DEVELOPMENT OF INTERACTIVE MODELING, SIMULATION, ANIMATION, AND REAL-TIME CONTROL (MoSART) TOOLS FOR RESEARCH AND EDUCATION

by

Chen-I Lim

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 1999

DEVELOPMENT OF INTERACTIVE MODELING, SIMULATION, ANIMATION,

AND REAL-TIME CONTROL (MoSART) TOOLS FOR RESEARCH AND EDUCATION

by

Chen-I Lim

has been approved

December 1999

APPROVED:

______________________________, Chair

____

____

Supervisory Committee

ACCEPTED:

____

Department Chair

____

Dean, Graduate College

ABSTRACT

This thesis describes the development of several Modeling, Simulation, Animation, and Real-Time Control (MoSART) tools that are useful for research, designing of control systems, and education.

These tools include an Interactive (MoSART) Environment for analyzing, designing, visualizing and evaluating robust multivariable controller performance for complex helicopter systems. The described *Interactive MoSART Helicopter Environment* is based on Microsoft Windows NT/95, Visual C++, Microsoft Direct-3D, and MATLAB/SIMULINK. The environment consists of several key modules: (i) a program user-interface module (PIM), (ii) a simulation module (SIM), (iii) a graphical animation (GAM) module, (iv) a MATLAB communication and Analysis Module (CAM), (v) a help/instruct module (HIM), and (vi) an animation laboratory (A-Lab) module. The program user-interface module allows the user to interact with the program. Users may select model parameters, control laws, control law gains, reference commands, disturbances, initial conditions, integration routines, visual indicators, and graphics to be displayed. The simulation module is responsible for solving the equations which govern the motion of the helicopter system. The graphics/animation module updates plots, visual indicators, and 3D animations on the screen using data generated by the simulation module. The help/instruct module consists of detailed system/control law documentation that may be viewed/searched/accessed from within a browser. The environment also accommodates data exchange with MATLAB. This feature is useful for importing new system models and for control system analysis and redesign. This makes the developed environment very extensible with respect to mathematical modeling and control. Users may readily export simulation data to MATLAB and all of the associated toolboxes for post-processing and further analysis.

The development of a stand-alone animation and visualization tool known as Animation-Lab (A-LAb) is also described. This feature permits users to drive the Direct-3D animation models directly from a SIMULINK block diagram. Additional Direct-3D animation models may be readily imported. User-issued joystick commands are supported within the environment and within A-Lab.

A framework for performing distributed computation is also described. This framework permits large-scale and complex problems to be solved using a network of PCs and MATLAB.

The ability to control actual hardware in real-time via data-acquisition boards is also discussed. A real-time cart-position control system developed in Visual C++ and using a data-acquisition board is presented.

Examples are presented to illustrate the utility of the developed tools. It is specifically shown how they may be used to gain fundamental understanding with respect to modal analysis, linear approximations of nonlinear models, model extensibility, and multivariable system analysis, design and evaluation. As such, the developed tools are shown to be a valuable for enhancing both research and education.

ACKNOWLEDGMENTS

To my family and friends. To those who have made the journey possible, those who have made the journey bearable, and those who have made the journey worthwhile.

This research has been supported, in part, by a White House Presidential Excellence Award, by National Science Foundation (NSF) Grant # 9851422, by the Western Alliance to Expand Student Opportunities (WAESO), the Coalition to Increase Minority Degrees (CIMD), the ASU Center for Innovation in Engineering Education (CIEE), the Boeing A.D. Welliver Faculty Fellowship Program, Intel Corporation, Microsoft, CADSI, Knowledge Revolution, and Integrated Systems.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

CHAPTER 1

Introduction: The Need for Interactive Visualization Tools

New Technologies. Today's affordable PC technology, object-oriented-programming (OOP) languages, and other software development tools now permit the development of new interactive graphical visualization environments - environments that could revolutionize systems and controls analysis, design, research, and education. These technologies have been exploited by MoSART researchers at ASU to develop several system-specific environments for a variety of systems [44] - [81]. This thesis describes the development of several key MoSART tools. Helicopter-based systems, which are complex, open-loop unstable, highly coupled multiple-input-multiple-output (MIMO) systems, provide an excellent vehicle for demonstrating the utility of the developed tools. This body of work includes discussion of an *Interactive MoSART Helicopter Environment*, Animation-Lab (A-Lab), a framework for distributed computation over a network, and a framework for real-time hardware control using PCs and data-acquisition boards. The details of how these tools were developed is the focus of this thesis.

The Need For Design Tools. Because of ever increasing performance requirements, the need for systematic multivariable control system design techniques [92], [93] that suitably accommodate MIMO cross-coupling effects and uncertainty have become essential. Since control system design is a highly iterative process, tools that facilitate the evaluation, "visualization", and redesign of

MIMO control laws are invaluable to designers. Given this, tools that simultaneously accommodate interactive modeling, simulation, analysis, graphical visualization, animation, design, and real-time control features are of particular use to designers. Only until very recently, the development of such tools has been restricted to mainframe and workstation platforms. This thesis shows that such development may be effectively carried out using relatively affordable PC hardware and software technologies.

Contributions of Work. This thesis demonstrates how affordable state-of-the-art PC technologies may be combined to develop high quality *system-specific Interactive MoSART Environments* which are useful for enhancing both research and education. Specifically, this paper describes a PC/Windows NT/95/Visual C++/MATLAB/SIMULINK-based *Interactive MoSART Helicopter Environment*. This environment permits control system engineers and students to analyze, design, and visualize the performance of controllers via real-time and faster-than-real-time 2D and 3D animation.

Through a user-friendly graphical interface, users can alter model, controller, and signal parameters on-the-fly while commands are issued by either a program function generator or a user-controlled joystick. The *system-specific* nature of the environment permits it to exploit the dynamical and animation model structures which are not exploited in general purpose simulation and animation packages. As such, the environment discussed within offers significant advantages over general-purpose packages that have recently emerged (e.g. Working Model [42], DADS/Plant [40], SimMaster 3D [41]). One major feature of the environment is the ability to establish a direct link with SIMULINK and access the MATLAB 5 suite of toolboxes (e.g. optimization, system identification, signal processing, etc.) via the MATLAB computing engine. These features make the environment an ideal centerpiece for a highly extensible virtual design and test platform. The utility of the environment is clearly demonstrated within this paper through examples addressing fundamental issues that are of concern to all control system designers. A copy of the executable

Interactive MoSART Helicopter Environment code may be obtained from the *MoSART* web site at: *http://www.eas.asu.edu/~aar/research/mosart/projects/projects.html.*

This document also describes the development of the software. The environment was written in Microsoft Visual C++, and takes full advantage of object-oriented programming techniques. It is demonstrated how the use of these methodologies make the code highly modular and extensible. The core of the software has allowed the *Interactive MoSART Helicopter Environment* to serve as the platform for developing other interactive environments for different systems [47], [51], [72].

The development of Animation-LAb (A-Lab) is also discussed. This feature permits users to drive the Direct-3D animation models directly from a SIMULINK block diagram. Additional Direct-3D animation models may be readily imported. User-issued joystick commands are supported within the environment and within A-Lab.

A framework for performing distributed computation is also presented. This framework permits large-scale and complex problems to be solved using a network of PCs and MATLAB.

The ability to control actual hardware in real-time via data-acquisition boards and software-based controllers is also discussed. A real-time cart-position control system developed in Visual C++ and using a data-acquisition board is presented.

Literature Survey. The following references relate to helicopters and aerodynamics [4] - [12]. The following references describe interactive environments [13] - [43]. *Interactive MoSART Environments* which are under development at ASU are described in [44] - [81]. Control system design methodologies are described in [82] - [93]. Computer aided design (CAD) software technologies for controls are described in [94] - [108]. Windows NT, DirectX and Direct3D are described in [109] - [125].

Outline. The remainder of this thesis is organized as follows. Chapter 2 describes the mathematical models and control laws for helicopter systems that are used within this thesis. In Chapter 3, the environment's key features are described, as is the development process, and the utility of the environment as a research and educational tool is demonstrated. Chapter 4 describes the development and utility of A-Lab. Chapter 5 describes the framework for a MATLAB-based distributed computation toolbox. Chapter 6 describes real-time hardware control interfaces, and a cart-track position control system. Chapter 7 summarizes the thesis and presents directions for future research.

CHAPTER 2

Helicopter Models and Control Laws

This chapter describes the mathematical models and control laws for the helicopter systems discussed in this thesis. All of the systems are based on the negative-feedback structure shown in Figure 2.1.

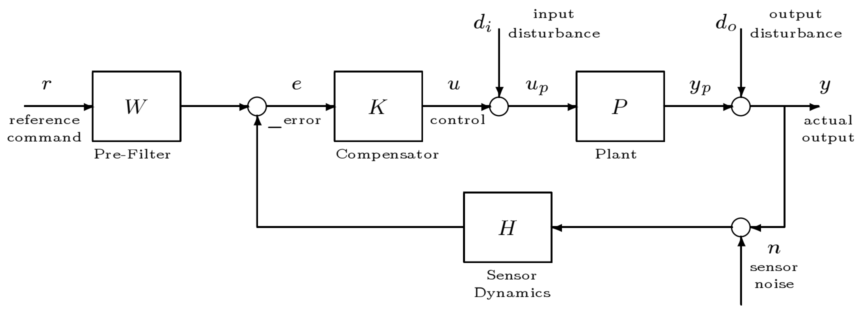


Figure 2.1: Standard Negative Feedback System with Pre-Filter

2.1 3-DOF UH-60 Blackhawk Model and Control Laws

One simple model implemented within the *interactive MoSART helicopter environment* is that of a 3-degree-of-freedom Sikorsky UH-60 Blackhawk helicopter [11].

2.1.1 3rd-order Blackhawk System Model and Dynamics

At hovering trim, the longitudinal and vertical dynamics are essentially decoupled from one another. This allows us to study both of them as single-input-single-output (SISO) systems. By so doing, insight is developed that may be applied to more complicated systems and models. Table 2.1 lists aerodynamic and control coefficients for the Blackhawk helicopter near hovering trim. These will be used to introduce the defining vertical and longitudinal (horizontal-pitching) dynamics.

$Z_{\Theta_c} = 5.95\ ft/deg\ sec^2$
$Z_w = -0.346\ sec^{-1}$
$g = 32.283\ ft\ sec^{-2}$
$X_u = -0.06 ft\ sec^{-2}/sec^{-1}$
$M_q = -3.1\ deg\ sec^{-2}/deg\ sec^{-1}$
$X_{Blc} = 0.478\ ftsec^{-2}/deg$
$M_u = 2.3493\ deg\ sec^{-2}/ft\ sec^{-1}$
$M_{Blc} = -47.24\ deg\ sec^{-2}/deg$

Table 2.1: Blackhawk Aerodynamic Coefficients Near Hovering Trim

Vertical Dynamics. The vertical dynamics of a helicopter near hover may be described by a second order linear ordinary differential equation involving the vertical altitude z (ft) and the collective pitch control Θ_c (deg). The associated transfer function takes the form:

$$\frac{z}{\Theta_c} = \frac{Z_{\Theta_c}}{s(s + Z_w)} \tag{2.1}$$

where Z_{Θ_c} is a control derivative and Z_w is an aerodynamic derivative - both given in Table 2.1.

Helicopter Longitudinal Dynamics. The longitudinal dynamics of a helicopter near hover may be described by two third-order linear ordinary differential equations. The associated transfer functions which relate the horizontal speed $\dot{x}$ (ft/sec) and the pitch attitude θ (deg) to the cyclic pitch control B_{l_c} (deg) are of the form:

$$\frac{\dot{x}}{B_{l_c}} = X_{Blc}\left[\frac{s^2 - M_q s - \frac{gM_{Blc}}{X_{Blc}}}{s^3 - (X_u + M_q)s^2 + M_q X_u s + gM_u}\right] \tag{2.2}$$

$$\frac{\theta}{B_{l_c}} = M_{Blc}\left[\frac{s + (\frac{X_{Blc}M_u}{M_{Blc}} - X_u)}{s^3 - (X_u + M_q)s^2 + M_q X_u s + gM_u}\right] \tag{2.3}$$

where $(X_u,\ M_u,\ M_q)$ are aerodynamic derivatives, and $(M_{Blc},\ X_{Blc})$ are control derivatives - the values for these parameters for the helicopter near hover are given in Table 2.1. At this operating point, the equations of motion are:

$$\frac{\dot{x}}{B_{l_c}} = \frac{27.4s^2 + 84.94s + 1525}{s^3 + 3.16s^2 + 0.186s + 1.324} \tag{2.4}$$

$$\frac{\theta}{B_{l_c}} = \frac{-47.24s - 1.711}{s^3 + 3.16s^2 + 0.186s + 1.324} \tag{2.5}$$

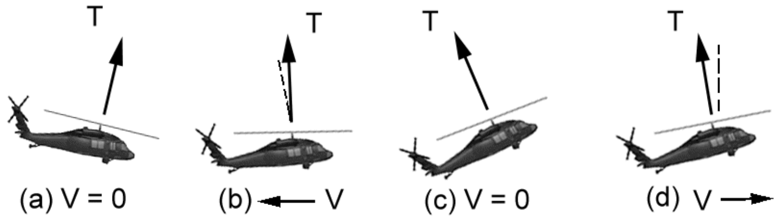


Figure 2.2: Visualization of Backflapping Instability Characteristic of Hovering Helicopters

Horizontal Instability Due to Rotor Backflapping. All single rotor helicopters - such as the UH-60 Blackhawk - exhibit a natural instability near hovering trim [5], [11], [12]. This instability is due to the backflapping of the main rotor with forward motion. To understand this instability, let's suppose that the helicopter experiences a small horizontal velocity disturbance (Figure 2.2a). The relative airspeed from a horizontal velocity disturbance causes the main rotor to tilt backwards and exert a nose-up pitching moment about the helicopter's center-of-gravity. A nose-up pitch attitude then begins to develop and the backward component of rotor thrust decelerates the helicopter until its forward motion is arrested. At this point (Figure 2.2c) the disc tilt and rotor moment vanish but the nose-up pitch attitude remains so that backward motion begins. This causes the rotor to tilt forward and exert a nose-down pitching moment (Figure 2.2b). Following this, a nose-down pitch attitude develops (Figure 2.2a) which accelerates the helicopter forward and returns it to the situation where the cycle begins again. A similar instability exists in the lateral (rolling) axis of the hovering helicopter.

Because the backflapping instability is characteristic of all hovering helicopters, flight control systems are an essential feature on helicopters. Flight control systems are necessary to make the helicopter easier to maneuver and by so doing alleviate the workload on a pilot.

The control systems for the helicopter are now discussed.

2.1.2 Blackhawk Flight Control Systems

Helicopter flight control systems are very complex dynamical systems. They offer a pilot a myriad of features (e.g. altitude-hold, vertical speed-hold, horizontal speed-hold, etc.). In this section we describe three control features which are typically found: (1) altitude-hold, (2) pitch-attitude hold, and (3) horizontal speed-hold. Each of these features may be implemented using classical unity feedback controllers. How this is done is described below.

Altitude-Hold Controller. An effective altitude hold control system may be built around the following simple proportional control law:

$$\Theta_c = k_v(z_c - z) \tag{2.6}$$

where z_c denotes the commanded altitude - measured in feet - and $k_v \geq 0$ is a control system design parameter referred to as the vertical altitude proportional gain constant. This controller structure can be used to stabilize the marginally stable open loop vertical dynamics (see Equation 2.1) and

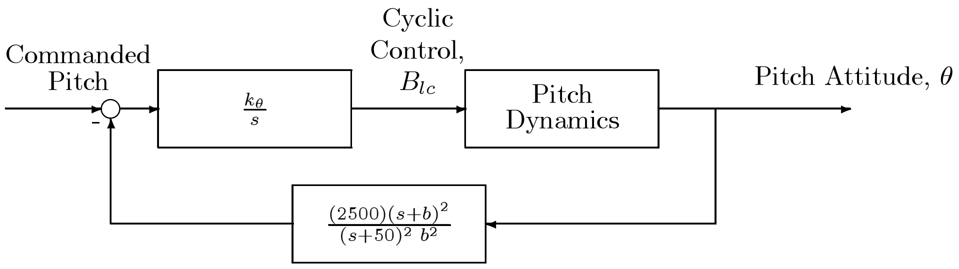


Figure 2.3: Single Helicopter Horizontal Pitch-Hold Controller

achieve desirable transient and steady state altitude command following characteristics.

Pitch-Attitude-Hold Controller. An effective pitch-attitude-hold control system may be built using the structure shown in Figure 2.4. The controller equation is given by:

$$B_{l_c} = \left[\frac{k_z}{s}\right] \left(\theta_c - \left[\frac{50^2(s+b)^2}{(s+50)^2\, b^2}\right] \theta\right) \tag{2.7}$$

where θ_c denotes the commanded pitch attitude (measured in radians), b is a controller design parameter, and $k_\theta \leq 0$ is a control system design parameter referred to as the pitch attitude proportional gain constant. A suitable controller was designed with $b = 2.0$, and $k_\theta = -1$. This stabilizing controller provides good low-frequency pitch attitude command following of step commands.

Horizontal Speed-Hold Controller. An effective horizontal speed-hold control system may be built using the structure shown in Figure 2.4. The controller equation is given by:

$$B_{l_c} = \left[\frac{k_h(s+a)}{s}\right] \left(\dot{x}_c - \left[\frac{50^2(s+b)^2}{(s+50)^2\, b^2}\right] \dot{x}\right) \tag{2.8}$$

where $\dot{x}_c$ denotes the commanded horizontal speed (measured in ft/sec), a and b are controller design parameters, and $k_h \geq 0$ is a control system design parameter referred to as the horizontal speed

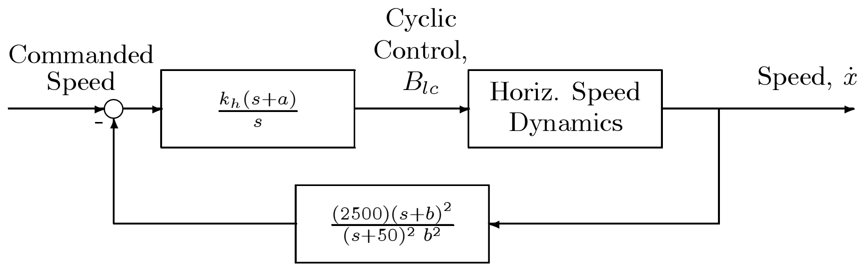


Figure 2.4: Single Helicopter Horizontal Speed-Hold Controller

proportional gain constant. A suitable controller was designed with $a = 2.5$, $b = 1$, and $k_h = 0.0005$. This stabilizing controller provides good speed response to speed step commands while producing acceptable levels of pitching in forward flight. In Chapter 3 we demonstrate how our environment may be used to analyze/visualize the performance associated with the above closed-loop system.

2.2 6-DOF AH-64 Apache Model and Control Laws

This section describes the models and control laws for a 6-degree-of-freedom McDonnell Douglas (now Boeing) AH-64 Apache helicopter gunship that are used in this thesis.

Two models for the AH-64 Apache gunship about hovering trim are presented in this section: An 8th-order rigid-body model (Model #1) [7], and a 10th-order model that includes balanced sensor and actuator dynamics (Model #2) [6].

The following control systems are presented in this thesis: An $\mathcal{H}^\infty$ Velocity Controller design (based on Model #1), and an Eigenstructure Assignment Velocity Controller with Loop Transfer Recovery (ESA/LTR) design (based on Model #2) [6].

2.2.1 8th-order Apache System Model and Dynamics

An 8-th order linear model for the AH-64 Apache helicopter system was obtained from [7]. This model is a linearization of the rigid-body dynamics taken about hovering trim.

The Apache's equations of motion (plant) take the form

$$\dot{x}_p = A_p x_p + B_p u_p \tag{2.9}$$

$$y_p = C_p x_p \tag{2.10}$$

where the control vector $u_p \in \mathcal{R}^{4\times 1}$, and state vector $x_p \in \mathcal{R}^{8\times 1}$ are given by:

$$u_p = \begin{bmatrix} \delta_{coll} & - & \text{Main rotor collective control} & \text{(deg)} \\ \delta_{lat} & - & \text{Lateral cyclic pitch control} & \text{(deg)} \\ \delta_{long} & - & \text{Longitudinal cyclic pitch control} & \text{(deg)} \\ \delta_{TR} & - & \text{Tail rotor collective control} & \text{(deg)} \end{bmatrix} \tag{2.11}$$

$$x_p = \begin{bmatrix} U, V, W & - & \text{Vehicle x,y,z velocity components} & \text{ft/sec} \\ P, Q, R & - & \text{Vehicle rotational velocities} & \text{rad/sec} \\ \theta, \phi & - & \text{Pitch and roll Euler angles} & \text{rad} \end{bmatrix} \tag{2.12}$$

The dimensional linear matrices are:

$$A_p = \begin{bmatrix} -0.0199 & -0.0058 & -0.0058 & -0.01259 & 0.01934 & 0.0005002 & 0 & -0.5545 \\ -0.0452 & -0.0526 & -0.0061 & -0.02168 & -0.01292 & 0.01234 & 0.5542 & -0.0002501 \\ -0.0788 & -0.0747 & -0.3803 & 0.0006669 & -0.004002 & 0.03501 & 0.01901 & 0.008503 \\ 0.5466 & -3.112 & -0.2144 & -2.998 & -0.5308 & 0.4155 & 0 & 0 \\ 0.4424 & 0.2316 & -0.2103 & 0.071 & -0.5943 & 0.0013 & 0 & 0 \\ 1.312 & 0.8769 & -0.04294 & 0.4058 & 0.4069 & -0.494 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0.0005 & -0.0154 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.9994 & 0.0343 & 0 & 0 \end{bmatrix} \tag{2.13}$$

$$B_p = \begin{bmatrix} 0.5262 & -0.01581 & -0.08446 & -0.001442 \\ 0.009557 & 0.5306 & -0.06835 & 0.3244 \\ 0.008446 & -0.006096 & -5.765 & 0.0001802 \\ 1.359 & 47.61 & -5.386 & 9.934 \\ -8.442 & 0.58 & -0.7121 & -0.07004 \\ 3.177 & -5.989 & 12.86 & -11.99 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.14}$$

$$C_p = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.15}$$

The eigenvalues and associated eigenvectors for the plant are given in Table 2.2. A pole-zero map of the plant is shown in Figure 2.5. A singular-value plot of the plant is shown in Figure 2.6. The eigenvalues of the plant show that the system is open-loop unstable, due to the characteristic backflapping modes of a helicopter. There is considerable cross-coupling between the longitudinal and lateral dynamics.

$\lambda_1 = -3.24$	$\lambda_2 = 0.0353 \pm j0.743$	$\lambda_3 = 0.211 \pm j0.53$
$x_1 = \begin{bmatrix} 0.00593 \\ 0.0577 \\ 0.00522 \\ 0.941 \\ -0.0308 \\ -0.156 \\ -0.291 \\ 0.0112 \end{bmatrix}$	$x_2 = \begin{bmatrix} 0.12 \pm j-0.0333 \\ 0.202 \pm j0.421 \\ -0.0443 \pm j0.00869 \\ -0.254 \pm j-0.39 \\ 0.102 \pm j-0.0365 \\ 0.31 \pm j-0.149 \\ -0.537 \pm j0.322 \\ -0.0486 \pm j-0.154 \end{bmatrix}$	$x_3 = \begin{bmatrix} 0.374 \pm j0.0591 \\ 0.0886 \pm j-0.163 \\ -0.0237 \pm j0.000181 \\ 0.042 \pm j0.115 \\ 0.167 \pm j-0.115 \\ 0.549 \pm j-0.505 \\ 0.221 \pm j0.0248 \\ -0.0947 \pm j-0.387 \end{bmatrix}$
$\lambda_4 = -0.902$	$\lambda_5 = -0.57$	$\lambda_6 = -0.322$
$x_4 = \begin{bmatrix} 0.324 \\ 0.0311 \\ 0.0878 \\ 0.00319 \\ -0.426 \\ -0.676 \\ -0.0148 \\ 0.498 \end{bmatrix}$	$x_5 = \begin{bmatrix} -0.134 \\ 0.06 \\ -0.193 \\ 0.0641 \\ 0.0417 \\ 0.954 \\ -0.0869 \\ -0.131 \end{bmatrix}$	$x_6 = \begin{bmatrix} 0.0735 \\ 0.0939 \\ 0.331 \\ 0.0326 \\ -0.043 \\ 0.932 \\ -0.0565 \\ 0.0342 \end{bmatrix}$

Table 2.2: 8th-Order Apache Model: Plant Eigenvalues and (dimensional) Eigenvectors

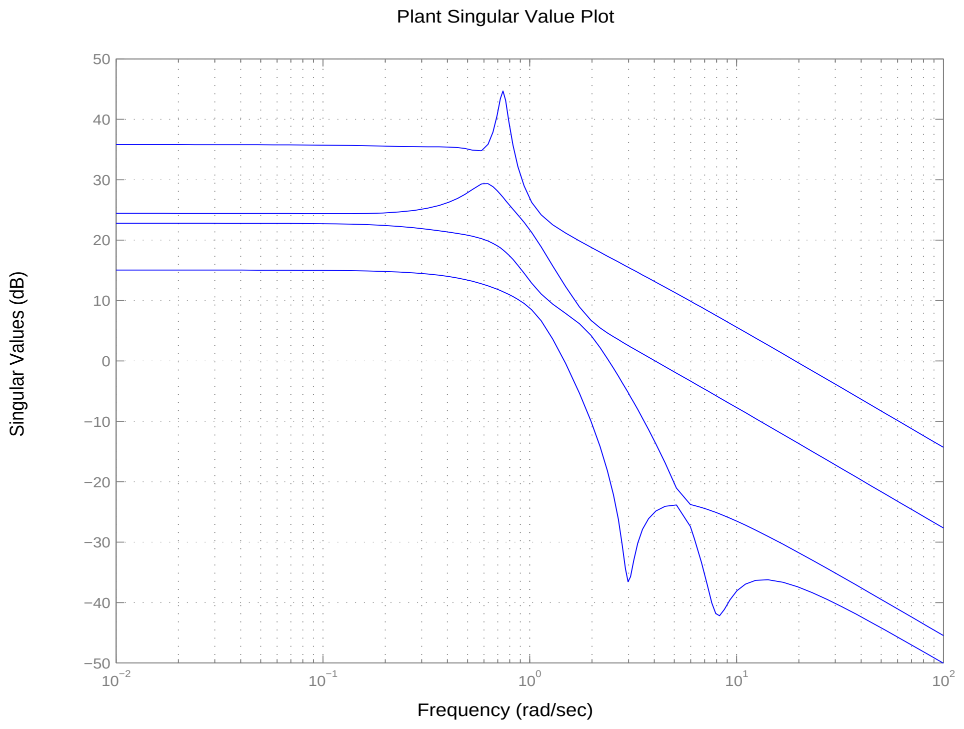


Figure 2.5: 8th-order Apache Model: Plant Pole-Zero Map

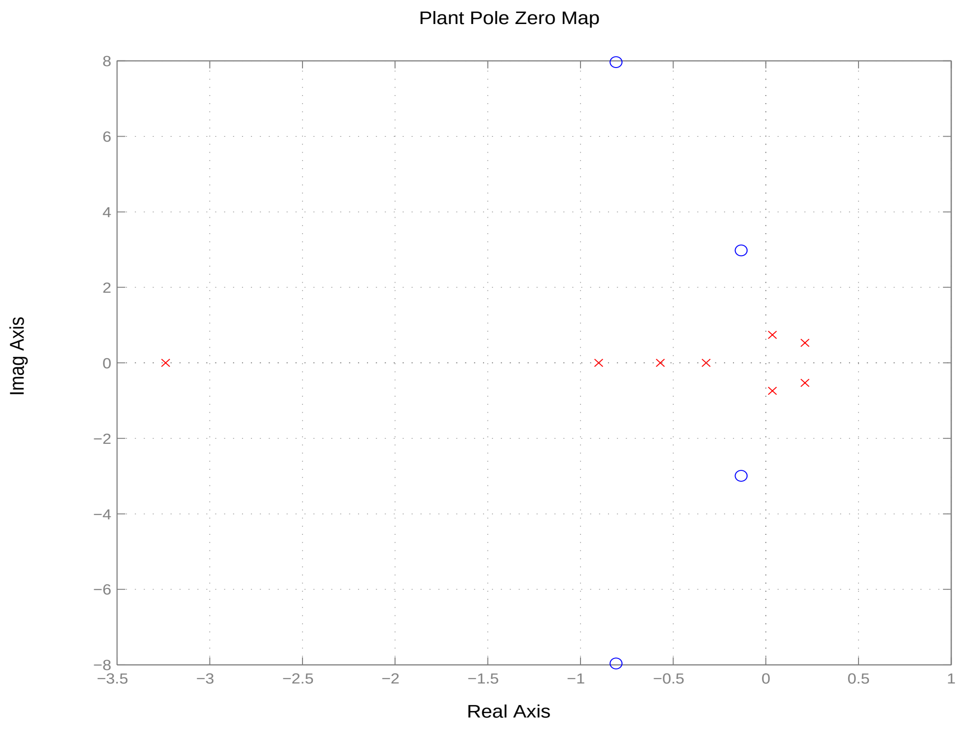


Figure 2.6: 8th-order Apache Model: Plant Singular Value Plot

A flight control system for the 8th-order Apache model is now discussed.

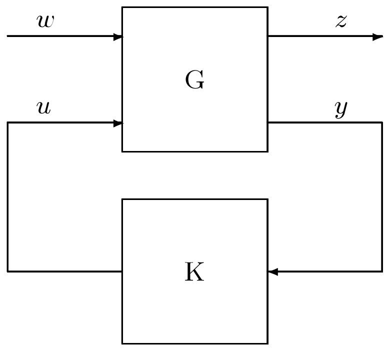


Figure 2.7: Control System Structure: General Design Framework

2.2.2 $\mathcal{H}^\infty$ Controller

Control System Structure: General Design Framework. The design of this control law is based on the structure indicated in Figure 2.7.

In this Figure, G is the augmented plant model to be controlled, K is the MIMO compensator (to be designed), w is an exogenous signal, z is the controlled signal, y is a signal to be processed by K and u is the control signal generated by K.

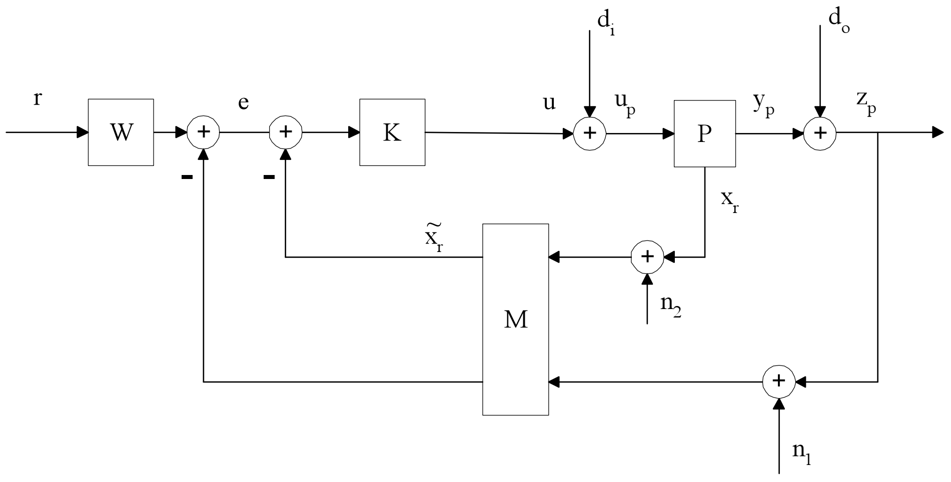


Figure 2.8: $\mathcal{H}^\infty$ Controller Implementation Structure

Control System Structure. The Apache speed-control system is based on the structure shown in Figure 2.8. In this figure, $z_p = [u\ v\ w\ r]^T$ are the commanded quantities, $x_r = [\theta\ \phi]^T$ are additional states that are fed back, r is the reference command signal, d_i is a plant input disturbance signal, d_o is a plant output disturbance signal, $n = [n_1 \; n_2]^T$ is a sensor noise signal, P is the plant, K is the MIMO compensator (to be designed), W is a static gain command mixing matrix used to ensure unity dc gains in each channel, and M is a mixing matrix used in the design process to tradeoff between performance bandwidth and the amount of pitching and rolling during longitudinal and lateral accelerations. M has the following structure:

$$M = \begin{bmatrix} \begin{bmatrix} -\alpha & 0 \\ 0 & -\beta \end{bmatrix} & O_{2\times 2} \\ O_{4\times 2} & I_{4\times 4} \end{bmatrix} \tag{2.16}$$

where α and β are design parameters.

The relationship between e, u, u_p, z_p (z) and the exogenous signals r, d_i, d_o, n (w) is as follows:

$$e = T_{re}r + T_{d_oe}d_o + T_{d_ie}d_i + T_{ne}n \tag{2.17}$$

$$u = T_{ru}r + T_{d_ou}d_o + T_{d_iu}d_i + T_{nu}n \tag{2.18}$$

$$u_p = T_{ru_p}r + T_{d_ou_p}d_o + T_{d_iu_p}d_i + T_{nu_p}n \tag{2.19}$$

$$z_p = T_{rz_p}r + T_{d_oz_p}d_o + T_{d_iz_p}d_i + T_{nz_p}n \tag{2.20}$$

Since (r, d_i, d_o) are typically low frequency signals and n is typically a high frequency signal, the above suggests that we typically desire the following:

- $T_{rz_p} \approx 1$ at low frequencies for good low frequency command following,
- $T_{d_oz_p}$, $T_{d_iz_p}$ small at low frequencies for good disturbance rejection,
- T_{nz_p} small at high frequencies for good high frequency noise attenuation.

Closed Loop Design Specifications. The design specifications for the closed loop system are as follows:

- the nominal closed loop system is stable;
- the nominal closed loop system exhibits zero steady state error to step reference commands r, step output disturbances d_o, and step input disturbances d_i;

- the maximum T_{re} singular values lie below 2 dB for all frequencies above $\omega = 1$ rad/sec. ($\sigma_{max}\left[\, T_{re}(jw)\,\right] < 1.26$ (2 dB) for all $\omega \geq 1$ rad/sec);
- $\sigma_{max}\left[\, T_{rz_p}(jw)\,\right] < 1.26$ (2 dB) for all $\omega \geq 1$ rad/sec;
- the nominal closed loop system is robust with respect to a multiplicative uncertainty $\Delta = \omega I$ at the plant output; i.e.

$$\sigma_{max}\left[\, T(jw)\,\right] < \frac{1}{\sigma_{max}\ \left[\, \Delta(j\omega)\,\right]} = \frac{1}{\omega} \tag{2.21}$$

for all $\omega \geq 5$ rad/sec;

Design Framework. The design of the Apache speed-control law is based on the structure shown in Figure 2.9. In this figure, K is the MIMO compensator (to be designed), Pd is the design plant, r is the reference command signal, and W_{re}, W_{ru}, and W_{rz_p} are design weights (to be chosen).

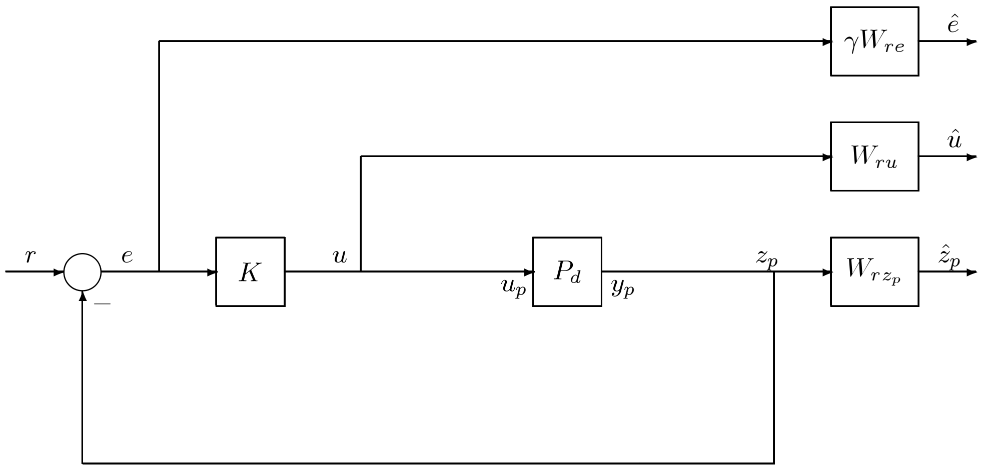


Figure 2.9: Control System Structure: System Specific Design Framework

Relating this system-specific structure to the general structure shown in Figure 2.7, we have:

$$w = \left[\ r \ - \ \text{reference command vector}\ \right] \tag{2.22}$$

$$z = \begin{bmatrix} \hat{e} & - & \text{weighted error vector} \\ \hat{u}_p & - & \text{weighted control signal vector} \\ \hat{z}_p & - & \text{weighted output vector} \end{bmatrix} \tag{2.23}$$

$$y = \begin{bmatrix} r & - & \text{reference vector} \\ 0_{6\times 1} & - & \text{zero vector} \end{bmatrix} - x \tag{2.24}$$

$$= \begin{bmatrix} e & - & \text{error vector} \\ -x_r & - & \text{rest of states vector} \end{bmatrix} \tag{2.25}$$

$$u = \begin{bmatrix} u & - & \text{controller output vector} \end{bmatrix} \tag{2.26}$$

The Design Plant. This controller design is based on full-state feedback. As such, it is assumed that all the states may be measured and/or estimated. The design plant is specified as:

$$P_d = C_d(sI - A_d)^{-1}B_d \tag{2.27}$$

where

$$A_d = \begin{bmatrix} -0.0199 & -0.0058 & -0.0058 & -0.01259 & 0.01934 & 0.0005002 & 0 & -0.5545 \\ -0.0452 & -0.0526 & -0.0061 & -0.02168 & -0.01292 & 0.01234 & 0.5542 & -0.0002501 \\ -0.0788 & -0.0747 & -0.3803 & 0.0006669 & -0.004002 & 0.03501 & 0.01901 & 0.008503 \\ 0.5466 & -3.112 & -0.2144 & -2.998 & -0.5308 & 0.4155 & 0 & 0 \\ 0.4424 & 0.2316 & -0.2103 & 0.071 & -0.5943 & 0.0013 & 0 & 0 \\ 1.312 & 0.8769 & -0.04294 & 0.4058 & 0.4069 & -0.494 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0.0005 & -0.0154 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.9994 & 0.0343 & 0 & 0 \end{bmatrix} \tag{2.28}$$

$$B_d = \begin{bmatrix} 0.5262 & -0.01581 & -0.08446 & -0.001442 \\ 0.009557 & 0.5306 & -0.06835 & 0.3244 \\ 0.008446 & -0.006096 & -5.765 & 0.0001802 \\ 1.359 & 47.61 & -5.386 & 9.934 \\ -8.442 & 0.58 & -0.7121 & -0.07004 \\ 3.177 & -5.989 & 12.86 & -11.99 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.29}$$

$$C_d = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & \alpha \\ 0 & 1 & 0 & 0 & 0 & 0 & \beta & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.30}$$

$$\alpha = -1.3 \tag{2.31}$$

$$\beta = 1.4 \tag{2.32}$$

C_d is a transformed version of M, a mixing matrix used to place weights on the pitch and roll frequency responses. This is used to trade-off between performance bandwidth and the amount of

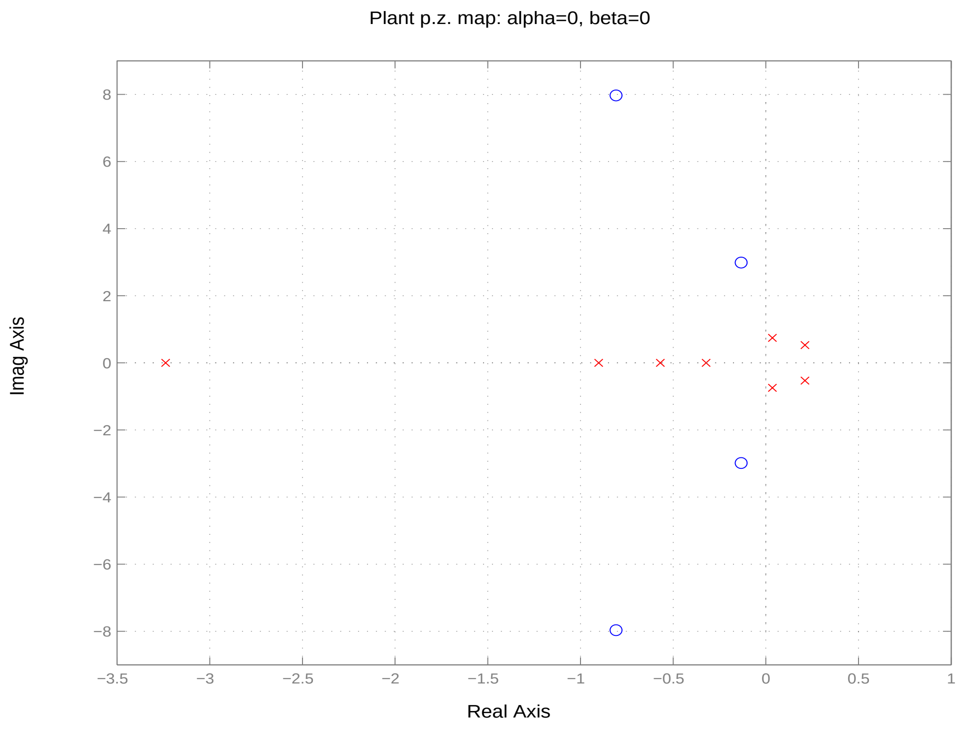


Figure 2.10: $\mathcal{H}^{\infty}$ Controller: Effect of Mixing Matrix: $\alpha = 0, \beta = 0$

pith and roll experienced during longitudinal and lateral acceleration. This mixing matrix also helps place the closed-loop poles at acceptable locations with large (>0.7) damping ratios. It does this by moving the transmission zeros of the design plant. For $\alpha = 0, \beta = 0$, the transmission zeros of the design plant are at very lightly damped locations (ζ_{min}= 0.045) (Figure 2.10). These lightly-damped open-loop transmission zeros result in lightly-damped closed-loop poles, which manifests itself as a large amount of pitch and roll action during longitudinal and lateral accelerations. Decreasing α and increasing β cause these transmission zeros to move to more highly damped locations (Figures 2.11 - 2.12). α is negative because of the negative-relationship between pitch and forward acceleration. For $\alpha = -1.3, \beta = 1.4$, the closed-loop transmission zeros occur at overdamped locations ($\zeta_{min} = 1$) as shown in Figure 2.13.

The $\mathcal{H}^{\infty}$ design methodology was then applied to this design plant in order to obtain the controller K. This procedure is described below.

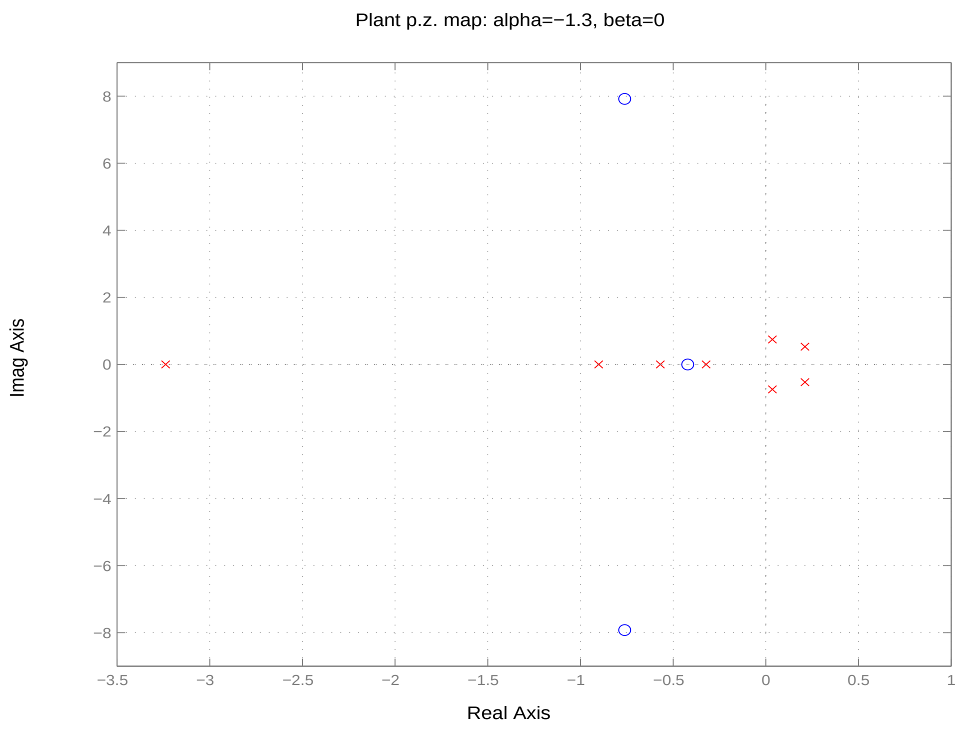


Figure 2.11: $\mathcal{H}^{\infty}$ Controller: Effect of Mixing Matrix: $\alpha = -1.3, \beta = 0$

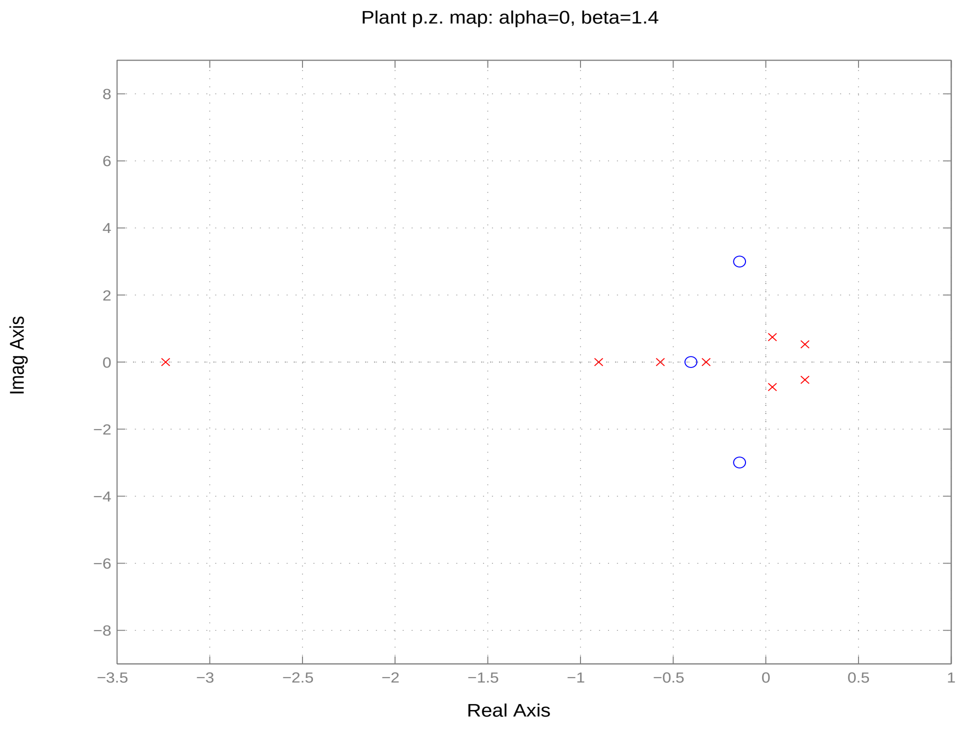


Figure 2.12: $\mathcal{H}^{\infty}$ Controller: Effect of Mixing Matrix: $\alpha = 0, \beta = 1.4$

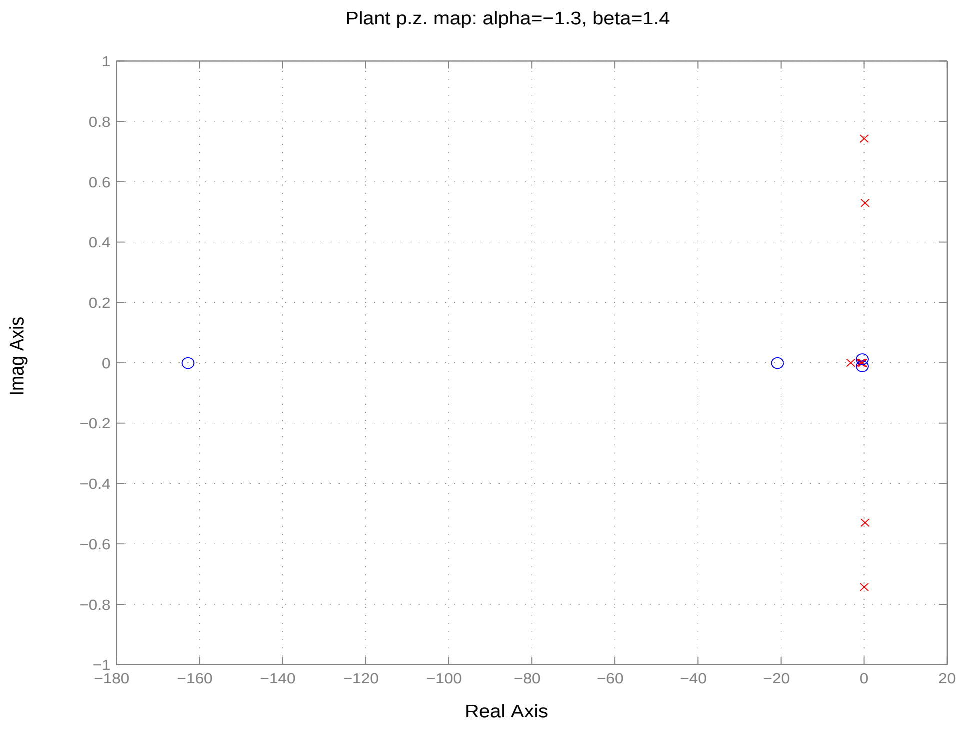


Figure 2.13: $\mathcal{H}^{\infty}$ Controller: Effect of Mixing Matrix: $\alpha = -1.3, \beta = 1.4$

$\mathcal{H}^{\infty}$ Design Methodology. The design methodology for an H^{∞} based controller is now described. The H^{∞} controller is obtained by solving the following nonlinear optimization problem:

$$\|T_{rz}\|_{H^{\infty}} = \left\| \begin{bmatrix} \gamma W_{re} \; T_{re} \\ W_{ru} \; T_{ru} \\ W_{rz_p} T_{rz_p} \end{bmatrix} \right\|_{H^{\infty}} \leq 1 \tag{2.33}$$

where W_{re}, W_{ru}, W_{rz_p} and γ are selected by the designer to obtain desired closed loop properties. For the helicopter speed-control problem, these were selected as follows:

$$W_{re} = diag(W_{re1}, W_{re2}, ..., W_{re4}) \tag{2.34}$$

$$W_{ru} = diag(W_{ru1}, W_{ru2}, ..., W_{ru4}) \tag{2.35}$$

$$W_{rz_p} = diag(W_{rz_p1}, W_{rz_p2}, ..., W_{rz_p4}) \tag{2.36}$$

where the weights for the individual channels are specified in Tables 2.3-2.5.

The above transfer functions and weights under the H^{∞} norm are mapped from the s domain to the $\tilde{s}$ domain by means of the bilinear transform and the transform parameters $p_1 = -0.4$ and

$W_{re1} = \frac{s+1}{3.5s+3.5e-005}$
$W_{re2} = \frac{s+1}{3.5s+3.5e-005}$
$W_{re3} = \frac{s+1}{1.5s+1.5e-005}$
$W_{re4} = \frac{s+3}{1.5s+0.0015}$

Table 2.3: $\mathcal{H}^\infty$ Design: Weighting Functions, W_{re}

$W_{ru1} = \frac{1}{10}$
$W_{ru2} = \frac{1}{10}$
$W_{ru3} = \frac{1}{10}$
$W_{ru4} = \frac{1}{10}$

Table 2.4: $\mathcal{H}^\infty$ Design: Weighting Functions, W_{ru}

$W_{rz_p1} = \frac{33.33s+1000}{3s+3000}$
$W_{rz_p2} = \frac{43.49s+1000}{2.3s+2300}$
$W_{rz_p3} = \frac{76.92s+1000}{1.3s+1300}$
$W_{rz_p4} = \frac{133.3s+1000}{1.5s+1500}$

Table 2.5: $\mathcal{H}^\infty$ Design: Weighting Functions, W_{rz_p}

$p_2 = -100$. The actual H^∞ procedure is done in the $\tilde{s}$ domain. The resulting controller $\tilde{K}(\tilde{s})$ is mapped back to s using the inverse bilinear transform to obtain the final controller $K(s)$.

When the augmented plant (plant and weighting functions) has poles or zeros on the imaginary axis, the H^∞ algorithm cannot be reliably computed. In such case, the controller would have closed-loop poles very near the imaginary axis. Using a bilinear transformation can be very helpful in this case. Moreover, the bilinear transformation provides a way to control the location of the dominant closed-loop poles, hence allowing direct control over closed-loop system performance characteristics such as settling time, damping ratio, rise time, overshoot, etc.

The bilinear transform is the following jw-axis pole shifting transformation:

$$s = \frac{\tilde{\mathrm{s}} + \mathrm{p}_1}{\frac{\tilde{\mathrm{s}}}{\mathrm{p}_2} + 1} \tag{2.37}$$

-p_1 and -p_2 represent points that are located at the end of the diameter of a circle in the left side of the s-plane. Such an s-plane is mapped onto the j$\tilde{w}$-axis in the $\tilde{s}$-plane. The following is the inverse bilinear transform:

$$\tilde{s} = \frac{-\mathrm{s} + \mathrm{p}_1}{\frac{\mathrm{s}}{\mathrm{p}_2} - 1} \tag{2.38}$$

Prior to computing the H^∞ algorithm, the bilinear transformation was applied to the augmented plant. Afterwards, the inverse bilinear transform was used on the resulting H^∞ controller.

The *MATLAB Robust Control Toolbox* provides a *"bilin"* command for computing a bilinear transformation [100]. For more information on this refer to [1].

The resulting controller has the form:

$$K(s) = C_c(sI - A_c)^{-1}B_c + D_c \tag{2.39}$$

where

$$A_c = [A_{c1}\ A_{c2}] \tag{2.40}$$

$$A_{c1} = \begin{bmatrix} -4.398 & -0.4573 & -3.569 & 2.165 & -0.5422 & -0.6663 & 5.8 & -1.688 \\ -0.4774 & -6.502 & 0.7177 & -6.828 & 6.412 & -6.039 & 34.46 & -1.014 \\ -0.9098 & 0.1399 & -1.365 & 0.8554 & -0.3797 & -0.3637 & 0.1293 & -0.2406 \\ 0.1285 & -1.691 & 0.6364 & -2.612 & 1.835 & -0.8088 & 9.014 & -0.383 \\ 0.1848 & 1.121 & -0.2561 & 1.343 & -1.543 & 0.2063 & -6.065 & 0.4506 \\ 0.02537 & -0.1728 & 0.04063 & -0.1958 & 0.1748 & -0.9087 & 0.9942 & -0.0787 \\ -0.02284 & 0.6664 & -0.03619 & 0.7189 & -0.7441 & 1.062 & -4.297 & 0.06023 \\ 0.002671 & -0.02177 & 0.002825 & -0.0169 & 0.01887 & -0.1375 & 0.1344 & -0.4327 \\ -0.03301 & -0.07742 & 0.05866 & -0.2353 & 0.2491 & 2.701 & 0.1438 & 0.4784 \\ 0.2148 & 0.09675 & -0.1399 & -0.01684 & 0.0103 & 0.1173 & -0.3792 & 0.4435 \\ 0.04822 & -0.2199 & 0.08775 & 0.0264 & 0.4236 & -0.2184 & 1.208 & -0.5611 \\ 0.03688 & 0.03588 & -0.05907 & -0.009068 & -0.000786 & 0.04112 & -0.21 & -2.176 \\ -0.001366 & 0.001245 & -0.0005951 & 0.001109 & -0.002198 & 0.008863 & -0.005571 & -0.000457 \\ -0.001101 & 0.0001394 & 0.00142 & 0.002823 & -0.002221 & -0.0276 & -0.001495 & 0.003435 \\ -0.0008048 & 0.001136 & 0.001134 & 0.003936 & -0.003249 & -0.03057 & -0.007189 & -0.004031 \\ 0.006958 & 0.005365 & -0.01118 & -0.001761 & 0.002189 & 0.01029 & -0.03076 & -0.3798 \end{bmatrix} \tag{2.41}$$

$$A_{c2} = \begin{bmatrix} 5.852 & -120.1 & -34.63 & -15.95 & -60.73 & -40.25 & 29.76 & -13.94 \\ -2.39 & -51.01 & 210.9 & -65.71 & 53.01 & -159.6 & 49.77 & -42.32 \\ 4.287 & -29.93 & -17.77 & 1.932 & -18.71 & 1.374 & 13.12 & -4.737 \\ -6.379 & 1.171 & 64.03 & -23.09 & 23.56 & -50.43 & -3.353 & -4.999 \\ 5.973 & 4.025 & -39.9 & 18.37 & -13.07 & 38.59 & 6.705 & 1.566 \\ 2.009 & -0.4222 & 6.654 & -1.965 & 1.766 & 0.326 & 7.521 & -0.8763 \\ -2.177 & 7.62 & -20.02 & 4.918 & -3.541 & 11.62 & -11.83 & 5.824 \\ 0.7179 & -0.01168 & 0.7483 & -0.3485 & 0.123 & 1.153 & 2.31 & 0.04865 \\ -17.85 & -2.615 & -3.617 & 1.171 & 0.8666 & -38.26 & -52.03 & -1.645 \\ -0.5261 & -8.456 & 2.555 & 4.422 & -1.796 & -8.07 & -1.251 & -7.116 \\ 0.1736 & 1.581 & -15.4 & -3.827 & -4.599 & 10.05 & -2.171 & 9.23 \\ -0.1827 & -0.5335 & -0.3008 & -30.71 & 0.08312 & 0.269 & 1.967 & 25.03 \\ -0.03009 & -0.02887 & -0.1153 & -0.009555 & -90.05 & -3.691 & -4.781 & 0.1537 \\ 0.1127 & 0.02098 & 0.05104 & 0.09986 & -0.5072 & -74.5 & 19.45 & -0.466 \\ 0.1263 & 0.03589 & 0.01037 & 0.08497 & -0.6136 & 18.17 & -68.02 & 0.1443 \\ -0.05012 & -0.115 & -0.09726 & -3.956 & 0.01183 & -0.1841 & -0.07052 & -52.11 \end{bmatrix} \tag{2.42}$$

$$B_c = \begin{bmatrix} 0.7284 & -0.1977 & -0.05881 & -0.3041 \\ 0.1951 & -4.431 & -0.33 & -1.167 \\ 1.605 & -0.9954 & -0.02311 & 0.07217 \\ 0.1445 & -3.118 & -0.3762 & -0.4901 \\ -1.056 & -0.168 & -0.3607 & 0.4147 \\ -0.03178 & -0.1693 & -6.601 & -0.02911 \\ 0.2416 & 0.6763 & -1.121 & -0.05215 \\ -0.007816 & -0.03869 & 0.1509 & -11.07 \\ 0.07593 & 0.08625 & -3.264 & -0.8097 \\ -0.7224 & 0.5427 & -0.09207 & -1.041 \\ -0.2314 & -1.494 & -0.05888 & 1.374 \\ -0.1979 & 0.2285 & -0.03974 & 5.287 \\ 0.002856 & 0.004545 & -0.03045 & 0.003678 \\ 0.00617 & -0.008653 & 0.1266 & -0.01346 \\ 0.005329 & -0.007726 & 0.1451 & 0.007445 \\ -0.03915 & 0.03649 & -0.02771 & 1.08 \end{bmatrix} \tag{2.43}$$

$$C_c = [C_{c1}\ C_{c2}] \tag{2.44}$$

$$C_{c1} = \begin{bmatrix} -0.4075 & -0.02067 & -0.4295 & 0.3394 & -0.1515 & 0.1284 & 0.2515 & -0.1964 \\ 0.03843 & 0.1299 & 0.02073 & 0.1738 & -0.1723 & 0.207 & -1.132 & 0.08776 \\ -0.002867 & 0.029 & -0.001516 & 0.02366 & -0.02124 & -0.2378 & -0.04764 & -0.01636 \\ -0.1311 & -0.1233 & -0.1761 & 0.09732 & -0.05929 & -0.2884 & 0.5496 & -0.2501 \end{bmatrix} \tag{2.45}$$

$$C_{c2} = \begin{bmatrix} -0.1208 & -13.32 & -6.688 & -1.541 & -7.596 & -4.489 & 0.4548 & -0.7179 \\ -0.0806 & 2.547 & -6.423 & 2.795 & -1.176 & 5.088 & -2.241 & 0.9591 \\ 1.573 & 0.335 & 0.0555 & 0.3627 & -0.1245 & 3.658 & 4.572 & -0.1055 \\ 1.618 & -4.3 & 1.453 & -3.459 & -1.559 & 0.2742 & 6.243 & 0.9643 \end{bmatrix} \tag{2.46}$$

$$D_c = \begin{bmatrix} 0.09215 & 0.02809 & 0.004504 & -0.03374 \\ -0.01447 & 0.1459 & 0.012 & 0.05939 \\ -0.001111 & 0.0007313 & -0.01215 & 0.003386 \\ 0.03301 & -0.0673 & -0.01699 & -0.1029 \end{bmatrix} \tag{2.47}$$

A MATLAB macro used to generate this design is included in Appendix A.

