atoi(argv[i+1]);
          }
      }
  }

if((ifp = fopen(infile, "r")) == NULL)
  {
    fprintf(stderr, "Unable to open input file \"%s\"\n\n", infile);
    exit(1);
  }

if((ofp = fopen(outfile, "w")) == NULL)
  {
    fprintf(stderr, "Unable to open output file \"%s\"\n\n", outfile);
    exit(1);
  }

if(!fread((void *) &matrows, sizeof(int), 1, ifp))
  {
    fprintf(stderr, "Error encountered reading from \"%s\"\n\n", infile);
    fclose(ifp);
    fclose(ofp);
    exit(1);
  }

if(!fread((void *) &matcols, sizeof(int), 1, ifp))
  {
    fprintf(stderr, "Error encountered reading from \"%s\"\n\n", infile);
    fclose(ifp);
    fclose(ofp);
    exit(1);
  }

if(!rows)
  rows = matrows;

if(!cols)
  cols = matcols;

if(!mag)
  mag = 1;

if(begrow+rows > matrows)
```

```
    {
      fprintf(stderr,
              "Requested rows (%d) is greater than matrix rows (%d)\n\n",
              begrow+rows, matrows);
      fclose(ifp);
      fclose(ofp);
      exit(1);
    }

  if(begcol+cols > matcols)
    {
      fprintf(stderr,
              "Requested columns (%d) is greater than matrix columns (%d)\n\n",
              begcol+cols, matcols);
      fclose(ifp);
      fclose(ofp);
      exit(1);
    }

  if(begrow)
    fseek(ifp, begrow*matcols, SEEK_CUR);

  if((buffer = (char *) calloc(matcols, sizeof(char))) == NULL)
    {
      MEMERR("mat2text");
      fclose(ifp);
      fclose(ofp);
      exit(1);
    }

  for(i=0; i<rows; i++)
    {
      if(fread((void *) buffer, sizeof(char), matcols, ifp) != matcols)
        {
          fprintf(stderr, "Data read error ...\n");
          fclose(ifp);
          fclose(ofp);
          exit(1);
        }
      for(j=begcol; j<begcol+cols; j++)
        if(buffer[j] == 1)
          {
            if(j%19 < 10)
              damp = ((j%19) + 1) * 0.1;
            else
              damp = (j%19) - 9.0;
            time = 1000.0 + (((int) j/19) * 1000.0);
            freq = 10.0 + (((begrow + i)%100) * 10.0);
            bhat = 10.0 + (((int) ((begrow + i) / 100)) * 10.0);
            fprintf(ofp, "%7.1f    %5.2f    %6.1f    %6.1f\n", time, damp,
                    bhat, freq);
          }
    }

  fclose(ifp);
  fclose(ofp);
  exit(0);

}
```

## H.1.3 mat2tiff.c

This program examines the binary character matrix produced by limits.c and creates a TIFF file using the four dimensional mapping technique described in a previous chapter. This

allowed this researcher to visualize the effect that changes in controller variables had on closed loop system stability. This program made it possible to create and delete this TIFF file from the binary file at any time.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define MEMERR(x) \
  fprintf(stderr, "Out of memory error - %s (%d)\n", x, __LINE__);
#define FUNCERR(x,y) \
  fprintf(stderr, "Error encountered in %s - %s (%d)\n", x, y, __LINE__);

#define STATUS_OK  0
#define STATUS_NOK 1

#define FIELDNUM       18
#define STRIPSIZE      8*1024

#ifdef TIFF_COMP
#define HASH_SIZE  101
enum { LZW_Initialize, LZW_Compress, LZW_Cleanup };
#endif

typedef union i_tiffoff
{
  unsigned int    uint;
  unsigned short  ushort[2];
} iTiffOff;

#define  ITiffOffInt(p)         ((p).uint)
#define  ITiffOffShort(p)       ((p).ushort)
#define  ITiffOffShortHigh(p)   (((p).ushort)[0])
#define  ITiffOffShortLow(p)    (((p).ushort)[1])


typedef struct i_tiffdir
{
  unsigned short Tag;
  unsigned short Type;
  int            Length;
  iTiffOff       Offset;
} iTiffDirStruct, *iTiffDir;

#define  ITiffDirTag(p)           ((p)->Tag)
#define  ITiffDirType(p)          ((p)->Type)
#define  ITiffDirLength(p)        ((p)->Length)
#define  ITiffDirOffset(p)        (ITiffOffInt((p)->Offset))
#define  ITiffDirOffsetShort(p)   (ITiffOffShortHigh((p)->Offset))

/* TIFF Tag Types */
#define BYTE     1
#define ASCII    2
#define SHORT    3
#define LONG     4
#define RATIONAL 5

/* TIFF Tags */
#define NEW_SUBFILE_TYPE            254
#define IMAGE_WIDTH                 256
#define IMAGE_LENGTH                257
#define BITS_PER_SAMPLE             258
#define COMPRESSION                 259
#define PHOTOMETRIC_INTERPRETATION  262
#define DOCUMENT_NAME               269
```

```
#define STRIP_OFFSETS               273
#define SAMPLES_PER_PIXEL           277
#define ROWS_PER_STRIP              278
#define STRIP_BYTE_COUNTS           279
#define X_RESOLUTION                282
#define Y_RESOLUTION                283
#define PLANAR_CONFIGURATION        284
#define RESOLUTION_UNIT             296
#define SOFTWARE                    305
#define DATE_TIME                   306
#define COLOR_MAP                   320

/* Tiff Color Representations */
#define RGB_COLOR      2
#define PALETTE_COLOR  3

/* Compression type defines */
#define NO_COMPRESS         1
#define HUFFMAN_COMPRESS    2
#define LZW_COMPRESS        5
#define PACKBITS_COMPRESS   32773

#ifdef TIFF_COMP
/* hash table member */
typedef struct i_tiffhash
{
  struct i_tiffhash *Next;
  unsigned char     *Values;
  unsigned int       Size;
  unsigned int       Index;
} iTiffHashStruct, *iTiffHash;

#define  ITiffHashNext(p)     ((p)->Next)
#define  ITiffHashChars(p)    ((p)->Values)
#define  ITiffHashSize(p)     ((p)->Size)
#define  ITiffHashIndex(p)    ((p)->Index)

/* compressed data buffer union */
typedef union i_lzwvalue
{
  unsigned int   Integer;
  unsigned char  Byte[4];
} iLZWValueUnion, *iLZWValue;

#define  ILZWValueInt(p)    ((p).Integer)
#define  ILZWValueByte(p)   ((p).Byte)

/* LZW table constants */
#define LZWClear    256
#define LZWEOI      257

#endif

static int WriteTIFFMatrix(FILE *ifp, FILE *ofp, int begrow, int begcol,
                           int rows, int cols, int rowlen, int compress,
                           int color_type, int mag);
#ifdef TIFF_COMP
static void TiffCompressLZW(void *old, int size, int bytes, void *new,
                            int *newsize, int flag);
#endif

void main(int argc, char **argv)
{
  int i,j;
  int begrow, begcol, rows, cols, mag;
  int matrows, matcols;
  char *infile, *outfile;
  FILE *ifp, *ofp;
```

```
begrow = begcol = rows = cols = mag = 0;

if(argc < 3)
  {
    fprintf(stderr, "syntax error ...\n\n");
    fprintf(stderr, "Syntax:  mat2tiff infile outfile [-r n] [-c m]");
    fprintf(stderr, " [-nr l] [-nc k] [-m j]\n");
    fprintf(stderr, "         where:  infile  -> matrix pathname\n");
    fprintf(stderr, "                 outfile -> TIFF pathname\n");
    fprintf(stderr, "                 -r  n   -> start at nth row\n");
    fprintf(stderr, "                 -c  m   -> start at mth column\n");
    fprintf(stderr, "                 -nr l   -> convert l rows\n");
    fprintf(stderr, "                 -nc k   -> convert k columns\n");
    fprintf(stderr, "                 -m  j   -> magnify j times\n\n");
    exit(1);
  }

for(i=1; i<argc; i++)
  {
    if(i == 1)
      infile = argv[i];
    else if(i == 2)
      outfile = argv[i];
    else if(argv[i][0] == '-')
      {
        if(argv[i][1] == 'r')
          begrow = atoi(argv[i+1]);
        else if(argv[i][1] == 'c')
          begcol = atoi(argv[i+1]);
        else if(argv[i][1] == 'm')
          mag = atoi(argv[i+1]);
        else if(argv[i][1] == 'n')
          {
            if(argv[i][2] == 'r')
              rows = atoi(argv[i+1]);
            else if(argv[i][2] == 'c')
              cols = atoi(argv[i+1]);
          }
      }
  }

if((ifp = fopen(infile, "r")) == NULL)
  {
    fprintf(stderr, "Unable to open input file \"%s\"\n\n", infile);
    exit(1);
  }

if((ofp = fopen(outfile, "w")) == NULL)
  {
    fprintf(stderr, "Unable to open output file \"%s\"\n\n", outfile);
    exit(1);
  }

if(!fread((void *) &matrows, sizeof(int), 1, ifp))
  {
    fprintf(stderr, "Error encountered reading from \"%s\"\n\n", infile);
    fclose(ifp);
    fclose(ofp);
    exit(1);
  }

if(!fread((void *) &matcols, sizeof(int), 1, ifp))
  {
    fprintf(stderr, "Error encountered reading from \"%s\"\n\n", infile);
    fclose(ifp);
    fclose(ofp);
    exit(1);
  }
```

```
  if(!rows)
    rows = matrows;

  if(!cols)
    cols = matcols;

  if(!mag)
    mag = 1;

  if(begrow+rows > matrows)
    {
      fprintf(stderr,
              "Requested rows (%d) is greater than matrix rows (%d)\n\n",
              begrow+rows, matrows);
      fclose(ifp);
      fclose(ofp);
      exit(1);
    }

  if(begcol+cols > matcols)
    {
      fprintf(stderr,
              "Requested columns (%d) is greater than matrix columns (%d)\n\n",
              begcol+cols, matcols);
      fclose(ifp);
      fclose(ofp);
      exit(1);
    }

  if(begrow)
    fseek(ifp, begrow*matcols, SEEK_CUR);

  if(WriteTIFFMatrix(ifp, ofp, begrow, begcol, rows, cols, matcols,
                     LZW_COMPRESS, PALETTE_COLOR, mag) != STATUS_OK)
    {
      FUNCERR("WriteTIFFMatrix", "mat2tiff");
      fclose(ifp);
      fclose(ofp);
      exit(1);
    }

  fclose(ifp);
  fclose(ofp);
  exit(0);

}

/***************************** MODULE INFORMATION ***************************
 * NAME OF MODULE : WriteTIFFNetgraph
 * DESCRIPTION    :
 ****************************************************************************/
static int WriteTIFFMatrix(FILE *ifp, FILE *ofp, int begrow, int begcol,
                           int rows, int cols, int rowlen, int compress,
                           int color_type, int mag)
{
  int i, j, k, m, index, size;
  int shift, bytes, pindsize;
  int *pixind;
  short temp;
  long diroff, offset, max;
  iTiffDir *expdir;
  const char *whoami = "mat2tiff v1.0";
  char *date;
  void *strip, **rgb;
#ifdef TIFF_COMP
  void *cstrip;
  int l, cstrsize;
  unsigned int *coffsets, *csizes;
#endif
```

```
  unsigned short red, green, blue;
  struct tm *tstruc;
  time_t tmstr;
  char *image;
  unsigned long pix;

  /* write TIFF file header */

#if defined(__OS2__) || defined(MIPSEL)
  fprintf(ofp, "II");
#else
  fprintf(ofp, "MM");
#endif

  bytes = sizeof(unsigned char);

  if(color_type != PALETTE_COLOR)
    bytes *= 3;

  size = FIELDNUM;
  if(color_type != PALETTE_COLOR)
    size--;

  if((expdir = (iTiffDir *) calloc(size, sizeof(iTiffDir)))
     == NULL)
    {
      MEMERR("WriteTIFFMatrix");
      return(STATUS_NOK);
    }

  for(i=0; i<size; i++)
    if((expdir[i] = (iTiffDir) calloc(1, sizeof(iTiffDirStruct))) == NULL)
      {
        MEMERR("WriteTIFFMatrix");
        for(; i>=0; i--)
          if(expdir[i])
            free((char *) expdir[i]);
        if(expdir)
          free((char *) expdir);
        return(STATUS_NOK);
      }

  ITiffDirTag(expdir[0]) = 42;
  fwrite((void *) &ITiffDirTag(expdir[0]), sizeof(char),
         sizeof(ITiffDirTag(expdir[0])), ofp);
  ITiffDirOffset(expdir[0]) = 12;
  fwrite((void *) &ITiffDirOffset(expdir[0]), sizeof(char),
         sizeof(ITiffDirOffset(expdir[0])), ofp);
  ITiffDirOffset(expdir[0]) = 0;
  fwrite((void *) &ITiffDirOffset(expdir[0]), sizeof(char),
         sizeof(ITiffDirOffset(expdir[0])), ofp);


  /* write TIFF Image file directory entry */
  ITiffDirTag(expdir[0]) = size;
  if(fwrite((void *) &ITiffDirTag(expdir[0]), sizeof(char),
            sizeof(ITiffDirTag(expdir[0])), ofp) !=
     sizeof(ITiffDirTag(expdir[0])))
    goto werror1;

  /* get file offset so we can write TIFF directory later */
  diroff = ftell(ofp);

  /* write dummy TIFF directory */
  for(i=0; i<size; i++)
    if(fwrite((void *) expdir[i], sizeof(char), sizeof(*expdir[i]), ofp) !=
       sizeof(*expdir[i]))
      goto werror1;
```

```
  if(fwrite((void *) &ITiffDirOffset(expdir[0]), sizeof(char),
            sizeof(ITiffDirOffset(expdir[0])), ofp) !=
     sizeof(ITiffDirOffset(expdir[0])))
    goto werror1;

  /* 0  write NewSubfileType field */
  index = 0;
  ITiffDirTag(expdir[index]) = NEW_SUBFILE_TYPE;
  ITiffDirType(expdir[index]) = LONG;
  ITiffDirLength(expdir[index]) = 1;
  ITiffDirOffset(expdir[index]) = 0;  /* default */

  /* 1  write ImageWidth field */
  index++;
  ITiffDirTag(expdir[index]) = IMAGE_WIDTH;
  ITiffDirType(expdir[index]) = LONG;
  ITiffDirLength(expdir[index]) = 1;
  ITiffDirOffset(expdir[index]) = mag * cols;

  /* 2  write ImageLength field */
  index++;
  ITiffDirTag(expdir[index]) = IMAGE_LENGTH;
  ITiffDirType(expdir[index]) = LONG;
  ITiffDirLength(expdir[index]) = 1;
  ITiffDirOffset(expdir[index]) = mag * rows;

  /* 3  write BitsPerSample field */
  index++;
  ITiffDirTag(expdir[index]) = BITS_PER_SAMPLE;
  ITiffDirType(expdir[index]) = SHORT;
  temp = 8;
  if(color_type == PALETTE_COLOR)
    {
      ITiffDirLength(expdir[index]) = 1;
      ITiffDirOffsetShort(expdir[index]) = temp;
    }
  else
    {
      ITiffDirLength(expdir[index]) = 3;
      ITiffDirOffset(expdir[index]) = ftell(ofp);

      for(i=0; i<3; i++)
        if(fwrite((void *) &temp, sizeof(char), sizeof(temp), ofp) !=
           sizeof(temp))
          goto werror1;
    }
  shift = (int) temp;  /* save for RGB correction */

  /* 4  write Compression field - packbits recommended, easy to implement */
  index++;
  ITiffDirTag(expdir[index]) = COMPRESSION;
  ITiffDirType(expdir[index]) = SHORT;
  ITiffDirLength(expdir[index]) = 1;
#ifdef TIFF_COMP
  switch(compress)
    {
    case LZW_COMPRESS:
    case NO_COMPRESS:
      ITiffDirOffsetShort(expdir[index]) = compress;  /* LZW compression */
      break;
    default:
      ITiffDirOffsetShort(expdir[index]) = NO_COMPRESS; /* No compression */
      break;
    }
#else
  ITiffDirOffsetShort(expdir[index]) = NO_COMPRESS;  /* No compression */
#endif

  /* 5  write PhotometricInterpretation field */
```

```
index++;
ITiffDirTag(expdir[index]) = PHOTOMETRIC_INTERPRETATION;
ITiffDirType(expdir[index]) = SHORT;
ITiffDirLength(expdir[index]) = 1;
switch(color_type)
  {
  case RGB_COLOR:
  case PALETTE_COLOR:
    ITiffDirOffsetShort(expdir[index]) = color_type;
    break;
  default: /* pixel values are RGB */
    ITiffDirOffsetShort(expdir[index]) = RGB_COLOR;
    break;
  }

/* 6  write DocumentName field */
index++;
ITiffDirTag(expdir[index]) = DOCUMENT_NAME;
ITiffDirType(expdir[index]) = ASCII;
i = strlen("No Name")+1;
ITiffDirLength(expdir[index]) = i;
ITiffDirOffset(expdir[index]) = ftell(ofp);
fwrite((void *) "No Name", sizeof(char), i, ofp);
if(i%2 > 0)
  if(fwrite((void *) (((unsigned int) "No Name")+i-1), sizeof(char),
            1, ofp) != sizeof(char))
    goto werror1;

/* 8  write SamplesPerPixel field */
index = 8;
ITiffDirTag(expdir[index]) = SAMPLES_PER_PIXEL;
ITiffDirType(expdir[index]) = SHORT;
ITiffDirLength(expdir[index]) = 1;
/* 3 = RGB, 1 = bilevel, grayscale, palette color */
if(color_type == RGB_COLOR)
  ITiffDirOffsetShort(expdir[index]) = 3;
else
  ITiffDirOffsetShort(expdir[index]) = 1;

/* 11  write XResolution field - Pixels per Centimeter */
index = 11;
ITiffDirTag(expdir[index]) = X_RESOLUTION;
ITiffDirType(expdir[index]) = RATIONAL;
ITiffDirLength(expdir[index]) = 1;
ITiffDirOffset(expdir[index]) = ftell(ofp);
i = (int) (((1280.0/356.0) + 0.5) * 10);
if(fwrite((void *) &i, sizeof(char), sizeof(int), ofp) != sizeof(int))
  goto werror1;
i = 1;
if(fwrite((void *) &i, sizeof(char), sizeof(int), ofp) != sizeof(int))
  goto werror1;

/* 12  write YResolution field - Pixels per Centimeter */
index++;
ITiffDirTag(expdir[index]) = Y_RESOLUTION;
ITiffDirType(expdir[index]) = RATIONAL;
ITiffDirLength(expdir[index]) = 1;
ITiffDirOffset(expdir[index]) = ftell(ofp);
i = (int) (((1024.0/284.0) + 0.5) * 10);
if(fwrite((void *) &i, sizeof(char), sizeof(int), ofp) != sizeof(int))
  goto werror1;
i = 1;
if(fwrite((void *) &i, sizeof(char), sizeof(int), ofp) != sizeof(int))
  goto werror1;

/* 13  write PlanarConfiguration field */
index++;
ITiffDirTag(expdir[index]) = PLANAR_CONFIGURATION;
ITiffDirType(expdir[index]) = SHORT;
```

```
ITiffDirLength(expdir[index]) = 1;
ITiffDirOffsetShort(expdir[index]) = 1;  /* samples stored contiguously */

/* 14  write ResolutionUnit field */
index++;
ITiffDirTag(expdir[index]) = RESOLUTION_UNIT;
ITiffDirType(expdir[index]) = SHORT;
ITiffDirLength(expdir[index]) = 1;
ITiffDirOffsetShort(expdir[index]) = 3;  /* centimeter */

/* 15  write Software field */
index++;
ITiffDirTag(expdir[index]) = SOFTWARE;
ITiffDirType(expdir[index]) = ASCII;
i = strlen(whoami)+1;
ITiffDirLength(expdir[index]) = i;
ITiffDirOffset(expdir[index]) = ftell(ofp);
if(fwrite((void *) whoami, sizeof(char), i, ofp) != i)
  goto werror1;
if(i%2 > 0)
  if(fwrite((void *) (((unsigned int) whoami)+i-1), sizeof(char), 1, ofp) !=
     1)
    goto werror1;

/* 16  write DateTime field */
index++;
ITiffDirTag(expdir[index]) = DATE_TIME;
ITiffDirType(expdir[index]) = ASCII;
if((date = (char *) calloc(20, sizeof(char))) != NULL)
  {
    ITiffDirLength(expdir[index]) = 20;
    tmstr = time(NULL);
    tstruc = localtime(&tmstr);
    sprintf(date, "%04d:%02d:%02d %02d:%02d:%02d", tstruc->tm_year+1900,
            tstruc->tm_mon, tstruc->tm_mday, tstruc->tm_hour, tstruc->tm_min,
            tstruc->tm_sec);
    ITiffDirOffset(expdir[index]) = ftell(ofp);
    if(fwrite((void *) date, sizeof(char), 20, ofp) != 20)
      goto werror1;
  }
else
  {
    ITiffDirLength(expdir[index]) = 0;
    ITiffDirOffset(expdir[index]) = 0;
  }

free(date);

/* 17  write ColorMap field */
if(color_type == PALETTE_COLOR)
  {
    index++;
    ITiffDirTag(expdir[index]) = COLOR_MAP;
    ITiffDirType(expdir[index]) = SHORT;
    temp = 8;
    ITiffDirLength(expdir[index]) = 3*(1<<temp);
    ITiffDirOffset(expdir[index]) = ftell(ofp);
    temp = 0;
    for(i=0; i<ITiffDirLength(expdir[index]); i++)
      if(fwrite((void *) &temp, sizeof(short), sizeof(char), ofp) !=
         sizeof(char))
        goto werror1;
  }

/* 9  write RowsPerStrip field                                          */
/* recommended that this be set such that the size of each strip is */
/* about 8K bytes.                                                      */
index = 9;
if((i = (int) (STRIPSIZE/(ITiffDirOffset(expdir[1])*bytes))) < 1)
```

```
  {
    i = 1;
    size = ITiffDirOffset(expdir[1])*bytes;
  }
else
  size = i*ITiffDirOffset(expdir[1])*bytes;

ITiffDirTag(expdir[index]) = ROWS_PER_STRIP;
ITiffDirType(expdir[index]) = LONG;
ITiffDirLength(expdir[index]) = 1;
ITiffDirOffset(expdir[index]) = i;

/* 10  write StripByteCounts field */
index = 10;
ITiffDirTag(expdir[index]) = STRIP_BYTE_COUNTS;
ITiffDirType(expdir[index]) = LONG;
ITiffDirLength(expdir[index]) =
  (int) (ITiffDirOffset(expdir[2])/ITiffDirOffset(expdir[9]));
if(ITiffDirOffset(expdir[2])%ITiffDirOffset(expdir[9]) > 0)
   ITiffDirLength(expdir[index])++;
if(ITiffDirLength(expdir[index]) > 1)
  ITiffDirOffset(expdir[index]) = ftell(ofp);

offset = ITiffDirOffset(expdir[1])*bytes;
offset *= ITiffDirOffset(expdir[2]);
while(offset > size)
  {
    if(fwrite((void *) (&size), sizeof(char), sizeof(int), ofp) !=
       sizeof(int))
      goto werror1;
    offset -= size;
  }
if(offset)
  {
    if(ITiffDirLength(expdir[index]) > 1)
      {
        if(fwrite((void *) (&offset), sizeof(char), sizeof(int), ofp) !=
           sizeof(int))
          goto werror1;
      }
    else
      ITiffDirOffset(expdir[index]) = offset;
  }

/* 7  write StripOffsets field */
index = 7;
if(size%2 > 0)  /* make sure size is even so offsets are correct */
  size++;
ITiffDirTag(expdir[index]) = STRIP_OFFSETS;
ITiffDirType(expdir[index]) = LONG;
ITiffDirLength(expdir[index]) = ITiffDirLength(expdir[10]);
ITiffDirOffset(expdir[index]) = offset = ftell(ofp);
if(ITiffDirLength(expdir[index]) > 1)
  {
    offset += (ITiffDirLength(expdir[index])*sizeof(int));
    for(i=0; i<ITiffDirLength(expdir[index]); i++)
      {
        if(fwrite((void *) (&offset), sizeof(char), sizeof(unsigned int),
                  ofp) != sizeof(int))
          goto werror1;
        offset += size;
      }
  }

image = NULL;
if((image = (char *) calloc(mag*rowlen, sizeof(char))) == NULL)
  {
    MEMERR("WriteTIFFMatrix");
    goto werror1;
```

```
    }

  /* allocate memory for netgraph drawing routine */
  if((strip = (void *) calloc(size, sizeof(char))) == NULL)
    {
      MEMERR("WriteTIFFMatrix");
      goto werror2;
    }

  pindsize = 2;

  /* needed in order for werror3 to work properly */
  pixind = NULL;
#ifdef TIFF_COMP
  csizes = coffsets = NULL;
  cstrip = NULL;
#endif

  if((rgb = (void **) calloc(pindsize, sizeof(void *))) == NULL)
    {
      MEMERR("WriteTIFFMatrix");
      goto werror3;
    }

  if((pixind = (int *) calloc(pindsize, sizeof(int))) == NULL)
    {
      MEMERR("WriteTIFFMatrix");
      goto werror3;
    }

#ifdef TIFF_COMP
  if(ITiffDirOffsetShort(expdir[4]) != NO_COMPRESS)
    {
      if((coffsets = (unsigned int *) calloc(ITiffDirLength(expdir[10]),
                                             sizeof(unsigned int))) == NULL)
        {
          MEMERR("WriteTIFFMatrix");
          goto werror3;
        }

      if((csizes = (unsigned int *) calloc(ITiffDirLength(expdir[10]),
                                           sizeof(unsigned int))) == NULL)
        {
          MEMERR("WriteTIFFMatrix");
          goto werror3;
        }

      if((cstrip = (void *) calloc(size, sizeof(char))) == NULL)
        {
          MEMERR("WriteTIFFMatrix");
          goto werror3;
        }

      if(ITiffDirOffsetShort(expdir[4]) == LZW_COMPRESS)
        TiffCompressLZW(NULL, 0, 0, NULL, &cstrsize, LZW_Initialize);
      if(cstrsize != STATUS_OK)
        {
          FUNCERR("TiffCompressLZW", "WriteTIFFMatrix");
          goto werror3;
        }
      1 = 0;
    }
#endif

  /* get Netgraph pixel/RGB values */
  k = bytes;
  if(color_type != PALETTE_COLOR)
    k /= 3;
  for(i=0; i<2; i++)
```

```
    {
      if((rgb[i] = (void *) calloc(3, k)) == NULL)
        {
          MEMERR("WriteTIFFMatrix");
          goto werror4;
        }
      pixind[i] = i+1;
      if(i)
        {
          ((unsigned char *) rgb[i])[0] = 255;
          ((unsigned char *) rgb[i])[1] = 255;
          ((unsigned char *) rgb[i])[2] = 255;
        }
      else
        {
          ((unsigned char *) rgb[i])[0] = 0;
          ((unsigned char *) rgb[i])[1] = 0;
          ((unsigned char *) rgb[i])[2] = 0;
        }
    }

  /* write pixel RGB data */
  max = size/bytes;
  offset = 0;
  m = mag;

  for(i=0; i<mag*rows; i++)
    {
      if(mag > 1)
        {
          if(m == mag)
            {
              for(j=0; j<rowlen; j++)
                {
                  if(!fread((void *) (((unsigned int) image)+(j*mag)),
                            sizeof(char), 1, ifp))
                    goto werror4;
                  for(k=1; k<mag; k++)
                    image[(j*mag)+k] = image[j*mag];
                }
              m = 0;
            }
        }
      else
        if(fread((void *) image, sizeof(char), rowlen, ifp) != rowlen)
          goto werror4;

      for(j=0; j<mag*cols; j++)
        {

          if(color_type == PALETTE_COLOR)
            memcpy((void *) (((unsigned int) strip)+offset),
                   (void *) (((unsigned int) image)+begcol+j), bytes);
          else
            {
              pix = (unsigned long) (((unsigned int) image)+begcol+j);
              for(k=0; k<pindsize; k++)
                if(pixind[k] == pix)
                  break;
              if(k)
                memcpy((void *) (((unsigned int) strip)+offset), rgb[k],
                       bytes);
            }
          offset += (bytes);
        }
      if((offset+(mag*cols*bytes)) > size)
        {
#ifdef TIFF_COMP
          if(ITiffDirOffsetShort(expdir[4]) != NO_COMPRESS)
```

```
            {
              if(ITiffDirOffsetShort(expdir[4]) == LZW_COMPRESS)
                TiffCompressLZW(strip, max, bytes, cstrip, &cstrsize,
                                LZW_Compress);
              csizes[l] = (unsigned int) cstrsize;
              coffsets[l++] = (unsigned int) ftell(ofp);
#if 0
              printf("l = %d    Offset = %d    Compressed size = %d\n",
                     l-1, coffsets[l-1], cstrsize);
#endif
              if(cstrsize%2 > 0)
                cstrsize++;

              if(fwrite(cstrip, sizeof(char), cstrsize, ofp) != cstrsize)
                goto werror4;
            }
          else
            if(fwrite(strip, bytes, max, ofp) != max)
              goto werror4;
#else
          if(fwrite(strip, bytes, max, ofp) != max)
            goto werror4;
#endif
          offset = 0;
          if(color_type != PALETTE_COLOR)
            for(k=0;
                k<ITiffDirOffset(expdir[1])*ITiffDirOffset(expdir[9]);
                k++)
              memcpy((void *) (((unsigned int) strip)+(k*bytes)),
                     rgb[0], bytes);
        }
      m++;
#if 0
      printf("line = %d\n", i);
#endif
    }

  if(offset)
    {
#ifdef TIFF_COMP
      if(ITiffDirOffsetShort(expdir[4]) != NO_COMPRESS)
        {
          if(ITiffDirOffsetShort(expdir[4]) == LZW_COMPRESS)
            TiffCompressLZW(strip, offset, bytes, cstrip, &cstrsize,
                            LZW_Compress);
          csizes[l] = (unsigned int) cstrsize;
          coffsets[l] = (unsigned int) ftell(ofp);
#if 0
          printf("l = %d    Offset = %d    Compressed size = %d\n",
                 l-1, coffsets[l-1], cstrsize);
#endif
          if(cstrsize%2 > 0)
            cstrsize++;
          if(fwrite(cstrip, sizeof(char), cstrsize, ofp) != cstrsize)
            goto werror4;
        }
      else
        if(fwrite(strip, bytes, offset, ofp) != offset)
          goto werror4;
#else
      if(fwrite(strip, bytes, offset, ofp) != offset)
        goto werror4;
#endif
    }

  if(image)
    free((char *) image);
#ifdef TIFF_COMP
  if(ITiffDirOffsetShort(expdir[4]) != NO_COMPRESS)
```

```
    {
      if(ITiffDirOffsetShort(expdir[4]) == LZW_COMPRESS)
        TiffCompressLZW(NULL, 0, 0, NULL, &cstrsize, LZW_Cleanup);
    }

  if(ITiffDirLength(expdir[7]) > 1)
    {
#if 0
      printf("Offsets offset = %d\n", ITiffDirOffset(expdir[7]));
#endif
      fseek(ofp, ITiffDirOffset(expdir[7]), SEEK_SET);
      if(fwrite((void *) coffsets, sizeof(unsigned int),
                ITiffDirLength(expdir[7]), ofp) !=
         ITiffDirLength(expdir[7]))
        goto werror4;
    }
  else
      ITiffDirOffset(expdir[7]) = coffsets[0];

  if(ITiffDirLength(expdir[10]) > 1)
    {
      fseek(ofp, ITiffDirOffset(expdir[10]), SEEK_SET);
      if(fwrite((void *) csizes, sizeof(unsigned int),
                ITiffDirLength(expdir[10]), ofp) !=
         ITiffDirLength(expdir[10]))
        goto werror4;
    }
  else
      ITiffDirOffset(expdir[10]) = csizes[0];
#endif

  for(i=0; i<pindsize; i++)
    {
      if(color_type == PALETTE_COLOR)
        {
          offset = ITiffDirOffset(expdir[17]) + (pixind[i]*sizeof(short));
          red = (unsigned short) ((unsigned char *) rgb[i])[0];
          green = (unsigned short) ((unsigned char *) rgb[i])[1];
          blue = (unsigned short) ((unsigned char *) rgb[i])[2];
          red = red * ((1 << shift)+1);
          green = green * ((1 << shift)+1);
          blue = blue * ((1 << shift)+1);
          fseek(ofp, offset, SEEK_SET);
          if(fwrite((void *) &red, sizeof(char), sizeof(unsigned short), ofp)
             != sizeof(unsigned short))
            goto werror4;
          offset += ((1<<shift)*sizeof(short));
          fseek(ofp, offset, SEEK_SET);
          if(fwrite((void *) &green, sizeof(char), sizeof(unsigned short), ofp)
             != sizeof(unsigned short))
            goto werror4;
          offset += ((1<<shift)*sizeof(short));
          fseek(ofp, offset, SEEK_SET);
          if(fwrite((void *) &blue, sizeof(char), sizeof(unsigned short), ofp)
             != sizeof(unsigned short))
            goto werror4;
        }
      free((char *) rgb[i]);
    }
  free((char *) rgb);
#ifdef TIFF_COMP
  free((char *) coffsets);
  free((char *) csizes);
  free((char *) cstrip);
#endif

  free((char *) pixind);
  free((char *) strip);
```

```
  fseek(ofp, diroff, SEEK_SET);

  size = FIELDNUM;
  if(color_type != PALETTE_COLOR)
    size--;

  for(i=0; i<size; i++)
    if(fwrite((void *) expdir[i], sizeof(char), sizeof(*expdir[i]), ofp) !=
       sizeof(*expdir[i]))
      goto werror4;

  for(i=0; i<size; i++)
    free((char *) expdir[i]);
  free((char *) expdir);

  return(STATUS_OK);

 werror4:
  for(i=0; i<pindsize; i++)
    if(rgb[i])
      free((char *) rgb[i]);
  if(rgb)
    free((char *) rgb);
#ifdef TIFF_COMP
  if(ITiffDirOffsetShort(expdir[4]) != NO_COMPRESS)
    {
      if(ITiffDirOffsetShort(expdir[4]) == LZW_COMPRESS)
        TiffCompressLZW(NULL, 0, 0, NULL, &cstrsize, LZW_Cleanup);
    }
#endif

 werror3:
#ifdef TIFF_COMP
  if(coffsets)
    free((char *) coffsets);
  if(csizes)
    free((char *) csizes);
  if(cstrip)
    free((char *) cstrip);
#endif
  if(pixind)
    free((char *) pixind);
  free((char *) strip);

 werror2:
  if(image)
    free((char *) image);

 werror1:
  size = FIELDNUM;
  if(color_type != PALETTE_COLOR)
    size--;
  for(i=0; i<size; i++)
    free((char *) expdir[i]);
  free((char *) expdir);
  fprintf(stderr, "Error encountered writing to file - WriteTIFFMatrix.\n");
  return(STATUS_NOK);

}

#ifdef TIFF_COMP
/***************************** MODULE INFORMATION ************************
 * NAME OF MODULE : TiffCompressLZW
 * DESCRIPTION    :
 *************************************************************************/
static void TiffCompressLZW(void *old, int size, int bytes, void *new,
                            int *newsize, int flag)
{
  static iTiffHash *hashtab = NULL;
```

```
  static int hashsize = 0;
  int i, j, chars, bsize, shift, bits;
  unsigned int hashval;
  iTiffHash val, nextval;
  unsigned char *ptr, *nptr;
  iLZWValueUnion lzwun;

  if(flag == LZW_Initialize)
    {
      if(hashtab)
        {
          fprintf(stderr, "Tiff hash table is not NULL during initialization -
TiffCompressLZW\n");
          (*newsize) = STATUS_NOK;
          return;
        }
      if((hashtab = (iTiffHash *) calloc(HASH_SIZE, sizeof(iTiffHash)))
         == NULL)
        {
          MEMERR("TiffCompressLZW");
          (*newsize) = STATUS_NOK;
          return;
        }
      hashsize = 0;
      (*newsize) = STATUS_OK;
      return;
    }
  else if(flag == LZW_Cleanup)
    {
      if(!hashtab)
        {
          fprintf(stderr, "Tiff hash table is NULL during cleanup -
TiffCompressLZW\n");
          (*newsize) = STATUS_OK;
          return;
        }
      for(i=0; i<HASH_SIZE; i++)
        {
          if((val = hashtab[i]))
            {
              do {
                nextval = ITiffHashNext(val);
                free((char *) val);
                val = nextval;
              } while(val);
            }
        }
      free((char *) hashtab);
      hashtab = NULL;
      hashsize = 0;
    }
  else
    {
      if(!old)
        {
          fprintf(stderr, "Input buffer passed is NULL - TiffCompressLZW\n");
          (*newsize) = STATUS_NOK;
          return;
        }
      if(!size)
        {
          fprintf(stderr, "Input buffer size passed is 0 - TiffCompressLZW\n");
          (*newsize) = STATUS_NOK;
          return;
        }
      if(!new)
        {
          fprintf(stderr, "Output buffer passed is NULL - TiffCompressLZW\n");
          (*newsize) = STATUS_NOK;
```

```
          return;
        }
      bsize = bytes*size;
      ptr = (unsigned char *) old;
#if 0
      for(i=0; i<bsize; i++)
        {
          if(i%16 == 0)
            printf("\n");
          printf("%02x ", ptr[i]);
        }
      printf("\n");
#endif
      memset(new, 0, bsize);
      bits = 9;
      shift = 23;
      nptr = (unsigned char *) new;
      ILZWValueInt(lzwun) = LZWClear;
#if 0
      printf("%5d    ", ILZWValueInt(lzwun));
#endif
      ILZWValueInt(lzwun) <<= shift;
#if 0
      printf("%02x%02x%02x%02x    0x%08x\n", ILZWValueByte(lzwun)[0],
             ILZWValueByte(lzwun)[1], ILZWValueByte(lzwun)[2],
             ILZWValueByte(lzwun)[3], nptr);
#endif
      *nptr |= ILZWValueByte(lzwun)[0];
      *(++nptr) |= ILZWValueByte(lzwun)[1];
      *(nptr+1) |= ILZWValueByte(lzwun)[2];
      shift--;
      ptr = (unsigned char *) old;
      ILZWValueInt(lzwun) = (unsigned int) (*ptr);
      chars = 2;

      for(i=1; i<bsize; i++)
        {
          for(hashval=0,j=0; j<chars; j++)
            hashval = (ptr[j] + (31 * (hashval + j)));
          hashval = hashval % HASH_SIZE;
          if((val = hashtab[hashval]))
            {
              do {
                if(chars == ITiffHashSize(val))
                  if(!memcmp(ptr,(void *) ITiffHashChars(val), chars))
                    break;
                val = ITiffHashNext(val);
              } while(val);
              if(val)
                {
                  ILZWValueInt(lzwun) = ITiffHashIndex(val);
                  chars++;
                }
            }
          if(!val)
            {
#if 0
              printf("%5d    ", ILZWValueInt(lzwun));
#endif
              ILZWValueInt(lzwun) <<= shift;
#if 0
              printf("%02x%02x%02x%02x    0x%08x\n", ILZWValueByte(lzwun)[0],
                     ILZWValueByte(lzwun)[1], ILZWValueByte(lzwun)[2],
                     ILZWValueByte(lzwun)[3], nptr);
#endif
              *nptr |= ILZWValueByte(lzwun)[0];
              *(++nptr) |= ILZWValueByte(lzwun)[1];
              *(nptr+1) |= ILZWValueByte(lzwun)[2];
              shift -= (bits - 8);
```

```
            if((j = (sizeof(int)<<3) - bits - shift) >= 8)
              {
                shift = (sizeof(int)<<3) - bits - (j - 8);
                nptr++;
              }
            if((val = (iTiffHash) calloc(1, sizeof(iTiffHashStruct)))
               == NULL)
              {
                MEMERR("TiffCompressLZW");
                (*newsize) = STATUS_NOK;
                return;
              }
            hashsize++;
            if((hashsize+LZWEOI+1) == (1<<bits))
              {
                bits++;
                shift--;
              }
            if(bits > 12)
              fprintf(stderr, "hash table overflow");
            ITiffHashNext(val) = hashtab[hashval];
            ITiffHashIndex(val) = LZWEOI + hashsize;
            ITiffHashChars(val) = ptr;
            ITiffHashSize(val) = chars;
            hashtab[hashval] = val;
            ptr = &(ptr[chars-1]);
            ILZWValueInt(lzwun) = (unsigned int) *ptr;
            chars = 2;
          }
      }

    ILZWValueInt(lzwun) <<= shift;
    *nptr |= ILZWValueByte(lzwun)[0];
    *(++nptr) |= ILZWValueByte(lzwun)[1];
    *(nptr+1) |= ILZWValueByte(lzwun)[2];
    shift -= (bits - 8);
    if((j = (sizeof(int)<<3) - bits - shift) >= 8)
      {
        shift = (sizeof(int)<<3) - bits - (j - 8);
        nptr++;
      }
    ILZWValueInt(lzwun) = LZWEOI;
    ILZWValueInt(lzwun) <<= shift;
    *nptr |= ILZWValueByte(lzwun)[0];
    *(++nptr) |= ILZWValueByte(lzwun)[1];
    *(nptr+1) |= ILZWValueByte(lzwun)[2];
    (*newsize) = (int) (((unsigned int) nptr) - ((unsigned int) new) + 1);
    if(shift < 16)
      (*newsize)++;

    for(i=0; i<HASH_SIZE; i++)
      {
#if 0
        printf("Hash index = %3d    ", i);
#endif
        if((val = hashtab[i]))
          {
            do {
              nextval = ITiffHashNext(val);
#if 0
              printf("Value = ");
              for(j=0; j<ITiffHashSize(val); j++)
                printf("%02x", ITiffHashChars(val)[j]);
              printf("(%3d)  ", ITiffHashIndex(val));
#endif
              free((char *) val);
              val = nextval;
            } while(val);
          }
```

```
            hashtab[i] = NULL;
#if 0
            printf("\n");
#endif
          }
        hashsize = 0;
      }

}
#endif
```

## H.1.4 crosses.c

This program recurses through a range of values for the controller parameters sampling rate, damping ratio, natural frequency, and feedback gain, determines whether the system is stable, determines the theoretical closed loop frequency response, determines the system bandwidth and maximum closed loop gain, and increments a counter pertaining to which frequency decade the bandwidth and maximum gain lie in. The totals are accumulated for each sampling rate and then written to a text file.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <values.h>

#ifdef NMALLOC
#include "nmalloc.h"
#endif

#define RADIX 2.0
#define NR_END 1
#define MAXM 8

#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))

#define MYERR       1e-8

void DigStabFunc(double A1, double A2, double P, double Q, double newt,
                 double a22, double a21, double *J1, double *J2, double *J3,
                 double *J4, double *J5, double *M1, double *M2, double *M3,
                 double *M4, double *N1, double *N2, double *N3, double *N4);
int DigBode(double *num, double *den, int numsize, int densize, double ts,
            double begfreq, double endfreq, double **mag, double **freq,
            int *magsize);

int zrhqr(double a[], int m, double rtr[], double rti[]);
void balanc(double **a, int n);
int hqr(double **a, int n, double wr[], double wi[]);
double **matrix(long nrl, long nrh, long ncl, long nch);
void free_matrix(double **m, long nrl, long nrh, long ncl, long nch);

void main(int argc, char **argv)
{
  int i, j, k, l, m, ind, begin;
  int ebara, axis, total[4], good[4], magsize;
  int defout, sizet, sizeb, sized, sizef;
  double maxb, minb, maxd, mind, maxf, minf;
  char buffer[200], *limfile, *outfile;
  FILE *ifp, *ofp;
  double P, Q, A1, A2;
  double bhat, freq, damp, newt;
```

