A MODEL FOLLOWING INVERSE CONTROLLER WITH ADAPTIVE COMPENSATION FOR GENERAL AVIATION AIRCRAFT

A Dissertation by

Hugh S. Bruner

Master of Science, Wichita State University, 1986

Bachelor of Science, University of Kansas, 1979

Submitted to the Department of Aerospace Engineering
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

May 2007

© Copyright 2007 by Hugh S. Bruner

All Rights Reserved

A MODEL FOLLOWING INVERSE CONTROLLER WITH ADAPTIVE COMPENSATION FOR GENERAL AVIATION AIRCRAFT

I have examined the final copy of this dissertation for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Doctor of Philosophy with a major in Aerospace Engineering.

____
James Steck, Committee Chair

We have read this dissertation
and recommend its acceptance:

____
Benham Bahr, Committee Member

____
Roy Myose, Committee Member

____
Gawad Nagati, Committee Member

____
Kamran Rokhsaz, Committee Member

Accepted for the College of Engineering

____
Zulma Toro-Ramos, Dean

Accepted for the Graduate School

____
Susan Kovar, Dean

ACKNOWLEGEMENTS

Earning my Ph.D. has been a long process. It has been fun, but I never could have done it on my own. I have received help from so many people. Some are teachers, some are well-wishers, and some are family. Space doesn't allow me to name them all, so I must summarize the list. My apologies to those I don't name.

First, let me thank my advisor, Dr. James Steck. Dr. Steck sparked my first curiosities with neural networks and has been fanning the flames ever since with his encouragement and good advice. I appreciate his patience as I searched and searched, looking for a topic for which I could have a real passion. At last, I found it.

Next, I need to thank the Hawker Beechcraft Corporation for its support. I appreciate the financial support I received for my education through Hawker Beechcraft's tuition reimbursement program. It is a pleasure to work for an employer that recognizes the need to invest in the continuing education of its employees. But beyond that, I truly want to thank the people there that provided moral support and kept me going when my motivation was waning: Mr. Gary Bickel, Dr. Leland Johnson Jr., Mr. David Riemer, and Mr. Randy Nelson.

Most of all, I want to thank my loved ones that both cheered me on and, at the same time, endured my obsession with school: Diane Bruner, Heather and Justin Ryan, Stephanie and Tim Bruner, and my mother Maxine Bruner. I am grateful for their forbearance, and now I look forward to making amends for the time I didn't spend with them.

ABSTRACT

The theory for an adaptive inverse flight controller, suitable for use on General Aviation aircraft, is developed in this research. The objectives of this controller are to separate the normally coupled modes of the basic aircraft and thereby permit direct control of airspeed and flight-path angle, meet prescribed performance characteristics as defined by damping ratio and natural frequency, adapt to uncertainties in the physical plant, and be computationally efficient.

The three basic elements of the controller are a linear prefilter, an inverse transfer function, and an adaptive neural network compensator. The linear prefilter shapes accelerations required of the overall system in order to achieve the desired system performance characteristics. The inverse transfer function is used to compute the aircraft control inputs required to achieve the necessary accelerations. The adaptive neural network compensator is used to compensate for modeling errors during design or real-time changes in the physical plant. This architecture is patterned after the work of Calise, but differs by not requiring dynamic feedback of the state variables.

The controller is coded in ANSI C and integrated with a simulation of a typical General Aviation aircraft. Twenty-three cases are simulated to prove that the objectives for the controller are met. Among these cases are simulated stability and controllability failures in the physical plant, as well as several simulated failures of the neural network. With the exception of some bounded speed-tracking error, the controller is capable of continued flight with any foreseeable failure of the neural network.

Recommendations are provided for follow-on investigations by other researchers.

TABLE OF CONTENTS

Chapter Page

1. INTRODUCTION 1

1.1 Motivation 1
1.2 Research Objectives 2
1.3 Literature Review 3

2. THEORETICAL DEVELOPMENT OF CONTROLLER 9

2.1 Overview 9
2.2 Physical Plant 10
2.3 Linear Inverse Transfer Function 13
2.4 Linear Prefilter 24
2.5 Adaptive Neural Network Compensator 35

3. IMPLEMENTATION 39

3.1 Bonanza Sample Aircraft 39
3.2 Aircraft Simulation 41
3.3 Neural Network Configuration 42
3.4 Controller Coding 50

4. RESULTS 53

4.1 Introduction 53
4.2 Nominal Controller Performance 54
4.3 Failures Involving Stability and Controllability of the Physical Plant 75
4.4 Nonlinear Physical Plant Characteristics 110
4.5 Neural Network Failures 119

5. CONCLUSIONS 143

5.1 Summary of Results 143
5.2 Recommendations for Future Study 144

REFERENCES 146

APPENDICES 150

A. State Space Representation for the Linearized Aircraft Model 151
B. Equations of Motion Used in Modeling Aircraft Flight Dynamics 155
C. Program Listing 159