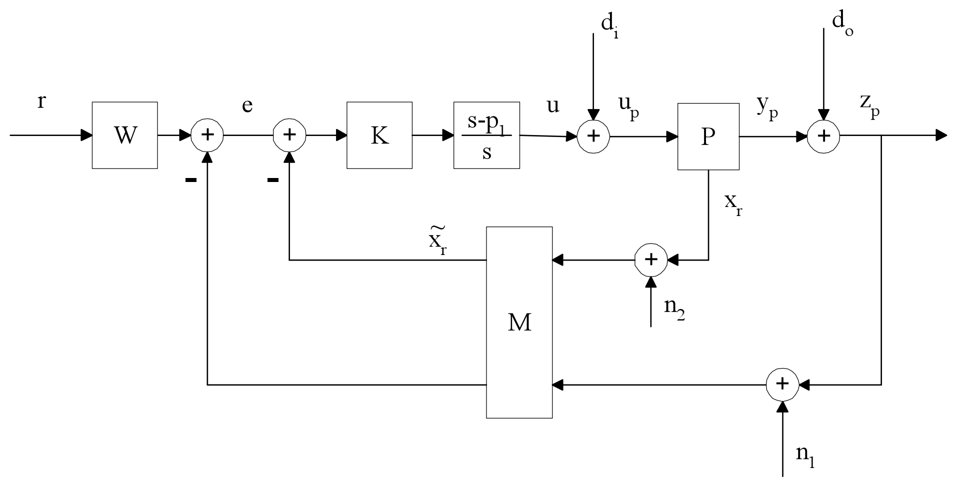


Figure 2.14: $\mathcal{H}^\infty$ Controller Implementation

This controller is implemented as shown in Figure 2.14. Due to the nature of the bilinear transform, integrators that are introduced by the $\mathcal{H}^\infty$ algorithm in the $\hat{s}$ plane are shifted to $s = p_1$ in the s plane. The resulting controller, K, is augmented with the transfer function $\frac{s-p_1}{s}$ in each output channel in order to shift these integrators back to the origin. This procedure fulfills the design specification of zero steady state error to step commands, and the rejection of step plant-input and plant-output disturbances.

W is a command mixing filter (a static gain matrix) that is designed to eliminate steady state error at d.c.. This matrix is obtained by inverting the d.c. gain matrix of the closed-loop system from the reference command to the commanded quantities, $z_p = [u, v, w, r]$. This matrix was computed to be

$$W = \begin{bmatrix} 0.9817 & -0.02619 & 0.02249 & -0.001104 \\ 0.06042 & 1.024 & 0.05457 & -0.001749 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.48}$$

Analysis of the $\mathcal{H}^\infty$ Controller. The closed-loop poles of the system with the $\mathcal{H}^\infty$ controller are listed in Table 6.1. The smallest damping ratio of the closed-loop poles is $\zeta = 0.73$.

Figure 2.15 shows the closed-loop singular value plot for the developed $\mathcal{H}^\infty$ control system. This is the singular value plot from reference commands to commanded quantities, $z_p = [u, v, w, r]$.

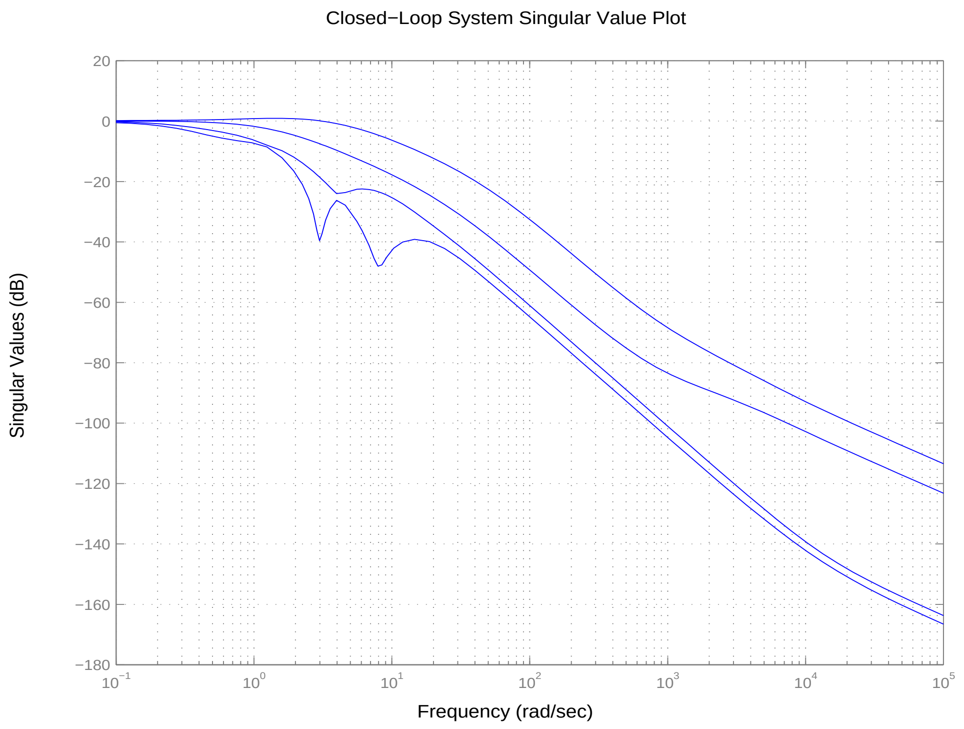


Figure 2.15: $\mathcal{H}^\infty$ Controller: Closed-Loop System SV Plot

This plot shows that the closed-loop system has excellent low-frequency command following with a maximum bandwidth of approximately 10 rad/sec. The pole-zero map for the closed-loop system is shown in Figure 2.16.

Figure 2.17 shows the singular value plot from reference commands to tracking error. Again, this plot shows that low-frequency command following is very good, with no steady-state errors to step-reference commands.

Figure 2.18 shows the singular value plot from plant input disturbances to the outputs $z_p = [u, v, w, r]$. This plot shows that step disturbances are attenuated, resulting in negligible steady-state errors.

Figure 2.19 shows the singular value plot from sensor noise to the outputs $z_p = [u, v, w, r]$. This plot shows that high frequency sensor noise is attenuated.

The frequency-response plots show that the design meets all of the design specifications.

The response of the closed-loop system to step reference commands in each channel (i.e. forward-

$\lambda_i(T_{rz_p})$	Damping Ratio (ζ)	Natural Frequency (ω_n) (rad/sec)
$\lambda_1 = -90.2$	1.0000	90.1610
$\lambda_2 = -90.4$	1.0000	90.3717
$\lambda_3 = -48.7$	1.0000	48.6569
$\lambda_4 = -51.9$	1.0000	51.9238
$\lambda_5 = -29.9$	1.0000	29.8513
$\lambda_6 = -17.5$	1.0000	17.5273
$\lambda_{7,8} = -13.2 \pm j8.43$	0.8434	15.6977
$\lambda_{9,10} = -5.56 \pm j4.27$	0.7930	7.0106
$\lambda_{11} = -4.1$	1.0000	4.0958
$\lambda_{12} = -3.24$	1.0000	3.2378
$\lambda_{13,14} = -1.2 \pm j1.05$	0.7527	1.5961
$\lambda_{15,16} = -1.01 \pm j0.931$	0.7348	1.3720
$\lambda_{17} = -1.23$	1.0000	1.2314
$\lambda_{18} = -0.902$	1.0000	0.9021
$\lambda_{19} = -0.744$	1.0000	0.7444
$\lambda_{20} = -0.57$	1.0000	0.5695
$\lambda_{21,22} = -0.48 \pm j0.0263$	0.9985	0.4804
$\lambda_{23,24} = -0.414 \pm j0.0114$	0.9996	0.4144
$\lambda_{25,26} = -0.4 \pm j2.75e-005$	1.0000	0.4000
$\lambda_{27} = -0.4$	1.0000	0.4000
$\lambda_{28} = -0.4$	1.0000	0.4000

Table 2.6: $\mathcal{H}^\infty$ Controller: Closed-Loop Poles

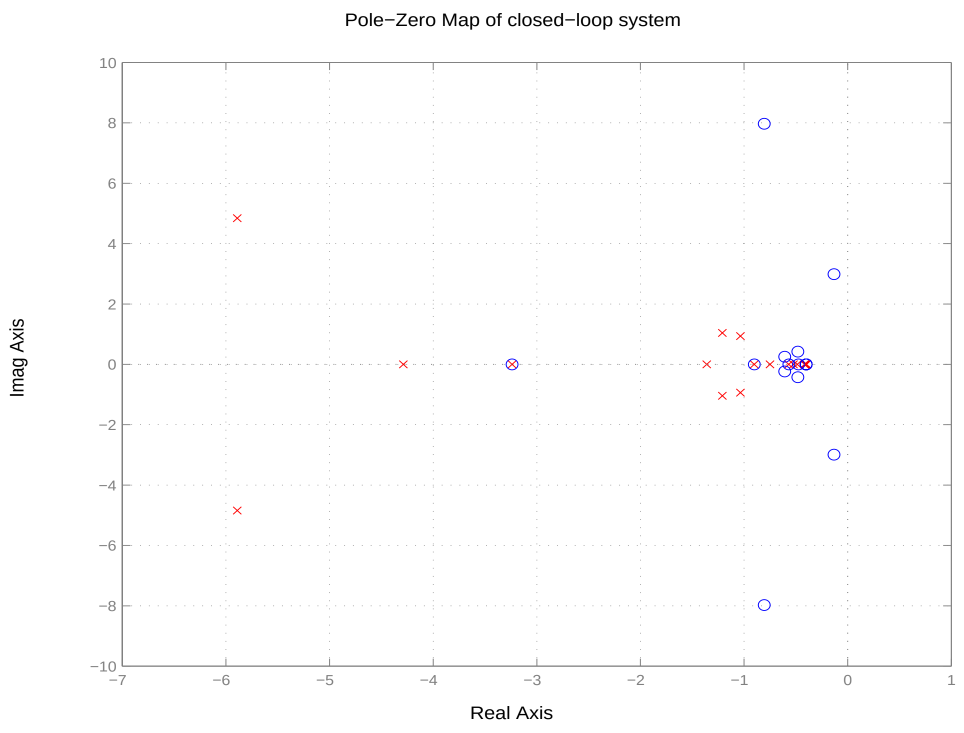


Figure 2.16: $\mathcal{H}^{\infty}$ Controller: Closed-Loop Pole-Zero Map

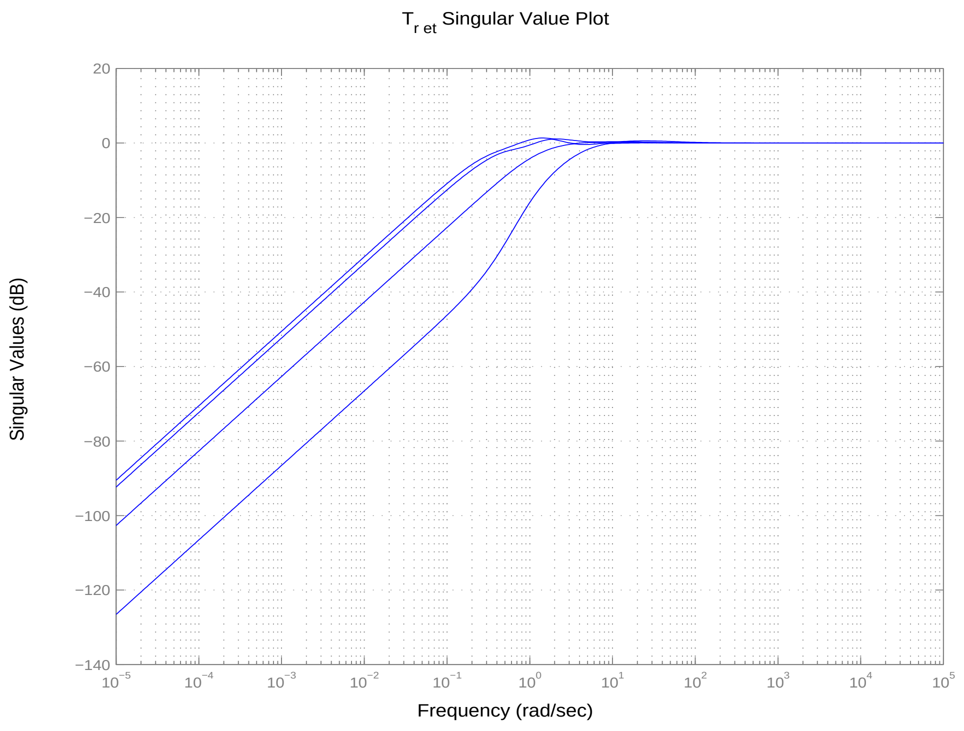


Figure 2.17: $\mathcal{H}^{\infty}$ Controller: Reference to Tracking Error SV Plot

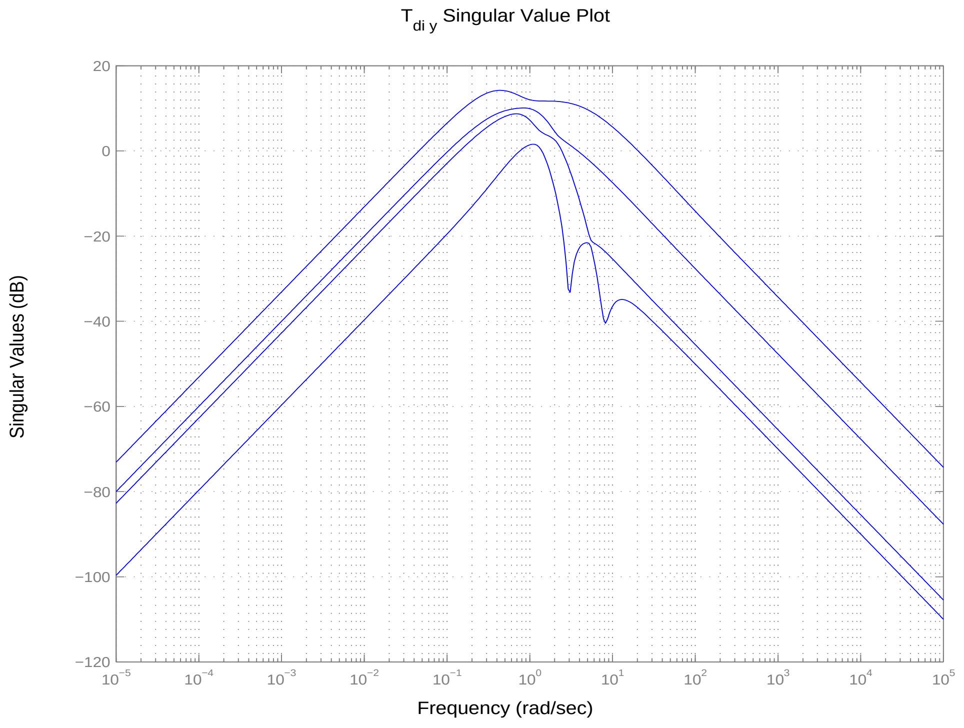


Figure 2.18: $\mathcal{H}^{\infty}$ Controller: Input Disturbance to Output SV Plot

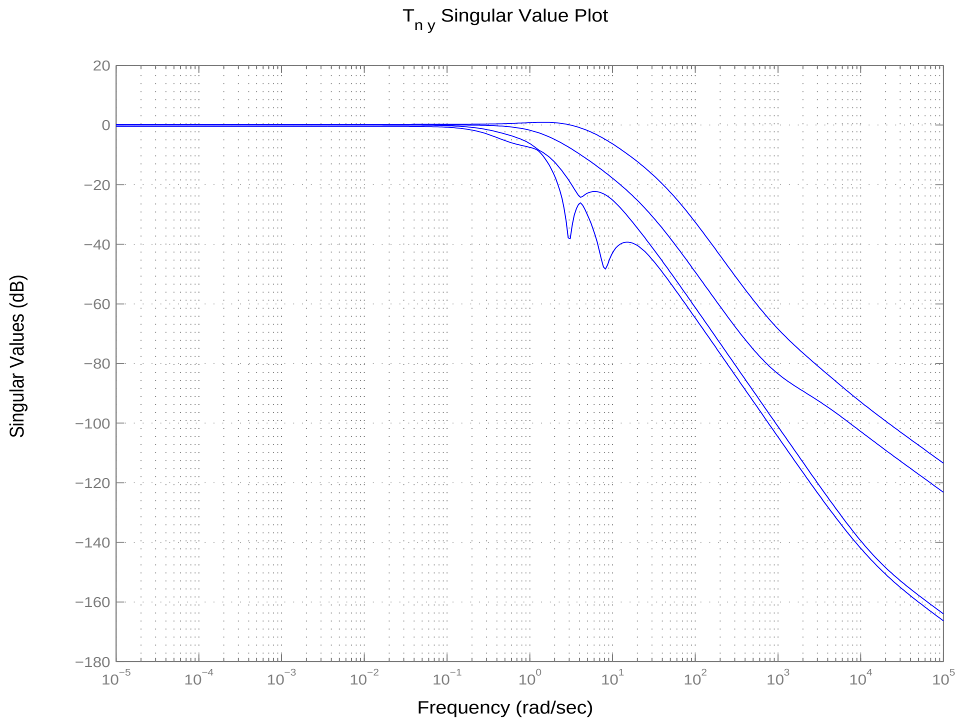


Figure 2.19: $\mathcal{H}^{\infty}$ Controller: Sensor Noise to Output SV Plot

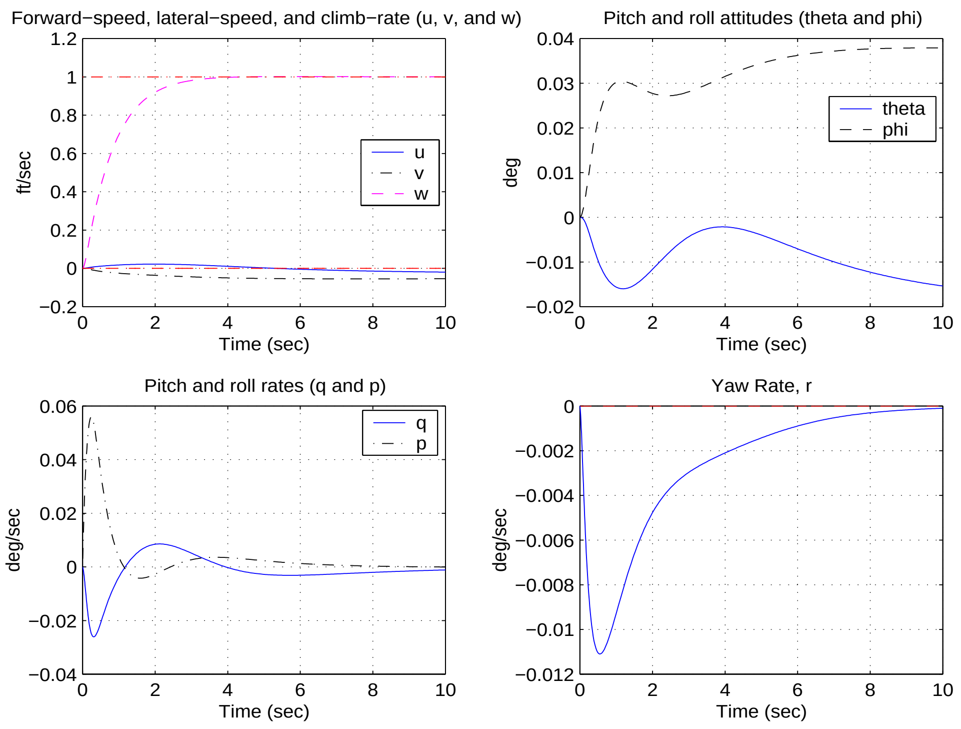


Figure 2.20: $\mathcal{H}^{\infty}$ Controller: Forward-Speed (u) Step Command Response: Outputs

speed, lateral-speed, climb-rate, and yaw-rate) are shown in Figures 2.20 - 2.27.

2.2.3 10th-order Apache System Model and Dynamics

A 10-th order linear model for this system was obtained from [6]. This model was obtained by reducing a 28th-order model consisting of linearized rigid-body dynamics augmented with sensor and actuator dynamics. The rigid body dynamics were obtained by linearizing the nonlinear equations about the hover trim condition.

The plant takes the form

$$\dot{x}_p = A_p x_p + B_p u_p \tag{2.49}$$

$$y_p = C_p x_p + D_p u_p \tag{2.50}$$

where the control vector $u_p \in \mathcal{R}^{4\times1}$, and state vector $x_p \in \mathcal{R}^{8\times1}$ are given by:

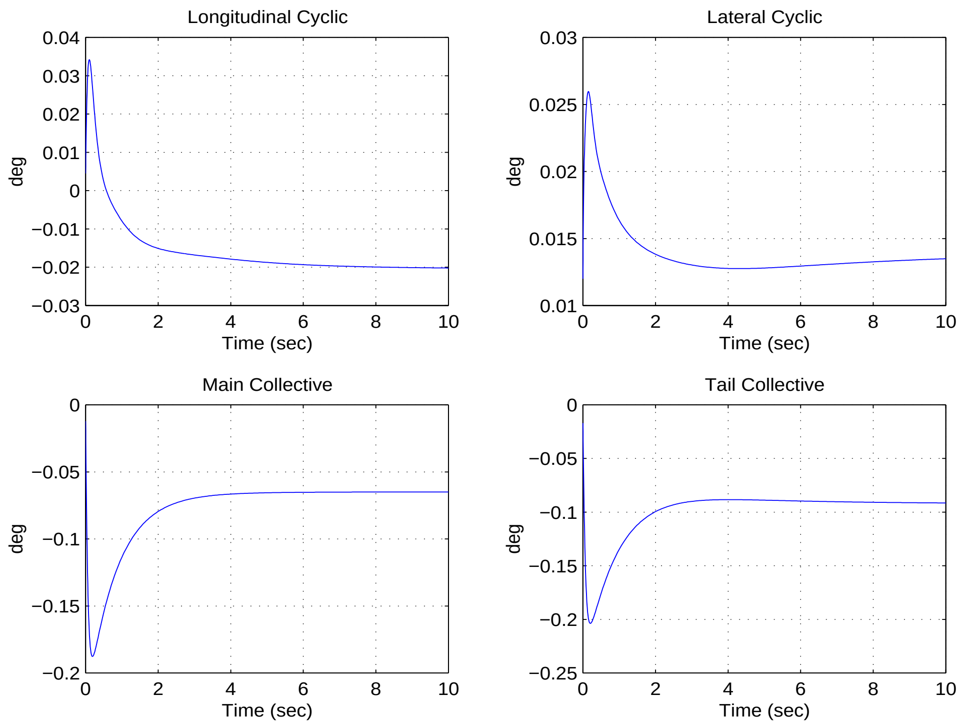


Figure 2.21: $\mathcal{H}^{\infty}$ Controller: Forward-Speed (u) Step Command Response: Controls

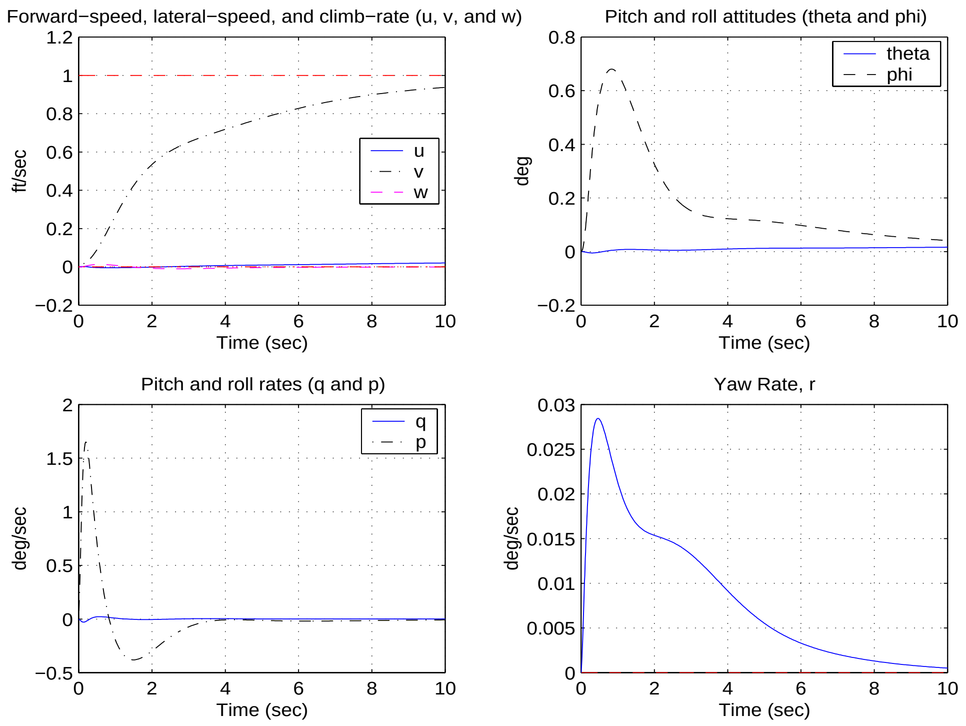


Figure 2.22: $\mathcal{H}^{\infty}$ Controller: Lateral-Speed (v) Step Command Response: Outputs

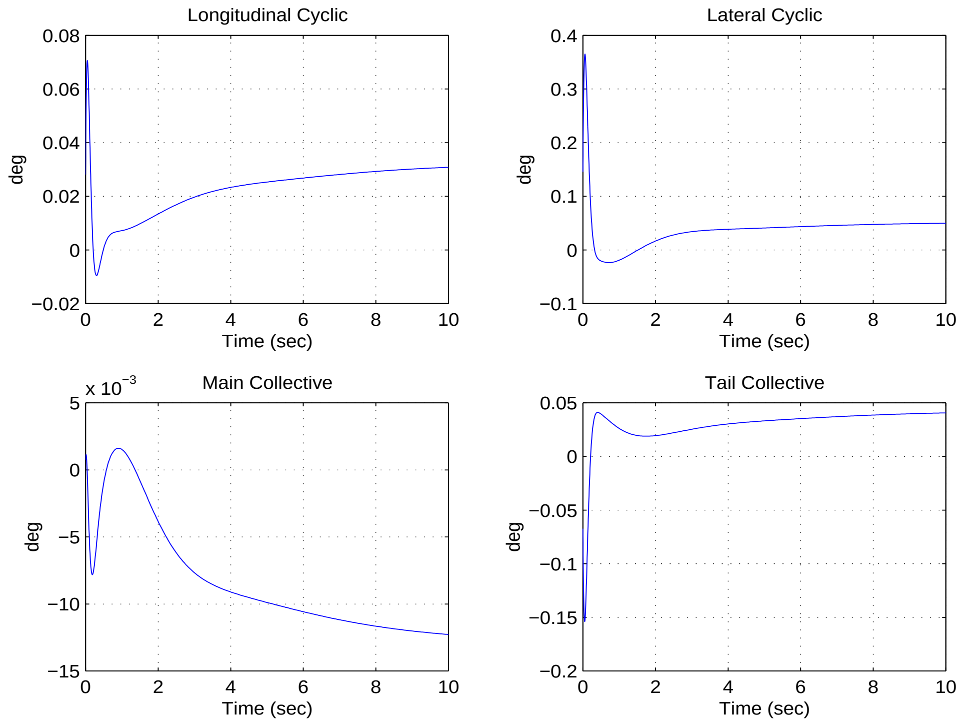


Figure 2.23: $\mathcal{H}^\infty$ Controller: Lateral-Speed (v) Step Command Response: Controls

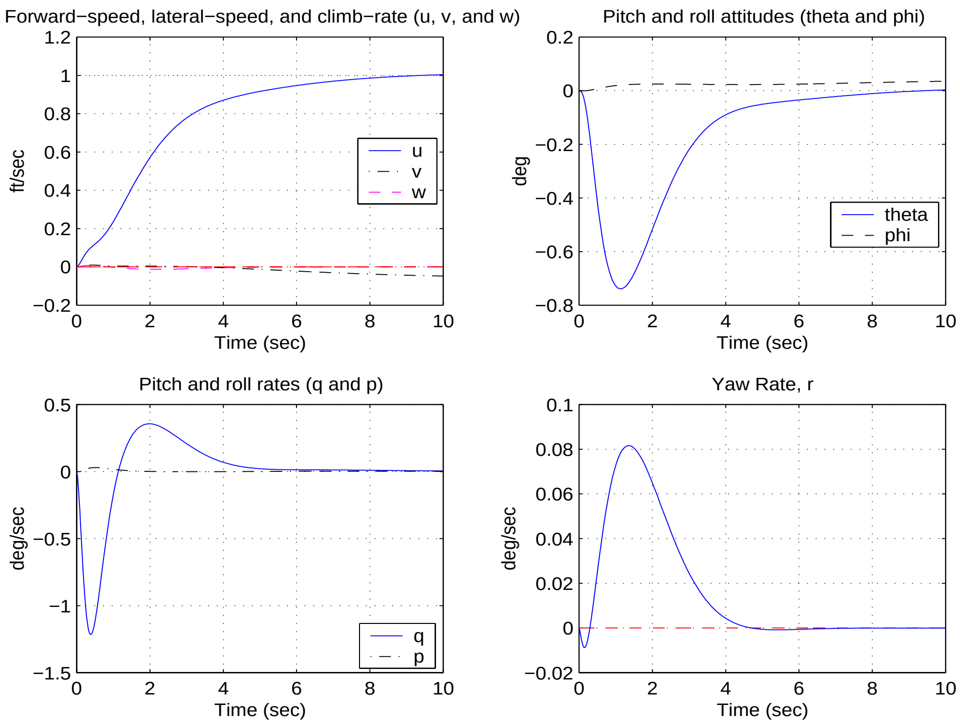


Figure 2.24: $\mathcal{H}^\infty$ Controller: Climb-Rate (w) Step Command Response: Outputs

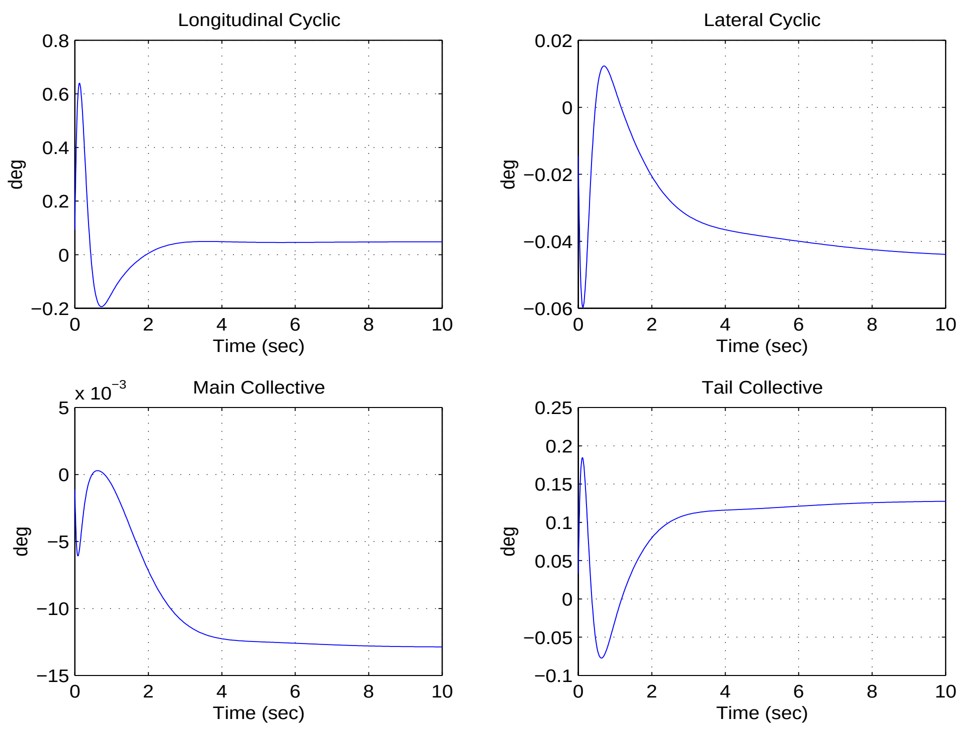


Figure 2.25: $\mathcal{H}^{\infty}$ Controller: Climb-Rate (w) Step Command Response: Controls

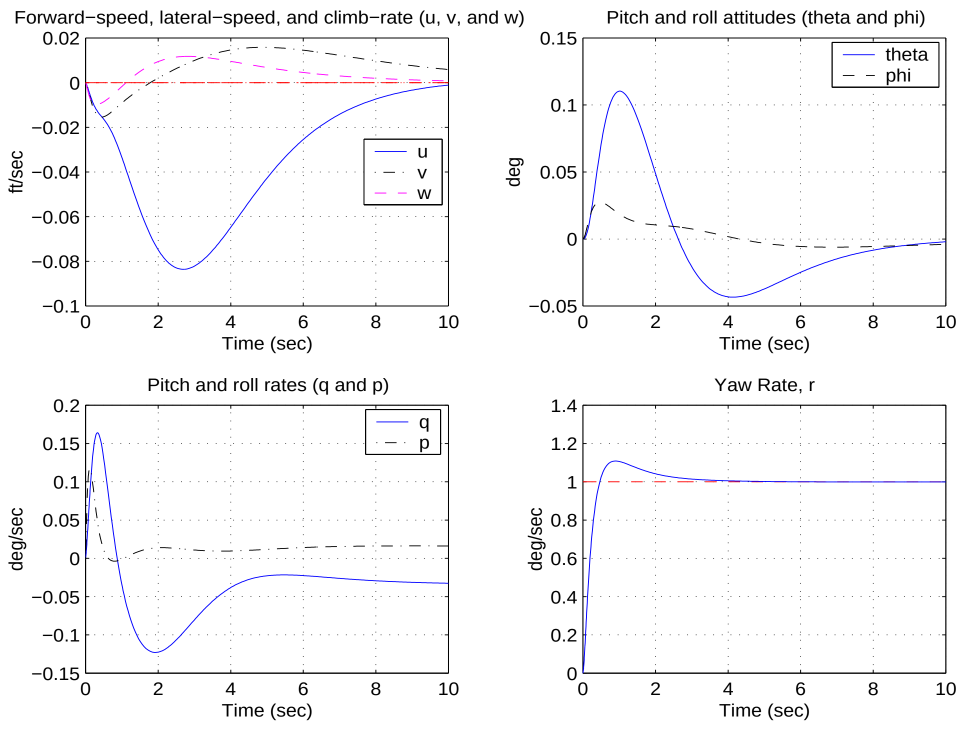


Figure 2.26: $\mathcal{H}^{\infty}$ Controller: Yaw-Rate (r) Step Command Response: Outputs

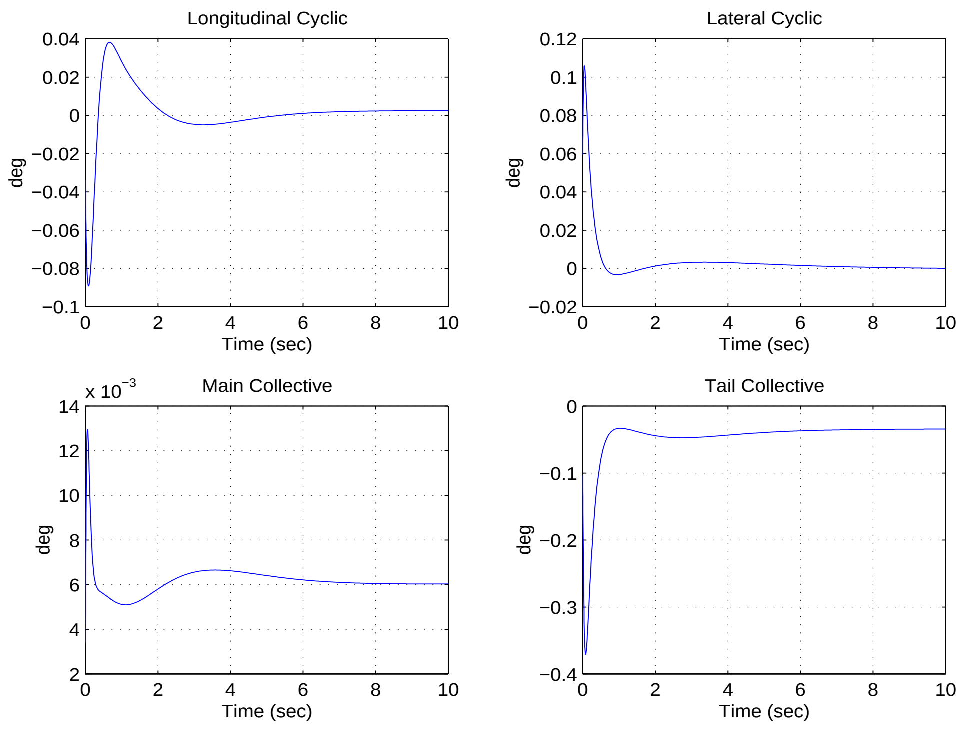


Figure 2.27: $\mathcal{H}^\infty$ Controller: Yaw-Rate (r) Step Command Response: Controls

$$u_p = \begin{bmatrix} \delta_{coll} & - & \text{Main rotor collective control} & \text{(deg)} \\ \delta_{lat} & - & \text{Lateral cyclic pitch control} & \text{(deg)} \\ \delta_{long} & - & \text{Longitudinal cyclic pitch control} & \text{(deg)} \\ \delta_{TR} & - & \text{Tail rotor collective control} & \text{(deg)} \end{bmatrix} \tag{2.51}$$

$$x_p = \begin{bmatrix} U, V, W & - & \text{Vehicle x,y,z velocity components} & \text{ft/sec} \\ P, Q, R & - & \text{Vehicle rotational velocities} & \text{rad/sec} \\ \theta, \phi & - & \text{Pitch and roll Euler angles} & \text{rad} \\ sa1, sa2 & - & \text{Reduced balanced sensor and actuator states} & \end{bmatrix} \tag{2.52}$$

The dimensional linear matrices are:

$$A_p = \begin{bmatrix} -0.0286 & -0.0637 & 0.0205 & 0.229 & 7.97 & -0.257 & 0 & -32 & 0 & 0 \\ 0.0779 & -0.231 & 0.00593 & -8.29 & -1.03 & -1.64 & 32 & 0.164 & 0 & 0 \\ 0.00463 & -0.0257 & -0.261 & -0.379 & 2.25 & 2.19 & 1.6 & -3.28 & 0 & 0 \\ 0.00793 & -0.05 & 0.00952 & -2.7 & -0.134 & -0.662 & 0 & 0 & 0 & 0 \\ 0.00473 & 0.0118 & 0.000163 & -0.00915 & -0.75 & 0.0244 & 0 & 0 & 0 & 0 \\ 0.00393 & -0.0049 & 0.0008 & -1.05 & 0.413 & -0.4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -0.00513 & 0.103 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.999 & 0.0499 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2.187 & 34.76 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -34.76 & -4.551 \end{bmatrix} \tag{2.53}$$

$$B_p = \begin{bmatrix} 0.435 & 0.576 & -0.114 & -0.00086 \\ -0.158 & 0.136 & 0.491 & 0.282 \\ -4.27 & 0.0575 & -0.025 & 0.00118 \\ -0.0438 & -0.06 & 0.647 & 0.08 \\ 0.0072 & -0.101 & -0.09 & -0.00187 \\ 0.08 & 0.00973 & 0.2 & -0.0455 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1.328 & -0.0109 & 0.0103 & -0.0023 \\ 1.749 & -0.0255 & 0.0046 & -0.0003 \end{bmatrix} \tag{2.54}$$

$$C_p = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0.0104 & 0.0008 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -0.0021 & -0.001 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -0.0182 & 0.0208 \\ -0.0001 & 0.0008 & 0.0081 & 0.0118 & -0.0699 & -0.0681 & 0 & 0 & 1.328 & -1.749 \end{bmatrix} \tag{2.55}$$

$$D_p = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.1327 & -0.0018 & 0.0008 & 0 \end{bmatrix} \tag{2.56}$$

The states being fed back into the controller are the three body angular rates, and a heave accelerometer sensor measurement.

The eigenvalues and associated eigenvectors for the plant are given in Table 2.7. A pole-zero map of the plant is shown in Figure 2.28. A singular-value plot of the plant is shown in Figure 2.29. Just as for the 8^{th}-order model, the eigenvalues and eigenvectors of the plant show that the system is open-loop unstable, due to the characteristic backflapping modes of a helicopter. There is considerable cross-coupling between the longitudinal and lateral dynamics.

A flight control system for the 10^{th}-order Apache model is now discussed.

$\lambda_1 = -3.26$	$\lambda_2 = -0.977$	$\lambda_3 = 0.0813 \pm j0.629$
$x_1 = \begin{bmatrix} 0.00295 \\ 0.0978 \\ 0.00151 \\ 0.891 \\ -0.0267 \\ 0.34 \\ -0.284 \\ 0.00298 \\ 0 \\ 0 \end{bmatrix}$	$x_2 = \begin{bmatrix} 0.434 \\ 0.0125 \\ 0.0633 \\ 0.0363 \\ -0.592 \\ 0.321 \\ -0.0742 \\ 0.589 \\ 0 \\ 0 \end{bmatrix}$	$x_3 = \begin{bmatrix} 0.185 \pm j - 0.334 \\ 0.291 \pm j - 0.0718 \\ 0.00334 \pm j - 0.0291 \\ -0.185 \pm j0.179 \\ 0.0983 \pm j - 0.258 \\ -0.202 \pm j - 0.465 \\ 0.165 \pm j0.35 \\ -0.421 \pm j - 0.194 \\ 0 \\ 0 \end{bmatrix}$
$\lambda_4 = 0.111 \pm j0.516$	$\lambda_5 = -0.259 \pm j0.0428$	$\lambda_6 = -3.37 \pm j34.7$
$x_4 = \begin{bmatrix} 0.436 \pm j - 0.34 \\ 0.19 \pm j - 0.0231 \\ 0.0221 \pm j - 0.0344 \\ -0.0922 \pm j0.0981 \\ 0.148 \pm j - 0.229 \\ -0.0657 \pm j - 0.46 \\ 0.0562 \pm j0.205 \\ -0.408 \pm j - 0.369 \\ 0 \\ 0 \end{bmatrix}$	$x_5 = \begin{bmatrix} -0.055 \pm j0.217 \\ -0.0266 \pm j - 0.124 \\ 0.329 \pm j - 0.542 \\ -0.0695 \pm j - 0.0354 \\ -0.0325 \pm j - 0.0387 \\ 0.609 \pm j0.382 \\ 0.0272 \pm j - 0.0116 \\ -0.00438 \pm j0.075 \\ 0 \\ 0 \end{bmatrix}$	$x_6 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.707 \pm j - 0.024 \\ 0 \pm j0.707 \end{bmatrix}$

Table 2.7: 10th-Order Apache Model: Plant Eigenvalues and (dimensional) Eigenvectors

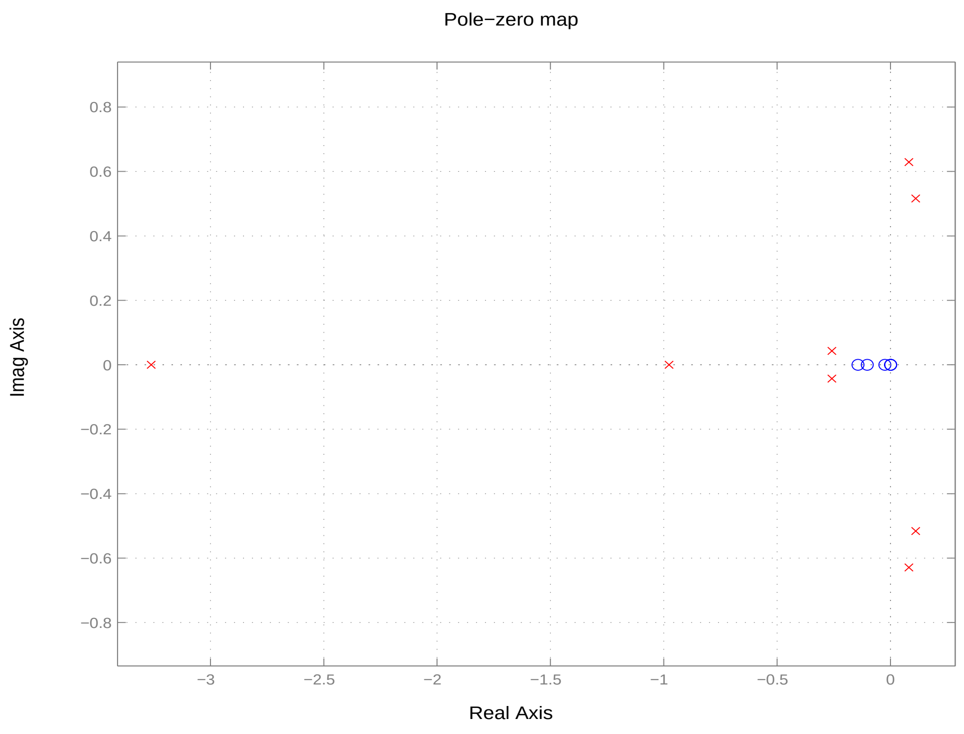


Figure 2.28: 10th-order Apache Model: Plant Pole-Zero Map

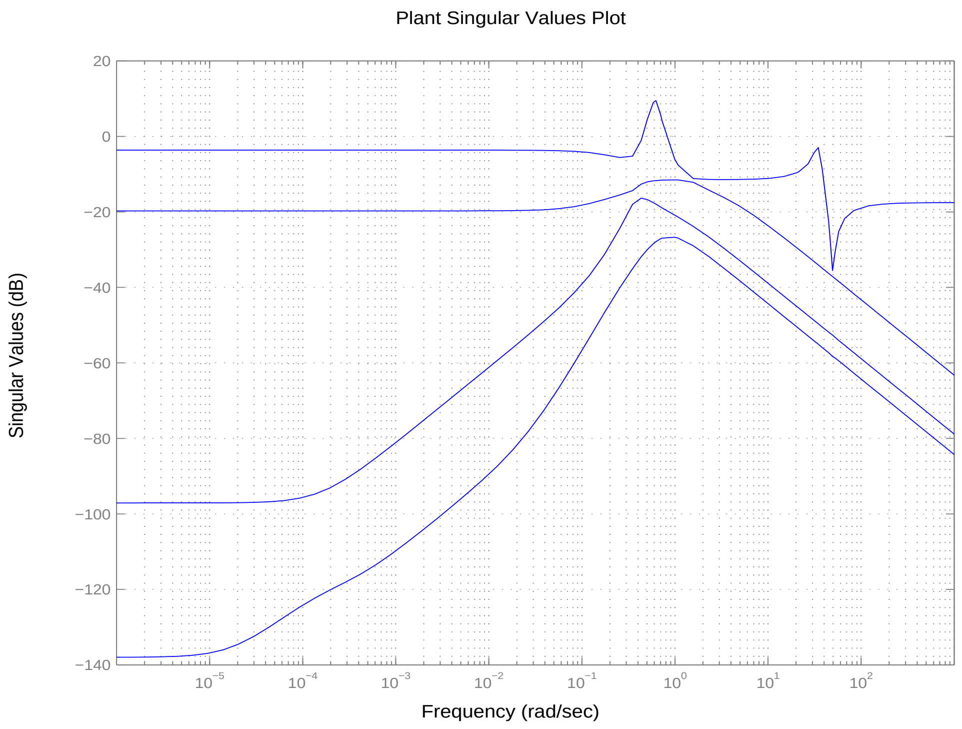


Figure 2.29: 10th-order Apache Model: Plant Singular Value Plot

2.2.4 Eigenstructure Assignment Controller with Loop Transfer Recovery

A controller design based on Eigenstructure Assignment with Loop Transfer Recovery (ESA/LTR) was published by Ekblad [6]. The controlled outputs are $y_c = [w, u, v, r]$.

The controller structure is shown in Figure 2.30, where the controller matrices are:

$$G = \begin{bmatrix} 0.0034 & 0.0161 & -0.0161 & 0.082 & -0.7626 & 0.1876 & -0.4536 & 0.4166 & 0 & 0 \\ 0.5345 & -0.1732 & 0.0385 & -0.3369 & -32.41 & -3.971 & -6.324 & -52.95 & 0 & 0 \\ 0.0625 & 0.0047 & 0.0143 & 0.3257 & -0.4237 & 4.683 & 6.841 & -2.604 & 0 & 0 \\ 0.1936 & -0.2146 & 0.0256 & 21.61 & -21.87 & -45.24 & 10.23 & -24.16 & 0 & 0 \end{bmatrix} \qquad (2.57)$$

$$H = \begin{bmatrix} -1.584 & 241.9 & 117 & -2.657 \\ -73.84 & 54.04 & 13.61 & 51.82 \\ 9.202 & 6.823 & 7.41 & 496.9 \\ 10.1 & -0.1241 & -0.7049 & 0.2388 \\ -0.128 & 11.6 & 0.259 & -0.9404 \\ -0.7092 & 0.2572 & 11.86 & -0.9402 \\ 0.0177 & 0.0536 & -0.0914 & 0.0317 \\ 0.0178 & 0.08 & -0.34 & -0.0481 \\ 0.976 & -0.5505 & -2.137 & 123.4 \\ 0.0644 & -0.2759 & 1.884 & -179.1 \end{bmatrix} \qquad (2.58)$$

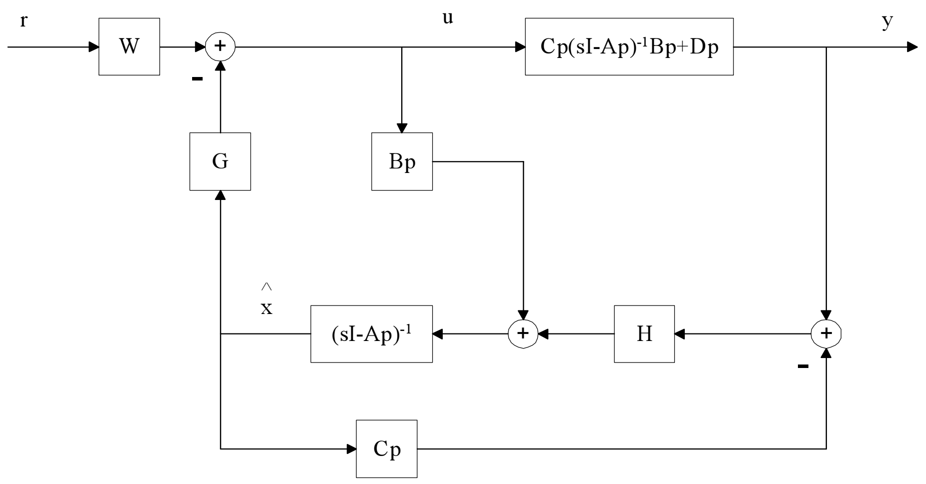


Figure 2.30: ESA/LTR Controller Structure

$$W = \begin{bmatrix} 0.0772 & 0.0055 & 0.0106 & 0.0124 \\ -0.0524 & 0.6 & -0.079 & -0.0302 \\ -0.0144 & 0.03 & 0.1003 & 0.1012 \\ 0.0681 & 0.2044 & 0.0311 & -0.8352 \end{bmatrix} \tag{2.59}$$

Analysis of the ESA/LTR Controller. The closed-loop poles of the system with the ESA/LTR controller are listed in Table 2.8. The smallest damping ratio of the closed-loop poles is $\zeta = 0.62$.

Figure 2.31 shows the closed-loop singular value plot for the ESA/LTR control system. This is the singular value plot from reference commands to commanded quantities, $y_c = [w, u, v, r]$. This plot shows that the closed-loop system has excellent low-frequency command following with a maximum bandwidth of approximately 1.5 rad/sec. The pole-zero map for the closed-loop system is shown in Figure 2.32.

Figure 2.33 shows the singular value plot from reference commands to tracking error. Again, this plot shows that low-frequency command following is very good, with negligible steady-state errors to step-reference commands.

Figure 2.34 shows the singular value plot from plant input disturbances to the outputs $y_c = [w, u, v, r]$. This plot reveals that this control system has no capability to reject low-frequency plant input disturbances. Thus, step input disturbances will result in finite steady-state disturbances in the outputs.