```
  double a21, a22;
  double subtot[9], rroots[9], iroots[9];
  double J1, J2, J3, J4, J5, M1, M2, M3, M4, N1, N2, N3, N4;
  double *t, num[8], den[8];
  double *mag, *frq, max;
#ifdef DETAILS
  int details[36], n;
  double test[4][9] = { {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 },
                        {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 },
                        {10.0, 20.0, 30.0, 40.0, 50.0,
                         60.0, 70.0, 80.0, 90.0 },
                        {100.0, 200.0, 300.0, 400.0, 500.0,
                         600.0, 700.0, 800.0, 900.0 } };
  char *buf;
  FILE *ofp2;
#endif

  ebara = defout = 0;
  begin = axis = 0;
  limfile = outfile = NULL;

  if(argc < 2 || argc > 7)
    {
      fprintf(stderr, "Syntax error\n");
      fprintf(stderr, "Syntax:  crosses limits_file [outfile] [-a n] [-b m]\n");
      fprintf(stderr, "         where:  limits_file - filename of range data\n");
      fprintf(stderr, "                 outfile     - output filename\n");
      fprintf(stderr, "                 -a n -> bearing axis [0-4]\n");
      fprintf(stderr, "                 -b m -> begin time [0-49]\n");
      exit(1);
    }

  for(i=1; i<argc; i++)
    {
      if(argv[i][0] == '-')
        {
          if(argv[i][1] == 'a')
            axis = atoi(argv[++i]);
          else if(argv[i][1] == 'b')
            begin = atoi(argv[++i]);
        }
      else if(limfile == NULL)
        limfile = argv[i];
      else if(outfile == NULL)
        outfile = argv[i];
    }

  if((ifp = fopen(limfile, "r")) == NULL)
    {
      fprintf(stderr, "Unable to open limits file \"%s\" (r)\n", limfile);
      exit(1);
    }

  if(outfile == NULL)
    {
      ofp = stdout;
      defout = 1;
    }
  else if((ofp = fopen(outfile, "w")) == NULL)
    {
      fprintf(stderr, "Unable to open output file \"%s\" (w)\n", outfile);
      fclose(ifp);
      exit(1);
    }

#ifdef DETAILS
  if(!defout)
    {
      if((buf = (char *) calloc(strlen(outfile)+10, sizeof(char))) == NULL)
```

```
        {
          fprintf(stderr, "Out of Memory Error (%d) - crosses\n", __LINE__);
          fclose(ifp);
          if(!defout)
            fclose(ofp);
          exit(1);
        }
      strcpy(buf, outfile);
      strcat(buf,".details");
      if((ofp2 = fopen(buf, "w")) == NULL)
        {
          fprintf(stderr, "Unable to open output file \"%s\" (w)\n", buf);
          fclose(ifp);
          if(!defout)
            fclose(ofp);
          exit(1);
        }
      free(buf);
    }
#endif

  sizet = 0;
  maxb = maxd = maxf = 0.0;
  minb = mind = minf = 1000.0;
  t = NULL;
  while(fgets(buffer, 200, ifp))
    {
      if(sscanf(buffer, "%lf %lf %lf %lf %lf %lf %lf\n", &subtot[0],
                &subtot[1], &subtot[2], &subtot[3], &subtot[4], &subtot[5],
                &subtot[6])
         != 7)
        {
          fprintf(stderr, "Error encountered reading limit file (sscanf)\n");
          fclose(ifp);
          if(!defout)
            {
              fclose(ofp);
#ifdef DETAILS
              fclose(ofp2);
#endif
            }
          exit(1);
        }
      sizet++;
      if((t = (double *) realloc(t, sizet*7*sizeof(double))) == NULL)
        {
          fprintf(stderr, "Out of Memory Error (%d) - crosses\n", __LINE__);
          fclose(ifp);
          if(!defout)
            {
              fclose(ofp);
#ifdef DETAILS
              fclose(ofp2);
#endif
            }
          exit(1);
        }
      for(i=0; i<7; i++)
        t[((sizet-1)*7)+i] = subtot[i];
      if(mind > subtot[1])
        mind = subtot[1];
      if(maxd < subtot[2])
        maxd = subtot[2];
      if(minb > subtot[3])
        minb = subtot[3];
      if(maxb < subtot[4])
        maxb = subtot[4];
      if(minf > subtot[5])
        minf = subtot[5];
```

```
    if(maxf < subtot[6])
      maxf = subtot[6];
  }

for(i=0; i<7; i++)
  subtot[i] = 0.0;

fclose(ifp);

sizeb = ((int) (((maxb - minb) / 10.0) + 0.5)) + 1;
sized = ((int) (((maxd - mind) / 0.1) + 0.5)) + 1;
sizef = ((int) (((maxf - minf) / 10.0) + 0.5)) + 1;

if(ebara)
  {
    switch(axis)
      {
      default:
      case 0:
        P = 11.081;
        Q = 27701.879;
        A1 = 1666.622;
        A2 = 1700.000;
        break;
      case 1:
        P = 10.278;
        Q = 41113.653;
        A1 = 14222.428;
        A2 = 4370.000;
        break;
      case 2:
        P = 10.278;
        Q = 41113.653;
        A1 = 14575.775;
        A2 = 4030.000;
        break;
      case 3:
        P = 20.263;
        Q = 81050.754;
        A1 = 4027.457;
        A2 = 2450.000;
        break;
      case 4:
        P = 20.263;
        Q = 81050.754;
        A1 = 4108.477;
        A2 = 2600.000;
        break;
      }
  }
else
  {
    switch(axis)
      {
      default:
      case 0:
        P = 7.990;
        Q = 22739.568;
        A1 = 13324.943;
        A2 = 13310.0;
        break;
      case 1:
        P = 7.123;
        Q = 7737.770;
        A1 = 13105.209;
        A2 = 13130.0;
        break;
      case 2:
        P = 8.296;
```

```
          Q = 8882.644;
          A1 = 13043.913;
          A2 = 13080.0;
          break;
        case 3:
          P = 16.926;
          Q = 35530.574;
          A1 = 11112.759;
          A2 = 11140.0;
          break;
        case 4:
          P = 15.113;
          Q = 33201.348;
          A1 = 12273.613;
          A2 = 12290.0;
          break;
        }
    }

  subtot[8] = 1.0;

  for(i=begin; i<sizet; i++)    /* time loop */
    {
#ifdef DETAILS
      memset((char *) &details[0], 0, 36*sizeof(int));
#endif
      total[0] = total[1] = total[2] = total[3] = 0;
      good[0] = good[1] = good[2] = good[3] = 0;
      newt = t[(i*7)];
      for(j=0; j<sizeb; j++)    /* bhat loop */
        {
          bhat = minb + (10.0 * j);
          if(bhat < t[(i*7)+3] || bhat > t[(i*7)+4])
            continue;
          for(k=0; k<sizef; k++)    /* freq loop */
            {
              freq = minf + (k * 10.0);
              if(freq < t[(i*7)+5] || freq > t[(i*7)+6])
                continue;
              a21 = freq * freq;
              for(l=0; l<sized; l++)    /* damp loop */
                {
                  damp = mind + (l * 0.1);
                  if(damp < t[(i*7)+1] || damp > t[(i*7)+2])
                    continue;
                  a22 = 2.0 * damp * freq;
                  DigStabFunc(A1, A2, P, Q, newt, a22, a21, &J1, &J2, &J3,
                              &J4, &J5, &M1, &M2, &M3, &M4, &N1, &N2, &N3,
                              &N4);
                  subtot[7] = ((J1*M2)/bhat)+(N2-1.0);
                  subtot[6] = (((J1*M3)+(J2*M2))/bhat)+(N3-N2);
                  subtot[5] = (((J1*M4)+(J2*M3)+(J3*M2))/bhat)+(N4-N3);
                  subtot[4] = (((J2*M4)+(J3*M3)+(J4*M2))/bhat)-N4;
                  subtot[3] = ((J3*M4)+(J4*M3)+(J5*M2))/bhat;
                  subtot[2] = ((J4*M4)+(J5*M3))/bhat;
                  subtot[1] = (J5*M4)/bhat;
                  if(zrhqr(subtot, 8, rroots, iroots))
                    {
                      fprintf(stderr, "Eigenvalue calculation problem ");
                      fprintf(stderr, "(i = %d  j = %d  k = %d  l = %d)\n",
                              i, j, k, l);
                      for(m=1; m<=8; m++)
                        {
                          iroots[m] = 0.0;
                          rroots[m] = 2.0;
                        }
                    }
                  for(m=1; m<=8; m++)
                    if((rroots[m]*rroots[m])+(iroots[m]*iroots[m]) > 1.0)
```

```
                    break;
                if(m == 9)
                  {
                    num[0] = 1.0;
                    num[1] = J1*M2;
                    num[2] = (J1*M3)+(J2*M2);
                    num[3] = (J1*M4)+(J2*M3)+(J3*M2);
                    num[4] = (J2*M4)+(J3*M3)+(J4*M2);
                    num[5] = (J3*M4)+(J4*M3)+(J5*M2);
                    num[6] = (J4*M4)+(J5*M3);
                    num[7] = (J5*M4);
                    den[0] = bhat;
                    den[1] = (bhat*(N2-1.0))+(J1*M2);
                    den[2] = (bhat*(N3-N2))+(J2*M2)+(J1*M3);
                    den[3] = (bhat*(N4-N3))+(J3*M2)+(J2*M3)+(J1*M4);
                    den[4] = (bhat*(-N4))+(J4*M2)+(J3*M3)+(J2*M4);
                    den[5] = (J5*M2)+(J4*M3)+(J3*M4);
                    den[6] = (J5*M3)+(J4*M4);
                    den[7] = (J5*M4);

                    if(DigBode(num, den, 8, 8, newt, 0.1, 1000.0, &mag,
                               &frq, &magsize))
                      {
                        fprintf(stderr, "Bode calculation problem ");
                        fprintf(stderr, "(i = %d  j = %d  k = %d  l = %d)\n",
                                i, j, k, l);
                      }
                    else
                      {
                        for(max=0.0,ind=m=0; m<magsize; m++)
                          if(mag[m] > max)
                            {
                              max = mag[m];
                              ind = m;
                            }
                        for(m=ind; m<magsize; m++)
                          if(mag[m] < -3.0)
                            {
                              ind = m-1;
                              break;
                            }
                        if(frq[ind] < 1.0)
                          m = 0;
                        else if(frq[ind] < 10.0)
                          m = 1;
                        else if(frq[ind] < 100.0)
                          m = 2;
                        else if(frq[ind] < 1000.0)
                          m = 3;
                        else
                          m = 4;
                        if(m < 4)
                          {
                            if(max < 10.0)
                              good[m]++;
                            total[m]++;
                            for(n=0; n<9; n++)
                              if(test[m][n] > frq[ind])
                                break;
                            details[(m*9)+(n-1)]++;
                          }
                        free(mag);
                        free(frq);
                      }
                  }
              }
          }
#ifdef DEBUG
        printf("t = %7.1f, b = %6.1f, d = %3.1f, f = %6.1f\n", newt, bhat, damp,
```

```
freq);
#endif
        }
      fprintf(ofp, "%7.1f %d (%d) %d (%d) %d (%d) %d (%d)\n", newt, good[0],
              total[0], good[1], total[1], good[2], total[2], good[3],
              total[3]);
      fflush(ofp);
      if(!defout)
        {
#ifdef DETAILS
          for(j=0; j<36; j++)
            {
              if(!j)
                fprintf(ofp2, "%7.1f", newt);
              fprintf(ofp2, " %d", details[j]);
            }
          fprintf(ofp2, "\n");
          fflush(ofp2);
#endif
          printf("%3d of %3d (%d, %d, %d, %d)\n", i+1, sizet, total[0],
                 total[1], total[2], total[3]);
        }
    }

  if(!defout)
    {
      fclose(ofp);
#ifdef DETAILS
      fclose(ofp2);
#endif
    }
  exit(0);
}


void DigStabFunc(double A1, double A2, double P, double Q, double T,
                 double a22, double a21, double *J1, double *J2, double *J3,
                 double *J4, double *J5, double *M1, double *M2, double *M3,
                 double *M4, double *N1, double *N2, double *N3, double *N4)
{
  double K1, K2, K3, K4, L1, L2, L3, L4, T1, T2, sq;

  T1 = 1/T;
  T2 = T1*T1;
  sq = sqrt(Q);

#ifdef BACKWARD
  /* central difference acceleration and backward difference velocity */
  *J1 = (3.0/(4.0*(T2)))+((3.0*a22)/(2.0*T1))+(a21);
  *J2 = (-4.0/(4.0*(T2)))-((4.0*a22)/(2.0*T1));
  *J3 = (-2.0/(4.0*(T2)))+(a22/(2.0*T1));
  *J4 = 1.0/(T2);
  *J5 = -1.0/(4.0*(T2));
#else
  /* central difference acceleration and  velocity */
  *J1 = (1.0/(4.0*(T2)))+(a22/(2.0*T1))+(a21);
  *J2 = 0.0;
  *J3 = (-2.0/(4.0*(T2)))-(a22/(2.0*T1));
  *J4 = 0.0;
  *J5 = 1.0/(4.0*(T2));
#endif

  K1 = (-A1*P)/(A2*Q);
  K2 = (-A1*P)/(A2*((A2*A2)-Q));
  K3 = (A1*P)/(2.0*Q*(A2-sq));
  K4 = (A1*P)/(2.0*Q*(A2+sq));

  L1 = exp(-A2*T1);
  L2 = exp(-sq*T1);
```

```
    L3 = exp(sq*T1);
    L4 = 1;

    *M1 = 0;
    *M2 = (-K1*(L1+L2+L3))-(K2*(L2+L3+L4))-(K3*(L1+L3+L4))-(K4*(L1+L2+L4));
    *M3 = (K1*((L1*L2)+(L3*(L1+L2))))+(K2*((L2*L4)+(L3*(L2+L4))))+
      (K3*((L1*L4)+(L3*(L1+L4))))+(K4*((L1*L4)+(L2*(L1+L4))));
    *M4 = (-K1*L1*L2*L3)-(K2*L2*L3*L4)-(K3*L1*L3*L4)-(K4*L1*L2*L4);

    *N1 = 1.0;
    *N2 = -L1-L2-L3;
    *N3 = (L1*L2)+(L1*L3)+(L2*L3);
    *N4 = -(L1*L2*L3);

}

int zrhqr(double a[], int m, double rtr[], double rti[])
{
  int j, k;
  double **hess, xr, xi;

  hess = matrix(1,MAXM,1,MAXM);
  if(m > MAXM || a[m] == 0.0)
    {
      fprintf(stderr, "bad args in zrhqr\n");
      free_matrix(hess,1,MAXM,1,MAXM);
      return(1);
    }
  for(k=1;k<=m;k++)
    {
      hess[1][k] = -a[m-k]/a[m];
      for(j=2;j<=m;j++)
        hess[j][k] = 0.0;
      if(k != m)
        hess[k+1][k] = 1.0;
    }
  balanc(hess, m);
  if(hqr(hess,m,rtr,rti))
    {
      free_matrix(hess,1,MAXM,1,MAXM);
      return(1);
    }
  for(j=2; j<=m; j++)
    {
      xr = rtr[j];
      xi = rti[j];
      for(k=j-1; k>=1; k--)
        {
          if(rtr[k] <= xr)
            break;
          rtr[k+1] = rtr[k];
          rti[k+1] = rti[k];
        }
      rtr[k+1] = xr;
      rti[k+1] = xi;
    }
  free_matrix(hess,1,MAXM,1,MAXM);
  return(0);
}


void balanc(double **a, int n)
{
  int last,j,i;
  double s,r,g,f,c,sqrdx;

  sqrdx = RADIX*RADIX;
  last = 0;
  while(last == 0)
```

```
    {
      last = 1;
      for(i=1; i<=n; i++)
        {
          r = c = 0.0;
          for(j=1; j<=n; j++)
            if(j != i)
              {
                c += fabs(a[j][i]);
                r += fabs(a[i][j]);
              }
          if(c && r)
            {
              g = r/RADIX;
              f = 1.0;
              s = c+r;
              while(c<g)
                {
                  f *= RADIX;
                  c *= sqrdx;
                }
              g = r*RADIX;
              while(c > g)
                {
                  f /= RADIX;
                  c /= sqrdx;
                }
              if((c+r)/f < 0.95*s)
                {
                  last = 0;
                  g = 1.0/f;
                  for(j=1; j<=n; j++)
                    a[i][j] *= g;
                  for(j=1; j<=n; j++)
                    a[j][i] *= f;
                }
            }
        }
    }
}


int hqr(double **a, int n, double wr[], double wi[])
{
  int nn, m, l, k, j, its, i, mmin;
  double z, y, x, w, v, u, t, s, r, q, p, anorm;

  anorm = fabs(a[1][1]);
  for(i=2; i<=n; i++)
    for(j=(i-1); j<=n; j++)
      anorm += fabs(a[i][j]);
  nn=n;
  t=0.0;
  while(nn >= 1)
    {
      its = 0;
      do {
        for(l=nn; l>=2; l--)
          {
            s = fabs(a[l-1][l-1])+fabs(a[l][l]);
            if(s == 0.0)
              s=anorm;
            if((double)(fabs(a[l][l-1]) + s) == s)
              break;
          }
        x = a[nn][nn];
        if(l == nn)
          {
            wr[nn] = x+t;
```

```
    wi[nn--] = 0.0;
  }
else
  {
    y = a[nn-1][nn-1];
    w = a[nn][nn-1]*a[nn-1][nn];
    if(l == (nn-1))
      {
        p = 0.5*(y-x);
        q = p*p+w;
        z = sqrt(fabs(q));
        x += t;
        if(q >= 0.0)
          {
            z = p+SIGN(z,p);
            wr[nn-1] = wr[nn] = x+z;
            if(z)
              wr[nn] = x-w/z;
            wi[nn-1] = wi[nn] = 0.0;
          }
        else
          {
            wr[nn-1] = wr[nn] = x+p;
            wi[nn-1] = -(wi[nn] = z);
          }
        nn -= 2;
      }
    else
      {
        if(its == 30)
          {
            fprintf(stderr, "Too many iterations in hqr\n");
            return(1);
          }
        if(its == 10 || its == 20)
          {
            t += x;
            for(i=1; i<=nn; i++)
              a[i][i] -= x;
            s = fabs(a[nn][nn-1])+fabs(a[nn-1][nn-2]);
            y = x = 0.75*s;
            w = -0.4375*s*s;
          }
        ++its;
        for(m=(nn-2); m>=l; m--)
          {
            z = a[m][m];
            r = x-z;
            s = y-z;
            p = (r*s-w)/a[m+1][m]+a[m][m+1];
            q = a[m+1][m+1]-z-r-s;
            r = a[m+2][m+1];
            s = fabs(p)+fabs(q)+fabs(r);
            p /= s;
            q /= s;
            r /= s;
            if(m == l)
              break;
            u = fabs(a[m][m-1])*(fabs(q)+fabs(r));
            v = fabs(p)*(fabs(a[m-1][m-1])+fabs(z)+fabs(a[m+1][m+1]));
            if((double)(u+v) == v)
              break;
          }
        for(i=m+2; i<=nn; i++)
          {
            a[i][i-2] = 0.0;
            if(i != (m+2))
              a[i][i-3] = 0.0;
          }
```

```
                for(k=m; k<=nn-1; k++)
                  {
                    if(k != m)
                      {
                        p = a[k][k-1];
                        q = a[k+1][k-1];
                        r = 0.0;
                        if(k != (nn-1))
                          r = a[k+2][k-1];
                        if((x = fabs(p)+fabs(q)+fabs(r)) != 0.0)
                          {
                            p /= x;
                            q /= x;
                            r /= x;
                          }
                      }
                    if((s = SIGN(sqrt(p*p+q*q+r*r),p)) != 0.0)
                      {
                        if(k == m)
                          {
                            if(l != m)
                              a[k][k-1] = -a[k][k-1];
                          }
                        else
                          a[k][k-1] = -s*x;
                        p += s;
                        x = p/s;
                        y = q/s;
                        z = r/s;
                        q /= p;
                        r /= p;
                        for(j=k; j<=nn; j++)
                          {
                            p = a[k][j]+q*a[k+1][j];
                            if(k != (nn-1))
                              {
                                p += r*a[k+2][j];
                                a[k+2][j] -= p*z;
                              }
                            a[k+1][j] -= p*y;
                            a[k][j] -= p*x;
                          }
                        mmin = nn<k+3 ? nn : k+3;
                        for(i=l; i<=mmin; i++)
                          {
                            p = x*a[i][k]+y*a[i][k+1];
                            if(k != (nn-1))
                              {
                                p += z*a[i][k+2];
                                a[i][k+2] -= p*r;
                              }
                            a[i][k+1] -= p*q;
                            a[i][k] -= p;
                          }
                      }
                  }
              }
          }
      } while(l < nn-1);
    }
  return(0);
}

double **matrix(long nrl, long nrh, long ncl, long nch)
{
  long i, nrow = nrh-nrl+1, ncol = nch-ncl+1;
  double **m;

  /* allocate pointers to rows */
```

```
  if((m = (double **) malloc((size_t) ((nrow+NR_END) * sizeof(double *))))
     == NULL)
    {
      fprintf(stderr, "allocation failure 1 in matrix()\n");
      exit(1);
    }
  m += NR_END;
  m -= nrl;

  /* allocate rows and set pointers to them */
  if((m[nrl] = (double *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(double))))
     == NULL)
    {
      fprintf(stderr, "allocation failure 2 in matrix()\n");
      exit(1);
    }
  m[nrl] += NR_END;
  m[nrl] -= ncl;

  for(i=nrl+1; i<=nrh; i++)
    m[i]=m[i-1]+ncol;

  return(m);

}


void free_matrix(double **m, long nrl, long nrh, long ncl, long nch)
{
  free((char *) (m[nrl]+ncl-NR_END));
  free((char *) (m+nrl-NR_END));
}


int DigBode(double *num, double *den, int numsize, int densize, double ts,
            double begfreq, double endfreq, double **mag, double **freq,
            int *magsize)
{
  int i, j, sampsize, dec;
  double *mg, *fq, inc, t;
  double frq, ansr, ansi;

  t = 1.0/ts;
  dec = (int) (log10(endfreq) - log10(begfreq));
  sampsize = dec * 81;
  sampsize += 2;

  if((mg = (double *) calloc(sampsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigBode.\n");
      return(1);
    }

  if((fq = (double *) calloc(sampsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigBode.\n");
      free(mg);
      return(1);
    }

  inc = begfreq / 10.0;
  fq[0] = begfreq;
  for(i=0; i<dec; i++)
    {
      for(j=0; j<81; j++)
        fq[(i*81)+j+1] = fq[(i*81)+j] + inc;
      inc *= 10.0;
    }
  fq[sampsize-1] = endfreq;
```

```
  for(i=0; i<sampsize; i++)
    {
      frq = 2.0 * M_PI * fq[i];
      ansr = ansi = 0.0;
      mg[i] = 20.0 * log10(num[0]);
      for(j=1; j<numsize; j++)
        {
          if(num[j] != 0.0)
            {
              inc = frq * t * (numsize - 1 - j);
              ansr += (num[j] * cos(inc));
              ansi += (num[j] * sin(inc));
            }
        }
      mg[i] += (20.0 * log10(sqrt(pow(ansr,2.0) + pow(ansi,2.0))));
      ansr = ansi = 0.0;
      for(j=0; j<densize; j++)
        {
          if(den[j] != 0.0)
            {
              inc = frq * t * (densize - 1 - j);
              ansr += (den[j] * cos(inc));
              ansi += (den[j] * sin(inc));
            }
        }
      mg[i] -= (20.0 * log10(sqrt(pow(ansr,2.0) + pow(ansi,2.0))));
    }

  if(mag)
    (*mag) = mg;
  if(freq)
    (*freq) = fq;
  if(magsize)
    (*magsize) = sampsize;

  return(0);

}
```

## H.1.5 getsubset.c

This program reads each line of the file produced by mat2text.c and prints all lines that match a specific value in a specific column. The column and value are command line parameters to the program. This program was used to create a text file of the stable systems that conformed to a particular sampling rate from the set of all systems which were stable over many sampling rates.

```
#include <stdio.h>
#include <stdlib.h>

#define BUFSIZE    1024


void main(int argc, char **argv)
{
  int i, col, line, once, once_size;
  char *infile, *outfile, *buffer;
  char name[80];
  FILE *ifp, *ofp;
  FILE **ofps;
  float *values;
  float finp[4], value;
```

```
if(argc < 2)
  {
    fprintf(stderr, "Syntax:  getsubset infile [outfile] [-c n] ");
    fprintf(stderr, "[-v value]\n");
    fprintf(stderr, "         where:  infile   - input pathname\n");
    fprintf(stderr, "                 outfile  - output pathname ");
    fprintf(stderr, "(default = stdout)\n");
    fprintf(stderr, "                 -c n     - column interested in ");
    fprintf(stderr, "(default = 0)\n");
    fprintf(stderr, "                 -v value - keep records value ");
    fprintf(stderr, "(default = 0.0)\n");
    fprintf(stderr, "                 -once    - create file for each ");
    fprintf(stderr, "unique value in column\n\n");
    exit(1);
  }

infile = argv[1];
outfile = NULL;
once = col = 0;
value = 0.0;

if(argc > 2)
  {
    for(i=2; i<argc; i++)
      {
        if(!(strcmp(argv[i],"-c")))
          col = atoi(argv[i+1]);
        else if(!(strcmp(argv[i],"-v")))
          value = (float) atof(argv[i+1]);
        else if(!(strcmp(argv[i],"-once")))
          once = 1;
        else if(!outfile)
          outfile = argv[i];
      }
  }

if(col > 3)
  {
    fprintf(stderr, "only four columns expected\n\n");
    exit(1);
  }

if((ifp = fopen(infile, "r")) == NULL)
  {
    fprintf(stderr, "unable to open input file \"%s\"\n\n", infile);
    exit(1);
  }

if(!once && outfile)
  if((ofp = fopen(outfile, "w")) == NULL)
    {
      fclose(ifp);
      fprintf(stderr, "unable to open output file \"%s\"\n\n", outfile);
      exit(1);
    }
else if(once && !outfile)
  outfile = "getsub";

if((buffer = (char *) calloc(BUFSIZE, sizeof(char))) == NULL)
  {
    fprintf(stderr, "Out of memory error - getsubset (%d)\n\n",
            __LINE__);
    fclose(ifp);
    if(outfile)
      fclose(ofp);
    exit(1);
  }

if(once)
```

```
  {
    ofps = NULL;
    values = NULL;
    once_size = 0;
  }
line = 0;

while(fgets(buffer, BUFSIZE, ifp))
  {
    line++;
    sscanf(buffer, "%f %f %f %f", &finp[0], &finp[1], &finp[2], &finp[3]);

    if(once)
      {
        for(i=0; i<once_size; i++)
          if(finp[col] == values[i])
            break;
        if(i == once_size)
          {
            if((ofps = (FILE **) realloc(ofps, (once_size+1)*sizeof(FILE *)))
               == NULL)
              {
                fprintf(stderr, "Out of memory error - getsubset (%d)\n\n",
                        __LINE__);
                fclose(ifp);
                for(i=0; i<once_size; i++)
                  fclose(ofps[i]);
                exit(1);
              }
            if((values =
                (float *) realloc(values, (once_size+1)*sizeof(float)))
               == NULL)
              {
                fprintf(stderr, "Out of memory error - getsubset (%d)\n\n",
                        __LINE__);
                fclose(ifp);
                for(i=0; i<once_size; i++)
                  fclose(ofps[i]);
                exit(1);
              }
            values[once_size] = finp[col];
            sprintf(name, "%s.%d", outfile, (int) finp[col]);
            if((ofps[once_size] = fopen(name, "w")) == NULL)
              {
                fclose(ifp);
                fprintf(stderr, "unable to open output file \"%s\"\n\n",
                        outfile);
                for(i=0; i<once_size; i++)
                  fclose(ofps[i]);
                exit(1);
              }
            fprintf(ofps[once_size], "%s", buffer);
            once_size++;
          }
        else
          fprintf(ofps[i], "%s", buffer);
      }
    else if(finp[col] == value)
      {
        if(outfile)
          fprintf(ofp, "%s", buffer);
        else
          printf("%s", buffer);
      }
    if(line%100000 == 0)
      fprintf(stderr, "Processed %d records ...\n", line);
  }

free(buffer);
```

```
  fclose(ifp);
  if(!once && outfile)
    fclose(ofp);
  if(once)
    {
      for(i=0; i<once_size; i++)
        fclose(ofps[i]);
      free(values);
      free(ofps);
    }
  exit(0);

}
```

## H.1.6 uniqcount.c

This program reads the text file produced by getsubset.c and a text file containing a list of the unique values and the number of times they occur for a given column. The column is specified as a command line parameter. This was used to quickly produce the histogram data for display in Matlab. This could have been handled directly by Matlab but it would have been very slow and memory intensive.

```
#include <stdio.h>


#define BUFSIZE    1024
#define ALLOC_INC   100

float *fsort;

int sortme(void *elem1, void *elem2);

void main(int argc, char **argv)
{
  int i, col, *num, got, uniq;
  int line, *sortind;
  char *infile, *outfile, *buffer;
  FILE *ifp, *ofp;
  float finp[4];
  float *fout;

  if(argc < 2)
    {
      fprintf(stderr, "Syntax:  uniqcount infile [outfile] [-c n]\n");
      fprintf(stderr, "         where:  infile  - input pathname\n");
      fprintf(stderr, "                 outfile - output pathname ");
      fprintf(stderr, "(default = stdout)\n");
      fprintf(stderr, "                 -c n    - column interested in ");
      fprintf(stderr, "(default = 0)\n\n");
      exit(1);
    }

  infile = argv[1];
  outfile = NULL;
  col = 0;

  if(argc > 2)
    {
      for(i=2; i<argc; i++)
        {
          if(!(strcmp(argv[i],"-c")))
            col = atoi(argv[i+1]);
```

```
        else if(!outfile)
          outfile = argv[i];
      }
  }

if(col > 3)
  {
    fprintf(stderr, "only four columns expected\n\n");
    exit(1);
  }

if((ifp = fopen(infile, "r")) == NULL)
  {
    fprintf(stderr, "unable to open input file \"%s\"\n\n", infile);
    exit(1);
  }

if(outfile)
  if((ofp = fopen(outfile, "w")) == NULL)
    {
      fclose(ifp);
      fprintf(stderr, "unable to open output file \"%s\"\n\n", outfile);
      exit(1);
    }

if((buffer = (char *) calloc(BUFSIZE, sizeof(char))) == NULL)
  {
    fprintf(stderr, "Out of memory error - uniqcount (%d)\n\n",
            __LINE__);
    fclose(ifp);
    if(outfile)
      fclose(ofp);
    exit(1);
  }

if((fout = (float *) calloc(ALLOC_INC, sizeof(float))) == NULL)
  {
    fprintf(stderr, "Out of memory error - uniqcount (%d)\n\n",
            __LINE__);
    free(buffer);
    fclose(ifp);
    if(outfile)
      fclose(ofp);
    exit(1);
  }

if((num = (int *) calloc(ALLOC_INC, sizeof(int))) == NULL)
  {
    fprintf(stderr, "Out of memory error - uniqcount (%d)\n\n",
            __LINE__);
    free(buffer);
    free(fout);
    fclose(ifp);
    if(outfile)
      fclose(ofp);
    exit(1);
  }

got = ALLOC_INC;

if(!(fgets(buffer, BUFSIZE, ifp)))
  {
    fprintf(stderr, "Couldn't read first line of input file\n\n");
    free(buffer);
    free(fout);
    free(num);
    fclose(ifp);
    if(outfile)
      fclose(ofp);
```

```
    exit(1);
  }

sscanf(buffer, "%f %f %f %f", &finp[0], &finp[1], &finp[2], &finp[3]);

fout[0] = finp[col];
num[0] = 1;
line = uniq = 1;

while(fgets(buffer, BUFSIZE, ifp))
  {
    line++;
    sscanf(buffer, "%f %f %f %f", &finp[0], &finp[1], &finp[2], &finp[3]);

    for(i=0; i<uniq; i++)
      if(fout[i] == finp[col])
        {
          num[i]++;
          break;
        }

    if(i == uniq)
      {
        if(uniq == got)
          {
            if((fout =
                (float *) realloc(fout, (got+ALLOC_INC)*sizeof(float)))
               == NULL)
              {
                fprintf(stderr, "Out of memory error - uniqcount (%d)\n\n",
                        __LINE__);
                free(buffer);
                free(num);
                fclose(ifp);
                if(outfile)
                  fclose(ofp);
                exit(1);
              }
            if((num =
                (int *) realloc(num, (got+ALLOC_INC)*sizeof(int)))
               == NULL)
              {
                fprintf(stderr, "Out of memory error - uniqcount (%d)\n\n",
                        __LINE__);
                free(buffer);
                free(fout);
                fclose(ifp);
                if(outfile)
                  fclose(ofp);
                exit(1);
              }
            got += ALLOC_INC;
          }
        fout[uniq] = finp[col];
        num[uniq] = 1;
        uniq++;
      }
    if(line%100000 == 0)
      fprintf(stderr, "Processed %d records ...\n", line);
  }

if((sortind = (int *) calloc(got, sizeof(int))) == NULL)
  {
    fprintf(stderr, "Out of memory error - uniqcount (%d)\n\n",
            __LINE__);
    free(buffer);
    free(fout);
    free(num);
    fclose(ifp);
```

```
      if(outfile)
        fclose(ofp);
      exit(1);
    }

  for(i=0; i<uniq; i++)
    sortind[i] = i;

  fsort = fout;

  fprintf(stderr, "Sorting unique (%d) records ...\n", uniq);

  qsort((void *) sortind, uniq, sizeof(int), sortme);

  if(outfile)
    {
      for(i=0; i<uniq; i++)
        fprintf(ofp, "%f  %d\n", fsort[sortind[i]], num[sortind[i]]);
    }
  else
    {
      for(i=0; i<uniq; i++)
        printf("%f  %d\n", fsort[sortind[i]], num[sortind[i]]);
    }

  free(buffer);
  free(fout);
  free(num);
  free(sortind);
  fclose(ifp);
  if(outfile)
    fclose(ofp);
  exit(0);

}


int sortme(void *elem1, void *elem2)
{
  int i = *((int *) elem1);
  int j = *((int *) elem2);

  if(fsort[i] == fsort[j])
    return(0);
  else if(fsort[i] > fsort[j])
    return(1);
  else
    return(-1);
}
```

## H.1.7 settlestats.m

The Matlab script reads the text files produced by getsubset.c for each bearing axis, plots the histograms of the damping ratio, natural frequency, and feedback gain, and calculates the mean and standard deviation of each controller parameter. The mean and standard deviation are used to further restrict the controller parameters range of values in the compli.c program to speed calculations.

```
% print histogram stats from stability matrix after mat2text conversion

fname = str2mat('/usr/tmp/axial.cut', ...
    '/usr/tmp/rad1x.cut', ...
```

```
    '/usr/tmp/rad1y.cut', ...
    '/usr/tmp/rad2x.cut', ...
    '/usr/tmp/rad2y.cut');

fvalname = str2mat('/usr/tmp/axial.val', '/usr/tmp/rad1x.val', ...
    '/usr/tmp/rad1y.val', '/usr/tmp/rad2x.val', '/usr/tmp/rad2y.val');

bearname = str2mat('Axial', 'Rad1X', 'Rad1Y', 'Rad2X', 'Rad2Y');

histname = str2mat(' sampling time delay histogram', ...
    ' damping ratio histogram', ' bhat histogram', ' frequency histogram');

valname = str2mat(' sampling time delay histogram counts', ...
    ' damping ratio histogram counts', ' bhat histogram counts', ...
    ' frequency histogram counts');

colname = str2mat('Sampling Rate', 'Damping Ratio', 'Bhat', 'Frequency');

myprinter = '-Pshiva';

% set print variable to 1 if you want hardcopy results
printme = 0;

% set test variable to 0 if no subset required
test = [1 12000];

for ii=1:size(bearname,1)
  fid = fopen(fname(ii,:),'r');

  if(fid < 3)
    disp(['Unable to open "' fname(ii,:) '"']);
  else
    [d,count] = fscanf(fid,'%7f %5f %6f %6f',[4 inf]);
    d = d';

    fclose(fid);

    for jj=1:size(histname,1);
      s = sort(d(:,jj));
      u = zeros(size(s));
      u(1) = s(1);
      ll = 1;
      for kk=2:length(s)
        if(s(kk) ~= s(kk-1))
          ll = ll+1;
          u(ll) = s(kk);
        end
      end

      u = u(1:ll);

      if(size(u,1) ~= 1)
        [n,x] = hist(d(:,jj),u);
        bar(x,n);
        if(test(1) == 0)
          title([bearname(ii,:) deblank(histname(jj,:))]);
        else
          title([bearname(ii,:) deblank(histname(jj,:)) '  (' ...
                  deblank(colname(test(1),:)) ' = ' num2str(test(2)) ')']);
        end
        if(printme == 1)
          orient landscape;
          command = ['print -dps ' myprinter ' -h'];
          eval(command);
        end

        out = [x(:) n(:)];
        out = out';
```

```
        if(printme == 1)
          fid = fopen(fvalname(ii,:),'w');

          if(fid < 3)
            disp(['Unable to open "' fvalname(ii,:) '"']);
          else

            if(test(1) == 0)
              count = fprintf(fid,'%s%s\n\n', bearname(ii,:), ...
                  deblank(valname(jj,:)));
            else
              count = fprintf(fid,'%s%s (%s = %f)\n\n', bearname(ii,:), ...
                  deblank(valname(jj,:)), deblank(colname(test(1),:)), ...
                  test(2));
            end
            command = sprintf('%f    %4d',out(:,1));
            n = length(command)-4;
            if(length(deblank(colname(jj,:)))+4 > n)
              n = length(deblank(colname(jj,:)))+4;
            end
            count = fprintf(fid,'%-*sQuantity\n', n, deblank(colname(jj,:)));
            for kk=1:size(out,2)
              count = fprintf(fid,'%-*f    %4d\n',n,out(:,kk));
            end
            count = fprintf(fid,'\nmean = %f    std dev = %f\n\n',        ...
                mean(d(:,jj)), std(d(:,jj)));

            fclose(fid);