LIST OF TABLES

Table Page

1. Beechcraft Model F33C Summary Characteristics 11

2. Summary of Simulation Cases Examined 56

LIST OF FIGURES

Figure Page

1. The decline of General Aviation deliveries since 1978 1

2. Typical configuration of dynamic inversion controller 5

3. Functional decomposition of the controller 9

4. Summary architecture of adaptive inverse controller 10

5. Three-view drawing of the Beechcraft F33C Bonanza 12

6. Pass-through function of $\mathbf{G}^{-1}(s)\mathbf{G}(s)$ 16

7. Parallel operation of $\mathbf{G}(s)$ and $\mathbf{P}_A(s)$ 17

8. The combination of $\mathbf{G}(s)$ and $\mathbf{P}_A(s)$ acting as an integrator 18

9. MATLAB™ block diagram illustrating integration of $\dot{u}$ commands 18

10. Time history results of integration of $\dot{u}$ commands via $\mathbf{G}^{-1}(s)\mathbf{G}(s)$ 20

11. Time histories of state variables resulting from $\dot{u}$ commands 21

12. MATLAB™ block diagram illustrating integration of $\dot{\alpha}$ commands 22

13. Time history results of integration of $\dot{\alpha}$ commands via $\mathbf{G}^{-1}(s)\mathbf{G}(s)$ 23

14. Time histories of state variables resulting from $\dot{\alpha}$ commands 24

15. Simple second-order system 25

16. Rearrangement of second-order system, with no change in connectivity 26

17. Customizing second-order system for speed control 27

18. Functional grouping of elements within speed control 27

19. High-level arrangement of speed controller 28

20. Block diagram of flight-path angle controller 32

Figure Page

21. High-level arrangement of flight-path angle controller 32

22. A linear inverse controller with model following, but no adaptation 34

23. Function of an inverse controller 35

24. An approximate inverse transfer function with modeling error 35

25. Incorporation of anticipated modeling error into controller 36

26. Incorporation of adaptive neural network to approximate modeling error 37

27. Addition of neural network preserves integration function of controller 38

28. Summary of controller architecture 38

29. In-flight photo of Hawker Beechcraft Corporation's fly-by-wire Bonanza, CJ-144 40

30. Cockpit of Hawker Beechcraft Corporation's fly-by-wire Bonanza, CJ-144 40

31. Equivalency of neural network and state space matrix 43

32. Neural network training process 46

33. Data before training 47

34. Results of training network on $\dot{V}_p$ modeling error 48

35. Results of training network on $\dot{\alpha}$ modeling error 48

36. Topology of final network architecture 49

37. Case 1 time history 57

38. Case 2 time history 60

39. Case 3 time history 63

40. Case 4 time history 66

41. Case 5 time history 69

42. Case 6 time history 72

Figure Page

43. Case 7 time history 77

44. Case 8 time history 80

45. Case 9 time history 83

46. Case 10 time history 86

47. Case 11 time history 91

48. Case 12 time history 94

49. Case 13 time history 97

50. Case 14 time history 100

51. Case 15 time history 104

52. Case 16 time history 107

53. Simulated nonlinear lift coefficient characteristics 110

54. Simulated nonlinear pitching moment coefficient characteristics 111

55. Case 17 time history 113

56. Case 18 time history 116

57. Neuron activation function self-limits output 120

58. Case 19 time history 122

59. Case 20 time history 125

60. Case 21 time history 129

61. Case 22 time history 132

62. Case 23 time history 135

63. Speed controller block diagram with erroneous neural network signal 138

64. Speed sensitivity to oscillatory $\Delta\dot{u}$ errors from the neural network 140

Figure Page

65. Flight-path sensitivity to oscillatory $\Delta\dot{u}$ errors from the neural network..........................142

66. Flight-path sensitivity to oscillatory $\Delta\dot{\alpha}$ errors from the neural network 142

LIST OF SYMBOLS

b	Wing span
C_D	Drag coefficient
C_{D_K}	Partial derivative of drag coefficient with respect to lift coefficient squared
C_{D_0}	Drag coefficient at zero lift
C_L	Lift coefficient
C_{L_0}	Lift coefficient at zero angle of attack
C_{L_α}	Partial derivative of lift coefficient with respect to angle of attack
$C_{L_{\dot{\alpha}}}$	Partial derivative of lift coefficient with respect to non-dimensional rate of change of angle of attack
$C_{L_{\delta e}}$	Partial derivative of lift coefficient with respect to elevator deflection
$C_{L_{\delta tab}}$	Partial derivative of lift coefficient with respect to elevator tab deflection
$C_{L_{\delta t}}$	Partial derivative of lift coefficient with respect to thrust
C_{L_q}	Partial derivative of lift coefficient with respect to non-dimensional pitch rate
C_l	Rolling moment coefficient, in stability axis
C_{l_β}	Partial derivative of rolling moment coefficient with respect to sideslip angle
$C_{l_{\delta a}}$	Partial derivative of rolling moment coefficient with respect to aileron deflection
$C_{l_{\delta r}}$	Partial derivative of rolling moment coefficient with respect to rudder deflection
C_{l_p}	Partial derivative of rolling moment coefficient with respect to non-dimensional roll rate