The response of the closed-loop system to step reference commands in each channel (i.e. forward-

$\lambda_i(T_{rz_p})$	Damping Ratio (ζ)	Natural Frequency (ω_n) (rad/sec)
$\lambda_1 = -484$	1.0000	484.1321
$\lambda_2 = -12.5$	1.0000	12.5145
$\lambda_{3,4} = -12.2 \pm j0.22$	0.9998	12.2109
$\lambda_{5,6} = -2.11 \pm j2.62$	0.6275	3.3599
$\lambda_7 = -3.26$	1.0000	3.2610
$\lambda_8 = -3.3$	1.0000	3.3000
$\lambda_9 = -2.8$	1.0000	2.8005
$\lambda_{10,11} = -0.247 \pm j0.272$	0.6730	0.3671
$\lambda_{12,13} = -0.1 \pm j0.0929$	0.7325	0.1365
$\lambda_{14} = -0.33$	1.0000	0.3300
$\lambda_{15,16} = -0.806 \pm j0.268$	0.9488	0.8490
$\lambda_{17,18} = -0.816 \pm j0.268$	0.9499	0.8586

Table 2.8: ESA/LTR Controller: Closed-Loop Poles

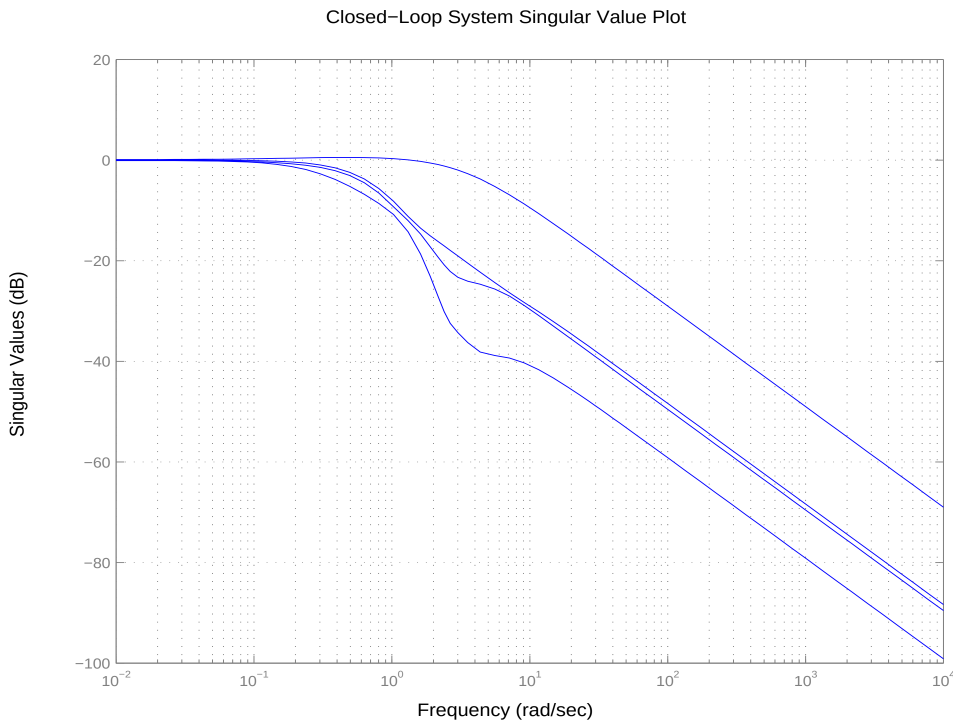


Figure 2.31: ESA/LTR Controller: Closed-Loop System SV Plot

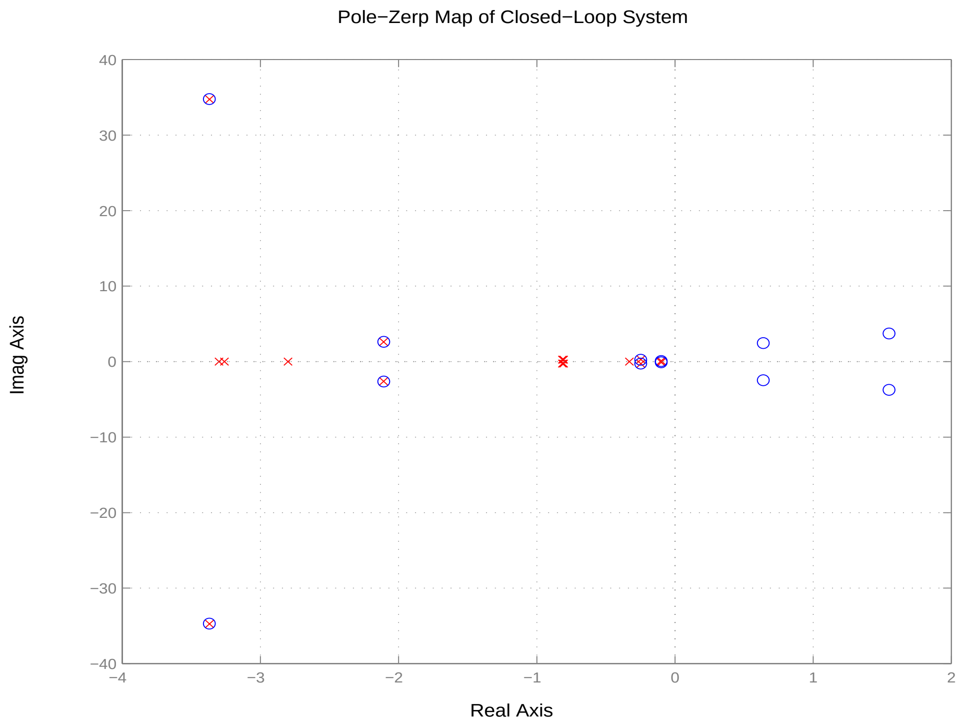


Figure 2.32: ESA/LTR Controller: Closed-Loop Pole-Zero Map

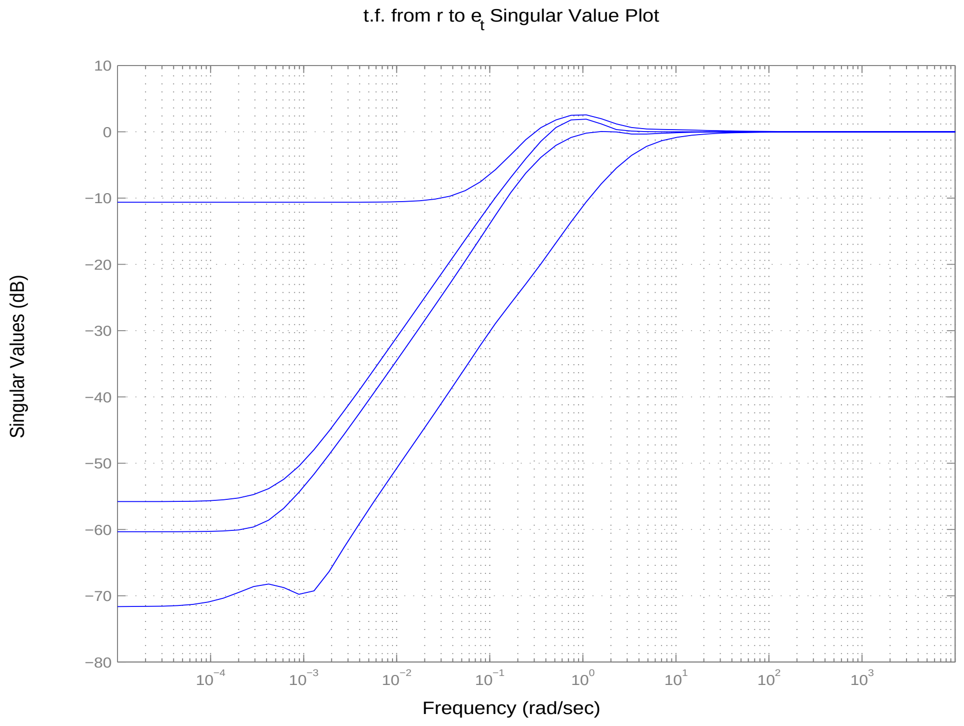


Figure 2.33: ESA/LTR Controller: Reference to Tracking Error SV Plot

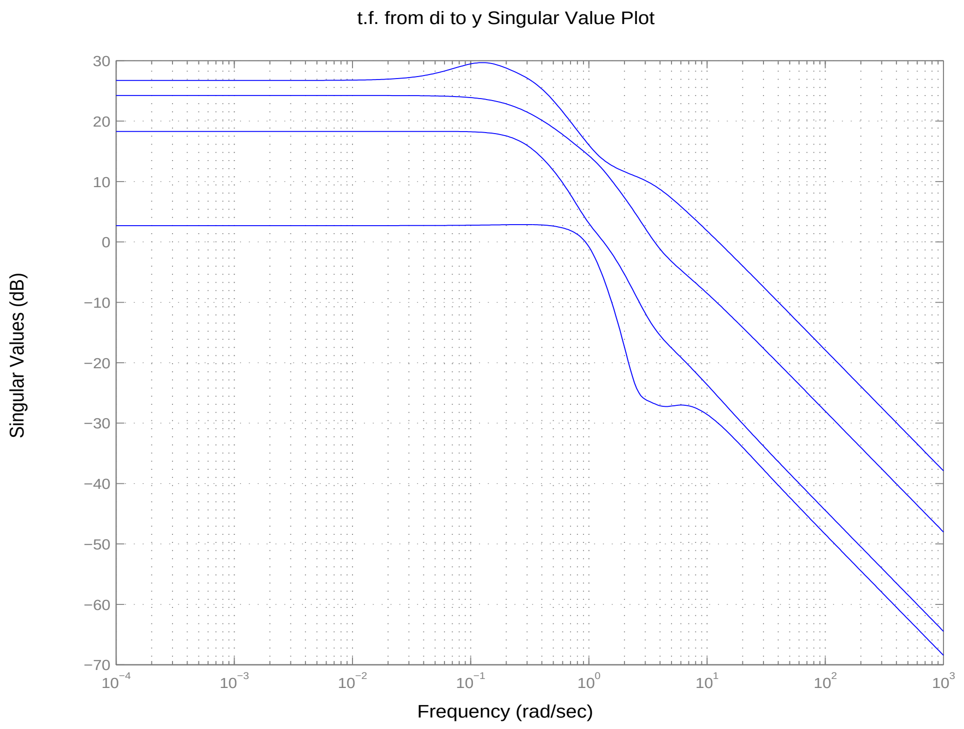


Figure 2.34: ESA/LTR Controller: Input Disturbance to Output SV Plot

speed, lateral-speed, climb-rate, and yaw-rate) are shown in Figures 2.35 - 2.42.

2.3 7-DOF Twin-Lift Helicopter System (TLHS) Model and Control Laws

This section describes the models and control laws for a 6-degree-of-freedom McDonnell Douglas (now Boeing) AH-64 Apache helicopter gunship that are used in this thesis.

The dynamical models will be presented first, followed by the control systems.

2.3.1 14th-order TLHS System Model and Dynamics

The TLHS longitudinal dynamics near hovering trim [11], [12] are now presented. A seven degree of freedom model is used to describe these dynamics. Each helicopter (master and slave) is described by the three degrees of freedom $(x_m, z_m, \theta_m, x_s, z_s, \theta_s)$. Because the payload is constrained by a tether bar assembly, only one additional degree of freedom is required - the load's horizontal coordinate,

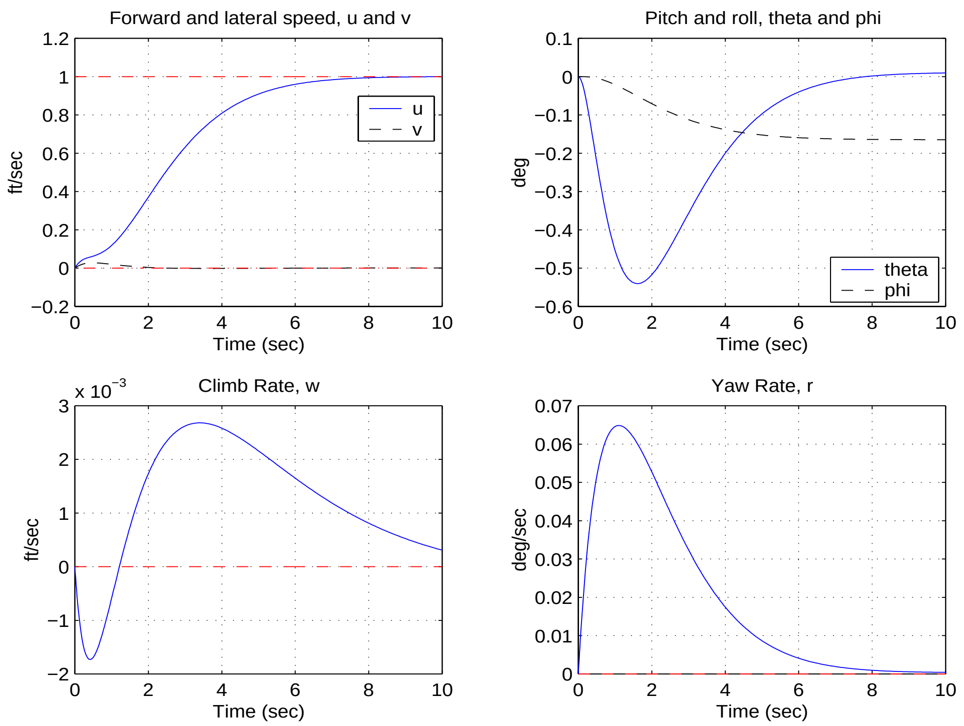


Figure 2.35: ESA/LTR Controller: Forward-Speed (u) Step Command Response: Outputs

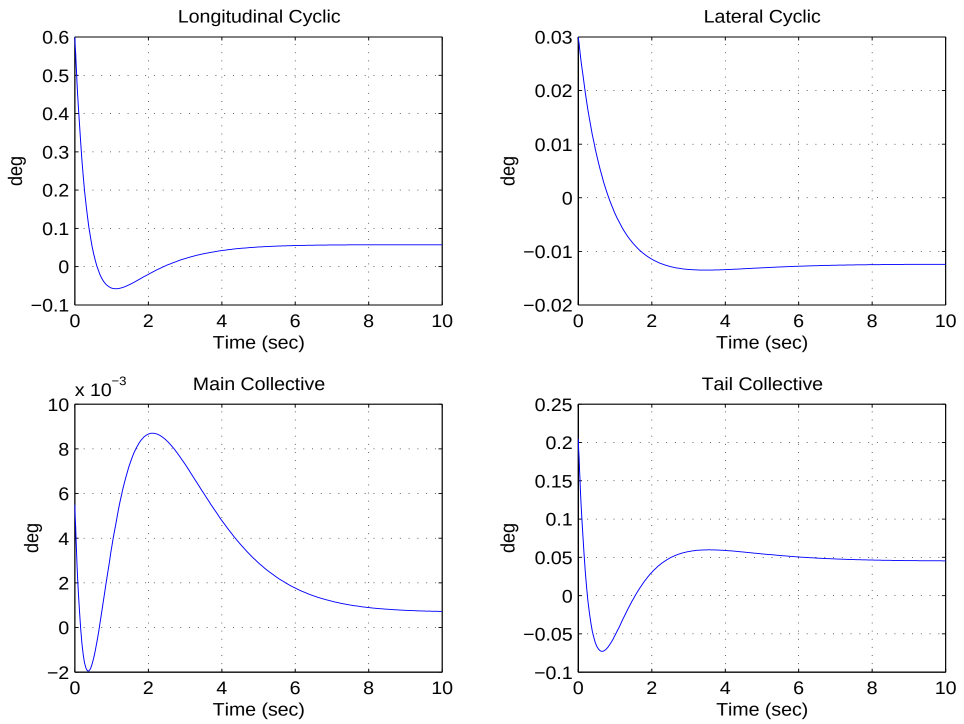


Figure 2.36: ESA/LTR Controller: Forward-Speed (u) Step Command Response: Controls

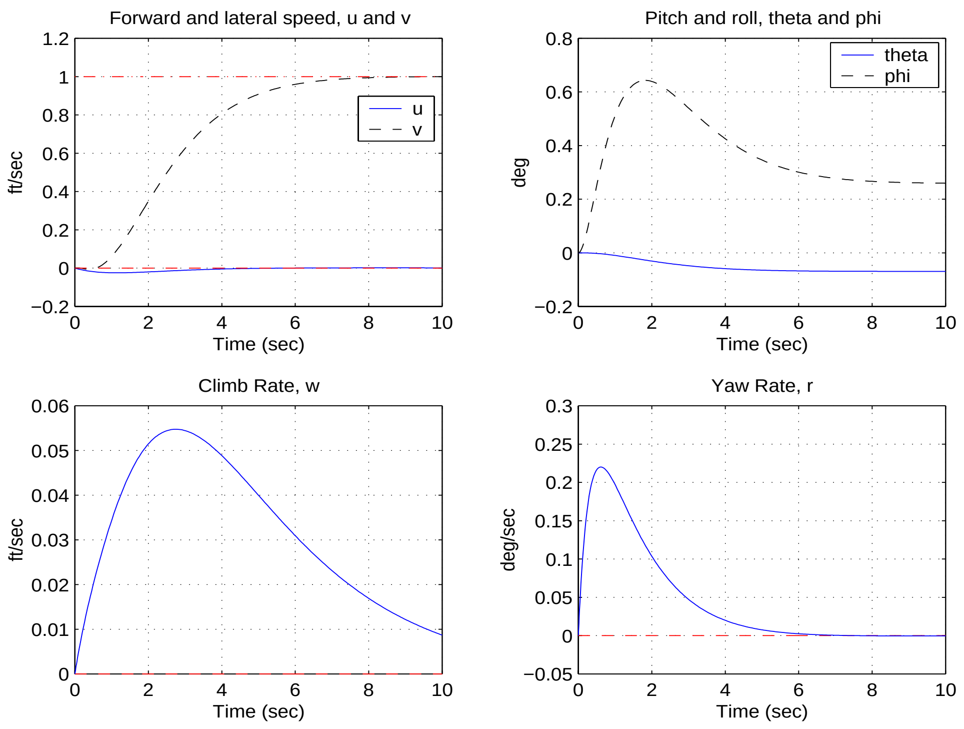


Figure 2.37: ESA/LTR Controller: Lateral-Speed (v) Step Command Response: Outputs

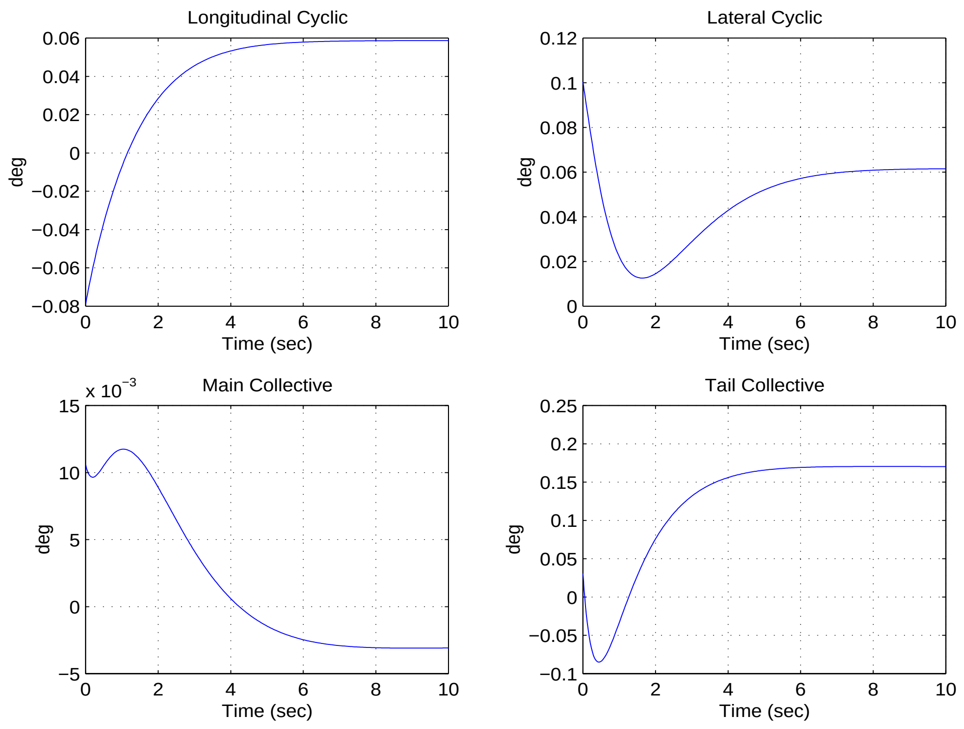


Figure 2.38: ESA/LTR Controller: Lateral-Speed (v) Step Command Response: Controls

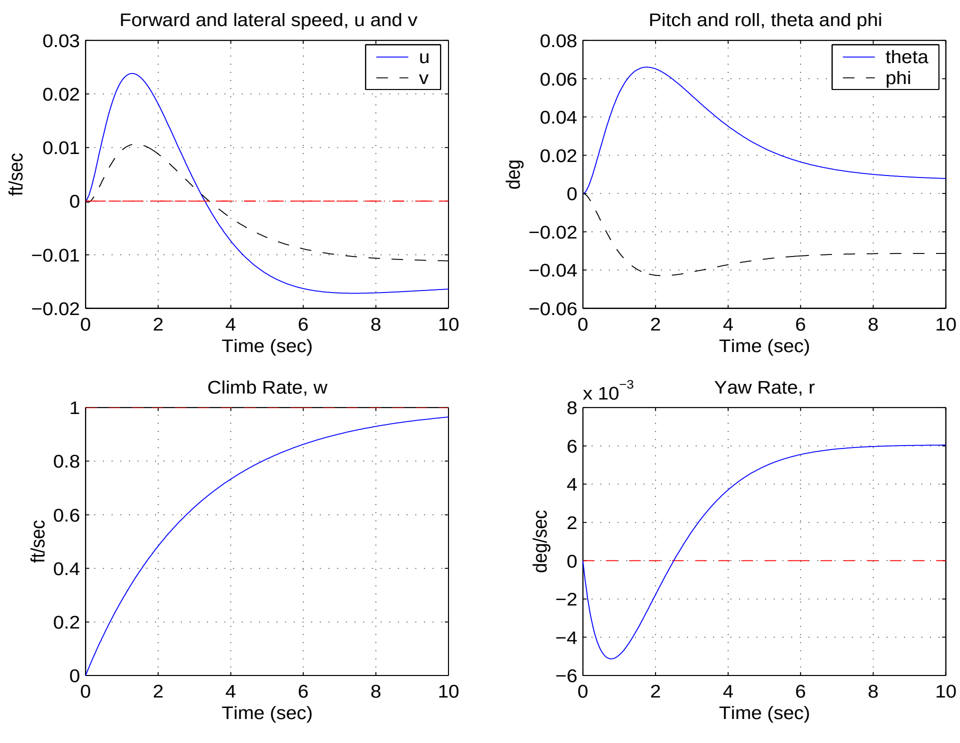


Figure 2.39: ESA/LTR Controller: Climb-Rate (w) Step Command Response: Outputs

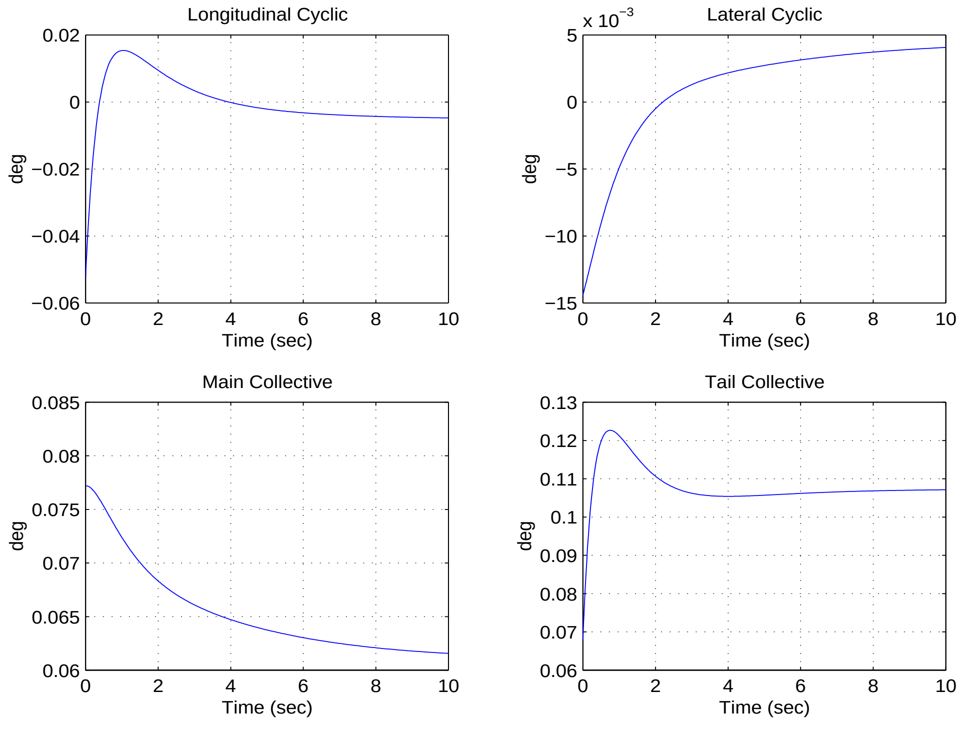


Figure 2.40: ESA/LTR Controller: Climb-Rate (w) Step Command Response: Controls

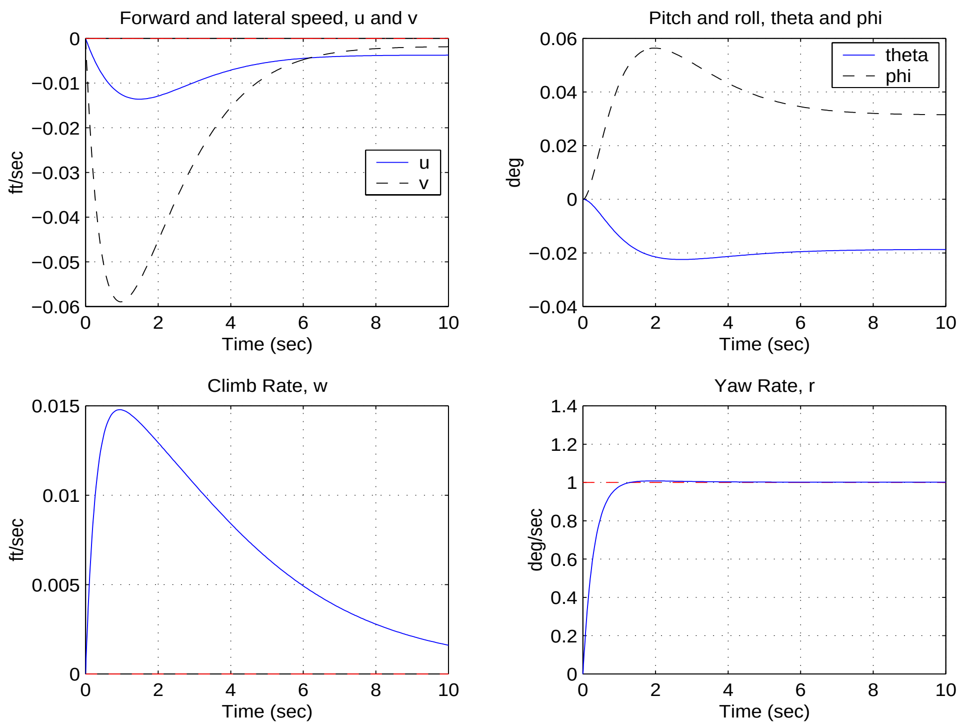


Figure 2.41: ESA/LTR Controller: Yaw-Rate (r) Step Command Response: Outputs

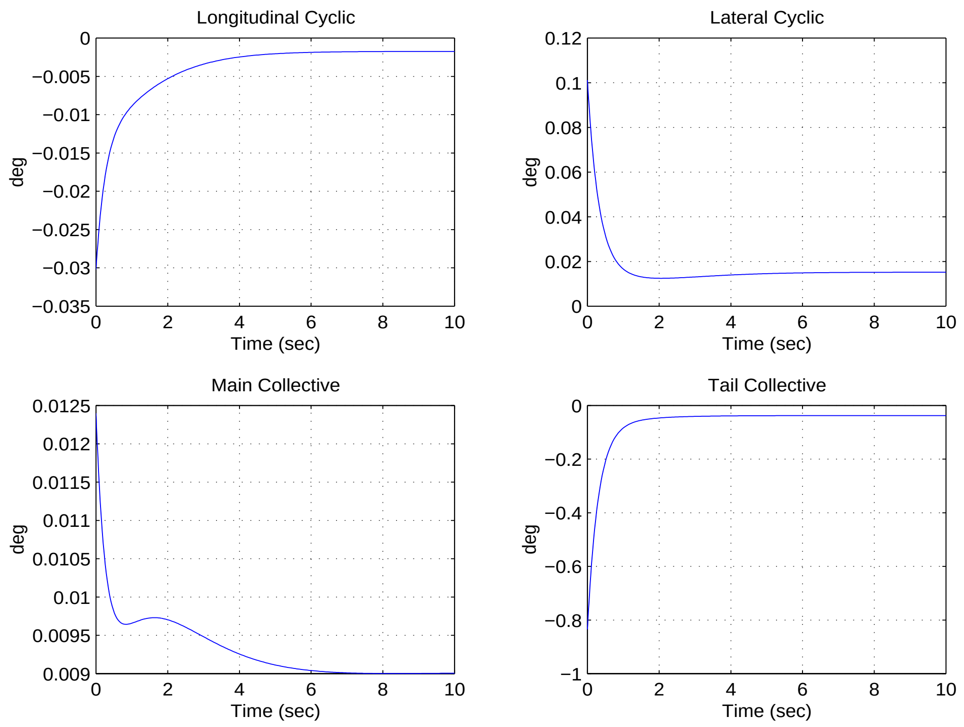


Figure 2.42: ESA/LTR Controller: Yaw-Rate (r) Step Command Response: Controls

x_L. From these degrees of freedom, Lagrangian techniques were used to derive the equations of motion for the TLHS - a set of 12 linear ordinary differential equations [11], [12]. Each helicopter has associated with it a collective pitch control (Θ_m, Θ_s) and a cyclic pitch control (B_{c_m}, B_{c_s}), thus giving four controls for the TLHS. These controls permit independent control over four quantities.

Generically, the TLHS exibits two instabilities near hover [11]. One is the backflapping instability of the main rotor with forward velocity characteristic of a single hovering helicopter [5]. The other instability is characteristic of a hovering helicopter tethered to a fixed point in space [8].

When the tether lengths are the same and the helicopters assumed to be identical, the 12^{th} order TLHS model decouples into three systems of differential equations. These three systems describe the "three basic motions" of the TLHS. These three basic motions are referred to as the Average Vertical motion (AVM), Symmetric Motion (SM), and the Anti-Symmetric Motion (ASM).

The TLHS is described by seven degrees of freedom: three average variables, three difference variables, and one generalized load coordinate. The TLHS equations of motion (plant) implemented within the developed environment take the form

$$\begin{aligned} \dot{x}_p &= A_p x_p + B_p u_p && (2.60) \\ y_p &= C_p x_p && (2.61) \end{aligned}$$

where the control vector $u_p \in \mathcal{R}^{4\times 1}$, state vector $x_p \in \mathcal{R}^{12\times 1}$, and output vector $y_p \in \mathcal{R}^{4\times 1}$ are given by:

$$u_p = \left[\begin{array}{lllll} \text{AVM}: & \Sigma\Theta_c & - & \text{average collective controls} & \text{(rad)} \\ \text{SM}: & \Delta B_{lc} & - & \text{differential cyclic controls} & \text{(rad)} \\ \text{ASM}: & \Delta\Theta_c & - & \text{differential collective controls} & \text{(rad)} \\ & \Sigma B_{lc} & - & \text{average cyclic controls} & \text{(rad)} \end{array}\right] \qquad (2.62)$$

$$x_p = \begin{bmatrix} \text{AVM}: & \Sigma\dot{z} & - & \text{average vertical velocity} & \text{(ft/sec)} \\ \text{SM}: & \Delta x & - & \text{horizontal separation between helicopters} & \text{(ft)} \\ & \Delta\theta & - & \text{differential pitch attitude} & \text{(rad)} \\ & \Delta\dot{x} & - & \text{differential horizontal velocity} & \text{(ft/sec)} \\ & \Delta\dot{\theta} & - & \text{differential pitch rate} & \text{(rad/sec)} \\ \text{ASM}: & \Sigma\theta & - & \text{average pitch attitude} & \text{(rad)} \\ & \Delta z & - & \text{vertical separation between helicopters} & \text{(ft)} \\ & x'_L & - & \text{generalized load coordinate} & \text{(ft)} \\ & \Sigma\dot{x} & - & \text{average horizontal velocity} & \text{(ft/sec)} \\ & \Sigma\dot{\theta} & - & \text{average pitch rate} & \text{(rad/sec)} \\ & \Delta\dot{z} & - & \text{differential vertical velocity} & \text{(ft/sec)} \\ & \dot{x}'_L & - & \text{generalized load velocity} & \text{(ft/sec)} \end{bmatrix} \tag{2.63}$$

$$y_p = \begin{bmatrix} \text{AVM}: & \Sigma\dot{z} & - & \text{average vertical velocity} & \text{(ft/sec)} \\ \text{SM}: & \Delta x & - & \text{horizontal separation} & \text{(ft)} \\ \text{ASM}: & x_L - \Sigma_x & - & \text{load deviation from center} & \text{(ft)} \\ & \Sigma\dot{x} & - & \text{average horizontal velocity} & \text{(ft/sec)} \end{bmatrix} \tag{2.64}$$

The state-space matrices for the TLHS Equal-Tether Configuration plant are listed below:

$$Ap_1 = [-0.2384] \tag{2.65}$$

$$Ap_2 = \begin{bmatrix} 0 & 0 & 1.0000 & 0 \\ 0 & 0 & 0 & 1.0000 \\ -1.0974 & -0.8847 & -0.0600 & 0 \\ -17.2659 & -5.0777 & 2.3491 & -3.100 \end{bmatrix} \tag{2.66}$$

$$Ap_3 = \begin{bmatrix} 0.0000 & 0.0000 & 0.0000 & 0.0000 & 1.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 1.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 1.0000 \\ -0.5620 & 0.0000 & 1.0974 & -0.0600 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 17.2659 & 2.3491 & -3.1000 & 0.0000 & 0.0000 \\ 0.4679 & -0.3885 & 2.0233 & 0.0000 & 0.0000 & -0.3361 & 0.0000 \\ -0.2220 & 0.1844 & -9.5654 & -0.6308 & 0.9117 & 0.1595 & 0.0000 \end{bmatrix} \tag{2.67}$$

$$Ap = diag(Ap_1, Ap_2, Ap_3) \tag{2.68}$$

$$Bp_1 = [4.0989] \tag{2.69}$$

$$Bp_2 = \begin{bmatrix} 0.0000 \\ 0.0000 \\ 0.4782 \\ -47.2400 \end{bmatrix} \tag{2.70}$$

$$Bp_3 = \begin{bmatrix} 0.0000 & 0.0000 \\ 0.0000 & 0.0000 \\ 0.0000 & 0.0000 \\ 0.0000 & 0.4782 \\ 0.0000 & -47.2400 \\ 5.7794 & 0.0000 \\ -2.7425 & 13.4145 \end{bmatrix} \tag{2.71}$$

$$Bp = diag(Bp_1, Bp_2, Bp_3) \tag{2.72}$$

$$Cp_1 = [1] \tag{2.73}$$

$$Cp_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \tag{2.74}$$

$$Cp_3 = \begin{bmatrix} 0.2941 & 0.5000 & 1.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 & 0.0000 & 0.0000 & 0.0000 \end{bmatrix} \tag{2.75}$$

$$Cp = diag(Cp_1, Cp_2, Cp_3) \tag{2.76}$$

The model represents a linearizarion of the nonlinear dynamical equations about hover [5].

Figure 2.43 shows the poles of the TLHS and their associated modes. Generically, the TLHS exibits two instabilities near hover [11]. One is the backflapping instability of the main rotor with forward velocity characteristic of a single hovering helicopter [5]. The other instability is characteristic of a hovering helicopter tethered to a fixed point in space [8]. These instabilities, and the desire for good command-following, motivate the need for an effective automatic flight control system.

2.3.2 LQG/LTR Controller

This section describes a Linear Quadratic Gaussian/Loop Transfer Recovery (LQG/LTR) controller for the TLHS. The design of this control system is discussed in [11][12].

TLHS Equal Tether Flight Control System. Because the TLHS equal tether-length configuration decouples into the AVM, SM, and ASM subsystems, we can separately apply our design methodology to each. In order to guarantee zero stady-state error to step commands, each of these

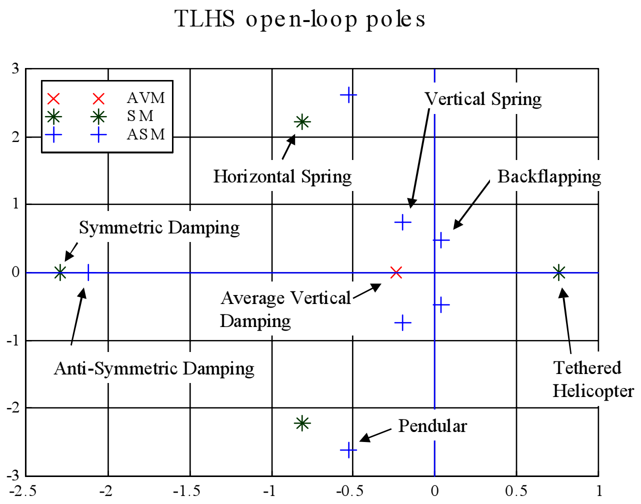


Figure 2.43: TLHS: Open-Loop Poles and Associated Modes

plants are augmented with integrators in each channel as follows:

$$A_i = \begin{bmatrix} A_{pi} & B_{pi} \\ O & O \end{bmatrix} \tag{2.77}$$

$$B_i = \begin{bmatrix} O \\ I \end{bmatrix} \tag{2.78}$$

$$C_i = \begin{bmatrix} C_{pi} & O \end{bmatrix} \tag{2.79}$$

where (A_1, B_1, C_1) are associated with the AVM, (A_2, B_2, C_2) with the SM, and (A_3, B_3, C_3) with the ASM.

These augmented plants form the *design plants* used in the AFCS design. A Linear Quadratic Gaussian/Loop Transfer Recovery (LQG/LTR) methodology was then applied to obtain the compensator [11][12]. The resulting compensator has the following structure:

$$K = diag(K_1, K_2, K_3) \tag{2.80}$$

where

$$K_i = G_i(sI - A_i + B_iG_i + H_iC_i)^{-1}H_i \tag{2.81}$$

For the AVM,

$$H_1 = \left[\begin{array}{cc} 0.4921 & 0.582 \end{array}\right]^T \tag{2.82}$$

$$G_1 = \left[\begin{array}{cc} 994.75 & 90.3 \end{array}\right] \tag{2.83}$$

for the SM,

$$H_2 = \left[\begin{array}{ccccc} 2.2063 & -4.9246 & 2.4339 & -4.5238 & 0.2194 \end{array}\right]^T \tag{2.84}$$

$$G_2 = \left[\begin{array}{ccccc} 955.46 & -36.449 & 282.73 & -2.6377 & 22.795 \end{array}\right] \tag{2.85}$$

and for the AVM,

$$H_3 = \left[\begin{array}{ccccccccc} 0.1201 & -0.0239 & 0.3605 & 1.6237 & 0.0017 & -0.1552 & 0.2494 & 0.6729 & 0.025 \\ 0.0605 & 0.0477 & -1.0979 & 0.3121 & 0.0117 & 1.0734 & 0.0244 & -0.2014 & -0.0129 \end{array}\right]^T \tag{2.86}$$

$$G_3 = \left[\begin{array}{ccccccccc} 13.0342 & -1.1653 & 49.7344 & 138.3322 & 156.5858 & 149.8413 & 45.5383 & 87.648 & 153.479 \\ -1.1653 & 23.592 & -63.2715 & -74.8953 & -79.9454 & 275.6631 & -17.1355 & -27.2046 & -49.3773 \end{array}\right] \tag{2.87}$$

Analysis of TLHS Equal Tether Flight Control System. In this section, the singular value plots or bode plots for each of the three subsystems (AVM, SM, and ASM) are presented.

AVM. The open-loop bode plot, the sensitivity, and complementary sensitivity plots for the AVM are shown in Figure 2.44. Figure 2.45 shows the time-domain response of the AVM to a step reference command of $\Sigma\dot{z} = 5ft/sec$.

SM. The open-loop bode plot, the sensitivity, and complementary sensitivity plots for the SM are shown in Figure 2.46. Figure 2.47 shows the time-domain response of the SM to a disturbance $\Delta x = 1ft$. This plot demonstrates the rejection of step disturbances by the SM AFCS. Figure 2.48 shows the corresponding control signal.

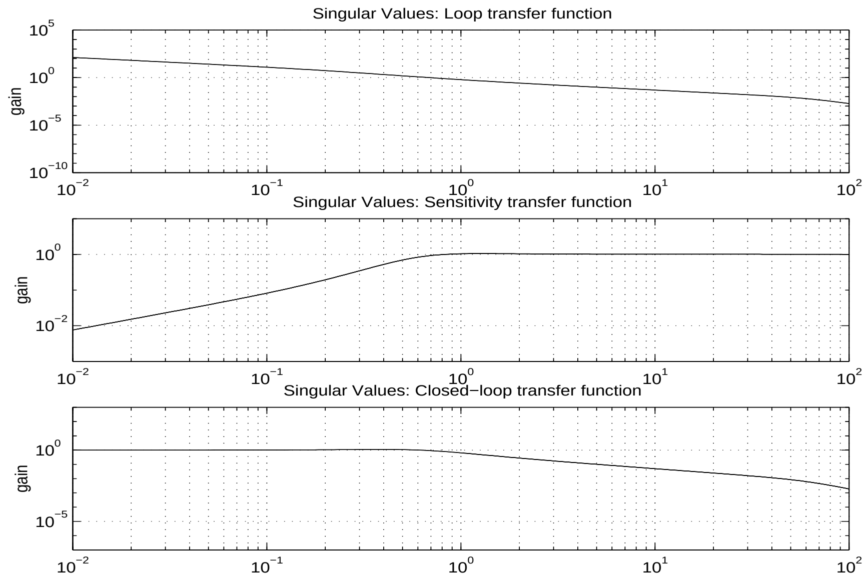


Figure 2.44: Bode plots for AVM

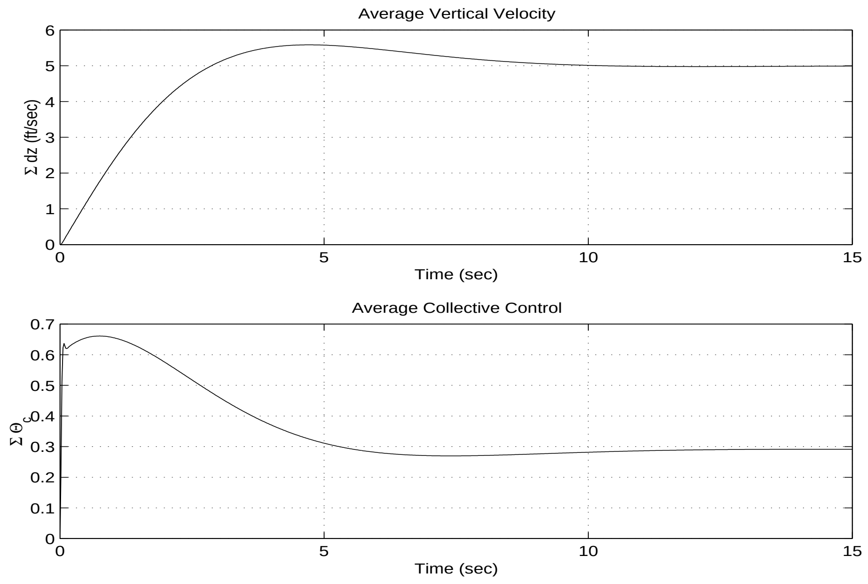


Figure 2.45: AVM AFCS command following

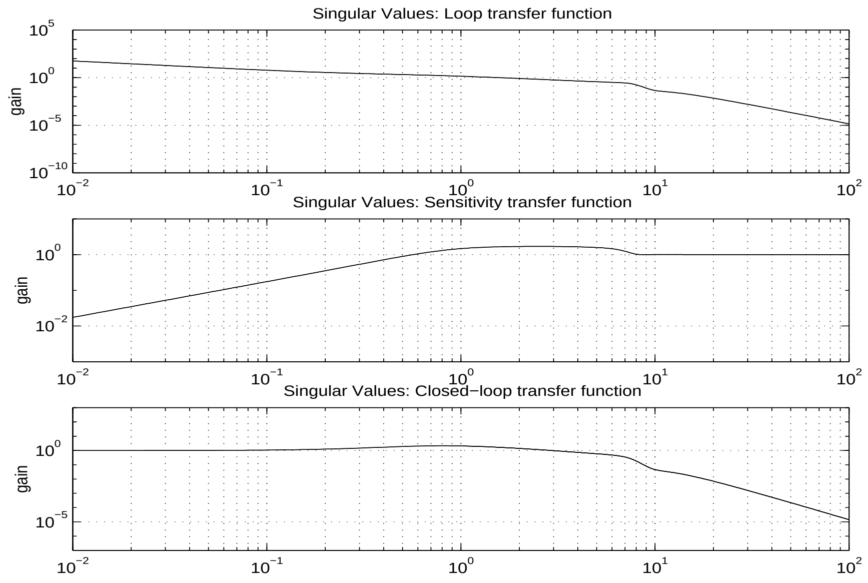


Figure 2.46: Bode plots for SM

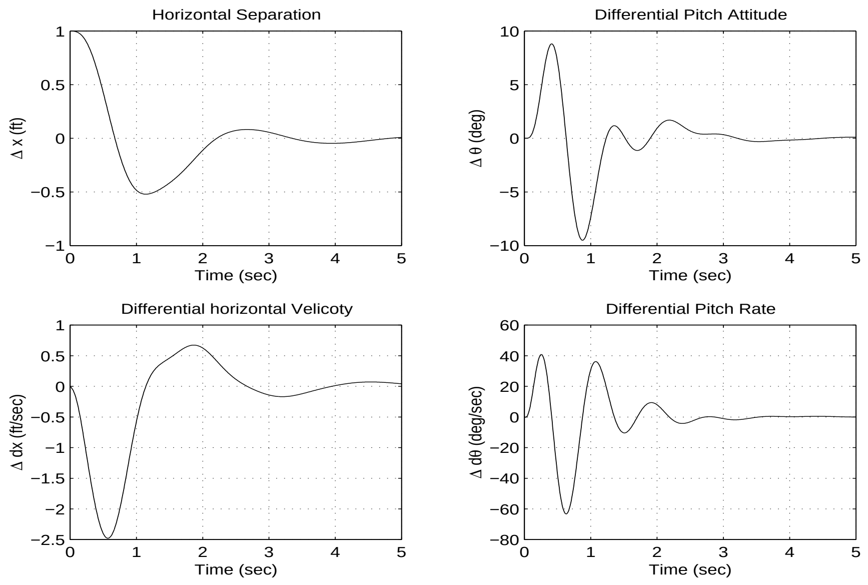


Figure 2.47: SM AFCS disturbance rejection

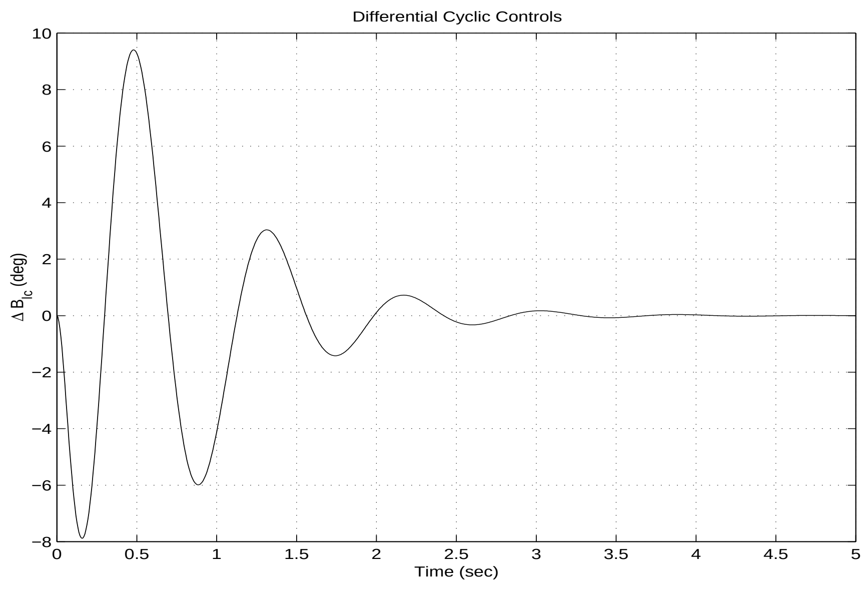


Figure 2.48: SM AFCS disturbance rejection: control signal

ASM. The open-loop bode plot, the sensitivity, and complementary sensitivity plots for the ASM are shown in Figure 2.49. Figure 2.50 shows the time-domain response of the ASM to both a disturbance of $x_L - \Sigma x = -1ft$ and a reference command of $\Sigma\dot{x} = 5ft/sec$ simultaneously. Figure 2.51 shows the corresponding control signals.

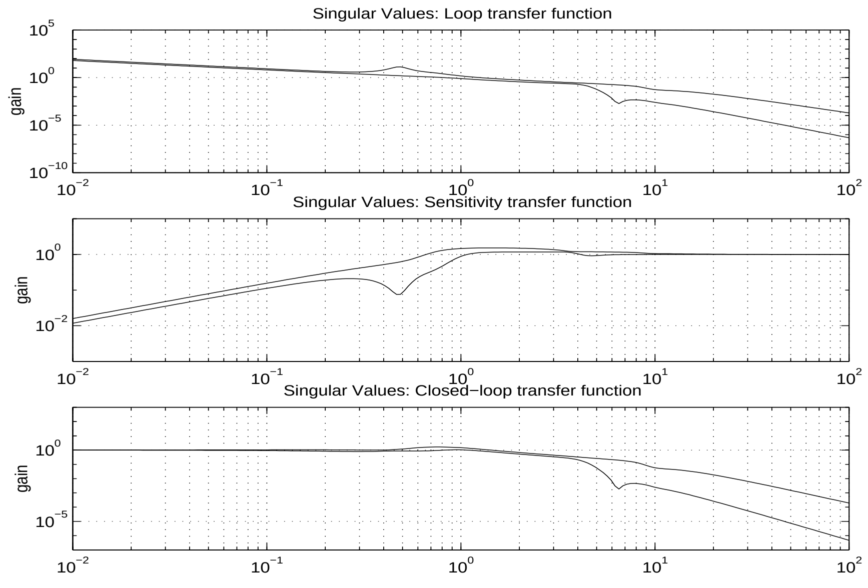


Figure 2.49: Bode plots for ASM

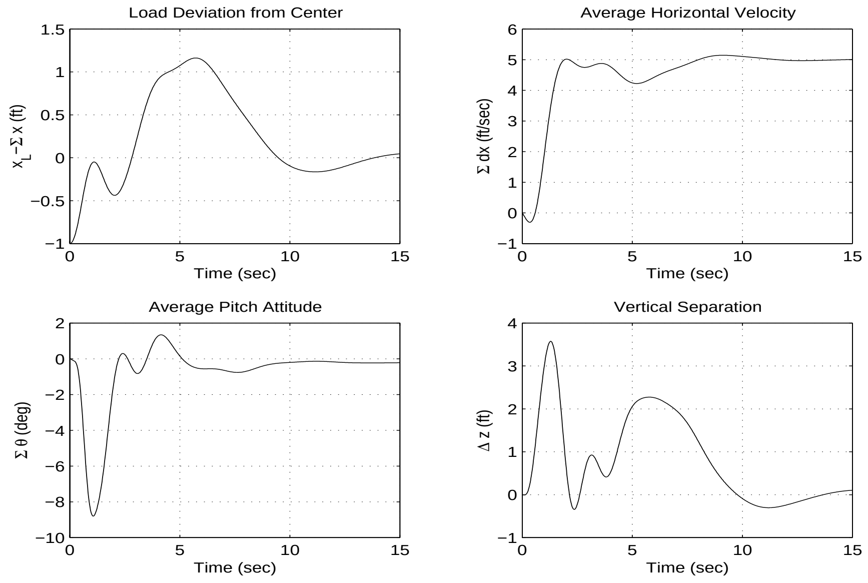


Figure 2.50: ASM AFCS disturbance rejection

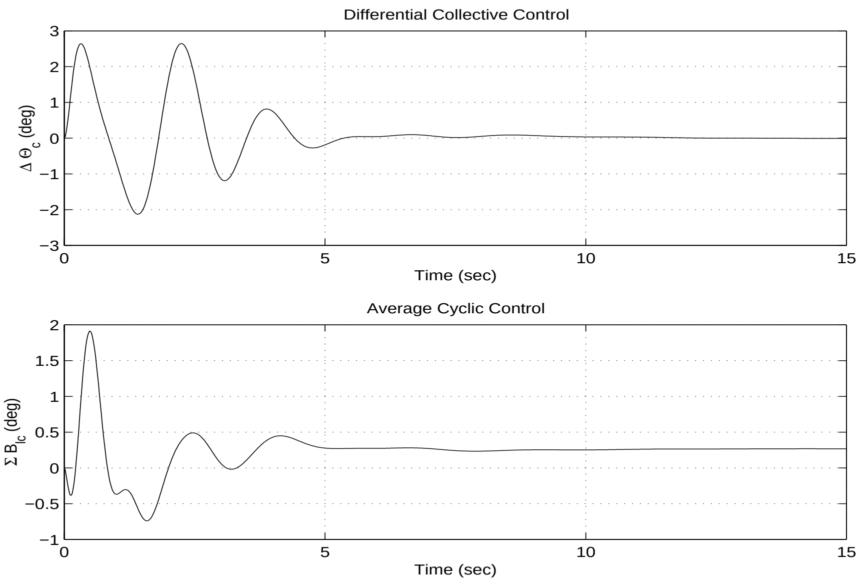


Figure 2.51: ASM AFCS disturbance rejection: control signals

CHAPTER 3

Description of Interactive MoSART Single-Lift Helicopter Environment

The *Interactive MoSART Helicopter Environment* is an interactive application intended to serve as a virtual testbed to facilitate the analysis, design, and evaluation of high performance control laws for helicopter systems. The software runs on any PC-compatible computer running Microsoft Windows '95, '98, or NT. For optimum performance, a fast Pentium processor (300MHz$^+$) and a 3D-accelerated video card is recommended.

We begin our discussion with some development background. The functionality and content of the individual modules that make up the environment are then described.

Development Background. The *Interactive MoSART Helicopter Environment* was written in C++ and developed using Microsoft Visual C++ version 5.0. C++ is a modern *object-oriented programming* (OOP) language. It supports the full complement of OOP features such as objects, inheritance, polymorphism, etc. Practically, the use of an OOP language (in contrast to non-OOP languages such as BASIC, C, or Fortran) significantly improves written code by enforcing the use of consistent structures, making it easier to write more modular, readable, and maintainable code. The develpment of this software makes full use of OOP methodologies.

The developed C++ code was built upon the Microsoft Foundation Classes (MFC). The MFCs are a set of C++ classes that encapsulate the functionality of Windows, allowing Windows programs to be more easily developed. The use of the MFC framework provides a flexible and powerful function-base and provides a basic structure for the code. Benefits of this framework include an (inherited) modern user-friendly interface and many standard features such as timers, graphics, data file storage and retrieval. The use of the MFCs mean that the programmer has to worry much less about implementing the basic Windows features, and concentrate more on implementing the application. This translates into less development and testing time, and fewer bugs.

The environment is a Multiple Document Interface (MDI) application using the Document/View class model [114]. The use of this model provides a multiple-window user interface with menu bars, toolbars, child windows etc. This will be described in greater detail below. The basic (two dimensional) visualization functions are enabled by the Windows General Drawing Interface (GDI), while three-dimentional visualization is made possible with the use of Direct-3D.

Interactive MoSART Environment Organization: Six Core Modules. The *Interactive MoSART Helicopter Environment* is organized into the following six core modules

- The Program User Interface Module (PUI),
- the Simulation Module (SIM),
- the Graphical Animation Module (GAM),
- the MATLAB Communication and Analysis Module (CAM),
- the Help/Instruct Module (HIM), and
- the Animation Laboratory (A-Lab) module.

Each module is now discussed.

3.1 Program User Interface Module (PUI)

The program user interface module (PUI) provides an interface between a user and the program. A sample screen dump of the interface is shown in Figure 3.1. Written in the MFC framework of Microsoft Visual C++ and standard Windows '95/'98/NT, the environment provides pull-down menus that permit the user to select system models, controllers, signal models, algorithms, and parameters. A user, for example, will be able to

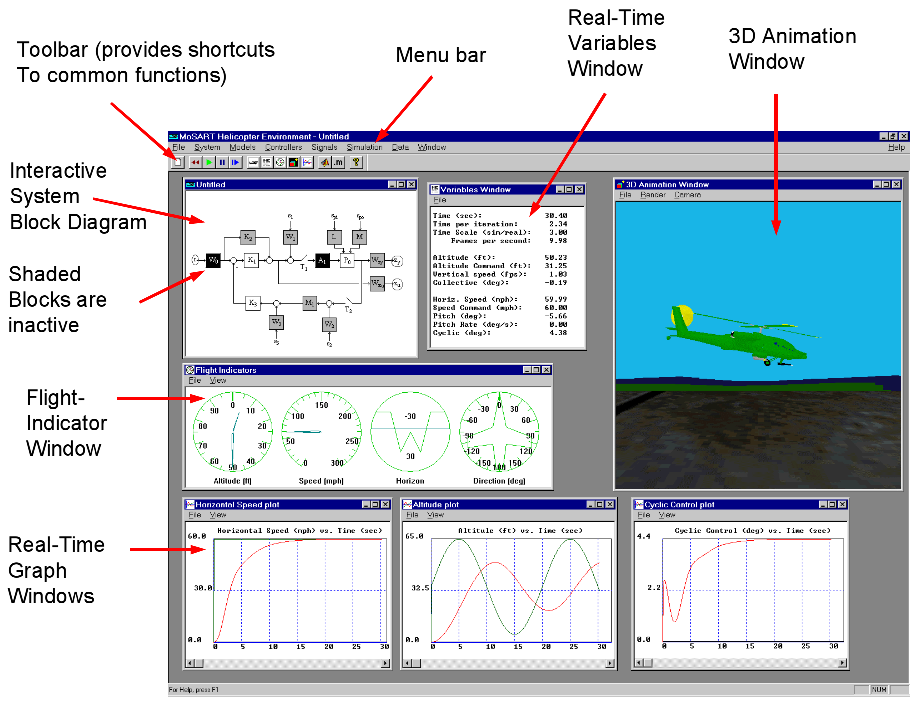
1. select/edit a simulation model,
2. select an animation model,
3. select/edit simulation reference commands, disturbances, and sensor noise (e.g. steps, sinusoids, etc.),
4. view/alter the simulation parameters (e.g. initial conditions, integration routine, integration step size, etc.)
5. select/alter control laws, control law parameters, and control law design parameters for control law design using MATLAB/Toolbox engines.

Menu options for data storage and plotting also exist. An active child window contains a block-diagram representation of the complete dynamical structure which has been implemented within the environment (see Figure 3.2). A user may edit parameters associated with any of the available components (i.e. systems and signals) by using the menus or by using the mouse to click on the block diagram. Common functions are also accessible through a floating/docking toolbar (Figure 3.3), which has a *VCR-style* control panel for controlling the simulation.