            command = ['!enscript ' myprinter ' -h -2rG ' fvalname(ii,:)];
            eval(command);
          end
        else
          if(test(1) == 0)
            command = sprintf('%s%s\n', bearname(ii,:), ...
                deblank(valname(jj,:)));
          else
            command = sprintf('%s%s (%s = %f)\n', bearname(ii,:), ...
                deblank(valname(jj,:)), deblank(colname(test(1),:)), test(2));
          end
          disp(command);
          command = sprintf('%f    %4d',out(:,1));
          n = length(command)-4;
          if(length(deblank(colname(jj,:)))+4 > n)
            n = length(deblank(colname(jj,:)))+4;
          end
          command = sprintf('%-*sQuantity', n, deblank(colname(jj,:)));
          disp(command);
          for kk=1:size(out,2)
            command = sprintf('%-*f    %4d',n,out(:,kk));
            disp(command);
          end
          command = sprintf('\nmean = %f    std dev = %f\n\n',        ...
              mean(d(:,jj)), std(d(:,jj)));
          disp(command);
          disp('Press any key to continue ...');
          pause;
          disp(' ');
        end
      else
        disp('hist command returned empty matrices');
      end
    end
  end
end
```

## H.1.8 compli.c

This program uses the mean standard deviation calculated by settlestats.m to determine the range of values for the controller variables of damping ratio, natural frequency, and feedback gain. Using these newly calculated ranges, the gain at 1000 Hz, bandwidth, maximum compliance, maximum steady state control signal, and maximum closed loop gain are calculated and output to a binary file to conserve disk space.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#define RADIX 2.0
#define NR_END 1
#define MAXM 8

#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))

#define MYERR       1e-8
#define OVERSHOOT   0.1
#define MAXSETTLE   0.6
#define MYSAMPLE    20000.0
#define SAMP_SIZE     1
#define PI          3.141592654
#ifdef NOISE
#ifdef CLOSE_TEST
#define SAVENUM     5
#else
#define SAVENUM     4
#endif
#else
#define SAVENUM     3
#endif

void DigStabFunc(double A1, double A2, double P, double Q, double newt,
                 double a22, double a21, double *J1, double *J2, double *J3,
                 double *J4, double *J5, double *M1, double *M2, double *M3,
                 double *M4, double *N1, double *N2, double *N3, double *N4);
int DigReference(double bfreq, double efreq, double dfreq, double bT,
                 double eT, double dT, double initx, double initu, double maxt,
                 double **settle, int *n);
#ifdef NOISE
int DigGetNoiseData(int bearing, FILE *ifp, int size, int offset,
                    double *noise);
int DigStepWithNoise(int bearing, double b, double damp, double freq, double T,
                     double initx, double initu, double maxt, double P,
                     double Q, double A1, double A2, double *noise, double **y,
                     double **u, double **t, int *n);
#endif
int DigStep(int bearing, double b, double damp, double freq, double T,
            double initx, double initu, double maxt, double P, double Q,
            double A1, double A2, double **y, double **u, double **t, int *n);
int DigBode(double *num, double *den, int numsize, int densize, double ts,
            double begfreq, double endfreq, double **mag, double **freq,
            int *magsize);
int zrhqr(double a[], int m, double rtr[], double rti[]);
void balanc(double **a, int n);
int hqr(double **a, int n, double wr[], double wi[]);
double **matrix(long nrl, long nrh, long ncl, long nch);
void free_matrix(double **m, long nrl, long nrh, long ncl, long nch);

void main(int argc, char **argv)
{
```

```
  int i, j, l, m, begin;
  int curcols, currows, ebara, axis;
  int ibeg, jbeg, setsize;
  int bhat_size, damp_size, freq_size;
  float *buffer;
  char *filename;
  FILE *fp, *ifp;
  double P, Q, A1, A2;
  double bbhat, ebhat, bdamp, edamp, bfreq, efreq;
  double bhat, freq, damp;
  double a21, a22, *settle, mysettle, udelta, ufinal, umax;
  double *y, *u, *t, initx, initu, over, set, overu, setu;
  double subtot[9], rroots[9], iroots[9];
  double J1, J2, J3, J4, J5, M1, M2, M3, M4, N1, N2, N3, N4;
  double num[8], den[8], *mag, *frq, compl;
#ifdef NOISE
  int nsize;
  double *noise;
  char *fnames[] = {
    "/tmp/axial20k.dat",
    "/tmp/rad1x20k.dat",
    "/tmp/rad1y20k.dat",
    "/tmp/rad2x20k.dat",
    "/tmp/rad2y20k.dat"
    };
#endif
  int magsize;

  ebara = 0;
  begin = axis = 0;

  if(argc < 2 || argc > 6)
    {
      fprintf(stderr, "Syntax error\n");
      fprintf(stderr, "Syntax:  compli filename [-a n] [-b m]\n");
      fprintf(stderr, "         where:  filename - binary array filename\n");
      fprintf(stderr, "                 -b n -> start integer [0-1000000]\n");
      fprintf(stderr, "                 -a m -> bearing axis [0-4]\n");
      exit(1);
    }

  for(i=1; i<argc; i++)
    {
      if(i == 1)
        filename = argv[1];
      else if(argv[i][0] == '-')
        {
          if(argv[i][1] == 'b')
            begin = atoi(argv[i+1]);
          else if(argv[i][1] == 'a')
            axis = atoi(argv[i+1]);
        }
    }

  currows = 0;

  if(begin)
    {
      if((fp = fopen(filename, "r+")) == NULL)
        {
          fprintf(stderr, "Unable to open file \"%s\" (r+)\n", filename);
          exit(1);
        }
    }
  else
    {
      if((fp = fopen(filename, "w+")) == NULL)
        {
          fprintf(stderr, "Unable to open file \"%s\" (w+)\n", filename);
```

```
        exit(1);
      }
  }

if(ebara)
  {
    switch(axis)
      {
      default:
      case 0:
        P = 11.081;
        Q = 27701.879;
        A1 = 1666.622;
        A2 = 1700.000;
        bdamp = floor((4.033827 - 2.101680) * 10.0) / 10.0;
        edamp = ceil((4.033827 + 2.101680) * 10.0) / 10.0;
        bbhat = floor(402.967777 - 254.562878);
        ebhat = ceil(402.967777 + 254.562878);
        bfreq = floor((527.319758 - 238.103058) / 10.0) * 10.0;
        efreq = ceil((527.319758 + 238.103058) / 10.0) * 10.0;
        break;
      case 1:
        P = 10.278;
        Q = 41113.653;
        A1 = 14222.428;
        A2 = 4370.000;
        bdamp = floor((4.033827 - 2.101680) * 10.0) / 10.0;
        edamp = ceil((4.033827 + 2.101680) * 10.0) / 10.0;
        bbhat = floor(402.967777 - 254.562878);
        ebhat = ceil(402.967777 + 254.562878);
        bfreq = floor((527.319758 - 238.103058) / 10.0) * 10.0;
        efreq = ceil((527.319758 + 238.103058) / 10.0) * 10.0;
        break;
      case 2:
        P = 10.278;
        Q = 41113.653;
        A1 = 14575.775;
        A2 = 4030.000;
        bdamp = floor((4.033827 - 2.101680) * 10.0) / 10.0;
        edamp = ceil((4.033827 + 2.101680) * 10.0) / 10.0;
        bbhat = floor(402.967777 - 254.562878);
        ebhat = ceil(402.967777 + 254.562878);
        bfreq = floor((527.319758 - 238.103058) / 10.0) * 10.0;
        efreq = ceil((527.319758 + 238.103058) / 10.0) * 10.0;
        break;
      case 3:
        P = 20.263;
        Q = 81050.754;
        A1 = 4027.457;
        A2 = 2450.000;
        bdamp = floor((4.033827 - 2.101680) * 10.0) / 10.0;
        edamp = ceil((4.033827 + 2.101680) * 10.0) / 10.0;
        bbhat = floor(402.967777 - 254.562878);
        ebhat = ceil(402.967777 + 254.562878);
        bfreq = floor((527.319758 - 238.103058) / 10.0) * 10.0;
        efreq = ceil((527.319758 + 238.103058) / 10.0) * 10.0;
        break;
      case 4:
        P = 20.263;
        Q = 81050.754;
        A1 = 4108.477;
        A2 = 2600.000;
        bdamp = floor((4.033827 - 2.101680) * 10.0) / 10.0;
        edamp = ceil((4.033827 + 2.101680) * 10.0) / 10.0;
        bbhat = floor(402.967777 - 254.562878);
        ebhat = ceil(402.967777 + 254.562878);
        bfreq = floor((527.319758 - 238.103058) / 10.0) * 10.0;
        efreq = ceil((527.319758 + 238.103058) / 10.0) * 10.0;
        break;
```

```
      }
  }
else
  {
    switch(axis)
      {
      default:
      case 0:
        P = 7.990;
        Q = 22739.569;
        A1 = 11097.006;
        A2 = 13310.0;
        bdamp = floor((3.272777 - (1.25 * 2.717406)) * 10.0) / 10.0;
        edamp = ceil((3.272777 + (1.25 * 2.717406)) * 10.0) / 10.0;
        bbhat = floor(370.414059 - (1.25 * 271.290529));
        ebhat = ceil(370.414059 + (1.25 * 271.290529));
        bfreq = floor((347.082893 - (1.25 * 232.298166)) / 10.0) * 10.0;
        efreq = ceil((347.082893 + (1.25 * 232.298166)) / 10.0) * 10.0;
        break;
      case 1:
        P = 7.123;
        Q = 7737.770;
        A1 = 4619.584;
        A2 = 13130.0;
        bdamp = floor((3.918438 - (1.25 * 2.817466)) * 10.0) / 10.0;
        edamp = ceil((3.918438 + (1.25 * 2.817466)) * 10.0) / 10.0;
        bbhat = floor(328.021404 - (1.25 * 279.017239));
        ebhat = ceil(328.021404 + (1.25 * 279.017239));
        bfreq = floor((276.262071 - (1.25 * 209.387555)) / 10.0) * 10.0;
        efreq = ceil((276.262071 + (1.25 * 209.387555)) / 10.0) * 10.0;
        break;
      case 2:
        P = 8.296;
        Q = 8882.644;
        A1 = 4558.848;
        A2 = 13080.0;
        bdamp = floor((3.814237 - (1.25 * 2.808301)) * 10.0) / 10.0;
        edamp = ceil((3.814237 + (1.25 * 2.808301)) * 10.0) / 10.0;
        bbhat = floor(332.704460 - (1.25 * 278.550625));
        ebhat = ceil(332.704460 + (1.25 * 278.550625));
        bfreq = floor((285.769852 - (1.25 * 212.645306)) / 10.0) * 10.0;
        efreq = ceil((285.769852 + (1.25 * 212.645306)) / 10.0) * 10.0;
        break;
      case 3:
        P = 16.926;
        Q = 35530.576;
        A1 = 3863.909;
        A2 = 11140.0;
        bdamp = floor((3.412286 - (1.25 * 2.731543)) * 10.0) / 10.0;
        edamp = ceil((3.412286 + (1.25 * 2.731543)) * 10.0) / 10.0;
        bbhat = floor(349.211106 - (1.25 * 273.471228));
        ebhat = ceil(349.211106 + (1.25 * 273.471228));
        bfreq = floor((336.787694 - (1.25 * 226.591365)) / 10.0) * 10.0;
        efreq = ceil((336.787694 + (1.25 * 226.591365)) / 10.0) * 10.0;
        break;
      case 4:
        P = 15.113;
        Q = 33201.349;
        A1 = 4212.3;
        A2 = 12290.0;
        bdamp = floor((3.550881 - (1.25 * 2.756218)) * 10.0) / 10.0;
        edamp = ceil((3.550881 + (1.25 * 2.756218)) * 10.0) / 10.0;
        bbhat = floor(343.223786 - (1.25 * 275.796596));
        ebhat = ceil(343.223786 + (1.25 * 275.796596));
        bfreq = floor((335.875802 - (1.25 * 225.363705)) / 10.0) * 10.0;
        efreq = ceil((335.875802 + (1.25 * 225.363705)) / 10.0) * 10.0;
        break;
      }
  }
```

```
  if(bdamp < 0.0)
    bdamp = 0.1;
  if(bbhat < 0.0)
    bbhat = 2.0;
  if(bfreq < 0.0)
    bfreq = 10.0;

#ifdef NOISE
  noise = NULL;
  nsize = 0;
#endif

  switch(axis)
    {
    default:
    case 0:
      initx = -0.0002;
      initu = 1.167;
      break;
    case 1:
    case 2:
    case 3:
    case 4:
      initx = 0.0001;
      initu = 0.0;
      break;
    }

  /*
   * newt = 0.00015;
   * damp = [1:0.1:7];
   * freq = [250:10:700];
   * bhat = [100:2:700];
   */

  bhat_size = ((int) ceil((ebhat - bbhat) / 2.0)) + 1;
  freq_size = ((int) ceil((efreq - bfreq) / 10.0)) + 1;
  damp_size = ((int) ceil((edamp - bdamp) / 0.1)) + 1;

  curcols = (damp_size * SAMP_SIZE);
  currows = (int) (((float) begin) / ((float) curcols));

  fseek(fp, 0L, SEEK_SET);
  if(!fwrite((void *) &currows, sizeof(int), 1, fp))
    {
      fprintf(stderr, "row data write failed\n");
      fclose(fp);
      exit(1);
    }
  if(!fwrite((void *) &curcols, sizeof(int), 1, fp))
    {
      fprintf(stderr, "columns data write failed\n");
      fclose(fp);
      exit(1);
    }

  if((buffer = (float *) calloc(damp_size*freq_size*SAVENUM, sizeof(float)))
     == NULL)
    {
      fprintf(stderr, "Out of memory error - compliance.\n");
      exit(1);
    }

  /* get maximum settling times for given frequencies */
  if(DigReference(10.0, 1000.0, 10.0, 1000.0, 50000.0, 1000.0,
                  0.0, 1.0, MAXSETTLE, &settle, &setsize))
    {
      fprintf(stderr, "Error encountered in DigReference - compliance.\n");
      free(buffer);
```

```
      exit(1);
    }

  mysettle = (MYSAMPLE - 1000.0) / 1000.0;
  if((mysettle - floor(mysettle)) <= (ceil(mysettle) - mysettle))
    mysettle = floor(mysettle);
  else
    mysettle = ceil(mysettle);
  /* 100 = size of frequency value range */
  j = ((int) mysettle) * 100;
  mysettle = settle[j];

  free(settle);

#ifdef NOISE
  if((ifp = fopen(fnames[axis], "r")) == NULL)
    {
      fprintf(stderr, "Unable to open noise file \"%s\" - compliance.\n",
              fnames[axis]);
      free(buffer);
      exit(1);
    }
#endif

  if(begin)
    {
      ibeg = (int) (((float) currows) / ((float) freq_size));
      jbeg = (int) (currows - (ibeg * freq_size));
      fseek(fp, (currows*curcols*SAVENUM*sizeof(float)) + (2 * sizeof(int)),
            SEEK_SET);
    }
  else
    ibeg = jbeg = 0;

  begin = 0;
  for(i=ibeg; i<bhat_size; i++)    /* bhat loop */
    {
      bhat = (bbhat + (i * 2.0));
      for(j=jbeg; j<freq_size; j++)    /* freq loop */
        {
          freq = (bfreq + (j * 10.0));
          a21 = freq * freq;
          for(l=0; l<damp_size; l++)    /* damp loop */
            {
              damp = (bdamp + (l * 0.1));
              a22 = 2.0 * damp * freq;
              DigStabFunc(A1, A2, P, Q, MYSAMPLE, a22, a21, &J1, &J2, &J3, &J4,
                          &J5, &M1, &M2, &M3, &M4, &N1, &N2, &N3, &N4);
              subtot[8] = 1.0;
              subtot[7] = ((J1*M2)/bhat)+(N2-1.0);
              subtot[6] = (((J1*M3)+(J2*M2))/bhat)+(N3-N2);
              subtot[5] = (((J1*M4)+(J2*M3)+(J3*M2))/bhat)+(N4-N3);
              subtot[4] = (((J2*M4)+(J3*M3)+(J4*M2))/bhat)-N4;
              subtot[3] = ((J3*M4)+(J4*M3)+(J5*M2))/bhat;
              subtot[2] = ((J4*M4)+(J5*M3))/bhat;
              subtot[1] = (J5*M4)/bhat;
              if(zrhqr(subtot, 8, rroots, iroots))
                {
                  fprintf(stderr, "Eigenvalue calculation problem ");
                  fprintf(stderr, "(i = %d  j = %d  l = %d)\n", i, j, l);
                  goto error;
                }
              for(m=1; m<=8; m++)
                if((rroots[m]*rroots[m])+(iroots[m]*iroots[m]) > 1.0)
                  break;
              if(m == 9)
                {
                  if(DigStep(axis, bhat, damp, freq, MYSAMPLE, initx,
                             initu, 0.6, P, Q, A1, A2, &y, &u,
```

```
                        &t, &setsize))
                {
                  fprintf(stderr, "DigStep calculation problem ");
                  fprintf(stderr, "(i = %d  j = %d  l = %d)\n", i, j, l);
                  goto error;
                }
              if(axis)
                ufinal = 0.0;
              else
                {
                  for(m=0; m<setsize; m++)
                    if(t[m] >= mysettle)
                      break;
                  ufinal = setsize-m;
                  udelta = 0.0;
                  for(; m<setsize; m++)
                    udelta += u[m];
                  ufinal = udelta / ufinal;
                }
              udelta = fabs(0.05 * initx);
              for(m=setsize-1; m>=0; m--)
                if(fabs(y[m]) > udelta)
                  {
                    set = t[m];
                    break;
                  }
              free(y);
              free(u);
              free(t);
#ifdef NOISE
              if((int) ((2.0*set*MYSAMPLE)+0.5) > nsize)
                {
                  nsize = (int) ((2.0*set*MYSAMPLE)+0.5);
                  if((noise =
                      (double *) realloc(noise, nsize*sizeof(double)))
                     == NULL)
                    {
                      fprintf(stderr,
                              "Out of memory error - compliance.\n");
                      exit(1);
                    }
                  if(DigGetNoiseData(axis, ifp, nsize, 1000, noise))
                    {
                      fprintf(stderr,
                              "DigGetNoiseData error - compliance.\n");
                      exit(1);
                    }
                }
              if(DigStepWithNoise(axis, bhat, damp, freq, MYSAMPLE, initx,
                                  initu, 2.0*set, P, Q, A1, A2, noise, &y,
                                  &u, &t, &setsize))
                {
                  fprintf(stderr, "DigStepWithNoise calculation problem ");
                  fprintf(stderr, "(i = %d  j = %d  l = %d)\n", i, j, l);
                  goto error;
                }
              overu = 0.0;
              for(m=0; m<setsize; m++)
                {
                  if(t[m] < set)
                    continue;
                  if(fabs(u[m]-ufinal) > overu)
                    overu = fabs(u[m]-ufinal);
                }
              free(y);
              free(u);
              free(t);
#endif
#ifdef CLOSE_TEST
```

```
                num[0] = 1.0;
                num[1] = J1*M2;
                num[2] = (J1*M3)+(J2*M2);
                num[3] = (J1*M4)+(J2*M3)+(J3*M2);
                num[4] = (J2*M4)+(J3*M3)+(J4*M2);
                num[5] = (J3*M4)+(J4*M3)+(J5*M2);
                num[6] = (J4*M4)+(J5*M3);
                num[7] = (J5*M4);
                den[0] = bhat;
                den[1] = (bhat*(N2-1.0))+(J1*M2);
                den[2] = (bhat*(N3-N2))+(J2*M2)+(J1*M3);
                den[3] = (bhat*(N4-N3))+(J3*M2)+(J2*M3)+(J1*M4);
                den[4] = (bhat*(-N4))+(J4*M2)+(J3*M3)+(J2*M4);
                den[5] = (J5*M2)+(J4*M3)+(J3*M4);
                den[6] = (J5*M3)+(J4*M4);
                den[7] = (J5*M4);
                if(DigBode(num, den, 8, 8, MYSAMPLE, 0.1, 1000.0, &mag, &frq,
                          &magsize))
                  {
                    fprintf(stderr, "Bode calculation problem ");
                    fprintf(stderr, "(i = %d  j = %d  l = %d)\n",
                            i, j, l);
                    goto error;
                  }
#if 0
                for(m=0; m<magsize; m++)
                  printf("%6.2f    %8.5f\n", frq[m], mag[m]);
                exit(0);
#endif
                udelta = -1000.0;
                set = -1.0;
                for(m=0; m<magsize; m++)
                  {
                    if(mag[m] > udelta)
                      udelta = mag[m];
                    if(mag[m] < 0.0 && mag[m-1] > 0.0)
                        set = frq[m-1];
                  }
                over = mag[magsize-1];
                free(mag);
                free(frq);
#endif
                num[0] = P;
                num[1] = bhat*M2;
                num[2] = bhat*(M3-M2);
                num[3] = bhat*(M4-M3);
                num[4] = bhat*(-M4);
                num[5] = num[6] = num[7] = 0.0;
                den[0] = bhat;
                den[1] = (bhat*(N2-1.0))+(J1*M2);
                den[2] = (bhat*(N3-N2))+(J2*M2)+(J1*M3);
                den[3] = (bhat*(N4-N3))+(J3*M2)+(J2*M3)+(J1*M4);
                den[4] = (bhat*(-N4))+(J4*M2)+(J3*M3)+(J2*M4);
                den[5] = (J5*M2)+(J4*M3)+(J3*M4);
                den[6] = (J5*M3)+(J4*M4);
                den[7] = (J5*M4);
                if(DigBode(num, den, 8, 8, MYSAMPLE, 0.1, 1000.0, &mag, &frq,
                          &magsize))
                  {
                    fprintf(stderr, "Bode calculation problem ");
                    fprintf(stderr, "(i = %d  j = %d  l = %d)\n",
                            i, j, l);
                    goto error;
                  }
#if 0
                for(m=0; m<magsize; m++)
                  printf("%6.2f    %8.5f\n", frq[m], mag[m]);
                exit(0);
#endif
```

```
                compl = mag[0];
                for(m=1; m<magsize; m++)
                  {
                    if(mag[m] > compl)
                      compl = mag[m];
                  }
                buffer[(j*SAVENUM*damp_size)+(1*SAVENUM)] = (float) over;
                buffer[(j*SAVENUM*damp_size)+(1*SAVENUM)+1] = (float) set;
                buffer[(j*SAVENUM*damp_size)+(1*SAVENUM)+2] = (float) compl;
#ifdef NOISE
#ifdef CLOSE_TEST
                buffer[(j*SAVENUM*damp_size)+(1*SAVENUM)+3] = (float) overu;
                buffer[(j*SAVENUM*damp_size)+(1*SAVENUM)+4] = (float) udelta;
#else
                buffer[(j*SAVENUM*damp_size)+(1*SAVENUM)+3] = (float) overu;
#endif
#endif
                free(mag);
                free(frq);
                begin++;
              }
            else  /* m != 8 */
              {
              error:
                buffer[(j*SAVENUM*damp_size)+(1*SAVENUM)] = 0.0;
                buffer[(j*SAVENUM*damp_size)+(1*SAVENUM)+1] = (float) -1.0;
                buffer[(j*SAVENUM*damp_size)+(1*SAVENUM)+2] = (float) 0.0;
#ifdef NOISE
#ifdef CLOSE_TEST
                buffer[(j*SAVENUM*damp_size)+(1*SAVENUM)+3] = (float) 0.0;
                buffer[(j*SAVENUM*damp_size)+(1*SAVENUM)+4] = (float) 0.0;
#else
                buffer[(j*SAVENUM*damp_size)+(1*SAVENUM)+3] = (float) 0.0;
#endif
#endif
              }  /* damp loop */
          }      /* freq loop */
        if(jbeg)
          jbeg = 0;
        currows++;
      }          /* bhat loop */
    if(fwrite((void *) buffer, sizeof(float), damp_size*freq_size*SAVENUM,
              fp) != damp_size*freq_size*SAVENUM)
      {
        fprintf(stderr, "data write failed\n");
        fclose(fp);
        exit(1);
      }

    fseek(fp, 0L, SEEK_SET);
    if(!fwrite((void *) &currows, sizeof(int), 1, fp))
      {
        fprintf(stderr, "row data write failed\n");
        fclose(fp);
        exit(1);
      }
    if(!fwrite((void *) &curcols, sizeof(int), 1, fp))
      {
        fprintf(stderr, "columns data write failed\n");
        fclose(fp);
        exit(1);
      }
    fflush(fp);
    fseek(fp, (currows*curcols*SAVENUM*sizeof(float)) + (2 * sizeof(int)),
          SEEK_SET);
    printf("%3d of %3d (%d)\n", i+1, bhat_size, begin);
  }

  fclose(fp);
```

```
  fclose(ifp);
  free(buffer);
  exit(0);
}


void DigStabFunc(double A1, double A2, double P, double Q, double T,
                 double a22, double a21, double *J1, double *J2, double *J3,
                 double *J4, double *J5, double *M1, double *M2, double *M3,
                 double *M4, double *N1, double *N2, double *N3, double *N4)
{
  double K1, K2, K3, K4, L1, L2, L3, L4, T1, T2, sq;

  T1 = 1.0/T;
  T2 = T1*T1;
  sq = sqrt(Q);

#ifdef BACKWARD
  /* central difference acceleration and backward difference velocity */
  *J1 = (3.0/(4.0*(T2)))+((3.0*a22)/(2.0*T1))+(a21);
  *J2 = (-4.0/(4.0*(T2)))-((4.0*a22)/(2.0*T1));
  *J3 = (-2.0/(4.0*(T2)))+(a22/(2.0*T1));
  *J4 = 1.0/(T2);
  *J5 = -1.0/(4.0*(T2));

#else
  /* central difference acceleration and  velocity */
  *J1 = (1.0/(4.0*(T2)))+(a22/(2.0*T1))+(a21);
  *J2 = 0.0;
  *J3 = (-2.0/(4.0*(T2)))-(a22/(2.0*T1));
  *J4 = 0.0;
  *J5 = 1.0/(4.0*(T2));

#endif

  K1 = (-A1*P)/(A2*Q);
  K2 = (-A1*P)/(A2*((A2*A2)-Q));
  K3 = (A1*P)/(2.0*Q*(A2-sq));
  K4 = (A1*P)/(2.0*Q*(A2+sq));

  L1 = exp(-A2*T1);
  L2 = exp(-sq*T1);
  L3 = exp(sq*T1);
  L4 = 1;

  *M1 = 0;
  *M2 = (-K1*(L1+L2+L3))-(K2*(L2+L3+L4))-(K3*(L1+L3+L4))-(K4*(L1+L2+L4));
  *M3 = (K1*((L1*L2)+(L3*(L1+L2))))+(K2*((L2*L4)+(L3*(L2+L4))))+
    (K3*((L1*L4)+(L3*(L1+L4))))+(K4*((L1*L4)+(L2*(L1+L4))));
  *M4 = (-K1*L1*L2*L3)-(K2*L2*L3*L4)-(K3*L1*L3*L4)-(K4*L1*L2*L4);

  *N1 = 1.0;
  *N2 = -L1-L2-L3;
  *N3 = (L1*L2)+(L1*L3)+(L2*L3);
  *N4 = -(L1*L2*L3);

}

int zrhqr(double a[], int m, double rtr[], double rti[])
{
  int j, k;
  double **hess, xr, xi;

  hess = matrix(1,MAXM,1,MAXM);
  if(m > MAXM || a[m] == 0.0)
    {
      fprintf(stderr, "bad args in zrhqr\n");
      free_matrix(hess,1,MAXM,1,MAXM);
      return(1);
```

```
    }
  for(k=1;k<=m;k++)
    {
      hess[1][k] = -a[m-k]/a[m];
      for(j=2;j<=m;j++)
        hess[j][k] = 0.0;
      if(k != m)
        hess[k+1][k] = 1.0;
    }
  balanc(hess, m);
  if(hqr(hess,m,rtr,rti))
    {
      free_matrix(hess,1,MAXM,1,MAXM);
      return(1);
    }
  for(j=2; j<=m; j++)
    {
      xr = rtr[j];
      xi = rti[j];
      for(k=j-1; k>=1; k--)
        {
          if(rtr[k] <= xr)
            break;
          rtr[k+1] = rtr[k];
          rti[k+1] = rti[k];
        }
      rtr[k+1] = xr;
      rti[k+1] = xi;
    }
  free_matrix(hess,1,MAXM,1,MAXM);
  return(0);
}


void balanc(double **a, int n)
{
  int last,j,i;
  double s,r,g,f,c,sqrdx;

  sqrdx = RADIX*RADIX;
  last = 0;
  while(last == 0)
    {
      last = 1;
      for(i=1; i<=n; i++)
        {
          r = c = 0.0;
          for(j=1; j<=n; j++)
            if(j != i)
              {
                c += fabs(a[j][i]);
                r += fabs(a[i][j]);
              }
          if(c && r)
            {
              g = r/RADIX;
              f = 1.0;
              s = c+r;
              while(c<g)
                {
                  f *= RADIX;
                  c *= sqrdx;
                }
              g = r*RADIX;
              while(c > g)
                {
                  f /= RADIX;
                  c /= sqrdx;
                }
```

```
            if((c+r)/f < 0.95*s)
              {
                last = 0;
                g = 1.0/f;
                for(j=1; j<=n; j++)
                  a[i][j] *= g;
                for(j=1; j<=n; j++)
                  a[j][i] *= f;
              }
          }
      }
  }
}


int hqr(double **a, int n, double wr[], double wi[])
{
  int nn, m, l, k, j, its, i, mmin;
  double z, y, x, w, v, u, t, s, r, q, p, anorm;

  anorm = fabs(a[1][1]);
  for(i=2; i<=n; i++)
    for(j=(i-1); j<=n; j++)
      anorm += fabs(a[i][j]);
  nn=n;
  t=0.0;
  while(nn >= 1)
    {
      its = 0;
      do {
        for(l=nn; l>=2; l--)
          {
            s = fabs(a[l-1][l-1])+fabs(a[l][l]);
            if(s == 0.0)
              s=anorm;
            if((double)(fabs(a[l][l-1]) + s) == s)
              break;
          }
        x = a[nn][nn];
        if(l == nn)
          {
            wr[nn] = x+t;
            wi[nn--] = 0.0;
          }
        else
          {
            y = a[nn-1][nn-1];
            w = a[nn][nn-1]*a[nn-1][nn];
            if(l == (nn-1))
              {
                p = 0.5*(y-x);
                q = p*p+w;
                z = sqrt(fabs(q));
                x += t;
                if(q >= 0.0)
                  {
                    z = p+SIGN(z,p);
                    wr[nn-1] = wr[nn] = x+z;
                    if(z)
                      wr[nn] = x-w/z;
                    wi[nn-1] = wi[nn] = 0.0;
                  }
                else
                  {
                    wr[nn-1] = wr[nn] = x+p;
                    wi[nn-1] = -(wi[nn] = z);
                  }
                nn -= 2;
              }
```

```
else
  {
    if(its == 30)
      {
        fprintf(stderr, "Too many iterations in hqr\n");
        return(1);
      }
    if(its == 10 || its == 20)
      {
        t += x;
        for(i=1; i<=nn; i++)
          a[i][i] -= x;
        s = fabs(a[nn][nn-1])+fabs(a[nn-1][nn-2]);
        y = x = 0.75*s;
        w = -0.4375*s*s;
      }
    ++its;
    for(m=(nn-2); m>=l; m--)
      {
        z = a[m][m];
        r = x-z;
        s = y-z;
        p = (r*s-w)/a[m+1][m]+a[m][m+1];
        q = a[m+1][m+1]-z-r-s;
        r = a[m+2][m+1];
        s = fabs(p)+fabs(q)+fabs(r);
        p /= s;
        q /= s;
        r /= s;
        if(m == l)
          break;
        u = fabs(a[m][m-1])*(fabs(q)+fabs(r));
        v = fabs(p)*(fabs(a[m-1][m-1])+fabs(z)+fabs(a[m+1][m+1]));
        if((double)(u+v) == v)
          break;
      }
    for(i=m+2; i<=nn; i++)
      {
        a[i][i-2] = 0.0;
        if(i != (m+2))
          a[i][i-3] = 0.0;
      }
    for(k=m; k<=nn-1; k++)
      {
        if(k != m)
          {
            p = a[k][k-1];
            q = a[k+1][k-1];
            r = 0.0;
            if(k != (nn-1))
              r = a[k+2][k-1];
            if((x = fabs(p)+fabs(q)+fabs(r)) != 0.0)
              {
                p /= x;
                q /= x;
                r /= x;
              }
          }
        if((s = SIGN(sqrt(p*p+q*q+r*r),p)) != 0.0)
          {
            if(k == m)
              {
                if(l != m)
                  a[k][k-1] = -a[k][k-1];
              }
            else
              a[k][k-1] = -s*x;
            p += s;
            x = p/s;
```

```
                        y = q/s;
                        z = r/s;
                        q /= p;
                        r /= p;
                        for(j=k; j<=nn; j++)
                          {
                            p = a[k][j]+q*a[k+1][j];
                            if(k != (nn-1))
                              {
                                p += r*a[k+2][j];
                                a[k+2][j] -= p*z;
                              }
                            a[k+1][j] -= p*y;
                            a[k][j] -= p*x;
                          }
                        mmin = nn<k+3 ? nn : k+3;
                        for(i=1; i<=mmin; i++)
                          {
                            p = x*a[i][k]+y*a[i][k+1];
                            if(k != (nn-1))
                              {
                                p += z*a[i][k+2];
                                a[i][k+2] -= p*r;
                              }
                            a[i][k+1] -= p*q;
                            a[i][k] -= p;
                          }
                      }
                  }
              }
          }
      } while(l < nn-1);
    }
  return(0);
}

double **matrix(long nrl, long nrh, long ncl, long nch)
{
  long i, nrow = nrh-nrl+1, ncol = nch-ncl+1;
  double **m;

  /* allocate pointers to rows */
  if((m = (double **) malloc((size_t) ((nrow+NR_END) * sizeof(double *))))
     == NULL)
    {
      fprintf(stderr, "allocation failure 1 in matrix()\n");
      exit(1);
    }
  m += NR_END;
  m -= nrl;

  /* allocate rows and set pointers to them */
  if((m[nrl] = (double *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(double))))
     == NULL)
    {
      fprintf(stderr, "allocation failure 2 in matrix()\n");
      exit(1);
    }
  m[nrl] += NR_END;
  m[nrl] -= ncl;

  for(i=nrl+1; i<=nrh; i++)
    m[i]=m[i-1]+ncol;

  return(m);

}
```

```
void free_matrix(double **m, long nrl, long nrh, long ncl, long nch)
{
  free((char *) (m[nrl]+ncl-NR_END));
  free((char *) (m+nrl-NR_END));
}


int DigReference(double bfreq, double efreq, double dfreq, double bT,
                 double eT, double dT, double initx, double initu, double maxt,
                 double **settle, int *n)
{
  int i, j, k, fsize, tsize, setsize, ee, bb;
  double *time, upt, *freq, output, *set;

  (*settle) = NULL;
  (*n) = 0;

  tsize = ((int) (((eT - bT) / dT) + 0.5)) + 1;
  fsize = ((int) ((efreq - bfreq) / dfreq)) + 1;
  setsize = fsize * tsize;

  if((time = (double *) calloc(tsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigReference.\n");
      return(1);
    }

  if((freq = (double *) calloc(fsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigReference.\n");
      free(time);
      return(1);
    }

  if((set = (double *) calloc(setsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigReference.\n");
      free(time);
      free(freq);
      return(1);
    }

  for(time[0]=bT,i=1; i<tsize; i++)
    time[i] = time[i-1] + dT;

  for(freq[0]=bfreq,i=1; i<fsize; i++)
    freq[i] = freq[i-1] + dfreq;

  for(i=0; i<tsize; i++)
    {
      for(j=0; j<fsize; j++)
        {
          ee = (int) (maxt*time[i]);
          bb = 1;
          k = ee >> 1;
          while(k != 0)
            {
              upt = k / time[i];
              /*
               * This equation is the solution of the inverse Laplace transform
               * of a second order system having a damping ratio of 1.0 and
               * determining the closest sampling time to that point
               */
              output = 1.0 - ((1.0 + (freq[j]*upt)) * exp(-freq[j] * upt));
              if(output < 0.95)
                bb = k;
              else
                ee = k;
              k = bb + ((ee - bb) >> 1);
```

```
              if(k == ee || k == bb)
                {
                  set[(i*fsize)+j] = k / time[i];
                  k = 0;
                }
            }
        }
    }

  free(time);
  free(freq);

  (*settle) = set;
  (*n) = setsize;

  return(0);

}


#ifdef NOISE
int DigGetNoiseData(int bearing, FILE *ifp, int size, int offset,
                    double *noise)
{
  int i, j;
  char buf[80];
  float temp;
  double mean, conv;

  if(!ifp || !noise || !size)
    return(1);

  for(i=0; i<offset; i++)
    {
      if(!fgets(buf, 80, ifp))
        {
          fprintf(stderr, "Could not read file to offset (%d) - DigGetNoiseData.\n",
offset);
          return(1);
        }
    }

  for(mean=0.0,i=0; i<size; i++)
    {
      if(!fgets(buf, 80, ifp))
        {
          fseek(ifp, 0L, SEEK_SET);
          for(j=0; j<offset; j++)
            {
              if(!fgets(buf, 80, ifp))
                {
                  fprintf(stderr,
                          "Could not read file to offset (%d) - DigGetNoiseData.\n",
                          offset);
                  return(1);
                }
            }
          if(!fgets(buf, 80, ifp))
            {
              fprintf(stderr, "Could not read file to size (%d) - DigGetNoiseData.\n",
size);
              return(1);
            }
        }
      if(sscanf(buf, "%f", &temp) != 1)
        {
          fprintf(stderr, "Error encountered in sscanf - DigGetNoiseData.\n");
          return(1);
        }
```

```
      noise[i] = (double) temp;
      mean += noise[i];
    }

  rewind(ifp);

  if(bearing)
    conv = 5.0/(8192.0*25000.0);
  else
    conv = 5.0/(8192.0*9450.0);

  mean /= ((double) size);
  for(i=0; i<size; i++)
    noise[i] = (noise[i] - mean) * conv;

  return(0);

}


int DigStepWithNoise(int bearing, double b, double damp, double freq, double T,
                     double initx, double initu, double maxt, double P,
                     double Q, double A1, double A2, double *noise, double **y,
                     double **u, double **t, int *n)
{
  int i, j, k, ttsize, uptsize;
  double a22, a21, g, divs, rT, maxx, T1;
  double *tt, *upt, *ty, *tu, cx[5];
  double px[2], pxdot[2], am[5], vm[3], xm, cuT, cu;
  double ampx, ampu, accel, vel, ampxdot, maxu;
  double rkt[2][4], rk[4];

  /*
   * [y, u, t] = DigStep(bearing, b, damp, freq, T, initx, initu, maxt)
   *     bearing - vacuum pump bearing number
   *               0 = axial bearing
   *               1 = radial 1X bearing
   *               2 = radial 1Y bearing
   *               3 = radial 2X bearing
   *               4 = radial 2Y bearing
   *     b       - controller gain (b hat)
   *     damp    - controller damping ratio
   *     freq    - controller natural frequency
   *     T       - controller sampling interval
   *     initx   - initial position
   *     initu   - initial control
   *     maxt    - maximum response time
   */

  T1 = 1.0/T;
  a22 = 2.0 * damp * freq;
  a21 = freq * freq;
  g = 9.807;
  divs = 10.0;
  rT = T1 / divs;
  (*u) = (*y) = (*t) = NULL;
  (*n) = 0;

  ttsize = ((int) ((maxt * T) + 0.5)) + 1;

  if((tt = (double *) calloc(ttsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigStep.\n");
      return(1);
    }

  uptsize = ((int) ((maxt / rT) + 0.5)) + 1;

  if((upt = (double *) calloc(uptsize, sizeof(double))) == NULL)
```

```
    {
      fprintf(stderr, "Out of memory error - DigStep.\n");
      free(tt);
      return(1);
    }

  if((ty = (double *) calloc(uptsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigStep.\n");
      free(tt);
      free(upt);
      return(1);
    }

  if((tu = (double *) calloc(uptsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigStep.\n");
      free(tt);
      free(upt);
      free(ty);
      return(1);
    }

  for(i=1; i<uptsize; i++)
    upt[i] = upt[i-1] + rT;

  for(j=0,i=10; i<uptsize; i+=10)
    tt[j++] = upt[i];

  for(i=0; i<5; i++)
    cx[i] = initx;

  if(!bearing)
    {
      maxx = 0.0002;
      maxu = 3.0;
    }
  else
    {
      maxx = 0.0001;
      maxu = 2.0;
    }

  px[1] = pxdot[0] = pxdot[1] = 0.0;
  px[0] = initx;

#ifdef BACKWARD
  am[0] = 3.0; am[1] = -4.0; am[2] = -2.0; am[3] = 4.0; am[4] = -1.0;
#else
  am[0] = 1.0; am[1] = 0.0; am[2] = -2.0; am[3] = 0.0; am[4] = 1.0;
#endif
  for(i=0; i<5; i++)
    am[i] /= (4.0 * (T1 * T1));

#ifdef BACKWARD
  vm[0] = 3.0 * a22; vm[1] = -4.0 * a22; vm[2] = a22;
#else
  vm[0] = a22; vm[1] = 0.0; vm[2] = -a22;
#endif
  for(i=0; i<3; i++)
    vm[i] /= (2.0 * T1);

  xm = a21;
  cuT = cu = initu;

  ampx = ampu = initu;

  rk[0] = rk[3] = 1.0/6.0;
  rk[1] = rk[2] = 1.0/3.0;
```

```
for(j=i=0; i<uptsize; i++)
  {
    if(upt[i] == tt[j])
      {
        /* update controller position variables */
        for(k=4; k>0; k--)
          cx[k] = cx[k-1];
        cx[0] = px[0] + noise[j];
        /* determine control signal */
        for(accel=0.0,k=0; k<5; k++)
          accel += (am[k] * cx[k]);
        for(vel=0.0,k=0; k<3; k++)
          vel += (vm[k] * cx[k]);
        cu = cuT - ((accel + vel + (xm*cx[0]))/b);
        if(cu > maxu)
          cu = maxu;
        else if(cu < -maxu)
          cu = -maxu;
        cuT = cu;
        j++;
      }

    /* determine amplified control */
    rkt[0][0] = (-A2*ampx) + (A1*cu);          /* f{y(j),t(j)}            */
    ampxdot = ampx + (rkt[0][0]*(rT/2));       /* y*(j+1/2)               */
    rkt[0][1] = (-A2*ampxdot) + (A1*cu);       /* f{y*(j+1/2),t(j+1/2)}   */
    ampxdot = ampx + (rkt[0][1]*(rT/2));       /* y**(j+1/2)              */
    rkt[0][2] = (-A2*ampxdot) + (A1*cu);       /* f{y**(j+1/2),t(j+1/2)}  */
    ampxdot = ampx + (rkt[0][2]*rT);           /* y*(j+1)                 */
    rkt[0][3] = (-A2*ampxdot) + (A1*cu);       /* f{y*(j+1),t(j+1)}       */
    for(ampxdot=0.0,k=0; k<4; k++)
      ampxdot += (rkt[0][k]*rk[k]);
    ampx += (ampxdot * rT);
    ampu = ampx;

    /* determine new position */
    rkt[0][0] = px[1];                         /* f{y(j),t(j)}            */
    rkt[1][0] = (Q*px[0]) + (P*ampu);          /* f{y(j),t(j)}            */
    pxdot[0] = px[0] + (rkt[0][0]*(rT/2));     /* y*(j+1/2)               */
    pxdot[1] = px[1] + (rkt[1][0]*(rT/2));     /* y*(j+1/2)               */
    rkt[0][1] = pxdot[1];                      /* f{y*(j+1/2),t(j+1/2)}   */
    rkt[1][1] = (Q*pxdot[0]) + (P*ampu);       /* f{y*(j+1/2),t(j+1/2)}   */
    pxdot[0] = px[0] + (rkt[0][1]*(rT/2));     /* y**(j+1/2)              */
    pxdot[1] = px[1] + (rkt[1][1]*(rT/2));     /* y**(j+1/2)              */
    rkt[0][2] = pxdot[1];                      /* f{y**(j+1/2),t(j+1/2)}  */
    rkt[1][2] = (Q*pxdot[0]) + (P*ampu);       /* f{y**(j+1/2),t(j+1/2)}  */
    pxdot[0] = px[0] + (rkt[0][2]*rT);         /* y*(j+1)                 */
    pxdot[1] = px[1] + (rkt[1][2]*rT);         /* y*(j+1)                 */
    rkt[0][3] = pxdot[1];                      /* f{y*(j+1),t(j+1)}       */
    rkt[1][3] = (Q*pxdot[0]) + (P*ampu);       /* f{y*(j+1),t(j+1)}       */
    pxdot[0] = pxdot[1] = 0.0;
    for(k=0; k<4; k++)
      {
        pxdot[0] += (rkt[0][k]*rk[k]);
        pxdot[1] += (rkt[1][k]*rk[k]);
      }
    if(bearing == 0)
      pxdot[1] -= g;

    /* save values for plotting */
    ty[i] = px[0];
    tu[i] = cu;

    /* update variables */
    px[0] += (pxdot[0] * rT);
    px[1] += (pxdot[1] * rT);

    /* compensate for touchdown bearings */
    if(px[0] > maxx)
```

```
        px[0] = maxx;
      else if(px[0] < -maxx)
        px[0] = -maxx;

    }

  free(tt);

  (*y) = ty;
  (*u) = tu;
  (*t) = upt;
  (*n) = uptsize;

  return(0);

}
#endif


int DigStep(int bearing, double b, double damp, double freq, double T,
            double initx, double initu, double maxt, double P, double Q,
            double A1, double A2, double **y, double **u, double **t, int *n)
{
  int i, j, k, ttsize, uptsize;
  double a22, a21, g, divs, rT, maxx, T1;
  double *tt, *upt, *ty, *tu, cx[5];
  double px[2], pxdot[2], am[5], vm[3], xm, cuT, cu;
  double ampx, ampu, accel, vel, ampxdot, maxu;
  double rkt[2][4], rk[4];
  /*
   * [y, u, t] = DigStep(bearing, b, damp, freq, T, initx, initu, maxt)
   *     bearing - vacuum pump bearing number
   *               0 = axial bearing
   *               1 = radial 1X bearing
   *               2 = radial 1Y bearing
   *               3 = radial 2X bearing
   *               4 = radial 2Y bearing
   *     b       - controller gain (b hat)
   *     damp    - controller damping ratio
   *     freq    - controller natural frequency
   *     T       - controller sampling interval
   *     initx   - initial position
   *     initu   - initial control
   *     maxt    - maximum response time
   */

  T1 = 1.0/T;
  a22 = 2.0 * damp * freq;
  a21 = freq * freq;
  g = 9.807;
  divs = 10.0;
  rT = T1 / divs;
  (*u) = (*y) = (*t) = NULL;
  (*n) = 0;

  ttsize = ((int) ((maxt / T1) + 0.5)) + 1;

  if((tt = (double *) calloc(ttsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigStep.\n");
      return(1);
    }

  uptsize = ((int) ((maxt / rT) + 0.5)) + 1;

  if((upt = (double *) calloc(uptsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigStep.\n");
      free(tt);
```

```
      return(1);
    }

  if((ty = (double *) calloc(uptsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigStep.\n");
      free(tt);
      free(upt);
      return(1);
    }

  if((tu = (double *) calloc(uptsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigStep.\n");
      free(tt);
      free(upt);
      free(ty);
      return(1);
    }

  for(i=1; i<uptsize; i++)
    upt[i] = upt[i-1] + rT;

  for(j=0,i=10; i<uptsize; i+=10)
    tt[j++] = upt[i];

  for(i=0; i<5; i++)
    cx[i] = initx;

  if(!bearing)
    {
      maxx = 0.0002;
      maxu = 3.0;
    }
  else
    {
      maxx = 0.0001;
      maxu = 2.0;
    }

  px[1] = pxdot[0] = pxdot[1] = 0.0;
  px[0] = initx;

#ifdef BACKWARD
  am[0] = 3.0; am[1] = -4.0; am[2] = -2.0; am[3] = 4.0; am[4] = -1.0;
#else
  am[0] = 1.0; am[1] = 0.0; am[2] = -2.0; am[3] = 0.0; am[4] = 1.0;
#endif
  for(i=0; i<5; i++)
    am[i] /= (4.0 * (T1 * T1));

#ifdef BACKWARD
  vm[0] = 3.0 * a22; vm[1] = -4.0 * a22; vm[2] = a22;
#else
  vm[0] = a22; vm[1] = 0.0; vm[2] = -a22;
#endif
  for(i=0; i<3; i++)
    vm[i] /= (2.0 * T1);
  xm = a21;
  cuT = cu = initu;

  ampx = ampu = initu;

  rk[0] = rk[3] = 1.0/6.0;
  rk[1] = rk[2] = 1.0/3.0;

  for(j=i=0; i<uptsize; i++)
    {
      if(upt[i] == tt[j])
```

```
  {
    /* update controller position variables */
    for(k=4; k>0; k--)
      cx[k] = cx[k-1];
    cx[0] = px[0];
    /* determine control signal */
    for(accel=0.0,k=0; k<5; k++)
      accel += (am[k] * cx[k]);
    for(vel=0.0,k=0; k<3; k++)
      vel += (vm[k] * cx[k]);
    cu = cuT - ((accel + vel + (xm*cx[0]))/b);
    if(cu > maxu)
      cu = maxu;
    else if(cu < -maxu)
      cu = -maxu;
    cuT = cu;
    j++;
  }

/* determine amplified control */
rkt[0][0] = (-A2*ampx) + (A1*cu);          /* f{y(j),t(j)}              */
ampxdot = ampx + (rkt[0][0]*(rT/2));       /* y*(j+1/2)                 */
rkt[0][1] = (-A2*ampxdot) + (A1*cu);       /* f{y*(j+1/2),t(j+1/2)}     */
ampxdot = ampx + (rkt[0][1]*(rT/2));       /* y**(j+1/2)                */
rkt[0][2] = (-A2*ampxdot) + (A1*cu);       /* f{y**(j+1/2),t(j+1/2)}    */
ampxdot = ampx + (rkt[0][2]*rT);           /* y*(j+1)                   */
rkt[0][3] = (-A2*ampxdot) + (A1*cu);       /* f{y*(j+1),t(j+1)}         */
for(ampxdot=0.0,k=0; k<4; k++)
  ampxdot += (rkt[0][k]*rk[k]);
ampx += (ampxdot * rT);
ampu = ampx;

/* determine new position */
rkt[0][0] = px[1];                         /* f{y(j),t(j)}              */
rkt[1][0] = (Q*px[0]) + (P*ampu);          /* f{y(j),t(j)}              */
pxdot[0] = px[0] + (rkt[0][0]*(rT/2));     /* y*(j+1/2)                 */
pxdot[1] = px[1] + (rkt[1][0]*(rT/2));     /* y*(j+1/2)                 */
rkt[0][1] = pxdot[1];                      /* f{y*(j+1/2),t(j+1/2)}     */
rkt[1][1] = (Q*pxdot[0]) + (P*ampu);       /* f{y*(j+1/2),t(j+1/2)}     */
pxdot[0] = px[0] + (rkt[0][1]*(rT/2));     /* y**(j+1/2)                */
pxdot[1] = px[1] + (rkt[1][1]*(rT/2));     /* y**(j+1/2)                */
rkt[0][2] = pxdot[1];                      /* f{y**(j+1/2),t(j+1/2)}    */
rkt[1][2] = (Q*pxdot[0]) + (P*ampu);       /* f{y**(j+1/2),t(j+1/2)}    */
pxdot[0] = px[0] + (rkt[0][2]*rT);         /* y*(j+1)                   */
pxdot[1] = px[1] + (rkt[1][2]*rT);         /* y*(j+1)                   */
rkt[0][3] = pxdot[1];                      /* f{y*(j+1),t(j+1)}         */
rkt[1][3] = (Q*pxdot[0]) + (P*ampu);       /* f{y*(j+1),t(j+1)}         */
pxdot[0] = pxdot[1] = 0.0;
for(k=0; k<4; k++)
  {
    pxdot[0] += (rkt[0][k]*rk[k]);
    pxdot[1] += (rkt[1][k]*rk[k]);
  }
if(bearing == 0)
  pxdot[1] -= g;

/* save values for plotting */
ty[i] = px[0];
tu[i] = cu;

/* update variables */
px[0] += (pxdot[0] * rT);
px[1] += (pxdot[1] * rT);

if(px[0] > maxx)
  px[0] = maxx;
else if(px[0] < -maxx)
  px[0] = -maxx;
```

```
    }

  free(tt);

  (*y) = ty;
  (*u) = tu;
  (*t) = upt;
  (*n) = uptsize;

  return(0);

}


int DigBode(double *num, double *den, int numsize, int densize, double ts,
            double begfreq, double endfreq, double **mag, double **freq,
            int *magsize)
{
  int i, j, sampsize, dec;
  double *mg, *fq, inc, t;
  double frq, ansr, ansi;

  t = 1.0/ts;
  dec = (int) (log10(endfreq) - log10(begfreq));
  sampsize = dec * 81;
  sampsize += 2;

  if((mg = (double *) calloc(sampsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigBode.\n");
      return(1);
    }

  if((fq = (double *) calloc(sampsize, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Out of memory error - DigBode.\n");
      free(mg);
      return(1);
    }

  inc = begfreq / 10.0;
  fq[0] = begfreq;
  for(i=0; i<dec; i++)
    {
      for(j=0; j<81; j++)
        fq[(i*81)+j+1] = fq[(i*81)+j] + inc;
      inc *= 10.0;
    }
  fq[sampsize-1] = endfreq;

  for(i=0; i<sampsize; i++)
    {
      frq = 2.0 * PI * fq[i];
      ansr = ansi = 0.0;
      mg[i] = 20.0 * log10(num[0]);
      for(j=1; j<numsize; j++)
        {
          if(num[j] != 0.0)
            {
              inc = frq * t * (numsize - 1 - j);
              ansr += (num[j] * cos(inc));
              ansi += (num[j] * sin(inc));
            }
        }
      mg[i] += (20.0 * log10(sqrt(pow(ansr,2.0) + pow(ansi,2.0))));
      ansr = ansi = 0.0;
      for(j=0; j<densize; j++)
        {
          if(den[j] != 0.0)
```

```
            {
              inc = frq * t * (densize - 1 - j);
              ansr += (den[j] * cos(inc));
              ansi += (den[j] * sin(inc));
            }
        }
      mg[i] -= (20.0 * log10(sqrt(pow(ansr,2.0) + pow(ansi,2.0))));
    }

  if(mag)
    (*mag) = mg;
  if(freq)
    (*freq) = fq;
  if(magsize)
    (*magsize) = sampsize;

  return(0);

}
```

## H.1.9 comp2text.c

This program reads the binary file produced by compli.c and outputs the values of the damping ratio, natural frequency, feedback gain, gain at 1000 Hz, bandwidth, maximum compliance, maximum steady state control signal, and maximum closed loop gain to a text file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define MEMERR(x) \
  fprintf(stderr, "Out of memory error - %s (%d)\n", x, __LINE__);
#define FUNCERR(x,y) \
  fprintf(stderr, "Error encountered in %s - %s (%d)\n", x, y, __LINE__);

#define STATUS_OK  0
#define STATUS_NOK 1
#ifdef NOISE
#ifdef CLOSE_TEST
#define SAVENUM     5
#else
#define SAVENUM     4
#endif
#else
#define SAVENUM     3
#endif


void main(int argc, char **argv)
{
  int i,j;
  int begrow, begcol, rows, cols, mag;
  int matrows, matcols;
  int bhat_size, damp_size, freq_size, ebara, axis;
  char *infile, *outfile;
  FILE *ifp, *ofp;
  float freq, damp, bhat, *buffer;
  float bbhat, ebhat, bdamp, edamp, bfreq, efreq;

  begrow = begcol = rows = cols = mag = axis = 0;
  ebara = 0;
  infile = outfile = NULL;
```

```
if(argc < 2)
  {
    fprintf(stderr, "syntax error ...\n\n");
    fprintf(stderr, "Syntax:  comp2text infile [outfile] [-r n] [-c m]");
    fprintf(stderr, " [-nr l] [-nc k] [-a x]\n");
    fprintf(stderr, "         where:  infile  -> matrix pathname\n");
    fprintf(stderr, "                 outfile -> text file pathname\n");
    fprintf(stderr, "                 -r  n   -> start at nth row\n");
    fprintf(stderr, "                 -c  m   -> start at mth column\n");
    fprintf(stderr, "                 -nr l   -> convert l rows\n");
    fprintf(stderr, "                 -nc k   -> convert k columns\n");
    fprintf(stderr, "                 -a  x   -> bearing axis (Def=0)\n");
    exit(1);
  }

for(i=1; i<argc; i++)
  {
    if(argv[i][0] == '-')
      {
        if(argv[i][1] == 'r')
          begrow = atoi(argv[++i]);
        else if(argv[i][1] == 'c')
          begcol = atoi(argv[++i]);
        else if(argv[i][1] == 'a')
          axis = atoi(argv[++i]);
        else if(argv[i][1] == 'n')
          {
            if(argv[i][2] == 'r')
              rows = atoi(argv[++i]);
            else if(argv[i][2] == 'c')
              cols = atoi(argv[++i]);
          }
      }
    else if(infile == NULL)
      infile = argv[i];
    else if(outfile == NULL)
      outfile = argv[i];
  }

if((ifp = fopen(infile, "r")) == NULL)
  {
    fprintf(stderr, "Unable to open input file \"%s\"\n\n", infile);
    exit(1);
  }

if(outfile)
  {
    if((ofp = fopen(outfile, "w")) == NULL)
      {
        fprintf(stderr, "Unable to open output file \"%s\"\n\n", outfile);
        exit(1);
      }
  }
else
  ofp = stdout;

if(!fread((void *) &matrows, sizeof(int), 1, ifp))
  {
    fprintf(stderr, "Error encountered reading from \"%s\"\n\n", infile);
    fclose(ifp);
    if(outfile)
      fclose(ofp);
    exit(1);
  }

if(!fread((void *) &matcols, sizeof(int), 1, ifp))
  {
    fprintf(stderr, "Error encountered reading from \"%s\"\n\n", infile);
    fclose(ifp);
```

```
    if(outfile)
      fclose(ofp);
    exit(1);
  }

if(!rows)
  rows = matrows;

if(!cols)
  cols = matcols;

if(!mag)
  mag = 1;

if(begrow+rows > matrows)
  {
    fprintf(stderr,
            "Requested rows (%d) is greater than matrix rows (%d)\n\n",
            begrow+rows, matrows);
    fclose(ifp);
    if(outfile)
      fclose(ofp);
    exit(1);
  }

if(begcol+cols > matcols)
  {
    fprintf(stderr,
            "Requested columns (%d) is greater than matrix columns (%d)\n\n",
            begcol+cols, matcols);
    fclose(ifp);
    if(outfile)
      fclose(ofp);
    exit(1);
  }

if(begrow)
  fseek(ifp, begrow*matcols*SAVENUM, SEEK_CUR);

if((buffer = (float *) calloc(matcols*SAVENUM, sizeof(float))) == NULL)
  {
    MEMERR("mat2text");
    fclose(ifp);
    if(outfile)
      fclose(ofp);
    exit(1);
  }

if(ebara)
  {
    switch(axis)
      {
      default:
      case 0:
        bdamp = floor((4.033827 - 2.101680) * 10.0) / 10.0;
        edamp = ceil((4.033827 + 2.101680) * 10.0) / 10.0;
        bbhat = floor(402.967777 - 254.562878);
        ebhat = ceil(402.967777 + 254.562878);
        bfreq = floor((527.319758 - 238.103058) / 10.0) * 10.0;
        efreq = ceil((527.319758 + 238.103058) / 10.0) * 10.0;
        break;
      case 1:
        edamp = ceil((4.033827 + 2.101680) * 10.0) / 10.0;
        bbhat = floor(402.967777 - 254.562878);
        ebhat = ceil(402.967777 + 254.562878);
        bfreq = floor((527.319758 - 238.103058) / 10.0) * 10.0;
        efreq = ceil((527.319758 + 238.103058) / 10.0) * 10.0;
        break;
      case 2:
```

```
        bdamp = floor((4.033827 - 2.101680) * 10.0) / 10.0;
        edamp = ceil((4.033827 + 2.101680) * 10.0) / 10.0;
        bbhat = floor(402.967777 - 254.562878);
        ebhat = ceil(402.967777 + 254.562878);
        bfreq = floor((527.319758 - 238.103058) / 10.0) * 10.0;
        efreq = ceil((527.319758 + 238.103058) / 10.0) * 10.0;
        break;
      case 3:
        bdamp = floor((4.033827 - 2.101680) * 10.0) / 10.0;
        edamp = ceil((4.033827 + 2.101680) * 10.0) / 10.0;
        bbhat = floor(402.967777 - 254.562878);
        ebhat = ceil(402.967777 + 254.562878);
        bfreq = floor((527.319758 - 238.103058) / 10.0) * 10.0;
        efreq = ceil((527.319758 + 238.103058) / 10.0) * 10.0;
        break;
      case 4:
        bdamp = floor((4.033827 - 2.101680) * 10.0) / 10.0;
        edamp = ceil((4.033827 + 2.101680) * 10.0) / 10.0;
        bbhat = floor(402.967777 - 254.562878);
        ebhat = ceil(402.967777 + 254.562878);
        bfreq = floor((527.319758 - 238.103058) / 10.0) * 10.0;
        efreq = ceil((527.319758 + 238.103058) / 10.0) * 10.0;
        break;
      }
  }
else
  {
    switch(axis)
      {
      default:
      case 0:
        bdamp = floor((3.272777 - (1.25 * 2.717406)) * 10.0) / 10.0;
        edamp = ceil((3.272777 + (1.25 * 2.717406)) * 10.0) / 10.0;
        bbhat = floor(370.414059 - (1.25 * 271.290529));
        ebhat = ceil(370.414059 + (1.25 * 271.290529));
        bfreq = floor((347.082893 - (1.25 * 232.298166)) / 10.0) * 10.0;
        efreq = ceil((347.082893 + (1.25 * 232.298166)) / 10.0) * 10.0;
        break;
      case 1:
        bdamp = floor((3.918438 - (1.25 * 2.817466)) * 10.0) / 10.0;
        edamp = ceil((3.918438 + (1.25 * 2.817466)) * 10.0) / 10.0;
        bbhat = floor(328.021404 - (1.25 * 279.017239));
        ebhat = ceil(328.021404 + (1.25 * 279.017239));
        bfreq = floor((276.262071 - (1.25 * 209.387555)) / 10.0) * 10.0;
        efreq = ceil((276.262071 + (1.25 * 209.387555)) / 10.0) * 10.0;
        break;
      case 2:
        bdamp = floor((3.814237 - (1.25 * 2.808301)) * 10.0) / 10.0;
        edamp = ceil((3.814237 + (1.25 * 2.808301)) * 10.0) / 10.0;
        bbhat = floor(332.704460 - (1.25 * 278.550625));
        ebhat = ceil(332.704460 + (1.25 * 278.550625));
        bfreq = floor((285.769852 - (1.25 * 212.645306)) / 10.0) * 10.0;
        efreq = ceil((285.769852 + (1.25 * 212.645306)) / 10.0) * 10.0;
        break;
      case 3:
        bdamp = floor((3.412286 - (1.25 * 2.731543)) * 10.0) / 10.0;
        edamp = ceil((3.412286 + (1.25 * 2.731543)) * 10.0) / 10.0;
        bbhat = floor(349.211106 - (1.25 * 273.471228));
        ebhat = ceil(349.211106 + (1.25 * 273.471228));
        bfreq = floor((336.787694 - (1.25 * 226.591365)) / 10.0) * 10.0;
        efreq = ceil((336.787694 + (1.25 * 226.591365)) / 10.0) * 10.0;
        break;
      case 4:
        bdamp = floor((3.550881 - (1.25 * 2.756218)) * 10.0) / 10.0;
        edamp = ceil((3.550881 + (1.25 * 2.756218)) * 10.0) / 10.0;
        bbhat = floor(343.223786 - (1.25 * 275.796596));
        ebhat = ceil(343.223786 + (1.25 * 275.796596));
        bfreq = floor((335.875802 - (1.25 * 225.363705)) / 10.0) * 10.0;
        efreq = ceil((335.875802 + (1.25 * 225.363705)) / 10.0) * 10.0;
```

```
          break;
        }
    }

  if(bdamp < 0.0)
    bdamp = 0.1;
  if(bbhat < 0.0)
    bbhat = 2.0;
  if(bfreq < 0.0)
    bfreq = 10.0;

  bhat_size = ((int) ceil((ebhat - bbhat) / 2.0)) + 1;
  freq_size = ((int) ceil((efreq - bfreq) / 10.0)) + 1;
  damp_size = ((int) ceil((edamp - bdamp) / 0.1)) + 1;

  for(i=0; i<rows; i++)
    {
      if(fread((void *) buffer, sizeof(float), matcols*SAVENUM, ifp)
         != matcols*SAVENUM)
        {
          fprintf(stderr, "Data read error ...\n");
          fclose(ifp);
          if(outfile)
            fclose(ofp);
          exit(1);
        }
      /*
       * i = bhat, j = freq, l = damp
       * buffer[(j*SAVENUM*damp_size)+(l*SAVENUM)] = 0.0;
       * buffer[(j*SAVENUM*damp_size)+(l*SAVENUM)+1] = (float) -1.0;
       * buffer[(j*SAVENUM*damp_size)+(l*SAVENUM)+2] = (float) 0.0;
       */
      for(j=begcol; j<begcol+cols; j++)
        {
          if(buffer[(j*SAVENUM)+1] > 0.0)
            {
              bhat = (bbhat + (((begrow+i)/freq_size) * 2.0));
              freq = (bfreq + (((begrow+i)%freq_size) * 10.0));
              damp = (bdamp + (j * 0.1));
#ifdef NOISE
#ifdef CLOSE_TEST
              fprintf(ofp, "%5.2f  %6.1f  %6.1f  %6.2f  %6.2f  %6.2f  %6.4f  %5.2f\n",
                      damp, freq, bhat, buffer[(j*SAVENUM)],
                      buffer[(j*SAVENUM)+1], buffer[(j*SAVENUM)+2],
                      buffer[(j*SAVENUM)+3], buffer[(j*SAVENUM)+4]);
#else
              fprintf(ofp, "%5.2f  %6.1f  %6.1f  %6.2f  %6.2f  %6.2f  %6.4f\n",
                      damp, freq, bhat, buffer[(j*SAVENUM)],
                      buffer[(j*SAVENUM)+1], buffer[(j*SAVENUM)+2],
                      buffer[(j*SAVENUM)+3]);
#endif
#else
              fprintf(ofp, "%5.2f  %6.1f  %6.1f  %6.2f  %6.2f  %6.2f\n",
                      damp, freq, bhat, buffer[(j*SAVENUM)],
                      buffer[(j*SAVENUM)+1], buffer[(j*SAVENUM)+2]);
#endif
            }
        }
    }

  fclose(ifp);
  if(outfile)
    fclose(ofp);
  exit(0);

}
```

# H.2 System Response Programs

The section lists the programs that display the actual and theoretical system responses. The actual closed loop and disturbance rejection responses are plotted from data obtained from the system analyzer. The theoretical responses are determined from the best fit model and the parameters of bearing number, sampling interval, damping ratio, natural frequency, and feedback gain provided by the user.

## H.2.1 PrtAllData.m

The Matlab script plots the actual and theoretical system responses using the data obtained from the system analyzer and the best fit system model. All of the responses are plotted on the same graph to facilitate easy comparison.

```
function [] = PrtAllData(w, units)

if(nargin < 1 | nargin > 2)
  disp('Syntax error ...');
  disp('PrtAllData(which, units)');
  disp('where:  which - what data to print');
  disp('                ''DIST-30''  Digital Disturbance plot (30000 rpm)');
  disp('                ''DIST-15''  Digital Disturbance plot (15000 rpm)');
  disp('                ''DIST''     Digital Disturbance plot (static)');
  disp('                ''CLOOP''    Closed-loop plot (static)');
  disp('                ''STARTUP''  Initial startup plot (static)');
  disp('                ''ALL''      All of the above plots');
  disp('        units - what units to use on y axis');
  disp('                ''VOLTS''    raw system analyzer plot (default)');
  disp('                ''FORCE''    microns/Newton');
  error;
end

if(~isstr(w))
  disp('Syntax error ...');
  disp('PrtAllData(which, units)');
  disp('where:  which - what data to print');
  disp('                ''DIST-30''  Digital Disturbance plot (30000 rpm)');
  disp('                ''DIST-15''  Digital Disturbance plot (15000 rpm)');
  disp('                ''DIST''     Digital Disturbance plot (static)');
  disp('                ''CLOOP''    Closed-loop plot (static)');
  disp('                ''STARTUP''  Initial startup plot (static)');
  disp('                ''ALL''      All of the above plots');
  disp('        units - what units to use on y axis');
  disp('                ''VOLTS''    raw system analyzer plot (default)');
  disp('                ''FORCE''    microns/Newton');
  error;
end

if(nargin == 2)
  if(~isstr(units))
    disp('Syntax error ...');
    disp('PrtAllData(which, units)');
    disp('where:  which - what data to print');
    disp('                ''DIST-30''  Digital Disturbance plot (30000 rpm)');
    disp('                ''DIST-15''  Digital Disturbance plot (15000 rpm)');
    disp('                ''DIST''     Digital Disturbance plot (static)');
    disp('                ''CLOOP''    Closed-loop plot (static)');
    disp('                ''STARTUP''  Initial startup plot (static)');
```

```
    disp('                ''ALL''      All of the above plots');
    disp('        units - what units to use on y axis');
    disp('                ''VOLTS''    raw system analyzer plot (default)');
    disp('                ''FORCE''    microns/Newton');
    error;
  end
else
  units = 'volts';
end

printme = 0;
myprinter = ' -Phayden';
temp_file = ' /tmp/matplot';
avg_len = 8000;
startup_pts = 4000;
sengain = [9450.0 25000.0 25000.0 25000.0 25000.0];    % volts/meter
ampgain = [2.776 1.175 1.165 1.159 1.144];             % amps/volt
meter_conv = (2.5)./(8192*sengain);
offset = [1 1 1 1 1; 3015 3293 4247 3238 3525];
bhat = [170 160 180 260 260];
freq = [100 130 110 100 100];
damp = [0.85 1.0 1.05 1.90 1.80];
T = 1/10000;
wa = zeros(1,5);
mass = 2.2;

w = lower(deblank(w));
units = lower(deblank(units));

if(strcmp(units,'force'))
  distconv = [];
  for ii=1:5
    [dir_num, file_num] = ConPumpNoAmp(ii-1);
    distconv = [distconv (1e6 / (dir_num * mass * sengain(ii)))];
  end
elseif(~strcmp(units, 'volts'))
  disp(['unknown units: ' units]);
  error;
end

cmpstrs = str2mat('dist-30', 'dist-15', 'dist', 'cloop', 'startup', 'all');

dirs = str2mat('/usr/tmp/disturb/digital/dynamic/rpm30', ...
    '/usr/tmp/disturb/analog/dynamic/rpm30', ...
    '/usr/tmp/disturb/digital/dynamic/rpm15', ...
    '/usr/tmp/disturb/analog/dynamic/rpm15', ...
    '/usr/tmp/disturb/digital/static', ...
    '/usr/tmp/disturb/analog/static', ...
    '/usr/tmp/closloop/digital', '/usr/tmp/closloop/analog', ...
    '/usr/tmp/startup/digital', '/usr/tmp/startup/analog');
dir_num = 2;

file_com = str2mat('axialmg.dat', 'rad1xmg.dat', 'rad1ymg.dat', ...
    'rad2xmg.dat', 'rad2ymg.dat');
file_diff = str2mat('axial.in', 'rad1x.in', 'rad1y.in', 'rad2x.in', ...
    'rad2y.in', ...
    'axial.in', 'rad1x.in', 'rad1y.in', 'rad2x.in', 'rad2y.in');
files = str2mat(file_com, file_com, file_com, file_com, file_com, ...
    file_com, file_com, file_com, file_diff);
file_num = 5;

legend_str = str2mat('Digital', 'Analog', 'Theory');
line_type = str2mat('g-', 'r-.', 'y.');
axis_str = str2mat('Axial', 'Rad1X', 'Rad1Y', 'Rad2X', 'Rad2Y');
title_str = str2mat('Dynamic Disturbance Rejection at 28000 rpm', ...
    'Dynamic Disturbance Rejection at 15000 rpm', ...
    'Static Disturbance Rejection', ...
    'Static Closed Loop Response', ...
    'Static Time Response');
```

```
x_labels = str2mat('Frequency (Hz)', 'Frequency (Hz)', 'Frequency (Hz)', ...
    'Frequency (Hz)', 'Time (secs)');
if(strcmp(units,'volts'))
  y_labels = str2mat('Compliance (Volts/Volts)', ...
      'Compliance (Volts/Volts)', 'Compliance (Volts/Volts)', ...
      'Gain', 'Position (meters)');
else
  y_labels = str2mat('Compliance (Microns/Newton)', ...
      'Compliance (Microns/Newton)', 'Compliance (Microns/Newton)', ...
      'Gain', 'Position (meters)');
end

for ii=1:size(cmpstrs,1)-1
  if(strcmp(w,deblank(cmpstrs(ii,:))))
    wa(ii) = 1;
  end
end

if(strcmp(w,deblank(cmpstrs(size(cmpstrs,1),:))))
  wa = ones(1,5);
end

if(~any(wa))
  disp('nothing to do ...');
  return;
end

more off;

for ii=1:5
  if(wa(ii) == 1)
    for jj=1:file_num
      hold off;
      for kk=1:dir_num
        tstr = [deblank(dirs(((ii-1)*dir_num)+kk,:)) '/' ...
                deblank(files(((ii-1)*dir_num*file_num)+ ...
                ((kk-1)*file_num)+jj,:))];
        fid = fopen(tstr, 'r');
        if(fid > 2)
          dat = [];
          if(ii < 5)    % log plots
            [dat] = fscanf(fid, '%f %f', [2 inf]);
          else          % time plots
            [avg] = fscanf(fid, '%d', [1 inf]);
            dat(1,:) = [0:T:(length(avg)-1)*T];
            dat(2,:) = avg;
            avg = mean(dat(2,size(dat,2)-avg_len:size(dat,2)));
            dat(2,:) = dat(2,:)-avg;    % eliminate A/D offset
          end
          fclose(fid);
          % ------------- plot data
          if(ii < 5)
            if(ii < 4 & strcmp(units,'force'))
              loglog(dat(1,:),distconv(jj)*(10.^(dat(2,:)/20)), ...
                  deblank(line_type(kk,:)));
            else
              semilogx(dat(1,:),dat(2,:),deblank(line_type(kk,:)));
            end
          else
            plot(dat(1,1:startup_pts), ...
                dat(2,offset(kk,jj):offset(kk,jj)+startup_pts-1)* ...
                meter_conv(jj),deblank(line_type(kk,:)));
          end
          hold on;
        else
          disp(['unable to open file "' tstr '"']);
        end
      end
      if(ii == 3)         % Static Disturbance rejection
```

```
        [m,h] = DigPlotComp(jj-1,bhat(jj),damp(jj),freq(jj),T,units);
        if(strcmp(units,'force'))
          loglog(h,m,deblank(line_type(3,:)));
        else
          semilogx(h,20*log10(m),deblank(line_type(3,:)));
        end
      elseif(ii == 4)      % Static Closed Loop response
        [m,p,h] = DigClosePlot(jj-1,bhat(jj),damp(jj),freq(jj),T);
        semilogx(h,20*log10(m),deblank(line_type(3,:)));
      elseif(ii == 5)      % Startup Initial Condition response
      end
      title([deblank(axis_str(jj,:)) ' ' deblank(title_str(ii,:))]);
      xlabel(deblank(x_labels(ii,:)));
      ylabel(deblank(y_labels(ii,:)));
      if(ii > 2)
        legend(deblank(line_type(1,:)), deblank(legend_str(1,:)), ...
            deblank(line_type(2,:)), deblank(legend_str(2,:)), ...
            deblank(line_type(3,:)), deblank(legend_str(3,:)));
      else
        legend(deblank(line_type(1,:)), deblank(legend_str(1,:)), ...
            deblank(line_type(2,:)), deblank(legend_str(2,:)));
      end
      if(printme == 1)
        orient landscape;
        tstr = ['print -dps' temp_file];
        eval(tstr);
        tstr = ['!lpr -h' myprinter temp_file '.ps'];
        eval(tstr);
      end
      hold off;
      disp('Press any key to continue ...');
      pause;
    end
  end
end

more on;
```

## H.2.2 DigClosePlot.m

This Matlab script returns the magnitude, phase, and frequency range of the closed loop frequency response of the system given the bearing number, feedback gain, damping ratio, natural frequency, and sampling interval. If no output arguments are provided, the closed loop frequency response is plotted instead.

```
function [mag, phase, w] = DigClosePlot(bearing, bp,drat,freq,samp)
% Plots closed loop Bode plot

if(nargin < 5)
  disp('Four input arguments required.');
  disp('Syntax:  [mag, phase, w] = DigClosePlot(bearing, B,rat,freq,samp)');
  disp('    bearing - vacuum pump bearing number');
  disp('              0 = axial bearing');
  disp('              1 = radial 1X bearing');
  disp('              2 = radial 1Y bearing');
  disp('              3 = radial 2X bearing');
  disp('              4 = radial 2Y bearing');
  disp('         B    - control function divisor');
  disp('         rat  - controller damping ratio');
  disp('         freq - controller natural frequency');
  disp('         samp - controller sampling interval');
  error;
end;
```

```
ebara = 0;

if(ebara ~= 1)
  [opnum, opden] = ConPumpIdealNoAmp(bearing);
  [ampnum, ampden] = ConAmpIdeal(bearing);
else
  [opnum, opden] = ConPumpNoAmp(bearing);
  [ampnum, ampden] = ConAmp(bearing);
end

P = opnum;
Q = abs(opden(length(opden)));

A1 = ampnum;
A2 = abs(ampden(length(ampden)));

a21 = freq^2;
a22 = 2*drat*freq;

[J1,J2,J3,J4,J5,M1,M2,M3,M4,N1,N2,N3,N4] = ...
    DigStabFunc(A1,A2,P,Q,samp,a22,a21);

clnum = [(J1*M2) (J1*M3)+(J2*M2) (J1*M4)+(J2*M3)+(J3*M2) ...
         (J2*M4)+(J3*M3)+(J4*M2) (J3*M4)+(J4*M3)+(J5*M2) (J4*M4)+(J5*M3) ...
         (J5*M4)];

clden = [bp (J1*M2)+(bp*(N2-1)) (J1*M3)+(J2*M2)+(bp*(N3-N2)) ...
         (J1*M4)+(J2*M3)+(J3*M2)+(bp*(N4-N3)) ...
         (J2*M4)+(J3*M3)+(J4*M2)-(bp*N4) (J3*M4)+(J4*M3)+(J5*M2) ...
         (J4*M4)+(J5*M3) (J5*M4)];

begin = 0.1;
endd = 1000.0;
ind = begin;
range = [begin];
while(ind < endd)
  inc = ind/10.0;
  range = [range [ind+inc:inc:ind*10.0]];
  ind = ind * 10.0;
end

[tmag, tphase] = dbode(clnum,clden,samp,2*pi*range);

if(nargout == 0)
  if(bearing == 0)
    fid = fopen('/usr/tmp/closloop/analog/axialmg.dat', 'r');
    mytitle = 'Axial';
  elseif(bearing == 1)
    fid = fopen('/usr/tmp/closloop/analog/rad1xmg.dat', 'r');
    mytitle = 'Rad1X';
  elseif(bearing == 2)
    fid = fopen('/usr/tmp/closloop/analog/rad1ymg.dat', 'r');
    mytitle = 'Rad1Y';
  elseif(bearing == 3)
    fid = fopen('/usr/tmp/closloop/analog/rad2xmg.dat', 'r');
    mytitle = 'Rad2X';
  elseif(bearing == 4)
    fid = fopen('/usr/tmp/closloop/analog/rad2ymg.dat', 'r');
    mytitle = 'Rad2Y';
  end
  if(fid > 0)
    [dat, ind] = fscanf(fid, '%f %f', [2 inf]);
    fclose(fid);
    semilogx(dat(1,:), dat(2,:),'-');
    hold on;
    semilogx(range, 20*log10(tmag),'--');
    hold off;
    xlabel('Frequency (Hz)');
    ylabel('Magnitude (dB)');
```

```
    title([mytitle ' Bearing Close Loop Bode Plot  (B = ' num2str(bp) ...
          ', Nat Freq = ' num2str(freq) ', Damp Ratio = ' num2str(drat) ')']);
    [m, ind] = max(dat(2,:));
    band = min(find(dat(2,min(ind):size(dat,2))<0));
    band = dat(1,band+min(ind));
    e = min(find(dat(1,min(ind):size(dat,2))>=1000));
    e = dat(2,e+min(ind));
    disp(['Analog:  max = ' num2str(m) ', bandwidth = ' num2str(band) ...
           ', final = ' num2str(e)]);
    [m, ind] = max(tmag);
    m = 20*log10(m);
    band = min(find(20*log10(tmag(min(ind):length(tmag)))<0));
    band = range(band+ind);
    e = min(find(range(min(ind):length(range))>=1000));
    e = 20*log10(tmag(min([e+min(ind) length(tmag)])));
    disp(['Digital:  max = ' num2str(m) ', bandwidth = ' num2str(band) ...
           ', final = ' num2str(e)]);
  else
    semilogx(range, 20*log10(tmag),'-');
    xlabel('Frequency (Hz)');
    ylabel('Magnitude (dB)');
    title([mytitle ' Bearing Close Loop Bode Plot  (B = ' num2str(bp) ...
          ', Nat Freq = ' num2str(freq) ', Damp Ratio = ' num2str(drat) ')']);
    [m, ind] = max(tmag);
    m = 20*log10(m);
    band = min(find(20*log10(tmag(min(ind):length(tmag)))<0));
    band = range(band+ind);
    e = min(find(range(min(ind):length(range))>=1000));
    e = 20*log10(tmag(e+min(ind)));
    disp(['max = ' num2str(m) ', bandwidth = ' num2str(band) ', final = ', ...
           num2str(e)]);
  end
elseif(nargout == 1)
  mag = tmag;
elseif(nargout == 2)
  mag = tmag;
  phase = tphase;
elseif(nargout == 3)
  mag = tmag;
  phase = tphase;
  w = range;
end
```

## H.2.3 DigPlotComp.m

This Matlab script returns the magnitude and frequency range of the disturbance rejection frequency response of the system given the bearing number, feedback gain, damping ratio, natural frequency, and sampling interval. If no output arguments are provided, the disturbance rejection frequency response is plotted instead.

```
function [mag,w] = DigPlotComp(bearing, B, damp, freq, samp, units)
% plot compliance transfer function of closed loop system

volt = 0;
if(nargin < 5 | nargin > 6)
  disp('Wrong number of arguments ...');
  disp('[mag] = DigPlotComp(bearing, B, damp, freq, samp, units)');
  disp('where:');
  disp('        bearing - vacuum pump bearing number');
  disp('                  0 = axial bearing');
  disp('                  1 = radial 1X bearing');
  disp('                  2 = radial 1Y bearing');
  disp('                  3 = radial 2X bearing');
```

```
  disp('                    4 = radial 2Y bearing');
  disp('        B       - Controller B hat');
  disp('        damp    - Controller Damping Ratio');
  disp('        freq    - Controller Natural Frequency');
  disp('        samp    - Controller sampling delay');
  disp('        units   - y axis units');
  disp('                  ''volts'' - Volts/Volts (default)');
  disp('                  ''force'' - Microns/Newton');
  error;
elseif(nargin == 6)
  units = lower(deblank(units));
  if(strcmp(units, 'volts'))
    volt = 1;
  elseif(~strcmp(units,'force'))
    disp(['unknown units requested:  ' units]);
    error;
  end
end

voltconv = [9450.0 25000.0 25000.0 25000.0 25000.0];    % volts/meter
ampgain = [2.776 1.175 1.165 1.159 1.144];              % amps/volt
mass = 2.2;

if(bearing == 0)
  fname = '/usr/tmp/disturb/analog/static/axialmg.dat';
  mytitle = 'Axial';
elseif(bearing == 1)
  fname = '/usr/tmp/disturb/analog/static/rad1xmg.dat';
  mytitle = 'Rad1X';
elseif(bearing == 2)
  fname = '/usr/tmp/disturb/analog/static/rad1ymg.dat';
  mytitle = 'Rad1Y';
elseif(bearing == 3)
  fname = '/usr/tmp/disturb/analog/static/rad2xmg.dat';
  mytitle = 'Rad2X';
elseif(bearing == 4)
  fname = '/usr/tmp/disturb/analog/static/rad2ymg.dat';
  mytitle = 'Rad2Y';
else
  disp(['unknown bearing number:  ' num2str(bearing)]);
  error;
end

% flag set if Ebara supplied variables are to be used
ebara = 0;

if(ebara ~= 1)
  [opnum, opden] = ConPumpNoAmp(bearing);
  tP = opnum;
  [opnum, opden] = ConPumpIdealNoAmp(bearing);
  [ampnum, ampden] = ConAmpIdeal(bearing);
  type = 'Ideal';
else
  [opnum, opden] = ConPumpNoAmp(bearing);
  tP = opnum;
  [ampnum, ampden] = ConAmp(bearing);
  type = 'Ebara';
end

P = opnum;
Q = abs(opden(length(opden)));

A1 = ampnum;
A2 = abs(ampden(length(ampden)));

a21 = freq^2;
a22 = 2*damp*freq;

begin = 0.1;
```

```
endd = 1000.0;
ind = begin;
range = [begin];
while(ind < endd)
  inc = ind/10.0;
  range = [range [ind+inc:inc:ind*10.0]];
  ind = ind * 10.0;
end

minmax = 10000;
minparm = zeros(1,4);

num = zeros(1,8);
den = zeros(1,8);

[J1,J2,J3,J4,J5,M1,M2,M3,M4,N1,N2,N3,N4] = DigStabFunc(A1,A2,P,Q,samp,a22,a21);
% check stability first
subtot(1) = 1;
subtot(2) = ((J1*M2)+(J2*M1)+(B*(N2-N1)))/((J1*M1)+(B*N1));
subtot(3) = ((J1*M3)+(J2*M2)+(J3*M1)+(B*(N3-N2)))/((J1*M1)+(B*N1));
subtot(4) = ((J1*M4)+(J2*M3)+(J3*M2)+(J4*M1)+(B*(N4-N3))) ...
    /((J1*M1)+(B*N1));
subtot(5) = ((J2*M4)+(J3*M3)+(J4*M2)+(J5*M1)-(B*N4)) ...
    /((J1*M1)+(B*N1));
subtot(6) = ((J3*M4)+(J4*M3)+(J5*M2))/((J1*M1)+(B*N1));
subtot(7) = ((J4*M4)+(J5*M3))/((J1*M1)+(B*N1));
subtot(8) = (J5*M4)/((J1*M1)+(B*N1));
rt = abs(roots(subtot));
if(isempty(find(rt > 1.0)))
  % Ok its stable so compute compliance
  den(1) = B;
  den(2) = (J1*M2)+(B*(N2-1));
  den(3) = (J1*M3)+(J2*M2)+(B*(N3-N2));
  den(4) = (J1*M4)+(J2*M3)+(J3*M2)+(B*(N4-N3));
  den(5) = (J2*M4)+(J3*M3)+(J4*M2)-(B*N4);
  den(6) = (J3*M4)+(J4*M3)+(J5*M2);
  den(7) = (J4*M4)+(J5*M3);
  den(8) = (J5*M4);
  num(1) = 0;
  num(2) = B*M2;
  num(3) = B*(M3-M2);
  num(4) = B*(M4-M3);
  num(5) = -B*M4;
  num(6:8) = zeros(1,3);
  if(volt == 1)
    num = num * voltconv(bearing+1) * ampgain(bearing+1);
  else
    num = (1e6 * num)/(tP * mass);
    distconv = 1e6 / (tP * mass * voltconv(bearing+1));
  end
  [tmag] = dbode(num,den,samp,2*pi*range);
else
  disp('System unstable:  roots follow');
  for ii=1:length(rt)
    disp(sprintf('%.4f', rt(ii)));
  end
  return;
end

if(nargout == 0)
  fid = fopen(fname, 'r');
  if(fid > 2)
    [dat, ind] = fscanf(fid, '%f %f', [2 inf]);
    fclose(fid);
    if(volt == 1)
      semilogx(dat(1,:), dat(2,:),'g-');
      hold on;
      semilogx(range, 20*log10(tmag), 'r--');
      hold off;
```

```
      ylabel('Magnitude (Volts/Volts)');
      m = max(dat(2,:));
      disp(['Analog:  max = ' num2str(m)]);
      m = max(tmag);
      m = 20*log10(m);
      disp(['Digital:  max = ' num2str(m)]);
    else
      loglog(dat(1,:), distconv*(10.^(dat(2,:)/20)),'g-');
      hold on;
      loglog(range, tmag, 'r--');
      hold off;
      ylabel('Magnitude (microns/Newton)');
      m = max(dat(2,:));
      m = m/20;
      disp(['Analog:  max = ' num2str(m)]);
      m = max(tmag);
      disp(['Digital:  max = ' num2str(log10(m))]);
    end
    title([mytitle ' Bearing Compliance  (B = ' num2str(B) ...
            ', Nat Freq = ' num2str(freq) ...
            ', Damp Ratio = ' num2str(damp) ')']);
    xlabel('Frequency (Hz)');
  else
    if(volt == 1)
      semilogx(range, 20*log10(tmag), 'g-');
      ylabel('Magnitude (Volts/Volts)');
      m = max(tmag);
      m = 20*log10(m);
      disp(['Digital:  max = ' num2str(m)]);
    else
      loglog(range, tmag, '-');
      ylabel('Magnitude (microns/Newton)');
      m = max(tmag);
      disp(['Digital:  max = ' num2str(log10(m))]);
    end
    title([mytitle ' Bearing Compliance  (B = ' num2str(B) ...
          ', Nat Freq = ' num2str(freq) ', Damp Ratio = ' num2str(damp) ')']);
    xlabel('Frequency (Hz)');
  end
elseif(nargout == 1)
  mag = tmag;
else
  mag = tmag;
  w = range;
end
```

## H.2.4 DigStabFunc.m

This Matlab script returns the parameters necessary to compute the appropriate transfer function for both the closed loop system frequency response and the disturbance rejection frequency response. This script was used instead of Matlab's internal functions because rounding errors were producing false poles.

```
function [J1,J2,J3,J4,J5,M1,M2,M3,M4,N1,N2,N3,N4] = ...
        DigStabFunc3(A1,A2,P,Q,T,a22,a21)
% Function that evaluates constants for DigStability

if(nargin ~= 7 | nargout ~= 13)
  disp('Wrong number of arguments ...');
  disp('[J1,J2,J3,J4,J5,M1,M2,M3,M4,N1,N2,N3,N4] = ');
  disp('         DigStabFunc(A1,A2,P,Q,T,a22,a21)');
  error;
end
```

```
[m,n] = size(a22);

if(n ~= 1 & m ~= 1)
  disp('a22 must be scalar');
  error;
end

[m,n] = size(a21);

if(n ~= 1 & m ~= 1)
  disp('a21 must be scalar');
  error;
end

[m,n] = size(T);
sq = sqrt(Q);
if(~isreal(sq))
  sq = abs(sq);
end

J1 = (1/(4.*(T^2)))+(a22/(2.*T))+(a21);
J2 = 0;
J3 = (-2/(4.*(T^2)))-(a22/(2.*T));
J4 = 0;
J5 = 1/(4*(T^2));

%J1 = (3./(4.*(T.^2)))+((3*a22)./(2.*T))+(a21*ones(m,n));
%J2 = (-4./(4.*(T.^2)))-((4*a22)./(2.*T));
%J3 = (-2./(4.*(T.^2)))+(a22./(2.*T));
%J4 = 1./(T.^2);
%J5 = -1./(4.*(T.^2));

K1 = (-A1*P)/(A2*Q);
K2 = (-A1*P)/(A2*((A2^2)-Q));
K3 = (A1*P)/(2*Q*(A2-sq));
K4 = (A1*P)/(2*Q*(A2+sq));

L1 = exp(-A2.*T);
L2 = exp(-sq.*T);
L3 = exp(sq.*T);
L4 = 1;

%M1 = K1+K2+K3+K4;
M1 = zeros(m,n);
M2 = (-K1*(L1+L2+L3))-(K2*(L2+L3+L4))-(K3*(L1+L3+L4))-(K4*(L1+L2+L4));
M3 = (K1*((L1.*L2)+(L3.*(L1+L2))))+(K2*((L2.*L4)+(L3.*(L2+L4))))+ ...
     (K3*((L1.*L4)+(L3.*(L1+L4))))+(K4*((L1.*L4)+(L2.*(L1+L4))));
M4 = (-K1*L1.*L2.*L3)-(K2*L2.*L3.*L4)-(K3*L1.*L3.*L4)-(K4*L1.*L2.*L4);

N1 = ones(m,n);
N2 = -L1-L2-L3;
N3 = (L1.*L2)+(L1.*L3)+(L2.*L3);
N4 = -(L1.*L2.*L3);

return;
```

# H.3 Component Model Programs

This section lists the programs that return the transfer functions of various components of the theoretical model. These components are represented in both their continuous and digital forms.

## H.3.1 ConAmpIdeal.m

The Matlab script returns the numerator and denominator of the transfer function obtained from performing a recursive best fit analysis on the actual driver transfer function.

```
function [ampnum, ampden] = ConAmpIdeal(bearing)
% Continuous model Amp transfer function
% All values were derived using the FindAmpFunc programs

if(nargin == 0)
 disp('One input argument required:  [ampnum, ampden] = ConAmpIdeal(bearing)');
 disp('    bearing - vacuum pump bearing number');
 disp('              0 = axial bearing');
 disp('              1 = radial 1X bearing');
 disp('              2 = radial 1Y bearing');
 disp('              3 = radial 2X bearing');
 disp('              4 = radial 2Y bearing');
 error;
end

if(bearing == 0)
 A1 = 11097.020;
 A2 = 13310.0;
elseif(bearing == 1)
 A1 = 4619.585;
 A2 = 13130.0;
elseif(bearing == 2)
 A1 = 4558.846;
 A2 = 13080.0;
elseif(bearing == 3)
 A1 = 3863.909;
 A2 = 11140.0;
elseif(bearing == 4)
 A1 = 4212.303;
 A2 = 12290.0;
end

anum = A1;
aden = [1 A2];

if(nargout == 0)
 [n, m] = size(anum);
 msg = ['ampnum = [' sprintf('%1.3f',anum(1))];
 for i=2:m
  msg = [msg ' ' sprintf('%1.3f',anum(i))];
 end
 msg = [msg ']'];
 disp(' ');
 disp(msg);
 disp(' ');
 [n, m] = size(aden);
 msg = ['ampden = [' sprintf('%1.3f',aden(1))];
 for i=2:m
  msg = [msg ' ' sprintf('%1.3f',aden(i))];
 end
 msg = [msg ']'];
 disp(msg);
 disp(' ');
else
 ampnum = anum;
 ampden = aden;
end
```

## H.3.2 ConPumpIdealNoAmp.m

The Matlab script returns the numerator and denominator of the transfer function obtained from performing a recursive best fit analysis on the actual turbopump transfer function.