The following provides a brief description of menu items:

File

- **New.** Creates a new active document (i.e. simulation).
- **Generate Report.** Generates a report of the current simulation parameters.



```
Time (sec):                 30.40
Time per iteration:          2.34
Time Scale (sim/real):       3.00
     Frames per second:      9.98

Altitude (ft):              50.23
Altitude Command (ft):      31.25
Vertical speed (fps):        1.03
Collective (deg):           -0.19

Horiz. Speed (mph):         59.99
Speed Command (mph):        60.00
Pitch (deg):                -5.66
Pitch Rate (deg/s):          0.00
Cyclic (deg):                4.38
```

Figure 3.1: Visualization of the Program User Interface (PUI) - System Diagram, Real-Time Variable Display Window, Real-Time Graphics, 3D Animation, and Simulation Parameters Window

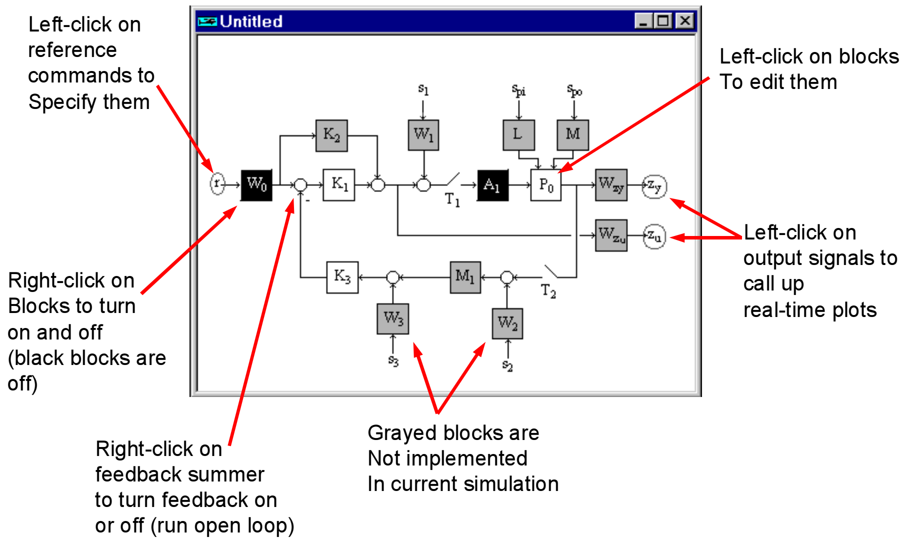


Figure 3.2: Visualization of the Program User Interface (PUI) - System Block-Diagram

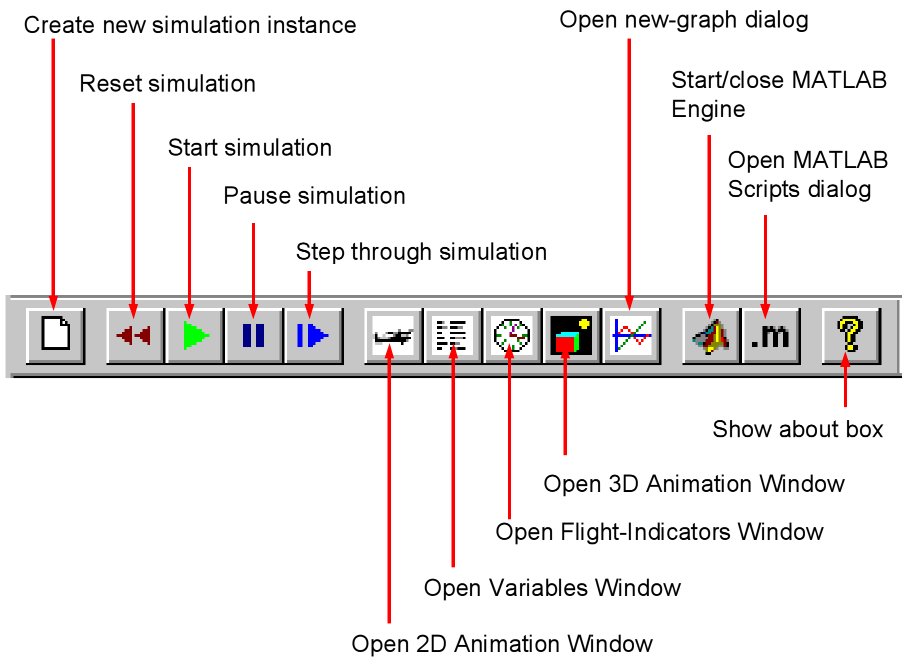
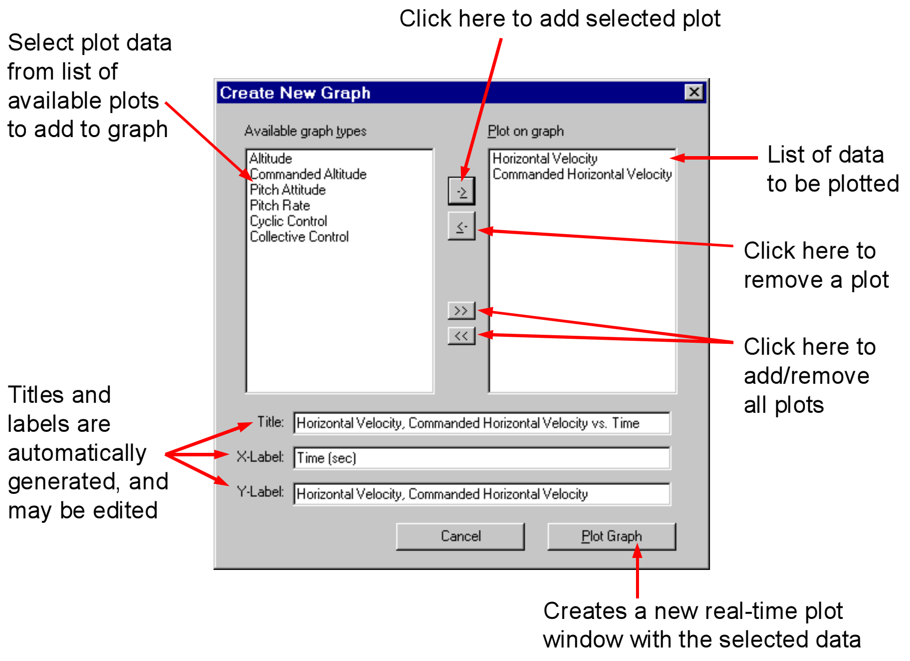


Figure 3.3: Visualization of the Program User Interface (PUI) - Floating/Docking Toolbar


Select plot data from list of available plots to add to graph
Click here to add selected plot
List of data to be plotted
Click here to remove a plot
Click here to add/remove all plots
Titles and labels are automatically generated, and may be edited
Creates a new real-time plot window with the selected data
Create New Graph
Cancel
Plot Graph

Available graph types
Altitude
Commanded Altitude
Pitch Attitude
Pitch Rate
Cyclic Control
Collective Control

Plot on graph
Horizontal Velocity
Commanded Horizontal Velocity

Title: Horizontal Velocity, Commanded Horizontal Velocity vs. Time
X-Label: Time (sec)
Y-Label: Horizontal Velocity, Commanded Horizontal Velocity

Figure 3.4: Visualization of the Program User Interface (PUI) - New Graph Dialog

- **Toolbar.** Shows or hides the floating/docking toolbar.
- **Status Bar.** Shows or hides the status bar.
- **Exit.** Quits the application.

System

- **Single Lift.** Used to select a specific system. Allows more system options to be added in the future.

Models

- **Actuator(A_1).** Permits a user to alter actuator parameters.
- **Plant(P_0).** Permits the user to select a plant model and to alter plant parameters.
- **Initial Conditions.** Permits user to edit plant state initial conditions. Includes a menu option for selecting initial conditions which correspond to a specific plant eigenvector - thus permitting the study of specific plant tendencies (i.e. natural modes),

Animations

- **Variables.** Open the variables window.
- **2D bitmap.** Open a 2D animation window.
- **3D Window.** Open a 3D animation window.
- **Flight Indicators.** Open the flight-indicators window.

Controllers

- **Series (K_1).** Edit the series controller.
- **Feedback (K_3).** Edit the series controller.
- **Proportional.** Implement a proportional controller.

- **Proportional-Integral.** Implement a proportional-integral controller.

Signals

- **Reference Commands.** Select reference command signal. The following options are available: joystick input, constant input, sine wave, square wave, and sawtooth wave.
- **Disturbances.** Select disturbance signal. Feature not implemented at this time.
- **Noise.** Select of noise signal. Feature not implemented at this time.

Simulation

- **Stop.** Pauses current simulation,
- **Restart.** Restarts current simulation,
- **Run.** Runs simulation,
- **Step.** Performs single simulation step,
- **Initial conditions.** Specify the initial conditions of the simulation.
- **Parameters.** The user can select the integration algorithm, simulation mode (real-time or faster than real-time), sampling time, graph resolution and simulation stop time.

Data

- **Load.** Load simulation data.
- **Save.** Save simulation data.
- **Variables.** Display variable window containing variable data about the simulation.
- **Plots.** Opens menu which lists available plotting options. Includes plotting of states, controls, tracking error on individual plots. Also includes a multi-plot option.

- **MATLAB Scripts.** Post-process current simulation model data via MATLAB scripts.
- **MATLAB Engine.** Activate or deactivate MATLAB engine.

Window

- **New Window.** Opens another window for the active document.
- **Cascade.** Arranges windows so they overlap.
- **Tile.** Arranges windows as non-overlapping tiles.
- **Arrange Icons.** Arranges icons at the bottom of the window.

Help/Instruct

- **Report.** Creates a report of the current model and simulation settings. This is useful for documenting/recreating complex simulation scenario.
- **Help topics.** Opens a list of available help topics.
- **Short Guide.** Opens the short guide to the *Interactive MoSART Helicopter Environment*.
- **About.** Display program information, version number and copyright.

3.2 Simulation Module (SIM)

The simulation module (SIM) was written in C++ and is optimized for speed and accuracy. It utilizes recursive algorithms which numerically solve the ordinary differential equations describing the plant, actuator, sensor dynamics, control law, and signal filters. When operating in linear-simulation mode, the engine exploits a matrix-algebra C++ class toolset specifically developed for this application. Users can select - via the PUI - from different integration methods (e.g. basic-Euler, 4th order Runge-Kutta, etc.), integration method parameters, control laws, control law parameters, control law design parameters for redesign using MATLAB engine, exogenous signals, and other parameters. More complex simulations may be developed that can take advantage of direct access

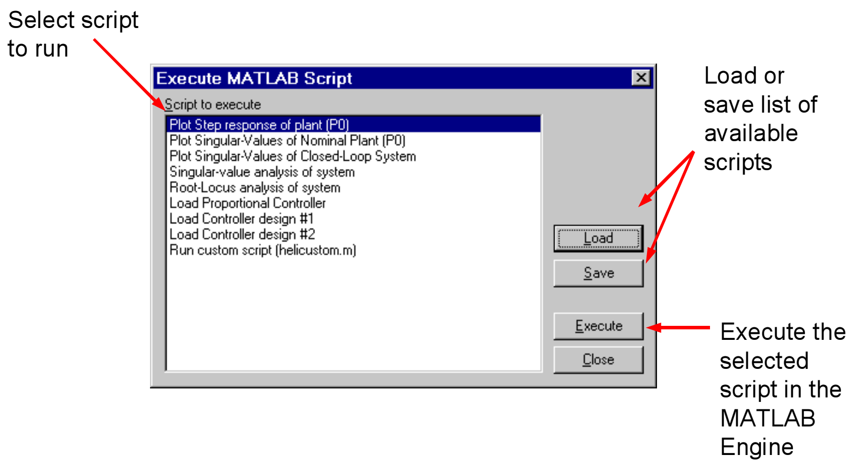

Select script to run
Execute MATLAB Script
Script to execute
Plot Step response of plant (P0)
Plot Singular-Values of Nominal Plant (P0)
Plot Singular-Values of Closed-Loop System
Singular-value analysis of system
Root-Locus analysis of system
Load Proportional Controller
Load Controller design #1
Load Controller design #2
Run custom script (helicustom.m)
Load
Save
Execute
Close
Load or save list of available scripts
Execute the selected script in the MATLAB Engine

Figure 3.5: Type Of Analysis That May Be Conducted From Within The *Interactive MoSART Helicopter Environment* By Accessing The MATLAB Engine And Toolboxes

to MATLAB 5.0 scripts and toolboxes using the environment's *MATLAB Engine Communication Link*. Figure 3.5 illustrates the type of analysis that may be conducted from within our *Interactive MoSART Environment* by accessing the MATLAB engine and toolboxes.

Simulations typically run faster than real-time but can be set to real-time using by selecting the fix-to-real-time environment feature.

Extensibliltiy via SIMULINK: A-Lab. Users can also create models (e.g. plant, actuator, sensors, controller, exogenous signals, etc.) in SIMULINK that can then be used to drive our animation models. This makes the environment very extensible in terms of simulation models. With the many systems under development by the ASU MoSART research team, this offers users a flexible *Animation Laboratory (A-Lab)*. A-Lab is described in more detail in Chapter 4.

3.3 Graphics/Animation Module (GAM)

The main purpose of the graphics/animation module (GAM) is to update graphics and animations using data provided by the simulation module. Data, graphs, visual indicators, and animations are displayed within child windows. The ability to visualize the simulation via various visual aids is a key feature of this environment. Several visual representations of the simulation are available to the user, including: a real-time variable display window, real-time graphing windows, and 3-dimensional animation windows.

Real-Time Variable Display Window. The Real-Time Variable Display Window shows a continuously updated list of the simulation state variables. Additional information such as the simulated-time/real-time ratio and the number of frames-per-second is also displayed.

Real-Time Graphing Windows. Real-Time Graphing Windows contain dynamically updated graphs for a user-specified simulation variable. Graphs scale automatically as the simulation progresses so that appropriate ranges are displayed. Graph titles are also displayed. Graphs may be interactively scrolled to focus on desired time ranges.

3-Dimensional Animation Windows. 3-Dimensional Animation Windows display animated real-time 3-dimensional graphics. The object elements are represented by texture-mapped light-shaded polygons. Users may specify a view-point and viewing perspective in real-time. This feature allows the viewer to quickly discern the spatial relationships of the animated objects. Animation is achieved using Microsoft Direct-3D [125]. The environment utilizes Direct-3D version 3.0 which is supported on Windows '95/NT. Direct-3D offers many features, such as

- advanced rendering options (e.g. Gouraud shading and Phong shading [125]),
- texture mapping, and

- a standard object-definition file-format.

Visual indicators are also provided for an enhanced simulation experience.

Animation Module Extensibility. Direct 3D objects (i.e. meshes) may be downloaded from many websites (for free). Extensive Direct 3D libraries exist. Direct 3D objects can also be created using a variety of mesh creation packages. 3D Studio, for example, can be used together with a (free) Microsoft Direct 3D conversion utility.

A MATLAB-based Direct 3D toolbox is currently under development at ASU [66].

Other Features. Microsoft DirectX provides: 3D-animation, sound, video, user-input, etc. Plans for exploiting each of these features are being considered.

3.4 MATLAB Communication and Analysis Module (CAM)

Communication with MATLAB. To enable communication (i.e. data exchange, etc.) between MATLAB and our *Interactive MoSART Environment* a MATLAB communication link was created [52]. This link is made possible by the MATLAB-Engine feature and exploits ActiveX technology.

3.5 Help/Instruct Module (HIM)

The help/instruct module (HIM) is intended to provide users with help on using the environemnt as well as instructional information regarding the underlying system models, control laws, and concepts. The module specifically gives users access to Hypertext-Markup Language (HTML) format documents, pdf documents (e.g. theses), environment tutorials, AVI-based demonstrations, MATLAB macros, SIMULINK diagrams, and other system-specific documentation which elaborates on models, control system design methodologies, algorithms, etc.

The above allows the creation of detailed on-line model/control system documentation, tutorials, and interactive lessons.

Environment Core Module	Associated Classes
Main application classes	RMApp, CHeliApp, CChildFrame, CMainFrame, CSplashDlg
Program User Interface Module (PUI)	CHeliView, CABCDDlg, CCommandsDlg, CCtrlrDlg, CCtrlr-PDlg, CGraphDlg, CInitDlg, CInputDlg, CMatlabScriptDlg, CNew-GraphDlg, CSimDlg, CStrNum
Simulation Module (SIM)	CSimDoc, CHeliDoc, CHeliSimData, CBlock, CCmdParam, CReal-Matrix, CfloatArray, CReport
Graphical Animation Module (GAM)	2D Animation: C2DAnimWnd, CAnimWnd, RMWin Variables: CVarWnd Flight-indicators: CFlightIndicatorWnd 3D Animation: D3DAnimWnd, CD3DObject, CD3DObjHeli, CD3DObjLand, CD3DObjShadowHeli Real-time graphing: CGraphList, CGraphDlg
Communication and Analysis Module (CAM)	CmlDataList, CmlScript, CmlScriptList
Help/Instruct Module (HIM)	C_HyperLinkControl

Table 3.1: The Main Classes Of The *MoSART Helicopter Environment* Application Organized by Fucntional Modules

3.6 Code Structure

The Visual C++ code for the *Interactive MoSART Helicopter Environment* has been developed using Microsoft Developer Studio 97/Visual C++ 5.0. It is organized in different source files. `heli32.dsw` is the project workspace file. This is the main file that is loaded into Visual C++ and contains information on all the files in the project and the required project settings. Even though there are many source files, only a few of them are specific to the *Interactive MoSART Helicopter Environment* application itself. The rest of them provide basic framework and general functionality such as matrix-algebra, MATLAB interfacing, data arrays, visual indicators, real-time plotting, and Direct-3D visual objects. Table 3.1 lists the main classes in the project, organized by their core module.

Tables 3.2 and 3.3 provide a description of each of the header files that is part of the *Interactive MoSART Helicopter Environment* project.

Header File (*.h)	File Description
2DAnimWnd	Handles the 2D Animation Window
_Hlink	Creation of links to the help documents (html, PDF...)
ABCDDlg	Handles dialog box for editing linear state space representation of systems
AnimWnd	Handles the animation window
Block	Class structure defining a state-space transfer function(i.e. a 'block' in the block-diagram)
ChildFrm	Child window (part of the MFC doc/view framework)
cmdparam	Class structure for defining a standard reference command signal
CommandsDlg	Handles dialog box for standard reference command signals
CtrlrDlg	Controller parameters dialog
CtrlrPDlg	Controller parameters dialog
d3dAnimWnd	Handles the Direct-3D animation window
D3DObj	Direct 3D Object parent class
Defines	Global definitions
FlightIndicatorWnd	Handles display of the flight-indicator window
FltArray	Maintains an array of floating-point numbers
globals	Global variables definition
GraphDlg	Handles standard Plotting
GraphList	Allows the creation and maintenance of multiple graph-plot windows. This class can: Call up a Create-New-Graph dialog, allowing users to specify multiple plots per graph; maintain a list of active graph windows; automatically update all windows with a call to `UpdateViews()`
GraphWin	Handles the plotting window
heli32	Main file for the application. It includes other project specific headers (including `Resource.h`) and declares the `CHeliApp` class
heliDoc	Simulation engine
HeliSimData	Holds simulation numerical output functions to save/load data to/from .MAT files
heliView	The view of the document. Handles the user interface
InitDlg	Handles dialog box that sets initial conditions for the states of the plant
intfunc	Integration routines used by the simulation engine (Supports Basic-Euler, Modified-Euler and 4th order Runge-Kutta)
MainFrm	Controls the multiple document interface frame features
matfunc	Functions to load and save CFloatArray data structures to MATLAB .mat files

Table 3.2: Description Of The Header Files That Are Part Of The Code

Header File	File Description
Matinv	Matrix inversion functions
MatlabScriptDlg	Handles the dialog box related to the communication with MATLAB
matparse	Parsing functions
MLDataList	Maintains a list of all 'MATLAB-Engine-able' matrices used in `CmlScript` for interactive MATLAB scripting
MLScript	Defines an interactive MATLAB script, which consists of: Variables to send to MATLAB Engine from C++ Environment, MATLAB script/command to run and variables to read back from MATLAB Engine to C++ Environment.
MLScriptList	Communication with MATLAB. List of scripts
NewGraphDlg	Handles Dialog box for multiple plotting
ObjHeli	Direct-3D Helicopter Object
ObjLand	Direct-3D Ground (terrain) Object
ObjShadowHeli	Direct-3D 'Shadow' Helicopter Object
RealMat	Matrix-algebra C++ class toolset specifically developed for this application
report	Creates a report of the current model and simulation settings
resource	Microsoft Developer Studio generated include file. Used by `RtGraph.rc`
RMapp	Contains the base class for a Direct3D specific application class.
RMwin	Contains the base class for a Direct3D specific window class.
RtGraph.rc	Listing of resources that the multi-plot library uses
SimDlg	Handles dialog box for setting the simulation parameters
SimDoc	Contains the generic simulation document class. Houses the simulation code that is general for all environments
SplashDlg	Handles the dialog box containing the introductory (splash) window
StdAfx	These files are used to build a precompiled header (PCH) file named `heli32.pch` and a precompiled types file named `StdAfx.obj`
StrNum	Handles string/number variables
util	Miscellaneous global utility functions
gfxutil	2D graphics functions (mainly for flight indicators)
VarWnd	Real-time variable window

Table 3.3: Description Of Additional Header Files That Are Part Of The Code

The *Interactive MoSART Helicopter Environment* was based on the code created by the MFC Multiple-Document Interface (MDI) Appwizard. The MFC Appwizard is a utility in Visual Studio that generates simple code templates for a programmer to start working with. The Appwizard generated the following classes: `CHeliApp`, `CHeliDoc`, `CHeliView`, `CMainFrame`, and `CChildFrame`. These classes form the foundation of the MFC Document-View application template. The Document-View template is discussed in detail within [122]. Essentially, this template separates the functionality of an application (the 'document') from its user interface (the 'view'). For the *Interactive MoSART Helicopter Environment* application, the 'document' consists of the simulation module (SIM), and the 'view' contains the program user interface (PUI), graphical animation module (GAM) and access to the other modules, namely the help-instruct module (HIM) and the MATLAB communication/analysis module (CAM). In addition to using the MFC framework, additional C++ class hierarchies were developed and implemented as necessary in order to structure the code and to make it modular. Under this scheme, classes are created to encapsulate separate program functionality, with derived child-classes possessing increasingly specific functionality than their parent classes. For instance, the `CAnimWnd` class encapsulates the basic functionality of an animation window. This class provides the bare minimum an 'animation window' must have in order to function in the program. Classes derived from `CAnimWnd` implement additional functionality. `C2DAnimWnd` implements a two-dimensional bitmapped-pictorial representation of the simulation for example, while `CFlightIndicatorWnd` implements a flight instrument panel representation. This is essentially an implementation of object-oriented inheritance and polymorphism properties. Basic OOP concepts such as these are discussed in any elementary text on C++ such as [118].

The functionality and code structure of each of the main modules of the *Interactive MoSART Helicopter Environment* (PUI, SIM, GAM, CAM, HIM) will now be discussed in detail.

3.6.1 Program User Interface Module (PUI) code

The main PUI functionality is contained within the class `CHeliView.` The application is based on the windows message-based event-driven model. This means that functions within a class are created in order to service user requests, and that these functions are tied to various user-interface elements via message maps. This model allows for multitasking. The simulation continues to run while there are no new user requests. When a new user request is made (for instance, the user clicks on a menu item), a message is generated and posted to the application. A message-map is a look-up table that maps a message to a particular function. The application uses the message map to call the proper service function (or message handler) in order to process the user's request. The MFC framework generates certain messages in response to events (for instance, ' `OnPaint()`' is called in response to a `WM_PAINT` message that is generated whenever Windows detects the need to redraw the window on the screen). Other handlers may be tied to events specific to the application (e.g. whenever the user clicks on a specific button) by the programmer. The Windows message-based event-driven architecture and common Windows messages are described in [121]. Most of the functions within `CHeliView` are message handlers. By convention, these are typically denoted with a function name beginning with ' `On`' (e.g. `OnRun()).` Most of these message handlers are self-explanatory. All of them have a descriptive comment before the function definition. In contrast to a procedural-based program, the main program loop is handled by the operating system. The application essentially just waits for the user to issue a command or for other system events.

The user may issue commands in several ways. Firstly, via the pull-down menu system. Secondly, via the toolbar. Thirdly, by clicking on the interactive block diagram. These three elements are shown in Figure 3.1. All three methods generate the same messages, and the correct functions are called in the same way irrespective of which method of input the user prefers. This is an example of the power of abstraction afforded by the message-based system. Additional means of user input may be added to the program in a modular and transparent manner.

Tables 3.4-3.5 list the main functions of `CHeliView.` The definitions for these functions may be found in `heliview.cpp`. The other classes associated with the PUI are: `CABCDDlg, CCommandsDlg, CCtrlrDlg, CCtrlrPDlg, CGraphDlg, CInitDlg, CInputDlg, CMatlabScriptDlg, CNewGraphDlg, CSimDlg, and CStrNum`. The classes whose name end in ' `Dlg`' are Dialog handlers. These classes encapsulate the data associated with a particular dialog interface. For example, CABCDDlg has functions that present a state-space representation of a block to the user and solicits changes to the A,B,C, and D matrices. `CStrNum` is a helper data class that allows numbers to be stored in string format, and is helpful in remembering previously entered user data. A description of each class may be found in Tables 3.2-3.3.

3.6.2 Simulation Module (SIM) code

The main functionality of the simulation module is contained within the classes `CSimDoc` and `CHeliDoc`. `CSimDoc` is the parent class that contains general simulation functions, while `CHeliDoc` is derived from it and contains simulation functionality specific to the helicopter simulation. `CSimDoc` also contains several virtual functions (these are functions that are declared in the parent class, and must be implemented in all child classes). This serves as a template for the system-specific simulation child class. The purpose of implementing this structural hierarchy (here, and in other parts of the program) is to increase code modularity and maintainability.

The simulation parameters (i.e. the data defining the block diagram) are stored in a file named ' `heli_default.mat`' and are loaded at startup in the function `InitMatrices()`. This function is called automatically at startup by the MFC framework via the parent class `CSimDoc`. During run-time, the simulation parameters are stored in various `CBlock` data-types that are declared as member variables in `CSimDoc`. `CBlock` is a data class that stores the A, B, C, D, X, and U matrices ([A,B,C,D] is the state-space representation of the block, X is the current state, and U is the current input). The simulation itself is run in the function `Step()` in `CHeliDoc`.

Each call to the function `Step()` performs 0.1 seconds worth of simulation. This may consist of one or more iterations, depending on the specified integration step. The basic flow of `Step()`

Function	Description
DoMatlabScriptDlg()	Open the 'run script' dialog
EditBlock()	Edit a block in state space format
InitDataList()	Initialize Data-lists and Script-lists. Default script definitions go here
InitGraphList()	Initialize list of available plots
InitJoy()	Initialize joystick
OnComparam()	Open reference command specifier dialog
OnEditCtrlP()	Edit Proportional Speed Controller
OnEditCtrlr()	Edit Proportional-Integral Speed Controller
OnEditK1()	Edit K1 block in state-space form
OnEditK3()	Edit K3 block in state-space form
OnEditW0()	Edit W0 block in state-space form
OnEditPlant()	Edit Plant block in state-space form
OnInitCond()	Opens initial conditions dialog box
OnInitialUpdate()	Called for first initialization
OnJoyChange()	Read updated joystick status
OnLButtonDown()	Check if user clicked on interactive block diagram
OnLoadSimData()	Load previously saved data to be overlaid on current simulation
OnMatlabEng()	Initialize or close the 'global' MATLAB Engine
OnMatlabScripts()	Open MATLAB scripts dialog
OnMultiPlot()	Open 'create new plot' dialog
OnNew2dWnd()	Open new 2D Bitmap animation window

Table 3.4: Main Functions of `CHeliView` (part 1)

Function	Description
OnNew3dWnd()	Open new Direct-3D animation window
OnNewFlightIndicatorWnd()	Open new Flight Indicator animation window
OnNewVarWnd()	Open new Variables window
OnRButtonUp()	Check if user clicked on interactive block diagram
OnReport()	Generates a report on the current simulation and views it with notepad
OnReset()	Reset simulation, loading initial conditions
OnRun()	Start simulation or continues if paused
OnSaveAll()	Save current simulation data to .mat file
OnShowGuide()	Opens the ‘short guide’ in WinHelp
OnSimParam()	Opens simulation parameters dialog
OnStep()	Perform one simulation iteration
OnStop()	Halts (pauses) current simulation
OnTimer()	Perform simulation step if running, update views
UpdateViewData()	Updates the list of arrays to plot for each plot
UpdateViews()	Call all child windows (i.e. animation windows) to redraw

Table 3.5: Main Functions of `CHeliView` (part 2)

proceeds as follows:

1. Update the states and blocks (`UpdateMatrices()`) from any parameter changes.
2. Compute the number of iterations to make based on the integration step-size.
3. Perform the following loop that number of times:
   (a) Take a 'snapshot' of the simulation data, and update the data arrays by calling `UpdateArrays()`.
   (b) Update the reference commands (from user input or function generator), by calling `UpdateCommands()`.
   (c) Sequentially compute the output of each block given its input (via a call to `sstf()`, the output of each block is used as the input to the next one in series with it.

The function `sstf()` is a wrapper function that calls the actual integration function based on the chosen integration method. By default, a 4th-order Runge-Kutta algorithm is used, in the function `sstf_rk4()`. Other integration routines are also available, including basic Euler and Modified-Euler. Due to this modularity, additional integration algorithms may be easily added to the environment without having to change the basic procedure in `Step()`.

All the main functions in `CHeliDoc` are summarized in Table 3.6. All the main functions in `CSimDoc` are summarized in Table 3.7.

The main 'heartbeat' of the program is governed by the Windows timer. When the simulation is started, this timer is programmed to call the `OnTimer()` function in `CHeliView` at regular intervals (10 times a second). If the simulation is fixed to real-time, then the `Step()` function is called once to simulate 0.1 seconds worth of simulation time, thus synchronizing the simulation time to real-time. If the simulation is set to 'run as fast as possible', the `Step()` function is called repeatedly as long as no other messages are pending. In this way, the simulation runs as fast as the computer hardware allows. The `OnTimer()` function is also used to update the user interface, redrawing plots, animations, etc. This synchronizes the display to the simulation as well.

Function	Description
InitMatrices()	Load matrices from file (`helidefault.mat`), initialize signal vectors
LoadNewSimData()	Load previously saved data and overlay it on current simulation. The overlaid data will be shown on plots and as shadow helicopters in the 3D Animation Window
ResetSim()	Re-initializes the simulation variables
SaveAlltoMatFile()	Saves the current simulation data to a `.mat` file
SetBlockActive()	Activates or deactivates a block
SimWhileIdle()	Repeatedly calls Step() as long as no other messages are pending
sstf()	Calls appropriate integration routine based on the one selected
Step()	Performs one integration step
UpdateArrays()	Adds current simulation data to capture buffer
UpdateCommands()	Updates the current reference commands (set point)
UpdateMatrices()	Updates the blocks based on parameter changes
EditBlock()	Edit a block in state space format

Table 3.6: Main Functions of `CHeliDoc`

Function	Description
Destroy()	Deallocates allocated memory in preparation for program termination
LoadFromMatFile()	Loads simulation parameters (all the blocks) from a .mat file
OnNewDocument()	Called by the MFC framework on startup. Calls `InitMatrices()` and `ResetSim()`
SaveMatAll()	Let's user specify a file, and saves simulation data to a `.mat` file by calling `SaveAlltoMatFile`

Table 3.7: Main Functions of `CSimDoc`

3.6.3 Graphical Animation Module (GAM) code

The GAM implements the visualization features of the *Interactive MoSART Helicopter Environment*. These include: real-time plots, the variables window, the flight-indicators window, the 2D animated bitmap window, and the 3D animation window. All of the two-dimensional display windows utilize the Windows Graphics Device Interface (GDI). This is a set of Application Programming Interfaces (API) that provide a library of basic two-dimensional drawing functions. These APIs are built into Windows and are common to all versions of the operating system

The operation of each of these different visualization features in the GAM will now be discussed.

Real-time plots. The real-time plotting capability is encapsulated in the class `CGraphDlg.` One or more real-time plot windows may be open at the same time, each plotting one or more different variables. The class `CGraphList` manages the list of currently active real-time plot windows. The user-interface to creating a new graph is encapsulated in the class `CNewGraphDlg.` This class presents the user with a list of available simulation data to be plotted (Figure 3.4. The user may select one or more plots to be displayed on the same graph. The list of available graphs are stored as pointers in `CGraphList,` and are initialized at startup in the `InitGraphList()` function in `CHeliView.` The main functions in `CGraphList` are listed in Table 3.8. The main entry-point is the function `CreateNewGraph.` Calling this function displays the new-graph dialog box, allows the user to specify the data to plot, and opens a new real-time plot window with the specified data. The PUI (`CHeliView`) periodically calls `UpdateViews()` in `CGraphList` to refresh the plots when new data is added.

The operation of the `CGraphDlg` class is now discussed. This class is responsible for plotting the specified data arrays on the screen in a window. These windows automatically scale the vertical axis depending on their size, and allow the user to scroll the vertical offset. In order to present the user with a flicker-free plot, a double-buffer is used. If the plot were drawn directly to the screen

Function	Description
AddGraphType()	Make a data array available for plotting
CreateNewGraph()	Opens a new-graph dialog and allows the user to specify the data to plot
DestroyAll()	Deallocates memory and closes all open real-time-plot windows
UpdateViews()	Redraws all open windows

Table 3.8: Main Functions of `CGraphList`

Function	Description
AddFunc()	Adds a data array to be plotted
Init()	Initializes the window and opens it
DrawGrid()	Internal function to draw the auto-scale grid
OnPaint()	Called by the MFC framework to update the display
PlotGraph()	Internal function to perform the plotting

Table 3.9: Main Functions of `CGraphDlg`

buffer, this would cause the user to see incomplete images of the plot, because the screen refresh rate (typically 60 Hz or 70 Hz) is faster than the GDI can draw the new plot. Each time the display needs to be updated, the new data is drawn to a hidden buffer. When the drawing is complete, the hidden buffer is copied to the screen in a single pass, thus eliminating visible flicker. This is done in the function `OnPaint()` that is called automatically by the MFC framework whenever the screen needs to be updated. The use of a hidden buffer is used in all two-dimensional graphics displays within the environment, including in the flight-indicator window and the 2D animation window.

Variables window. This window is used to display key textual data about the current simulation. It displays the instantaneous values of the important quantities (e.g. altitude, horizontal speed, etc), the current simulation time, as well as the simulation/real time scale. This last quantity is an indication of how fast the simulation is running relative to real-time and is dependent on the computer hardware, the integration algorithm, the integration time-step, and the number of open windows.

Function	Description
DrawDial()	Internal function that draws a dial on the given device context
DrawDirection()	Internal function that draws a direction indicator on the given device context
DrawHorizon()	Internal function that draws a horizon indicator on the given device context
DrawSpeedometer()	Internal function that draws a speed indicator on the given device context
Init()	Initializes and opens the window
OnPaint()	Draws the instrument panel in the window

Table 3.10: Main Functions of `CFlightIndicatorWnd`

The code for the variables window is located in the class `CVarWnd`. The main functionality of this class is located in its `OnPaint()` function.

Flight-indicators window. This window displays the simulated instruments on the cockpit instrument panel. The code resides in the class `CFlightIndicatorWnd`. The main functions are listed in Table 3.10. This class also uses basic graphics functions found in the file `gfxutil.cpp`.

The flight-indicators window displays the following indicator dials: an altimeter (for altitude), a speedometer (for forward speed), an artificial horizon indicator (for roll and pitch), and a direction indicator (for yaw).

2D animation window. The 2D animation window implements a bitmap representation of the helicopter. Different modes of visualization may be selected by the user, including a trajectory-view and a polar-view. The trajectory view plots the flight-path of the helicopter on a two-dimensional longitudinal-vertical grid. The polar view shows the pitch attitude of the helicopter. The functionality of this window is coded in the class C2DAnimWnd. The main functions of `C2DAnimWnd` are listed in Table 3.11. At startup, the main helicopter bitmap is loaded from a resource (`IDB_HELI00`). This bitmap is then rotated to regular angles and the rotated versions are stored in memory. This is

Function	Description
DrawBigPolarGrid()	Draws a polar grid on the screen
GetHeliBmp()	Selects the corrent bitmap based on the helicopter pitch attitude
Init()	
OnPaint()	

Table 3.11: Main Functions of `C2DAnimWnd`

done in the function `InitGraphics()` in the class `CHeliApp`. Performing this pre-rotation speeds up execution when the simulation is run. For the 2D animation, the correct bitmap is selected based on the pitch attitude of the helicopter by the function `GetHeliBmp`. The procedure of translating the simulation state into the current animation frame is performed in the `OnPaint()` function.

3D animation window. Microsoft Direct-3D is a highly optimized set of graphics routines that can be used to display high-quality three-dimensional polygon-based graphics. The Direct-3D library is freely available, and will even be shipped with future versions of windows (e.g. Windows 2000). The *Interactive MoSART Helicopter Environment* uses Direct-3D in its 3D Animation window.

Direct-3D is part of the Direct-X library (from Microsoft). The Direct-X library provides functions that allow programmers to directly access (thus the name Direct-X) many low-level multimedia features that are available on the computer. These are organized into subset libraries that include support for sound/music (DirectSound), input devices such as joysticks and flight-yokes (DirectInput), networking (DirectPlay), and graphics (DirectDraw for 2D and Direct-3D for 3D).

Direct-3D competes with several other graphics libraries, such as OpenGL. OpenGL was developed for SGI UNIX workstations and has been ported to the PC-Windows platform. In contrast, Direct-3D was developed specifically for Windows, and provides much better features to the programmer. OpenGL was initially used as the graphics engine for the environment, but was dropped in favor of Direct-3D when the new version (version 3.0 at that time) proved superior in the amount of features while being comparable in performance (rendering speed). Before the advent of these

Function	Description
AddSimData()	Adds a (previously saved) simulation data set used to animate shadow-helicopters overlaid on the current animation
CreateScene()	Sets up the virtual 3D 'scene' by creating and positions the 3D elements including meshes, bitmaps, lights, and the camera.
Init()	Creates, initializes and opens the 3D animation window
SetDof()	Passes the current position (degrees of freedom) of the simulated helicopter to the animation window
UpdateCamera()	Update the camera position - to allow tracking, etc.

Table 3.12: Main Functions of `D3DAnimWnd`

graphics libraries, incorporating 3D graphics into an application required a programmer to develop basic 3D graphics functions themselves. Direct-3D provides a framework for implementing these basic operations, and allows the programmer to work at a 'scene' level. This means that the programmer only has to worry about the positioning of the 3D graphics elements in the virtual 3D 'scene', and not have to worry about the actual rendering process that is taken care of by Direct-3D. Since the development work for the *Interactive MoSART Helicopter Environment* was done on a Windows NT 4 platform, only Direct-3D version 3 could be used. The latest versions of Direct-3D (currently version 7) will be available on the next operation system upgrade (Windows 2000) or on Windows 95/98 platforms. However, all the required functionality is present in version 3, and so this did not pose a major issue.

The functionality of the 3D animation window is found in the class `D3DAnimWnd.` The main functions of this class are listed in Table 3.12. The function `CreateScene()` is where the 'scene' of the helicopter 3D animation is defined. In this function, the individual elements of the scene are created and placed (like actors and props) in the virtual 3D stage. These elements include the helicopter itself, a ground/terrain, shadow helicopters (if they have been activated), and other background elements. Each of these elements are encapsulated within a separate class that is derived from the `CD3DObject` class.

Function	Description
CreateScene()	Creates the elements of this object and adds them to the specified scene (virtual function to be defined in child classes)
SetDof()	Sets the state of the object (generally the position)
SetQuality()	Sets the rendering mode for this object

Table 3.13: Main Functions of `CD3DObject`

The `CD3DObject` class encapsulated the basic functionality that is common to all the Direct-3D elements. The main functions of this class are listed inTable 3.13. This class forms a parent class for other classes that encapsulate specific 3D object elements.

The object elements and their classes used in the *Interactive MoSART Helicopter Environment* are as follows:

- a helicopter object (`CD3DObjHeli),`
- a ground/terrain object (`CD3DObjLand),` and
- a shadow-helicopter object (`CD3DObjShadowHeli).`

The specific implementation for each of the above classes are defined in their `CreateScene()` functions. When the scene is created in the `CreateScene()` function of `D3DAnimWnd,` each of the `CreteScene()` functions for each element in the scene are called to build it up. Animation is performed by calling the `SetDof` function for the main helicopter object in order to update its position based on the current state of the simulation (obtained from the SIM in the class `CHeliDoc).` This is done in the OnPaint() function in `D3DAnimWnd,` which is called periodically by the OnTimer() function in CHeliView.

As with the emphasis on object-oriented programming elsewhere in the code, the implementation of a hierarchy of 3D object classes makes it very easy to add multiple instances of a particular object (e.g. have one or more shadow helicopters on the screen at once) and to define new types of 3D object elements. New types of objects are defined by creating new classes derived from `CD3DObj` and using them in `D3DAnimWnd.` Existing elements can be easily modified or even replaced with new objects

Function	Description
AddSsType()	Maps a `CBlock` variable to a data variable
AddVarType()	Maps a `CRealMatrix` variable to a data variable
Clear()	Clears the current list
Create()	Creates a new list
GetData()	Retrieves a data variable from the specified MATLAB engine
SendData()	Sends a data variable to the specified MATLAB engine

Table 3.14: Main Functions of `CmlDataList`

(e.g. different helicopter object) with minimal changes to other parts of the code. Object-oriented programming allows the programmer to organize the code logically and efficiently.

3.6.4 MATLAB Communication/Analysis Module (CAM) code

The CAM allows the *Interactive MoSART Helicopter Environment* to interactively communicate with MATLAB at run time. This is made possible through the use of the 'MATLAB Engine' functionality of MATLAB. The use of this feature requires that MATLAB 5 be installed on the computer.

The CAM works by maintaining a list of scripts that can be run from within the environment. Each of these scripts consists of a list of variables to send to the MATLAB workspace, a command or an m-file to execute, and a list of variables to read back from the workspace. This mechanism allows arbitrary operations to be performed on the current simulation parameters.

The list of variables that can be communicated and operated upon in this way is maintained by the `CmlDataList` class. The main functions of this class are listed in Table 3.14. the function `InitDataList` in `CHeliView` initializes the list of data variables. The `CmlDataList` class also has functions to transfer data variables to and from the MATLAB engine.

The `CmlScript` class defines a particular operation script that can be executed. The main functions of this class are listed in Table 3.14. The `InitDataList` function in `CHeliView` also

Function	Description
AddToGetList()	Adds a variable to the list of variables to be retrieved from the engine after the script is executed
AddToSendList()	Adds a variable to the list of variables to be sent to the engine before the script is executed
Clear()	Clears this script
Execute()	Sends the 'send' list of variables to the engine, executes the command, and then retrieves the 'get' list of variables from the engine
SetCommand()	Specifies the command (or m-file) to be executed in the engine
SetDataList()	Specifies the data list object to use

Table 3.15: Main Functions of `CmlScript`

Function	Description
AddScript()	Adds a script to the list
Clear()	Clears the list
Create()	Creates the list
SetDataList()	Specifies the data list object to use

Table 3.16: Main Functions of `CmlScriptList`

initializes a list of scripts that may be executed.

The `CmlScriptList` class is used to maintain the list of available scripts. The main functions of this class are listed in Table 3.16. After the scripts are created, `AddScript()` is called to populate the list.

The class `CMatlabScriptDlg` is the part of the PUI that provides a user interface to the CAM.

3.6.5 Help/Instruct Module (HIM) code

The program help is available as a help file (`heli32.hlp`) that is accessible via the 'help' menu. This file was created using Wordpad in the rich-text format (`rtf`) and compiled into a help file with Visual Studio. This file provides basic information about the program, and a reference to common features.

The `C_Hlink` class allows the creation of direct links to Hypertext-Markup Language (HTML) format documents. This allows other external documents to be linked to the environment that may be called up by the user.

3.7 Utility of the Environment: Open-Loop Control and Command Following

In this section, the utility of the developed *Interactive MoSART Helicopter Environment* as a research and educational tool will be demonstrated. Our discussion will focus on issues related to controlling the horizontal speed of a hovering helicopter - in our case, Sikorsky's UH-60 Blackhawk. Specifically, the following will be demonstrated within this section: A helicopter is difficult to maneuver without an automatic control system because helicopters are open-loop unstable, and a horizontal speed control system may significantly improve a pilot's ability to control and maneuver the helicopter.

Open-Loop Instability and Horizontal Speed Control Via Joystick. In this section, it is shown how difficult it is to control a helicopter's speed by issuing direct cyclic control inputs via joystick. Two specific scenarios are considered: a horizontal speed perturbation and a horizontal speed hold maneuver.

Backflapping Instability: Horizontal Speed Perturbation. As noted in Chapter 2, the helicopter's horizontal dynamics are open-loop unstable. This is demonstrated with the MoSART helicopter environment as shown in Figure 3.6. In this simulation, the feedback controller system is disabled, and a small initial velocity of 3 ft/sec is introduced. The resulting speed and pitch attitude plots show that the magnitudes of the oscillations grow with time as expected - given the presence of the unstable backflapping mode.

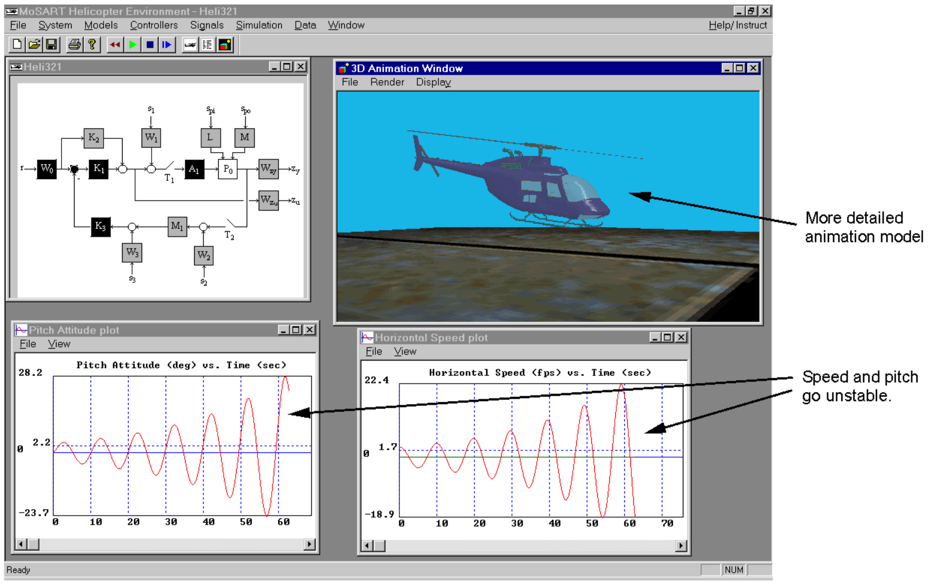

MoSART Helicopter Environment - Heli321
File System Models Controllers Signals Simulation Data Window
Help/ Instruct
Heli321
3D Animation Window
File Render Display
More detailed animation model
Pitch Attitude plot
Pitch Attitude (deg) vs. Time (sec)
Horizontal Speed plot
Horizontal Speed (fps) vs. Time (sec)
Speed and pitch go unstable.
Ready
NUM

Figure 3.6: Visualization of Unstable Backflapping Mode: Speed Pertubation

Open-Loop Control: Pitch-Hold Maneuver. The presence of the unstable backflapping mode makes it very difficult to control the helicopter open-loop. What if the helicopter started out with zero horizontal speed and pitch-attitude and we wished to pitch forward to some constant pitch-attitude and maintain that pitch attitude in forward flight? An attempt to do this by issuing cyclic control signals via a joystick is shown in Figure 3.7. The response is highly oscillatory due to the backflapping tendencies of the helicopter. While it is possible to stabilize the system in this manner, it is impossible to achieve the desired pitch or speed response.

The above illustrates clearly that a helicopter is difficult to maneuver by issuing cyclic commands directly via joystick; i.e. without the assistance of an automatic control system. Given this, it is clear that a closed-loop feedback control system is essential to make the helicopter easier to maneuver and significantly alleviate pilot workload.

Closed-Loop Pitch Control System. The dynamical feedback pitch-hold controller described in Chapter 2 is now applied to a pitch-attitude command maneuver. Figure 3.8 shows the response of the closed-loop system with the pitch-attitude-hold controller to a pitch command of 5 *deg*. The pitch response is overdamped, and all the responses are smooth and reasonable.

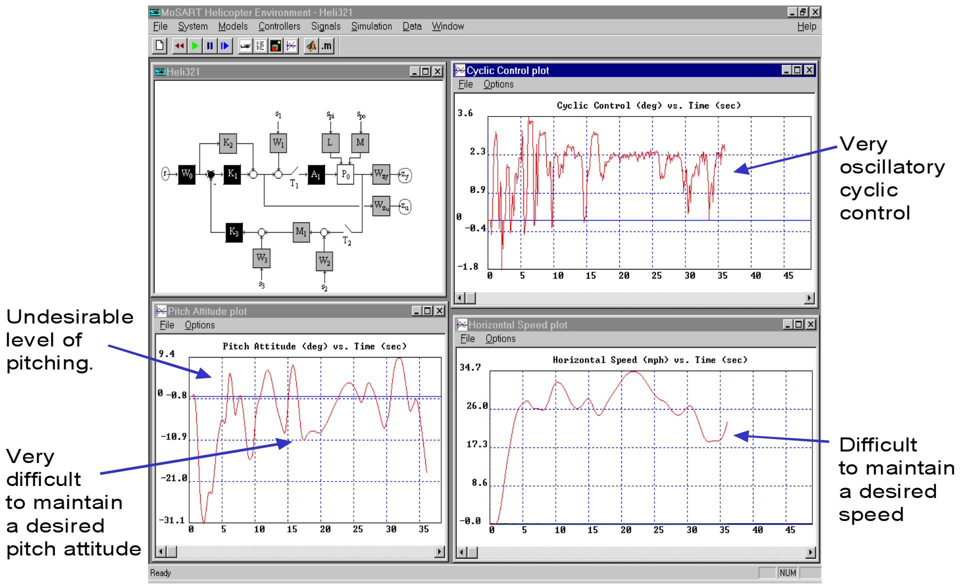

MoSART Helicopter Environment - Heli321
File System Models Controllers Signals Simulation Data Window Help
Heli321
Cyclic Control plot
Cyclic Control (deg) vs. Time (sec)
Pitch Attitude plot
Pitch Attitude (deg) vs. Time (sec)
Horizontal Speed plot
Horizontal Speed (mph) vs. Time (sec)
Very oscillatory cyclic control
Undesirable level of pitching.
Very difficult to maintain a desired pitch attitude
Difficult to maintain a desired speed
Ready
NUM

Figure 3.7: Open-Loop Joystick Control of Helicopter: Speed-Hold Maneuver

Closed-Loop Horizontal Speed Control System. Another common maneuver is a speed-hold acceleration command. The dynamical feedback speed-hold control system described in Chapter 2 is now activated. Figure 3.9 shows the response of the closed-loop system to an acceleration command from rest to 60 mph. This command was issued from a user-controlled joystick. The acceleration is smooth and the pitch response does not produce large pitch-rates.

The above examples clearly show how a feedback control system can improve the helicopters maneuverability and significantly alleviate pilot workload.

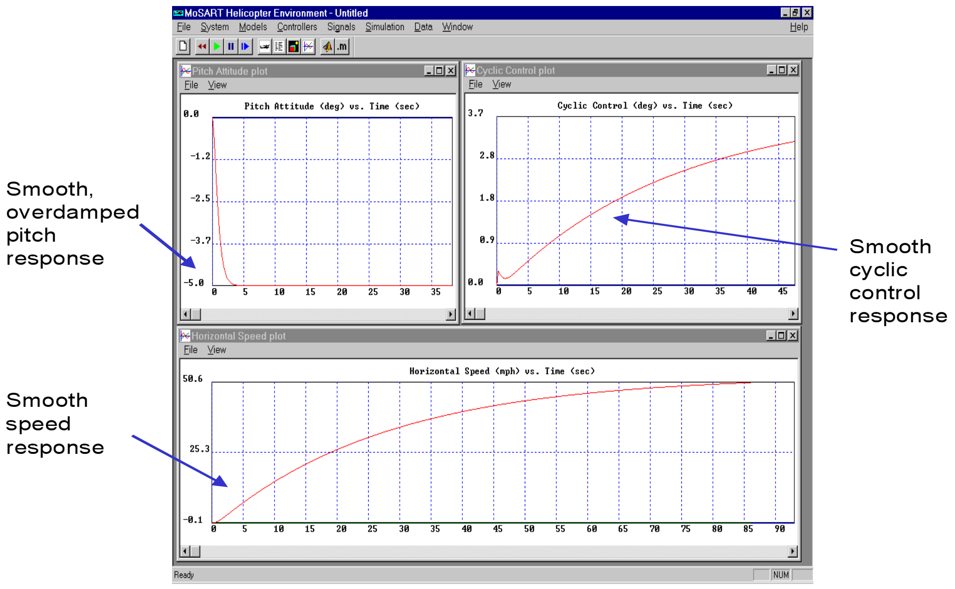

MoSART Helicopter Environment - Untitled
File System Models Controllers Signals Simulation Data Window Help
Pitch Attitude plot
Pitch Attitude (deg) vs. Time (sec)
Cyclic Control plot
Cyclic Control (deg) vs. Time (sec)
Horizontal Speed plot
Horizontal Speed (mph) vs. Time (sec)
Smooth, overdamped pitch response
Smooth cyclic control response
Smooth speed response
Ready
NUM

Figure 3.8: Pitch-Attitude-Hold maneuver: Response to 5 *deg* Pitch-Attitude Step Command

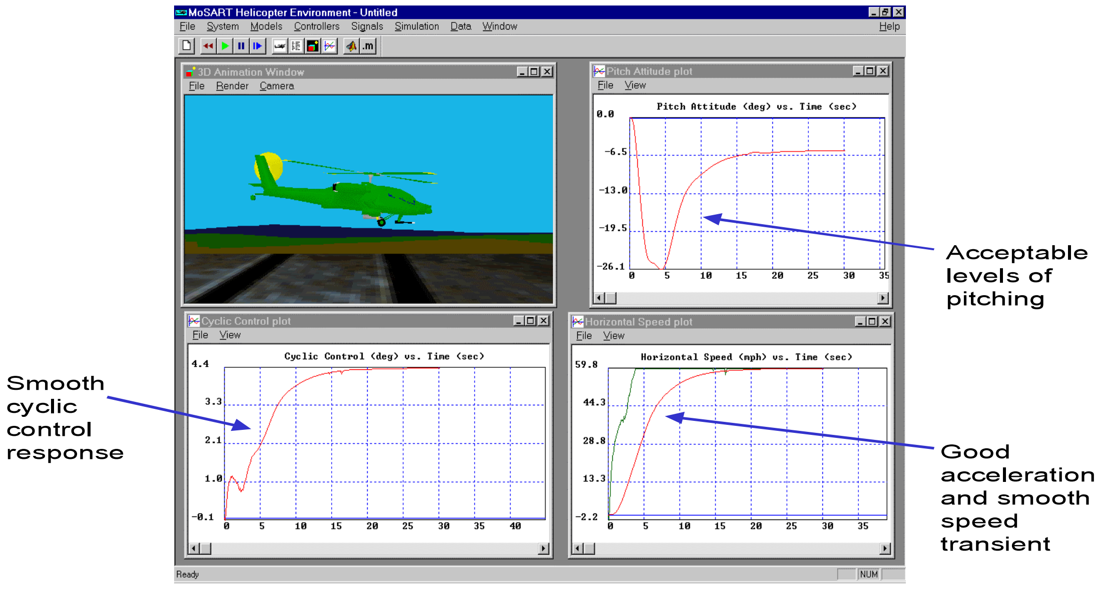

MoSART Helicopter Environment - Untitled
File System Models Controllers Signals Simulation Data Window Help
3D Animation Window
File Render Camera
Pitch Attitude plot
Pitch Attitude (deg) vs. Time (sec)
Cyclic Control plot
Cyclic Control (deg) vs. Time (sec)
Horizontal Speed plot
Horizontal Speed (mph) vs. Time (sec)
Acceptable levels of pitching
Smooth cyclic control response
Good acceleration and smooth speed transient
Ready
NUM

Figure 3.9: Speed-Hold acceleration maneuver: Response to User's 60 mph Speed Command

CHAPTER 4

Animation Lab (A-Lab)

4.1 Introduction

In order to provide complete model extensibility, A-Lab was developed to allow the animation features of the *Interactive MoSART Helicopter Environment* to be used with any simulation model developed in SIMULINK (the graphical simulation package by Mathworks [105]). A-Lab is a library of SIMULINK blocks that can be used to transform any SIMULINK simulation into a real-time animation environment. The A-Lab block library is shown in Figures 4.1 and 4.2.

A typical A-Lab enabled SIMULINK block diagram is depicted in Figure 4.3. The three A-Lab blocks that are used are:

- A Real-Time Governer (RTG),
- a Joystick Input Block (JIB), and
- An Animation Enabler Block (AEB).

The RTG limits the speed of the simulation to real-time, allowing accurate real-time user input and visualization. The JIB allows the user to specify inputs to the simulation (e.g. reference commands, disturbances, control signals, etc.) in real-time. The AEB allows the state of the simulation (i.e. helicopter position) to be visualized in real-time.

The functionality and design of each of the A-Lab components will be described in the following section.