```
function [opnum, opden] = ConPumpIdealNoAmp(bearing)
% linearized ebara pump state space equations

if(nargin == 0)
 disp('One input argument required:  [opnum, opden] = ConPumpIdealNoAmp(bearing)');
 disp('    bearing - vacuum pump bearing number');
 disp('              0 = axial bearing');
 disp('              1 = radial 1X bearing');
 disp('              2 = radial 1Y bearing');
 disp('              3 = radial 2X bearing');
 disp('              4 = radial 2Y bearing');
 error;
end

% Convert from state-space representation to transfer function
if(bearing == 0)
 P = 7.990;
 Q = 22739.569;
elseif(bearing == 1)
 P = 7.123;
 Q = 7737.770;
elseif(bearing == 2)
 P = 8.296;
 Q = 8882.644;
elseif(bearing == 3)
 P = 16.926;
 Q = 35530.576;
elseif(bearing == 4)
 P = 15.113;
 Q = 33201.349;
end

onum = P;
oden = [1 0 -Q];

if(nargout == 0)
 [n, m] = size(onum);
 msg = ['opnum = [' sprintf('%1.3f',onum(1))];
 for i=2:m
  msg = [msg ' ' sprintf('%1.3f',onum(i))];
 end
 msg = [msg ']'];
 disp(' ');
 disp(msg);
 disp(' ');
 [n, m] = size(oden);
 msg = ['opden = [' sprintf('%1.3f',oden(1))];
 for i=2:m
  msg = [msg ' ' sprintf('%1.3f',oden(i))];
 end
 msg = [msg ']'];
 disp(msg);
 disp(' ');
else
 opnum = onum;
 opden = oden;
end
```

## H.3.3 ConPumpNoAmp.m

The Matlab script returns the numerator and denominator of the transfer function obtained from the system parameters provided by the turbopump manufacturer.

```
function [opnum, opden] = ConPumpNoAmp(bearing)
% linearized ebara pump state space equations with no amp included

if(nargin == 0)
 disp('One input argument required:  [opnum, opden] = ConPumpNoAmp(bearing)');
 disp('    bearing - vacuum pump bearing number');
 disp('              0 = axial bearing');
 disp('              1 = radial 1X bearing');
 disp('              2 = radial 1Y bearing');
 disp('              3 = radial 2X bearing');
 disp('              4 = radial 2Y bearing');
 error;
end

% Constants
axlm = 2.2;
radm = 2.2;
a = 0.0238;
b = 0.0691;
irr = 0.008285;
izz = 0.001555;
g = 9.81;
gamma = (12000.0*2.0*pi)/60.0;
io = 0.5;
axlh0 = 0.0004;
radh0 = 0.00025;
axlag = 0.0007;
radag = 0.0000975;
mu = 0.00000126;
% axln = 133.0/2;
axln = 133.0;
radn = 100.0;
angle = (pi / 180) * (45 / 2);

% Calculate often used constants for bearings
C1 = ((1.0/radm) + ((a^2)/irr));
C2 = ((1.0/radm) + ((b^2)/irr));
C3 = ((1.0/radm) - ((a*b)/irr));
C4 = ((a*izz)/((a+b)*irr));
C5 = ((b*izz)/((a+b)*irr));
krad = mu*(radn^2)*radag;
elem1 = (4.0*krad*(io^2)*cos(angle))/(radh0^3);
elem2 = (2.0*krad*io*cos(angle))/(radh0^2);
kaxl = mu*(axln^2)*axlag;
% elem3 = (4.0*kaxl*(io^2))/(axlm*(axlh0^3));
% elem4 = (2.0*kaxl*io)/(axlm*(axlh0^2));
elem3 = kaxl/(2.0*axlm*(axlh0^3));
elem4 = kaxl/(2.0*axlm*(axlh0^2));

% Create 10x10 A matrix filled with zeroes
A = [zeros(5),eye(5);zeros(5,10)];

% Create 10x5 B matrix filled with zeroes
B = [zeros(10,5)];

% Create 10x10 C Identity matrix
C = eye(10);

% Create D matrix
D = zeros(10,5);
```

```
% Fill A matrix will proper elements
A(6,1) = C1*elem1; A(6,2) = C3*elem1; A(6,8) = -C4; A(6,9) =  C4;
A(7,1) = C3*elem1; A(7,2) = C2*elem1; A(7,8) =  C5; A(7,9) = -C5;
A(8,3) = C1*elem1; A(8,4) = C3*elem1; A(8,6) =  C4; A(8,7) = -C4;
A(9,3) = C3*elem1; A(9,4) = C2*elem1; A(9,6) = -C5; A(9,7) =  C5;
A(10,5) = elem3;

% Fill B matrix will proper elements
B(6,1) = elem2*C1; B(6,2) = elem2*C3;
B(7,1) = elem2*C3; B(7,2) = elem2*C2;
B(8,3) = elem2*C1; B(8,4) = elem2*C3;
B(9,3) = elem2*C3; B(9,4) = elem2*C2;
B(10,5) = elem4;

if(bearing == 0)  % axial bearing
 aa(1,1:2) = A(5,9:10); aa(2,1) = A(10,5); aa(2,2) = A(10,10);
 ab(1,1) = B(5,1); ab(2,1) = B(10,5);
 ac = zeros(2); ac(1,1) = 1.0;
 ad = zeros(2,1);
elseif(bearing == 1)
 ra(1,1:2) = A(1,5:6); ra(2,1) = A(6,1); ra(2,2) = A(6,6);
 rb(1,1) = B(1,1); rb(2,1) = B(6,1);
 rc = zeros(1,2); rc(1,1) = 1.0;
 rd = 0;
elseif(bearing == 2)
 ra(1,1:2) = A(3,7:8); ra(2,1) = A(8,3); ra(2,2) = A(8,8);
 rb(1,1) = B(3,3); rb(2,1) = B(8,3);
 rc = zeros(1,2); rc(1,1) = 1.0;
 rd = 0;
elseif(bearing == 3)
 ra(1,1:2) = A(2,6:7); ra(2,1) = A(7,2); ra(2,2) = A(7,7);
 rb(1,1) = B(2,2); rb(2,1) = B(7,2);
 rc = zeros(1,2); rc(1,1) = 1.0;
 rd = 0;
elseif(bearing == 4)
 ra(1,1:2) = A(4,8:9); ra(2,1) = A(9,4); ra(2,2) = A(9,9);
 rb(1,1) = B(4,4); rb(2,1) = B(9,4);
 rc = zeros(1,2); rc(1,1) = 1.0;
 rd = 0;
end

% Convert from state-space representation to transfer function
if(bearing == 0)
 [onum, oden] = ss2tf(aa, ab, ac, ad);
 onum = onum(1,3);
 % Remove rounding error
 oden(2) = 0;
else
 [onum, oden] = ss2tf(ra, rb, rc, rd);
 onum = onum(1,3);
 % Remove rounding error
 oden(2) = 0;
end

if(nargout == 0)
 [n, m] = size(onum);
 msg = ['opnum = [' sprintf('%1.3f',onum(1))];
 for i=2:m
  msg = [msg ' ' sprintf('%1.3f',onum(i))];
 end
 msg = [msg ']'];
 disp(' ');
 disp(msg);
 disp(' ');
 [n, m] = size(oden);
 msg = ['opden = [' sprintf('%1.3f',oden(1))];
 for i=2:m
  msg = [msg ' ' sprintf('%1.3f',oden(i))];
 end
```

```
 msg = [msg ']'];
 disp(msg);
 disp(' ');
else
 opnum = onum;
 opden = oden;
end
```

## H.3.4 DigControl.m

This Matlab script returns the transfer function of the digital controller given the bearing number, feedback gain, natural frequency, damping ratio, and sampling interval.

```
function [hnum, hden] = DigControl(bearing,B,freq,damp,T)
% Continuous linearized controller transfer function

if(nargin < 5)
 disp('Five input arguments required:  [hnum, hden] =
DigControl(bearing,B,freq,damp,T)');
 disp('    B       - digital controller gain (divisor)');
 disp('    freq    - digital controller model natural frequency');
 disp('    damp    - digital controller model damping ratio');
 disp('    T       - digital controller sampling interval');
 disp('    bearing - vacuum pump bearing number');
 disp('              0 = axial bearing');
 disp('              1 = radial 1X bearing');
 disp('              2 = radial 1Y bearing');
 disp('              3 = radial 2X bearing');
 disp('              4 = radial 2Y bearing');
 error;
end

% Defining linear controller constants
a22 = 2.0*damp*freq;
a21 = freq^2;
bplus = B;

% the controller model is a second order model having the following form:
%
% U(t) = U(t-T) - 1/B*[a(t-T) + a22*v(t) + a21*x(t)]
%
%    where:  U = control action
%              a = acceleration
%              v = velocity
%             x = position

% Velocity algorithm - backward difference accurate to T^2
%
% v(t) = [3x(t) - 4x(t-T) + x(t-2T)]/2*T    v(z)/x(z) = [3z^2 - 4z + 1]/2*T*z^2

velnum = [3/(2*T) -4/(2*T) 1/(2*T)];
velden = [1 0 0];

% Acceleration algorithm - central difference accurate to T^2
%
% a(t) = [v(t) - v(t-2T)]/2*T
%      = {[3x(t) - 4x(t-T) + x(t-2T)] - [3x(t-2T) - 4x(t-3T) + x(t-4T)]}/4*T^2
% a(z)/x(z) = [3z^4 - 4z^3 - 2z^2 + 4z - 1]/4*T^2*z^4

accnum = [3/(4*(T^2)) -4/(4*(T^2)) -2/(4*(T^2)) 4/(4*(T^2)) -1/(4*(T^2))];
accden = [1 0 0 0 0];

% Define controller
```

```
%
% [(z - 1)/z] U(z) = -1/B * [a(z) + a22*v(z) + a21*x(z)

%connum = -(accnum + conv((a22*velnum),deconv(accden,velden)) + (a21*accden));
%conden = bplus*accden;

% multiply by z/(z-1)
%[conden, r] = deconv(conden,[1 0]);
%conden = conv(conden, [1 -1]);

J1 = (1/(4.*(T^2)))+(a22/(2.*T))+(a21);
J2 = 0;
J3 = (-2/(4.*(T^2)))-(a22/(2.*T));
J4 = 0;
J5 = 1/(4*(T^2));

connum = -[J1 J2 J3 J4 J5];
conden = [B -B 0 0 0];

if(nargout == 0)
 [n, m] = size(connum);
 msg = ['hnum = [' sprintf('%1.3f',connum(1))];
 for i=2:m
  msg = [msg ' ' sprintf('%1.3f',connum(i))];
 end
 msg = [msg ']'];
 disp(' ');
 disp(msg);
 disp(' ');
 [n, m] = size(conden);
 msg = ['hden = [' sprintf('%1.3f',conden(1))];
 for i=2:m
  msg = [msg ' ' sprintf('%1.3f',conden(i))];
 end
 msg = [msg ']'];
 disp(msg);
 disp(' ');
else
 hnum = connum;
 hden = conden;
end
```

## H.3.5 DigPump.m

This Matlab script returns the discrete time transfer function of the driver and turbopump components. The turbopump transfer function on which this function is based uses the parameters provided by the manufacturer. The driver transfer function on which this function is based on a recursive best fit analysis.

```
function [opnum, opden] = ConPump(bearing)
% linearized ebara pump state space equations

if(nargin == 0)
 disp('One input argument required:  [opnum, opden] = ConPump(bearing)');
 disp('    bearing - vacuum pump bearing number');
 disp('              0 = axial bearing');
 disp('              1 = radial 1X bearing');
 disp('              2 = radial 1Y bearing');
 disp('              3 = radial 2X bearing');
 disp('              4 = radial 2Y bearing');
 error;
end
```

```
[onum, oden] = ConPumpNoAmp(bearing);

[ampnum, ampden] = ConAmpIdeal(bearing);

onum = conv(ampnum, onum);
oden = conv(ampden, oden);

if(nargout == 0)
 [n, m] = size(onum);
 msg = ['opnum = [' sprintf('%1.3f',onum(1))];
 for i=2:m
  msg = [msg ' ' sprintf('%1.3f',onum(i))];
 end
 msg = [msg ']'];
 disp(' ');
 disp(msg);
 disp(' ');
 [n, m] = size(oden);
 msg = ['opden = [' sprintf('%1.3f',oden(1))];
 for i=2:m
  msg = [msg ' ' sprintf('%1.3f',oden(i))];
 end
 msg = [msg ']'];
 disp(msg);
 disp(' ');
else
 opnum = onum;
 opden = oden;
end
```

## H.3.6 DigPumpIdeal.m

This Matlab script returns the discrete time transfer function of the driver and turbopump components. The turbopump and driver transfer function on which this function is based on a recursive best fit analysis.

```
function [opnum, opden] = DigPumpIdeal(bearing, T)
% linearized ebara pump state space equations

if(nargin < 2)
  disp('Two input argument required:  [opnum, opden] = DigPumpIdeal(bearing, T)');
  disp('    T       - controller sampling period');
  disp('    bearing - vacuum pump bearing number');
  disp('              0 = axial bearing');
  disp('              1 = radial 1X bearing');
  disp('              2 = radial 1Y bearing');
  disp('              3 = radial 2X bearing');
  disp('              4 = radial 2Y bearing');
  error;
end

[opnum, opden] = ConPumpIdealNoAmp(bearing);
P = opnum(size(opnum,2));
Q = abs(opden(size(opden,2)));

[ampnum, ampden] = ConAmpIdeal(bearing);

A1 = ampnum;
A2 = abs(ampden(size(ampden,2)));

K1 = -(A1*P)/(A2*Q);
K2 = -(A1*P)/(A2*((A2^2)-Q));
K3 = (A1*P)/(2*Q*(A2-sqrt(Q)));
```

```
K4 = (A1*P)/(2*Q*(A2+sqrt(Q)));

L1 = exp(-A2*T);
L2 = exp(sqrt(Q)*T);
L3 = exp(-sqrt(Q)*T);
L4 = 1;

onum = K1*conv([1 -L1],conv([1 -L2], [1 -L3]));
onum = onum + (K2*conv([1 -1], conv([1 -L2],[1 -L3])));
onum = onum + (K3*conv([1 -1], conv([1 -L1],[1 -L2])));
onum = onum + (K4*conv([1 -1], conv([1 -L1],[1 -L3])));

if( ~isreal(onum))
 onum = real(onum);
end

oden = conv([1 -L1], conv([1 -L2], [1 -L3]));
if( ~isreal(oden))
 oden = real(oden);
end

if(nargout == 0)
 [n, m] = size(onum);
 msg = ['opnum = [' sprintf('%1.3f',onum(1))];
 for i=2:m
  msg = [msg ' ' sprintf('%1.3f',onum(i))];
 end
 msg = [msg ']'];
 disp(' ');
 disp(msg);
 disp(' ');
 [n, m] = size(oden);
 msg = ['opden = [' sprintf('%1.3f',oden(1))];
 for i=2:m
  msg = [msg ' ' sprintf('%1.3f',oden(i))];
 end
 msg = [msg ']'];
 disp(msg);
 disp(' ');
else
 opnum = onum;
 opden = oden;
end
```

## H.3.7 DigPumpNoAmp.m

This Matlab script returns the discrete time transfer function of the turbopump components. The turbopump and driver transfer function on which this function is based on the parameters provided by the turbopump manufacturer.

```
function [opnum, opden] = DigPumpNoAmp(bearing, T)
% linearized ebara pump state space equations

if(nargin < 2)
  disp('Two input argument required:  [opnum, opden] = DigPumpNoAmp(bearing, T)');
  disp('    T       - controller sampling period');
  disp('    bearing - vacuum pump bearing number');
  disp('              0 = axial bearing');
  disp('              1 = radial 1X bearing');
  disp('              2 = radial 1Y bearing');
  disp('              3 = radial 2X bearing');
  disp('              4 = radial 2Y bearing');
  error;
```

```
end

[onum, oden] = ConPumpNoAmp(bearing);

P = onum(1);
Q = abs(oden(length(oden)));

K1 = -P/Q;
K2 = P/(2*Q);
K3 = P/(2*Q);

L1 = 1;
L2 = exp(-sqrt(Q)*T);
L3 = exp(sqrt(Q)*T);

onum = K1*(conv([1 -L2], [1 -L3]));
onum = onum + (K2*(conv([1 -L1], [1 -L3])));
onum = onum + (K3*(conv([1 -L1], [1 -L2])));

if( ~isreal(onum))
  onum = real(onum);
end

oden = conv([1 -L2], [1 -L3]);
if( ~isreal(oden))
  oden = real(oden);
end

if(nargout == 0)
  [n, m] = size(onum);
  msg = ['opnum = [' sprintf('%1.3f',onum(1))];
  for i=2:m
    msg = [msg ' ' sprintf('%1.3f',onum(i))];
  end
  msg = [msg ']'];
  disp(' ');
  disp(msg);
  disp(' ');
  [n, m] = size(oden);
  msg = ['opden = [' sprintf('%1.3f',oden(1))];
  for i=2:m
    msg = [msg ' ' sprintf('%1.3f',oden(i))];
  end
  msg = [msg ']'];
  disp(msg);
  disp(' ');
else
  opnum = onum;
  opden = oden;
end
```

# H.4 Miscellaneous Programs

This section lists miscellaneous programs that were used for data analysis during the course of writing this thesis.

## H.4.1 NoiseSpect.m

This Matlab script performs a power spectrum analysis upon data obtained from monitoring the position sensor signal when the bearings of the turbopump were unpowered. This function produces the sensor noise graph.

```
function [density, w] = NoiseSpect(bearing, T)

if(nargin ~= 2)
  disp('[density, w] = NoiseSpect(bearing, T)');
  disp('where:');
  disp('        bearing - vacuum pump bearing number');
  disp('                  0 = axial bearing');
  disp('                  1 = radial 1X bearing');
  disp('                  2 = radial 1Y bearing');
  disp('                  3 = radial 2X bearing');
  disp('                  4 = radial 2Y bearing');
  disp('        T       - sampling interval');
  error;
end

if(bearing == 0)
  filename = '/tmp/axial10k.dat';
elseif(bearing == 1)
  filename = '/tmp/rad1x10k.dat';
elseif(bearing == 2)
  filename = '/tmp/rad1y10k.dat';
elseif(bearing == 3)
  filename = '/tmp/rad2x10k.dat';
elseif(bearing == 4)
  filename = '/tmp/rad2y10k.dat';
end

fid = fopen(deblank(filename), 'r');
if(fid < 3)
  disp(['Unable to open input file ''' deblank(filename) '''']);
  error;
end

[dat] = fscanf(fid, '%d', [1 inf]);
fclose(fid);

avg = mean(dat);
dat = dat - avg;
nfft = 1024;
off = 6;

[p,w] = spectrum(dat,nfft,0,hanning(nfft),1/T);

if(nargout == 0)
  plot(w(off:size(w,1),1),p(off:size(p,1),1));
else
  density = p(off:size(p,1));
  w = w(off:size(w,1));
end
```

## H.4.2 PhaseFix.m

This Matlab script attempts to smooth the phase data obtained by the system analyzer. The system analyzer normally saves the phase data in a form in which the data ranges from ±180°. This form causes the phase data to jump around when the value approaches -180°. This script attempts to eliminate such jumps by not limiting the minimum value to -180°.

```
function [phase] = PhaseFix(bearing, phdata)
% This attempts to fix phase data from system analyzer

if(nargin ~= 2)
  disp('Syntax error');
  disp('[phase] = PhaseFix(bearing, phdata)');
```

```
  disp('where:');
  disp('        bearing - vacuum pump bearing number');
  disp('                  0 = axial bearing');
  disp('                  1 = radial 1X bearing');
  disp('                  2 = radial 1Y bearing');
  disp('                  3 = radial 2X bearing');
  disp('                  4 = radial 2Y bearing');
  disp('        phdata  - analyzer phase data');
  error;
end

phase = phdata;

% insure that initial phase is positive
while(phase(1) > 170.0)
  phase(1) = phase(1) - 360.0;
end

% smooth phase anomalies
for ii=2:length(phase);
  while(abs(phase(ii-1)-phase(ii)) > 170.0)
    if(phase(ii-1) >= 0.0)
      phase(ii) = phase(ii)+360.0;
    else
      phase(ii) = phase(ii)-360.0;
    end
  end
end
```

## H.4.3 veltest.m

This Matlab script reads the sensor position data obtained when the turbopump bearings are unpowered and applies central difference and forward difference representations of the velocity function to determine which is better suited for this noisy environment.

```
function [] = veltest(bearing)

if(bearing == 0)
  infile = '/tmp/axial10k.dat';
elseif(bearing == 1)
  infile = '/tmp/rad1x10k.dat';
elseif(bearing == 2)
  infile = '/tmp/rad1y10k.dat';
elseif(bearing == 3)
  infile = '/tmp/rad2x10k.dat';
elseif(bearing == 4)
  infile = '/tmp/rad2y10k.dat';
end

fid = fopen(infile, 'r');
if(fid > 2)
  [dat,count] = fscanf(fid, '%d', [1 inf]);
  fclose(fid);
  count = mean(dat);
  dat = dat - count;
  cvel = zeros(1,length(dat));
  fvel = zeros(1,length(dat));
  for ii = 3:length(dat)-1
    cvel(ii) = (dat(ii+1)-dat(ii-1)) * 10000 / 2;
    fvel(ii) = ((3*dat(ii))-(4*dat(ii-1))+dat(ii-2)) * 10000 / 2;
  end
  for ii = 1:50:length(dat)
    plot(fvel(ii:ii+50),'y-');
```

```
    hold on;
    plot(cvel(ii:ii+50),'g-');
    hold off;
    pause;
  end
end
```

# Append I

# Digital Controller Listing

---

This appendix provides a brief explanation of important aspects of the digital controller program and the actual listings of both the architecture file and the controller assembly file.

## I.1 Digital Controller Details

The digital controller is composed of an Architecture File (AF) and the Controller Assembly File (CAF). The AF describes the memory layout that the processor is constrained by. The AF is used by the assembler to resolve the absolute memory locations necessary to successfully assemble the CAF. The AF describes the interrupt vector table, the Program Memory (PM) address range, the Data Memory (DM) address range, and the registers of auxiliary components that are mapped in the PM and DM address space. The AF also provides named aliases for these addresses so that these names can be used instead of the actual addresses as references in the CAF.

The CAF contains the native DSP assembly language instructions that implement the control algorithm. The CAF defines variable names to all of the registers mapped in either the PM or DM address range. In this way, the program can treat reads and writes to these registers as simply reads and writes to and from any other memory location. The file also modifies the interrupt vector table located at the front of the PM address space to alter the behavior of the processor when it receives a reset and timer interrupt. This file also defines the instructions that will be loaded in the PM address space and the provides names for the variables loaded into the DM address space. Any program variable that could possibility require alteration during the controller design process is aliased to a name (i.e. the #define statements) that is used throughout the controller program. In this way changes can be made quickly that effect the entire controller. The variables used within the actual control loop are organized in their order of use in DM so that the control loop can be optimized to use indirect addressing. Indirect addressing allows for two instructions to the be implemented in one cycle thus increasing the efficiency of the control loop.

The PM contains the initialization code and the control loop. The initialization code uses the named aliases of the program variables to compute the DM variables that will be used in the control loop. This increases the delay from program initiation to controller initiation but allows for changes to be made quickly and easily to the controller. Wherever possible, the initialization code removes division operations from the control loop by calculating reciprocals. After

calculating all of the control loop variables, the initialization code initializes the I/O Controller board and sets the number of A/Ds and D/As. Next the initial control signals are sent to the D/As and all DM variables calculated during the control loop are initialized to zero. Finally the timer counter register is set to the appropriate value calculated from the desired sampling rate and processor interrupts are allowed. The processor then enters an idle state waiting for an interrupt to occur.

The timer interrupt directs program execution to the control loop. The control loop queries the bearing number variable to determine which is the axis of interest and signals the A/D to start the conversion process. Next, certain memory pointer registers are initialized to point to the areas in the DM that contain the variables applicable to the axis of interest. The program then waits till the A/D conversion process has finished. The A/D offset is subtracted from the value and the signal is processed by the low-pass filter and the sensor notch filter. The value is then converted to the proper units and the control action calculates the proper control action. The control action is tested if it exceeds the user defined control signal limit and is then passed through a notch filter to remove frequencies that might excite the first bending mode. The control loop variables are updated for the next interrupt and the control signal is sent to the D/A. The controller waits for the D/A conversion process to complete before incrementing the bearing number variable and returning from the interrupt.

This program is downloaded to the DSP controller board using a utility run from the Personal Computer (PC) command line. Another PC command line utility is then used to trigger a DSP reset. This in turn triggers the reset interrupt which initializes certain memory access chip registers and jumps to the program initialization code which starts the controller.

## I.1.1 Architecture File

```
.SYSTEM EBARA;

.PROCESSOR = ADSP21020;

.SEGMENT /ROM /BEGIN=0x000000 /END=0x000007 /PM emu_svc;
.SEGMENT /RAM /BEGIN=0x000008 /END=0x00000F /PM rst_svc;
.SEGMENT /ROM /BEGIN=0x000010 /END=0x000017 /PM resrvd1;
.SEGMENT /ROM /BEGIN=0x000018 /END=0x00001F /PM sovf_svc;
.SEGMENT /RAM /BEGIN=0x000020 /END=0x000027 /PM tmzh_svc;
.SEGMENT /ROM /BEGIN=0x000028 /END=0x00002F /PM irq3_svc;
.SEGMENT /ROM /BEGIN=0x000030 /END=0x000037 /PM irq2_svc;
.SEGMENT /ROM /BEGIN=0x000038 /END=0x00003F /PM irq1_svc;
.SEGMENT /ROM /BEGIN=0x000040 /END=0x000047 /PM irq0_svc;
.SEGMENT /ROM /BEGIN=0x000048 /END=0x00004F /PM resrvd2;
.SEGMENT /ROM /BEGIN=0x000050 /END=0x000057 /PM resrvd3;
.SEGMENT /ROM /BEGIN=0x000058 /END=0x00005F /PM cb7_svc;
.SEGMENT /ROM /BEGIN=0x000060 /END=0x000067 /PM cb15_svc;
.SEGMENT /ROM /BEGIN=0x000068 /END=0x00006F /PM resrvd4;
.SEGMENT /ROM /BEGIN=0x000070 /END=0x000077 /PM tmzl_svc;
.SEGMENT /ROM /BEGIN=0x000078 /END=0x00007F /PM fix_svc;
.SEGMENT /ROM /BEGIN=0x000080 /END=0x000087 /PM flto_svc;
.SEGMENT /ROM /BEGIN=0x000088 /END=0x00008F /PM fltu_svc;
.SEGMENT /ROM /BEGIN=0x000090 /END=0x000097 /PM flti_svc;
.SEGMENT /ROM /BEGIN=0x000098 /END=0x00009F /PM resrvd5;
.SEGMENT /ROM /BEGIN=0x0000A0 /END=0x0000A7 /PM resrvd6;
.SEGMENT /ROM /BEGIN=0x0000A8 /END=0x0000AF /PM resrvd7;
.SEGMENT /ROM /BEGIN=0x0000B0 /END=0x0000B7 /PM resrvd8;
.SEGMENT /ROM /BEGIN=0x0000B8 /END=0x0000BF /PM resrvd9;
```

```
.SEGMENT /ROM /BEGIN=0x0000C0 /END=0x0000C7 /PM sft0_svc;
.SEGMENT /ROM /BEGIN=0x0000C8 /END=0x0000CF /PM sft1_svc;
.SEGMENT /ROM /BEGIN=0x0000D0 /END=0x0000D7 /PM sft2_svc;
.SEGMENT /ROM /BEGIN=0x0000D8 /END=0x0000DF /PM sft3_svc;
.SEGMENT /ROM /BEGIN=0x0000E0 /END=0x0000E7 /PM sft4_svc;
.SEGMENT /ROM /BEGIN=0x0000E8 /END=0x0000EF /PM sft5_svc;
.SEGMENT /ROM /BEGIN=0x0000F0 /END=0x0000F7 /PM sft6_svc;
.SEGMENT /ROM /BEGIN=0x0000F8 /END=0x0000FF /PM sft7_svc;

.SEGMENT /RAM /BEGIN=0x000100 /END=0x007FFF /PM pm_code;

.SEGMENT /RAM /BEGIN=0x00000000 /END=0x00007FFF /DM dm_data;

.BANK /PM0 /WTSTATES=0 /WTMODE=INTERNAL /BEGIN=0X000000;
.BANK /DM0 /WTSTATES=0 /WTMODE=INTERNAL /BEGIN=0X00000000;
.BANK /DM1 /WTSTATES=1 /WTMODE=INTERNAL /BEGIN=0X40000000;

.SEGMENT /PORT /BEGIN=0X800000   /END=0X800000   /PM ioadin;
.SEGMENT /PORT /BEGIN=0X800001   /END=0X800001   /PM iostat;

.SEGMENT /PORT /BEGIN=0X20000000 /END=0X20000000 /DM status;
.SEGMENT /PORT /BEGIN=0X20000001 /END=0X20000001 /DM timer;
.SEGMENT /PORT /BEGIN=0X20000002 /END=0X20000002 /DM digio;

.SEGMENT /PORT /BEGIN=0X40000000 /END=0X40000000 /DM iodaout;
.SEGMENT /PORT /BEGIN=0X40000001 /END=0X40000001 /DM iocntrl;
.SEGMENT /PORT /BEGIN=0X40000002 /END=0X40000002 /DM iochans;


.ENDSYS;
```

## I.1.2 Controller Assembly File

```
{ Contrl11.asm - uses only the ADSP timer.                    }

{ The following ports are found on the ADSP board             }

.SEGMENT /DM status;
.VAR DSPSTAT;
.ENDSEG;

.SEGMENT /DM timer;
.VAR DSPTIMER;
.ENDSEG;

{ The following ports are found on the 32-channel ADC board   }

.SEGMENT /PM ioadin;
.VAR ADFIFO;
.ENDSEG;

.SEGMENT /PM iostat;
.VAR IOSTAT;
.ENDSEG;

.SEGMENT /DM iodaout;
.VAR DAFIFO;
.ENDSEG;

.SEGMENT /DM iochans;
.VAR CHANNELS;
.ENDSEG;

.SEGMENT /DM iocntrl;
.VAR CONTROL;
.ENDSEG;
```

```
{ division macro - if there is a quicker way, just change macro }
{                  macro requires the following:                 }
{                  F0  = numerator                               }
{                  F12 = denominator                             }

{-------------------------------------------------------------------------}
{ Division Algorithm - Given Q = D/N, multiply N and D by the same set  }
{                      of factors, Rn.                                  }
{      N x R0 x R1 x ... x Rn                                           }
{  Q = ----------------------                                           }
{      D x R0 x R1 x ... x Rn                                           }
{                                                                       }
{ Choose Rn such that as the number of factors increases, the           }
{ denominator approaches 1.  The quotient is then approximately equal   }
{ to the numerator.                                                     }
{                                                                       }
{ R0 is the seed provided by the RECIPS instruction.  Succssive Rn are  }
{ calculated by the following formula:                                  }
{  Ri = 2-D(i-1)                                                        }
{                                                                       }
{ NOTE:  The macro uses the following registers:                        }
{        F0, F7, F11, F12                                               }
{        These registers will be over-written upon exit from macro.     }
{-------------------------------------------------------------------------}

#define div     F11 = 2.0; \
                F0 = RECIPS F12, F7 = F0; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F0 = F0*F7

#define DIV     F11 = 2.0; \
                F0 = RECIPS F12, F7 = F0; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F0 = F0*F7

{-------------------------------------------------------------------------}
{ Axial bearing constants                                               }
{                                                                       }
{       AXIAL_NATFREQ   natural frequency                               }
{       AXIAL_DAMP      damping ratio                                   }
{       AXIAL_PERIOD    period constant (divisor = axial_period*T)      }
{       AXIAL_BPLS      B+ matrix/scalar                                }
{       AXIAL_SENGAIN   sensor gain (volts/meter)                       }
{       AXIAL_AMPGAIN   amp gain (amps/volt)                            }
{       AXIAL_INITSIG   initial control signal (Amps)                   }
{       AXIAL_MAXU      maximum control signal (Amps)                   }
{-------------------------------------------------------------------------}

#define AXIAL_NATFREQ   100.0
#define AXIAL_DAMP      0.85
#define AXIAL_PERIOD    2.0
#define AXIAL_BPLS      170.0
#define AXIAL_SENGAIN   9450.0
#define AXIAL_AMPGAIN   2.515
#define AXIAL_INITSIG   -1.0
#define AXIAL_MAXU       2.0


{ Radial bearing constants - common     }
```

```
#define RAD_PERIOD      2.0

{------------------------------------------------------------------------}
{ Radial bearing constants - bearing 1X                                  }
{                                                                        }
{       RAD1X_NATFREQ   natural frequency                                }
{       RAD1X_DAMP      damping ratio                                    }
{       RAD1X_PERIOD    period constant (divisor = rad1x_period*T)       }
{       RAD1X_BPLS      B+ matrix/scalar                                 }
{       RAD1X_SENGAIN   sensor gain (volts/meter)                        }
{       RAD1X_AMPGAIN   amp gain (amps/volt)                             }
{       RAD1X_INITSIG   initial control signal (Amps)                    }
{       RAD1X_MAXU      maximum control signal (Amps)                    }
{------------------------------------------------------------------------}

#define RAD1X_NATFREQ  130.0
#define RAD1X_DAMP      1.00
#define RAD1X_BPLS     160.0
#define RAD1X_SENGAIN  25000.0
#define RAD1X_AMPGAIN  1.175
#define RAD1X_INITSIG   0.0
#define RAD1X_MAXU      2.0

{------------------------------------------------------------------------}
{ Radial bearing constants - bearing 1Y                                  }
{                                                                        }
{       RAD1Y_NATFREQ   natural frequency                                }
{       RAD1Y_DAMP      damping ratio                                    }
{       RAD1Y_PERIOD    period constant (divisor = rad1y_period*T)       }
{       RAD1Y_BPLS      B+ matrix/scalar                                 }
{       RAD1Y_SENGAIN   sensor gain (volts/meter)                        }
{       RAD1Y_AMPGAIN   amp gain (amps/volt)                             }
{       RAD1Y_INITSIG   initial control signal (Amps)                    }
{       RAD1Y_MAXU      maximum control signal (Amps)                    }
{------------------------------------------------------------------------}

#define RAD1Y_NATFREQ  110.0
#define RAD1Y_DAMP      1.05
#define RAD1Y_BPLS     180.0
#define RAD1Y_SENGAIN  25000.0
#define RAD1Y_AMPGAIN  1.165
#define RAD1Y_INITSIG   0.0
#define RAD1Y_MAXU      2.0

{------------------------------------------------------------------------}
{ Radial bearing constants - bearing 2X                                  }
{                                                                        }
{       RAD2X_NATFREQ   natural frequency                                }
{       RAD2X_DAMP      damping ratio                                    }
{       RAD2X_PERIOD    period constant (divisor = rad2x_period*T)       }
{       RAD2X_BPLS      B+ matrix/scalar                                 }
{       RAD2X_SENGAIN   sensor gain (volts/meter)                        }
{       RAD2X_AMPGAIN   amp gain (amps/volt)                             }
{       RAD2X_INITSIG   initial control signal (Amps)                    }
{       RAD2X_MAXU      maximum control signal (Amps)                    }
{------------------------------------------------------------------------}

#define RAD2X_NATFREQ  100.0
#define RAD2X_DAMP      1.90
#define RAD2X_BPLS     260.0
#define RAD2X_SENGAIN  25000.0
#define RAD2X_AMPGAIN  1.159
#define RAD2X_INITSIG   0.0
#define RAD2X_MAXU      2.0

{------------------------------------------------------------------------}
{ Radial bearing constants - bearing 2Y                                  }
{                                                                        }
```

```
{       RAD2Y_NATFREQ   natural frequency                                    }
{       RAD2Y_DAMP      damping ratio                                        }
{       RAD2Y_PERIOD    period constant (divisor = rad2y_period*T)           }
{       RAD2Y_BPLS      B+ matrix/scalar                                     }
{       RAD2Y_SENGAIN   sensor gain (volts/meter)                            }
{       RAD2Y_AMPGAIN   amp gain (amps/volt)                                 }
{       RAD2Y_INITSIG   initial control signal (Amps)                        }
{       RAD2Y_MAXU      maximum control signal (Amps)                        }
{--------------------------------------------------------------------------}

#define RAD2Y_NATFREQ  100.0
#define RAD2Y_DAMP      1.80
#define RAD2Y_BPLS     260.0
#define RAD2Y_SENGAIN  25000.0
#define RAD2Y_AMPGAIN  1.144
#define RAD2Y_INITSIG   0.0
#define RAD2Y_MAXU      2.0

{ D/A board does not output 0 when zero sent.  These offsets    }
{ were determined through experimentation                        }

#define DA0OFFSET  0x0000000C
#define DA1OFFSET  0xFFFFFFE6
#define DA2OFFSET  0xFFFFFFE6
#define DA3OFFSET  0xFFFFFFE6
#define DA4OFFSET  0x00000027

#define DA0CUROFF   0.0000
#define DA1CUROFF   0.0000
#define DA2CUROFF   0.0000
#define DA3CUROFF   0.0000
#define DA4CUROFF   0.0000

{ A/D board does not return 0 with no input.  These offsets      }
{ were determined through experimentation                        }

#define AD0OFFSET  0x00000918
#define AD1OFFSET  0x000008B1
#define AD2OFFSET  0x000008F6
#define AD3OFFSET  0x000008E6
#define AD4OFFSET  0x000000AF
#define AD5OFFSET  0x0000088D
#define AD6OFFSET  0x00000926

{ D/A constants                                                  }
#define DAVOLTSMAX   5.0
#define DAVOLTSMIN  -5.0
#define DABITSMAX   32768.0

{ A/D constants                                                  }
#define ADVOLTSMAX   5.0
#define ADVOLTSMIN  -5.0
#define ADBITSMAX   8192.0

{--------------------------------------------------------------------------}
{ Timer constants                                                          }
{                                                                          }
{       CPUSPEED   CPU speed (Hz)                                          }
{       SMPLSPEED  Sampling Rate (Hz)                                      }
{--------------------------------------------------------------------------}
#define CPUSPEED   33333333.333
#define SMPLSPEED  10000.0

#define BEARTOT       5
#define SAVEOFFSET  256
#define SAVEBEAR      3
#define DELAY      30000

#ifndef NO_FILTER
```

```
{ Filter variables      }
#define FIL0N2     4.0
#define FIL0N3     6.0
#define FIL0N4     4.0
#define FIL0N5     1.0
#define FIL0K1   206.8088
#define FIL0K2  -489.8232
#define FIL0K3   478.2204
#define FIL0K4  -217.9162
#define FIL0K5    38.7102
#define FIL1N2     4.0
#define FIL1N3     6.0
#define FIL1N4     4.0
#define FIL1N5     1.0
#define FIL1K1   206.8088
#define FIL1K2  -489.8232
#define FIL1K3   478.2204
#define FIL1K4  -217.9162
#define FIL1K5    38.7102
#define FIL2N2     4.0
#define FIL2N3     6.0
#define FIL2N4     4.0
#define FIL2N5     1.0
#define FIL2K1   206.8088
#define FIL2K2  -489.8232
#define FIL2K3   478.2204
#define FIL2K4  -217.9162
#define FIL2K5    38.7102
#define FIL3N2     4.0
#define FIL3N3     6.0
#define FIL3N4     4.0
#define FIL3N5     1.0
#define FIL3K1   206.8088
#define FIL3K2  -489.8232
#define FIL3K3   478.2204
#define FIL3K4  -217.9162
#define FIL3K5    38.7102
#define FIL4N2     4.0
#define FIL4N3     6.0
#define FIL4N4     4.0
#define FIL4N5     1.0
#define FIL4K1   206.8088
#define FIL4K2  -489.8232
#define FIL4K3   478.2204
#define FIL4K4  -217.9162
#define FIL4K5    38.7102
#endif

#ifndef NO_INOTCH_FILTER
#define  INFIL0COSWT    0.990461426
#define  INFIL0ALPHA    0.25
#define  INFIL0L        0.25
#define  INFIL0A0       1.0
#define  INFIL1COSWT    0.990461426
#define  INFIL1ALPHA    0.25
#define  INFIL1L        0.25
#define  INFIL1A0       1.0
#define  INFIL2COSWT    0.990461426
#define  INFIL2ALPHA    0.25
#define  INFIL2L        0.25
#define  INFIL2A0       0.8
#define  INFIL3COSWT    0.951056516
#define  INFIL3ALPHA    0.25
#define  INFIL3L        0.25
#define  INFIL3A0       0.9
#define  INFIL4COSWT    0.951056516
#define  INFIL4ALPHA    0.25
#define  INFIL4L        0.25
#define  INFIL4A0       1.0
```

```
#endif

#ifndef NO_NOTCH_FILTER
#define  NFIL0COSWT    0.827080574
#define  NFIL0ALPHA    0.20
#define  NFIL0L        0.20
#define  NFIL0A0       1.0
#define  NFIL1COSWT    0.809016994
#define  NFIL1ALPHA    0.20
#define  NFIL1L        0.20
#define  NFIL1A0       1.0
#define  NFIL2COSWT    0.823532598
#define  NFIL2ALPHA    0.20
#define  NFIL2L        0.20
#define  NFIL2A0       1.0
#define  NFIL3COSWT    0.770513243
#define  NFIL3ALPHA    0.20
#define  NFIL3L        0.20
#define  NFIL3A0       1.0
#define  NFIL4COSWT    0.770513243
#define  NFIL4ALPHA    0.20
#define  NFIL4L        0.20
#define  NFIL4A0       1.0
#endif

{ Axial bearing variable storage locations      }

.SEGMENT /DM dm_data;

{ Common radial bearing variable storage locations      }

.VAR AXIAL_X0;
.VAR AXIAL_X1;
.VAR AXIAL_X2;
.VAR AXIAL_V0;
.VAR AXIAL_V1;
.VAR AXIAL_V2;
.VAR AXIAL_A0;
.VAR AXIAL_U1;
.VAR AXIAL_U0;
.VAR AXIAL_CONV1;
.VAR AXIAL_PERIOD2;
.VAR AXIAL_A22;
.VAR AXIAL_A21;
.VAR AXIAL_BPLUS;
.VAR AXIAL_CONV2;

{ First radial bearing variable storage locations - X direction }

.VAR RAD1X_X0;
.VAR RAD1X_X1;
.VAR RAD1X_X2;
.VAR RAD1X_V0;
.VAR RAD1X_V1;
.VAR RAD1X_V2;
.VAR RAD1X_A0;
.VAR RAD1X_U1;
.VAR RAD1X_U0;
.VAR RAD1X_CONV1;
.VAR RAD1X_PERIOD2;
.VAR RAD1X_A22;
.VAR RAD1X_A21;
.VAR RAD1X_BPLUS;
.VAR RAD1X_CONV2;

{ First radial bearing variable storage locations - Y direction }

.VAR RAD1Y_X0;
.VAR RAD1Y_X1;
```

```
.VAR RAD1Y_X2;
.VAR RAD1Y_V0;
.VAR RAD1Y_V1;
.VAR RAD1Y_V2;
.VAR RAD1Y_A0;
.VAR RAD1Y_U1;
.VAR RAD1Y_U0;
.VAR RAD1Y_CONV1;
.VAR RAD1Y_PERIOD2;
.VAR RAD1Y_A22;
.VAR RAD1Y_A21;
.VAR RAD1Y_BPLUS;
.VAR RAD1Y_CONV2;

{ Second radial bearing variable storage locations - X direction }

.VAR RAD2X_X0;
.VAR RAD2X_X1;
.VAR RAD2X_X2;
.VAR RAD2X_V0;
.VAR RAD2X_V1;
.VAR RAD2X_V2;
.VAR RAD2X_A0;
.VAR RAD2X_U1;
.VAR RAD2X_U0;
.VAR RAD2X_CONV1;
.VAR RAD2X_PERIOD2;
.VAR RAD2X_A22;
.VAR RAD2X_A21;
.VAR RAD2X_BPLUS;
.VAR RAD2X_CONV2;