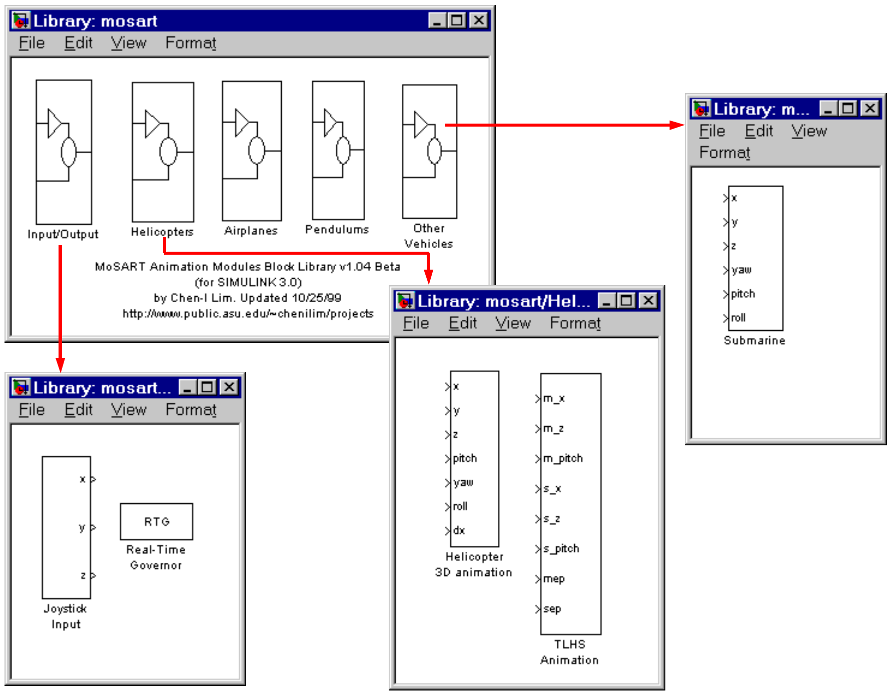


Figure 4.1: Visualization of the MoSART A-Lab SIMULINK library (part 1)

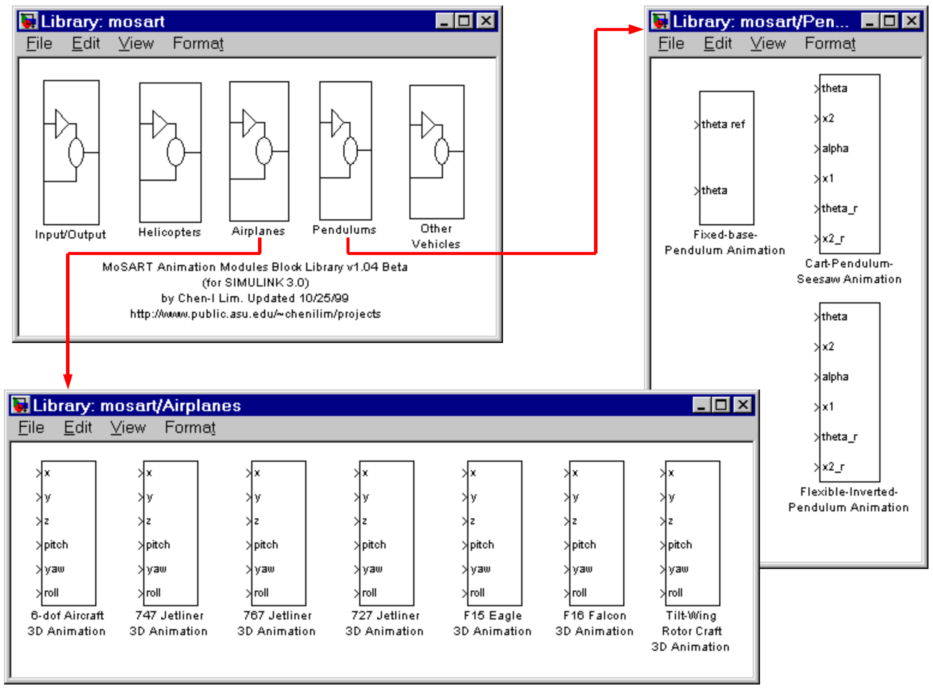

Library: mosart
File Edit View Format
Input/Output
Helicopters
Airplanes
Pendulums
Other Vehicles
MoSART Animation Modules Block Library v1.04 Beta
(for SIMULINK 3.0)
by Chen-I Lim. Updated 10/25/99
http://www.public.asu.edu/~chenilim/projects
Library: mosart/Pen...
File Edit View Format
theta ref
theta
Fixed-base-Pendulum Animation
theta
x2
alpha
x1
theta_r
x2_r
Cart-Pendulum-Seesaw Animation
theta
x2
alpha
x1
theta_r
x2_r
Flexible-Inverted-Pendulum Animation
Library: mosart/Airplanes
File Edit View Format
x
y
z
pitch
yaw
roll
6-dof Aircraft 3D Animation
747 Jetliner 3D Animation
767 Jetliner 3D Animation
727 Jetliner 3D Animation
F15 Eagle 3D Animation
F16 Falcon 3D Animation
Tilt-Wing Rotor Craft 3D Animation

Figure 4.2: Visualization of the MoSART A-Lab SIMULINK library (part 2)

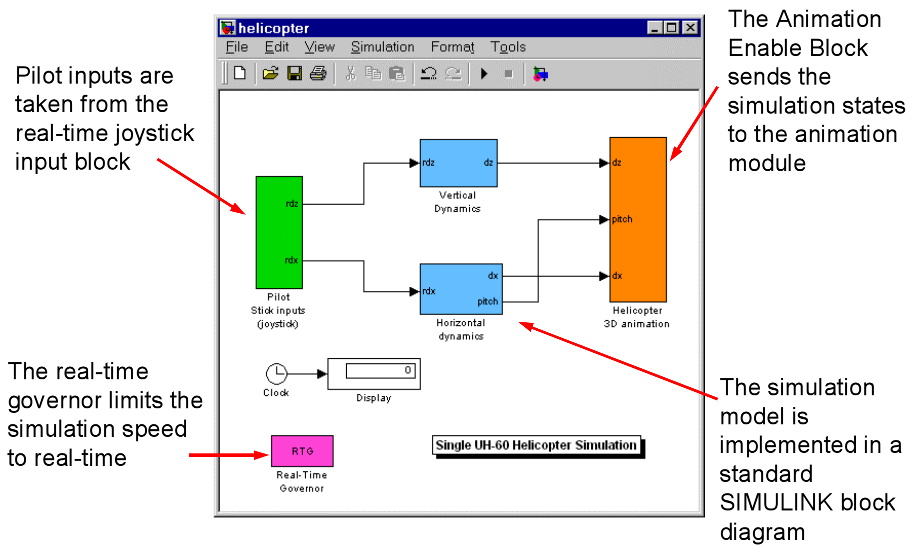

Pilot inputs are taken from the real-time joystick input block
The real-time governor limits the simulation speed to real-time
helicopter
File Edit View Simulation Format Tools
rdz
rdx
Pilot Stick inputs (joystick)
rdz dz
Vertical Dynamics
rdx dx pitch
Horizontal dynamics
dz pitch dx
Helicopter 3D animation
Clock
0
Display
RTG
Real-Time Governor
Single UH-60 Helicopter Simulation
The Animation Enable Block sends the simulation states to the animation module
The simulation model is implemented in a standard SIMULINK block diagram

Figure 4.3: Visualization of an A-Lab enabled SIMULINK block diagram

4.2 Code Structure

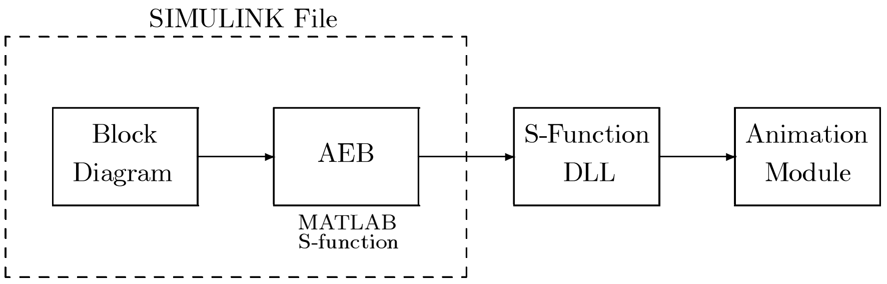


Figure 4.4: Visualization of SIMULINK/Animation-Module Communication

The A-Lab system for a particular environment consists of the following components (shown in Figure 4.4:

1. A SIMULINK block diagram of the system model,

2. a Real-Time Governor (RTG),

3. a Joystick Input Block (JIB),

4. an animation enabler block (AEB), and

5. an animation module.

A-Lab forms a seamless connection between a running SIMULINK simulation and a real-time animation window. The simulation states are passed through the AEB to the S-Function DLL to the animation module. The animation window then uses the simulation states to update the position of the next animation frame. Running this process at regular intervals (10 frames a second by default) produces the animation.

When SIMULINK runs a simulation, it does so at the fastest speed possible, given the simulation settings. This typically occurs at a rate faster than real-time. There is no built-in mechanism to limit the simulation speed to real-time and as a result, the animation may seem distorted and unnatural. Furthermore, the effects of real-time user input (via a joystick) will not be accurately simulated. To solve these problems, the real-time governor (RTG) block was developed.

The RTG block is a SIMULINK S-function written in C++. S-Functions are a mechanism whereby programmers can write code (typically in C/C++) that can be used to extend the functionality of SIMULINK. The S-Function's functionality resides in a compiled DLL file that conforms to the S-Function specifications. These functions can then be inserted in to any SIMULINK block-diagram as a regular block. For more information on S-Functions, please consult [106]. The RTG block is called by SIMULINK at regular intervals (10 times a second by default, but this rate may be increased) when the simulation is running. At these intervals, the RTG block checks the current simulation time to the real-time (from the system clock). If the simulation time is ahead of real-time, the RTG pauses execution until real-time catches up. In this manner, the simulation is prevented from running faster that real-time.

The joystick-input block (JIB) is also implemented as an S-Function. SIMULINK calls this block whenever the instantaneous input signals are required. At these times, the joystick-input block reads the state of the connected joystick, scales the readings to the specified range, and reports them to SIMULINK. The use of the JIB requires that a compatible joystick be attached to the computer. The

JIB can capture joystick inputs in up to three (3) axes of motion. These axes typically correspond to the horizontal (X), vertical (Y), and throttle (Z) axes of a typical analog joystick.

The animation enabler block (AEB) is another S-Function block that communicates the simulation state to the animation module. The animation module is where the 3D animation is actually displayed, and is implemented as a separate application. The AEB communicates with the animation module via Active-X. Active-X is a (Microsoft) technology standard that allows for inter-process communication between applications. This allows disparate applications to be combined together to form other applications, as in the case of A-Lab. The S-Function called by the AEB can be directed to connect to any one of the A-Lab animation modules available by specifying a unique code for that animation module called the program ID. Thus, an F-14 animation module, for example, could be called instead of the helicopter module just by changing the program ID of the S-Function block parameters. In fact, the different AEBs available in the MoSART A-Lab library are simply links to the same S-Function with different set program ID parameters. Because of this, new A-Lab animation modules that are created can be linked to simply by copying an existing AEB and specifying the new program ID. In this way, A-Lab is kept as modular as possible.

The animation module is a separate executable. The code for this module is contained in the `d3d_heli` project. This is a Visual C++ MFC dialog-based application. It consists of the following two modules:

1. A graphical Animation Module (GAM), and
2. an Active-X Communications Module (ACM).

Each module is now discussed.

The GAM in the animation module is entirely similar to the GAM in the *Interactive MoSART Helicopter Environment*. In fact, the code here is a subset of the code from the environment. The animation module provides the following visualization features:

- A 3D animation window, and

Function	Description
New3DWnd()	Opens the 3D animation window
IsCreated()	Returns TRUE if 3D animation window is open
SetMeshName()	Sets the filename to load mesh from (for future expansion)
ReceiveSimData()	Passed in an array of the current simulation data

Table 4.1: Main Functions of the `ID3d_heli` interface

- a flight-indicator window.

These features were discussed in Section 3.6.3.

The ACM has the responsibility of receiving data from the AEB S-Function via Active-X, decoding it, and updating the animation based on the data. The ACM is able to receive data remotely (from another process) via the use of Active-X. The ACM exposes a *dispatch interface* through which Active-X enables applications can communicate with it. Dispatch interfaces are covered in detail within [124]. This interface is defined in `ID3d_heli.` The functions exposed by the `ID3d_heli` interface are listed in Table 4.1.

When an external application calls these functions through the dispatch interface, they are mapped to calls to functions in the `CD3d_heliDlgAutoProxy` class. As the name implies, this class acts as a proxy, or intermediary, between the external application and the GAM. It decodes the incoming data, and routes it to the appropriate function in the GAM.

4.3 Overview of A-Lab Modules

Several A-Lab modules have been developed for various systems. These include the following:

- Fixed-Base Pendulum A-Lab,
- Flexible-Inverted Pendulum A-Lab,
- Cart Seesaw Pendulum A-Lab,

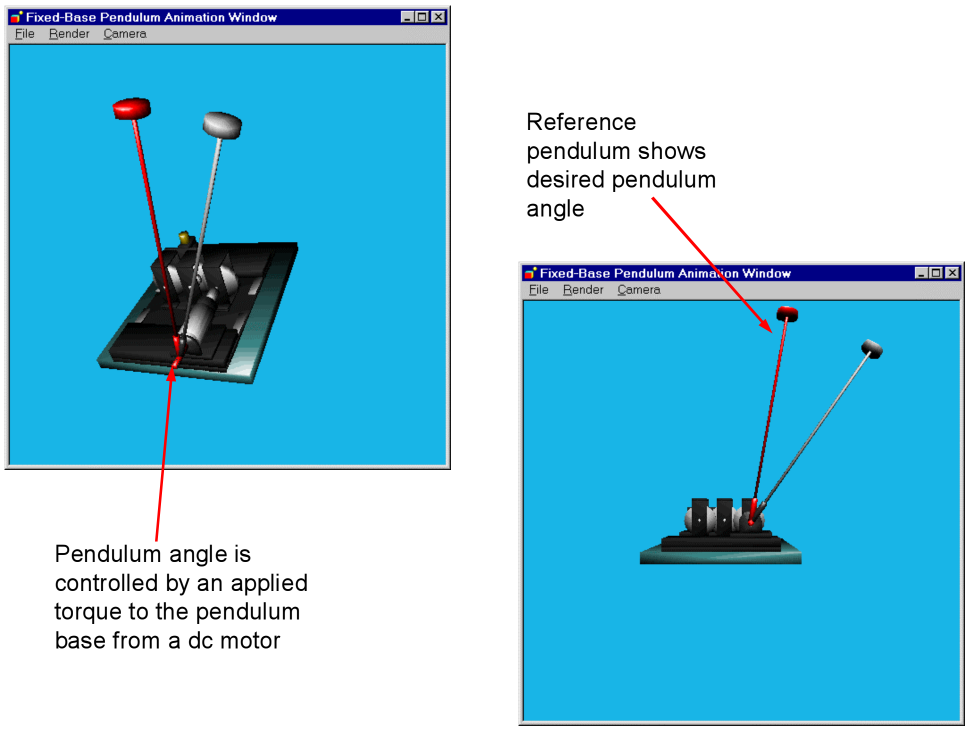

Fixed-Base Pendulum Animation Window
File Render Camera
Reference pendulum shows desired pendulum angle
Pendulum angle is controlled by an applied torque to the pendulum base from a dc motor
Fixed-Base Pendulum Animation Window
File Render Camera

Figure 4.5: Fixed-Base Pendulum A-Lab

- Helicopter A-Lab,
- Twin-Lift Helicopter systems A-Lab,
- Tilt-Wing RotorCraft A-Lab,
- Airplanes A-Lab,
- Submarine A-Lab,

Some of the developed A-Lab animation modules are shown in Figures 4.5 - 4.9. These modules have been widely used in various different research projects in various departments at ASU [47], [48], [51], [72]. Most of the animation modules were developed independently, based on the core A-Lab technology described in this thesis. In the next sections, the utility of the helicopter and twin-lift helicopter A-Labs as research and development tools will be demonstrated.

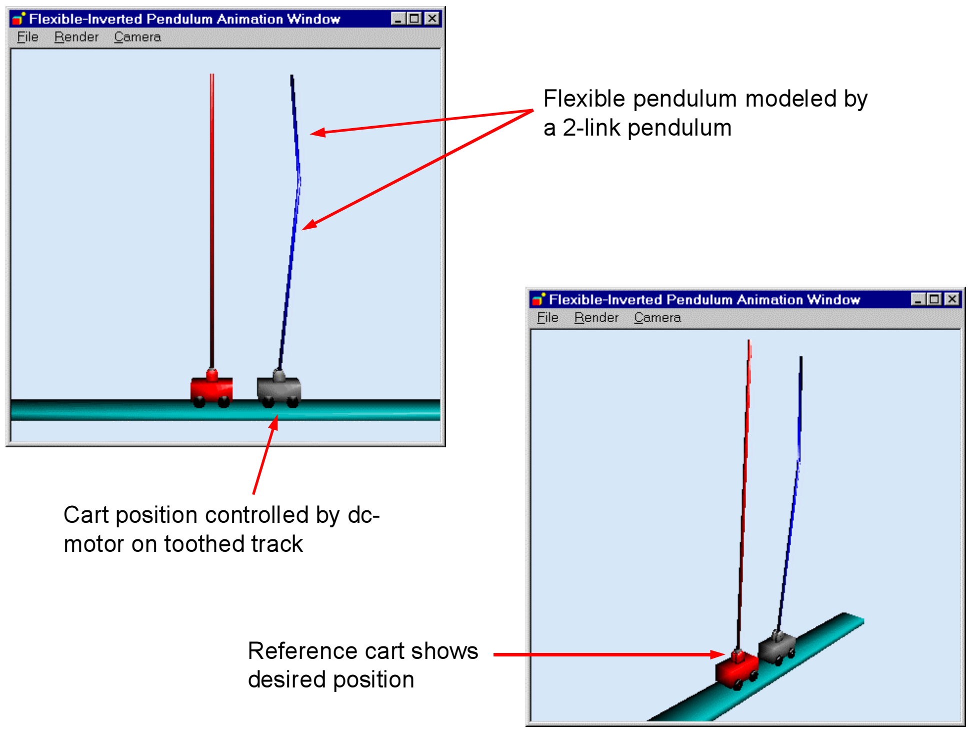

Flexible-Inverted Pendulum Animation Window
File Render Camera
Flexible pendulum modeled by a 2-link pendulum
Cart position controlled by dc-motor on toothed track
Reference cart shows desired position
Flexible-Inverted Pendulum Animation Window
File Render Camera

Figure 4.6: Flexible-Inverted-Pendulum A-Lab

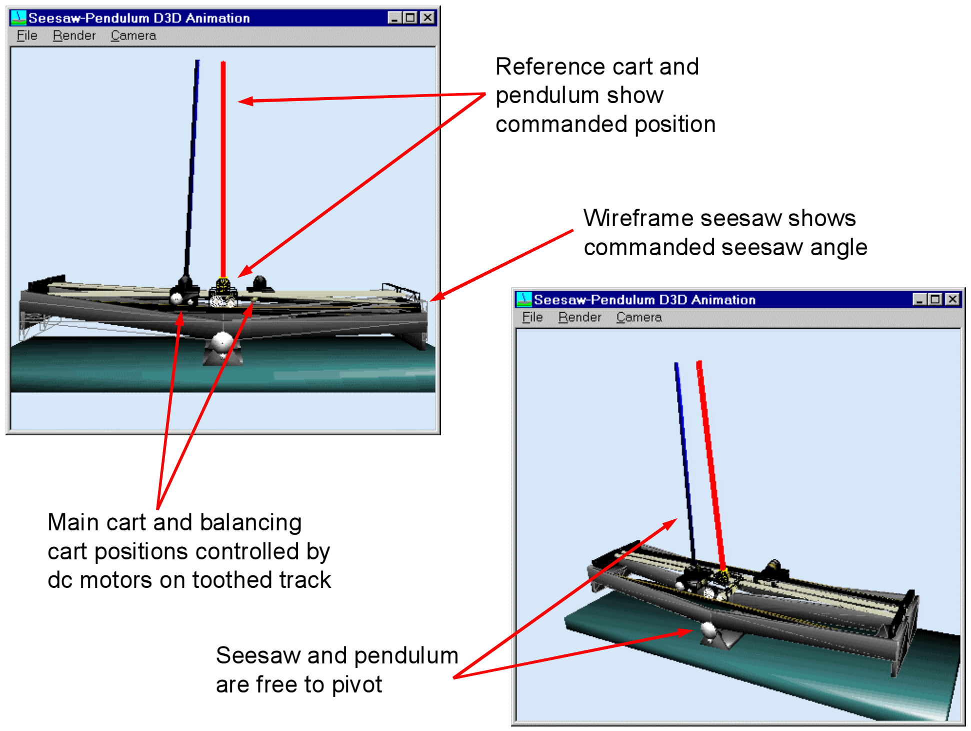

Seesaw-Pendulum D3D Animation
File Render Camera
Reference cart and pendulum show commanded position
Wireframe seesaw shows commanded seesaw angle
Main cart and balancing cart positions controlled by dc motors on toothed track
Seesaw and pendulum are free to pivot
Seesaw-Pendulum D3D Animation
File Render Camera

Figure 4.7: Cart-Pendulum-Seesaw A-Lab

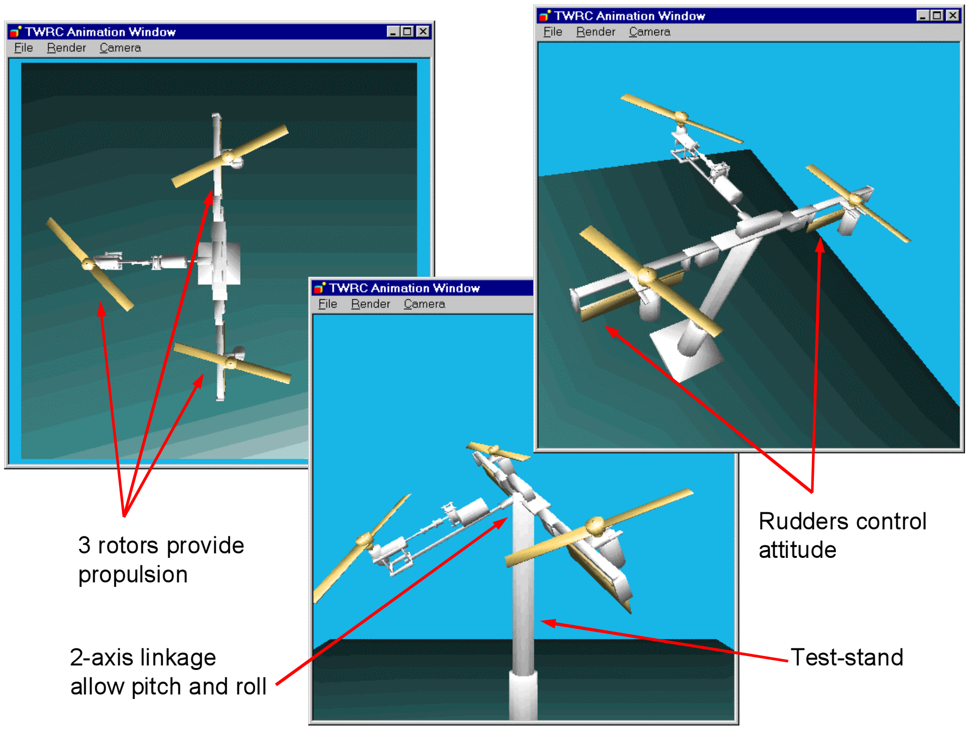


Figure 4.8: Tilt-Wing Rotorcraft A-Lab

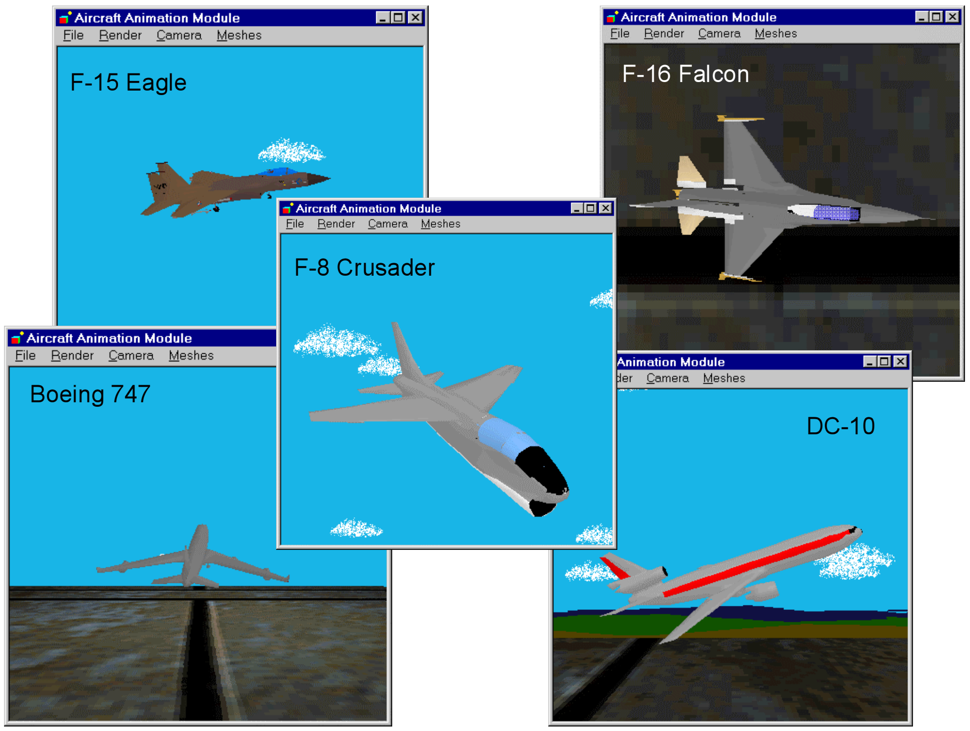


Figure 4.9: Airplane A-Lab Showing Various Aircraft Meshes Available

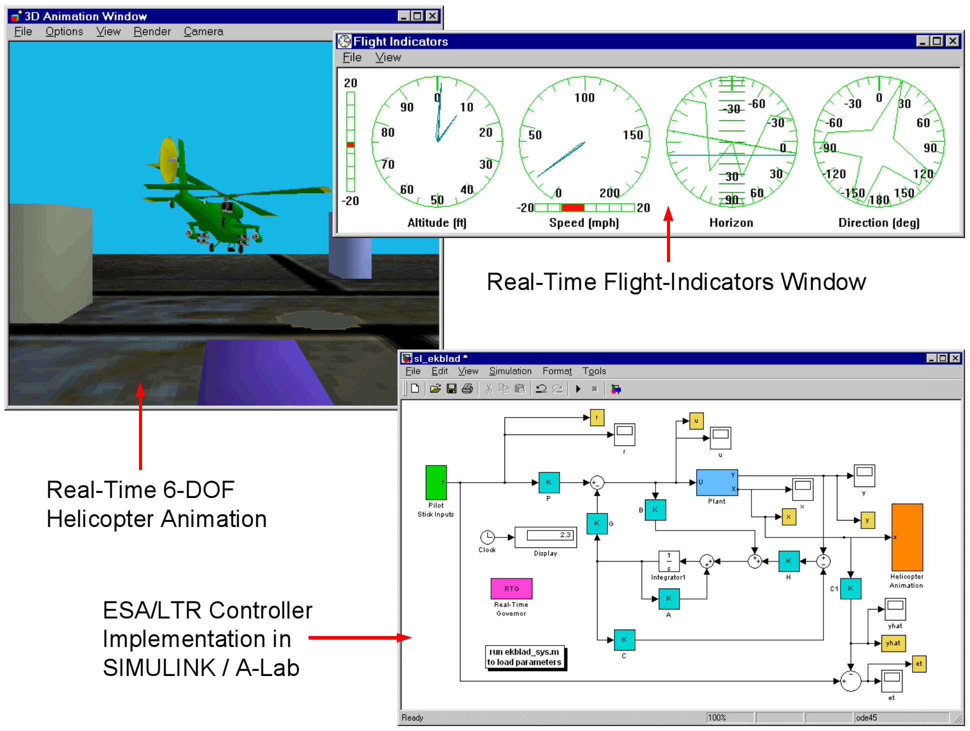


Figure 4.10: ESA/LTR Controller Performance: Simulation/A-Lab Setup

4.4 Helicopter A-Lab

In this section, the utility of the helicopter-system A-Lab is demonstrated. This tool is useful for evaluating the handling qualities of a particular control system. The helicopter-system A-Lab presents the user with a real-time three-dimensional animation of the aircraft, and a real-time flight-instrument panel (Figure 4.10). Various views of the helicopter may be selected via pull-down menus, or by hitting button #3 on an attached joystick. The flight-indicator window simulates a real flight-instrument panel and displays the current readings of the aircraft states, including the altitude, climb rate, forward speed, lateral speed, roll attitude, pitch attitude, and yaw attitude. The simulation is driven by a running SIMULINK block-diagram. The user may issue reference commands to the system via an attached joystick.

The combination of joystick-input, three-dimensional visualization, and flight indicators form a closed-loop system with the pilot (user). This allows the user to obtain an accurate feel of the dynamics of the implemented controller design. The testing of the system with A-Lab can then aid in the iterative redesign and improvement to the controller.

In the remainder of this section, two controller designs will be evaluated with A-Lab: The ESA/LTR controller design, and the $\mathcal{H}^\infty$ controller design. Both of these controllers were implemented as SIMULINK block-diagrams with similar settings. A Thrustmaster Flightstick Pro analog joystick was used to issue reference commands. The Y (up-down) axis of the joystick commanded forward speed, the X (left-right) axis commanded lateral speed, the Z (throttle wheel) axis commanded climb rate, and buttons 2 and 4 were used to issue step yaw-rate commands, simulating the pedals on an actual aircraft. The designs were evaluated by attempting to fly a take-off, forward-flight, and landing maneuver. This involved climbing to an altitude of approximately 60 ft, flying directly forward (maintaining a fixed heading within a $\pm$ 5 ft lateral corridor) at 45 ft/sec (30 mph) for approximately 40 seconds, coming to a complete stop, and then landing (return altitude to zero). The data from the simulation was then captured and analyzed.

4.4.1 ESA/LTR Controller Performance

The plots of the outputs are shown in Figure 4.11, the plots of the controls are shown in Figure 4.12, and a trajectory plot is shown in Figure 4.13. This controller exhibited very smooth command following in all channels. It was challenging at first to achieve a given altitude because of the slow rise-time in the climb-rate channel. There was some coupling in the longitudinal and lateral axes. This manifested itself as a side-slip tendency during forward flight. There is also a constant roll-attitude during constant-speed forward flight. There was also coupling in the longitudinal and yaw. This required the pilot to center the heading with the yaw-rate control (i.e. pedals) once forward flight had been achieved. The lateral-speed control was then used to maintain the desired direction during forward flight.

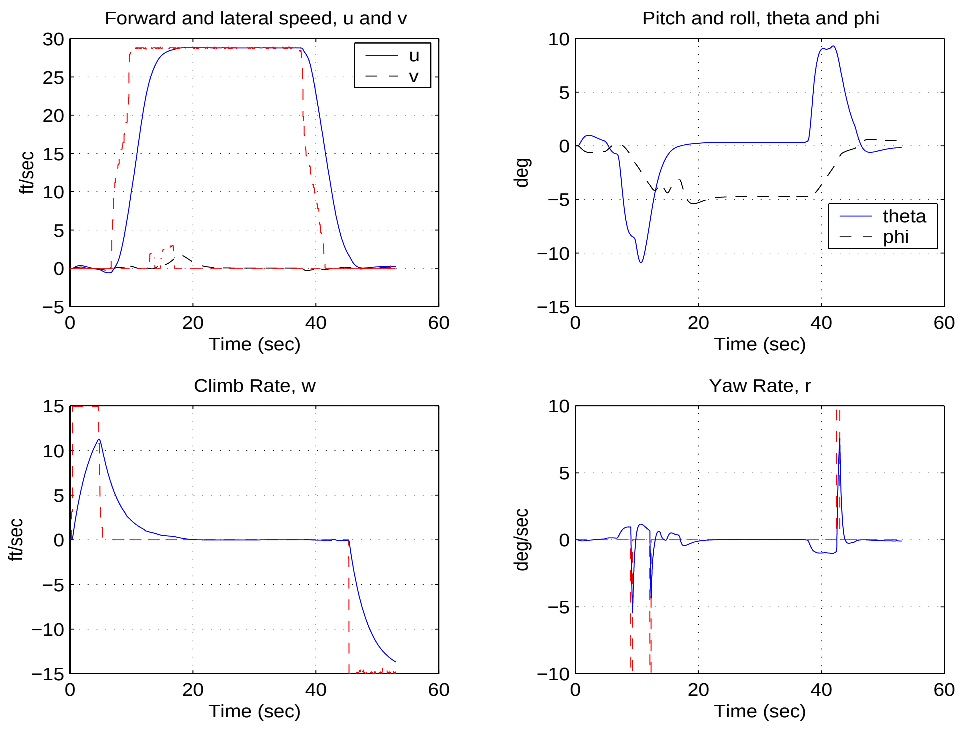


Figure 4.11: ESA/LTR Controller Performance: Flight Maneuver, Outputs

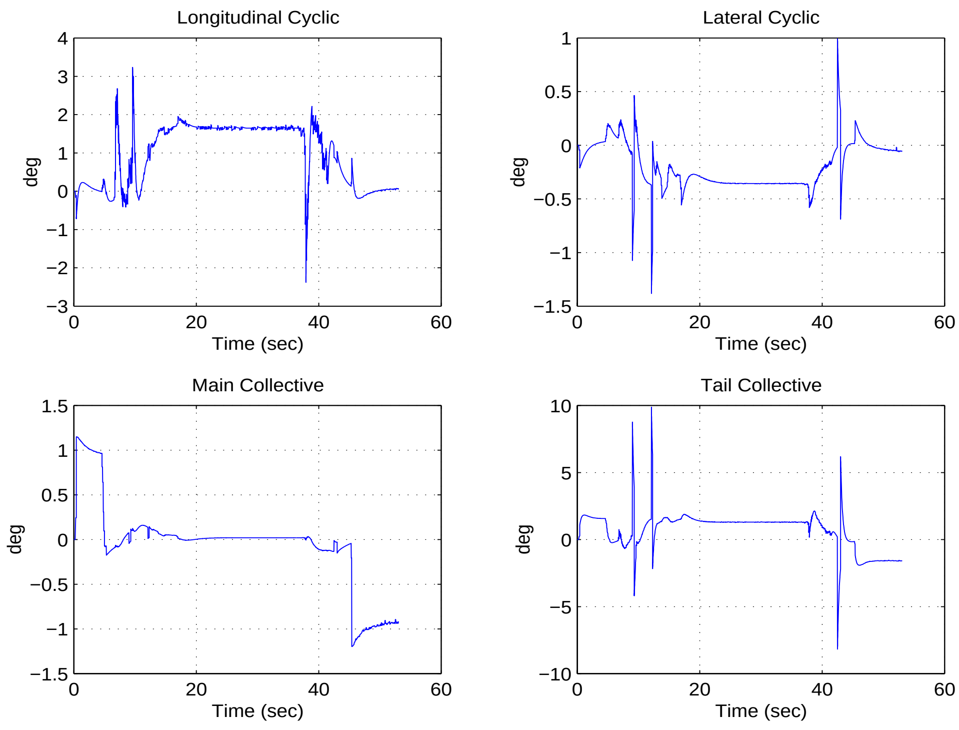


Figure 4.12: ESA/LTR Controller Performance: Flight Maneuver, Controls

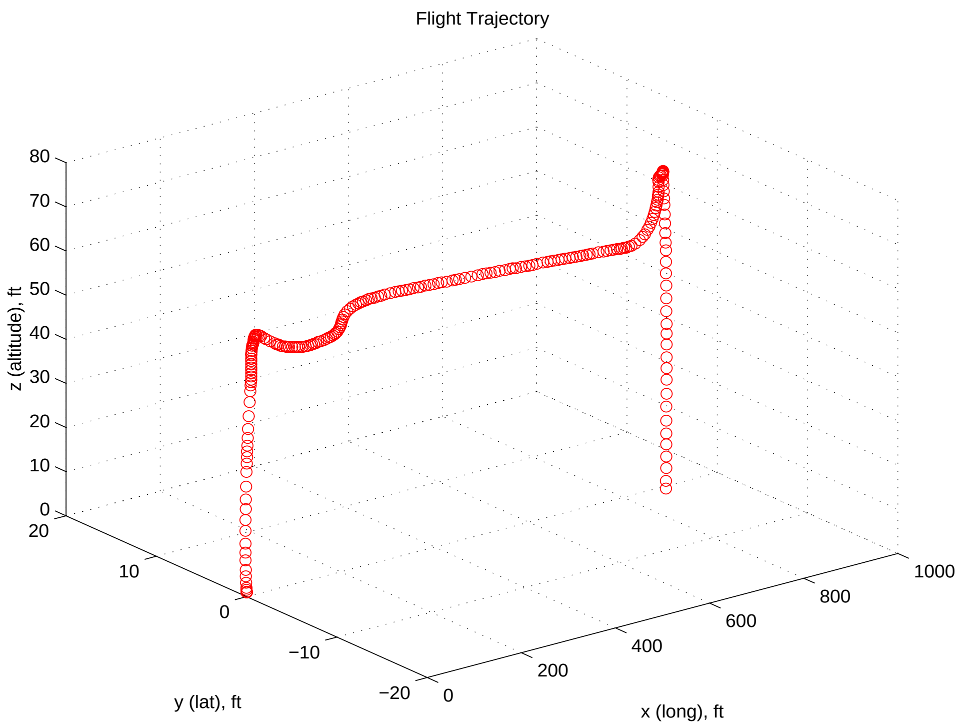


Figure 4.13: ESA/LTR Controller Performance: Flight Maneuver, Flight Trajectory

4.4.2 $\mathcal{H}^\infty$ Controller Performance

The plots of the outputs are shown in Figure 4.14, the plots of the controls are shown in Figure 4.15, and a trajectory plot is shown in Figure 4.16. This controller exhibited tight responses to the reference commands, due to the higher bandwidth compared to the ESA/LTR controller. It was very easy to achieve a given altitude. The level of decoupling is very similar to that of the ESA/LTR controller, and it was just as easy to complete the given maneuver (with the same pilot workload). Sideslip during forward flight was less than for the ESA/LTR controller, although more control action was required, particularly for the lateral cyclic. Roll-attitude was minimal once the aircraft had achieved a constant forward speed.

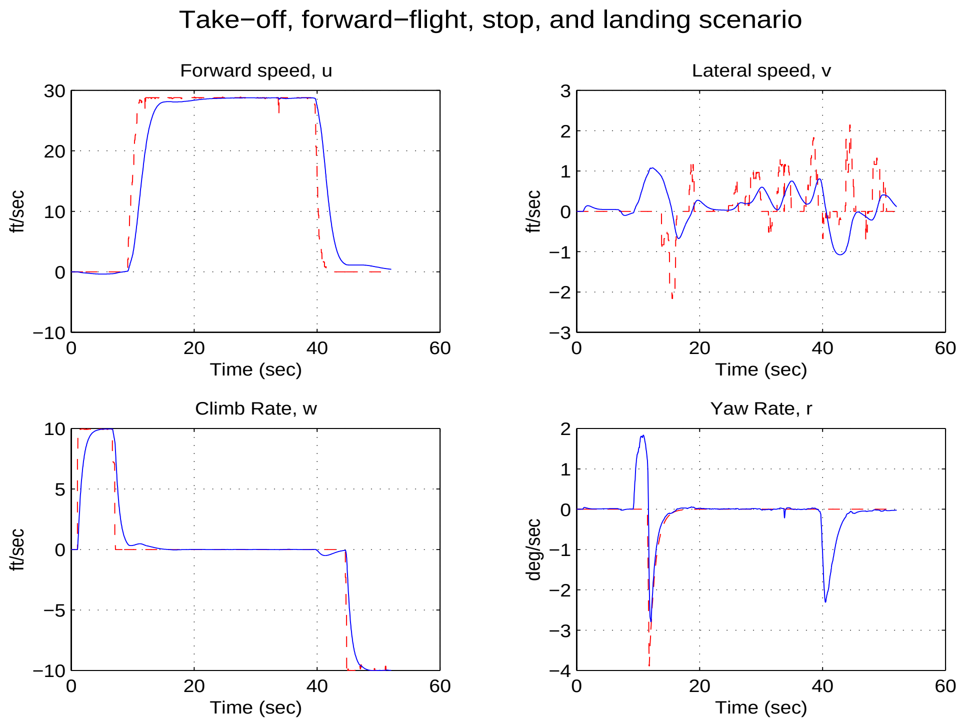


Figure 4.14: $\mathcal{H}^\infty$ Controller Performance: Flight Maneuver, Outputs

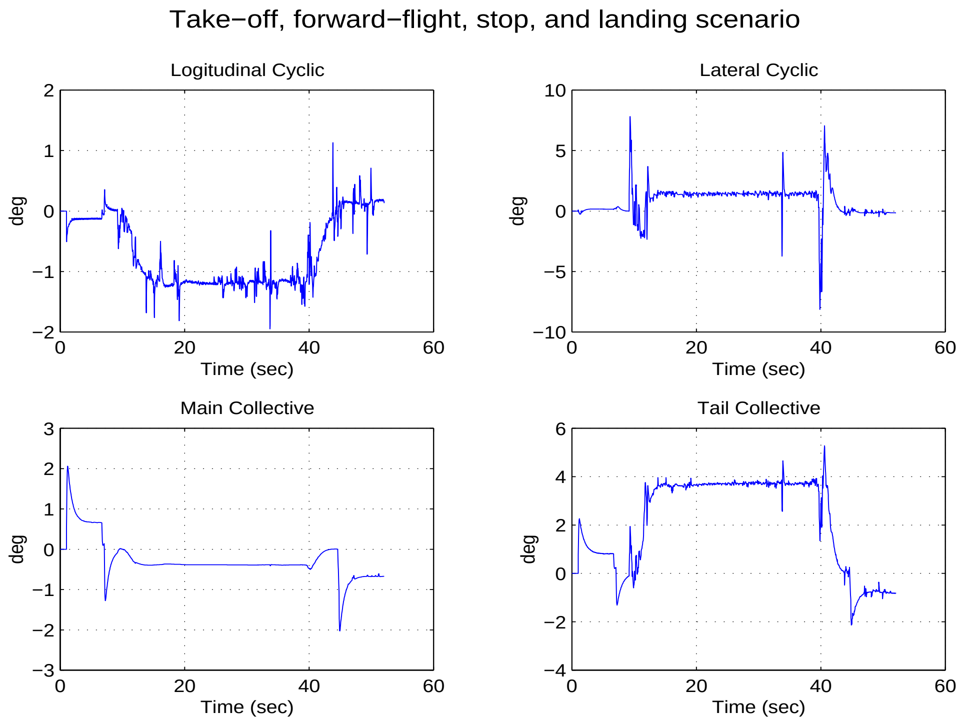


Figure 4.15: $\mathcal{H}^\infty$ Controller Performance: Flight Maneuver, Controls

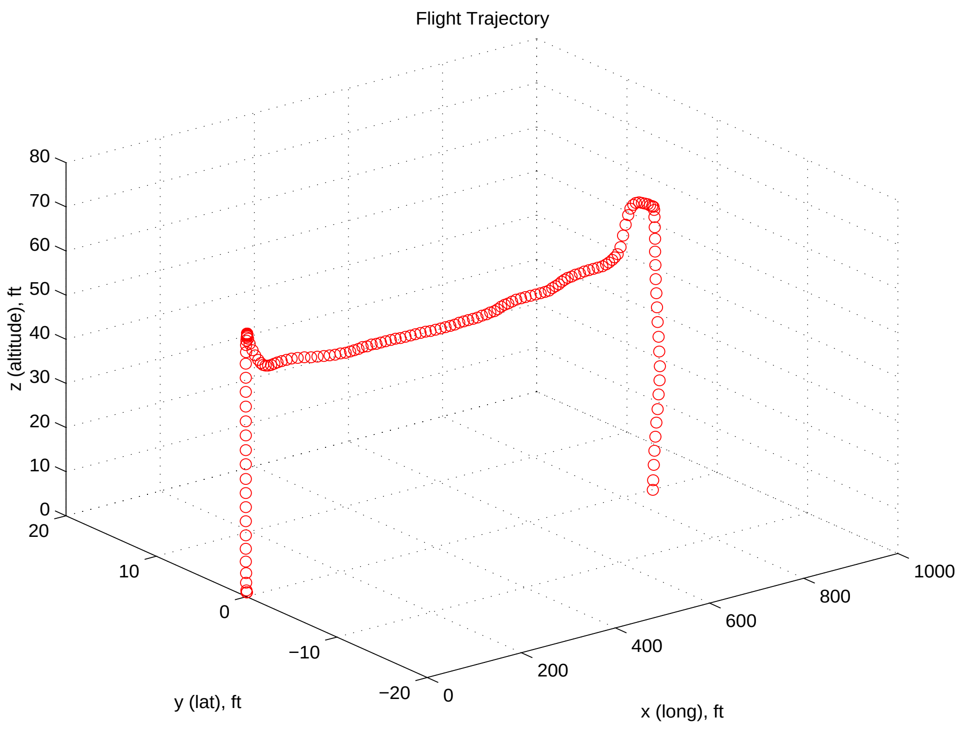


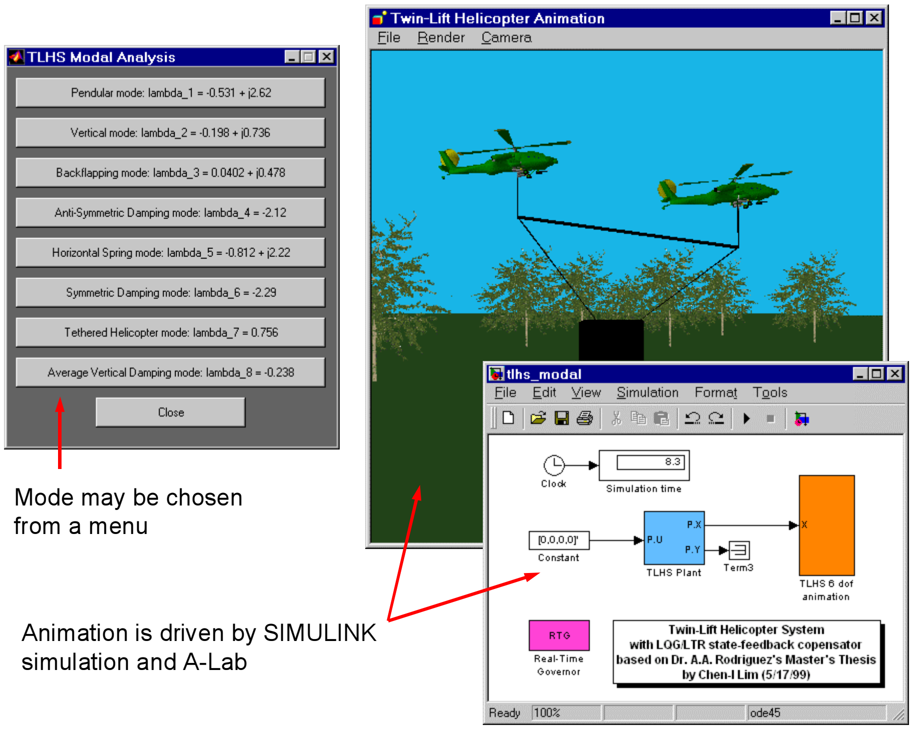
Figure 4.16: $\mathcal{H}^{\infty}$ Controller Performance: Flight Maneuver, Flight Trajectory

4.5 Twin-Lift Helicopter System A-Lab

Another A-Lab that has been developed is that for the Twin-Lift Helicopter System (TLHS). In this section, the utility of A-Lab in aiding in modal analysis will be demonstrated.

4.5.1 Modal Analysis

Before undertaking the design of a control system for a particular system, it is often essential to understand the open-loop system first. One method of gaining more insight into the system under study is through modal-analysis. First, an eigenvalue-eigenvector decomposition of the plant is performed. Each eigenvalue-eigenvector pair characterizes a particular mode of the system. In order to observe the time-domain behaviour of a particular mode in isolation of all other modes, the initial condition of the open-loop plant is set to the real-part of the corresponding eigenvector, and the unforced system response is observed. A-Lab may be used to animate each mode. This helps to develop a good understanding of the physical interpretation of each mode to the system.



Pendular mode: lambda_1 = -0.531 + j2.62
Vertical mode: lambda_2 = -0.198 + j0.736
Backflapping mode: lambda_3 = 0.0402 + j0.478
Anti-Symmetric Damping mode: lambda_4 = -2.12
Horizontal Spring mode: lambda_5 = -0.812 + j2.22
Symmetric Damping mode: lambda_6 = -2.29
Tethered Helicopter mode: lambda_7 = 0.756
Average Vertical Damping mode: lambda_8 = -0.238

Twin-Lift Helicopter System
with LQG/LTR state-feedback copensator
based on Dr. A.A. Rodriguez's Master's Thesis
by Chen-I Lim (5/17/99)

Figure 4.17: TLHS A-Lab Utility: Modal Analysis

This process was implemented for the TLHS. Figure 4.17 shows the setup of the analysis. A MATLAB macro was written in order to perform the eigenvalue-eigenvector decomposition. The user may choose a particular mode to view from a menu. When a mode is selected, the initial condition is loaded into a SIMULINK simulation and run. This simulation is attached to the TLHS A-Lab module, which visualizes the physical response due to exciting that mode.

CHAPTER 5

A Framework for Distributed Computation: Simulation and Optimization

5.1 Introduction

The *MoSART Distributed Computation Toolbox* is a suite of MATLAB-based *mex* and *m-files* that allow MATLAB scripts to be evaluated on remote computers, and results to be communicated accross TCP/IP networks. The system provides a general framework for performing distributed computation, allowing complex problems to be distributed to multiple computers on a network. The most direct application of the system is in dividing up a large problem, solving each piece on a separate machine, and then processing the combined results. More advanced applications such as intelligent optimization or search problems can also be accommodated.

In this chapter, the framework for the *MoSART Distributed Computation Toolbox* is described.

5.2 System Framework

System Requirements. The *MoSART Distributed Computation Toolbox* is meant for use on a network of PC-based computers running MATLAB version 5 and higher and Windows '95/'98/'NT4.

The system consists of:

1. The *Message-Server* software,

2. The *Server-Administrator* software,

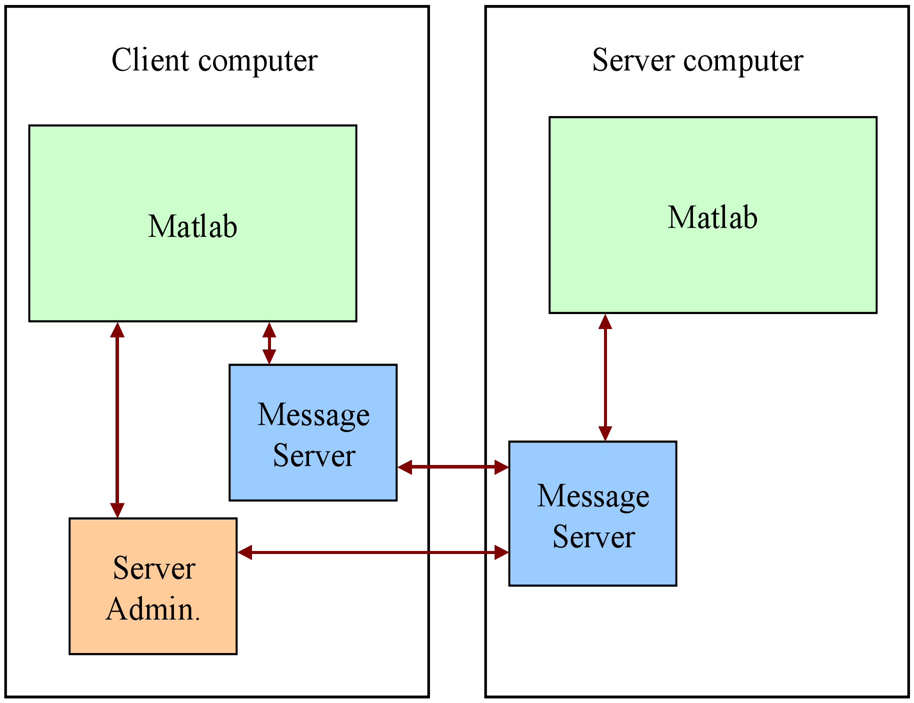


Figure 5.1: Distributed Computation: Communication between client and server computers

3. The MATLAB mex-files interface, and
4. The MATLAB m-files demos and tools.

The system will consist of at least one client computer and one or more server computers. The client computer is where the main script is executed, and is typically where the user operates the system from. Each server computer is remotely accessed and controlled from the client computer. The interaction between client and server computers is shown in Figure 5.1.

The Message-Server software. The *Message-Server* is a program that is installed and run on all of the workstations that are to be used for remote computation. Running the *Message-Server* on a workstation allows other computers to gain access to it and run MATLAB scripts on it. In addition, MsgServer must be installed on the client computer (where the main program resides) so that it can receive messages (mail, files, and commands) from remote servers.

The Server-Administrator software. The *Server-Administrator* serves as a directory to all the active *Message-Servers* on the network. Whenever a *Message-Server* is started on a computer, it sends its information to the *Server-Administrator* telling it that it is running and available for connections (except when it's run in client-mode). When a *Message-Server* is terminated, it tells the *Server-Administrator* to remove it from its list.

Whenever a program needs to gain access to a *Message-Server* (or several of them), it needs to contact the *Server-Administrator* to request the IP of an available *Message-Server*.

The MATLAB mex-files interface. Both client and server scripts gain access to the *Server-Administrator* and *Message-Servers* (and thus remote computers) via a set of MATLAB mex-file functions. These functions are:

1. `mexGetFreeServer` - to get the IP of an available *Message-Server*,
2. `mexFreeServer` - to release a *Message-Server* once you are done with it,
3. `mexSendFile` - to send a file to a *Message-Server*, and
4. `mexSendMsg` - to send a command or mail to a *Message-Server*,
5. `mexGetMail` - to read a mail sent to the local machine.

These mex-files are not used directly. Instead, the corresponding m-file 'wrappers' are called:

1. `GetFreeServer` - to get the IP of an available *Message-Server*,
2. `FreeServer` - to release a *Message-Server* once you are done with it,
3. `SendCmd` - to send a command to a *Message-Server*,
4. `SendFile` - to send a file to a *Message-Server*, and
5. `SendMail` - to send a mail to a *Message-Server*,

6. `GetMail` - to read a mail sent to the local machine.

7. `ClearMail` - Clears the incoming mailbox.

There is a special set-up file called `ServerParam.m`. This file contains the IP address for the *Server-Administrator* and the local machine.

Network Setup and system operation. A sample network setup for the system is shown in Figure 5.2. This diagram shows a cluster of 5 servers being controlled by one client computer with three servers currently allocated to it. A typical operation using the distributed computation framework proceeds as follows: First, the server administrator on the client computer is started, followed by the client message server. The message server on each server computer is then started up. When initiated, the message servers will communicate with the server administrator, and a list of available servers will be compiled. An available server is a server that is ready to receive commands but has not yet been allocated to a client. The client script is then executed. First, the client script must request the server administrator for one or more server to be allocated to it before commands may be sent to them. The `GetFreeServer` function is used to do this. This returns the IP address of newly allocated servers to the client script. The client script then uses these addresses to send commands (`SendCmd`), messages (`SendMail`), or files (`SendFile`) to the servers. Commands that are sent are executed on the instance of MATLAB running on the server computer. MATLAB is started on the server if necessary. Messages are stored on the message server, and may be retrieved with the `GetMail` function. Files that are sent are copied onto the current directory on the server computer. These operations allow arbitrary functions and scripts to be moved accross the network and executed, and thus can accommodate arbitrarily complex operations. Results may be communicated back to the client computer via messages or files in the same fashion. When the client script is finished with the servers, they should be freed with a call to `FreeServer`.

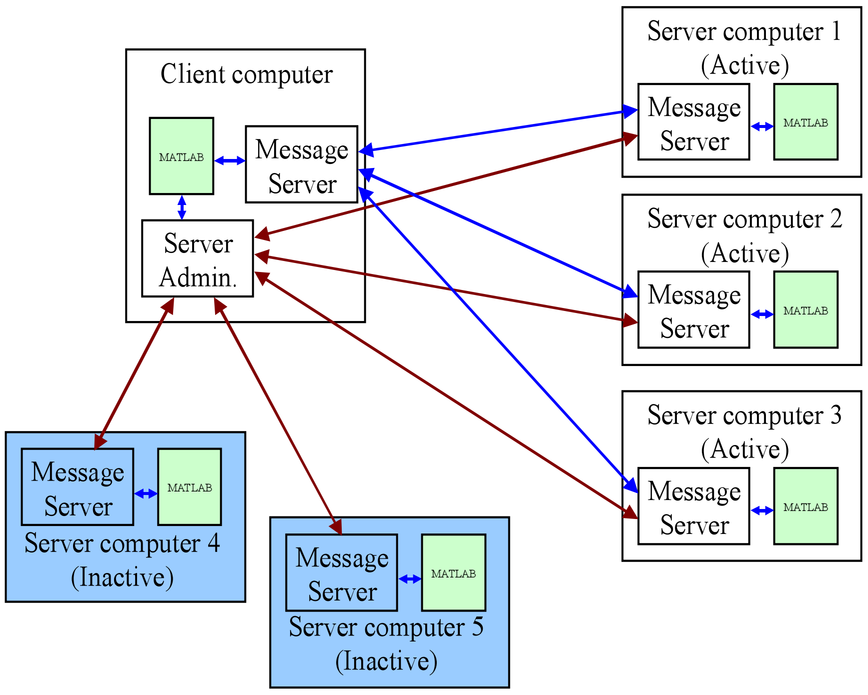


Figure 5.2: Distributed Computation: Network setup, showing a 5 server cluster with 3 active servers

CHAPTER 6

A Framework for Real-Time Hardware Control

6.1 Introduction

In this chapter, techniques for implementing real-time hardware control systems using PCs and data-acquisition boards are discussed. In particular, a real-time cart-on-track position control system is described. The model and control system are discussed first. This is followed by a description of the real-time hardware control framework, the hardware used, and the development of the real-time control software. Finally, the utility of the developed framework is demonstrated.

6.2 Cart-on-Track System Model and Control Law

In this section, the dynamics of a cart-on-track system are described, followed by the description of a position control system.

6.2.1 Cart Position Model and Dynamics

In order to design a cart position control system, it is essential to develop a mathematical model for the cart's translational dynamics. As shown in Figure 6.1, the cart can undergo translational motion - left and right parallel to the track. Assuming the cart never leaves the track, only the motion along the track needs to be considered.

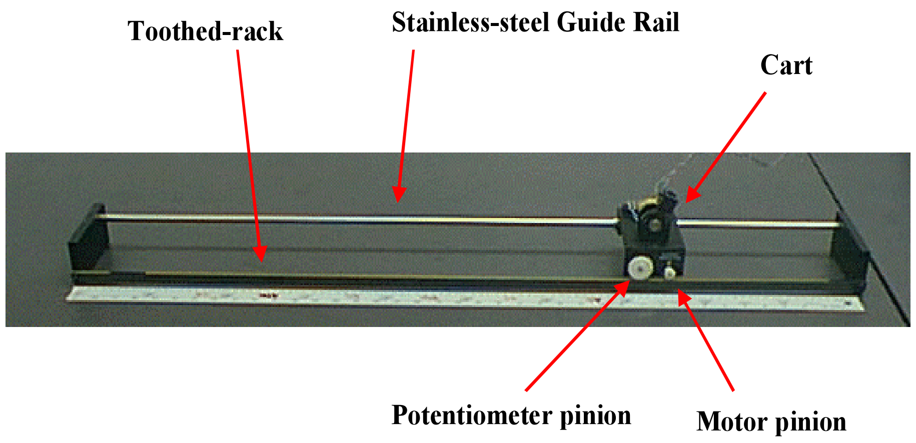


Figure 6.1: Visualization of Cart-Track System

If friction is neglected, it then follows from Newton's second law of motion that the cart displacement x is related to the applied force F by the following simple second order differential equation:

$$\ddot{x} = \frac{1}{m}F \tag{6.1}$$

where m is the mass of the cart. Taking the Laplace transform of each side of this differential equation (assuming zero initial conditions) yields the following transfer function in the s-domain:

$$\frac{X(s)}{F(s)} = \frac{1/m}{s^2}. \tag{6.2}$$

The dc motor used may be modeled as follows:

$$V = L_m \frac{dI}{dt} + IR_m + K_m \omega_m \tag{6.3}$$

where

V	Motor input voltage (V)
I	Motor armature current (A)
R_m	Motor armature resistance (Ω)
L_m	Motor armature inductance (Henrys)
K_m	Motor torque constant (N-m/A)
ω_m	Angular velocity of motor shaft (rad/sec)

The torque produced at the motor shaft is given by

$$T_m = K_m I \tag{6.4}$$

which, after the gearbox becomes

$$\begin{aligned} T &= K_g T_m && (6.5) \\ &= K_g K_m I && (6.6) \end{aligned}$$

This torque produces a force on the cart

$$F = T/r \tag{6.7}$$

where r is the radius of the output gear.

The linear velocity of the cart is related to the angular velocity of the motor shaft (taking into account the gear ratio) by:

$$\dot{x} = \frac{K_g \omega_m}{r} \tag{6.8}$$

From Equations 6.3 - 6.8, the relationship between applied force, applied voltage, and cart velocity may be derived:

$$F(s) = \left(\frac{K_m K_g}{r(sL_m + R_m)} \right) V(s) - \left(\frac{K_m^2 K_g^2}{r^2(sL_m + R_m)} \right) sX(s) \tag{6.9}$$

Inserting this equation into Equation 6.1 yields the transfer function from applied voltage to cart position:

$$P \overset{\text{def}}{=} \frac{X(s)}{V(s)} = \frac{(\frac{K_g K_m}{rR_a})/m}{\frac{L_m}{R_a}s^3 + s^2 + (\frac{K_m^2 K_g^2}{r^2 R_a}/m)s} \tag{6.10}$$

The inductance term is negligible, and may be ignored:

$$P = \frac{(\frac{K_g K_m}{rR_a})/m}{s^2 + (\frac{K_m^2 K_g^2}{r^2 R_a}/m)s} = \frac{1.716/m}{s(s + 7.658/m)} \tag{6.11}$$

The actuator has the effect of shifting one of the original poles of the force-mass system to the left.

This model is used to design a position control system for the cart.

6.2.2 Cart Position Control System

The position control system to be implemented may be visualized as shown in Figure 2.1. In this figure

- P is called the plant and represents the cart to be controlled
- y represents the cart position x
- u represents the control or applied force F
- K represents a dynamic compensator to be designed
- r is a reference command or the desired cart position
- e represents an error signal

- W is a reference command shaper or pre-filter
- H represents the sensor dynamics
- d_i and d_o represent disturbances modeled at the plant input and output respectively
- n represents sensor noise.

The goal is to design K and W such that the *tracking error*

$$e_T \overset{\text{def}}{=} r - y \tag{6.12}$$

is small for the anticipated reference commands, disturbances, sensor noise, and model uncertainty. Given this, we want to design K and W such that the closed loop system in Figure 2.1 exhibits the following properties:

- closed loop stability
- good low frequency command following
- good low frequency disturbance attenuation
- good high frequency noise attenuation
- robustness with respect to model uncertainty.

Control System Design. Given the above, the structure to be used for K and W are as follows:

$$K(s) = \frac{k(s+a)(s+b)}{s}\left[\frac{100}{s+100}\right]^2 \tag{6.13}$$

$$W(s) = \frac{b}{s+b} \tag{6.14}$$

where the term in brackets is intended to introduce high frequency compensator roll-off to reduce the sensitivity of the control u to the sensor noise n. The purpose of the pre-filter W is to ensure that the overshoot to step commands is small. This is done by compensating for the derivative action $(s+b)$ in the compensator K.

The design parameter a was selected to cancel the stable left-half-plane plant pole:

$$a = \frac{K_m^2 K_g^2}{r^2 R_m m} = 16.83 \tag{6.15}$$

while the design parameters k and b were selected to achieve dominant closed loop poles at

$$s = -5 \pm j5. \tag{6.16}$$

These pole-locations were selected based on the desire to achieve a rise-time of under 0.5 seconds, a settling time of approximately 1 second, and an overshoot of less than 4% to step commands.

Given this, the desired nominal closed loop characteristic polynomial is given by

$$\Phi_{CL_{desired}} \approx s^2 + 10s + 50. \tag{6.17}$$

The nominal open loop transfer function may be approximated as follows:

$$\begin{aligned} L(s) &= P(s)K(s) && (6.18)\\ &\approx \frac{k(s+b)}{s^2}\left[\frac{K_m K_g}{r R_m m}\right] && (6.19)\end{aligned}$$

From this, it follows that the nominal closed loop characteristic equation may be approximated as:

$$\Phi_{CL} \approx s^2 + k\left(\frac{K_m K_g}{r R_m m}\right)s + k\left(\frac{K_m K_g}{r R_m m}\right)b. \tag{6.20}$$

Given this, it follows that

Pole	Location
λ_1	-127.45
λ_2	-61.3018
λ_3	-16.8311
$\lambda_{4,5}$	$-5.6238 \pm j * 5.68648$
λ_6	-5 (in pre-filter)

Table 6.1: Cart Position Control System: Closed-loop Poles

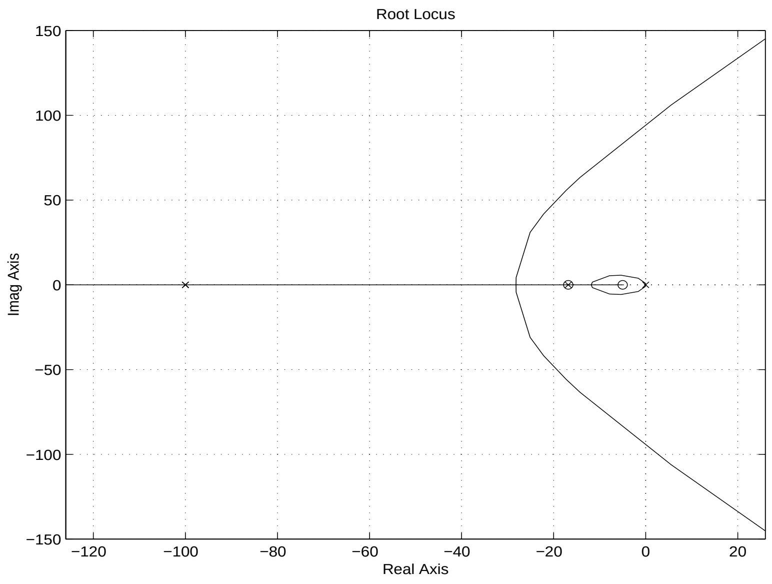


Figure 6.2: Root Locus

$$k = 10\frac{rR_m m}{K_m K_g} = 2.65 \tag{6.21}$$

$$b = 5 \tag{6.22}$$

With this selection, we obtain the closed loop poles shown in Table 6.1.

Control System Analysis.

Root Locus. The root locus for the resulting control system is shown in Figure 6.2. The compensator zero ensures stability for gain multiplicative perturbations $k_p \in (0, k_{max})$ for some k_{max} (nominally $k = 1$). For $k_p > k_{max}$ the closed loop system goes unstable.

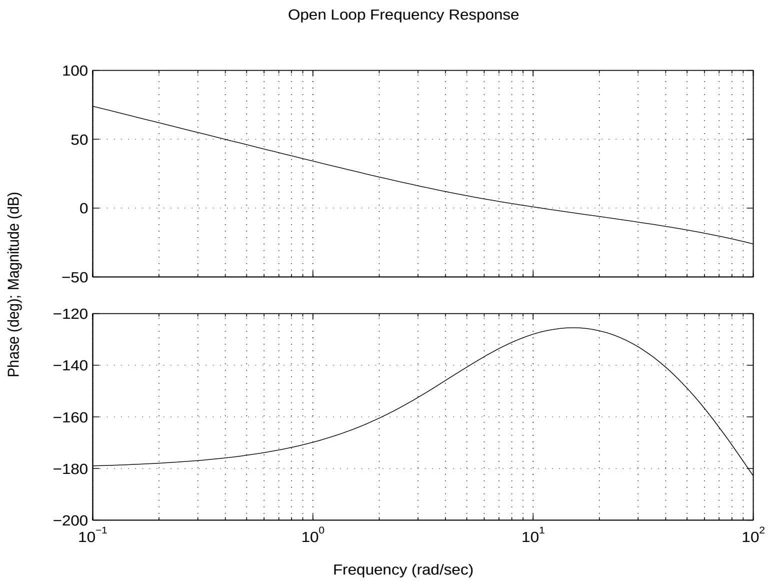


Figure 6.3: Open Loop Frequency Response

Open Loop Frequency Response. The resulting open loop frequency response is shown in Figure 6.3. The gain crossover frequency is approximately $\omega_g \approx 10\ rad/sec$. The -40 dB/decade slope at low fequencies will ensure zero steady state error to

- ramp and step reference commands r
- ramp and step output disturbances d_o.

Ramp and step input disturbances will not be rejected because K does not contain an internal model for these.

Stability Margins. The upward gain margin GM and phase margin PM for the resulting design are

$$GM \approx 17.8(25\,dB) \tag{6.23}$$

$$PM \approx 52^\circ \tag{6.24}$$

They may be visualized as shown in Figure 6.4.

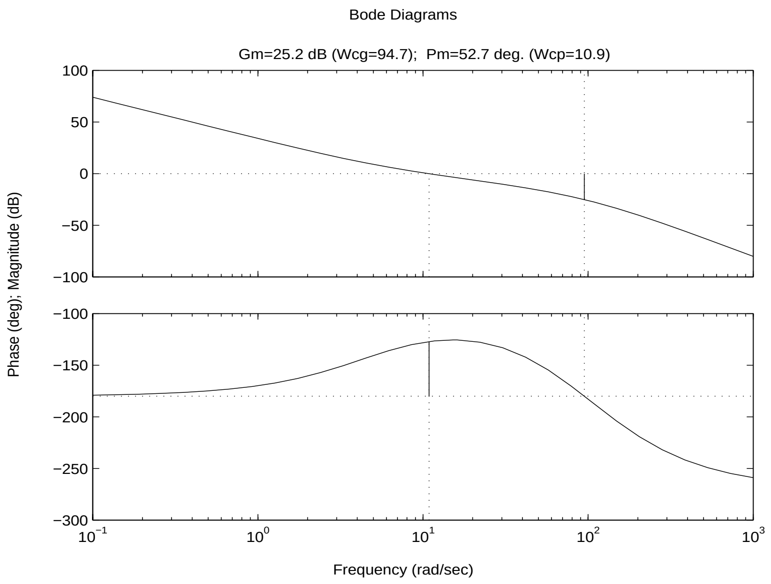


Figure 6.4: Gain and Phase margins

Sensitivity Frequency Response. The resulting sensitivity frequency response for

$$S \overset{\text{def}}{=} \frac{1}{1+L} \tag{6.25}$$

is given in Figure 6.5. Given this, we see that the sensitivity of our design to additive plant modeling errors at low frequencies will be small.

Input Disturbance Attenuation. The frequency response for the transfer function $T_{d_i y}$ from d_i to y is shown in Figure 6.6. It shows that low frequency disturbances are not attenuated. This unacceptable feature will be corrected subsequently by redesigning the controller K.

Complementary Sensitivity Frequency Response. The resulting frequency response for the *complementary sensitivity* transfrer function

$$T \overset{\text{def}}{=} 1 - S \tag{6.26}$$

$$= \frac{L}{1+L} \tag{6.27}$$

is given in Figure 6.7. It shows that sensor noise with frequency content above $\omega > 20$ rad/sec will be attenuated by at least 20 dB. The bump on the plot is a due to the derivative action $s+a = s+1$

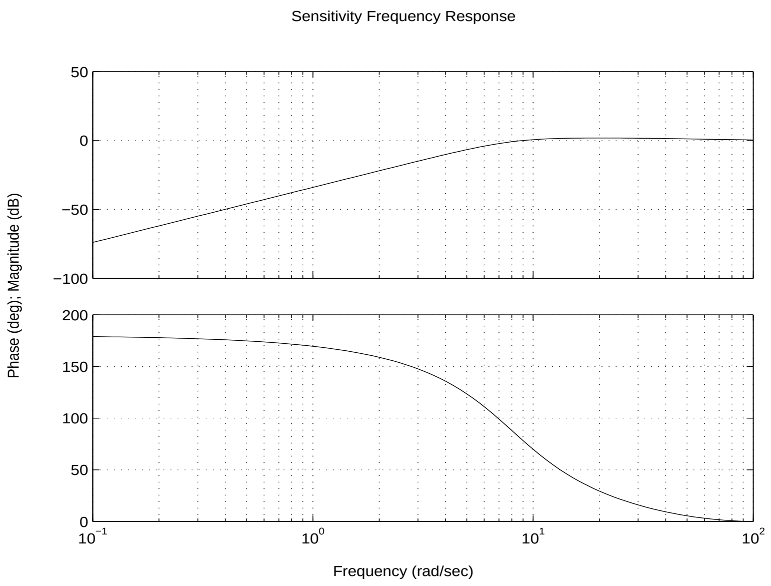


Figure 6.5: Sensitivity

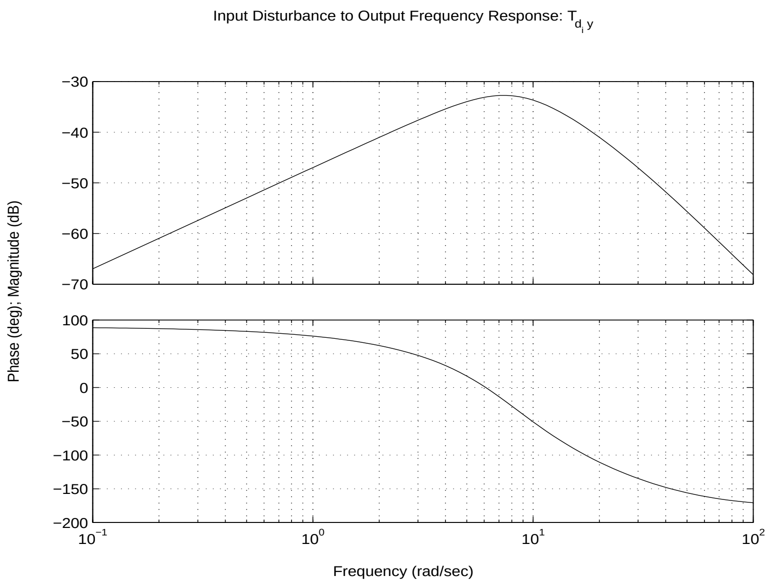


Figure 6.6: Input Disturbance to Output Frequency Response: $T_{d_i y}$

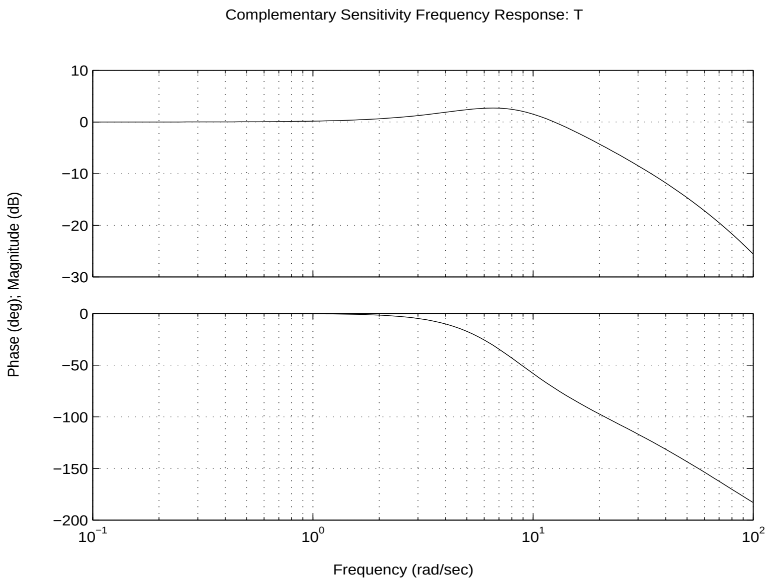


Figure 6.7: Complementary Sensitivity Frequency Response: T

within the compensator K. A consequence of this bump is that a large overshoot will result in the output y (cart position) when step reference commands r are issued, unless the commands r are processed by the pre-filter W. This justifies the use of the pre-filter W.

Reference to Output Frequency Response. The frequency response for the transfer function from r to y

$$T_{ry} = \frac{WL}{1+L} \tag{6.28}$$

$$\approx \frac{2}{s^2+2s+2} \tag{6.29}$$

is shown in Figure 6.8. The impact of the pre-filter W is seen in this plot. The plot specifically shows that low frequency reference commands will be followed well.

Reference to Control Frequency Response. The frequency response for the transfer function from r to u

$$T_{ru} = WKS \tag{6.30}$$

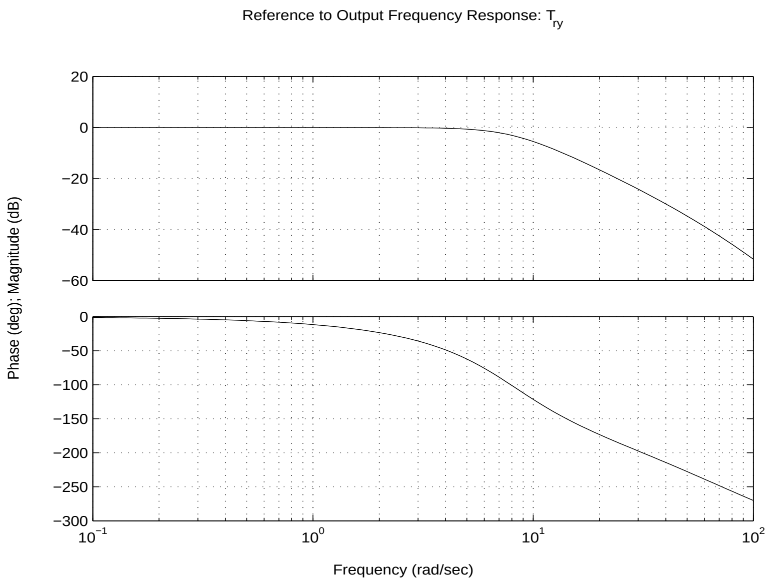


Figure 6.8: Reference to Output Frequency Response: T_{ry}

is shown in Figure 6.9.

Step Command Following. The response y to a step reference command is shown in Figure 6.10. As expected,

- the overshoot is small (less than 5%),
- the time to peak is about $t_p \approx 0.6$ *seconds*,
- the settling time is approximately $5\tau \approx 1$ *second*, and
- the steady state error is zero.

The corresponding control $u = V$ (applied motor voltage) is shown in Figure 6.11.

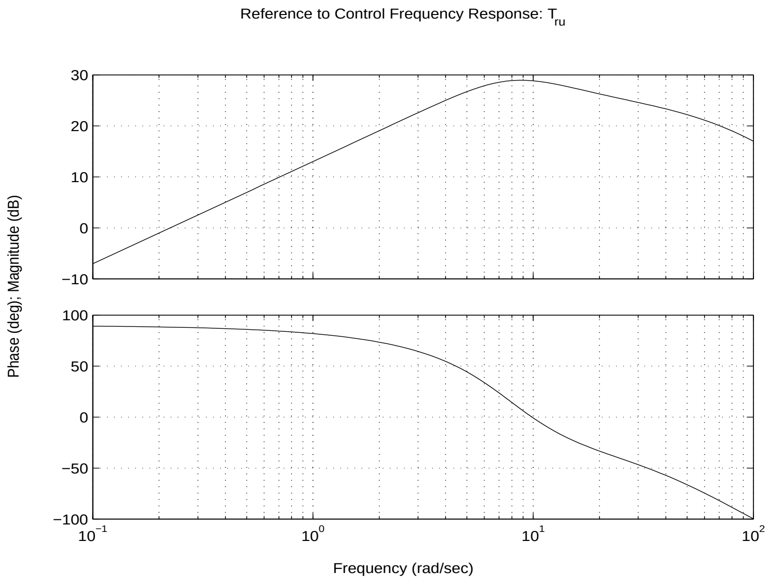


Figure 6.9: Reference to Control Frequency Response: T_{ru}

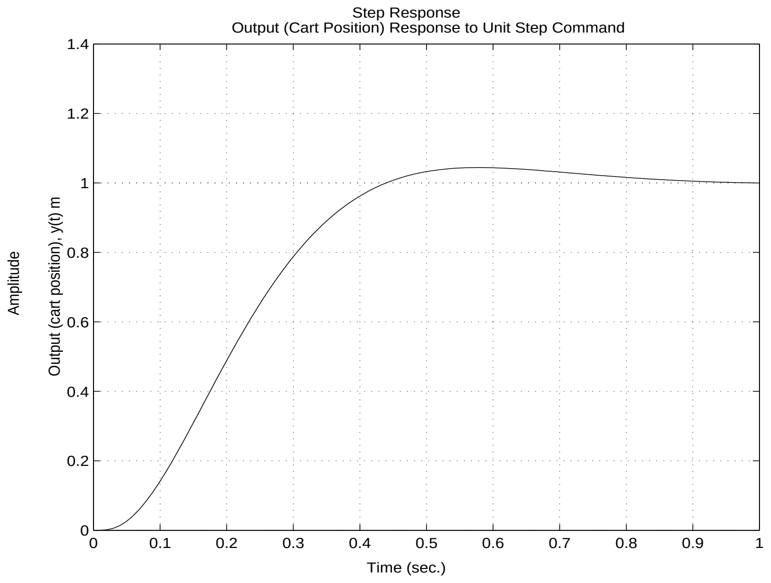


Figure 6.10: Output (Position) Response to Unit Step Command

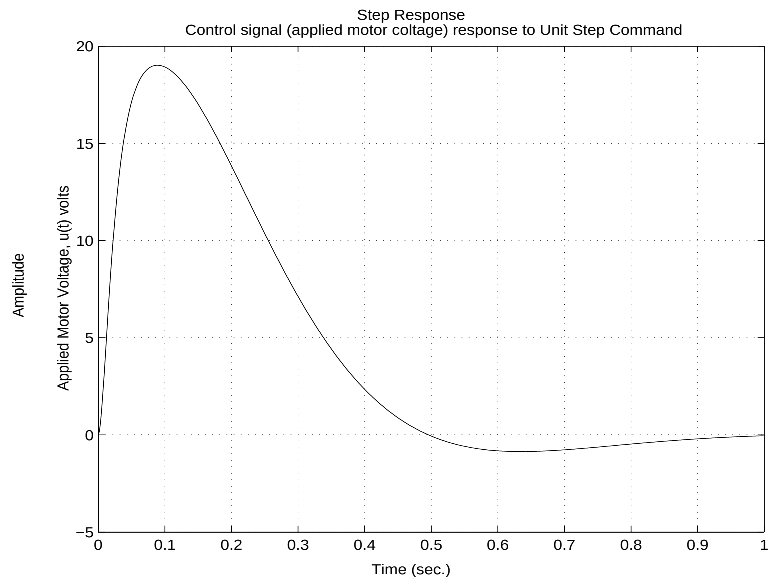


Figure 6.11: Control Signal (applied force) response to Unit Step Command

6.3 The Data-Acquisition Board

6.3.1 National Instruments Lab PC 1200

To interact with the cart-track system, a National Instruments Lab PC-1200 data acquisition (DAQ) board is used. The board, in its various form factors, can be visualized as shown in Figure 6.12. The Industry Standard Architecture (ISA) board is the form factor that is utilized in these experiments, but the functionality of the Peripheral Component Interconnect (PCI) and Personal Computer Memory Card International Association (PCMCIA) versions are similiar. For the experiment outlined in this report, similiar National Instrument DAQ board models, including the Lab PC+, may be used in place of the Lab PC-1200. The DAQ board is used to process position measurements obtained from the potentiometer on the cart. It is also used to send control signals to the armature controlled dc motor on the cart.

The National Instruments Lab PC-1200 data acquisition (DAQ) board is a low-cost, multifunction I/O device. The board has

- eight (8) 12-bit analog inputs (with a -5 to +5 volt range),
- two (2) 12-bit analog outputs (with a -5 to +5 volt range),

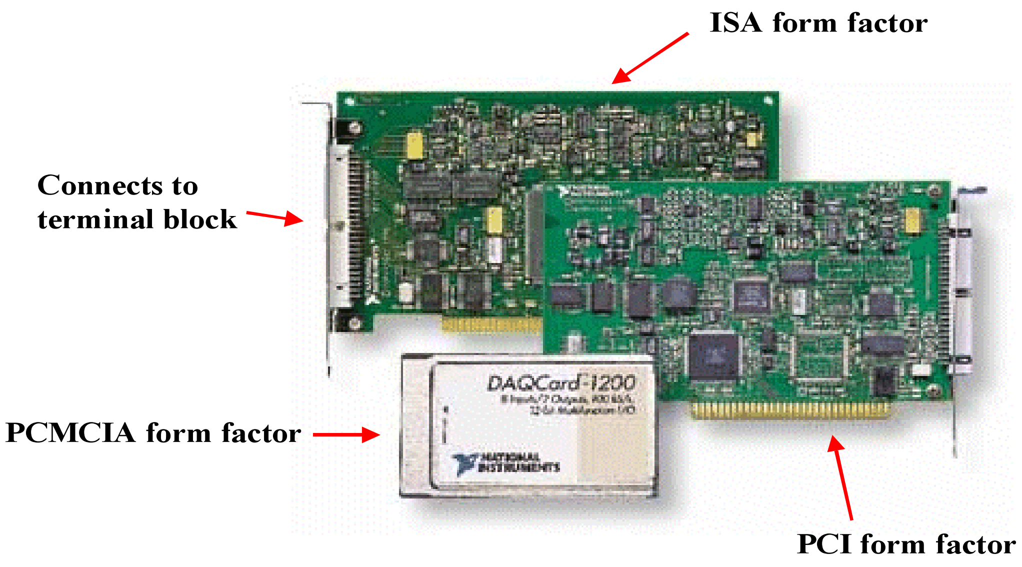


Figure 6.12: National Instuments Lab-PC 1200 DAQ board (in various form factors)

- twenty four (24) digital I/O lines, and
- three (3) on-board 16-bit 8MHz counters/timers.

As a data sampler, the board can sample up to 100 kS/s (kilo-samples per second). This sample rate is achievable when the board is programmed in buffered I/O mode, where data is read into a buffer or transferred out of a buffer at a fixed rate, and asynchronously with the rest of the pc. However, because we are using it as a real-time sample-control-output device within a synchronous software control loop, the maximum sustainable rate that can be achieved is approximately 1 KHz, for reasons given in Chapter 6.4.5.

The NI Lab PC-1200 DAQ board fits into an ISA slot on the PC, and is connected to an I/O connector block via a flat ribbon cable. The analog pins for the connector-block should be connected as listed in Table 6.2. Pins 2 and 4 are grounded because we are using *differential mode* for the board wiring. *Differential* mode means that each analog input pin has a seperate ground. For more

Pin	Function
1	Input channel 0
2	Input channel 0 ground
10	Output channel 0
11	Main Ground
12	Output channel 1

Table 6.2: Connector-Block Pin Descriptions (for differential mode)

information, please refer to [126].

6.3.2 Operation

To amplify the signal from the output of the controller (pin 10), a power amplifier unit - called the *Power Module PA-0103* - is used. This module may be used to amplify the control signal voltage and to boost the current. The power module also has a breadboard mounted on top of it. This can be used to create a low pass filter in order to filter out unwanted high frequency noise in the measurements. Additionally, voltage drops (resistors) can be added in series with the ± 12V supply voltage in order to provide the ± 3V bias for the potentiometers on the cart.

6.3.3 Board Setup for Cart-Track Position Control System

As mentioned earler, the complete hardware setup for this system consists of:

- cart mounted on track,
- armature controlled dc motor,
- potentiometer,
- a pc running Microsoft Windows NT 4.0,
- a National Instruments Lab PC-1200 data acquisition board (installed in pc),
- a ribbon cable,

Component Specification	Symbol	Value
Power supply		
DC Output		+12V and -12V
Power amplifier		
Maximum current output		3 Amperes
Maximum power output		40 Watts
Motor and gearhead		
Armature resistance	R_m	2.6 Ω
Armature inductance	L_m	0.18 $mHenry$
Torque constant	K_m	0.00767 N-m/A
Back EMF constant		0.00767 v/(rad/sec)
Gear ratio	K_g	3.7 (output is slower)
Max speed (gearhead)		1621 RPM (27 rev/sec)
Max force applied to cart		2 N (with 0.25” radius gear)
Output pinion radius	r	0.00635 m (or 0.25 in)
Rack		
Active length		0.91 m (36”)
Cart		
Mass with motor and parts	m	0.455 kg
Max speed		1.09 m/sec
Position Sensor		
Resistance		10 kΩ biased by two 7 kΩ resistors in series
Number of turns		10 turns, with stops
Conversion ratio (with $\pm$3V bias)		0.1524 m per volt (or 6” per volt)
Position pinion radius		0.1481 m (or 0.583 in)

Table 6.3: Component Specifications (from Quanser Consulting Documentation)

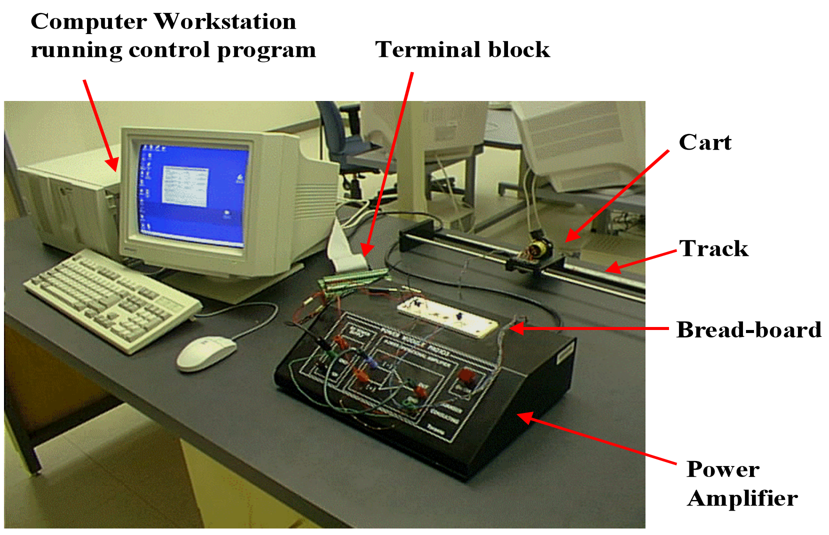


Figure 6.13: Cart Position Control System Set-Up

- a terminal block,
- a Quanser Consulting PA-0103 power module,
- a bread-board,
- two resistors and two capacitors for a filter, and
- wiring.

A fully set-up system is shown in Figure 6.13.

Set Up Instructions. To set up the system, one should proceed as follows:

1. Ensure that all equipment is off and unplugged before commencing hardware setup!
2. Install the NI Lab PC-1200 DAQ board and driver software into the PC (refer to [126] for complete and detailed instructions on how to do this).
3. Insert one end of the ribbon cable into the DAQ board connector.

4. Insert the other end of the cable into the terminal block.

5. Set up the cart and track experimental hardware as shown in Figure 6.1. If there is a pendulum on the cart, remove it.

6. Set up the power module as a voltage follower (see Figure 6.14). Connect Analog-Output-Channel 0 (pin 10 on the terminal block) to the positive (+) input terminal of the Op-Amp (See Figure 6.18). Connect the output of the Op-Amp back into its negative (-) input terminal. Also connect the output of the Op-Amp to the positive (+) input terminal of the cart motor.

7. Set up two 5.2 $k\Omega$ resistances on the bread-board to produce a 6V drop (see Figure 6.15, this figure shows the use of three 15.6 kΩ resistances in parallel to produce the required 5.2 kΩ resistance).

8. Attach the $+12$ V terminals to one of the 5.2 kΩ resistances in order to provide a $+3$ V output. Similarly, use the -12 V terminal to produce a -3 V output.

9. Attach the $+3$ V and -3 V outputs to the biasing pins of the position-sensing potentiometer on the cart as shown in Figure 6.16.

10. Attach the negative terminal of the motor to ground.

11. Attach the output of the position potentiometer to the channel 0 input of the DAQ board (pin 1 on the terminal block). See Figure 6.18.

Potentiometer Data Sampling: Position Information. For this system, the channel 0 input is used to sample the potentiometer on the cart. This gives the cart position information. Velocity information may be derived from this position information.

Control Signal: Applied Motor Voltage. The channel 0 output is used to drive the cart motor, via the power module.

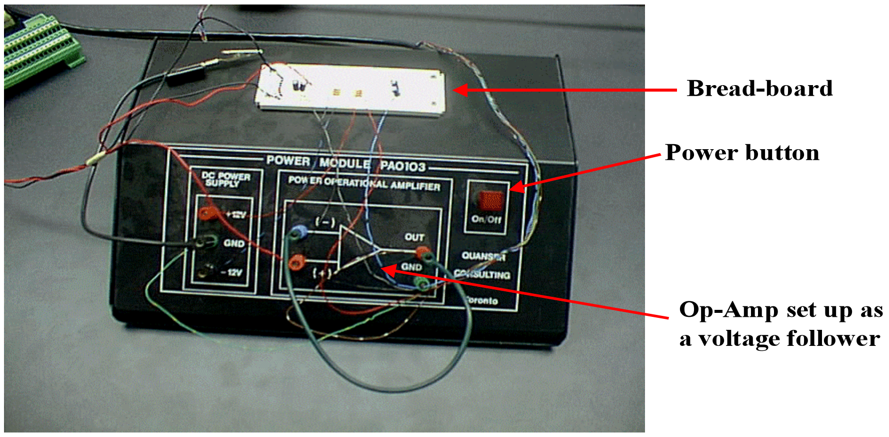


Figure 6.14: Power Module Set Up

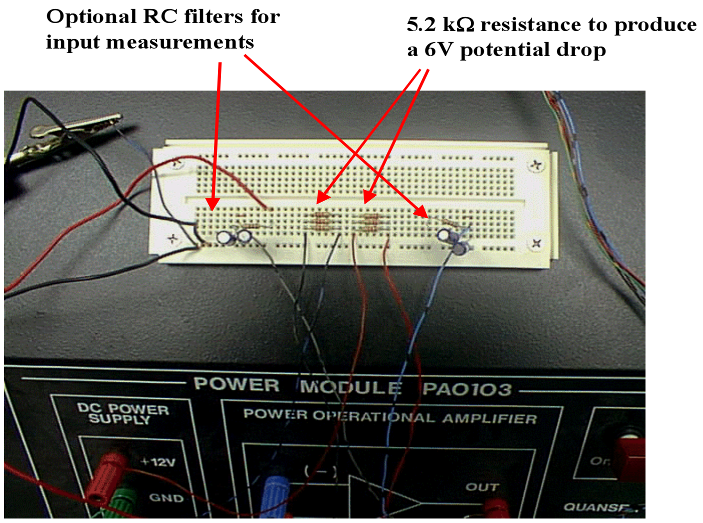

Optional RC filters for input measurements
5.2 kΩ resistance to produce a 6V potential drop
POWER MODULE PA0103
DC POWER SUPPLY
POWER OPERATIONAL AMPLIFIER

Figure 6.15: Bread-Board Set Up

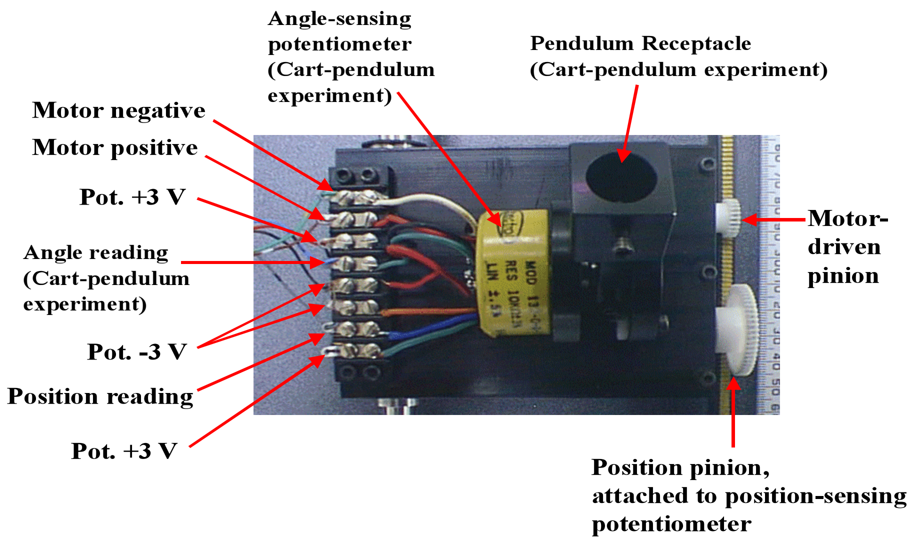

Angle-sensing potentiometer (Cart-pendulum experiment)
Pendulum Receptacle (Cart-pendulum experiment)
Motor negative
Motor positive
Pot. +3 V
Angle reading (Cart-pendulum experiment)
Pot. -3 V
Position reading
Pot. +3 V
Motor-driven pinion
Position pinion, attached to position-sensing potentiometer

Figure 6.16: Wiring for Motor and Potentiometer on Cart

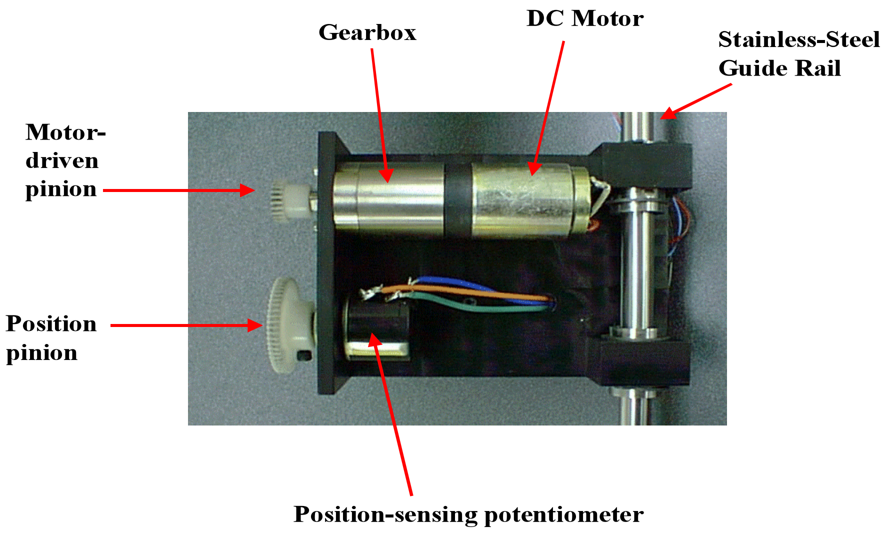


Figure 6.17: Underside of Cart

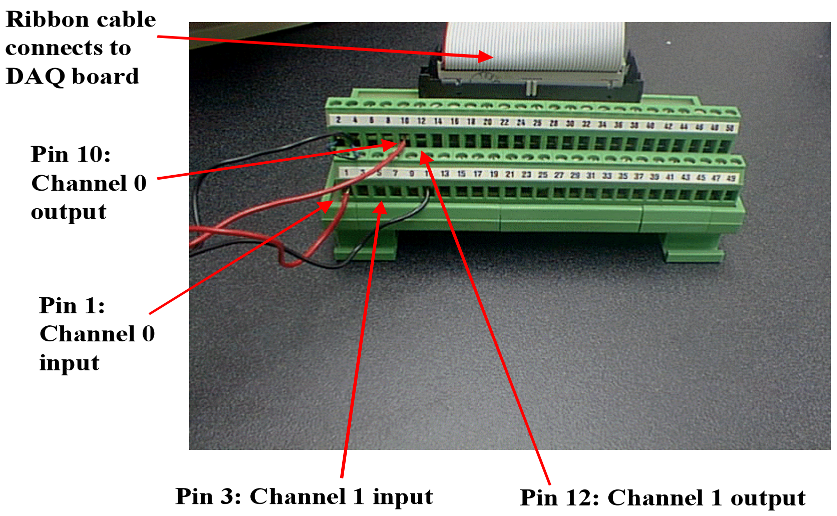


Figure 6.18: Terminal Block

6.4 Cart-on-Track Real-Time Hardware Setup and Control Using C++

In this section, we will describe the development of a Windows C++ based application that performs real-time control of the cart-track system via the National Instruments Lab PC-1200 DAQ board.

Many real-world control systems are implemented using embedded systems. Typically, the functionality of a DAQ-board and a microprocessor are combined onto a single chip called a micro-controller. The control algorithms would need to be implemented in a programming language such as C or C++ and then compiled into low-level assembly-language or machine-code to be downloaded onto the micro-controller. The controller is implemented as a digital and discrete algorithm that is called (executed) periodically by a timer. The control-loop will need to obtain the neccessary readings from the input ports of the hardware, run the control algorithm to calculate the output control signal, and then drive the outputs of the hardware.

The C++ application described in this section simulates this process, albeit on a much higher-

Function	Purpose
AO_Configure	Set the parameters of the DAQ board
NIDAQErrorHandler	Helper function to check return codes
AO_VWrite	Write a voltage to an output port (voltage changes when AO_Update is called)
AO_Update	Starts outputting the new voltages in all channels
AI_VRead	Reads a voltage from an input channel

Table 6.4: Basic NiDAQ functions

level with a PC, Windows, and a DAQ board instead of a microcontroller. This program was developed using Microsoft Visual C++ version 5.0 and the National Instruments C++ DAQ libraries.

6.4.1 Microsoft Visual C++

Microsoft Visual C++ version 5.0 is a Microsoft Windows-based Integrated Development Environment (IDE) and C++ compiler used for developing 32-bit graphical Windows applications. Microsoft Visual C++ 5.0 comes with the Microsoft Foundation Classes (MFC) library that provides a C++ class hierarchy that allow interactive graphical applications to be easily developed. For more information on using Visual C++ or programming with the MFCs, please consult [122].

6.4.2 The National Instruments C++ Libraries

National Instruments publishes a library of C routines (the NiDAQ library) for interfacing with their DAQ boards [1]. The two libraries that need to be linked with the C++ program are `nidaq32.lib` and `nidex32.lib`. The basic functions that are used for the real-time cart position control system are listed in Table 6.4. For more information on the NiDAQ library and a comprehensive reference guide for all the NiDAQ functions, please see [129] and [128].

In order to simplify using the NiDAQ functions, several helper functions were created. Thse are listed in Table 6.5. These functions automatically use device 1, and perform error checking (e.g.

[1]The NiDAQ libraries may be downloaded from the National Instruments web site at http://www.natinst.com

Function	Purpose
int ResetDevice()	Initializes DAQ Board, setting all outputs to ground
int SetOutV(int iOChan, f64 dVoltage)	Outputs voltage on specified channel
f64 GetInV(int iIChan)	Reads input voltage form specified channel
int SetDiffOutV(f64 dVoltage)	Sets up a differential voltage between channels 0 (+) and 1 (-)

Table 6.5: Helper functions for NiDAQ

output voltage is limited to ± 5 *volts*). The functions are contained in the file `ndqfun.h` and are implemented in `ndqfun.cpp`. The *SetDiffOutV* function is used to specify a differential voltage when the power-module is set up as a subtractor circuit. This allows a larger (± 10 *volt*) output range, but is not used for this control system. The source code for these functions are listed in Appendix B.

6.4.3 Creating a Real-Time Control Dialog with Microsoft Visual C++

The Microsoft Visual C++ 5.0 IDE was used to create this application. The AppWizard was first used to generate a skeleton dialog-based application.

6.4.4 Implementation of the Cart Position Control System

The real-time cart position control program was implemented as an MFC dialog-box application. The project workspace is stored in `RtcCart.dsw`. The main classes used in the project are listed in Table 6.6. The main functionality of the program is in `CRtcCartDlg.cpp`. The main source code files are listed in B. Table 6.7 lists the main functions in this file. The other functions in the file are mainly dialog box handlers that link the user-interface to the control program. The main program execution is as follows:

1. OnInitDialog() (line 271 of `RtcCartDlg.cpp`) is called. DAQ board reset, initial parameters loaded.

2. The screen-update timer is set up using the Windows timer, at the default rate of 10 Hz. This rate was chosen because it is sufficient to give the user a preception of a continuously updating display, while not being over-taxing on the processing time.

Class	Purpose
CAboutDlg	About dialog-box, with program info
CCmdParam	Structure for specifying a reference command setting
CCommandsDlg	Machine-generated reference command parameter dialog
CRtcCartApp	Main application module
CRtcCartDlg	Main user-interface dialog
CStrNum	Helper class for storing numbers in character format

Table 6.6: Classes in RtcCart project

3. When the “Run” button is pressed, the control-loop Windows high-performance timer loop is activated. The default processing rate is 50 Hz. This rate was chosen to be sufficient for this experiment. A variable time-step is implemented, meaning that the actual time-step is computed (using the high-performance counter) before each call to the control loop. Thus, if other multitasking event (e.g. screen updates) cause the actual time-step to be larger than it was specified to be, the actual time-step is used in the control-loop calculations.

4. Program now waits for timers to be executed.

5. The control loop timer calls ControlStep().

6. ControlStep() executes one iteration of the control algorithm:

   (a) Reads in current potentiometer voltage.

   (b) Compares that to current set-point, computes error.

   (c) Runs the control algorithm on the error to produce an output voltage.

   (d) Sends output voltage to DAQ board output port.

7. Screen update timer calls UpdateView(), which refreshes the interface with current measurements.

8. Program terminates when user shuts down the dialog box.

Since this program is an event-driven application, the user can interact with it on the fly via the user interface. The user interface is shown in Figure 6.19. The control loop is started or stopped

Function	Purpose
OnInitDialog()	Called on startup, initializes program and DAQ board
ControlStep()	Implements the control algorithm
GetReferenceCommands()	Loads current reference command (from user or from function generator)
OnTimer()	Fires at set interval, calls ControlStep()
OnUpdateSettings()	Loads new parameter values from dialog box into control algorithm
UpdateView()	Updates the user interface display

Table 6.7: Main functions in CRtcCartDlg

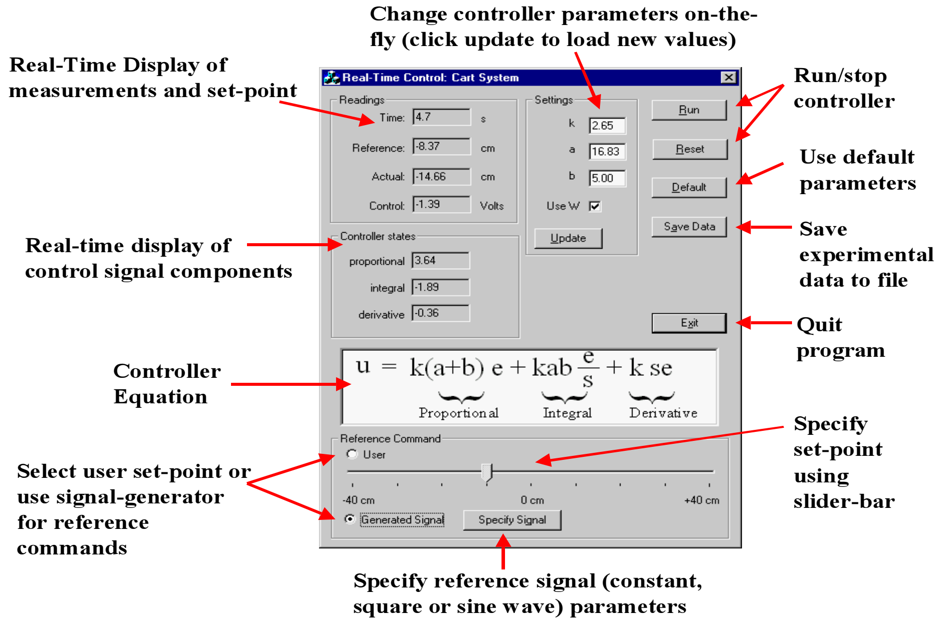
by pressing the *Run* or *Stop* button respectively. When the *Stop* button is pressed, it becomes the *Reset* button. Pressing the *Reset* button clears all logged data from memory, and resets the state of the controller. Controller parameters (a, b, and k) may be changed on-the-fly. The new values are not implemented until the *Update* button is pressed. Pressing *Default* will reset the controller parameters to their default nominal values.

The user may specify a set-point by dragging the pointer on the slider-bar to the desired reading. Alternatively, by activating *generated signal* mode (using the radio button), a software-based function generator may be employed to produce the reference command. Constant, sinusoidal, and square-wave reference commands may be generated with user-specified amplitudes, frequencies, and offsets.

The user interface also displays a real-time readout of key readings and voltages. These include:

- the current execution time,
- the current average processing rate,
- the current reference set-point,
- the actual cart-position,
- the output control signal, i.e. the commanded motor voltage, and
- the controller states (the proportional, integral, and derivative components of the control signal.

Pressing the *Save Data* button will allow the user to specify a file to save the current data stored in memory. The memory buffer size is set at twenty-thousand (20000) samples. At 50 samples-


Change controller parameters on-the-fly (click update to load new values)
Real-Time Display of measurements and set-point
Run/stop controller
Use default parameters
Save experimental data to file
Real-time display of control signal components
Quit program
Controller Equation
Specify set-point using slider-bar
Select user set-point or use signal-generator for reference commands
Specify reference signal (constant, square or sine wave) parameters
Run
Reset
Default
Save Data
Update
Exit
Specify Signal

Real-Time Control: Cart System

Readings
Time: 4.7 s
Reference: -8.37 cm
Actual: -14.66 cm
Control: -1.39 Volts

Settings
k 2.65
a 16.83
b 5.00
Use W ✓

Controller states
proportional 3.64
integral -1.89
derivative -0.36

$$u = \underbrace{k(a+b)\,e}_{\text{Proportional}} + \underbrace{kab\,\frac{e}{s}}_{\text{Integral}} + \underbrace{k\,se}_{\text{Derivative}}$$

Reference Command
User [✓ Generated Signal]
-40 cm 0 cm +40 cm

Figure 6.19: Real-Time Cart Position Control Application Main User Interface

per-second, this means that up to 400 seconds of experimental data can be stored at a time. The data is stored as a MATLAB m-file. To load this data in MATLAB, type the name of the file from the MATLAB command prompt . For example, if the name of the stored data file is `RtcCart.m`, typing *rtccart* on the MATLAB command line will load the data, assuming that MATLAB is in the correct current-directory. Use the *cd* command to change the current directory in MATLAB. Read the MATLAB on-line help for more information (type *help* from the MATLAB command line to access this). The following signals are stored:

- time (sec), in variable t,
- reference command (cm), in variable r,
- filtered reference command (cm), in variable rhat,
- control signal (applied motor voltage) (volts), in variable u, and
- output (actual cart position) (cm), in variable y.

6.4.5 Control Program Performance Limitations

The key performance criteria for the control program is the processing rate, that is, how many samples-per-second the control loop can process and output. The implemented controller reads in (position information) samples, calculates the control signal (motor voltage) from the error information, and outputs the control signal syncronously, and at the same rate. The nominal processing rate is 50 Hz, but may be reliably increased up to 1.7 KHz. The processing rate may be changed programmatically by altering the value of the `CONTROL_RATE` constant in `RtcCartDlg.cpp` (line 23). This processing rate is limited by several factors, including:

1. Windows timer resolution,
2. Windows-to-DAQ board communication speed,
3. Computer processor speed, and
4. Screen-update speed.

Windows is a multitasking operationg system, which means that in addition to executing the currently active application (i.e. the control program), it must also handle other functions such as user (keyboard and mouse) input, background processes (e.g. printer, network card, sound), graphical user interface (GUI) updates, and other inactive applications. The controller program is essentially implemented as a time-slice application, called by the Windows timer.

The Windows timer is an interrupt-driven countdown-timer that can be programmed to call a function at a fixed interval. The maximum resolution of this timer is 100 Hz (on the 266 MHz Pentium II, Windows NT 4.0 platform we used). The use of this timer is simple and it is part of the standard Windows event framework, and this is what was used initially. However, because of the 100 Hz limitation, a custom timer event mechanism was implemented using the Windows high-performance counter. This timer runs at a frequency of approximately 1 MHz, and so does not impose any restrictive limitations on the processing-rate. With this scheme, processing-rates of up to 1.7 KHz were reliably achieved (i.e. the control loop could sustain an average rate of up to 1.7 kHz). This rate is constrained by the processor speed, and the complexity of the control software.

The processing rate is also constrained by how fast the operating system can communicate input/output (I/O) data to the DAQ board. The reasons for this are documented within [129]. To test the maximum I/O capability of the software-control loop, an experiment was set up whereby a square-wave is output in a tight (infinite) loop. At each iteration, a variable is toggled between zero (0) and one (1), and this value is output on analog channel 0. The output of analog output channel 0 was hooked up to a digital oscilloscope, and the frequency of the resulting square-wave was measured to be 3.63 kHz. This means that the maximum processing-rate constraint of the "one-shot" output mode is 7.26 kHz (one voltage-level flip is performed at each iteration, thus the frequency of the square-wave, one period of which is two flips, is half the actual processing-frequency). When an analog-input call is also made at each iteration, the output square-wave frequency drops to 1.53 kHz (yielding a maximum processing-rate constraint of 3.06 kHz). One input and one-output is the minimum we require to implement a single-input-single-output (SISO) control system. Additional inputs or outputs will lower the maximum processing rate further. A 2-input 1-output system has a processing-rate limitation of 1.4 kHz, and a 2-input 2-output system has a limitation of 1.18 kHz. With the PID control algorithm implemented in the loop, the processing rate is constrained to 2.5 kHz. Adding joystick reference commands drops this rate to 2.0 kHz. This limitation reflects the limitation of the processor speed in executing the control loop and the speed of communicating with the DAQ board. For this experiment, it turns out that the real limiting-factor on the processing rate is the processor speed. This means that, even if a more sophisticated timer is implemented (and the overhead associated with the time-slice multitasking framework is completely eliminated), the processing rate cannot be pushed higher than 2.5 KHz.

The computer processor speed determines how many computer-instructions per second the system can execute, and limits the complexity of the control loop. Because the controller is implemented as a time-slice application, this means that each iteration of the control loop must finish its processing faster than the period of time allocated to it (i.e. 1 / processing rate). If it takes longer than this amount of time allocated to it, a timing overrun will occur, and the next iteration of the loop will not be called at the correct time. This will cause the actual processing rate to be lower than the specified rate. This can cause several problems if a fixed-time-step algorithm is employed. Firstly, if the derivative of the error is obtained by using a backwards-difference approximation assuming a fixed-time-step, the computed value for the derivative will be incorrect (it will be artificially higher). Second, the sampled data will be recorded at incorrect times. It was experimentally found that a processing rate of 1.7 kHz could be reliably achieved (this being the 1-input, 1-output, joystick input, and software-processing-time constraint).

Because of the above problems, a variable-step control routine was implemented. This means that at each control-loop iteration, the actual time-step (since the beginning of the last iteration) is measured (using the high-performance counter) and used. This eliminated derivative-approximation errors due to uneven time-steps, and records the correct sample-times.