{ Second radial bearing variable storage locations - Y direction }

.VAR RAD2Y_X0;
.VAR RAD2Y_X1;
.VAR RAD2Y_X2;
.VAR RAD2Y_V0;
.VAR RAD2Y_V1;
.VAR RAD2Y_V2;
.VAR RAD2Y_A0;
.VAR RAD2Y_U1;
.VAR RAD2Y_U0;
.VAR RAD2Y_CONV1;
.VAR RAD2Y_PERIOD2;
.VAR RAD2Y_A22;
.VAR RAD2Y_A21;
.VAR RAD2Y_BPLUS;
.VAR RAD2Y_CONV2;

{ Conversion factors                                          }
.VAR VOLT2BITS;
.VAR BITS2VOLT;

{ A/D input registers - required because of less than ideal     }
{ I/O board configuration  (somebody hit Jim)                   }

.VAR AD0INPUT;
.VAR AD1INPUT;
.VAR AD2INPUT;
.VAR AD3INPUT;
.VAR AD4INPUT;

{ D/A offset registers - required because of less than ideal    }
{ I/O board configuration  (somebody hit Jim)                   }

.VAR AD0OFF;
.VAR AD1OFF;
.VAR AD2OFF;
```

```
.VAR AD3OFF;
.VAR AD4OFF;

#ifndef NO_FILTER
{ Filter constants      }
.VAR FIL0XCN1;
.VAR FIL1XCN1;
.VAR FIL2XCN1;
.VAR FIL3XCN1;
.VAR FIL4XCN1;
.VAR FIL0X4;
.VAR FIL1X4;
.VAR FIL2X4;
.VAR FIL3X4;
.VAR FIL4X4;
.VAR FIL0XCN5;
.VAR FIL1XCN5;
.VAR FIL2XCN5;
.VAR FIL3XCN5;
.VAR FIL4XCN5;
.VAR FIL0YCN5;
.VAR FIL1YCN5;
.VAR FIL2YCN5;
.VAR FIL3YCN5;
.VAR FIL4YCN5;
.VAR FIL0X3;
.VAR FIL1X3;
.VAR FIL2X3;
.VAR FIL3X3;
.VAR FIL4X3;
.VAR FIL0XCN4;
.VAR FIL1XCN4;
.VAR FIL2XCN4;
.VAR FIL3XCN4;
.VAR FIL4XCN4;
.VAR FIL0YCN4;
.VAR FIL1YCN4;
.VAR FIL2YCN4;
.VAR FIL3YCN4;
.VAR FIL4YCN4;
.VAR FIL0X2;
.VAR FIL1X2;
.VAR FIL2X2;
.VAR FIL3X2;
.VAR FIL4X2;
.VAR FIL0XCN3;
.VAR FIL1XCN3;
.VAR FIL2XCN3;
.VAR FIL3XCN3;
.VAR FIL4XCN3;
.VAR FIL0YCN3;
.VAR FIL1YCN3;
.VAR FIL2YCN3;
.VAR FIL3YCN3;
.VAR FIL4YCN3;
.VAR FIL0X1;
.VAR FIL1X1;
.VAR FIL2X1;
.VAR FIL3X1;
.VAR FIL4X1;
.VAR FIL0XCN2;
.VAR FIL1XCN2;
.VAR FIL2XCN2;
.VAR FIL3XCN2;
.VAR FIL4XCN2;
.VAR FIL0YCN2;
.VAR FIL1YCN2;
.VAR FIL2YCN2;
.VAR FIL3YCN2;
```

```
.VAR FIL4YCN2;
#endif

#ifndef NO_INOTCH_FILTER
.VAR INFIL0X2;
.VAR INFIL1X2;
.VAR INFIL2X2;
.VAR INFIL3X2;
.VAR INFIL4X2;
.VAR INFIL0X1;
.VAR INFIL1X1;
.VAR INFIL2X1;
.VAR INFIL3X1;
.VAR INFIL4X1;
.VAR INFIL0DK2;
.VAR INFIL1DK2;
.VAR INFIL2DK2;
.VAR INFIL3DK2;
.VAR INFIL4DK2;
.VAR INFIL0DK1;
.VAR INFIL1DK1;
.VAR INFIL2DK1;
.VAR INFIL3DK1;
.VAR INFIL4DK1;
.VAR INFIL0NK1;
.VAR INFIL1NK1;
.VAR INFIL2NK1;
.VAR INFIL3NK1;
.VAR INFIL4NK1;
#endif

{ Positive maximum control current values                    }
.VAR AXIAL_MAX_CUR;
.VAR RAD1X_MAX_CUR;
.VAR RAD1Y_MAX_CUR;
.VAR RAD2X_MAX_CUR;
.VAR RAD2Y_MAX_CUR;

#ifndef NO_NOTCH_FILTER
.VAR NFIL0X2;
.VAR NFIL1X2;
.VAR NFIL2X2;
.VAR NFIL3X2;
.VAR NFIL4X2;
.VAR NFIL0X1;
.VAR NFIL1X1;
.VAR NFIL2X1;
.VAR NFIL3X1;
.VAR NFIL4X1;
.VAR NFIL0DK2;
.VAR NFIL1DK2;
.VAR NFIL2DK2;
.VAR NFIL3DK2;
.VAR NFIL4DK2;
.VAR NFIL0DK1;
.VAR NFIL1DK1;
.VAR NFIL2DK1;
.VAR NFIL3DK1;
.VAR NFIL4DK1;
.VAR NFIL0NK1;
.VAR NFIL1NK1;
.VAR NFIL2NK1;
.VAR NFIL3NK1;
.VAR NFIL4NK1;
#endif

.VAR DA0CRTOFF;
.VAR DA1CRTOFF;
.VAR DA2CRTOFF;
```

```
.VAR DA3CRTOFF;
.VAR DA4CRTOFF;

{ D/A offset registers - required because of less than ideal    }
{ I/O board configuration  (somebody hit Jim)                   }

.VAR DA0OFF;
.VAR DA1OFF;
.VAR DA2OFF;
.VAR DA3OFF;
.VAR DA4OFF;

{ D/A output registers - required because of less than ideal    }
{ I/O board configuration  (somebody hit Jim)                   }

.VAR DA0CONTROL;
.VAR DA1CONTROL;
.VAR DA2CONTROL;
.VAR DA3CONTROL;
.VAR DA4CONTROL;

{ Program flow control variables                                }

.VAR BEAR_NUM;

{ Variables needed to save data in memory for later dumping     }

.VAR SAVE_BEARING;
.VAR INPUT_OFFSET;
.VAR OUTPUT_OFFSET;
.VAR SAVE_END;
.VAR DELAY_COUNT;
.VAR UNDER_COUNT;

.ENDSEG;


.SEGMENT /PM rst_svc;

{-------------------------------------------------------------------------}
{ At reset, the BANK registers are as follows:                            }
{       PMBANK1 = 0x800000                                                }
{       DMBANK1 = 0x20000000                                              }
{       DMBANK2 = 0x40000000                                              }
{       DMBANK3 = 0x80000000                                              }
{ These values, by coincidence, are perfect for our I/O board             }
{-------------------------------------------------------------------------}

{-------------------------------------------------------------------------}
{ The default value of PMWAIT at reset is 0x0003DE.  This corresponds     }
{ to the following:                                                       }
{       bit  13    = 0   (No automatic wait state)                        }
{       bits 12-10 = 000 (memory page size = 256 words)                   }
{       bits  9-7  = 111 (7 PMBANK1 wait states)                          }
{       bits  6-5  = 10  (Int. and Ext. wait state ack mode)              }
{       bits  4-2  = 111 (7 PMBANK0 wait states)                          }
{       bits  1-0  = 10  (Int. and Ext. wait state ack mode)              }
{                                                                         }
{ For our setup, the DSP board accesses memory at 0 wait states and       }
{ accesses all ports and 1 wait state.  Therefore:                        }
{       bit  13    = 0   (No automatic wait state)                        }
{       bits 12-10 = 100 (memory page size = 4096 words)                  }
{       bits  9-7  = 001 (1 PMBANK1 wait state)                           }
{       bits  6-5  = 10  (Int. and Ext. wait state ack mode)              }
{       bits  4-2  = 000 (0 PMBANK0 wait states)                          }
{       bits  1-0  = 10  (Int. and Ext. wait state ack mode)              }
{-------------------------------------------------------------------------}

                PMWAIT = 0x0010C2;
```

```
{-------------------------------------------------------------------}
{ The default value of DMWAIT at reset is 0x000F7BDE.  This corresponds }
{ to the following:                                                   }
{       bit  23    = 0   (No automatic wait state)                    }
{       bits 22-20 = 000 (memory page size = 256 words)               }
{       bits 19-17 = 111 (7 DMBANK3 wait states)                      }
{       bits 16-15 = 10  (Int. and Ext. wait state ack mode)          }
{       bits 14-12 = 111 (7 DMBANK2 wait states)                      }
{       bits 11-10 = 10  (Int. and Ext. wait state ack mode)          }
{       bits  9-7  = 111 (7 DMBANK1 wait states)                      }
{       bits  6-5  = 10  (Int. and Ext. wait state ack mode)          }
{       bits  4-2  = 111 (7 DMBANK0 wait states)                      }
{       bits  1-0  = 10  (Int. and Ext. wait state ack mode)          }
{                                                                     }
{ For our setup, the DSP board accesses memory at 0 wait states and   }
{ accesses all ports and 1 wait state.  Therefore:                    }
{       bit  23    = 0   (No automatic wait state)                    }
{       bits 22-20 = 100 (memory page size = 4096 words)              }
{       bits 19-17 = 001 (1 DMBANK3 wait states)                      }
{       bits 16-15 = 10  (Int. and Ext. wait state ack mode)          }
{       bits 14-12 = 001 (1 DMBANK2 wait states)                      }
{       bits 11-10 = 10  (Int. and Ext. wait state ack mode)          }
{       bits  9-7  = 000 (0 DMBANK1 wait state)                       }
{       bits  6-5  = 10  (Int. and Ext. wait state ack mode)          }
{       bits  4-2  = 000 (0 DMBANK0 wait states)                      }
{       bits  1-0  = 10  (Int. and Ext. wait state ack mode)          }
{-------------------------------------------------------------------}

                DMWAIT = 0x00431842;

{-------------------------------------------------------------------}
{ Set FLAG2 to output mode so we can trigger I/O board conversion when  }
{ we need to.                                                         }
{-------------------------------------------------------------------}

                MODE2 = 0x00020000;


                JUMP initialize;
.ENDSEG;


.SEGMENT /PM pm_code;

initialize:

{ initialize registers }

                IMASK = 0;
                MODE1 = 0x00012000;

                I0 = 0;
                I1 = 0;
                I2 = 0;
                I3 = 0;
                I4 = 0;
                I5 = 0;
                I6 = 0;
                I7 = 0;
                M0 = 0;
                M1 = 0;
                M2 = 0;
                M3 = 0;
                M4 = 0;
                M5 = 0;
                M6 = 0;
                M7 = 0;
                L0 = 0;
                L1 = 0;
```

```
                L2 = 0;
                L3 = 0;
                L4 = 0;
                L5 = 0;
                L6 = 0;
                L7 = 0;

{ Make sure FLAG2 toggle is initially zero.                                    }
                BIT CLR ASTAT 0x00200000;

{ Calculate D/A conversion factor                                              }
                F0 = DABITSMAX;
                F12 = DAVOLTSMAX;
                DIV;
                DM(VOLT2BITS) = F0;

{ Calculate A/D conversion factor                                              }
                F12 = ADBITSMAX;
                F0 = ADVOLTSMAX;
                DIV;
                DM(BITS2VOLT) = F0;

{ Setup variables required to saving data in memory for later recall           }
                R0 = SAVEOFFSET;
                DM(INPUT_OFFSET) = R0;
                R1 = ASHIFT R0 BY 1;
                R2 = 0x8000;
                R2 = R2-R1;
                R2 = ASHIFT R2 BY -1;
                R1 = R2+R0;
                DM(OUTPUT_OFFSET) = R1;
                R1 = R1+R2;
                DM(SAVE_END) = R1;
                R0 = DELAY;
                DM(DELAY_COUNT) = R0;

{ Initialize A/D offset variables                                              }
                B0 = AD0OFF;
                R0 = AD0OFFSET;
                DM(I0,1) = R0;
                R0 = AD1OFFSET;
                DM(I0,1) = R0;
                R0 = AD2OFFSET;
                DM(I0,1) = R0;
                R0 = AD3OFFSET;
                DM(I0,1) = R0;
                R0 = AD4OFFSET;
                DM(I0,1) = R0;

{ Initialize A/D offset variables                                              }
                B0 = AXIAL_MAX_CUR;
                R0 = AXIAL_MAXU;
                DM(I0,1) = R0;
                R0 = RAD1X_MAXU;
                DM(I0,1) = R0;
                R0 = RAD1Y_MAXU;
                DM(I0,1) = R0;
                R0 = RAD2X_MAXU;
                DM(I0,1) = R0;
                R0 = RAD2Y_MAXU;
                DM(I0,1) = R0;

{ Initialize D/A offset variables                                              }
                B0 = DA0OFF;
                R0 = DA0OFFSET;
                DM(I0,1) = R0;
                R0 = DA1OFFSET;
                DM(I0,1) = R0;
                R0 = DA2OFFSET;
```

```
                DM(I0,1) = R0;
                R0 = DA3OFFSET;
                DM(I0,1) = R0;
                R0 = DA4OFFSET;
                DM(I0,1) = R0;

{ Initialize D/A current offset variables                                    }
                B0 = DA0CRTOFF;
                F1 = DM(VOLT2BITS);
                F0 = DA0CUROFF;
                F0 = F0*F1;
                DM(I0,1) = F0;
                F0 = DA1CUROFF;
                F0 = F0*F1;
                DM(I0,1) = F0;
                F0 = DA2CUROFF;
                F0 = F0*F1;
                DM(I0,1) = F0;
                F0 = DA3CUROFF;
                F0 = F0*F1;
                DM(I0,1) = F0;
                F0 = DA4CUROFF;
                F0 = F0*F1;
                DM(I0,1) = F0;

{ Initialize all control signals to zero                                     }
                B0 = DA0OFF;
                B1 = DA0CONTROL;
                F0 = DABITSMAX;
                R1 = FIX F0;
                LCNTR = 5;
                DO adj UNTIL LCE;
                  R0 = DM(I0,1);
                  R0 = R0+R1;
adj:              DM(I1,1) = R0;

#ifndef NO_FILTER
{ Initialize low-pass filter constants                                       }

                LCNTR = 5;
                DO dofil2 UNTIL LCE;
                  B0 = FIL0XCN1;
                  R3 = CURLCNTR;
                  R4 = R3-1;
                  IF NE JUMP fil2_1;
                  M3 = 4;
                  F12 = FIL4K1;
                  F1 = FIL4K2;
                  F2 = FIL4K3;
                  F3 = FIL4K4;
                  F4 = FIL4K5;
                  F5 = FIL4N2;
                  F6 = FIL4N3;
                  F8 = FIL4N4;
                  F9 = FIL4N5;
                  JUMP calcfil2;
fil2_1:
                  R4 = R4-1;
                  IF NE JUMP fil2_2;
                  M3 = 3;
                  F12 = FIL3K1;
                  F1 = FIL3K2;
                  F2 = FIL3K3;
                  F3 = FIL3K4;
                  F4 = FIL3K5;
                  F5 = FIL3N2;
                  F6 = FIL3N3;
                  F8 = FIL3N4;
                  F9 = FIL3N5;
```

```
                    JUMP calcfil2;
fil2_2:
                    R4 = R4-1;
                    IF NE JUMP fil2_3;
                    M3 = 2;
                    F12 = FIL2K1;
                    F1 = FIL2K2;
                    F2 = FIL2K3;
                    F3 = FIL2K4;
                    F4 = FIL2K5;
                    F5 = FIL2N2;
                    F6 = FIL2N3;
                    F8 = FIL2N4;
                    F9 = FIL2N5;
                    JUMP calcfil2;
fil2_3:
                    R4 = R4-1;
                    IF NE JUMP fil2_4;
                    M3 = 1;
                    F12 = FIL1K1;
                    F1 = FIL1K2;
                    F2 = FIL1K3;
                    F3 = FIL1K4;
                    F4 = FIL1K5;
                    F5 = FIL1N2;
                    F6 = FIL1N3;
                    F8 = FIL1N4;
                    F9 = FIL1N5;
                    JUMP calcfil2;
fil2_4:
                    M3 = 0;
                    F12 = FIL0K1;
                    F1 = FIL0K2;
                    F2 = FIL0K3;
                    F3 = FIL0K4;
                    F4 = FIL0K5;
                    F5 = FIL0N2;
                    F6 = FIL0N3;
                    F8 = FIL0N4;
                    F9 = FIL1N5;

calcfil2:
                    F0 = DM(I0,M3);
                    F0 = 1.0;
                    DIV;
                    DM(I0,10) = F0;
                    F4 = F4*F0;
                    F4 = -F4;
                    DM(I0,5) = F4;
                    DM(I0,10) = F9;
                    F3 = F3*F0;
                    F3 = -F3;
                    DM(I0,5) = F3;
                    DM(I0,10) = F8;
                    F2 = F2*F0;
                    F2 = -F2;
                    DM(I0,5) = F2;
                    DM(I0,10) = F6;
                    F1 = F1*F0;
                    F1 = -F1;
                    DM(I0,5) = F1;
dofil2:             DM(I0,5) = F5;
#endif

#ifndef NO_INOTCH_FILTER
{ Initialize notch filter constants                                    }

                LCNTR = 5;
                DO doinfil UNTIL LCE;
```

```
                    B0 = INFIL0DK2;
                    R3 = CURLCNTR;
                    R4 = R3-1;
                    IF NE JUMP infil_1;
                    M3 = 4;
                    F1 = INFIL4ALPHA;
                    F2 = INFIL4L;
                    F3 = INFIL4A0;
                    F4 = INFIL4COSWT;
                    JUMP calcinfil;
infil_1:
                    R4 = R4-1;
                    IF NE JUMP infil_2;
                    M3 = 3;
                    F1 = INFIL3ALPHA;
                    F2 = INFIL3L;
                    F3 = INFIL3A0;
                    F4 = INFIL3COSWT;
                    JUMP calcinfil;
infil_2:
                    R4 = R4-1;
                    IF NE JUMP infil_3;
                    M3 = 2;
                    F1 = INFIL2ALPHA;
                    F2 = INFIL2L;
                    F3 = INFIL2A0;
                    F4 = INFIL2COSWT;
                    JUMP calcinfil;
infil_3:
                    R4 = R4-1;
                    IF NE JUMP infil_4;
                    M3 = 1;
                    F1 = INFIL1ALPHA;
                    F2 = INFIL1L;
                    F3 = INFIL1A0;
                    F4 = INFIL1COSWT;
                    JUMP calcinfil;
infil_4:
                    M3 = 0;
                    F1 = INFIL0ALPHA;
                    F2 = INFIL0L;
                    F3 = INFIL0A0;
                    F4 = INFIL0COSWT;

calcinfil:
                    F0 = DM(I0,M3);
                    F0 = F1*F2;
                    F2 = F3;
                    F3 = F3*F2;
                    F0 = F0*F3;
                    F12 = 4.0;
                    DIV;
                    F1 = 1.0;
                    F0 = F1-F0;
                    F1 = F0;
                    F0 = F0*F1;
                    F0 = -F0;
                    DM(I0,5) = F0;
                    F2 = 2.0;
                    F2 = F2*F4;
                    F0 = F1*F2;
                    DM(I0,5) = F0;
                    F2 = -F2;
doinfil:            DM(I0,5) = F2;
#endif

#ifndef NO_NOTCH_FILTER
{ Initialize notch filter constants                                    }
```

```
                LCNTR = 5;
                DO donfil UNTIL LCE;
                  B0 = NFIL0DK2;
                  R3 = CURLCNTR;
                  R4 = R3-1;
                  IF NE JUMP nfil_1;
                  M3 = 4;
                  F1 = NFIL4ALPHA;
                  F2 = NFIL4L;
                  F3 = NFIL4A0;
                  F4 = NFIL4COSWT;
                  JUMP calcnfil;
nfil_1:
                  R4 = R4-1;
                  IF NE JUMP nfil_2;
                  M3 = 3;
                  F1 = NFIL3ALPHA;
                  F2 = NFIL3L;
                  F3 = NFIL3A0;
                  F4 = NFIL3COSWT;
                  JUMP calcnfil;
nfil_2:
                  R4 = R4-1;
                  IF NE JUMP nfil_3;
                  M3 = 2;
                  F1 = NFIL2ALPHA;
                  F2 = NFIL2L;
                  F3 = NFIL2A0;
                  F4 = NFIL2COSWT;
                  JUMP calcnfil;
nfil_3:
                  R4 = R4-1;
                  IF NE JUMP nfil_4;
                  M3 = 1;
                  F1 = NFIL1ALPHA;
                  F2 = NFIL1L;
                  F3 = NFIL1A0;
                  F4 = NFIL1COSWT;
                  JUMP calcnfil;
nfil_4:
                  M3 = 0;
                  F1 = NFIL0ALPHA;
                  F2 = NFIL0L;
                  F3 = NFIL0A0;
                  F4 = NFIL0COSWT;

calcnfil:
                  F0 = DM(I0,M3);
                  F0 = F1*F2;
                  F2 = F3;
                  F3 = F3*F2;
                  F0 = F0*F3;
                  F12 = 4.0;
                  DIV;
                  F1 = 1.0;
                  F0 = F1-F0;
                  F1 = F0;
                  F0 = F0*F1;
                  F0 = -F0;
                  DM(I0,5) = F0;
                  F2 = 2.0;
                  F2 = F2*F4;
                  F0 = F1*F2;
                  DM(I0,5) = F0;
                  F2 = -F2;
donfil:           DM(I0,5) = F2;
#endif

{-----------------------------------------------------------------}
```

```
{ Initialize all bearing constants.                                    }
{                                                                      }
{ Constants:                                                           }
{       ?????_A21                                                      }
{       ?????_A22                                                      }
{       ?????_CONV1                                                    }
{       ?????_CONV2                                                    }
{       ?????_PERIOD2                                                  }
{       ?????_BPLUS                                                    }
{                                                                      }
{ Uses:                                                                }
{       ?????_DAMP                                                     }
{       ?????_NATFREQ                                                  }
{       ?????_SENGAIN                                                  }
{       ?????_AMPGAIN                                                  }
{       ?????_PERIOD                                                   }
{       ?????_BPLUS                                                    }
{----------------------------------------------------------------------}

{ set-up bearing constant a21 (a1m)     }
{ a21 = 2*?????_damp*?????_natfreq      }

                F0 = 2.0;
                B0 = AXIAL_A21;
                LCNTR = 5;
                DO doa21 UNTIL LCE;
                  R3 = CURLCNTR;
                  R4 = R3-1;
                  IF NE JUMP rad1_a21;
                  F1 = RAD2Y_DAMP;
                  F2 = RAD2Y_NATFREQ;
                  JUMP calca21;
rad1_a21:
                  R4 = R4-1;
                  IF NE JUMP rad2_a21;
                  F1 = RAD2X_DAMP;
                  F2 = RAD2X_NATFREQ;
                  JUMP calca21;
rad2_a21:
                  R4 = R4-1;
                  IF NE JUMP rad3_a21;
                  F1 = RAD1Y_DAMP;
                  F2 = RAD1Y_NATFREQ;
                  JUMP calca21;
rad3_a21:
                  R4 = R4-1;
                  IF NE JUMP rad4_a21;
                  F1 = RAD1X_DAMP;
                  F2 = RAD1X_NATFREQ;
                  JUMP calca21;
rad4_a21:
                  F1 = AXIAL_DAMP;
                  F2 = AXIAL_NATFREQ;

calca21:          F3 = F0*F1;
                  F3 = F3*F2;

doa21:            DM(I0,15) = F3;


{ set-up bearing constant a22 (a2m)     }
{ a22 = ?????_natfreq*?????_natfreq     }

                B0 = AXIAL_A22;
                LCNTR = 5;
                DO doa22 UNTIL LCE;
                  R3 = CURLCNTR;
                  R4 = R3-1;
                  IF NE JUMP rad1_a22;
```

```
                    F1 = RAD2Y_NATFREQ;
                    F2 = RAD2Y_NATFREQ;
                    JUMP calca22;
rad1_a22:
                    R4 = R4-1;
                    IF NE JUMP rad2_a22;
                    F1 = RAD2X_NATFREQ;
                    F2 = RAD2X_NATFREQ;
                    JUMP calca22;
rad2_a22:
                    R4 = R4-1;
                    IF NE JUMP rad3_a22;
                    F1 = RAD1Y_NATFREQ;
                    F2 = RAD1Y_NATFREQ;
                    JUMP calca22;
rad3_a22:
                    R4 = R4-1;
                    IF NE JUMP rad4_a22;
                    F1 = RAD1X_NATFREQ;
                    F2 = RAD1X_NATFREQ;
                    JUMP calca22;
rad4_a22:
                    F1 = AXIAL_NATFREQ;
                    F2 = AXIAL_NATFREQ;

calca22:            F3 = F1*F2;

doa22:              DM(I0,15) = F3;


{ set-up bearing conversion factor for A/D                }
{ ?????_conv1 = bits2volt/?????_sengain)                  }
{   ?????_sengain - position sensor conversion factor     }

                  B0 = AXIAL_CONV1;
                  LCNTR = 5;
                  DO doconv1 UNTIL LCE;
                    F0 = DM(BITS2VOLT);
                    R3 = CURLCNTR;
                    R4 = R3-1;
                    IF NE JUMP rad1_conv1;
                    F12 = RAD2Y_SENGAIN;
                    JUMP calcconv1;
rad1_conv1:
                    R4 = R4-1;
                    IF NE JUMP rad2_conv1;
                    F12 = RAD2X_SENGAIN;
                    JUMP calcconv1;
rad2_conv1:
                    R4 = R4-1;
                    IF NE JUMP rad3_conv1;
                    F12 = RAD1Y_SENGAIN;
                    JUMP calcconv1;
rad3_conv1:
                    R4 = R4-1;
                    IF NE JUMP rad4_conv1;
                    F12 = RAD1X_SENGAIN;
                    JUMP calcconv1;
rad4_conv1:
                    F12 = AXIAL_SENGAIN;

calcconv1:          DIV;

doconv1:            DM(I0,15) = F0;


{ set-up bearing conversion factor for D/A                    }
{ ?????_conv2 = volt2bits/?????_ampgain                       }
{   ?????_ampgain - current to D/A volts conversion factor    }
```

```
                B0 = AXIAL_CONV2;
                LCNTR = 5;
                DO doconv2 UNTIL LCE;
                  F0 = DM(VOLT2BITS);
                  R3 = CURLCNTR;
                  R4 = R3-1;
                  IF NE JUMP rad1_conv2;
                  F12 = RAD2Y_AMPGAIN;
                  JUMP calcconv2;
rad1_conv2:
                  R4 = R4-1;
                  IF NE JUMP rad2_conv2;
                  F12 = RAD2X_AMPGAIN;
                  JUMP calcconv2;
rad2_conv2:
                  R4 = R4-1;
                  IF NE JUMP rad3_conv2;
                  F12 = RAD1Y_AMPGAIN;
                  JUMP calcconv2;
rad3_conv2:
                  R4 = R4-1;
                  IF NE JUMP rad4_conv2;
                  F12 = RAD1X_AMPGAIN;
                  JUMP calcconv2;
rad4_conv2:
                  F12 = AXIAL_AMPGAIN;

calcconv2:        DIV;

doconv2:          DM(I0,15) = F0;


{ set-up axial bearing 2*PERIOD constant        }

                F0 = SMPLSPEED;
                F12 = AXIAL_PERIOD;

                DIV;

                DM(AXIAL_PERIOD2) = F0;


{ set-up radial bearing 2*PERIOD constant }

                F0 = SMPLSPEED;
                F12 = RAD_PERIOD;

                DIV;

                DM(RAD1X_PERIOD2) = F0;
                DM(RAD1Y_PERIOD2) = F0;
                DM(RAD2X_PERIOD2) = F0;
                DM(RAD2Y_PERIOD2) = F0;


{ set-up radial bearing BPLUS constant }

                B0 = AXIAL_BPLUS;
                LCNTR = 5;
                DO dobplus UNTIL LCE;
                  F0 = 1.0;
                  R4 = CURLCNTR;
                  R4 = R4-1;
                  IF NE JUMP rad1_bplus;
                  F12 = RAD2Y_BPLS;
                  JUMP calcbplus;
rad1_bplus:
                  R4 = R4-1;
                  IF NE JUMP rad2_bplus;
```

```
                    F12 = RAD2X_BPLS;
                    JUMP calcbplus;
rad2_bplus:
                    R4 = R4-1;
                    IF NE JUMP rad3_bplus;
                    F12 = RAD1Y_BPLS;
                    JUMP calcbplus;
rad3_bplus:
                    R4 = R4-1;
                    IF NE JUMP rad4_bplus;
                    F12 = RAD1X_BPLS;
                    JUMP calcbplus;
rad4_bplus:
                    F12 = AXIAL_BPLS;

calcbplus:          DIV;

dobplus:            DM(I0,15) = F0;


{-------------------------------------------------------------}
{ initialize I/O board                                        }
{                                                             }
{ DM(40000000) - DM(DAFIFO)                                   }
{       Read  - undefined                                     }
{       Write - write data to D/A Fifo                        }
{               14-bit, sign extended, right justified        }
{                                                             }
{ DM(40000001) - DM(CHANNELS)                                 }
{       Read  - undefined                                     }
{       Write - number of A/D and D/A channels                }
{               bits 2-0: number of A/D channels              }
{               bits 5-3: number of D/A channels              }
{                                                             }
{ DM(40000002) - DM(CONTROL)                                  }
{       Read  - undefined                                     }
{       Write - I/O board control register                    }
{               bits 2-0: go mode                             }
{               bits 4-3: IRQ mode                            }
{               bit    5: analog reset                        }
{                                                             }
{ PM(800000) - PM(ADFIFO)                                     }
{       Read  - read data from A/D                            }
{               14-bit, sign extended, right-justified        }
{       Write - undefined                                     }
{                                                             }
{ PM(800001) - PM(ADSTAT)                                     }
{       Read  - A/D conversion status register                }
{               bit 0: D/A busy = 1                           }
{               bit 1: A/D busy = 1                           }
{               bit 2: D/A FIFO empty = 0                     }
{               bit 3: A/D FIFO empty = 0                     }
{               bit 4: A/D FIFO full = 0                      }
{               bit 5: D/A FIFO full = 0                      }
{       Write - undefined                                     }
{                                                             }
{-------------------------------------------------------------}

{-----------------------------------------------------------------------}
{ Set angular reset state to low.  When in this state, we can safely    }
{ set the number of A/Ds and D/As.  We can also safely set the go-mode  }
{ and/or the interrupt mode.                                            }
{                                                                       }
{ Control Register:  [default - DM(0x40000002)]                         }
{       bits 2-0 -> go mode                                             }
{       bits 5-3 -> IRQ mode                                            }
{       bit    6 -> Status select                                       }
{       bit    7 -> Analog reset                                        }
{                                                                       }
```

```
{ Go-Mode:  selects the type of event that triggers IO conversion.      }
{       0 -> trigger conversion on high level of FLAG2                  }
{       1 -> trigger conversion on high level of TIMEXP                 }
{       2 -> trigger conversion after ADBUSY and DABUSY flags are low   }
{       3 -> trigger conversion when D/A FIFO is not empty              }
{       4 -> trigger conversion when A/D FIFO is empty                  }
{       5 -> trigger conversion when external trigger is high           }
{                                                                       }
{ IR-Mode:  selects what event will trigger an interrupt to the DSP     }
{           board.  Either IRQ2 or IRQ3 are used depending on jumpers.  }
{           Interrupts are generated when line is asserted low.         }
{       0 -> interrupt generated when ADBUSY and DABUSY are 0           }
{       1 -> interrupt generated when A/D FIFO is not empty             }
{       2 -> interrupt generated when D/A FIFO is empty                 }
{       3 -> interrupt generated when A/D FIFO is full                  }
{       4 -> interrupt generated when D/A FIFO is full                  }
{       5 -> interrupt generated when A/D overflows                     }
{       6 -> interrupt generated when external trigger is high          }
{                                                                       }
{-----------------------------------------------------------------------}

                R0 = 0x0;
                DM(CONTROL) = R0;

{-----------------------------------------------------------------------}
{ Set BEARTOT A/D channels and BEARTOT D/A channels.  When a conversion }
{ is triggered, both the A/D and D/A conversions are triggered.         }
{                                                                       }
{ Number of Channels Register:  [default - DM(0x40000001)]              }
{       bits 2-0 -> number of A/D channels                              }
{       bits 5-3 -> number of D/A channels                              }
{-----------------------------------------------------------------------}

{ send initial control signals first so set number of A/Ds to zero      }
                R0 = BEARTOT;
                R1 = LSHIFT R0 BY 3;
                DM(CHANNELS) = R1;

{-----------------------------------------------------------------------}
{ We are done so we can turn off angular reset                          }
{                                                                       }
{ Control Register:  [default - DM(0x40000002)]                         }
{       bits 2-0 -> go mode                                             }
{       bits 5-3 -> IRQ mode                                            }
{       bit    6 -> Status select                                       }
{       bit    7 -> Analog reset                                        }
{-----------------------------------------------------------------------}

                R0 = 0x80;
                DM(CONTROL) = R0;


{ send initial bearing control signals  }

                B0 = DA0CONTROL;
                B1 = AXIAL_CONV2;
                B2 = DA0CRTOFF;
                B3 = AXIAL_U1;
                LCNTR = BEARTOT;
                DO initbear UNTIL LCE;
                  R4 = CURLCNTR;
                  R4 = R4-1;
                  IF NE JUMP rad1_init1;
                  F0 = RAD2Y_MAXU;
                  F1 = RAD2Y_INITSIG;
                  JUMP calcinit1;
rad1_init1:
                  R4 = R4-1;
                  IF NE JUMP rad2_init1;
```

```
                    F0 = RAD2X_MAXU;
                    F1 = RAD2X_INITSIG;
                    JUMP calcinit1;
rad2_init1:
                    R4 = R4-1;
                    IF NE JUMP rad3_init1;
                    F0 = RAD1Y_MAXU;
                    F1 = RAD1Y_INITSIG;
                    JUMP calcinit1;
rad3_init1:
                    R4 = R4-1;
                    IF NE JUMP rad4_init1;
                    F0 = RAD1X_MAXU;
                    F1 = RAD1X_INITSIG;
                    JUMP calcinit1;
rad4_init1:
                    F0 = AXIAL_MAXU;
                    F1 = AXIAL_INITSIG;

calcinit1:
                    COMP(F1,F0);
                    IF GT F1 = F0;
                    F0 = -F0;
                    COMP(F1,F0);
                    IF LT F1 = F0;

                    DM(I3,15) = F1;                 { ?????_U1      }

                    F3 = DM(I1,2);                  { ?????_CONV2   }
                    F1 = F1*F3;

                    F3 = DM(I2,5);                  { DA?CRTOFF     }
                    F1 = F1+F3;

                    R3 = DM(I2,-4);                 { DA?OFF        }
                    F3 = FLOAT R3;

                    F1 = F1+F3;
                    F2 = DABITSMAX;
                    F1 = F2+F1;
                    R0 = FIX F1;

initbear:           DM(I0,1) = R0;                  { DA?CONTROL    }


{-------------------------------------------------------------------------}
{ Send D/A value(s) to FIFO.  [default - DM(0x40000000)]                  }
{                                                                         }
{ The values sent to the D/A FIFO follow an odd convention which is       }
{ best shown by the value range.  The range of values for our particular}
{ D/A is:                                                                 }
{       +5V = 1111 1111 1111 1111                                         }
{        0V = 1000 0000 0000 0000                                         }
{       -5V = 0000 0000 0000 0000                                         }
{                                                                         }
{ Sample code for converting a FIFO value to floating point is as         }
{ follows:                                                                }
{               R0 = PM(ADFIFO);                                          }
{               R1 = ASHIFT R0 BY 16;                                     }
{               R1 = PASS R1;                                             }
{               IF LT R0 = BCLR R1 BY 31;                                 }
{               IF GE R0 = BSET R1 BY 31;                                 }
{               R1 = ASHIFT R0 BY -18;                                    }
{               F0 = FLOAT R1;                                            }
{                                                                         }
{ Sample code for converting floating point numbers to a FIFO value is    }
{ as follows:                                                             }
{               R7 = 8192;                                                }
{               R1 = FIX F0;                                              }
```

```
{               R0 = R1+R7;                                                 }
{               R0 = ASHIFT R0 BY 2;                                        }
{-----------------------------------------------------------------------}

{ Initial values have been calculated, so lets send them out       }

                B0 = DA0CONTROL;
                LCNTR = BEARTOT;
                DO initsend1 UNTIL LCE;
                  R0 = DM(I0,1);
initsend1:        DM(DAFIFO) = R0;


{-----------------------------------------------------------------------}
{ Wait for A/D conversion                                               }
{                                                                       }
{ Status Register:  [default - PM(0x800001)]                            }
{       bits 0 -> 1 if D/A is busy                                      }
{       bits 1 -> 1 if A/D is busy                                      }
{       bit  2 -> 0 if D/A FIFO is empty                                }
{       bit  3 -> 0 if A/D FIFO is empty                                }
{       bit  4 -> 0 if D/A FIFO is full                                 }
{       bit  5 -> 0 if A/D FIFO is full                                 }
{                                                                       }
{ Note:  Make sure that both D/A and A/D are not busy before reading    }
{        from A/D FIFO otherwise bogus values are obtained.             }
{-----------------------------------------------------------------------}

{ Toggle FLAG2 to signal I/O board to start conversion                  }

                BIT SET ASTAT 0x00200000;
                NOP; NOP;
                BIT CLR ASTAT 0x00200000;

                R1 = 0x03;
wait1:          R0 = PM(IOSTAT);
                R7 = R0 AND R1;
                IF NE JUMP wait1;

{-----------------------------------------------------------------------}
{ Get A/D value(s) from FIFO.  [default - PM(0x800000)]                 }
{                                                                       }
{ The values obtained are 14-bit 2^s-complement values sign extended to }
{ the left.  2^s complement is obtained by negating the real value and  }
{ adding 1.  The range of values for our particular A/D is:             }
{                                                                       }
{       +5V = 0001 1111 1111 1111                                       }
{        0V = 0000 0000 0000 0000                                       }
{       -5V = 1110 0000 0000 0001                                       }
{                                                                       }
{ Sample code for converting 2^s complement notation to floating point  }
{ is as follows:                                                        }
{               R1 = FEXT R0 BY 0:14 (SE);                              }
{               F0 = FLOAT R1;                                          }
{                                                                       }
{ Sample code for converting floatin point numbers to 2^s complement    }
{ notation is as follows:                                               }
{               R0 = FIX F0;                                            }
{                                                                       }
{-----------------------------------------------------------------------}

{ We have sent the initial control signal, now it is okay to read the   }
{ initial positions reported by the sensors so that derivatives can be  }
{ calculated.                                                           }

{-----------------------------------------------------------------------}
{ Set angular reset state to low.  When in this state, we can safely    }
{ set the number of A/Ds and D/As.  We can also safely set the go-mode  }
{ and/or the interrupt mode.                                            }
```

```
{-------------------------------------------------------------------------}

                R0 = 0x0;
                DM(CONTROL) = R0;

{ get initial position signals so set number of D/As to zero            }
                R0 = BEARTOT;
                DM(CHANNELS) = R0;

{-------------------------------------------------------------------------}
{ We are done so we can turn off angular reset                           }
{-------------------------------------------------------------------------}

                R0 = 0x80;
                DM(CONTROL) = R0;

{ Toggle FLAG2 to signal I/O board to start conversion                   }

                BIT SET ASTAT 0x00200000;
                NOP; NOP;
                BIT CLR ASTAT 0x00200000;

                R1 = 0x03;
wait2:          R0 = PM(IOSTAT);
                R7 = R0 AND R1;
                IF NE JUMP wait2;

{ Read A/D values and store                                              }

                LCNTR = BEARTOT;
                B0 = AD0INPUT;
                B1 = AD0OFF;
                B2 = AXIAL_X1;
                B3 = AXIAL_CONV1;
                DO getinit UNTIL LCE;
                  R0 = PM(ADFIFO);
                  DM(I0,1) = R0;                  { AD?INPUT      }
                  R1 = FEXT R0 BY 0:14 (SE);
                  R3 = DM(I1,1);                  { AD?OFF        }
                  R1 = R1+R3;
                  F0 = FLOAT R1;
                  F2 = DM(I3,15);                 { ?????_CONV1   }
                  F1 = F0*F2;
                  DM(I2,1) = F1;                  { ?????_X1      }
getinit:          DM(I2,14) = F1;                 { ?????_X2      }


{ initialize axial data locations to zero       }

                F0 = 0.0;
                DM(AXIAL_V2) = F0;
                DM(AXIAL_V1) = F0;
                DM(AXIAL_V0) = F0;
                DM(AXIAL_A0) = F0;

{ initialize first radial data locations to zero        }

                DM(RAD1X_V2) = F0;
                DM(RAD1X_V1) = F0;
                DM(RAD1X_V0) = F0;
                DM(RAD1X_A0) = F0;
                DM(RAD1Y_V2) = F0;
                DM(RAD1Y_V1) = F0;
                DM(RAD1Y_V0) = F0;
                DM(RAD1Y_A0) = F0;

{ initialize second radial data locations to zero       }

                DM(RAD2X_V2) = F0;
```

```
                DM(RAD2X_V1) = F0;
                DM(RAD2X_V0) = F0;
                DM(RAD2X_A0) = F0;
                DM(RAD2Y_V2) = F0;
                DM(RAD2Y_V1) = F0;
                DM(RAD2Y_V0) = F0;
                DM(RAD2Y_A0) = F0;

#ifndef NO_FILTER
                LCNTR = 5;
                B0 = FIL0X4;
                F0 = 0.0;
                M3 = -44;
                DO filinit UNTIL LCE;
                  DM(I0,15) = F0;
                  DM(I0,15) = F0;
                  DM(I0,15) = F0;
filinit:          DM(I0,M3) = F0;
#endif

#ifndef NO_INOTCH_FILTER
                LCNTR = 5;
                B0 = INFIL0X2;
                F0 = 0.0;
                DO infilinit UNTIL LCE;
                  DM(I0,5) = F0;
infilinit:        DM(I0,-4) = F0;
#endif

#ifndef NO_NOTCH_FILTER
                LCNTR = 5;
                B0 = NFIL0X2;
                F0 = 0.0;
                DO nfilinit UNTIL LCE;
                  DM(I0,5) = F0;
nfilinit:         DM(I0,-4) = F0;
#endif

{ initialize indirect addressing mode registers for radial bearings }

                R0 = 0;
                DM(BEAR_NUM) = R0;

{-------------------------------------------------------------------------}
{ initialize bearing sampling period of ADSP timer.  The timer is         }
{ assumed to run at the same speed as the ADSP chip (33.33 MHZ)           }
{-------------------------------------------------------------------------}

{ multiply SMPLSPEED by BEARTOT                                           }

                R1 = BEARTOT;
                F0 = FLOAT R1;
                F12 = SMPLSPEED;
                F12 = F0*F12;

                F0 = CPUSPEED;
                DIV;
                R1 = FIX F0;
                R0 = R1-1;

                TPERIOD = R0;
                TCOUNT = R0;


{ Set up input/output save pointers in last address registers             }
                I6 = DM(INPUT_OFFSET);
                I7 = DM(OUTPUT_OFFSET);

{ Save initial input/output values                                        }
```

```
#ifdef RAW
                R0 = DM(DELAY_COUNT);
                R1 = PASS R0;
                IF NE JUMP initskip;
                R0 = I6;
                R1 = DM(OUTPUT_OFFSET);
                R0 = R1-R0;
                IF EQ JUMP initskip;
                M3 = SAVEBEAR;
                B0 = AD0INPUT;
                R0 = DM(M3,I0);
                DM(I6,1) = R0;
                B0 = DA0CONTROL;
                R0 = DM(M3,I0);
                DM(I7,1) = R0;
#endif

#ifdef PLOT
                R0 = DM(DELAY_COUNT);
                R1 = PASS R0;
                IF NE JUMP initskip;
                R0 = I6;
                R1 = DM(OUTPUT_OFFSET);
                R0 = R1-R0;
                IF EQ JUMP initskip;
                M3 = SAVEBEAR;
                B0 = AD0INPUT;
                R0 = DM(M3,I0);
                R0 = FEXT R0 BY 0:14 (SE);
                DM(I6,1) = R0;
                B0 = DA0CONTROL;
                R0 = DM(M3,I0);
                R1 = 0XF000;
                R0 = R0-R1;
                DM(I7,1) = R0;
#endif

{ Reset interrupt latch register                                    }

initskip:
                IRPTL = 0x0;

{ Allow timer interrupts                                            }

                BIT SET IMASK 0x12;

{ Turn on timer                                                     }

                BIT SET MODE2 0x20;

{ Allow interrupt generation                                        }

                BIT SET MODE1 0x1000;

gag:            IDLE;
                JUMP gag;


{ radial bearing control loop }

rad_calc:

                R1 = 0x03;
rwait1:         R0 = PM(IOSTAT);
                R2 = R0 AND R1;
                IF NE JUMP rwait1;

{ Read A/D values and store                                         }
```

```
                B2 = AD0INPUT;
                LCNTR = BEARTOT, DO rread1 UNTIL LCE;
                  R0 = PM(ADFIFO);
rread1:           DM(I2,1) = R0;

                B2 = AD0INPUT;
                R0 = DM(I2,M3);                 { DUMMY READ     }
                R0 = DM(I2,5);                  { AD?INPUT       }

                R1 = FEXT R0 BY 0:14 (SE);
                R2 = DM(I2,5);                  { AD?OFF         }
                R1 = R1+R2;
#ifdef NO_FILTER
                F0 = FLOAT R1, F2 = DM(I1,1);   { ?????_CONV1    }
#else
                F0 = FLOAT R1, F4 = DM(I2,5);   { FIL?XCN1       }
                F0 = F0*F4, F4 = DM(I2,5);      { FIL?X4         }
                F5 = PASS F4, F1 = DM(I2,5);    { FIL?XCN5       }
                F1 = F1*F4, F3 = DM(I2,5);      { FIL?YCN5       }
                F3 = F3*F4, F4 = DM(I2,5);      { FIL?X3         }
                F0 = F0+F1, F1 = DM(I2,-20);    { FIL?XCN4       }
                F5 = F5+F3, DM(I2,25) = F4;     { FIL?X4         }
                F1 = F1*F4, F3 = DM(I2,5);      { FIL?YCN4       }
                F3 = F3*F4, F4 = DM(I2,5);      { FIL?X2         }
                F0 = F0+F1, F1 = DM(I2,-20);    { FIL?XCN3       }
                F5 = F5+F3, DM(I2,25) = F4;     { FIL?X3         }
                F1 = F1*F4, F3 = DM(I2,5);      { FIL?YCN3       }
                F3 = F3*F4, F4 = DM(I2,5);      { FIL?X1         }
                F0 = F0+F1, F1 = DM(I2,-20);    { FIL?XCN2       }
                F5 = F5+F3, DM(I2,25) = F4;     { FIL?X2         }
                F1 = F1*F4, F3 = DM(I2,-10);    { FIL?YCN2       }
                F3 = F3*F4;
                F0 = F0+F1;
                F5 = F5+F3, DM(I2,15) = F0;     { FIL?X1         }
#ifdef NO_INOTCH_FILTER
                F0 = F0+F5, F2 = DM(I1,1);      { ?????_CONV1    }
#else
                F0 = F0+F5;
#endif
#endif

#ifndef NO_INOTCH_FILTER
                F2 = DM(I2,10);                 { INFIL?X2        }
                F4 = PASS F2, F3 = DM(I2,5);    { INFIL?DK2       }
                F2 = F2*F3, F3 = DM(I2,-10);    { INFIL?DK1       }
                F0 = F0+F2, F2 = DM(I2,15);     { INFIL?X1        }
                F5 = F2*F3, F3 = DM(I2,-20);    { INFIL?NK1       }
                F0 = F0+F5, DM(I2,5) = F2;      { INFIL?X2        }
                F2 = F2*F3, DM(I2,20) = F0;     { INFIL?X1        }
                F0 = F0+F2;
                F0 = F0+F4, F2 = DM(I1,1);
#endif

{ multiply by conversion factor to get proper units     }

                F0 = F2*F0;

{ store X0      }

                DM(I0,2) = F0;                  { ?????_X0       }

{ determine velocity using central difference formula   }

                F1 = DM(I0,1);                  { ?????_X2       }
                F0 = F0-F1, F2 = DM(I1,0);      { ?????_PERIOD2  }

                F0 = F0*F2;

{ store velocity value  }
```

```
                DM(I0,2) = F0;                  { ?????_V0      }

{ determine acceleration using smoothing differentiation formula        }
{ accel = (1/6T)*[v(k)+3v(k-1)-3v(k-2)-v(k-3)]                          }

{ determine acceleration using central difference formula }

                F1 = DM(I0,1);                  { ?????_V2      }

                F0 = F0-F1, F2 = DM(I1,1);      { ?????_PERIOD2 }

                F0 = F0*F2, F1 = DM(I1,1);      { ?????_A22     }

{ store acceleration value }

                DM(I0,-6) = F0;                 { ?????_A0      }

{-------------------------------------------------------------}
{ Compute control function                                    }
{                                                             }
{ u(t) = u(t-1) - (1/bplus)*[a(t) + (a21*v(t)) + ((a22*x(t))] }
{                                                             }
{ For an ideal second order system                            }
{                                                             }
{               a21 = 2.0*rad??_damp*rad??_natfreq            }
{               a22 = rad??_natfreq*rad??_natfreq             }
{-------------------------------------------------------------}

{ determine a22*x(t) }

                F2 = DM(I0,3);                  { ?????_X0      }

                F3 = F2*F1, F1 = DM(I0,4);      { ?????_V0      }

{ determine a(t) + (a22*x(t)) }

                F3 = F0+F3, F0 = DM(I1,1);      { ?????_A21     }

{ determine a21*v(t) }

                F0 = F0*F1;

{ determine a(t) + (a22*x(t)) + (a21*v(t)) }

                F3 = F3+F0, F0 = DM(I1,1);      { ?????_BPLUS   }

{ divide temporary result by AXIAL_BPLUS }

                F3 = F3*F0, F0 = DM(I0,1);      { ?????_U1      }

{ get final value of u(t) }

                F1 = F0-F3, F2 = DM(I2,5);      { ?????_MAX_CUR }

{ test if u(t) greater than maximum allowable amperage  }

                COMP(F1,F2), F0 = DM(I1,-6);    { ?????_CONV2   }
                IF GT F1 = F2;

                F2 = -F2;
                COMP(F1,F2);
                IF LT F1 = F2;

{ store u(t) value }

                DM(I0,-1) = F1;                 { ?????_U0      }

#ifndef NO_NOTCH_FILTER
                F2 = DM(I2,10);                 { NFIL?X2        }
```

```
                F4 = PASS F2, F3 = DM(I2,5);     { NFIL?DK2       }
                F2 = F2*F3, F3 = DM(I2,-10);     { NFIL?DK1       }
                F1 = F1+F2, F2 = DM(I2,15);      { NFIL?X1        }
                F5 = F2*F3, F3 = DM(I2,-20);     { NFIL?NK1       }
                F1 = F1+F5, DM(I2,5) = F2;       { NFIL?X2        }
                F2 = F2*F3, DM(I2,20) = F1;      { NFIL?X1        }
                F1 = F2+F1;
                F1 = F1+F4;
#endif

{ multiply by conversion factor to obtain proper units }

                F1 = F0*F1, F2 = DM(I2,5);       { DA?CRTOFF      }

{ add offset value }

                F1 = F1+F2, R2 = DM(I2,5);       { DA?OFF         }
                F2 = FLOAT R2, F0 = DM(I1,-3);   { ?????_U0       }
                F1 = F1+F2, DM(I0,-3) = F0;      { ?????_U1       }

{ convert floating-point number to integer }

                F2 = DABITSMAX;
                F1 = F1+F2, F3 = DM(I0,-1);      { ?????_V1       }
                R0 = FIX F1, DM(I1,-1) = F3;     { ?????_V2       }

{ update control signal storage location   }

                DM(I2,0) = R0;                   { DA?CONTROL     }

{-------------------------------------------------------------------------}
{ Set angular reset state to low.  When in this state, we can safely      }
{ set the number of A/Ds and D/As.  We can also safely set the go-mode    }
{ and/or the interrupt mode.                                              }
{-------------------------------------------------------------------------}

                R0 = 0x0;
                DM(CONTROL) = R0;

{ set A/Ds to zero to speed up conversion        }
                R0 = BEARTOT;
                R1 = LSHIFT R0 BY 3;
                DM(CHANNELS) = R1;

{-------------------------------------------------------------------------}
{ We are done so we can turn off angular reset                            }
{-------------------------------------------------------------------------}

                R0 = 0x80;
                DM(CONTROL) = R0;

{ control values have been calculated, so lets send them out       }

                B2 = DA0CONTROL;
                LCNTR = BEARTOT,DO rsend2 UNTIL LCE;
                  R0 = DM(I2,1);
rsend2:           DM(DAFIFO) = R0;


{ Toggle FLAG2 to signal I/O board to start conversion                    }

                BIT SET ASTAT 0x00200000;
                F3 = DM(I0,-2);                  { ?????_V0 }
                DM(I1,-2) = F3;                  { ?????_V1 }
                BIT CLR ASTAT 0x00200000;

{ update control variables }

                F0 = DM(I0,-1);                  { ?????_X1 }
```

```
                DM(I1,-1) = F0;                 { ?????_X2 }

                F0 = DM(I0,0);                  { ?????_X0 }
                DM(I1,0) = F0;                  { ?????_X1 }

{ Save new output values                                                    }
#ifdef RAW
                R0 = DM(DELAY_COUNT);
                R1 = R0-1;
                DM(DELAY_COUNT) = R1;
                R1 = PASS R1;
                IF GT JUMP sskip4;
                DM(DELAY_COUNT) = R0;
                R0 = I7;
                R1 = DM(SAVE_END);
                R0 = R1-R0;
                IF EQ JUMP sskip4;
                R0 = M3;
                R1 = SAVEBEAR;
                R0 = R0-R1;
                IF NE JUMP sskip4;
                B0 = AD0INPUT;
                R0 = DM(M3,I0);
                DM(I6,1) = R0;
                B0 = DA0CONTROL;
                R0 = DM(M3,I0);
                DM(I7,1) = R0;
#endif

#ifdef PLOT
                R0 = DM(DELAY_COUNT);
                R1 = R0-1;
                DM(DELAY_COUNT) = R1;
                R1 = PASS R1;
                IF GT JUMP sskip4;
                DM(DELAY_COUNT) = R0;
                R0 = I7;
                R1 = DM(SAVE_END);
                R0 = R1-R0;
                IF EQ JUMP sskip4;
                R0 = M3;
                R1 = SAVEBEAR;
                R0 = R0-R1;
                IF NE JUMP sskip4;
                B0 = AD0INPUT;
                R0 = DM(M3,I0);
                R0 = FEXT R0 BY 0:14 (SE);
                DM(I6,1) = R0;
                B0 = DA0CONTROL;
                R0 = DM(M3,I0);
                R1 = 0XF000;
                R0 = R0-R1;
                DM(I7,1) = R0;
#endif

sskip4:         R0 = DM(BEAR_NUM);
                R0 = R0+1;
                DM(BEAR_NUM) = R0;
                R1 = BEARTOT;
                COMP(R0,R1);
                IF NE RTI;
                R0 = 0;
                DM(BEAR_NUM) = R0;

{ return }

                RTI;
```

```
sample:
{-------------------------------------------------------------------------}
{ Set angular reset state to low.  When in this state, we can safely      }
{ set the number of A/Ds and D/As.  We can also safely set the go-mode     }
{ and/or the interrupt mode.                                              }
{-------------------------------------------------------------------------}

                R0 = 0x0;
                DM(CONTROL) = R0;

{ set D/As to zero and A/Ds to number of bearings to control     }
                R0 = BEARTOT;
                DM(CHANNELS) = R0;

{-------------------------------------------------------------------------}
{ We are done so we can turn off angular reset                            }
{-------------------------------------------------------------------------}

                R0 = 0x80;
                DM(CONTROL) = R0;

{ Toggle FLAG2 to signal I/O board to start conversion                    }

                BIT SET ASTAT 0x00200000;
                R0 = DM(BEAR_NUM);
                M3 = R0;
                BIT CLR ASTAT 0x00200000;

                R0 = DM(BEAR_NUM);
                R1 = PASS R0;
                IF EQ JUMP rad_calc (DB);
                B0 = AXIAL_X0;
                B1 = AXIAL_CONV1;

rad1:
                R1 = R1 - 1;
                IF EQ JUMP rad_calc (DB);
                B0 = RAD1X_X0;
                B1 = RAD1X_CONV1;

rad2:
                R1 = R1 - 1;
                IF EQ JUMP rad_calc (DB);
                B0 = RAD1Y_X0;
                B1 = RAD1Y_CONV1;

rad3:
                R1 = R1 - 1;
                IF EQ JUMP rad_calc (DB);
                B0 = RAD2X_X0;
                B1 = RAD2X_CONV1;

rad4:
                JUMP rad_calc (DB);
                B0 = RAD2Y_X0;
                B1 = RAD2Y_CONV1;

.ENDSEG;

.SEGMENT /PM tmzh_svc;

                JUMP sample;

.ENDSEG;

.SEGMENT /PM fltu_svc;

                R15 = DM(UNDER_COUNT);
                R15 = R15+1;
```

```
                DM(UNDER_COUNT) = R15;
                BIT CLR STKY 0x01;
                RTI;

.ENDSEG;
```

# Appendix J

# DSP Programming Environment

---

This appendix describes the process of assembling, linking, and downloading controller assembly files to the DSP board. The second section describes the controller testing procedure that must be performed before the new controller is allowed to control the actual hardware. The final section contains listings of the auxiliary programs necessary to download code and upload data to the DSP board. All of the programs in the section are Disk Operating System (DOS) command line programs which are run on the Personal Computer (PC) that holds the DSP board. The actual writing, assembling, and linking may be performed on any PC and the finished program transferred via floppy to the PC that holds the DSP board.

## J.1 Digital Controller Assembling and Operation

It is recommended that all controllers be written in assembly language for two reasons. First, the code will be faster, the instruction set is small and therefore easy to learn, and the notation is simple and therefore very easy to master. Second, since we are using a custom designed DSP board, assembly language will have to be used at some point to address the I/O Controller board. If you choose to write your controller in C, it is recommended that you initially write a C callable library in assembly language which accesses the I/O Controller board.

The controller assembly file can be written with any editor that can save the file as ASCII text. The extension of the file should be ASM. To assemble the file, use the following command,

*asm21k controlr.asm*

This assembles the controller file *controlr.asm* and produces the *controlr.obj* object code file. Some controller files are more complicated then others and may use *ifdef* statements. If this is the case, this file should be assembled with the required defines set. For example,

*asm21k -Ddefine controlr.asm*

After the file has been successfully assembled, the object code must be linked into a binary form. To link the file, use the following command,

*ld21k -a pump -o controlr controlr.obj*

The linker converts the object code having a default extension of OBJ to a binary image file having the default extension of EXE. This file is in a binary form which is incompatible with DOS binary files even though it has the same extension. Running an EXE file created by *ld21k* from the DOS command line will cause the computer to hang. The *a* flag denotes the architecture file which is to be used during the linking process. The architecture file is assumed to have a default extension of ACH. The *o* flag specifies the name of the resulting binary image file. The final stage is to convert the binary image file to a form that allows for easy downloading to the DSP board. This is accomplished by invoking the PROM splitter,

*spl21k -pm -ram -a pump -f B -o controlr controlr.exe*

The splitter converts the binary image file having a default extension of EXE to an ASCII file containing the hex representation of the binary file having the default extension of STK. The *pm* flag denotes that only Program Memory (PM) segments should be converted. If the original assembly file contained initialized Data Memory (DM) variables, the splitter program should be rerun using the *dm* flag. The *ram* flag denotes that only memory addresses denoted as RAM addresses in the architecture file should be converted. The *a* flag denotes the architecture file which is to be used by the splitter. The architecture file is assumed to have a default extension of ACH. The *o* flag specifies the name of the resulting STK file. The *f* flag denotes the format of the output file. For this application, the byte-wise stacked format (B) is specified.

All of the above steps can be performed on any PC running DOS. The next phase is downloading the STK file, starting the DSP chip, and stopping the DSP chip. This phase begins with the following command,

*reset*

This is a BAT file which holds the following commands,

*stopdsp*
*dumpdsp reset.stk*
*startdsp*
*stopdsp*

The *stopdsp* program stops any program currently executing on the DSP board. The *dumpdsp* program dumps the byte-wise stacked ASCII file to the appropriate memory locations on the DSP board. The *reset.stk* byte-wise stacked file is a simple program that zeros the values of the D/As. When a program is stopped using *stopdsp* the last values sent to the D/As will remain active until replaced. This program replaces these values with zeros. The *startdsp* program triggers a reset interrupt on the DSP chip which effectively starts the downloaded program.

The digital controller files are downloaded to the DSP board next using the following command,

*dumpdsp controlr.stk*

The *dumpdsp* program must be run twice if both PM and DM byte-wise stacked format files were created. To start the downloaded program, use the following command

*startdsp*

This program triggers a reset interrupt on the DSP chip which starts the program running. Verify that all of the switches on the analog circuit boards of all the axes being controlled by the digital controller program are in the down position before starting the controller ("D" is for "Down" and for "Digital"). For best results, power up the analog controller before starting the digital controller. To stop the digital controller, power down the analog controller and use either the *stopdsp* or *reset* commands.

# J.2 Digital Controller Testing

It is extremely important that the turbopump be spared the punishment inflicted by an unstable controller. Therefore, the researcher should not use a new digital controller until it has been thoroughly tested. Before testing the new controller with the actual hardware, two initial steps should be performed. This section describes each step of the process.

The first step is to run the controller through the simulator and exhaustively test that the calculations performed in the control loop and initialization code (if any) are correct. The simulator is a program which mimics the DSP chip in software. The simulator can be invoked with the following command,

*sim21k -a pump -d ports.dat -e controlr.exe*

Before running the simulator, verify that the mouse driver is loaded. The *a* flag denotes the architecture file to be used by the simulator. The *e* flag denotes the binary image file to load into the simulator. Finally, the *d* flag denotes the file which describes the files which the simulator should access which reading or writing to ports mapped in the DSP chip address space. When your program reads from a port, the simulator reads the data from the file designated in the port description file. Writing data to a port also causes the simulator to write that data to the file designated in the port description file. The following is an example of a typical port description file,

```
port pm 800000   hex n ioadin.dat null
port pm 800001   fix y iostat.dat null
port dm 20000000 fix n null       status.dat
port dm 20000001 fix n null       timer.dat
port dm 20000002 hex n null       null
port dm 40000000 fix n null       iodaout.dat
port dm 40000001 fix n null       iocntrl.dat
port dm 40000002 fix n null       iochans.dat
```

The port description file is composed of seven columns. Column one designates that the memory address is a port. Column two designates which memory segment the port is mapped into. Column three designates the physical address of the memory mapped port. This address must agree with the address specified in the architecture file. Column four designates the type of data contained in the file. Possible values include hexadecimal, fixed point, and floating point. Column five denotes whether the file should be accessed as a circular buffer. If *y* is denoted, the file is treated as a circular buffer and the simulator will read from the beginning of file after it reaches the end of the file. Column six designates the name of the file that the simulator will read from when a read operation is performed by the assembly code. A value of *null* designates that the port is write only. Finally, column seven designates the name of the file that the simulator will write to when a write operation is performed by the assembly code. A value of *null* designates that the port is read only.

It is very important that the researcher perform the tedious process of verifying that the controller is performing the desired calculations correctly. Errors creep into code in very unexpected ways and only careful attention to detail will ferret them out. It is recommended that the data used as the input from the A/Ds be a properly sampled sine wave. This will give the most realistic results because the actual position signal is sinusoidal in nature.

The next stage of the testing process involves testing the program on the DSP board. One limitation of the simulator is that it emulates the interaction between the program and the memory mapped ports using files. Because this interface is emulated, it is always successful. That may not be the case using the actual hardware. In an effort to test the hardware access code, a function generator should be attached to one input of an oscilloscope and then to the A/D of one axis of the I/O Interface Board. The D/A of the I/O Interface Board should be attached to the remaining input of the oscilloscope. The controller program can now be tested on the actual hardware without effecting the turbopump. The output of the function generator should be a sine wave of low amplitude and the output of the D/A should be a sine-like wave having the appropriate phase lag. The controller should also respond appropriately to changes in amplitude, offset, and frequency. Only after passing these two tests should the controller actually be tested on the turbopump hardware.

# J.3 Auxiliary Program Listings

## J.3.1 dumpdsp.c

This DOS C program reads a byte-wise stacked format file and downloads the necessary data contained in that file to the memory locations specified within that file. The program can be run multiple times to download the separate PM and DM files that must be generated by the PROM splitter program. The separate byte-wise stacked format files can be concatenated and downloaded as a single entity provided that the file contains only one end of data header. This program has the following syntax,

*dumpdsp filename.ext [/a:xxxx]*

The *a* flag denotes the base port address of the DSP board in hexadecimal format. The program defaults to a default base port address of 0300h which corresponds to the default base port address of the DSP board.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>

#define BUFSIZE 1024
#define STATUS_OK 0
#define STATUS_NOK 1
#define MEMERR(str) \
  fprintf(stderr, "Out of Memory Error - %s (%d)\n", str, __LINE__)
#define FUNCERR(str1, str2) \
  fprintf(stderr, "Error encountered in %s - %s (%d)\n", str1, str2, __LINE__)
#define DEAD_HEADER "000000000000000000000000\n"

typedef struct Hex40_struct
{
  unsigned short  dmdh;
  unsigned short  dmdm;
  unsigned char dmdl;
  unsigned char pad;
} Hex40Struct, *Hex40;

typedef struct Hex48_struct
{
  unsigned short pmdh;
  unsigned short pmdm;
  unsigned short pmdl;
} Hex48Struct, *Hex48;

int asc2hexi(char *hex);
long asc2hexl(char *hex);
int getheader(char *buffer, int *type, short *addr, int *hline);
void asc2hex40(char *buffer, Hex40Struct *hex40);
void asc2hex48(char *buffer, Hex48Struct *hex48);
void upload_DSP(short addr, int type, short base, unsigned short *data);

int main(int argc, char **argv)
{
  int i, j;
  int line, hline, type;
  short addr, base;
  char *filename, *hex, *buffer;
  FILE *fp;
  Hex40Struct hex40;
  Hex48Struct hex48;

  filename = NULL;
  base = line = 0;
  if(argc > 1)
    {
      for(i=1; i<argc; i++)
        {
          if(!(strncmpi(argv[i], "/a:", 2)))
            {
              hex = &argv[i][3];
              base = asc2hexi(hex);
            }
          else
            filename = argv[i];
```

```
      }
  }

if((argc < 2) || (!filename))
  {
    fprintf(stderr, "DumpDSP infile.ext [/a:0000]\n\n");
    fprintf(stderr, "  where:  infile.ext = byte-stacked formatted file.\n");
    fprintf(stderr, "                  /a = DSP board address.\n");
    fprintf(stderr, "                       (default = 0300h)\n");
    exit(1);
  }

if(!base)
  base = 0x300;

if((fp = fopen(filename, "r")) == NULL)
  {
    fprintf(stderr, "Unable to open byte-stacked formatted file %s\n\n",
            filename);
    exit(1);
  }

if((buffer = (char *) calloc(BUFSIZE, sizeof(char))) == NULL)
  {
    MEMERR("DumpDSP");
    fclose(fp);
    exit(1);
  }

hline = 1;
while(fgets(buffer, BUFSIZE, fp))
  {
    if(strlen(buffer) == BUFSIZE-1)
      {
        fprintf(stderr, "Input buffer overflow - exitting ...\n\n");
        exit(1);
      }
    line++;
    if(line == hline)
      {
        if(!strcmp(buffer, DEAD_HEADER))
          {
            fprintf(stdout, "Finished processing file (line %d)...\n",
                    line);
            fclose(fp);
            free(buffer);
            exit(0);
          }
        if(getheader(buffer, &type, &addr, &hline) != STATUS_OK)
          {
            FUNCERR("getheader", "DumpDSP");
            fclose(fp);
            free(buffer);
            exit(1);
          }
        else if(type == 0x80)
          fprintf(stdout, "Processing PM data (line %d)...\n", line);
        else if(!type)
          fprintf(stdout, "Processing DM data (line %d)...\n", line);
        else
          {
            fprintf(stdout,
                    "Unknown type encountered (type %x, line %d)...\n",
                    type, line);
            fclose(fp);
            free(buffer);
            exit(1);
          }
        continue;
```

```
        }
      switch(type)
        {
        case 0:  /* write to DM address space */
          asc2hex40(buffer, &hex40);
          upload_DSP(addr, type, base, (unsigned short *) &hex40);
          break;
        case 0x80:  /* write to PM address space */
          asc2hex48(buffer, &hex48);
          upload_DSP(addr, type, base, (unsigned short *) &hex48);
          break;
        }
      addr++;
    }
  fprintf(stderr, "WARNING - Termination header not found\n");

  fclose(fp);
  free(buffer);

  exit(0);

}

int asc2hexi(char *hex)
{
  int i, shift, ret;

  ret = 0;
  shift = strlen(hex)-1;
  for(i=shift; i>=0; i--)
    {
      hex[i] -= '0';
      if(hex[i] > 9)  /* assume upper case (A = 0x41) */
        hex[i] -= 0x07;
      if(hex[i] > 15)  /* could be lower case (a = 0x61) */
        hex[i] -= 0x20;
      if(hex[i] < 0 || hex[i] > 15)
        {
          fprintf(stderr, "Bad hex value (%s) - asc2hex\n", hex);
          return(0);
        }
      ret += (hex[i] << ((shift - i) << 2));
    }
  return(ret);
}

long asc2hexl(char *hex)
{
  int i, shift;
  long ret;

  ret = 0;
  shift = strlen(hex)-1;
  for(i=shift; i>=0; i--)
    {
      hex[i] -= '0';
      if(hex[i] > 9)  /* assume upper case (A = 0x41) */
        hex[i] -= 0x07;
      if(hex[i] > 15)  /* could be lower case (a = 0x61) */
        hex[i] -= 0x20;
      if(hex[i] < 0 || hex[i] > 15)
        {
          fprintf(stderr, "Bad hex value (%s) - asc2hex\n", hex);
          return(0);
        }
      ret += (hex[i] << ((shift - i) << 2));
    }
  return(ret);
}
```

```
int getheader(char *buffer, int *type, short *addr, int *hline)
{
  int width;
  union {
   unsigned short nib[2];
   unsigned long  tot;
  } temp;

  buffer[2] = '\0';
  width = asc2hexi(buffer);
  if(width > (sizeof(long) << 3))
    {
      fprintf(stderr, "Address width too large (%d) - getheader\n", width);
      return(STATUS_NOK);
    }
  buffer[6] = '\0';
  (*type) = asc2hexi(&buffer[4]);
  buffer[24] = '\0';
  temp.tot = asc2hexl(&buffer[16]);
  if((*type) == 0x80)
    (*hline) += ((int) (temp.tot/6));
  else if(!(*type))
    (*hline) += ((int) (temp.tot/5));
  else
    {
      fprintf(stderr, "Unknown type (%d) - getheader\n", (*type));
      return(STATUS_NOK);
    }
  (*hline)++;
  buffer[16] = '\0';
  temp.tot = asc2hexl(&buffer[8]);
  if(temp.nib[1] != 0)  /* remember - little endian */
    {
      fprintf(stderr, "Bogus address (%ld) - getheader\n", temp.tot);
      return(STATUS_NOK);
    }
  (*addr) = temp.nib[0];

  return(STATUS_OK);

}


void asc2hex40(char *buffer, Hex40Struct *hex40)
{

  buffer[10] = '\0';
  hex40->pad = (unsigned char) '\0';
  hex40->dmdl = (unsigned char) asc2hexi(&buffer[8]);
  buffer[8] = '\0';
  hex40->dmdm = asc2hexi(&buffer[4]);
  buffer[4] = '\0';
  hex40->dmdh = asc2hexi(buffer);

}


void asc2hex48(char *buffer, Hex48Struct *hex48)
{

  buffer[12] = '\0';
  hex48->pmdl = asc2hexi(&buffer[8]);
  buffer[8] = '\0';
  hex48->pmdm = asc2hexi(&buffer[4]);
  buffer[4] = '\0';
  hex48->pmdh = asc2hexi(buffer);

}
```

```
void upload_DSP(short addr, int type, short base, unsigned short *data)
{

  if(type)
    {
       outpw(base+0x10, addr);
       outpw(base+0x18, data[2]);
       outpw(base+0x16, data[1]);
       outpw(base+0x14, data[0]);
       outpw(base+0x2, 0x01);
    }
  else
    {
       outpw(base+0x12, addr);
       outpw(base+0x1e, data[2]);
       outpw(base+0x1c, data[1]);
       outpw(base+0x1a, data[0]);
       outpw(base, 0x01);
    }
}
```

## J.3.2 reset.asm

This DSP assembly program places appropriate values into the D/A registers so that all the D/As on the I/O Interface Board output zero volts. After triggering a D/A conversion, the program waits for the conversion to finish, and enters an idle state until the DSP chip is stopped. The values sent to the D/As are the offsets determined by other means. Whenever the resistor packs on any of the D/As is changed, the offsets must be corrected and this program must be updated to the new values. If you are using this program as part of the *reset.bat* file, remember that reassembling the assembly code results in a *reset.exe* file being created. If this file is not deleted, when the user types the *reset* command on the command line, the *reset.exe* will be executed instead of *reset.bat* and the machine will hang.

```
{ reset.asm                                                    }

{ The following ports are found on the DSP board               }

.SEGMENT /DM status;
.VAR DSPSTAT;
.ENDSEG;

.SEGMENT /DM timer;
.VAR DSPTIMER;
.ENDSEG;

.SEGMENT /DM digio;
.VAR DIGIO;
.ENDSEG;

{ The following ports pertain in the A/D and conversion process }

.SEGMENT /PM ioadin;
.VAR ADFIFO;
.ENDSEG;

.SEGMENT /PM iostat;
.VAR IOSTAT;
.ENDSEG;
```

```
.SEGMENT /DM iodaout;
.VAR DAFIFO;
.ENDSEG;

.SEGMENT /DM iocntrl;
.VAR CONTROL;
.ENDSEG;

.SEGMENT /DM iochans;
.VAR CHANNELS;
.ENDSEG;

{ the following non-existent ports are for debugging purposes   }

{.PORT AD_CON_REG2; }
{.PORT AD_DATA_REG2;}
{.PORT DA_CON_REG2; }
{.PORT DA_DATA_REG2;}

{ division macro - if there is a quicker way, just change macro }
{                  macro requires the following:                }
{                  F0  = numerator                              }
{                  F12 = denominator                            }

{--------------------------------------------------------------------}
{ Division Algorithm - Given Q = D/N, multiply N and D by the same set  }
{                      of factors, Rn.                               }
{      N x R0 x R1 x ... x Rn                                        }
{  Q = ----------------------                                        }
{      D x R0 x R1 x ... x Rn                                        }
{                                                                    }
{ Choose Rn such that as the number of factors increases, the        }
{ denominator approaches 1.  The quotient is then approximately equal   }
{ to the numerator.                                                  }
{                                                                    }
{ R0 is the seed provided by the RECIPS instruction.  Succssive Rn are  }
{ calculated by the following formula:                               }
{  Ri = 2-D(i-1)                                                     }
{                                                                    }
{ NOTE:  The macro uses the following registers:                     }
{        F0, F7, F11, F12                                            }
{        These registers will be over-written upon exit from macro.  }
{--------------------------------------------------------------------}

#define div     F11 = 2.0; \
                F0 = RECIPS F12, F7 = F0; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F0 = F0*F7

#define DIV     F11 = 2.0; \
                F0 = RECIPS F12, F7 = F0; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F0 = F0*F7

{ Without TJs filter            }
{#define AD0OFFSET  0x01CA}
{#define AD1OFFSET  0x0204}
{#define AD2OFFSET  0x0234}
{#define AD3OFFSET  0x0245}
```

```
{#define AD4OFFSET  0X0252}
{#define AD5OFFSET  0x0252}
{#define AD6OFFSET  0x0261}

#define AD0OFFSET  0x00001076
#define AD1OFFSET  0x0000107A
#define AD2OFFSET  0x0000107A
#define AD3OFFSET  0x0000107B
#define AD4OFFSET  0x0000107B
#define AD5OFFSET  0x000008D0
#define AD6OFFSET  0x0000090C

{ Without TJs filter           }
{#define DA0OFFSET 0x020E}
{#define DA1OFFSET 0x01B3}
{#define DA2OFFSET 0x01F2}
{#define DA3OFFSET 0x01E6}
{#define DA4OFFSET 0x0230}

#define DA0OFFSET  0x00000003
#define DA1OFFSET  0x00000000
#define DA2OFFSET  0x00000001
#define DA3OFFSET  0xFFFFFFFF
#define DA4OFFSET  0xFFFFFFFF

{ D/A constants                                                 }
#define DAVOLTSMAX   5.0
#define DAVOLTSMIN  -5.0
#define DABITSMAX   8192.0

{ A/D constants                                                 }
#define ADVOLTSMAX   5.0
#define ADVOLTSMIN  -5.0
#define ADBITSMAX   8192.0


.SEGMENT /DM dm_data;

.VAR VOLT2BITS;

.VAR DA0OFF;
.VAR DA1OFF;
.VAR DA2OFF;
.VAR DA3OFF;
.VAR DA4OFF;

.ENDSEG;

.SEGMENT /PM rst_svc;

{-------------------------------------------------------------------------}
{ At reset, the BANK registers are as follows:                            }
{       PMBANK1 = 0x800000                                                }
{       DMBANK1 = 0x20000000                                              }
{       DMBANK2 = 0x40000000                                              }
{       DMBANK3 = 0x80000000                                              }
{ These values, by coincidence, are perfect for our I/O board             }
{-------------------------------------------------------------------------}

{-------------------------------------------------------------------------}
{ The default value of PMWAIT at reset is 0x0003DE.  This corresponds     }
{ to the following:                                                       }
{       bit  13    = 0   (No automatic wait state)                        }
{       bits 12-10 = 000 (memory page size = 256 words)                   }
{       bits  9-7  = 111 (7 PMBANK1 wait states)                          }
{       bits  6-5  = 10  (Int. and Ext. wait state ack mode)              }
{       bits  4-2  = 111 (7 PMBANK0 wait states)                          }
{       bits  1-0  = 10  (Int. and Ext. wait state ack mode)              }
{                                                                         }
```

```
{ For our setup, the DSP board accesses memory at 0 wait states and     }
{ accesses all ports and 1 wait state.  Therefore:                      }
{       bit  13    = 0   (No automatic wait state)                      }
{       bits 12-10 = 100 (memory page size = 4096 words)                }
{       bits  9-7  = 001 (1 PMBANK1 wait state)                         }
{       bits  6-5  = 10  (Int. and Ext. wait state ack mode)            }
{       bits  4-2  = 000 (0 PMBANK0 wait states)                        }
{       bits  1-0  = 10  (Int. and Ext. wait state ack mode)            }
{-----------------------------------------------------------------------}

                PMWAIT = 0x0010C2;

{-----------------------------------------------------------------------}
{ The default value of DMWAIT at reset is 0x000F7BDE.  This corresponds }
{ to the following:                                                     }
{       bit  23    = 0   (No automatic wait state)                      }
{       bits 22-20 = 000 (memory page size = 256 words)                 }
{       bits 19-17 = 111 (7 DMBANK3 wait states)                        }
{       bits 16-15 = 10  (Int. and Ext. wait state ack mode)            }
{       bits 14-12 = 111 (7 DMBANK2 wait states)                        }
{       bits 11-10 = 10  (Int. and Ext. wait state ack mode)            }
{       bits  9-7  = 111 (7 DMBANK1 wait states)                        }
{       bits  6-5  = 10  (Int. and Ext. wait state ack mode)            }
{       bits  4-2  = 111 (7 DMBANK0 wait states)                        }
{       bits  1-0  = 10  (Int. and Ext. wait state ack mode)            }
{                                                                       }
{ For our setup, the DSP board accesses memory at 0 wait states and     }
{ accesses all ports and 1 wait state.  Therefore:                      }
{       bit  23    = 0   (No automatic wait state)                      }
{       bits 22-20 = 100 (memory page size = 4096 words)                }
{       bits 19-17 = 001 (1 DMBANK3 wait states)                        }
{       bits 16-15 = 10  (Int. and Ext. wait state ack mode)            }
{       bits 14-12 = 001 (1 DMBANK2 wait states)                        }
{       bits 11-10 = 10  (Int. and Ext. wait state ack mode)            }
{       bits  9-7  = 000 (0 DMBANK1 wait state)                         }
{       bits  6-5  = 10  (Int. and Ext. wait state ack mode)            }
{       bits  4-2  = 000 (0 DMBANK0 wait states)                        }
{       bits  1-0  = 10  (Int. and Ext. wait state ack mode)            }
{-----------------------------------------------------------------------}

                DMWAIT = 0x00431842;

{-----------------------------------------------------------------------}
{ Set FLAG2 to output mode so we can trigger I/O board conversion when  }
{ we need to.                                                           }
{-----------------------------------------------------------------------}

                MODE2 = 0x00020000;


                JUMP initialize;
.ENDSEG;


.SEGMENT /PM pm_code;

initialize:

{ initialize registers }

                IMASK = 0;
                MODE1 = 0x0012000;

                I0 = 0;
                I1 = 0;
                I2 = 0;
                I3 = 0;
                I4 = 0;
                I5 = 0;
```

```
                I6 = 0;
                I7 = 0;
                M0 = 0;
                M1 = 0;
                M2 = 0;
                M3 = 0;
                M4 = 0;
                M5 = 0;
                M6 = 0;
                M7 = 0;
                L0 = 0;
                L1 = 0;
                L2 = 0;
                L3 = 0;
                L4 = 0;
                L5 = 0;
                L6 = 0;
                L7 = 0;
{-------------------------------------------------------------------------}
{ This program does the following:                                        }
{       1.  Triggers A/D conversion on one channel                        }
{       2.  Polls A/D status register to see when conversion complete     }
{       3.  Pushes A/D value out to D/A                                   }
{       4.  returns to step 1                                             }
{-------------------------------------------------------------------------}

                BIT CLR ASTAT 0X00200000;

                B0 = DA0OFF;
                R0 = DA0OFFSET;
                DM(I0,1) = R0;
                R0 = DA1OFFSET;
                DM(I0,1) = R0;
                R0 = DA2OFFSET;
                DM(I0,1) = R0;
                R0 = DA3OFFSET;
                DM(I0,1) = R0;
                R0 = DA4OFFSET;
                DM(I0,1) = R0;

                F0 = DABITSMAX;
                F12 = DAVOLTSMAX;
                DIV;
                DM(VOLT2BITS) = F0;

{ Reset I/O board                                                         }
                R0 = 0x0;
                DM(CONTROL) = R0;

                R0 = 5;
                R1 = LSHIFT R0 BY 3;
                DM(CHANNELS) = R1;

                R0 = 0x80;
                DM(CONTROL) = R0;

                R4 = 0x03;
wait:           R0 = PM(IOSTAT);
                R1 = R0 AND R4;
                IF NE JUMP wait;

                DM(DSPSTAT) = R1;
                F2 = DABITSMAX;
                R2 = FIX F2;
                LCNTR = 5;
                B0 = DA0OFF;
                DO adj UNTIL LCE;
                  R1 = DM(I0,1);
                  R2 = R1+R2;
```

```
                  R0 = ASHIFT R2 BY 2;
adj:              DM(DAFIFO) = R0;

                BIT SET ASTAT 0x00200000;
                NOP; NOP;
                BIT CLR ASTAT 0x00200000;

gag:            IDLE;
                JUMP gag;

.ENDSEG;


.SEGMENT /PM tmzh_svc;


.ENDSEG;
```

## J.3.3 startdsp.c

This DOS C program triggers a reset interrupt on the DSP chip which initiates DSP program execution. This program has the following syntax,

*startdsp [/a:xxxx]*

The *a* flag denotes the base port address of the DSP board in hexadecimal format. The program defaults to a default base port address of 0300h which corresponds to the default base port address of the DSP board. This program also has two undocumented flags. The *h* flag prints syntax information. The *s* flag instructs the program to constantly poll the status register of the DSP board and print the contents to the screen. Using this flag will require using Ctrl-Break to stop execution. The use of this flag allows a program running on the DSP chip to communicate with the DOS command line.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <dos.h>

#define STATUS_OK 0
#define STATUS_NOK 1
#define MEMERR(str) \
  fprintf(stderr, "Out of Memory Error - %s (%d)\n", str, __LINE__)
#define FUNCERR(str1, str2) \
  fprintf(stderr, "Error encountered in %s - %s (%d)\n", str1, str2, __LINE__)

int asc2hexi(char *hex);

void main(int argc, char **argv)
{
  int i;
  union {
        unsigned short word;
        unsigned char  nib[2];
  } stat;
  short base;
  char *hex;
```

```
  base = stat.word = 0;
  if(argc > 1)
    {
      for(i=1; i<argc; i++)
        {
          if(!(strncmpi(argv[i], "/a:", 3)))
            {
              hex = &argv[i][3];
              base = asc2hexi(hex);
            }
          else if(!(strcmpi(argv[i], "/s")))
            {
               stat.word = 1;
            }
          else if(!(strncmpi(argv[i], "/h", 2)))
            {
               fprintf(stderr, "StartDSP [/a:0000]\n\n");
               fprintf(stderr, "  where:  /a = DSP board address.\n");
               fprintf(stderr, "                (default = 0300h)\n");
               exit(0);
            }
        }
    }

  if(!base)
    base = 0x300;

  outpw(base+0x04, 0x01);

  if(stat.word)
    {
          while(1)
                {
                  sleep(1);
                  stat.word = 0x0;
                  stat.word = inpw(base+0x10);
                  fprintf(stderr, "Status = %02X%02X\n", stat.nib[1], stat.nib[0]);
                }
    }

  exit(0);

}


int asc2hexi(char *hex)
{
  int i, shift, ret;

  ret = 0;
  shift = strlen(hex)-1;
  for(i=shift; i>=0; i--)
    {
      hex[i] -= '0';
      if(hex[i] > 9)  /* assume upper case (A = 0x41) */
        hex[i] -= 0x07;
      if(hex[i] > 15)  /* could be lower case (a = 0x61) */
        hex[i] -= 0x20;
      if(hex[i] < 0 || hex[i] > 15)
        {
          fprintf(stderr, "Bad hex value (%s) - asc2hex\n", hex);
          return(0);
        }
      ret += (hex[i] << ((shift - i) << 2));
    }
  return(ret);
}
```

## J.3.4 stopdsp.c

This DOS C program triggers a reset interrupt on the DSP chip which stop its execution. This program has the following syntax,

*stopdsp [/a:xxxx]*

The *a* flag denotes the base port address of the DSP board in hexadecimal format. The program defaults to a default base port address of 0300h which corresponds to the default base port address of the DSP board. This program also has the undocumented *h* flag which prints syntax information.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>

#define STATUS_OK 0
#define STATUS_NOK 1
#define MEMERR(str) \
  fprintf(stderr, "Out of Memory Error - %s (%d)\n", str, __LINE__)
#define FUNCERR(str1, str2) \
  fprintf(stderr, "Error encountered in %s - %s (%d)\n", str1, str2, __LINE__)

int asc2hexi(char *hex);

int main(int argc, char **argv)
{
  int i;
  short base;
  char *hex;

  base = 0;
  if(argc > 1)
    {
      for(i=1; i<argc; i++)
        {
          if(!(strncmpi(argv[i], "/a:", 3)))
            {
              hex = &argv[i][3];
              base = asc2hexi(hex);
            }
          else if(!(strncmpi(argv[i], "/h", 2)))
            {
               fprintf(stderr, "StopDSP [/a:0000]\n\n");
               fprintf(stderr, "  where:  /a = DSP board address.\n");
               fprintf(stderr, "                (default = 0300h)\n");
               exit(0);
            }
        }
    }

  if(!base)
    base = 0x300;

  outpw(base+0x04, 0x00);

  exit(0);

}
```

```
int asc2hexi(char *hex)
{
  int i, shift, ret;

  ret = 0;
  shift = strlen(hex)-1;
  for(i=shift; i>=0; i--)
    {
      hex[i] -= '0';
      if(hex[i] > 9)  /* assume upper case (A = 0x41) */
        hex[i] -= 0x07;
      if(hex[i] > 15)  /* could be lower case (a = 0x61) */
        hex[i] -= 0x20;
      if(hex[i] < 0 || hex[i] > 15)
        {
          fprintf(stderr, "Bad hex value (%s) - asc2hex\n", hex);
          return(0);
        }
      ret += (hex[i] << ((shift - i) << 2));
    }
  return(ret);
}
```