The performance-limitation experimental results are summarized inTable 6.8.

Another issue is that, in addition to the periodic execution of the control-loop, there is a screen-update routine that is executed at regular intervals (10 Hz) as well. The execution of the screen-update rate can take a lot of time relative to the control-loop. This task involves (relatively) time-consuming operations such as writing to the screen-buffer, and updating the graphical user interface. Thus, if the processing-rate is set very high, each call to the screen-update routine will cause a very big overrun in the succeeding control-step. This one step will then have a longer sample-time (i.e. a lower sample-rate) than expected. In this experiment, a processing-rate of up to 100 Hz was found to be acceptable, causing less than 1 % of overrun time. In comparison, a 500 Hz processing rate setting would cause up to 4 % in overruns (this means that the control-loop would run at below the specified processing rate 4% of the time). The maximum processing rate achievable is approximately 1.9 kHz (with one analog output, one analog input, 10 Hz screen update rate, and a signal-generator reference command). Figure 6.20 shows the distribution of the actual processing-rates when the

Configuration	Average processing rate
One analog output in a tight loop, no screen update	7.26 kHz
One analog output and one analog input in a tight loop, no screen update	3.06 kHz
One analog output and two analog inputs in a tight loop, no screen update	1.40 kHz
Two analog outputs and two analog inputs in a tight loop, no screen update	1.18 kHz
One analog output, one analog input, and PID controller in a tight loop, no screen update	2.84 kHz
One analog output, one analog input, PID controller, and joystick control in a tight loop, no screen update	2.50 kHz
One analog output, one analog input, PID controller, signal generator reference, and 10 Hz screen update rate	1.9 kHz
One analog output, one analog input, PID controller, joystick control, and 10 Hz screen update rate	1.7 kHz

Table 6.8: Summary of performance limitations (running on a 266 MHz Pentium II with Windows NT 4.0)

processing rate is specified to be 10 kHz, and with a 10 Hz screen update rate.

6.4.6 Experimental Results

Using the save feature implemented in the C++ Real-Time Cart-Track control program, system response data was captured, saved to disk, and then plotted in MATLAB. In this section, we analyse the response of the system to step (square wave) commands and sinusoidal commands and compare them to the results predicted by the linear model. In particular, we will examine how the motor voltage saturation affects the accuracy of the predictions.

Step command following: Effect of different command amplitudes. Figure 6.29 shows the system response to a 0.2 Hz 10 cm square-wave command. From this plot, the measured 5 % rise-time is 0.4 seconds, the settling-time is under 1 second, and the overshoot is approximately 7.3 %. Except for the overshoot, these performance results agree very well with the approximate model and meet our design criteria. The overshoot is higher than predicted. This can be explained by motor parameter uncertainty (described below). The effects of nonlinearities in the system can greatly affect the response, particularly the effect of applied motor voltage saturation. The 10 cm square-wave command does not cause the motor voltage to saturate, but a 20 cm command does

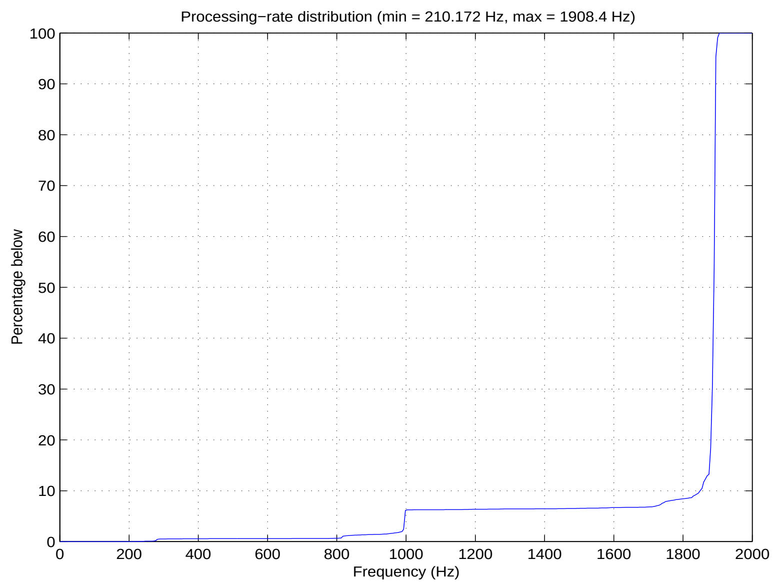


Figure 6.20: Processing-rate distribution

(Figure 6.30). This saturation causes the resulting overshoot to be approximately 35%. Figure 6.31 show the response to step commands of vaious sizes, clearly showing that larger commands have larger overshoots due to the motor operating in saturation for longer periods of time.

Step command following Effect of increasing the gain, k. The Real-Time Cart-Postion Control Environment was used to measure the system response to a square-wave reference command for different controller gains. Figures 6.21-6.28 show the system response to a 0.2 Hz 10 cm square-wave reference command for k = 0.265, 3.0, 3.5, 4.0, 5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 40.0, 50.0, 100.0, 1000.0, and 10000.0. the plots for k = 20.0 and above show that the motor voltage operates in saturation more than 90% of the time. The linear model predicts a finite upward-gain margin (estimated to be approximately 18.2 in Figure 6.4), indicating that the linear model would be unstable for k > approximately 50. However, the actual system does not go unstable, even for extremely large values of k.

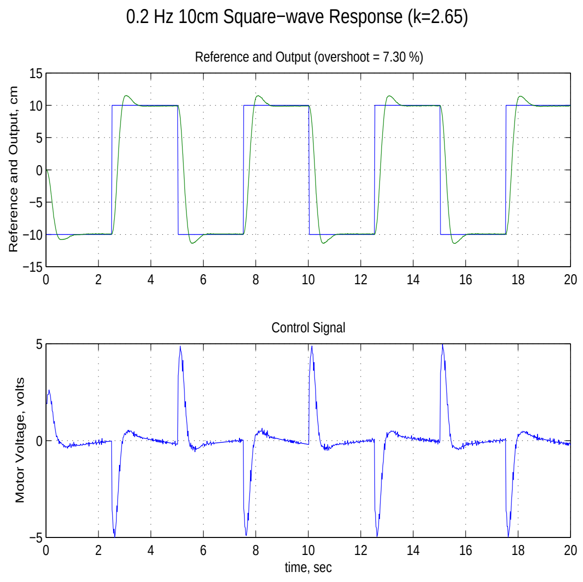


Figure 6.21: 0.2 Hz 10cm Square-wave Response (k=2.65)

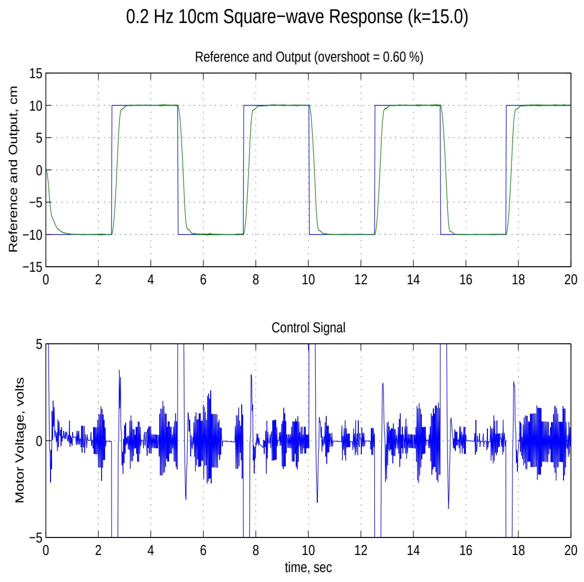


Figure 6.23: 0.2 Hz 10cm Square-wave Response (k=15.0)

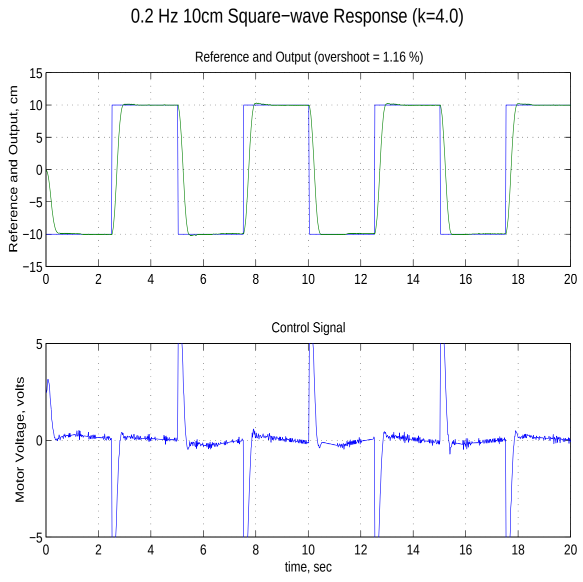


Figure 6.22: 0.2 Hz 10cm Square-wave Response (k=4.0)

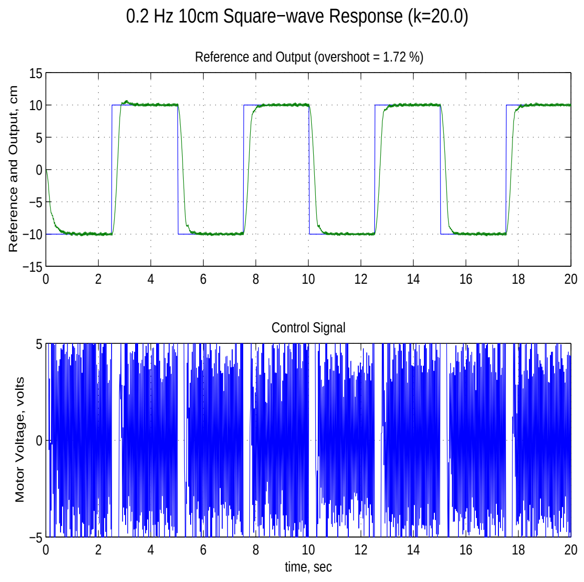


Figure 6.24: 0.2 Hz 10cm Square-wave Response (k=20.0)

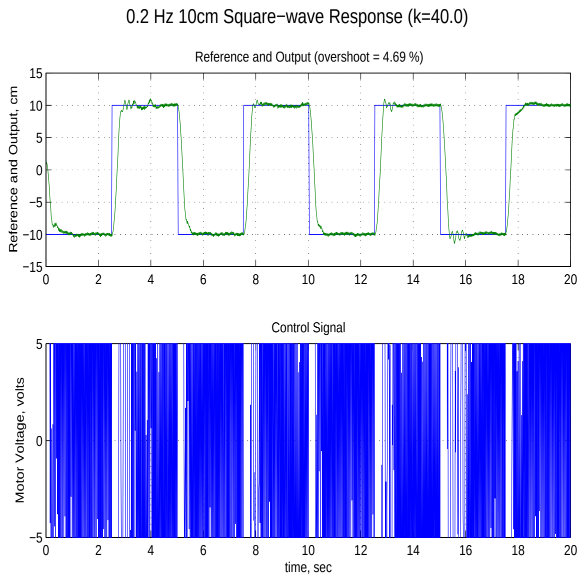


Figure 6.25: 0.2 Hz 10cm Square-wave Response (k=40.0)

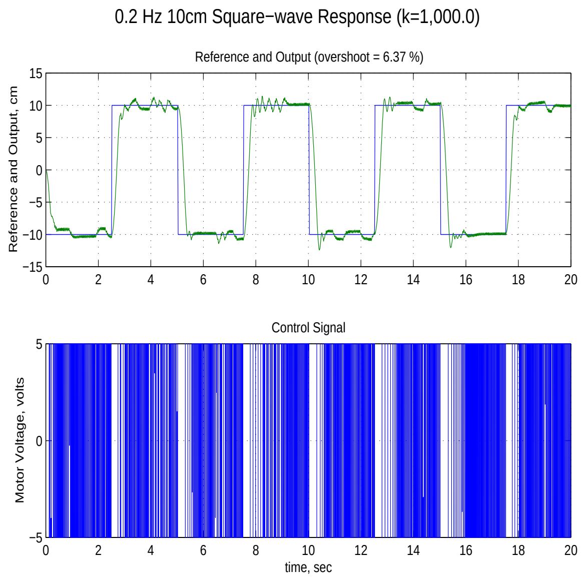


Figure 6.27: 0.2 Hz 10cm Square-wave Response (k=1,000.0)

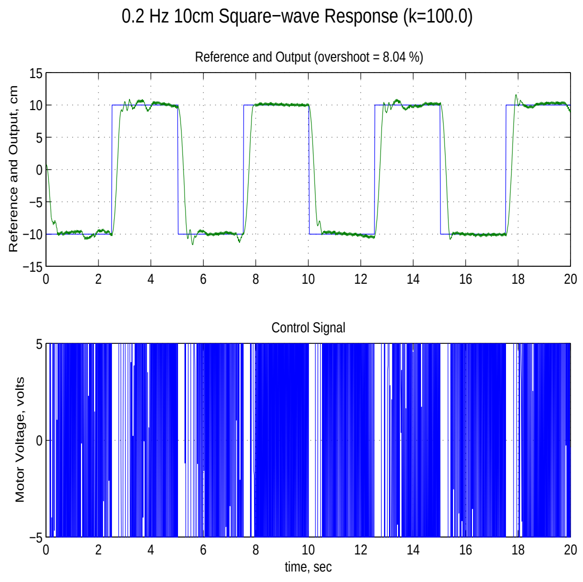


Figure 6.26: 0.2 Hz 10cm Square-wave Response (k=100.0)

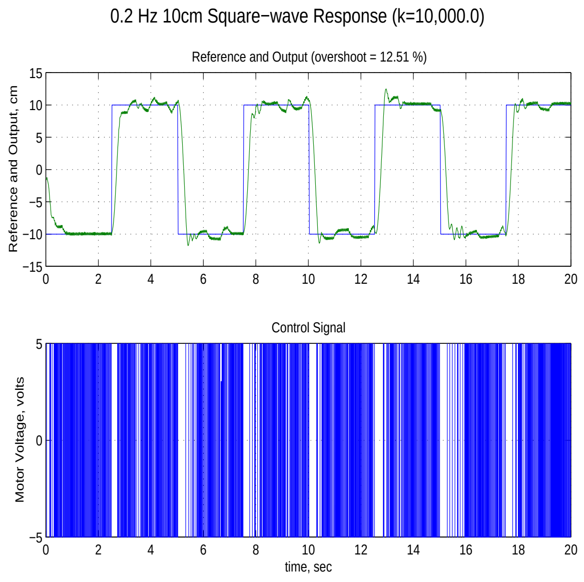


Figure 6.28: 0.2 Hz 10cm Square-wave Response (k=10,000.0)

Sinusoidal command following. A sinusoidal reference command was applied to the closed-loop system with an amplitude of 10 cm and frequencies of 0.2 Hz to 4.2 Hz in increments of 0.2 Hz. This amplitude was selected so as not to saturate the applied motor voltage. The output (cart position) responses were recorded, and the attenuation (or amplification) factors were measured. This data was then used to generate the frequency response plot shown in Figure 6.32, overlaid with the model-predicted response. However, it is seen that there is a significant discrepency between the experimental and theoretical results. This discrepency can be accounted for by system parameter uncertainty. The experimental frequency-response shows more attenuation than predicted at the higher frequencies (> 1.6 Hz). The breakpoint on the frequency-response plot occurs at a lower frequency than predicted by the model (which was at s = -16.8). This means that the pole location was incorrectly positioned. The correct this, the motor parameters were perturbed from their given values in order to better fit the model-predicted results to the experimental results. It was found that if the motor armature resistance (parameter R_m) were increased from 2.6 Ω to 3.8 Ω, the model-predicted and experimental data would match. This corresponds to stable pole due to the motor being at s = -11.5 rather than at s = -16.8.

This error in the pole-location also explains the discrepency in the step-response overshoot. The experimental overshoot was 7.3% while the original model predicted an overshoot of less than 5%. Because the pole is actually closer to the imaginary axis than predicted originally, it was not correctly cancelled by the zero in the compensator, and caused the poles at s $= -5 \pm j5$ to be pushed closer to the imaginary axis (and causing the dominant second-order response to be less damped). When the misplaced pole is moved to the correct location, the step response of the new model is 7.7%, which closely matches the experimental result.

Controller redesign. The controller was then redesigned to take into account the newly identified pole-location. Again, the goal was to place the dominant poles at $s = -5 \pm j5$. This is accomplished by selecting the following controller parameters:

$$a = 11.52 \tag{6.31}$$

$$b = 5 \tag{6.32}$$

$$k = 3.9 \tag{6.33}$$

The step response of this new closed-loop system is shown in Figure 6.33 (5 cm square-wave) and Figure 6.34 (10 cm square wave). These plots show that the overshoot is approximately 5%, which meets our design criteria.

A sinusoidal reference command was applied to the closed-loop system with an amplitude of 10 cm and frequencies of 0.2 Hz to 3.0 Hz in increments of 0.2 Hz, and for 4.0Hz. This amplitude was selected so as not to saturate the applied motor voltage. The output (cart position) responses were recorded, and the attenuation (or amplification) factors were measured. This data was then used to generate the frequency response plot shown in Figure 6.35.

Frequency (Hz)	Frequency (rad/sec)	Amplification (Theoretical)	Amplification (Actual) (10cm command)
0.2	1.26	1.00	1.00
0.4	2.51	0.99	1.00
0.6	3.77	0.98	1.00
0.8	5.03	0.93	0.99
1.0	6.28	0.85	0.93
1.2	7.54	0.74	0.80
1.4	8.80	0.63	0.67
1.6	10.05	0.53	0.52
1.8	11.31	0.44	0.41
2.0	12.57	0.37	0.31
2.2	13.82	0.31	0.25
2.4	15.08	0.26	0.19
2.6	16.34	0.22	0.15
2.8	17.59	0.19	0.13
3.0	18.85	0.17	0.11
3.2	20.11	0.15	0.09
3.4	21.36	0.13	0.07
3.6	22.62	0.11	0.06
3.8	23.88	0.10	0.05
4.0	25.13	0.09	0.05
4.2	26.39	0.08	0.04

Table 6.9: C++ Control: Closed-Loop System Frequency Response

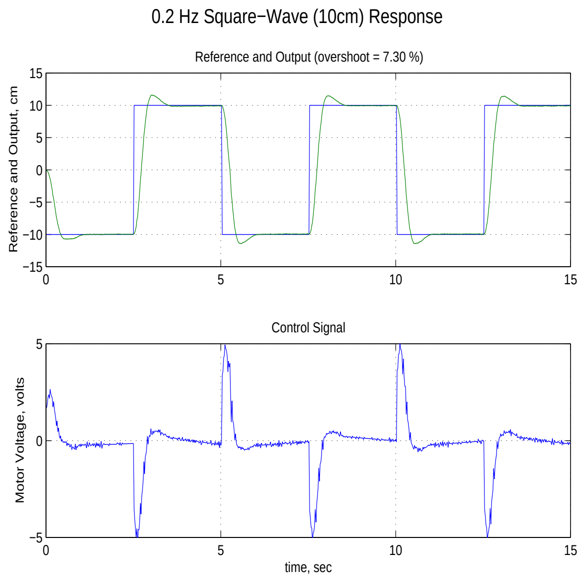


Figure 6.29: 0.2 Hz Square-Wave (10cm) Response

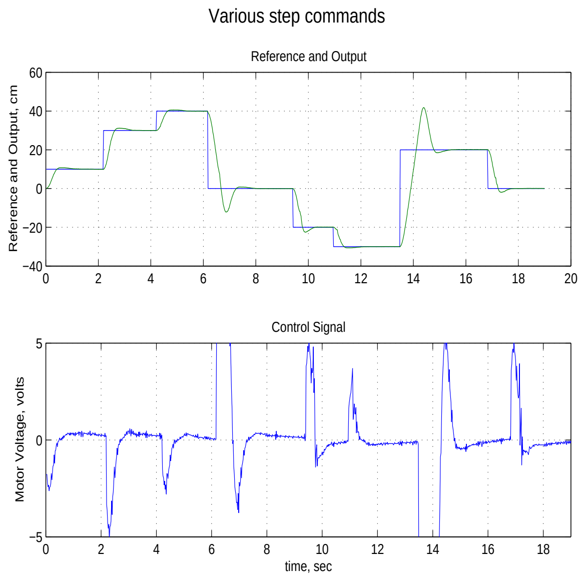


Figure 6.31: Various step commands

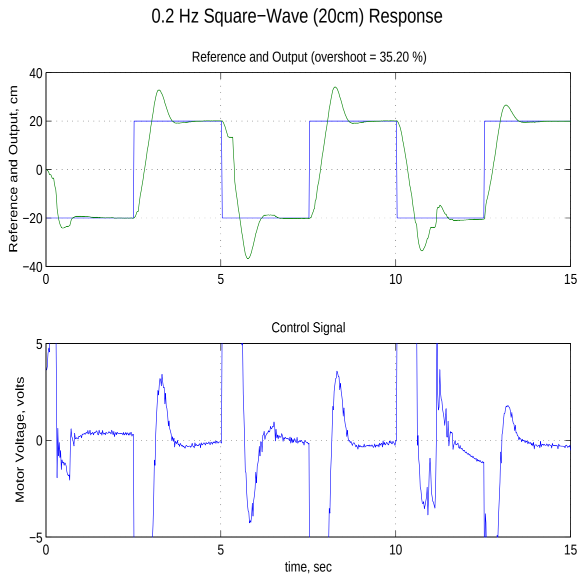


Figure 6.30: 0.2 Hz Square-Wave (20cm) Response

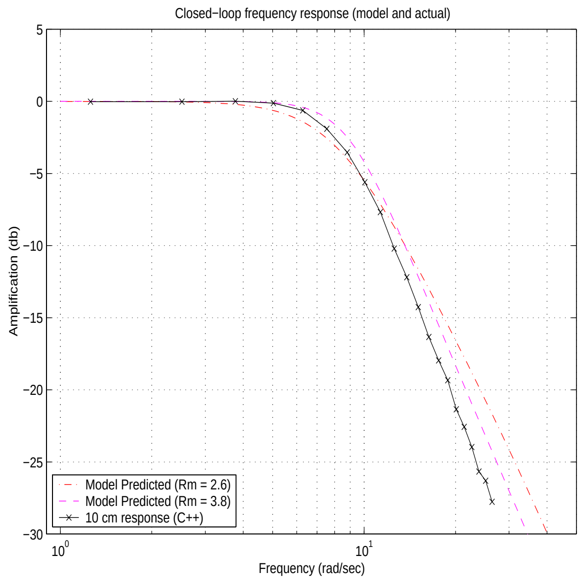


Figure 6.32: Closed-loop Frequency Response (Model and Actual)

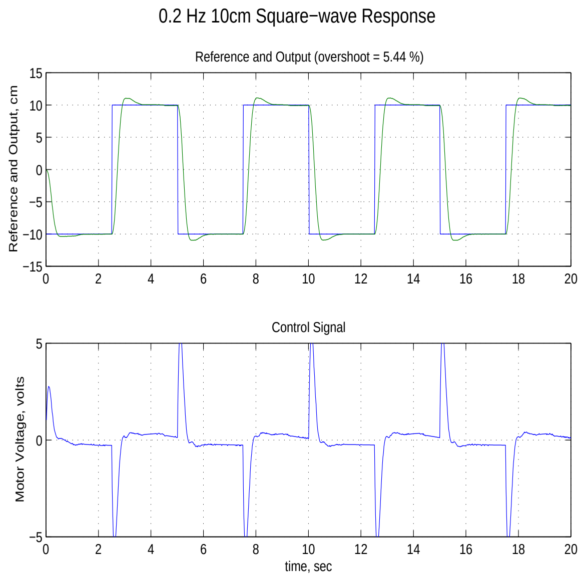


Figure 6.34: 0.2 Hz 10cm Square-wave Response

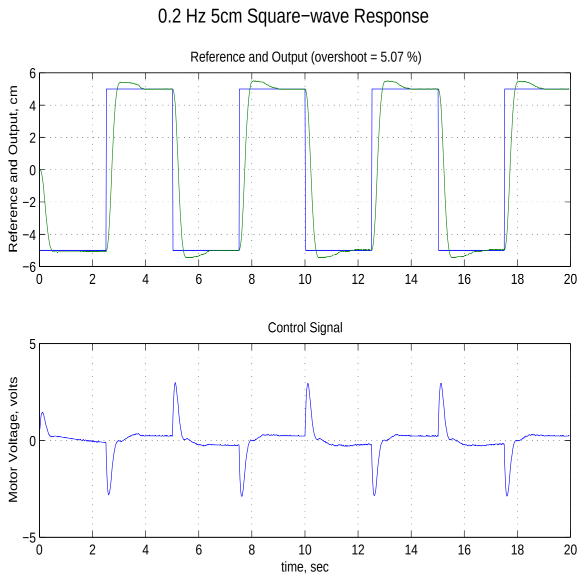


Figure 6.33: 0.2 Hz 5cm Square-wave Response

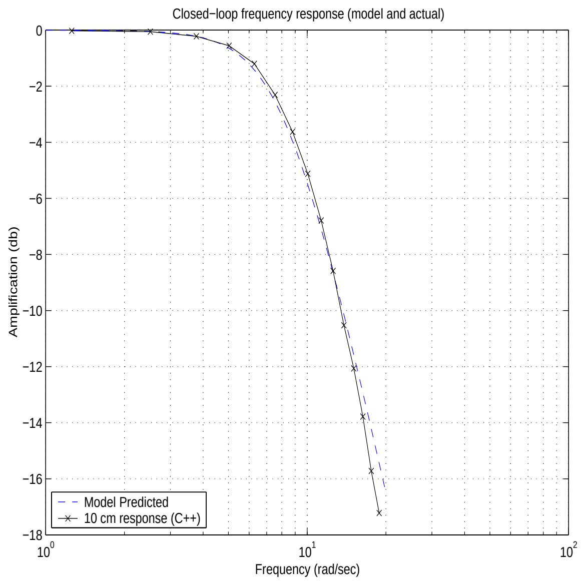


Figure 6.35: Closed-loop Frequency Response (Model and Actual) for redesigned control system

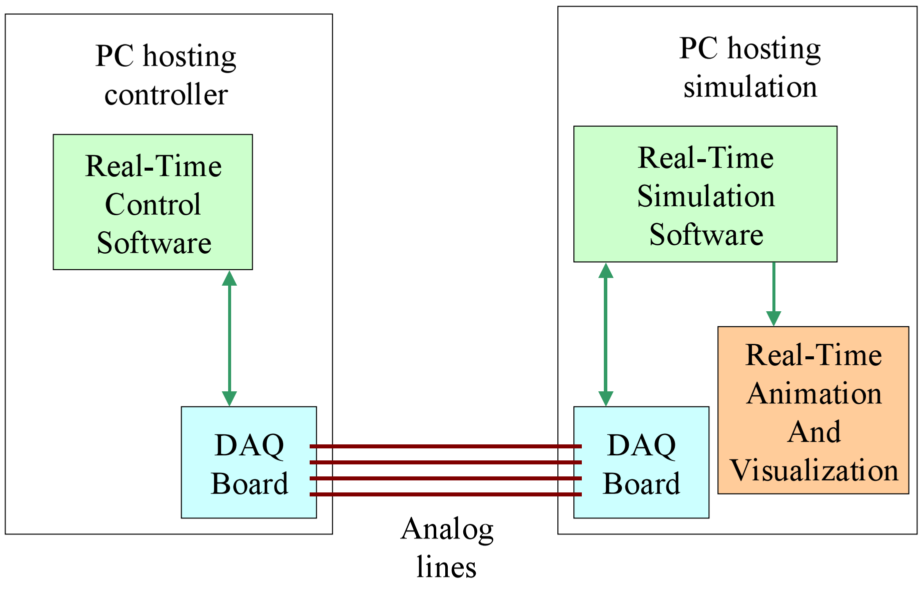

PC hosting controller
Real-Time Control Software
DAQ Board
Analog lines
PC hosting simulation
Real-Time Simulation Software
Real-Time Animation And Visualization
DAQ Board

Figure 6.36: V-Lab: Hardware Setup For PC-DAQ Control

6.5 Virtual-Lab (V-Lab)

Another application of the real-time hardware control technology is the creation of virtual hardware in-the-loop experimental setups. Instead of controlling the actual hardware, a simulation of the hardware system may be controlled instead. This is ideal for situations where access to the actual hardware is limited. It also serves to test the data acquisition and signalling hardware. An experimental setup featuring two PCs equiped with data acquisition boards is shown in Figure 6.36. The software controller is implemented on te first PC, while the virtual experiment, consisting of a data-acquisition board, a simulation of the plant, and animation/visualization features, is housed in the second PC. This virtual hardware may also be used to test other control system implementations, such as embedded systems (Figure 6.37).

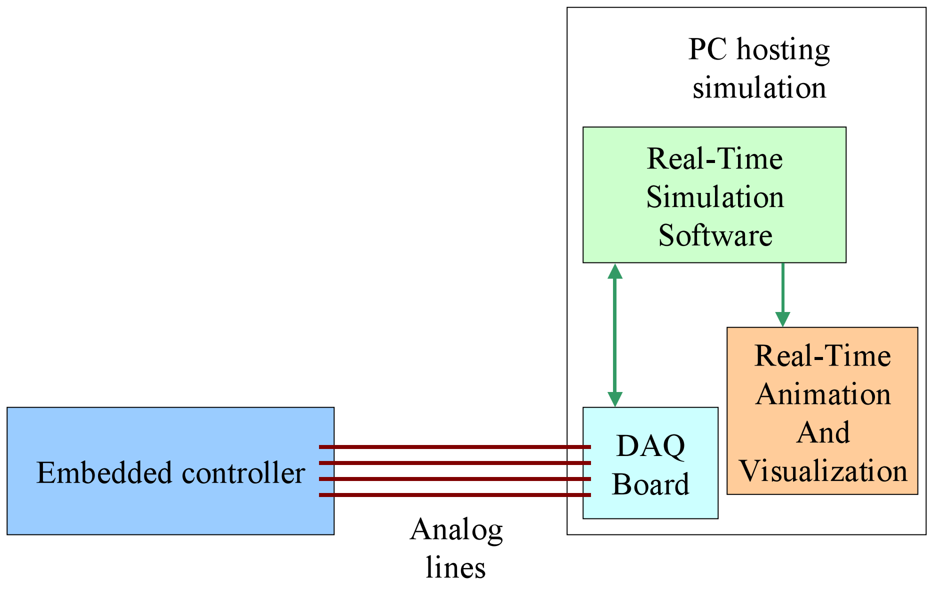

PC hosting simulation
Real-Time Simulation Software
Real-Time Animation And Visualization
DAQ Board
Embedded controller
Analog lines

Figure 6.37: V-Lab: Hardware Setup For Embedded System Control

CHAPTER 7

Summary and Future Directions

This thesis has described several Modeling, Simulation, Animation, and Real-Time Control (MoSART) tools. The utility of these tools for research, development, and education were demonstrated.

Future work will include the development of additional tools and the extension of the existing ones. Extensibility has been a key feature in the development of the tools. In particular, future work may involve the incorporation of additional simulation models, animation models and MIMO control laws (e.g. H_∞, μ-synthesis, gain scheduling, etc.). Some of this work has been initiated.

As the capability and affordability of processing power increases, we can only look forward to the more widespread use of more advanced MoSART tools.

REFERENCES

[1] R.Y. Chiang, M.G. Safanov,"H^∞ Synthesis Using Bilinear Pole Shifting Transform" *Journal of Guidance, Control, And Dynamics*, Vol. 15, No. 5, 1992 p. 1111-1117.

[2] P.M. DeRusso, R.J. Roy, C.M. Close, A.A. Desrochers, "State Variables for Engineers," *John Wiley and Sons,* 2nd Edition, 1998. 1995.

[3] Strang, G., *Linear Algebra and Its Applications,* 2nd Edition, Academic Press, 1980.

Helicopters and Aircraft

[4] J.H. Blakelock, *Automatic Control of Aircraft and Missiles (2nd Ed.)*, Wiley-Interscience, 1991.

[5] A.R.S. Bramwell, *Helicopter Dynamics*, John Wiley & Sons Inc., New York, NY, 1976.

[6] M. Ekblad, "Reduced Order Modeling and Controller Design for a High-Performance Helicopter," Journal of Guidance, Control, and Dynamics, Vol. 13, No. 3, 1990, pp.439-449.

[7] W.L. Garrard, E. Low, and S. Prouty, "Design of Attitude and Rate Command Systems for Helicopters Using Eigenstructure Assignment," Journal of Guidance, Control, and Dynamics, Vol. 12, No. 6, 1989, pp.783-791.

[8] L. Kaufman and E.R. Schultz, "The Stability and Control of Tethered Helicopters," *JAHS*, vol. 7, no. 4, October 1962.

[9] S. Osder and D. Caldwell, "Design and Robustness Issues for Highly Augmented Helicopter COntrols," Journal of Guidance, Control, and Dynamics, Vol. 15, No. 6, 1992, pp.1375-1380.

[10] R.W. Prouty, *Helicopter Performance, Stability, and Control*, PWS Engineering, Boston, MA, 1986.

[11] A.A. Rodriguez, *Multivariable Control of a Twin Lift Helicopter System using the LQG/LTR Design Methodology*, Master's Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1987.

[12] A.A. Rodriguez and M. Athans, "Multivariable Control of a Twin Lift Helicopter System using the LQG/LTR Design Methodology," Proceedings of the American Control Conference, Seattle, WA, June 18-20 1986, pp. 1325-1332, Invited Paper.

Interactive Modeling, Simulation, and Control Environments

[13] ASEE Project Report, "Engineering Education for A Changing World," *ASEE PRISM,* December, 1994, pp. 20-27.

[14] R.R. Barton and L.W. Schruben, "A New Graduate Course: Using Simulation Models for Engineering Design," *Simulation*, Vol. 64, No. 3, 1995, pp. 145–153.

[15] G. Bengu, "Computer-aided Education and Manufacturing Systems with Simulation and Animation Tools," *International Journal of Engineering Education*, Vol. 9, No. 6, 1993, pp. 484–494.

[16] Richard D. Braatz, Matthew R. Johnson, "Process Control Laboratory Education Using a Graphical Operator Interface," *Computer Applications in Engineering Education*, Vol. 6, No. 3, 1998, pp. 151–155.

[17] Francois Buret, Daniel Muller, Laurent Nicolas, "Computer-Aided Education for Magnetostatics,"*IEEE Transactions on Education,* Vol. 42, No. 1, February 1999, pp. 45–49.

[18] C.A. Cañizares, Z.T. Faur,"Advantages and Disadvantages of Using Various Computer Tools in Electrical Engineering Courses," *IEEE Transactions on Education*, Vol 40, No 3, 1997, pp. 166–171.

[19] Roy Crosbie, Lionel Brooks, "Simulation Software for Teaching Dynamic System Behavior," *Computer Applications in Engineering Education*, Vol. 5, No. 1, 1997, pp. 71–82.

[20] I. Cuiñas, A.M. Arias, A.G. Pino, A. Ramos, "Educational Software for Single- and Dual-Reflector Antennas," *Computer Applications in Engineering Education*, Vol. 7, No. 1, 1999, pp. 23–29.

[21] S.R. Cvetkovic, R.J.A. Seebold, K.N. Bateson, and V.K. Okretic, "CAL Programs Developed in Advanced Programming Environments for Teaching Electrical Engineering," *IEEE Transactions on Education,* Vol 37, No 2, May 1994, pp. 221-227.

[22] S.W. Director, P.K. Khosla, R.A. Rohrer, and R.A. Rutenbar, "Reengineering the Curriculum: Design and Analysis of a New Undergraduate Electrical and Computer Engineering Degree at Carnegie Mellon University,"*Proceedings of the IEEE,* Vol. 83, No. 9, September 1995, pp. 1246–1269.

[23] Francis J. Doyle III, Edward P. Gatzke, Robert S. Parker, "Practical Case Studies for Undergraduate Process Dynamics and Control Using Process Control Modules," *Computer Applications in Engineering Education*, Vol. 6, No. 3, 1998, pp. 181–190.

[24] A.J. Hejase and B.M. Hasbini, "Computer-Aided Instruction in Robotics," *International Journal of Engineering Education*, Vol 9, No 2, 1993, pp. 156-161.

[25] H. Higuchi and E.F. Spina, "Improving Undergraduate Engineering Education Through Scientific Visualization," *ASEE Annual Conference Proceedings*, Vol 2, Session 3226, 1993, pp. 1581–1583.

[26] M.F. Iskander, "Benefits of Virtual Teaching," *Computer Applications in Engineering Education*, Editorial, Vol. 5 (1) pp. 1–2, 1997.

[27] R.G. Jacquot, D.A. Smith, D.L. Whitman, "Animation Software for Enhancement of Lectures in Engineering Dynamics," *ASEE Annual Conference Proceedings*, Vol. 2, Session 3220, 1993, pp. 1529-1531.

[28] Gordana Jovanovic-Dolecek, "RANDEMO: Educational Software for Random Signal Analysis," *Computer Applications in Engineering Education*, Vol. 5, No. 1, 1997, pp. 93–97.

[29] Jus Kocijan, John O'Reilly, William E. Leithead, "An Integrated Undergraduate Teaching Laboratory Approach to Multivariable Control,"*IEEE Transactions on Education,* Vol. 40, No. 4, November 1997, pp. 266–272.

[30] T.W. Lam, Y.T. Wong, C.L. Lam, "Effective Teaching and Learning of Holographic Interferometry by a Computer Simulation Package," *Computer Applications in Engineering Education*, Vol. 6, No. 4, 1998, pp. 235–243.

[31] A.S. Lau,"Broadening student perspectives in engineering design courses" *IEEE Technology and Society Magazine*, Vol. 17 no3 Fall '98 p. 18-23.

[32] Heng Li, "Information-Technology-Based Tools for Reengineering Construction Engineering Education," *Computer Applications in Engineering Education*, Vol. 6, No. 1, 1998, pp. 15–21.

[33] Valentin Loyo-Maldonado, Rene DE J. Romero-Troncoso,"Interactive Tool in Silicon Semiconductor Modeling: SSMod," *Computer Applications in Engineering Education*, Vol. 6, No. 2, 1998, pp. 59–65.

[34] V.S. Pantelidis, "Virtual Reality and Engineering Education," *Computer Applications in Engineering Education*, Vol 5, No 1, 1997, pp. 3–12.

[35] Anton J. Pintar, David W. Caspary, Tomas B. Co, Edward R. Fisher, Nam K. Kim, "Process Simulation and Control Center: An Automated Pilot Plant Laboratory," *Computer Applications in Engineering Education*, Vol. 6, No. 3, 1998, pp. 145–150.

[36] A.A. Rodriguez, "Practical Systems and Controls with Interactive Simulation, Visualization, and Animation," 1998. See http://www.eas.asu.edu/~aar/.

[37] H.A. Smolleck, D.S. Dweyer, and L.M. Rust, "On-Screen Synthesis and Analysis of Harmonics: A Student-Oriented Package," *IEEE Transactions on Education,* Vol. 38, No. 3, August 1995, pp. 243–250.

[38] J.M. Taboada, J.L. Rodriguez, L.Landesa, F. Obelleiro, "Automatic Wire-Grid Generation for Electromagnetic Analysis of Arbitrary-Shaped Conducting Bodies by NEC," *Computer Applications in Engineering Education*, Vol. 6, No. 2, 1999, pp. 31–42.

[39] Mi-Ching Tsai, Chung-Chi Chou, Min-Fu Hsieh, "Development of a Real-Time Servo Control Test Bench ,"*IEEE Transactions on Education,* Vol. 40, No. 4, November 1997, pp. 242–252.

Interactive Animation Environments

[40] DADs/Plant, CADSI Product, Donation to ASU, 1998.

[41] SimMaster3D, Simulogix, www.simulogix.com, 1999.

[42] Working Model User's Manual, Knowledge Revolution, 1989, http://www.krev.com, info@krev.com.

[43] *Simulation News,* The Working Model Newsletter, No. 2, Vol. 1, Fall 1995.

Interactive MoSART Environments at ASU

[44] M.F. DeHerrera and A.A. Rodriguez, "Trying to 'Shoot' an Evasive Monkey: A Tool for Designing and Evaluating Adaptive Learning Algorithms," *Proceedings of the 1996 International Conference on Simulation in Engineering Education*, San Diego, CA, January 14-17, 1996, pp. 31-36.

[45] M.F. DeHerrera, A.A. Rodriguez, and R.P. Metzger Jr, "Teaching Systems and Controls Using a MATLAB-Based Interactive Environment," *Proceedings of the 1997 International Conference on Simulation in Engineering Education*, Phoenix, AZ, January 12-15, 1997, pp. 71-76.

[46] M.F. DeHerrera, A.A. Rodriguez, R.P. Metzger Jr, and D. Cartagena, "Modeling, Simulation, and Graphical Visualization of a Liquid Level control System," *Proceedings of the 1997 International Conference on Simulation in Engineering Education*, Phoenix, AZ, January 12-15, 1997, pp. 57-62.

[47] J.I. Hernandez-Sanchez, "Development of an Extensible Interactive Modeling, Simulation, Animation, and Real-Time Control (MoSART) Flexible Inverted Pendulum Environment: A Tool for Enhancing Research and Education", Arizona State University, Masters Thesis, December, 1999.

[48] J. Koenig and A.A. Rodriguez "Development of An Interactive Modeling, Simulation, Animation, and Real-Time Control (MoSART) Tilt Wing Rotorcraft Environment and Testbed," Presented and Distributed at *IEEE Southwest Regional Conference,* San Diego, CA, April, 1999, 22 pages, Best Paper Award.

[49] S.S. Kwak, C.I Lim, R.P. Metzger, and A.A. Rodriguez, "Development Of An Interactive MoSART Submarine System Environment," *Proceedings of the 1999 ICSEE,* San Francisco, CA, January 17-20, 1999, pp. 187-192, Invited Session.

[50] S.S. Kwak, C.I. Lim, Richard Metzger, and A.A. Rodriguez, "Multivariable Submarine Control System Analysis and Design Using an Interactive Visualization Tool," *Proceedings of the 1999 American Control Conference,* San Diego, CA, June 2-4, 1999, 6 pages.

[51] S.S. Kwak, "Development of an Extensible Interactive Modeling, Simulation, Animation, and Real-Time Control (MoSART) Submarine Environment: A Tool for Enhancing Research and Education," Arizona State University, Masters Thesis, December, 1999.

[52] C.I. Lim, "Development of an Extensible Interactive Modeling, Simulation, Animation, and Real-Time Control (MoSART) Helicopter Environment: A Tool for Enhancing Research and Education," Arizona State University, Masters Thesis, December, 1999.

[53] C.I. Lim, R.P. Metzger, and A.A. Rodriguez "Development of Interactive Modeling, Simulation, Animation, and Real-Time Control (MoSART) Environments: Tools For Enhancing Research and Education," Presented and Distributed at *1998 Frontiers In Education Conference,* Tempe, AZ, November, 1998, 6 pages.

[54] C.I. Lim and A.A. Rodriguez "An Interactive Modeling, Simulation, Animation, and Real-Time Control (MoSART) Helicopter Environment," Presented and Distributed at *IEEE Southwest Regional Conference,* Albuquerque, NM, April, 1998, 20 pages, Best Paper Award.

[55] C.I. Lim and A.A. Rodriguez, "An Interactive Modeling, Simulation, Animation, and Real-Time Control (MoSART) Helicopter Environment: A Tool For Enhancing Research and Education," *Proceedings of the WESCON 98 IEEE Conference,* Anaheim, CA, September, 1998, pp. 327-347, Best Paper Award.

[56] C.I. Lim and A.A. Rodriguez "An Interactive Modeling, Simulation, Animation, and Real-Time Control (MoSART) Helicopter Environment: A Tool for Enhancing Education and Research," to appear in the *Proceedings of the 1998 American Control Conference,* Philadelphia, PA, June, 1998.

[57] C.I. Lim and A.A. Rodriguez, "An Interactive Modeling, Simulation, Animation, and Real-TIme Control (MoSART) Helicopter Environment," *Proceedings of the* 37th *Conference on Decision and Control,* Tampa, FL, December, 1998, pp. 3659-3660.

[58] C.I. Lim and A.A. Rodriguez, "Development Of An Interactive MoSART Twin Lift Helicopter System (TLHS) Environment," *Proceedings of the 1999 ICSEE,* San Francisco, CA, January 17-20, 1999, pp. 193-198, Invited Session.

[59] C.I. Lim, T.Y. Kim, S.S. Kwak, R.P. Metzger, and A.A. Rodriguez, "Interactive Modeling, Simulation, Animation and Real-Time Control (MoSART) Vehicle-Based Environments," *Proceedings of the 1999 ASEE/PSW Conference,* Las Vegas, NV, March 19-20, 1999, pp. 183-192, Invited Session.

[60] C.I. Lim, R.P. Metzger, and A.A. Rodriguez, "An Interactive Modeling, Simulation, Animation and Real-Time Control (MoSART) Twin Lift Helicopter System (TLHS) Environment," *Proceedings of the 1999 American Control Conference,* San Diego, CA, June 2-4, 1999, 6 pages.

[61] R.P. Metzger, "Development of an Extensible Interactive Modeling, Simulation, Animation, and Real-Time Control (MoSART) Robotic Systems Environment: A Tool for Enhancing Research and Education," Arizona State University, Masters Thesis, December, 1999.

[62] R.P. Metzger, K.J. Elliott, and A.A. Rodriguez, "Modeling, Analysis, and Graphical Visualization of a Dual Robot Arm System: A PC Based Environment," *Proceedings of the 1996 International Conference on Simulation in Engineering Education*, San Diego, CA, January 14-17, 1996, pp. 175-180.

[63] R.P. Metzger Jr., A.A. Rodriguez, "Modeling, Simulation, Animation, and Control for a Single Robotic Manipulator," *Proceedings of the 1997 International Conference on Simulation in Engineering Education*, Phoenix, AZ, January 12-15, 1997, pp. 77-82.

[64] R.P. Metzger Jr., A.A. Rodriguez, R. Aguilar, C.I. Lim, "Teaching Control System Concepts Using a Virtual Inverted Pendulum Environment," *Proceedings of the 1997 International Conference on Simulation in Engineering Education*, Phoenix, AZ, January 12-15, 1997, pp. 134-139.

[65] R.P. Metzger, C.I. Lim, and A.A. Rodriguez "Interactive Modeling, Simulation, Animation, and Real-Time Control (MoSART) Environments," to appear in the *Proceedings of the 1998 FIE,* Phoenix, AZ, November, 1998.

[66] R.P. Metzger, C.I. Lim, and A.A. Rodriguez, "On The Development Of Interactive MoSART Environments: The Microsoft Software Suite," *Proceedings of the 1999 ICSEE,* San Francisco, CA, January 17-20, 1999, pp. 181-186, Invited Session.

[67] R.P. Metzger, Chen-l Lim, and A.A. Rodriguez, "Development Of An Interactive MoSART Multiple Pendulum Environment," *Proceedings of the 1999 ICSEE,* San Francisco, CA, January 17-20, 1999, pp. 199-204, Invited Session.

[68] R.P. Metzger, C.I. Lim, and A.A. Rodriguez, "On the Development Of Interactive MoSART Environments," *Proceedings of the 1999 ASEE/PSW Conference,* Las Vegas, NV, March 19-20, 1999, pp. 157-162, Invited Session.

[69] R.P. Metzger, C. Rios, J. Hernandez, and A.A. Rodriguez, "Interactive MoSART Pendulum Environments," *Proceedings of the 1999 ASEE/PSW Conference,* Las Vegas, NV, March 19-20, 1999, pp. 163-172, Invited Session.

[70] R.P. Metzger and A.A. Rodriguez, "An Interactive MoSART Robotic Manipulator Environment," *Proceedings of the 1999 ASEE/PSW Conference,* Las Vegas, NV, March 19-20, 1999, pp. 173-182, Invited Session.

[71] C. Rios, R.P. Metzger, and A.A. Rodriguez, "Development Of An Interactive MoSART Cart-Pendulum-Seesaw Environment," to appear in the *Proceedings of the 1999 ICSEE,* San Francisco, CA, January 17-20, 1999, pp. 205-210, Invited Session.

[72] C. Rios, "Development of an Extensible Interactive Modeling, Simulation, Animation, and Real-Time Control (MoSART) Cart-Pendulum-Seesaw Environment: A Tool for Enhancing Research and Education", Arizona State University, Masters Thesis, August, 1999.

[73] Carlos Rios, Chen-I Lim, Richard Metzger, A.A. Rodriguez, "Multivariable Control of a Cart-Pendulum-Seesaw (CPS) System: Comparisons and Tradeoffs," to appear in the *Procedings of the* 38th *Conference on Decision and Controls,* Phoenix, AZ, 1999, 6 pages.

[74] M. Roberts, M.F. DeHerrera, and A.A. Rodriguez, "The Evasive Monkey: An Environment for Evaluation Adaptive Learning Algorithms," *Proceedings of the 1997 International Conference on Simulation in Engineering Education*, Phoenix, AZ, January 12-15, 1997, pp. 51-56.

[75] A.A. Rodriguez and R. Aguilar, "Graphical Visualization of Missile-Target Air-to-Air Engagements: An Educational Tool for Designing and Evaluating Missile Guidance and Control Systems," *Journal of Computer Applications in Engineering Education*, Vol. 3, No. 1, 1995, pp. 5-20.

[76] A.A. Rodriguez, M.F. DeHerrera, and R.P Metzger, "An Interactive Matlab-Based Tool for Teaching Classical Systems and Controls," Proceedings of the 1996 Conference on *Frontiers In Education,* Salt Lake City, Utah, Nov 6-9, 1996.

[77] A.A. Rodriguez and M.F. DeHerrera, "Modeling, Simulation, and Graphical Visualization of a Twin Lift Helicopter System Under Automatic control: An Educational Tool," Proceedings of the 1996 Conference on *Frontiers In Education,* Salt Lake City, Utah, Nov 6-9, 1996.

[78] A.A. Rodriguez and C.I. Lim, "Interactive Environments for Teaching Systems and Controls: The Need for Educational Tools," *Proceedings of the 1997 International Conference on Simulation in Engineering Education*, Phoenix, AZ, January 12-15, 1997, pp. 9-14.

[79] A.A. Rodriguez, T.Y. Kim, R.P. Metzger, and C.I. Lim, "MoSART: A Unifying Paradigm For The Development Of Advanced Interactive Research And Education Tools," *Proceedings of the 1999 ICSEE,* San Francisco, CA, January 17-20, 1999, pp. 175-180, Invited Session.

[80] A.A. Rodriguez, T.Y. Kim, C.I. Lim, and R.P. Metzger, "A Tool For Analyzing, Designing, and Visualizing Multivariable Aircraft Control Systems," *Proceedings of the 1999 American Control Conference,* San Diego, CA, June 2-4, 1999, 6 pages.

[81] A.A. Rodriguez, Chen-I Lim, Kevin Hicks, Richard Metzger, "Multivariable Control of a Helicopter System Using an Interactive Modeling, Simulation, and Visualization Tool," to appear in the *Proceedings of the* 38th *Conference on Decision and Controls,* Phoenix, AZ, 1999, 6 pages.

Control System Design Methodologies

[82] S.P. Boyd and C.H. Barratt, *Linear Controller Design: Limits of Performance,* Prentice Hall, Englewood Cliffs, NJ, 1991.

[83] J.J. D'Azzo and C.H. Houpis, *Linear Control System Analysis & Design,* McGraw Hill, Third Edition, New York, 1988.

[84] R.C. Dorf, *Modern Control Systems,* Sixth Edition, Reading, MA, Addison-Wesley, 1992.

[85] G. Franklin, J.D. Powell, A. Emami-Naeini, *Feedback Control of Dynamic Systems,* Addison-Wesley, Third Edition, Reading, MA, 1994.

[86] B.C. Kuo, *Automatic Control Systems*, Sixth Edition, Upper Saddle River, N.J., Prentice Hall, 1991.

[87] K. Ogata, *Modern Control Engineering,* Prentice-Hall, Third Edition, Upper Saddle River, N.J., 1997.

[88] A.A. Rodriguez, "Missile Guidance," *Encyclopedia of Electrical and Electronics Engineering,* Wiley Publishing Company, Vol. 13, pp. 316-325, 1999.

[89] C.E. Rohrs, J.L. Melsa, and D.G. Schultz, *Linear Control Systems,* McGraw Hill, 1993.

[90] Vidyasagar, M., *Nonlinear Systems Analysis,* Prentice Hall, 2$^{\text{nd}}$ Edition, 1993.

[91] S.C. Warnick and A.A. Rodriguez, "A Systematic Anti-windup Strategy and the Longitudinal Control of a Platoon of Vehicles with Control Saturations," to appear in the *IEEE Transactions on Vehicular Technology*, 1999 (10 pages).

[92] Zhou, K., Doyle, J.C., and Glover, K., *Robust and Optimal Control*, Prentice Hall, 1996.

[93] Zhou, K., with Doyle, J.C., *Essentials of Robust Control*, Prentice Hall, 1998.

CAD Software Technologies for Controls

[94] C. Borghesani, *Controls Tutor Using MATLAB,* PWS, 1996.

[95] D. Etter, *Introduction to MATLAB for Engineers and Scientists,* Prentice Hall, 1996.

[96] D.K. Frederick and J.H. Chow, *Feedback Control Problems Using MATLAB and the Control Systems Toolbox,* PWS, 1995.

[97] A. Grace, *Optimization Toolbox for use with Matlab* , The Mathworks, Inc. Natick, MA, 1990.

[98] Duane Hanselman, Bruce Littlefield,*MATLAB User's Guide,* Prentice Hall, Upper Saddle River, NJ, 1997.

[99] *Control System Toolbox Version 4,* MathWorks Inc, 1996.

[100] R. Chiang and M. Safonov. *Robust Control Toolbox Version 2,* MathWorks Inc, 1998.

[101] L. Ljung, *System Identification Toolbox: User Guide,* MathWorks Inc, 1991.

[102] *MATRIX-X User's Guide,* Integrated Systems Inc., Donation to ASU, 1998.

[103] *MATLAB Based Books,* MathWorks Inc., Natick, Massachusetts, Spring 1996, http://www.mathworks.com, info@mathworks.com.

[104] C. Norman and J. Kinchen, "New Support for ActiveX in MATLAB 5.2, " MATLAB News and Notes, Summer 1998, pp. 6-7.

[105] *SIMULINK User's Guide,* SIMULINK: A Program for Simulating Dynamic Systems, MathWorks Inc., Natick, Massachusetts, 1990, info@mathworks.com.

[106] *Writing S-Functions,* SIMULINK: Dynamic System Simulation for MATLAB, MathWorks Inc., Natick, Massachusetts, October 1998, info@mathworks.com.

[107] B.C. Wheeler, “Matlab Exercises to Demonstrate Principles of Biological Modeling and Signal Processing,” *ASEE Annual Conference Proceedings*, Vol. 2, Session 3509, 1993, pp. 1789–1790.

[108] The Mathworks, “MATLAB & SIMULINK,”http://www.mathworks.com/.

C++, MFC, Windows NT, and Direct-3D

[109] B. Aaron and B. Thompson, “ActiveX,” *Prima Publishing,* 1996.

[110] J. Anderson, “ActiveX Programming with Visual C++,” *Que,* 1997.

[111] B. Bargen and P. Donnelly, “Inside DirectX”, Microsoft Press, Redmond Washington, 1998.

[112] N. Barkakat, *X Window System Programming*, Sams Publishing, Indianapolis, IN, 1994, second ed.

[113] B. Glazier, “The ‘Best Principle:’ Why OpenGL is Emerging as the 3D Graphics Standard,” *Computer Graphics World*, April 1992, pp. 116-117.

[114] K.J. Goodman, *Windows NT: A Developer's Guide*, M&T Books, New York, New York, 1994.

[115] T.R. Halfhill, “Inside the Mind of Microsoft,” *Byte*, August 1995, pp. 48-52.

[116] K. Husain and K. Wusain, “ActiveX Developer's Resource,” *Prentice Hall,* 1997.

[117] P.J. Kovach, “The Awesome Power of Direct3D/DirectX”, Manning Publishing, August, 1997.

[118] S. Lippman, et. al., “C++ Primer (3rd Ed.)”, Addison-Wesley, April, 1998.

[119] N. Nicolaisen, *The Visual Guide to Visual C++*, Ventana Press, 1995.

[120] P. Norton and J. Mueller, *Peter Norton's Complete Guide to Windows 95*, Sams Publishing, Indianapolis, IN, 1995.

[121] C. Petzold and P. Yao, “Programming Windows 95 (4th Ed.)” *Microsoft Press,* 1996.

[122] J. Prosise, “Programming Windows with MFC (2nd Ed.),” *Microsoft Press,* May, 1999.

[123] M. Root and J. Boer, “DirectX Complete,” *McGraw-Hill,* 1998.

[124] D.Rogerson, “Inside COM,” *Microsoft Press,* February, 1997.

[125] S. Trujillo, “Cutting Edge Direct 3D Programming,” *Coriolis Group books,* 2nd Edition, 1996.

Data Acquisition and Real-Time Hardware Control

[126] National Instruments, *Lab PC-1200/AI User Manual, July 1998 Edition*, Part Number 321230B-01, National Instruments Corporation, 1998.

[127] National Instruments, *LabVIEW 5 User Manual, January 1998 Edition*, Part Number 320999B-01, National Instruments Corporation, 1998.

[128] National Instruments, *Ni-DAQ Function Reference Manual for PC Compatibles, February 1997 Edition*, Part Number 321451A-01, National Instruments Corporation, 1997.

[129] National Instruments, *Ni-DAQ User Manual for PC Compatibles, Version 6.5*, Part Number 321644D-01, National Instruments Corporation, October 1998.