## J.3.5 readdsp.c

This DOS C program reads data from the memory on the DSP Board and outputs this data to a file. This program has the following syntax,

*readdsp filename.ext [/a:xxxx] [/(pm|dm)] [/r:xxxx-xxxx] [/f] [/u] [/i]*

The *a* flag denotes the base port address of the DSP board in hexadecimal format. The program defaults to a default base port address of 0300h which corresponds to the default base port address of the DSP board. The *pm* flag denotes that program memory should be read. The *dm* flag denotes that data memory should be read. If neither the *pm* or the *dm* flag are present, the data memory segment is assumed. The *r* flag denotes the range of memory addresses that should be read in hexadecimal format. If the *r* is not present, the default address range is 0000-7FFEh (Note: reading the 7FFFh data value causes the program to hang). The default format of the data output to the desired file is hexadecimal format. The data format can be changed by using the *f*, *u*, and *i* flags denote floating point, unsigned integer, and signed integer data respectively.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>

#define HEX      0
#define INT      1
#define UNSIGNED 2
#define FLOAT    3

#define BUFSIZE 1024
#define STATUS_OK 0
#define STATUS_NOK 1
#define MEMERR(str) \
  fprintf(stderr, "Out of Memory Error - %s (%d)\n", str, __LINE__)
#define FUNCERR(str1, str2) \
```

```
  fprintf(stderr, "Error encountered in %s - %s (%d)\n", str1, str2, __LINE__)

typedef struct Hex40_struct
{
  unsigned short  dmdh;
  unsigned short  dmdm;
  unsigned char dmdl;
  unsigned char pad;
} Hex40Struct, *Hex40;

typedef struct Hex48_struct
{
  unsigned short pmdh;
  unsigned short pmdm;
  unsigned short pmdl;
} Hex48Struct, *Hex48;

int asc2hexi(char *hex);
void download_DSP(short addr, int type, short base, unsigned short *data);
void conv_endian(unsigned char *data, int type, unsigned char *ans);

void main(int argc, char **argv)
{
  int i, j;
  int type, output;
  short brange, erange;
  short addr, base;
  char *filename, *hex;
  unsigned char conv[6];
  FILE *fp;
  Hex40Struct hex40;
  Hex48Struct hex48;

  filename = NULL;
  base = 0x300;
  brange = type = 0;
  erange = 0x7ffe;
  output = HEX;
  if(argc > 1)
        {
          for(i=1; i<argc; i++)
        {
          if(!(strncmpi(argv[i], "/a:", 3)))
                {
                  hex = &argv[i][3];
                  base = asc2hexi(hex);
                }
          else if(!(strncmpi(argv[i], "/pm", 3)))
                {
                  type = 0x80;
                }
          else if(!(strncmpi(argv[i], "/dm", 3)))
                {
                  type = 0;
                }
          else if(!(strncmpi(argv[i], "/i", 2)))
                {
              output = INT;
            }
          else if(!(strncmpi(argv[i], "/f", 2)))
            {
              output = FLOAT;
            }
          else if(!(strncmpi(argv[i], "/u", 2)))
            {
              output = UNSIGNED;
            }
          else if(!(strncmpi(argv[i], "/r:", 3)))
            {
```

```
            hex = strchr(&argv[i][3], '-');
            erange = (short) asc2hexi(&hex[1]);
            *hex = '\0';
            brange = (short) asc2hexi(&argv[i][3]);
          }
        else
          filename = argv[i];
      }
  }

if((argc < 2) || (!filename))
  {
    fprintf(stderr, "ReadDSP outfile.ext [/a:0000] [/(pm|dm)] ");
    fprintf(stderr, "[/r:0000-0000] [/f] [/i] [/u]\n\n");
    fprintf(stderr, "  where:  infile.ext  = byte-stacked formatted file.\n");
    fprintf(stderr, "                   /a  = DSP board address.\n");
        fprintf(stderr, "                         (default = 0300h)\n");
        fprintf(stderr, "                   /pm = download PM data.\n");
        fprintf(stderr, "                         (default = DM)\n");
        fprintf(stderr, "                   /dm = download DM data (default)\n");
        fprintf(stderr, "                   /r  = data range\n");
        fprintf(stderr, "                         (default PM = 0000-7FFEh)\n");
        fprintf(stderr, "                         (default DM = 0000-7FFEh)\n");
        fprintf(stderr, "                   /i  = integer output format\n");
        fprintf(stderr, "                   /f  = floating point output format\n");
        fprintf(stderr, "                   /u  = unsigned integer output format\n");
        exit(1);
      }

if(erange > 0x7ffe)
      erange = 0x7ffe;
if(brange > erange)
      brange = 0;

if((fp = fopen(filename, "w")) == NULL)
  {
    fprintf(stderr, "Unable to open output file %s\n\n",
            filename);
    exit(1);
  }

for(addr=brange; addr<=erange; addr++)
  {
    switch(type)
      {
      case 0:  /* write to DM address space */
        download_DSP(addr, type, base, (unsigned short *) &hex40);
        switch(output)
          {
           default:
           case HEX:
            fprintf(fp, "%04X%04X%02X\n", hex40.dmdh, hex40.dmdm,
                    hex40.dmdl);
            break;
           case INT:
            conv_endian((unsigned char *) &hex40, type, conv);
            fprintf(fp, "%ld\n", *((long *) &conv[2]));
            break;
           case UNSIGNED:
            conv_endian((unsigned char *) &hex40, type, conv);
            fprintf(fp, "%lu\n", *((unsigned long *) &conv[2]));
            break;
           case FLOAT:
            conv_endian((unsigned char *) &hex40, type, conv);
            fprintf(fp, "%f\n", *((float *) &conv[2]));
            break;
          }
        break;
      case 0x80:  /* write to PM address space */
```

```
          download_DSP(addr, type, base, (unsigned short *) &hex48);
          switch(output)
            {
             default:
             case HEX:
              fprintf(fp, "%04X%04X%04X\n", hex48.pmdh, hex48.pmdm,
                      hex48.pmdl);
              break;
             case INT:
              conv_endian((unsigned char *) &hex48, type, conv);
              fprintf(fp, "%ld\n", *((long *) &conv[2]));
              break;
             case UNSIGNED:
              conv_endian((unsigned char *) &hex48, type, conv);
              fprintf(fp, "%lu\n", *((unsigned long *) &conv[2]));
              break;
             case FLOAT:
                  conv_endian((unsigned char *) &hex48, type, &conv[2]);
              fprintf(fp, "%f\n", *((float *) conv));
              break;
            }
          break;
        }
    }

  fclose(fp);
  exit(0);

}


int asc2hexi(char *hex)
{
  int i, shift, ret;

  ret = 0;
  shift = strlen(hex)-1;
  for(i=shift; i>=0; i--)
    {
      hex[i] -= '0';
      if(hex[i] > 9)  /* assume upper case (A = 0x41) */
        hex[i] -= 0x07;
      if(hex[i] > 15)  /* could be lower case (a = 0x61) */
        hex[i] -= 0x20;
      if(hex[i] < 0 || hex[i] > 15)
        {
          fprintf(stderr, "Bad hex value (%s) - asc2hex\n", hex);
          return(0);
        }
      ret += (hex[i] << ((shift - i) << 2));
    }
  return(ret);
}


void download_DSP(short addr, int type, short base, unsigned short *data)
{

  if(type)
    {
       outpw(base+0x10, addr);
       data[0] = inpw(base+0x2);
       data[2] = inpw(base+0x18);
       data[1] = inpw(base+0x16);
       data[0] = inpw(base+0x14);
    }
  else
    {
       outpw(base+0x12, addr);
```

```
        data[0] = inpw(base);
        data[2] = inpw(base+0x1e);
        data[1] = inpw(base+0x1c);
        data[0] = inpw(base+0x1a);
    }
}

void conv_endian(unsigned char *data, int type, unsigned char *ans)
{

  if(type)
    {
      ans[0] = data[4];
      ans[1] = data[5];
    }
  else
    {
      ans[0] = data[5];
      ans[1] = data[4];
    }
  ans[2] = data[2];
  ans[3] = data[3];
  ans[4] = data[0];
  ans[5] = data[1];

}
```

## J.3.6 sineint.asm

This DSP assembly program is a timer interrupt driven program which takes the value received from the each A/D and outputs it to its corresponding D/A. This program is used to verify that all the DSP related hardware is in working order. Normally a function generator producing a sine wave is attached to the A/D and an oscilloscope is attached to the D/A to check system operation.

```
{ SineInt.asm                                                    }

{ The following ports are found on the ADSP board                }

.SEGMENT /DM status;
.VAR DSPSTAT;
.ENDSEG;

.SEGMENT /DM timer;
.VAR DSPTIMER;
.ENDSEG;

{ The following ports are found on the 32-channel ADC board      }

.SEGMENT /PM ioadin;
.VAR ADFIFO;
.ENDSEG;

.SEGMENT /PM iostat;
.VAR IOSTAT;
.ENDSEG;

.SEGMENT /DM iodaout;
.VAR DAFIFO;
.ENDSEG;
```

```
.SEGMENT /DM iochans;
.VAR CHANNELS;
.ENDSEG;

.SEGMENT /DM iocntrl;
.VAR CONTROL;
.ENDSEG;

{ division macro - if there is a quicker way, just change macro }
{                  macro requires the following:                }
{                  F0  = numerator                              }
{                  F12 = denominator                            }

{-------------------------------------------------------------------------}
{ Division Algorithm - Given Q = D/N, multiply N and D by the same set  }
{                      of factors, Rn.                                  }
{      N x R0 x R1 x ... x Rn                                           }
{  Q = ----------------------                                           }
{      D x R0 x R1 x ... x Rn                                           }
{                                                                       }
{ Choose Rn such that as the number of factors increases, the           }
{ denominator approaches 1.  The quotient is then approximately equal   }
{ to the numerator.                                                     }
{                                                                       }
{ R0 is the seed provided by the RECIPS instruction.  Succssive Rn are  }
{ calculated by the following formula:                                  }
{  Ri = 2-D(i-1)                                                        }
{                                                                       }
{ NOTE:  The macro uses the following registers:                        }
{        F0, F7, F11, F12                                               }
{        These registers will be over-written upon exit from macro.     }
{-------------------------------------------------------------------------}

#define div     F11 = 2.0; \
                F0 = RECIPS F12, F7 = F0; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F0 = F0*F7

#define DIV     F11 = 2.0; \
                F0 = RECIPS F12, F7 = F0; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F12 = F0*F12; \
                F7 = F0*F7, F0 = F11-F12; \
                F0 = F0*F7

{ D/A board does not output 0 when zero sent.  These offsets    }
{ were determined through experimentation                       }

{ Without TJs board             }
{#define DA0OFFSET 0x020E}
{#define DA1OFFSET 0x01B3}
{#define DA2OFFSET 0x01F2}
{#define DA3OFFSET 0x01E6}
{#define DA4OFFSET 0x0230}

{ With the low pass filter and Butterworth bandwidth 2.83k}
#define DA0OFFSET  0x00000003
#define DA1OFFSET  0x00000000
#define DA2OFFSET  0x00000001
#define DA3OFFSET  0xFFFFFFFF
#define DA4OFFSET  0xFFFFFFFF
```

```
{ A/D board does not return 0 with no input.  These offsets     }
{ were determined through experimentation                       }

{ With the low pass filter and Butterworth bandwidth 2.83k}
{#define AD0OFFSET  0x00000907}
{#define AD1OFFSET  0x000008E6}
{#define AD2OFFSET  0x00000905}
{#define AD3OFFSET  0x00000904}
{#define AD4OFFSET  0x000008D5}
{#define AD5OFFSET  0x000008E4}
{#define AD6OFFSET  0x00000926}

#define AD0OFFSET  0x00000918
#define AD1OFFSET  0x000008B1
#define AD2OFFSET  0x000008F6
#define AD3OFFSET  0x000008E6
#define AD4OFFSET  0x000000AF
#define AD5OFFSET  0x0000088D
#define AD6OFFSET  0x00000926

{ D/A constants                                                 }
#define DAVOLTSMAX   5.0
#define DAVOLTSMIN  -5.0
#define DABITSMAX   32768.0

{ A/D constants                                                 }
#define ADVOLTSMAX   5.0
#define ADVOLTSMIN  -5.0
#define ADBITSMAX   8192.0

{ Timer constants                                                       }
#define CPUSPEED  33333333.333
#define SMPLSPEED   20000.0

#define BEARTOT       5
#define SAVEOFFSET  256
#define SAVEBEAR      0
#define DELAY         0

.SEGMENT /DM dm_data;
{ Conversion factors                                                    }
.VAR VOLT2BITS;
.VAR BITS2VOLT;

.VAR AD0INPUT;
.VAR AD1INPUT;
.VAR AD2INPUT;
.VAR AD3INPUT;
.VAR AD4INPUT;

.VAR AD0OFF;
.VAR AD1OFF;
.VAR AD2OFF;
.VAR AD3OFF;
.VAR AD4OFF;

.VAR DA0OFF;
.VAR DA1OFF;
.VAR DA2OFF;
.VAR DA3OFF;
.VAR DA4OFF;

.VAR DA0CONTROL;
.VAR DA1CONTROL;
.VAR DA2CONTROL;
.VAR DA3CONTROL;
.VAR DA4CONTROL;
```

```
{ Variables needed to save data in memory for later dumping }

.VAR SAVE_BEARING;
.VAR INPUT_OFFSET;
.VAR OUTPUT_OFFSET;
.VAR SAVE_END;
.VAR DELAY_COUNT;

.ENDSEG;

.SEGMENT /PM rst_svc;

{-------------------------------------------------------------------------}
{ At reset, the BANK registers are as follows:                            }
{       PMBANK1 = 0x800000                                                }
{       DMBANK1 = 0x20000000                                              }
{       DMBANK2 = 0x40000000                                              }
{       DMBANK3 = 0x80000000                                              }
{ These values, by coincidence, are perfect for our I/O board             }
{-------------------------------------------------------------------------}

{-------------------------------------------------------------------------}
{ The default value of PMWAIT at reset is 0x0003DE.  This corresponds     }
{ to the following:                                                       }
{       bit  13    = 0   (No automatic wait state)                        }
{       bits 12-10 = 000 (memory page size = 256 words)                   }
{       bits  9-7  = 111 (7 PMBANK1 wait states)                          }
{       bits  6-5  = 10  (Int. and Ext. wait state ack mode)              }
{       bits  4-2  = 111 (7 PMBANK0 wait states)                          }
{       bits  1-0  = 10  (Int. and Ext. wait state ack mode)              }
{                                                                         }
{ For our setup, the DSP board accesses memory at 0 wait states and       }
{ accesses all ports and 1 wait state.  Therefore:                        }
{       bit  13    = 0   (No automatic wait state)                        }
{       bits 12-10 = 100 (memory page size = 4096 words)                  }
{       bits  9-7  = 001 (1 PMBANK1 wait state)                           }
{       bits  6-5  = 10  (Int. and Ext. wait state ack mode)              }
{       bits  4-2  = 000 (0 PMBANK0 wait states)                          }
{       bits  1-0  = 10  (Int. and Ext. wait state ack mode)              }
{-------------------------------------------------------------------------}

               PMWAIT = 0x0010C2;

{-------------------------------------------------------------------------}
{ The default value of DMWAIT at reset is 0x000F7BDE.  This corresponds   }
{ to the following:                                                       }
{       bit  23    = 0   (No automatic wait state)                        }
{       bits 22-20 = 000 (memory page size = 256 words)                   }
{       bits 19-17 = 111 (7 DMBANK3 wait states)                          }
{       bits 16-15 = 10  (Int. and Ext. wait state ack mode)              }
{       bits 14-12 = 111 (7 DMBANK2 wait states)                          }
{       bits 11-10 = 10  (Int. and Ext. wait state ack mode)              }
{       bits  9-7  = 111 (7 DMBANK1 wait states)                          }
{       bits  6-5  = 10  (Int. and Ext. wait state ack mode)              }
{       bits  4-2  = 111 (7 DMBANK0 wait states)                          }
{       bits  1-0  = 10  (Int. and Ext. wait state ack mode)              }
{                                                                         }
{ For our setup, the DSP board accesses memory at 0 wait states and       }
{ accesses all ports and 1 wait state.  Therefore:                        }
{       bit  23    = 0   (No automatic wait state)                        }
{       bits 22-20 = 100 (memory page size = 4096 words)                  }
{       bits 19-17 = 001 (1 DMBANK3 wait states)                          }
{       bits 16-15 = 10  (Int. and Ext. wait state ack mode)              }
{       bits 14-12 = 001 (1 DMBANK2 wait states)                          }
{       bits 11-10 = 10  (Int. and Ext. wait state ack mode)              }
{       bits  9-7  = 000 (0 DMBANK1 wait state)                           }
{       bits  6-5  = 10  (Int. and Ext. wait state ack mode)              }
{       bits  4-2  = 000 (0 DMBANK0 wait states)                          }
{       bits  1-0  = 10  (Int. and Ext. wait state ack mode)              }
```

```
{-------------------------------------------------------------------------}

                DMWAIT = 0x00431842;

{-------------------------------------------------------------------------}
{ Set FLAG2 to output mode so we can trigger I/O board conversion when  }
{ we need to.                                                           }
{-------------------------------------------------------------------------}

                MODE2 = 0x00020000;


                JUMP initialize;
.ENDSEG;


.SEGMENT /PM pm_code;

initialize:

{ initialize registers }

                IMASK = 0;
                MODE1 = 0x00012000;

                I0 = 0;
                I1 = 0;
                I2 = 0;
                I3 = 0;
                I4 = 0;
                I5 = 0;
                I6 = 0;
                I7 = 0;
                M0 = 0;
                M1 = 0;
                M2 = 0;
                M3 = 0;
                M4 = 0;
                M5 = 0;
                M6 = 0;
                M7 = 0;
                L0 = 0;
                L1 = 0;
                L2 = 0;
                L3 = 0;
                L4 = 0;
                L5 = 0;
                L6 = 0;
                L7 = 0;
{-------------------------------------------------------------------------}
{ This program does the following:                                      }
{       1.  Triggers A/D conversion on one channel                      }
{       2.  Polls A/D status register to see when conversion complete   }
{       3.  Pushes A/D value out to D/A                                 }
{       4.  returns to step 1                                           }
{-------------------------------------------------------------------------}

{ Make sure FLAG2 toggle is initially zero.                             }
                BIT CLR ASTAT 0x00200000;

{ Calculate D/A conversion factor                                       }
                F0 = DABITSMAX;
                F12 = DAVOLTSMAX;
                DIV;
                DM(VOLT2BITS) = F0;

{ Calculate A/D conversion factor                                       }
                F12 = ADBITSMAX;
                F0 = ADVOLTSMAX;
```

```
                DIV;
                DM(BITS2VOLT) = F0;

                B0 = DA0CONTROL;
                F3 = 0.0;
                F2 = DM(VOLT2BITS);
                F4 = DABITSMAX;
                LCNTR = BEARTOT;
                DO adj UNTIL LCE;
                  F1 = F3*F2;
                  F1 = F1+F4;
                  R0 = FIX F1;
adj:              DM(I0,1) = R0;

{ Setup variables required to saving data in memory for later recall    }
                R0 = SAVEOFFSET;
                DM(INPUT_OFFSET) = R0;
                R1 = ASHIFT R0 BY 1;
                R2 = 0x8000;
                R2 = R2-R1;
                R2 = ASHIFT R2 BY -1;
                R1 = R2+R0;
                DM(OUTPUT_OFFSET) = R1;
                R1 = R1+R2;
                DM(SAVE_END) = R1;
                R0 = SAVEBEAR;
                DM(SAVE_BEARING) = R0;
                R0 = DELAY;
                DM(DELAY_COUNT) = R0;

                B0 = AD0OFF;
                R0 = AD0OFFSET;
                DM(I0,1) = R0;
                R0 = AD1OFFSET;
                DM(I0,1) = R0;
                R0 = AD2OFFSET;
                DM(I0,1) = R0;
                R0 = AD3OFFSET;
                DM(I0,1) = R0;
                R0 = AD4OFFSET;
                DM(I0,1) = R0;

                B0 = DA0OFF;
                R0 = DA0OFFSET;
                DM(I0,1) = R0;
                R0 = DA1OFFSET;
                DM(I0,1) = R0;
                R0 = DA2OFFSET;
                DM(I0,1) = R0;
                R0 = DA3OFFSET;
                DM(I0,1) = R0;
                R0 = DA4OFFSET;
                DM(I0,1) = R0;


{ Reset I/O board                                                       }
                R0 = 0x0;
                DM(CONTROL) = R0;

{-----------------------------------------------------------------------}
{ Release angular reset (don^t know why) and set go mode to flag2 toggle}
{                                                                       }
{ Control Register:  [default - DM(0x40000002)]                         }
{       bits 2-0 -> go mode (0 = go on toggle of FLAG2)                 }
{                           (1 = go on interrupt)                       }
{       bits 5-3 -> IRQ mode                                            }
{       bit    6 -> Status select                                       }
{       bit    7 -> Analog reset                                        }
{-----------------------------------------------------------------------}
```

```
                R0 = 0x80;
                DM(CONTROL) = R0;

{ Set timer sampling period and counter                                    }

                R1 = BEARTOT;
                F0 = FLOAT R1;
                F12 = SMPLSPEED;
                F12 = F0*F12;

                F0 = CPUSPEED;
                DIV;
                R1 = FIX F0;
                R0 = R1-1;

                TPERIOD = R0;
                TCOUNT = R0;

{ Set up input/output save pointers in last address registers              }
                I6 = DM(INPUT_OFFSET);
                I7 = DM(OUTPUT_OFFSET);
                M3 = DM(SAVE_BEARING);

{ Reset interrupt latch register                                           }
                BIT SET IRPTL 0x0;

{ Allow timer interrupts                                                   }
                BIT SET IMASK 0x12;

{ Turn on timer                                                            }
                BIT SET MODE2 0x20;

{ Allow interrupt generation                                               }
                BIT SET MODE1 0x1000;

gag:             IDLE;
                JUMP gag;


sample:

{--------------------------------------------------------------------------}
{ Set 1 A/D channel and 0 D/A channels.  When a conversion is triggered,   }
{ both the A/D and D/A conversions are triggered.  Therefore, I set the    }
{ the number of D/A to 0 so that nothing goes out.                         }
{                                                                          }
{ Number of Channels Register:  [default - DM(0x40000001)]                 }
{       bits 2-0 -> number of A/D channels                                 }
{       bits 5-3 -> number of D/A channels                                 }
{--------------------------------------------------------------------------}

                R0 = BEARTOT;
                DM(CHANNELS) = R0;

{ Toggle FLAG2 to signal I/O board to start conversion                     }
                BIT SET ASTAT 0x00200000;
                NOP; NOP; NOP;
                BIT CLR ASTAT 0x00200000;

{--------------------------------------------------------------------------}
{ Wait for A/D conversion                                                  }
{                                                                          }
{ Status Register:  [default - PM(0x800001)]                               }
{       bits 0 -> 1 if D/A is busy                                         }
{       bits 1 -> 1 if A/D is busy                                         }
{       bit  2 -> 0 if D/A FIFO is empty                                   }
{       bit  3 -> 0 if A/D FIFO is empty                                   }
{       bit  4 -> 0 if D/A FIFO is full                                    }
{       bit  5 -> 0 if A/D FIFO is full                                    }
```

```
{                                                                          }
{ Note:  Make sure that both D/A and A/D are not busy before reading       }
{        from A/D FIFO otherwise bogus values are obtained.                }
{--------------------------------------------------------------------------}

                R1 = 0x03;
wait:           R0 = PM(IOSTAT);
                R7 = R0 AND R1;
                IF NE JUMP wait;

{--------------------------------------------------------------------------}
{ Get A/D value(s) from FIFO.  [default - PM(0x800000)]                    }
{                                                                          }
{ The values obtained are 14-bit 2^s-complement values sign extended to    }
{ the left.  2^s complement is obtained by negating the real value and     }
{ adding 1.  The range of values for our particular A/D is:                }
{                                                                          }
{       +5V = 0001 1111 1111 1111                                          }
{        0V = 0000 0000 0000 0000                                          }
{       -5V = 1110 0000 0000 0001                                          }
{                                                                          }
{ Sample code for converting 2^s complement notation to floating point    }
{ is as follows:                                                           }
{               R1 = FEXT R0 BY 0:16 (SE);                                 }
{               F0 = FLOAT R0;                                             }
{                                                                          }
{ Sample code for converting floatin point numbers to 2^s complement       }
{ notation is as follows:                                                  }
{               F0 = FIX R0;                                               }
{                                                                          }
{--------------------------------------------------------------------------}
                B0 = AD0INPUT;
                LCNTR = BEARTOT;
                DO read1 UNTIL LCE;
                  R0 = PM(ADFIFO);
read1:            DM(I0,1) = R0;

                B0 = AD0INPUT;
                LCNTR = BEARTOT;
                DO convert UNTIL LCE;
                  I1 = I0;
                  R0 = DM(I0,1);
                  R0 = FEXT R0 BY 0:14 (SE);
                  R1 = DM(5,I1);
                  R1 = R0+R1;
                  F0 = FLOAT R1;
                  F1 = DM(BITS2VOLT);
                  F1 = F0*F1;
                  F0 = DM(VOLT2BITS);
                  F0 = F0*F1;
                  R1 = DM(10,I1);
                  F1 = FLOAT R1;
                  F0 = F0+F1;
                  F1 = DABITSMAX;
                  F1 = F1+F0;
                  R0 = FIX F1;
convert:          DM(15,I1) = R0;

{--------------------------------------------------------------------------}
{ Set 1 A/D channel and 0 D/A channels.  When a conversion is triggered,   }
{ both the A/D and D/A conversions are triggered.  Therefore, I set the    }
{ the number of D/A to 0 so that nothing goes out.                         }
{                                                                          }
{ Number of Channels Register:  [default - DM(0x40000001)]                 }
{       bits 2-0 -> number of A/D channels                                 }
{       bits 5-3 -> number of D/A channels                                 }
{--------------------------------------------------------------------------}

                R0 = BEARTOT;
```

```
                R1 = LSHIFT R0 BY 3;
                DM(CHANNELS) = R1;

                B0 = DA0CONTROL;
                LCNTR = BEARTOT;
                DO dummy2 UNTIL LCE;
                  R0 = DM(I0,1);
dummy2:           DM(DAFIFO) = R0;

{ Save original input value before we over-write it                    }

#ifdef RAW
                R0 = DM(DELAY_COUNT);
                R1 = PASS R0;
                IF NE JUMP sskip3;
                R0 = I6;
                R1 = DM(OUTPUT_OFFSET);
                R0 = R1-R0;
                IF EQ JUMP sskip3;
                B1 = AD0INPUT;
                R0 = DM(I1,M3);
                R0 = DM(I1,0);
                DM(I6,1) = R0;
#endif

#ifdef PLOT
                R0 = DM(DELAY_COUNT);
                R1 = PASS R0;
                IF NE JUMP sskip3;
                R0 = I6;
                R1 = DM(OUTPUT_OFFSET);
                R0 = R1-R0;
                IF EQ JUMP sskip3;
                B1 = AD0INPUT;
                R0 = DM(I1,M3);
                R0 = DM(I1,0);
                R0 = FEXT R0 BY 0:14 (SE);
                DM(I6,1) = R0;
#endif

sskip3:
{ Toggle FLAG2 to signal I/O board to start conversion                 }
                BIT SET ASTAT 0x00200000;
                NOP; NOP; NOP;
                BIT CLR ASTAT 0x00200000;
                NOP; NOP; NOP;

                R1 = 0x03;
wait2:           R0 = PM(IOSTAT);
                R7 = R0 AND R1;
                IF NE JUMP wait2;

{ Save new output values                                               }
#ifdef RAW
                R0 = DM(DELAY_COUNT);
                R1 = PASS R0;
                IF NE JUMP sskip4;
                R0 = I7;
                R1 = DM(SAVE_END);
                R0 = R1-R0;
                IF EQ JUMP sskip4;
                B1 = DA0CONTROL;
                R0 = DM(I1,M3);
                R0 = DM(I1,0);
                DM(I7,1) = R0;
#endif

#ifdef PLOT
                R0 = DM(DELAY_COUNT);
```

```
                R1 = PASS R0;
                IF NE JUMP sskip4;
                R0 = I7;
                R1 = DM(SAVE_END);
                R0 = R1-R0;
                IF EQ JUMP sskip4;
                B1 = DA0CONTROL;
                R0 = DM(I1,M3);
                R0 = DM(I1,0);
                R0 = FEXT R0 BY 0:16;
                F1 = DABITSMAX;
                R1 = FIX F1;
                R0 = R0-R1;
                DM(I7,1) = R0;
#endif

sskip4:
                R0 = DM(DELAY_COUNT);
                R1 = PASS R0;
                IF EQ JUMP ret;
                R0 = R0-1;
                DM(DELAY_COUNT) = R0;

ret:
                R1 = PM(IOSTAT);
                DM(DSPSTAT) = R1;

                RTI;

.ENDSEG;


.SEGMENT /PM tmzh_svc;

                JUMP sample;

.ENDSEG;
```

# Appendix K

# The Rest of the Story

---

During the course of writing this thesis, there were certain pieces of information that this researcher thought would be of value to anyone working with the application but were difficult to justify including in the body of the thesis. This final appendix is an effort to fill the gaps in the information given thus far and to discuss certain issues that will help any student wishing to work on this system in the future.

## K.1 Digital Controller Testing

Your digital controller should be tested under both static and dynamic conditions. By static, I'm referring to the state in which the rotor is not spinning. Dynamic refers to the case where the rotor is spinning. Static testing is most easily performed with the clear plastic disk removed from the top of the turbopump. Controller failure can manifest itself in three ways. The first way is when the rotor alternates hitting the touchdown bearings on each side of a particular axis. This failure mode is obvious due to the noise it generates. The second way is when the rotor hits the touchdown bearing on one side of a particular axis, remains there for some time, moves toward the middle of the bearing, and strikes the same or the opposing touchdown bearing of the same axis. This failure mode is characterized by a sharp rap every time the rotor strikes the touchdown bearing. The third and final way is when the rotor strikes one side of a particular axis and stays there indefinitely. This failure mode is characterized by one sharp rap and then nothing.

The proper way to determine if the controller is working satisfactorily is to monitor the position signals of each axis. The easiest way however is to reach through the top of the turbopump and give the rotor a spin. Spinning the rotor by hand is also be used to test controller robustness. Once you believe you have a stable system, reach down through the turbopump top and give the rotor a small spin. Remember that the rotor is magnetically levitated and subject to almost no friction and therefore it will take very little effort to spin the rotor. You want to provide just enough momentum to spin the rotor approximately one revolution. This slow revolution with automatically cause the rotor to stop at its worst case position. This worse case position may be due to rotor imbalance or controller dynamics. Whatever the cause, at this worst case position, the displacement of the rotor is greatest and in the case of a marginally stable controller, can lead to instability.

Once a stable controller has been produced during static testing, its time to move on to dynamic testing. The clear plastic disk must now be reinstalled so spinning the rotor manually is not an option. Once a stable controller is produced during static testing, it is normally sufficient to monitor one axis of both the upper and lower bearing to determine stability. The final speed of the rotor is determined by two precision potentiometers mounted on the project box which normally sits on top of the analog controller box. The relationship between the potentiometer resistance values and the rotor speed must be determined by trial and error using the analog controller. Once the potentiometers are set in the appropriate values using the analog controller, determine the potentiometer resistances using an ohm meter. Do not rely on the values on the potentiometer dials.

Now that the rotor speed is set, turn on the auxiliary vacuum pump. The manufacture could not guarantee proper operation if there was an absence of vacuum at the intake. Therefore they provided an auxiliary vacuum pump to supply the necessary intake vacuum. Before starting the auxiliary vacuum pump, verify that the oil level is at the center of the circle in the clear circular indicator. If oil level is low, add the oil provided by the manufacturer. When the auxiliary vacuum pump is turned on, it will emit a small amount of oily smoke so perhaps opening a window is appropriate. Let the auxiliary vacuum pump operate for some time to assume that sufficient vacuum has built up.

Next, start the analog controller box and your digital controller. Check the position signals to verify that the system is stable. Push the start button to start the rotor spinning. When the start button is pushed, the RPM indicator will show the current speed and the "accelerating" light will be on. The rotor takes approximately fifteen minutes to reach is final operating speed no matter how high or low that final speed is. As the rotor approaches its final speed, the "accelerating" light goes out and the RPM indicator increases very slowly. Wait until the RPM indicator steadies before doing any further testing. Once testing is complete, push the stop button. This will cause the "decelerating" light to come on and the RPM indicator to fall. Wait till the rotor reaches about 300 RPM before turning the analog control box off and letting the touchdown bearings bring the rotor to a halt.

When the digital controller fails during the course of testing (and it will fail), the most important thing you can do is to do nothing. The worst possible thing that you could do is turn off the analog controller box. Instead just sit there and smile because everyone within hearing distance will know something went wrong. Not only will the rotor be hitting the touchdown bearings with all the intensity of a jackhammer, but the turbopump will purge vacuum thereby making a loud hissing noise. The analog controller box will also be lit up like a Christmas tree but more important, the rotor will automatically be decelerating. Generally, the digital controller will be able to regain stability at some point as the rotor decelerates. Dynamic testing is best performed when there is nobody else in the lab. Also, it is almost a certainty that the first bearing to fail will be one of the axis of the lower bearing (2X or 2Y) so monitor that bearing carefully during dynamic testing.

It is a good practice to periodically adjust the position sensors. This is best accomplished with the aid of helper. Remove the clear plastic disk from the top of the turbopump. Remove the ribbon cable connectors from the power amplifier circuit board. This board lies between the analog controller circuit boards and the actual power amplifier heat sinks at the back. Removing each ribbon cable isolates the controller from the bearing leaving that bearing unpowered. Turn

on the analog controller box and monitor the position signal. Turning on the box allows the position sensors to work but since the controllers are disconnected from the power amps, the rotor is lying on its touchdown bearings. Have your helper move the rotor through its entire displacement range and using an oscilloscope monitor the maximum and minimum voltage returned. Adjust the position sensor offsets by turning the proper potentiometer for each axis on the analog controller circuit boards. A properly adjusted position sensor has the same minimum and maximum absolute value.

Finally, I would like to discuss the manual tuning process. Manually tuning the axial bearing digital controller can be performed while the analog controller controls the radial bearings. However, you can not manually tune a radial bearing axis while any of the other axes are under analog control. The radial bearing tuning process must be performed while all of the radial bearings are under digital control to produce meaningful results. Also, testing a digital controller on just one radial bearing axis while leaving the remaining radial bearing axes under analog control will produce meaningless results. Results obtaining using this method only show how good the analog controllers are and say nothing about the performance of the digital controller. It is important that the all radial bearings be treated as a complete system and not as individual entities because their performance is closely coupled.

## K.2 The Project Box

The Project box is the aluminum box which generally sits on top of the analog controller box. The back face of the box contains two DB-25 ports, one male and one female. One port connects to a ribbon cable which in turn connects with various test points on the analog controller boards. The other port connects to another ribbon cable which attaches to the filter box underneath the I/O Interface Board. The top of the box contains two high precision potentiometers and five switches. The potentiometers are responsible for determining the rotation speed of the rotor. The switches were meant to control whether the analog or digital controller was in use however this was abandoned due to implementation difficulties. The sloping front face of the box contains one DB-50 port. This allows monitoring of all of the signals entering and leaving the Project box.

The decision was made at one point to try to provide easy access to the majority of the test points on the analog circuit boards. Prior to this time, access to the test points was provided by circuit board extenders which allowed the entire analog control board to extend beyond its protective enclosure. This provided complete access to the board but also placed the board in harm's way. There were only two of these extender boards so they had to be swapped often which lead to snarled wiring and bent components due to the tight clearances between boards. After much discussion, a design was agreed upon which would allow access to all important test points and yet allow the boards to remain in their protective enclosure. Because of the tight clearances between boards and between the board enclosure and the front panel, ribbon cable was chosen as the signal conveyor.

At first, we decided to place the controller switches on the project box. This necessitated bringing the analog board signal all the way from the analog board, through the ribbon cable to the project box switch, and back through the ribbon cable to the analog board. However, after

completing the necessary connections, both the analog and digital controllers refused to work. The resistance across the ribbon cable and switch were not measurable using a digital ohm meter so increased resistance did not seem to be a likely problem. After much trial and error, the only solution was to mount the switches on each of the circuit boards.

In retrospect, maybe ribbon cable was not such a good idea. Perhaps there was cross-over interference or noise induced by the other signals carried along the ribbon cable. The only way to eliminate interference between signals on a ribbon cable is to place a ground as the wire between any two signal carrying wires. Such a change would also have doubled the size of the ribbon cable and made very clumsy to work with. However, since this modification was not carried out, it does raise the question of whether other signals are also being effected by this ribbon cable configuration. Previous to the ribbon cable modification, the signals were carried by everyday speaker wire which has thicker insulation which holds both a signal and ground wire side by side. The project box did however allow greater simultaneous access to all of the test points which proved very convenient.

In an effort to eliminate any possible sources of noise from the system, the position signal adders on each of the analog controller boards were disabled. Disabling took the form of soldering a wire from the input of the adder to the output. Also the connection between the input and the first op-amp was broken and the adder test points were grounded. The adder was no longer needed because by then we had obtained all of the possible system analysis data. These modifications can be easily undone by looking at the circuit board and the circuit schematic provided by the manufacturer.

Another switch was added to each circuit board to eliminate a notch filter added by the manufacturer. This notch filter attenuated signals that might excite the second bending mode. However this notch filter was not implemented as one might expect. The notch filter was actually a band pass filter which only passed frequencies that would excite the second bending mode. However, the signal produced from the band pass filter was the opposite sign of the input signal. This negative signal was added to the control signal thereby canceling any signals having frequencies within the bandpass filter range. The problem was that the band pass filter took its input before the point where the output of the digital controller was spliced back into the analog control board and added its negative signal after that point. In order to remove the noise that this filter would create when the digital controller was operative, a switch was used to ground the signal emulating from the band pass filter.

## K.3 Turbopump Noise

The noise referred to in this section is the audible noise emitted by the turbopump. Under analog control, the turbopump emits very little noise during both static and dynamic operation. However this is not the case with all of the entirely digital controllers implemented thus far. This is particularly true during static testing when the clear plastic disk on top of the turbopump was removed. Even with the disk in place, the turbopump emits a high pitched whine which would probably make it unusable in a typical work environment. The noise is not painful but it is very uncomfortable. The noise does tend to decrease as the digital controller low pass filter's cutoff frequency decreases.

Accompanying this noise is vibration of the turbopump housing. This is not a visible vibration but it can be felt when actually touching the housing. To dampen this vibration, I placed thick manuals on top of the turbopump housing in the hope of altering the dynamic characteristics of the housing. This seemed to have the effect of increasing system stability at higher RPMs. The final height of the manuals placed on top of the housing was thirteen inches.

## K.4 Radial Bearing Coupling Experiments

An effort was made to determine the coupling transfer function between the same axes of the upper and lower bearing. This section will outline the procedure for these experiments and the reason for their abandonment. The experiments were performed satisfactorily for the upper bearing and I will use the test procedure for the 1X radial bearing as an example. The analog controller box was turned on with the 1X, 1Y, and axial axes under analog control. Both the 2X and 2Y axes were unpowered. The swept sine input from the HP System Analyzer was attached to the 1X radial bearing control signal adder. The output to the System Analyzer was attached to the position signal of the 2X bearing. The position signals of both the 2X and 2Y axes were monitored to assure that the rotor did not contact the touch down bearings during testing. Any contact would require retesting and perhaps lowering the amplitude of the input disturbance signal.

When this procedure was used to determine the coupling transfer function of the lower bearing, we ran into problems. When testing the upper bearing with an powered lower bearing, the rotor would naturally rest near the center of the unpowered bearing. However, with an unpowered upper bearing and a lower bearing under analog control, the rotor would lean to one side or the other of the unpowered bearing. This was perfectly natural considering the center of gravity of the rotor assembly is much closer to the upper bearing then to the lower bearing. However, this leaning always resulted in the rotor contacting the touch down bearings during testing. We then decided that this leaning may be considerably less if the turbopump was tested while upside down. In this way, the unpowered bearing would again be below the powered bearing. However, when the turbopump was turned over, the axial bearing refused to move the rotor to the equilibrium position. The rotor remained in contact with the touch down bearings of the axial upper bearing. Using both the analog and digital controller to control the axial bearing had no effect. These experiments were abandoned soon afterward. I guess the manufacturer never considered the case where a buyer might mount the turbopump on the ceiling.

[1] R. N. Arnold and L. Maunder, *Gyrodynamics and its Engineering Applications*, Academic Press, N.Y., 1961

[2] K. J. Åstrom and B. Wittenmark, *Computer Controlled Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1984

[3] M. Athans, *Lecture Notes in Multivariable Control Systems*, Department of Electrical Engineering, Massachusetts Institute of Technology, 1993

[4] Battery and EV Technology, *HYBRID VEHICLES: SatCon Develops Award Winning Technology*, Vol. 20, No. 9, Jan., 1996

[5] Birch and Stuart, "Advanced Research at BMW", *Automotive Engineering*, Vol. 102, No. 10, Pg. 12, Oct., 1994

[6] S. H. Crandall, D. C. Karnopp, and E. F. Kurtz, *Dynamics of Mechanical and Electromechanical Systems*, pp 224-246, R. E. Krieger Publishing Co., Malabar, FL., 1982

[7] V. Del Toro, *Electrical Engineering Fundamentals*, Prentice-Hall, Englewood Cliffs, N.J., 1986

[8] A. D. Dimarogonas and S. A. Paipets, *Analytical Methods in Rotor Dynamics*, pp 41-71, Applied Science Publishers, New York, N.Y., 1983

[9] S. Earnshaw, "On the Nature of Molecular Forces", *Transactions at the Cambridge Philosophical Society*, Vol. 7, Pg. 97-112, 1842

[10] T. Elliott, "Electric Energy Storage Hinges on Three Leading Technologies", *Power*, Vol. 139, No. 8, Pg. 42, Aug., 1995

[11] W. Klages, "The Cyroturbo: A New Concept in Vacuum Technology", *Solid State Technology*, Vol. 37, No. 4, Pg. 63, Apr., 1994

[12] B. C. Kuo, *Automatic Control Systems Fourth Edition*, Prentice-Hall, Englewood Cliffs, N.J., 1982

[13] B. C. Kuo, *Analysis and Synthesis of Sampled-Data Control Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1963

[14] P. A. Lynn and W. Fuerst, *Introductory Digital Signal Processing with Computer Applications*, John Wiley and Sons, N.Y., 1989

[15] L. O'Connor, "Electric Vehicles Move Closer To Market", *Mechanical Engineering - CIME*, Vol. 117, No. 3, Pg. 82, Mar., 1995

[16] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*, pp. 195-230, Prentice-Hall, Englewood Cliffs, N.J., 1975

[17] S. Reddy, "Theory of Time Delay Control and Application to Magnetic Bearings", Ph.D. thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, Sep., 1992

[18] F. W. Sears, M. W. Zemansky, and H. D. Young, *University Physics*, Addison-Wesley, Reading, MA., 1987

[19] I. H. Shames, *Engineering Mechanics*, Prentice-Hall, Englewood Cliffs, N.J., 1980

[20] S. M. Shinners, *Control System Design*, pp. 350-416, John Wiley and Sons, N.Y., 1964

[21] Urban Transport News, *CalStart Announces Projects for Hybrid Electric Bus Technology*, Vol. 24, No. 3, Jan. 1996

[22] J. Wetzel, "Time Delay Control of Magnetic Bearings", M.S. thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, Jan., 1992

[23] K. Youcef-Toumi, I. Vithiananthan, and S. Reddy, "Implementation of Time Delay Control to Active Magnetic Bearings, Final Report", *Laboratory of Manufacturing and Productivity Report*, Massachusetts Institute of Technology, Sep., 1989

[24] K. Youcef-Toumi and O. Ito, "Model Reference Control using Time Delay for Nonlinear Plants with Unknown Dynamics", Proceeding of the International Federation of Automatic Control World Congress, Munich, Federal Republic of Germany, July, 1987