APPENDIX A

MATLAB Script Listings

$\mathcal{H}^\infty$ Controller Design script

```
%===================================================================================
%
% 8th order Apache Helicopter
%
% model is from Gerrard, Low, and Prouty, "Design of Attitude and Rate Command Systems
% for Helicopters Using Eigenstructure Assignment", Journal of Guidance,
% Nov.-Dec. 1989
%
% H-infinity controller design
%
% Chen-I Lim
% 10-20-99
%
%===================================================================================


close all
clear
clc

bUseBilin = 1;
bInvdc = 1;

sysname = '8th order Apache Model from Garrard et. al.';

%
% Plant
%
% x = [u, v, w, p, q, r, phi, theta] (non-dimensionalized)
%
% u = [collective pitch, logitudinal cyclic, lateral cyclic, tail-collective] (nondimentionalized)
%
a = [
 -0.0199, -0.0058, -0.0058, -0.0125882, 0.0193408, 0.000500194, 0, -0.554549;
 -0.0452, -0.0526, -0.0061, -0.0216751, -0.0129217, 0.0123381, 0.554215, -0.000250097;
 -0.0788, -0.0747, -0.3803, 0.000666926, -0.00400155, 0.0350136, 0.0190074, 0.0085033;
 0.546628, -3.11195, -0.214357, -2.9979, -0.5308, 0.4155, 0, 0;
 0.442388, 0.23163, -0.210278, 0.071, -0.5943, 0.0013, 0, 0;
 1.31217, 0.876859, -0.0429433, 0.4058, 0.4069, -0.494, 0, 0;
 0, 0, 0, 1, 0.0005, -0.0154, 0, 0;
 0, 0, 0, 0, 0.9994, 0.0343, 0, 0;
];

b = [
    0.526216,   -0.0158127, -0.0844613, -0.00144173;
    0.00955747, 0.530582,   -0.068347,  0.324389;
    0.00844613, -0.00609646,    -5.76523,   0.000180216;
    1.35921,    47.6086,    -5.38587,   9.93359;
    -8.44227,   0.580005,   -0.712088,  -0.0700408;
    3.177,  -5.98927,   12.8618,    -11.9874;
    0,  0,  0,  0;
    0,  0,  0,  0;
];

%
% y = [u+theta,v+phi,w,r]'
%
alpha = -1.3;
beta = 1.4;
c = [
    1 0 0 0 0 0 0 alpha;
    0 1 0 0 0 0 beta 0;
    0 0 1 0 0 0 0 0;
    0 0 0 0 0 1 0 0;
 ];

d = zeros(size(c,1),size(b,2));

ap = a;
bp = b;
cp = c;
dp = d;

clear a b c d;
psys = ss(ap,bp,cp,dp);

```

```
%
% Design plant
%
ag = ap;
bg = bp;
cg = cp;
dg = dp;

%
% Bilinear transform
%
p2 = -100;
p1 = -0.4;
if bUseBilin
   [ag,bg,cg,dg] = bilin( ag,bg,cg,dg, 1, 'S_ftjw', [p2, p1] );
end


%---------------------------------------------------------------------------------------
%
%   Poles and zeros
%
%---------------------------------------------------------------------------------------
fprintf( '%s: Transmission zeros\n', sysname );
sysptzero = tzero( psys )

fprintf( '%s: Pole-zero map\n', sysname );
disp( '8th order Apache Model: Pole-zero map')
pzmap( psys )
grid
title( sprintf( '%s: Poles and Zeros', sysname) );


%---------------------------------------------------------------------------------------
%
%   System DC Gains
%
%   Output:  y = [th w ph r]'
%
%---------------------------------------------------------------------------------------
fprintf( '%s: DC Gain\n', sysname );
dcg = cp*inv(-ap)*bp
%[ u, s, v ] = svd(dcg);


%---------------------------------------------------------------------------------------
%
%   System Transfer Functions
%
%---------------------------------------------------------------------------------------
if 0
    fprintf( '%s: Transfer functions\n', sysname );
    sys = zpk( psys )
    disp( 'Press any key to continue...' ); pause;
end


%---------------------------------------------------------------------------------------
%
%   Open-Loop Singular Value plots
%
%---------------------------------------------------------------------------------------
fprintf( '%s: Open Loop Singular Values Plot\n', sysname );
sigma( psys );
title( sprintf( '%s: Plant Singular Values', sysname) );


%---------------------------------------------------------------------------------------
%
%   H-infinity controller
%
%---------------------------------------------------------------------------------------
w = logspace(-6,5,400);

%
%   W1 - penalizes e
%
nuw11 = [0 1 1e-5]*3.5;     % u+theta
```

```
dnw11 = [0 1 1];

nuw12 = [0 1 1e-5]*3.5;     % v+phi
dnw12 = [0 1 1];

nuw13 = [0 1 1e-5]*1.5;     % w
dnw13 = [0 1 1];

nuw14 = [0 1 1e-3]*1.5;     % r
dnw14 = [0 1 3];

%
%   W2 - penalizes u
%
nuw21 = [10];
dnw21 = [1];

nuw22 = nuw21;
dnw22 = dnw21;

nuw23 = nuw21;
dnw23 = dnw21;

nuw24 = nuw21;
dnw24 = dnw21;

%
%   W3 - penalizes y
%
nuw31 = [1 1e3]*3;      % u+theta
dnw31 = [1 10*3]*1e3/10/3;

nuw32 = [1 1e3]*2.3;        % v+phi
dnw32 = [1 10*2.3]*1e3/10/2.3;

nuw33 = [1 1e3]*1.3;        % w
dnw33 = [1 10*1.3]*1e3/10/1.3;

nuw34 = [1 1e3]*1.5;        % r
dnw34 = [1 5*1.5]*1e3/5/1.5;

svw1i = bode(nuw11,dnw11,w);        svw1i = 20*log10(svw1i);
svw2i = bode(nuw21,dnw21,w);        svw2i = 20*log10(svw2i);
svw3i = bode(nuw31,dnw31,w);        svw3i = 20*log10(svw3i);

figure
semilogx( w,svw1i, w,svw2i, w,svw3i );
grid on
title( sprintf('%s: Design Specifications',sysname) );
xlabel( 'Frequency - Rad/Sec' );
ylabel( '1/W1 & 1/W3 - db' );


Gam = 1;                     %   Initial Gamma value
ss_g = mksys(ag,bg,cg,dg);   %   Pack system matrices into "TREE" variable.

%
%       Pick W1, W2, and W3 such as
%
%           [W1S ]
%           [W2KS]  =< 1
%           [W3T ]
%
%           where S = 1/(I + PK):Sensitivity fc.
%                   T = PK/(I + PK):Complementary Sensitivety fc.
%                   P:plant
%                   K:controller
%
w1 = [Gam*dnw11;nuw11 ; Gam*dnw12;nuw12 ; Gam*dnw13;nuw13 ; Gam*dnw14;nuw14];
w2 = [dnw21;nuw21 ; dnw22;nuw22 ; dnw23;nuw23 ; dnw24;nuw24 ];
w3 = [dnw31;nuw31 ; dnw32;nuw32 ; dnw33;nuw33 ; dnw34;nuw34];
[TSS_]=augtf(ss_g,w1,w2,w3);

[Gam_fact,SS_CP,SS_CL] = hinfopt(TSS_);         %   Find Optimal Gamma Value

Gam = Gam*Gam_fact;

%
```

```
%   W1,W2, and W3 with optimal gamma
%
w1 = [Gam*dnw11;nuw11 ; Gam*dnw12;nuw12 ; Gam*dnw13;nuw13 ; Gam*dnw14;nuw14];
w2 = [dnw21;nuw21 ; dnw22;nuw22 ; dnw23;nuw23 ; dnw24;nuw24 ];
w3 = [dnw31;nuw31 ; dnw32;nuw32 ; dnw33;nuw33 ; dnw34;nuw34];
[TSS_] = augtf( ss_g, w1, w2, w3 );

%
% Print W
%
fprintf( '%s: Gamma = Gam, W1=sysw1, W2=sysw2, W3=sysw3\n', sysname );
fprintf( '%s: Optimal Gamma\n', sysname );
Gam
fprintf( '%s: W1\n', sysname );
sysw1 = tf( [Gam*dnw11], [Gam*nuw11] )
fprintf( '%s: W2\n', sysname );
sysw2 = tf( [dnw21],[nuw21] )
fprintf( '%s: W3\n', sysname );
sysw2 = tf( [dnw31],[nuw31] )

fprintf( '********* %s: H infinity Controller Design  *************\n', sysname );

%
%   AUGTF Augmentation of W1,W2,W3
%   (transfer function form) into two-port plant.
%
[A,B1,B2,C1,C2,D11,D12,D21,D22] = augtf(ag,bg,cg,dg,w1,w2,w3);

%
%   H_infinity Design
%
[ss_cp,ss_cl,hinfo] = hinf(TSS_);
[acp,bcp,ccp,dcp] = branch(ss_cp);
[acp_r, bcp_r, ccp_r, dcp_r]=balmr(acp,bcp,ccp,dcp,2,0.1);
zpk(ss(acp,bcp,ccp,dcp));
[acl,bcl,ccl,dcl] = branch(ss_cl);

%
%   acp,bcp,ccp,dcp     : H infinity Controller
%   acp1,bcp1,ccp1,dcp1 : Reduction Model of H infinity Controller
%
fprintf( '%s: H-infinity Controller - hsys=ss(acp,bcp,ccp,dcp)\n', sysname );

hsys1 = ss(acp,bcp,ccp,dcp);

%
% Inverse bilinear transform
%
if bUseBilin
   [acp,bcp,ccp,dcp] = bilin( acp,bcp,ccp,dcp, -1, 'S_ftjw', [p2, p1] );
end

hsys = ss(acp,bcp,ccp,dcp);
hsys_r = ss(acp_r,bcp_r,ccp_r,dcp_r);

syshp = series( hsys, psys );                              % Open-loop system
sysht = feedback( syshp, eye(size(syshp))' );              % Comp. Sensitivity
[aht,bht,cht,dht]=ssdata(sysht);
%%sensysht = ss(aht,bht,-cht,eye(size(cht,1)));              % Sensitivity

%
%   Controller Pole-Zero Map
%
if( 0 )
    figure
    pzmap( hsys );
    title( 'H-infinity controller: Pole-Zero Map' );
end


%-----------------------------------------------------------------------------------------
%
%   Closed-Loop Singular Value Plots
%
%-----------------------------------------------------------------------------------------

disp('     ..... Computing Bode plots of Sens. & Comp. Sens. functions .....')

```

```
Ssys = feedback( ss(eye(size(hsys.b,2))), series(hsys, psys) );
rusys = feedback( hsys, psys );
clsys = sysht;
svs = sigma( Ssys, w ); svs = 20*log10(svs);
svt = sigma( clsys, w ); svt = 20*log10(svt);

figure
subplot( 2,1,1 );
semilogx(w,svw1i,w,svs)
title( 'Sensitivity and 1/W1' );
xlabel('Frequency - Rad/Sec')
ylabel('Gain - db')
grid on
subplot( 2,1,2 );
semilogx(w,svw3i,w,svt)
title( 'Comp. Sensitivity and 1/W3' );
xlabel('Frequency - Rad/Sec')
ylabel('Gain - db')
grid on


%-----------------------------------------------------------------------------------------
%
%   Closed Loop Command Following
%
%-----------------------------------------------------------------------------------------
fprintf( '%s: Step Responses\n', sysname );

%
% Set up parameters for SIMULINK
% and obtain final closed-loop system
%
Ap = psys.a;
Bp = psys.b;
Cp = psys.c;
Dp = psys.d;

Ak = hsys.a;
Bk = hsys.b;
Ck = hsys.c;
Dk = hsys.d;

W = eye(4);

cd augk
getclsys
cd ..

%
% Invert the nondimentionalizing scaling factors:
%
% (ft/sec) for linear velocities (10 knots)
% (rad/sec) for angular rates (20 deg)
% (rad) for angular displacements (20 deg)
%
%scale_fact = [16.67, 16.67, 16.67, 0.349, 0.349, 0.349, 0.349, 0.349];
%scaling = diag(scale_fact);

scale_fact = [1, 1, 1, ones(1,5)*pi/180];
scaling = diag(scale_fact);

scale_fact_deg = ones(1,8);
scaling_deg = diag(scale_fact_deg);

y_scaling = eye(4);

%
% Control scaling (deg)
%
u_scaling = eye(4);

%
% Compute the 'actual' c.l. system (with [u,v,w,r] as outputs)
%
selo = eye(8);
selo = selo([1 2 3 6],:);
rclsys = selo * clsys;

```

```
y4_scaling = diag( scale_fact_deg([1, 2, 3, 6]));

if bInvdc
    %
    % Augment clsys with its inverse dc gain (placed in pre-filter, W)
    %
    dcl = dcgain(rclsys);
    dcl = dcl(1:4,1:4);
    idd = inv(dcl);
    W = idd;

    clsys = clsys * idd;
    rusys = rusys * idd;
    rclsys = rclsys * W(1:4,1:4);
end

maxt = 10;
t = linspace(0, maxt, 800);
[y t] = step( clsys, t );               % Step response for each input

HRUsys = rusys;
[u t]= step( HRUsys, t );               % Step response for each control signal

outnames = { 'u (ft/sec)', 'v (ft/sec)', 'Climb Rate (ft/sec)', 'Yaw Rate (deg/sec)' };

os= [];

figure
for i = 1:4
   subplot( 2,2,i )
   plot( t, y(:,1,i), 'k-');
   hold on
   grid on
   plot( t, y(:,2,i), 'r--' );
   plot( t, y(:,3,i), 'm-.' );
   plot( t, y(:,6,i), 'b:' );
   plot( t, y(:,7,i), 'b--' );
   plot( t, y(:,8,i), 'g-' );
   title( sprintf( '%s step response', outnames{i} ) );
   ylabel( ' ' );
   xlabel('Time (sec) ')
   os= [os (max(y(:,i,i))-1)*100];
end
%print -deps2 clstepresp.eps

legend( 'u (ft/sec)', 'v (ft/sec)', 'w (ft/sec)', 'r (deg/sec)', 'phi (deg)', 'theta (deg)' );

disp( 'Overshoots: (%)' );
os

figure
for i=1:4
   subplot( 2,2,i )
   plot( t, u(:,1,i), 'k-');
   hold on
   grid on
   plot( t, u(:,2,i), 'r:' );
   plot( t, u(:,3,i), 'g-.' );
   plot( t, u(:,4,i), 'b--' );
   title( sprintf( '%s step response: controls', outnames{i} ) );
   ylabel( ' ' );
   xlabel( 'Time (sec)' )
end

legend('Long. cyclic (deg)', 'Lat. cyclic (deg)', 'Main collective (deg)', 'Tail collective (deg)' );

%print -deps2 clstepresp.eps

%
% Closed-loop pole-zero map
%
figure
pzmap( clsys );
grid on
title( 'Closed-loop pole-zero map' );

%
% Put system parameters in variables for SIMULINK (sl_garrard_good_hinf)
```

```
%
Ap = psys.a;
Bp = psys.b;
Cp = psys.c;
Dp = psys.d;

Ak = hsys.a;
Bk = hsys.b;
Ck = hsys.c;
Dk = hsys.d;

clp = eig(clsys);
zeta = zetafn(clp)
min_zeta = min(zeta)
Gam

figure
pzmap(rclsys);
title( 'Pole-Zero Map of closed-loop system' );
grid on
axis( [-7 1 -10 10] );

%% print -depsc garrard_hinf_mixa_pzclsys

figure
sigma( rclsys );
title( 'Singular Value plot of closed-loop system ([w,v,u,r])' );
grid on;

figure
grid on
sigma(hsys)
title( 'S.V. Plot of compensator' );

save mixa_dat Ak Bk Ck Dk Ap Bp Cp Dp W scaling y_scaling u_scaling
```

APPENDIX B

C_{++} Source Code

Beginning of RtcCart.h

```
// RtcCart.h : main header file for the RTCCART application
//

#if !defined(AFX_RTCCART_H__AC418F76_6865_11D3_A571_0060971997A1__INCLUDED_)
#define AFX_RTCCART_H__AC418F76_6865_11D3_A571_0060971997A1__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#ifndef __AFXWIN_H__
        #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"                // main symbols

/////////////////////////////////////////////////////////////////////////////
// CRtcCartApp:
// See RtcCart.cpp for the implementation of this class
//

class CRtcCartApp : public CWinApp
{
public:
        CRtcCartApp();

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CRtcCartApp)
        public:
        virtual BOOL InitInstance();
        //}}AFX_VIRTUAL

// Implementation

        //{{AFX_MSG(CRtcCartApp)
                // NOTE - the ClassWizard will add and remove member functions here.
                //    DO NOT EDIT what you see in these blocks of generated code !
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};


/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_RTCCART_H__AC418F76_6865_11D3_A571_0060971997A1__INCLUDED_)
```

Beginning of RtcCart.cpp

```
// RtcCart.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "RtcCart.h"
#include "RtcCartDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CRtcCartApp

BEGIN_MESSAGE_MAP(CRtcCartApp, CWinApp)
        //{{AFX_MSG_MAP(CRtcCartApp)
                // NOTE - the ClassWizard will add and remove mapping macros here.
                //    DO NOT EDIT what you see in these blocks of generated code!
        //}}AFX_MSG
        ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CRtcCartApp construction

CRtcCartApp::CRtcCartApp()
{
        // TODO: add construction code here,
        // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CRtcCartApp object

CRtcCartApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CRtcCartApp initialization

BOOL CRtcCartApp::InitInstance()
{
        AfxEnableControlContainer();

        // Standard initialization
        // If you are not using these features and wish to reduce the size
        //  of your final executable, you should remove from the following
        //  the specific initialization routines you do not need.

#ifdef _AFXDLL
        Enable3dControls();                     // Call this when using MFC in a shared DLL
#else
        Enable3dControlsStatic();       // Call this when linking to MFC statically
#endif

        CRtcCartDlg dlg;
        m_pMainWnd = &dlg;
        int nResponse = dlg.DoModal();
        if (nResponse == IDOK)
        {
                // TODO: Place code here to handle when the dialog is
                //  dismissed with OK
        }
        else if (nResponse == IDCANCEL)
        {
                // TODO: Place code here to handle when the dialog is
                //  dismissed with Cancel
        }

        // Since the dialog has been closed, return FALSE so that we exit the
        //  application, rather than start the application's message pump.
        return FALSE;
}
```

Beginning of RtcCartDlg.h

```
// RtcCartDlg.h : header file
//

#if !defined(AFX_RTCCARTDLG_H__AC418F78_6865_11D3_A571_0060971997A1__INCLUDED_)
#define AFX_RTCCARTDLG_H__AC418F78_6865_11D3_A571_0060971997A1__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#include "SignalList.h"

class CCommandsDlg;

/////////////////////////////////////////////////////////////////////////////
```

```
// CRtcCartDlg dialog

class CRtcCartDlg : public CDialog
{
protected:
        CSignalList m_signalList;
        CCommandsDlg *m_pCmdDlg;

// Construction
public:
        void GetReferenceCommands();
        void ControlStep();
        void UpdateView();
        void UpdateSignalData();
        void OnSpecSignalUpdate();
        CRtcCartDlg(CWnd* pParent = NULL);        // standard constructor
        ~CRtcCartDlg();


// Dialog Data
        //{{AFX_DATA(CRtcCartDlg)
        enum { IDD = IDD_RTCCART_DIALOG };
        CButton        m_btnSaveData;
        CButton        m_btnStop;
        CButton        m_rbUser;
        CButton        m_btnSignal;
        CSliderCtrl        m_ctrlSlider;
        CString        m_strRefCmd;
        CString        m_strTime;
        CString        m_strActual;
        CString        m_strControlV;
        CString        m_strK;
        CString        m_strA;
        BOOL        m_bUseW;
        CString        m_strKp;
        CString        m_strKi;
        CString        m_strKd;
        CString        m_strB;
        //}}AFX_DATA

        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CRtcCartDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);        // DDX/DDV support
        //}}AFX_VIRTUAL

// Implementation
protected:
        void GetPotV();
        void OnReset();
        BOOL m_bRunning;
        HICON m_hIcon;

        // Generated message map functions
        //{{AFX_MSG(CRtcCartDlg)
        virtual BOOL OnInitDialog();
        afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
        afx_msg void OnPaint();
        afx_msg HCURSOR OnQueryDragIcon();
        virtual void OnOK();
        afx_msg void OnRun();
        afx_msg void OnStop();
        afx_msg void OnUpdateSettings();
        afx_msg void OnSettingsChanged();
        afx_msg void OnDefault();
        afx_msg void OnRefSignal();
        afx_msg void OnRefUser();
        afx_msg void OnSpecSignal();
        afx_msg void OnSaveData();
public:
        afx_msg void OnTimer(UINT nIDEvent);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

```

```
#endif // !defined(AFX_RTCCARTDLG_H__AC418F78_6865_11D3_A571_0060971997A1__INCLUDED_)
```

Beginning of RtcCartDlg.cpp

```
// RtcCartDlg.cpp : implementation file
//

#include "stdafx.h"
#include "RtcCart.h"
#include "RtcCartDlg.h"
#include "ndqfun.h"
#include "CmdParam.h"
#include "CommandsDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//
// Program constants
//
const int CONTROL_RATE = 50;                        // Sampling (output) rate, Hertz
const int SCREEN_UPDATE_RATE = 10;                // Hertz
const int CONTROL_TIMER_ID = 1;                          // Timer ID for control step
const int SCREEN_UPDATE_TIMER_ID = 2;        // Timer ID for updating the screen
const double PI = 3.14159265;

const double VOLTS_PER_METER = 6.5617;        // Position pot. resolution (6" per V)

const int SLIDER_RESOLUTION = 20;                // Points per meter
const double MIN_SLIDER_RANGE = -0.4;        // Min command (meters)
const double MAX_SLIDER_RANGE = 0.4;        // Max command (meters)

//
// Signal Information
//
const int NUMSIGNALSTOSAVE = 5;

enum {SIGNAL_TIME = 0, SIGNAL_R, SIGNAL_RHAT, SIGNAL_U, SIGNAL_Y};

struct
{
        int id;
        char szName[CSignalList::MAXNAME];
}
Signal[NUMSIGNALSTOSAVE] =
{
        {SIGNAL_TIME, "t"},
        {SIGNAL_R, "r"},
        {SIGNAL_RHAT, "rhat"},
        {SIGNAL_U, "u"},
        {SIGNAL_Y, "y"}
};

//
// Fixed time-step (sampling rate)
//
const double dt = (double) 1 / CONTROL_RATE;

//
// Run time
//
double dTime = 0.0;

//
// Controller parameters
//
const double DEFAULT_K = 2.65;
const double DEFAULT_A = 16.83;
const double DEFAULT_B = 5.0;

```

```
double k = DEFAULT_K;
double a = DEFAULT_A;
double b = DEFAULT_B;

//
// Internal control-system signals
//
double r, rhat, e, u, y;

//
// Controller states
//
double rhat_last = 0;
double e_last = 0;
double int_e = 0;
double d_e = 0;
double k_p, k_i, k_d;

//
// Physical signals
//
double dPotV, dControlV;

//
// Program flags
//
BOOL bUsePrefilter = TRUE;

//
// Reference command settings
//
double dRefFreq = 1.0, dRefOffset = 0.0, dRefAmplitude = 1.0;

CCmdParam cmdRef;                                               // the command currently in use
CCmdParam cmdRefGenerated;                               // stores function generator settings

/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
        CAboutDlg();

// Dialog Data
        //{{AFX_DATA(CAboutDlg)
        enum { IDD = IDD_ABOUTBOX };
        //}}AFX_DATA

        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CAboutDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
        //}}AFX_VIRTUAL

// Implementation
protected:
        //{{AFX_MSG(CAboutDlg)
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
        //{{AFX_DATA_INIT(CAboutDlg)
        //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CAboutDlg)
        //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
        //{{AFX_MSG_MAP(CAboutDlg)
                // No message handlers
```

```
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CRtcCartDlg dialog

CRtcCartDlg::CRtcCartDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CRtcCartDlg::IDD, pParent)
	, m_signalList(NUMSIGNALSTOSAVE)
{
	//{{AFX_DATA_INIT(CRtcCartDlg)
	m_strRefCmd = _T("");
	m_strTime = _T("");
	m_strActual = _T("");
	m_strControlV = _T("");
	m_strK = _T("");
	m_strA = _T("");
	m_bUseW = FALSE;
	m_strKp = _T("");
	m_strKi = _T("");
	m_strKd = _T("");
	m_strB = _T("");
	//}}AFX_DATA_INIT
	// Note that LoadIcon does not require a subsequent DestroyIcon in Win32
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);

	m_pCmdDlg = NULL;
	dTime = 0;
	m_bRunning = FALSE;
}

CRtcCartDlg::~CRtcCartDlg()
{
	ResetDevice();
}

void CRtcCartDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CRtcCartDlg)
	DDX_Control(pDX, ID_SAVE_DATA, m_btnSaveData);
	DDX_Control(pDX, ID_STOP, m_btnStop);
	DDX_Control(pDX, IDC_REFUSER, m_rbUser);
	DDX_Control(pDX, ID_SPEC_SIGNAL, m_btnSignal);
	DDX_Control(pDX, IDC_SLIDER_CMD, m_ctrlSlider);
	DDX_Text(pDX, IDC_REF, m_strRefCmd);
	DDX_Text(pDX, IDC_TIME, m_strTime);
	DDX_Text(pDX, IDC_POTVIN, m_strActual);
	DDX_Text(pDX, IDC_CONTROL, m_strControlV);
	DDX_Text(pDX, IDC_K, m_strK);
	DDX_Text(pDX, IDC_A, m_strA);
	DDX_Check(pDX, IDC_USEW, m_bUseW);
	DDX_Text(pDX, IDC_E, m_strKp);
	DDX_Text(pDX, IDC_INT_E, m_strKi);
	DDX_Text(pDX, IDC_D_E, m_strKd);
	DDX_Text(pDX, IDC_B, m_strB);
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CRtcCartDlg, CDialog)
	//{{AFX_MSG_MAP(CRtcCartDlg)
	ON_WM_SYSCOMMAND()
	ON_WM_PAINT()
	ON_WM_QUERYDRAGICON()
	ON_BN_CLICKED(ID_RUN, OnRun)
	ON_BN_CLICKED(ID_STOP, OnStop)
	ON_WM_TIMER()
	ON_BN_CLICKED(ID_UPDATE, OnUpdateSettings)
	ON_EN_CHANGE(IDC_K, OnSettingsChanged)
	ON_BN_CLICKED(ID_DEFAULT, OnDefault)
	ON_BN_CLICKED(IDC_REFSIGNAL, OnRefSignal)
	ON_BN_CLICKED(IDC_REFUSER, OnRefUser)
	ON_BN_CLICKED(ID_SPEC_SIGNAL, OnSpecSignal)
	ON_BN_CLICKED(ID_SAVE_DATA, OnSaveData)
	ON_EN_CHANGE(IDC_A, OnSettingsChanged)
	ON_BN_CLICKED(IDC_USEW, OnSettingsChanged)
	ON_EN_CHANGE(IDC_B, OnSettingsChanged)
	//}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CRtcCartDlg message handlers

BOOL CRtcCartDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        // Add "About..." menu item to system menu.

        // IDM_ABOUTBOX must be in the system command range.
        ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
        ASSERT(IDM_ABOUTBOX < 0xF000);

        CMenu* pSysMenu = GetSystemMenu(FALSE);
        if (pSysMenu != NULL)
        {
                CString strAboutMenu;
                strAboutMenu.LoadString(IDS_ABOUTBOX);
                if (!strAboutMenu.IsEmpty())
                {
                        pSysMenu->AppendMenu(MF_SEPARATOR);
                        pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
                }
        }

        // Set the icon for this dialog.  The framework does this automatically
        //  when the application's main window is not a dialog
        SetIcon(m_hIcon, TRUE);                        // Set big icon
        SetIcon(m_hIcon, FALSE);               // Set small icon

        // TODO: Add extra initialization here
        m_rbUser.SetCheck( TRUE );

        m_ctrlSlider.SetRange(
                (int)(MIN_SLIDER_RANGE*SLIDER_RESOLUTION),
                (int)(MAX_SLIDER_RANGE*SLIDER_RESOLUTION),
                TRUE );

        cmdRef.Set( USER, 1.0, 1.0, 0.0 );
        cmdRefGenerated.Set( SINE, 20, 0.2, 0.0 );

        //
        // Initialize user-changeable parameters
        //
        m_strK.Format( "%2.2f", k );
        m_strA.Format( "%2.2f", a );
        m_strB.Format( "%2.2f", b );
        m_bUseW = bUsePrefilter;

        for ( int i = 0; i < NUMSIGNALSTOSAVE; i++ )
        {
                m_signalList.SetName( i, Signal[i].szName );
        }

        //
        // Make sure the NiDAQ board is reset
        //
        OnStop();

        //
        // Keep the update-screen timer running
        // - So we'll have a running position display
        //
        SetTimer( SCREEN_UPDATE_TIMER_ID, 1000/SCREEN_UPDATE_RATE, NULL);

        return TRUE;  // return TRUE  unless you set the focus to a control
}

void CRtcCartDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
        if ((nID & 0xFFF0) == IDM_ABOUTBOX)
        {
                CAboutDlg dlgAbout;
                dlgAbout.DoModal();
        }
        else
```

```
	{
		CDialog::OnSysCommand(nID, lParam);
	}
}

// If you add a minimize button to your dialog, you will need the code below
//  to draw the icon.  For MFC applications using the document/view model,
//  this is automatically done for you by the framework.

void CRtcCartDlg::OnPaint()
{
	if (IsIconic())
	{
		CPaintDC dc(this); // device context for painting

		SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

		// Center icon in client rectangle
		int cxIcon = GetSystemMetrics(SM_CXICON);
		int cyIcon = GetSystemMetrics(SM_CYICON);
		CRect rect;
		GetClientRect(&rect);
		int x = (rect.Width() - cxIcon + 1) / 2;
		int y = (rect.Height() - cyIcon + 1) / 2;

		// Draw the icon
		dc.DrawIcon(x, y, m_hIcon);
	}
	else
	{
		CDialog::OnPaint();
	}
}

// The system calls this to obtain the cursor to display while the user drags
//  the minimized window.
HCURSOR CRtcCartDlg::OnQueryDragIcon()
{
	return (HCURSOR) m_hIcon;
}

void CRtcCartDlg::OnOK()
{
	OnStop();

	CDialog::OnOK();
}

void CRtcCartDlg::OnRun()
{
	if ( !m_bRunning )
	{
		m_btnSaveData.EnableWindow( TRUE );
		m_btnStop.EnableWindow( TRUE );
		m_btnStop.SetWindowText( "&Stop" );

		SetTimer( CONTROL_TIMER_ID, 1000/CONTROL_RATE, NULL);

		m_bRunning = TRUE;
	}

	UpdateView();
}

void CRtcCartDlg::OnStop()
{
	if ( m_bRunning )
	{
		KillTimer( CONTROL_TIMER_ID );

		m_bRunning = FALSE;
		m_btnStop.SetWindowText( "&Reset" );
	}
	else
	{
		//
		// User hit Stop button twice in a row
		//
```

```
                OnReset();
                m_btnStop.EnableWindow( FALSE );
                m_btnSaveData.EnableWindow( FALSE );
        }

        ResetDevice();
        UpdateView();
}

void CRtcCartDlg::UpdateView()
{
        m_strTime.Format( "%2.1f", dTime );
        m_strActual.Format( "%2.2f", dPotV * 100 / VOLTS_PER_METER );
        m_strControlV.Format( "%2.2f", dControlV );
        m_strRefCmd.Format( "%2.2f", rhat * 100 );

        m_strKp.Format( "%2.2f", k_p );
        m_strKi.Format( "%2.2f", k_i );
        m_strKd.Format( "%2.2f", k_d );

        UpdateData( FALSE );                        // Update interface
}

void CRtcCartDlg::UpdateSignalData()
{
        m_signalList.Add( SIGNAL_TIME, (float) dTime );                                // sec
        m_signalList.Add( SIGNAL_R, (float) r * 100 );                                // cm
        m_signalList.Add( SIGNAL_RHAT, (float) rhat * 100 );                // cm
        m_signalList.Add( SIGNAL_U, (float) dControlV );                        // volts
        m_signalList.Add( SIGNAL_Y, (float) (dPotV * 100 / VOLTS_PER_METER) );        // cm
}

void CRtcCartDlg::ControlStep()
{
        const int control_mode = 2;

        GetReferenceCommands();

        GetPotV();

        y = dPotV / VOLTS_PER_METER;                                // from volts to meters

        if (control_mode == 1)
        {
                if (bUsePrefilter)
                {
                        rhat = (a*dt / (1+a*dt)) * r + (1 / (1+a*dt)) * rhat_last;
                }
                else
                {
                        rhat = r;
                }

                e = rhat - y;
                u = k*(e-e_last)/dt + k*a*e;

                rhat_last = rhat;
                e_last = e;

                //
                // Set Control voltage
                //
                dControlV = -u;
        }
        else
        {
                if (bUsePrefilter)
                {
                        rhat = (b*dt / (1+b*dt)) * r + (1 / (1+b*dt)) * rhat_last;
                }
                else
                {
                        rhat = r;
                }

                e = (rhat - y);
                int_e = int_e + e * dt;                                        // approximation of integral
                d_e = (e-e_last)/dt;                                        // approximation of derivative

                k_p = k*(a+b)*e;                                        // proportional term
                k_i = k*a*b*int_e;                                        // integral term
                k_d = k*d_e;                                                // derivative term

                u = k_p + k_i + k_d;                                    // PID controller

                rhat_last = rhat;
                e_last = e;

                //
                // Set Control voltage
                //
                dControlV = -u;
        }

    //
        // update time
        //
        dTime += dt;

        //
        // Output control voltage
        //
        SetOutV( iOChan0, dControlV );
}

void CRtcCartDlg::OnTimer(UINT nIDEvent)
{
        if ( CONTROL_TIMER_ID == nIDEvent )
        {
                ControlStep();
                UpdateSignalData();
        }
        else if ( SCREEN_UPDATE_TIMER_ID == nIDEvent )
        {
                if (!m_bRunning)
                {
                        //
                        // Update the position display
                        //
                        GetPotV();
                }

                UpdateView();
        }

        CDialog::OnTimer(nIDEvent);
}

void CRtcCartDlg::OnUpdateSettings()
{
        k = atof( m_strK );
        a = atof( m_strA );
        b = atof( m_strB );
        bUsePrefilter = m_bUseW;
}

void CRtcCartDlg::OnSettingsChanged()
{
        //
        // Read data freom dialog box to member variables
        //
        UpdateData( TRUE );
}

void CRtcCartDlg::OnDefault()
{
        k = DEFAULT_K;
        a = DEFAULT_A;
        b = DEFAULT_B;

        m_strK.Format( "%2.2f", k );
        m_strA.Format( "%2.2f", a );
        m_strB.Format( "%2.2f", b );

        UpdateView();
}
```

```


void CRtcCartDlg::GetReferenceCommands()
{
        switch (cmdRef.mode)
        {
        case USER:
                //
                // Get Reference command from slider bar
                //
                r = (double) m_ctrlSlider.GetPos() / SLIDER_RESOLUTION;
                break;

        case CONSTANT:
                r = dRefAmplitude;
                break;

        case SINE:
                r = dRefAmplitude * sin(2*PI*dRefFreq*dTime) + dRefOffset;
                break;

        case SQUARE:
                double dPeriod = 1/dRefFreq;
                double x = fmod(dTime, dPeriod);
                if (x > dPeriod/2)
                {
                        r = dRefOffset + dRefAmplitude;
                }
                else
                {
                        r = dRefOffset - dRefAmplitude;
                }
                break;
        }
}

void CRtcCartDlg::OnRefSignal()
{
        m_ctrlSlider.EnableWindow( FALSE );
        m_btnSignal.EnableWindow( TRUE );
}

void CRtcCartDlg::OnRefUser()
{
        m_ctrlSlider.EnableWindow( TRUE );
        m_btnSignal.EnableWindow( FALSE );

        cmdRef.mode = USER;
}

void CRtcCartDlg::OnSpecSignal()
{
        if (!m_pCmdDlg)
        {
                m_pCmdDlg = new CCommandsDlg( this, &m_pCmdDlg );
                m_pCmdDlg->cmdDX = cmdRefGenerated;

                //
                // Create modeless dialog
                //
                m_pCmdDlg->Create( IDD_COMMANDS, this );
        }
}

void CRtcCartDlg::OnSpecSignalUpdate()
{
        cmdRef = m_pCmdDlg->cmdDX;
        cmdRefGenerated = m_pCmdDlg->cmdDX;

        dRefAmplitude = cmdRef.a / 100;
        dRefFreq = cmdRef.b;
        dRefOffset = cmdRef.c / 100;
}

void CRtcCartDlg::OnSaveData()
{
        //
        // Shows "save-file" dialog box, and saves data
        //
        const char szDefaultFileName[] = "RtcCart";
        OPENFILENAME ofn;
        char szDirName[256]= ".";
        char szFile[256], szFileTitle[256];
        char chReplace;                                                                     // string separator for szFilter
        char szFilter[256]= "MATLAB Files (*.M)|*.m|";
        int i;
        HWND hWnd = AfxGetMainWnd()->GetSafeHwnd();

        //
        // Set up filename request common dialog box
        //
        strcpy( szFile, szDefaultFileName );
        chReplace = szFilter[strlen(szFilter) - 1];                // retrieve wildcard

        for (i = 0; szFilter[i] != '\0'; i++)
        {
                if (szFilter[i] == chReplace)
                {
                        szFilter[i] = '\0';
                }
        }

        //
        // Set all structure members to zero
        //
        memset( &ofn, 0, sizeof(OPENFILENAME) );

        //
        // Initialize the OPENFILENAME members
        //
        ofn.lStructSize = sizeof(OPENFILENAME);
        ofn.hwndOwner = hWnd;
        ofn.lpstrFilter = szFilter;
        ofn.lpstrFile= szFile;
        ofn.nMaxFile = sizeof(szFile);
        ofn.lpstrFileTitle = szFileTitle;
        ofn.nMaxFileTitle = sizeof(szFileTitle);
        ofn.lpstrDefExt= "m";

        ofn.lpstrInitialDir = szDirName;
        ofn.Flags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT;

        if ( GetSaveFileName(&ofn) )
        {
                m_signalList.SaveToMFile( szFile );
        }
}

void CRtcCartDlg::OnReset()
{
        //
        // Reset signals
        //
        dTime = 0.0;
        r = 0.0;
        rhat = 0.0;
        e = 0.0;
        u = 0.0;
        y = 0.0;
        dPotV = 0.0;
        dControlV = 0.0;

        //
        // Reset controller states
        //
        rhat_last = 0;
        e_last = 0;
        int_e = 0;
        d_e = 0;

        k_p = 0;
        k_i = 0;
        k_d = 0;

        m_signalList.Reset();
}
```

```

void CRtcCartDlg::GetPotV()
{
        //
        // Read in Potentiometer voltage
        // - This gives the actual position information
        //
        dPotV = 0 + GetInV( iIChan0 );
}
```

Beginning of ndqfun.h

```
#ifndef __NDQFUN_H
#define __NDQFUN_H

#include <nidaqex.h>

int ResetDevice();
int SetDiffOutV( f64 dVoltage );
int SetOutV( int iOChan, f64 dVoltage );
f64 GetInV( int iIChan );

extern const i16 iOChan0;                                               // Output Channel numbers
extern const i16 iOChan1;
extern const i16 iIChan0;                                               // Input channel numbers
extern const i16 iIChan1;
extern f64 dChan0V;                                               // Output Channel voltages
extern f64 dChan1V;

#endif //__NDQFUN_H
```

Beginning of ndqfun.cpp

```
#include "stdafx.h"
#include "ndqfun.h"

#pragma comment( lib, "nidaq32.lib" )
#pragma comment( lib, "nidex32.lib" )

//========================================================================
//        Global Declarations
//========================================================================
static i16 iStatus = 0;
static i16 iRetVal = 0;
static const i16 iDevice = 1;
static i16 iGain = 1;
const i16 iOChan0 = 0;                                        // Output Channel numbers
const i16 iOChan1 = 1;
const i16 iIChan1 = 2;                                        // Input channel number
const i16 iIChan0 = 0;                                        // Input channel number
f64 dChan0V= 0.0;
f64 dChan1V= 0.0;
static const i16 iIgnoreWarning = 0;

//------------------------------------------------------------------------
//        Initializes NiDAQ Board
//------------------------------------------------------------------------
int ResetDevice()
{
        // Initialize channel voltages to ground
        dChan0V= 0.0;
        dChan1V= 0.0;

        // Configure Device Output Channels
        iStatus = AO_Configure(iDevice, iOChan0, 0, 0, 10, 1 );
    iRetVal = NIDAQErrorHandler(iStatus, "AO_Configure", iIgnoreWarning);
        iStatus = AO_Configure(iDevice, iOChan1, 0, 0, 10, 1 );
    iRetVal = NIDAQErrorHandler(iStatus, "AO_Configure", iIgnoreWarning);

        // Set Output Channel 0 voltage
        iStatus = AO_VWrite( iDevice, iOChan0, dChan0V );
    iRetVal = NIDAQErrorHandler(iStatus, "AO_VWrite", iIgnoreWarning);
        // Set Output Channel 1 voltage
        iStatus = AO_VWrite( iDevice, iOChan1, dChan1V );
    iRetVal = NIDAQErrorHandler(iStatus, "AO_VWrite", iIgnoreWarning);

        // Update channels simulataneously
    iStatus = AO_Update( iDevice );
    iRetVal = NIDAQErrorHandler( iStatus, "AO_Update", iIgnoreWarning );

        return( 1 );
}

//-------------------------------------------------------------------------
//        Sets up a differential voltage between channels 0 (+) and 1 (-)
//-------------------------------------------------------------------------
int SetDiffOutV( f64 dVoltage )
{
        const double dOffsetV= 4.99;

        // Limit voltage range
        if( dVoltage > 10 )
                dVoltage= 10;
        else if( dVoltage <-10 )
                dVoltage= -10;

        // Calculate new voltage settings
        if( dVoltage >= 0 )
        {
                dChan1V= - dOffsetV;
                dChan0V= dVoltage - dOffsetV;
        }
        else
        {
                dChan1V= - dOffsetV - dVoltage;
                dChan0V= - dOffsetV;
        }

        // Limit output voltages
        if( dChan0V > 4.99 )
                dChan0V= 4.99;
        else if( dChan0V < -4.99 )
                dChan0V= -4.99;
        if( dChan1V > 4.99 )
                dChan1V= 4.99;
        else if( dChan1V < -4.99 )
                dChan1V= -4.99;

        // Set Output Channel 0 voltage
        iStatus = AO_VWrite( iDevice, iOChan0, dChan0V );
    iRetVal = NIDAQErrorHandler(iStatus, "AO_VWrite", iIgnoreWarning);
        // Set Output Channel 1 voltage
        iStatus = AO_VWrite( iDevice, iOChan1, dChan1V );
    iRetVal = NIDAQErrorHandler(iStatus, "AO_VWrite", iIgnoreWarning);

        // Update channels simulataneously
    iStatus = AO_Update( iDevice );
    iRetVal = NIDAQErrorHandler( iStatus, "AO_Update", iIgnoreWarning );

        return( 1 );
}

//-------------------------------------------------------------------------
//        Outputs voltage on specified channel
//-------------------------------------------------------------------------
int SetOutV( int iOChan, f64 dVoltage )
{
        if( iOChan == iOChan0 )
                dChan0V= dVoltage;
        else if( iOChan == iOChan1 )
                dChan1V= dVoltage;

        // Limit output voltage
        if( dVoltage > 4.99 )
                dVoltage= 4.99;
        else if( dVoltage< -4.99 )
                dVoltage= -4.99;

        // Set Output Channel voltage
        iStatus = AO_VWrite( iDevice, iOChan, dVoltage );
    iRetVal = NIDAQErrorHandler(iStatus, "AO_VWrite", iIgnoreWarning);

        // Update channels
    iStatus = AO_Update( iDevice );
    iRetVal = NIDAQErrorHandler( iStatus, "AO_Update", iIgnoreWarning );

        return( 1 );
}

//---------------------------------------------------------------------------
//        Reads input voltage form specified channel
//---------------------------------------------------------------------------
f64 GetInV( int iChan )
{
        f64 dInputV;

        // Read Input Channel
        iStatus = AI_VRead( iDevice, iIChan, iGain, &dInputV );
    iRetVal = NIDAQErrorHandler( iStatus, "AI_VRead", iIgnoreWarning );

        return( dInputV );
}

```

Beginning of strnum.h

```
#ifndef __STRNUM_HPP
#define __STRNUM_HPP

// StrNum.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CStrNum command target

class CStrNum
{
public:
        CStrNum( char *nStr= "0" );
        ~CStrNum();

// Attributes
public:
        double n;
        char str[20];

// Operations
public:
        double Set( char *nStr );
        double Get( char *nStr= NULL );
        double operator=( char *nStr );
        double operator=( CStrNum &b );

        operator double()
                { return( n ); }
        operator char*()
                { return( str ); }
};

/////////////////////////////////////////////////////////////////////////////

#endif // __STRNUM_HPP
```

Beginning of strnum.cpp

```
// StrNum.cpp : implementation file
//

#include "stdafx.h"
#include "StrNum.h"
#include <stdio.h>
#include <string.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CStrNum

CStrNum::CStrNum( char *nStr )
{
        Set( nStr );
}

CStrNum::~CStrNum()
{
}

double CStrNum::Set( char *nStr )
{
        strcpy( str, nStr );
        n= atof( str );
        return( n );
}

double CStrNum::Get( char *nStr )
{
        if( nStr!= NULL )
                strcpy( nStr, str );

        return( n );
}

double CStrNum::operator=( char *nStr )
{
        return( Set( nStr ) );
}

double CStrNum::operator=( CStrNum &b )
{
        return( Set( b.str ) );
}

```

Beginning of cmdparam.h

```
#ifndef __CMDPARAM_H
#define __CMDPARAM_H

typedef enum {CONSTANT=0, SINE, SQUARE, SAWTOOTH, USER, RAMP } INPUTMODES;

//---------------------------------------------------------------------------
//        class CmdParam
//---------------------------------------------------------------------------
class CCmdParam
{
public:
        double a, b, c;
        INPUTMODES mode;

public:
        CCmdParam();
        void Set( INPUTMODES n_mode, double n_a, double n_b, double n_c );
        void Get( INPUTMODES &n_mode, double *n_a, double *n_b, double *n_c );
        CCmdParam& operator= ( CCmdParam& cmdParam );
};

#endif // __CMDPARAM_H
```

Beginning of cmdparam.cpp

```
#include "stdafx.h"
#include "CmdParam.h"

/////////////////////////////////////////////////////////////////////////////
// CCmdParam

CCmdParam::CCmdParam()
{
        mode= CONSTANT;
        a= b= c= 0.0;
}

void CCmdParam::Set( INPUTMODES n_mode, double n_a, double n_b, double n_c )
{
        mode= n_mode;
        a= n_a;
        b= n_b;
        c= n_c;
}

void CCmdParam::Get( INPUTMODES &n_mode, double *n_a, double *n_b, double *n_c )
{
        n_mode= mode;
        *n_a= a;
        *n_b= b;
        *n_c= c;
}

CCmdParam& CCmdParam::operator= ( CCmdParam& cmdParam )
{
        mode= cmdParam.mode;
        a= cmdParam.a;
        b= cmdParam.b;
        c= cmdParam.c;

        return( *this );
}

```

Beginning of CommandsDlg.h

```
// CommandsDlg.h : header file
//

#include "resource.h"
#include "cmdparam.h"

/////////////////////////////////////////////////////////////////////////////
// CCommandsDlg dialog

class CCommandsDlg : public CDialog
{
// Construction
public:
        CCommandsDlg(CWnd* pParent = NULL, CCommandsDlg **ppRef = NULL);   // standard constructor
        ~CCommandsDlg();

public:
        CCmdParam cmdDX;

// Dialog Data
	//{{AFX_DATA(CCommandsDlg)
	enum { IDD = IDD_COMMANDS };
		// NOTE: the ClassWizard will add data members here
	//}}AFX_DATA


// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CCommandsDlg)
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
	//}}AFX_VIRTUAL

// Implementation
protected:
	CWnd *m_pParent;
	CCommandsDlg **m_ppRef;

	// Generated message map functions
	//{{AFX_MSG(CCommandsDlg)
	virtual BOOL OnInitDialog();
	virtual void OnOK();
	afx_msg void OnRadio(UINT nID);
	virtual void OnCancel();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

Beginning of CommandsDlg.cpp

```
// CommandsDlg.cpp : implementation file
//

#include "stdafx.h"
#include "StrNum.h"
#include "CommandsDlg.h"
#include "RtcCartDlg.h"
#include <math.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

char* ftostr( double num );
double ToOneSigFig( double nd );

/////////////////////////////////////////////////////////////////////////////
// CCommandsDlg dialog


CCommandsDlg::CCommandsDlg(CWnd* pParent /*=NULL*/, CCommandsDlg **ppRef)
	: CDialog(CCommandsDlg::IDD, pParent)
{
	//{{AFX_DATA_INIT(CCommandsDlg)
		// NOTE: the ClassWizard will add member initialization here
	//}}AFX_DATA_INIT

	m_pParent = pParent;
	m_ppRef = ppRef;

	cmdDX.Set( CONSTANT, 0, 0, 0 );
}

CCommandsDlg::~CCommandsDlg()
{
	//
	// Clear our reference pointer
	//
	if (m_ppRef)
```

```
        {
                *m_ppRef = NULL;
                m_ppRef = NULL;
        }
}


void CCommandsDlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CCommandsDlg)
                // NOTE: the ClassWizard will add DDX and DDV calls here
        //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CCommandsDlg, CDialog)
        //{{AFX_MSG_MAP(CCommandsDlg)
        ON_COMMAND_RANGE(IDC_RB21, IDC_RB23, OnRadio)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CCommandsDlg message handlers

BOOL CCommandsDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        CheckDlgButton( IDC_RB21+(int)cmdDX.mode, TRUE );

        switch( cmdDX.mode )
        {
                case CONSTANT:
                        SetDlgItemText( IDC_TEXT2A, "Constant:" );
                        SetDlgItemText( IDC_TEXT2B, "" );
                        SetDlgItemText( IDC_TEXT2C, "" );
                        break;
                case USER:
                        SetDlgItemText( IDC_TEXT2A, "Maximum" );
                        SetDlgItemText( IDC_TEXT2B, "Minimum" );
                        SetDlgItemText( IDC_TEXT2C, "" );
                        break;
                default:
                        SetDlgItemText( IDC_TEXT2A, "Amplitude:" );
                        SetDlgItemText( IDC_TEXT2B, "Frequency:" );
                        SetDlgItemText( IDC_TEXT2C, "Offset:" );
                        break;
        }

        SetDlgItemText( IDC_PARAM2A, ftostr( cmdDX.a ));
        SetDlgItemText( IDC_PARAM2B, ftostr( cmdDX.b ));
        SetDlgItemText( IDC_PARAM2C, ftostr( cmdDX.c ));


        return TRUE;  // return TRUE unless you set the focus to a control
                      // EXCEPTION: OCX Property Pages should return FALSE
}

//
// This is the Apply button handler
//
void CCommandsDlg::OnOK()
{
        char str[20];

        GetDlgItemText( IDC_PARAM2A, str, 15 );
        cmdDX.a = atof(  str );
        GetDlgItemText( IDC_PARAM2B, str, 15 );
        cmdDX.b = atof(  str );
        GetDlgItemText( IDC_PARAM2C, str, 15 );
        cmdDX.c = atof(  str );

        CRtcCartDlg *pRtcCartDlg = (CRtcCartDlg*)m_pParent;
        pRtcCartDlg->OnSpecSignalUpdate();
}

void CCommandsDlg::OnRadio( UINT nID )
```

```
{
        static CStrNum va, vf, vo;
        static CStrNum ha, hf, ho;
        static CStrNum vmax="0", vmin="0";
        static CStrNum hmax="0", hmin="0";
        char str[20];

        if( cmdDX.mode == USER )
        {
                GetDlgItemText( IDC_PARAM2A, str, 20 );
                hmax.Set( str );
                GetDlgItemText( IDC_PARAM2B, str, 20 );
                hmin.Set( str );
                if( nID >= IDC_RB21 && nID <= IDC_RB23 )
                {
                        SetDlgItemText( IDC_PARAM2A, (char*)ha );
                        SetDlgItemText( IDC_PARAM2B, (char*)hf );
                        SetDlgItemText( IDC_PARAM2C, (char*)ho );
                }
        }
        else
        {
                GetDlgItemText( IDC_PARAM2A, str, 20 );
                ha.Set( str );
                GetDlgItemText( IDC_PARAM2B, str, 20 );
                hf.Set( str );
                GetDlgItemText( IDC_PARAM2C, str, 20 );
                ho.Set( str );
        }

        switch( nID )
        {
                case IDC_RB21:
                        cmdDX.mode= CONSTANT;
                        CheckRadioButton( IDC_RB21, IDC_RB23, nID );
                        SetDlgItemText( IDC_TEXT2A, "Constant:" );
                        SetDlgItemText( IDC_TEXT2B, "" );
                        SetDlgItemText( IDC_TEXT2C, "" );
                        break;
                case IDC_RB22:
                        cmdDX.mode= SINE;
                        CheckRadioButton( IDC_RB21, IDC_RB23, nID );
                        SetDlgItemText( IDC_TEXT2A, "Amplitude:" );
                        SetDlgItemText( IDC_TEXT2B, "Frequency:" );
                        SetDlgItemText( IDC_TEXT2C, "Offset:" );
                        break;
                case IDC_RB23:
                        cmdDX.mode= SQUARE;
                        CheckRadioButton( IDC_RB21, IDC_RB23, nID );
                        SetDlgItemText( IDC_TEXT2A, "Amplitude:" );
                        SetDlgItemText( IDC_TEXT2B, "Frequency:" );
                        SetDlgItemText( IDC_TEXT2C, "Offset:" );
                        break;
        }
}

char* ftostr( double num )
{
        static char buf[100];

        sprintf( buf, "%0.2f", ToOneSigFig(num) );

        return( buf );
}

double ToOneSigFig( double nd )
{
        double d, factor= 1.0;

        d = (double)fabs( nd );

        // SCALE d TO RANGE 1 TO 10
        if ( d<1.0 && d>0.0 )
        {
                while ( d<1.0 )
                {
                        d *= 10;
                        factor *= 10;
                }
        }
        else if ( d>10.0 )
        {
                while ( d>10.0 )
                {
                        d /= 10;
                        factor /= 10;
                }
        }

        // ROUND D TO 1 SIG FIG AND SCALE BACK
        d = (double)floor( d );
        d = d/factor;

        // SET CORRECT SIGN
        if ( nd<0 )
        {
                d = -d;
        }

        return( d );
}


//
// This is the Close button handler
//
void CCommandsDlg::OnCancel()
{
        //
        // Clear our reference pointer
        //
        if (m_ppRef)
        {
                *m_ppRef = NULL;
                m_ppRef = NULL;
        }

        DestroyWindow();
}
```

Beginning of SignalList.h

```
#ifndef  __SIGNALLIST_H
#define __SIGNALLIST_H

class CSignalList
{
public:
        enum {MAXSIG = 10, MAXSIZE = 2000, MAXNAME = 20};

protected:
        float *m_pSignalArray[MAXSIG];
        int m_iSignalPos[MAXSIG];
        int m_nSignals;
        char *m_pszName[MAXSIG];

public:
        CSignalList( int nSignals );
        ~CSignalList();
        void SetName( int nSignal, const char *szName );
        void Add( int nSignal, float data );
        void Reset();
        BOOL SaveToMFile( char *szFilename );
};

#endif //__SIGNALLIST_H
```

Beginning of SignalList.cpp

```
#include "stdafx.h"
#include "SignalList.h"

CSignalList::CSignalList( int nSignals )
{
        if (nSignals > MAXSIG)
        {
                nSignals = MAXSIG;
        }

        m_nSignals = nSignals;

        for (int i = 0; i < nSignals; i++ )
        {
                m_pSignalArray[i] = new float[MAXSIZE];
                ASSERT( m_pSignalArray[i] );

                m_pszName[i] = new char[MAXNAME];
                ASSERT( m_pszName[i] );

                m_pszName[i][0] = 0;

                m_iSignalPos[i] = 0;
        }
}

CSignalList::~CSignalList()
{
        for (int i = 0; i < m_nSignals; i++ )
        {
                delete m_pSignalArray[i];
                delete m_pszName[i];
        }

        m_nSignals = 0;
}

void CSignalList::SetName( int nSignal, const char *szName )
{
        ASSERT( nSignal < m_nSignals );

        if ( nSignal < m_nSignals )
        {
                strncpy( m_pszName[nSignal], szName, MAXNAME );
        }
}

void CSignalList::Add( int nSignal, float data )
{
        ASSERT( nSignal < m_nSignals );

        if ( nSignal < m_nSignals && m_iSignalPos[nSignal] < MAXSIZE )
        {
                m_pSignalArray[nSignal][m_iSignalPos[nSignal]] = data;
                m_iSignalPos[nSignal]++;
        }
}

void CSignalList::Reset()
{
        for ( int i = 0; i < m_nSignals; i++ )
        {
                m_iSignalPos[i] = 0;
        }
}

BOOL CSignalList::SaveToMFile( char *szFilename )
{
        FILE *fp = fopen( szFilename, "wt" );
        if ( fp )
        {
                fprintf( fp, "%%\n%% Data from RtcCart\n%%\n" );

                for ( int i = 0; i < m_nSignals; i++ )
                {
                        fprintf( fp, "\n%s = [", m_pszName[i] );
                        for ( int j = 0; j < m_iSignalPos[i]; j++ )
                        {
                                fprintf( fp, "%f;\n", m_pSignalArray[i][j] );
                        }
                        fprintf( fp, "];\n" );
                }

                fclose( fp );

                return TRUE;
        }
        else
        {
                return FALSE;
        }
}

```