C_{lr} Partial derivative of rolling moment coefficient with respect to non-dimensional yaw rate

C_M Pitching moment coefficient

C_{M_0} Pitching moment coefficient at zero angle of attack

C_{M_α} Partial derivative of pitching moment coefficient with respect to angle of attack

$C_{M_{\dot{\alpha}}}$ Partial derivative of pitching moment coefficient with respect to non-dimensional rate of change of angle of attack

$C_{M_{\delta e}}$ Partial derivative of pitching moment coefficient with respect to elevator deflection

$C_{M_{\delta tab}}$ Partial derivative of pitching moment coefficient with respect to elevator tab deflection

$C_{M_{\delta t}}$ Partial derivative of pitching moment coefficient with respect to thrust

C_{M_q} Partial derivative of pitching moment coefficient with respect to non-dimensional pitch rate

C_n Yawing moment coefficient, in stability axis

C_{n_β} Partial derivative of yawing moment coefficient with respect to sideslip angle

$C_{n_{\delta_a}}$ Partial derivative of yawing moment coefficient with respect to aileron deflection

$C_{n_{\delta_r}}$ Partial derivative of yawing moment coefficient with respect to rudder deflection

C_{n_p} Partial derivative of yawing moment coefficient with respect to non-dimensional roll rate

C_{n_r} Partial derivative of yawing moment coefficient with respect to non-dimensional yaw rate

LIST OF SYMBOLS (continued)

C_y Side force coefficient, in stability axis

C_{y_β} Partial derivative of side force coefficient with respect to sideslip angle

$C_{y_{\delta_a}}$ Partial derivative of side force coefficient with respect to aileron deflection

$C_{y_{\delta_r}}$ Partial derivative of side force coefficient with respect to rudder deflection

C_{y_p} Partial derivative of side force coefficient with respect to non-dimensional roll rate

C_{y_r} Partial derivative of side force coefficient with respect to non-dimensional yaw rate

$\bar{c}$ Mean aerodynamic chord length

F_{X_b} X component of force in body axis

F_{Y_b} Y component of force in body axis

F_{Z_b} Z component of force in body axis

g Gravitational constant

h Altitude

$\dot{h}$ Rate of change in altitude

I Identity matrix

I_{xx_b} Rolling moment of inertia, in body axis

I_{yy_b} Pitching moment of inertia, in body axis

I_{zz_b} Yawing moment of inertia, in body axis

KEAS Equivalent airspeed, in knots

k Computational step in discrete time domain

knots	Nautical miles per hour
L_b	Rolling moment, in body axis
L_s	Rolling moment, in stability axis
LTI	Linear time invariant
M_b	Pitching moment, in body axis
M_s	Pitching moment, in stability axis
m	Airplane mass
N_b	Yawing moment, in body axis
N_s	Yawing moment, in stability axis
n_z	Load factor in direction of lift vector
P_b	Roll rate, in body axis
P_S	Roll rate, in stability axis
PLA	Power-lever angle
$\hat{p}$	Non-dimensional roll rate
psf	pounds per square foot
Q_b	Pitch rate, in body axis
Q_s	Pitch rate, in stability axis
$\hat{q}$	Non-dimensional pitch rate
$\overline{q}$	Dynamic pressure
R_b	Yaw rate, in body axis

R_s	Yaw rate, in stability axis
$\hat{r}$	Non-dimensional yaw rate
$\mathbf{r}$	Command vector
rad	radians
S	Wing area
s	Laplace variable
sec	seconds
T	Period
t	time
U_b	X component of velocity in body axis
$\mathbf{u}$	Control vector
u_b	Perturbation X velocity in stability axis
V_b	Y component of velocity in body axis
V_p	Airplane true airspeed
W	Airplane weight
W_b	Z component of velocity in body axis
$\mathbf{x}$	State vector
$\mathbf{y}$	Output vector

Greek Symbols

α	Angle of attack
$\dot{\alpha}$	Rate of change of angle of attack

$\hat{\dot{\alpha}}$	Non-dimensional rate of change of angle of attack
β	Sideslip angle
γ	Flight-path angle
Δ	Physical plant rate error vector
$\Delta\dot{\alpha}$	Neural network angle of attack rate error
Δu_b	Neural network velocity rate error
δ_a	Aileron deflection
δ_e	Elevator deflection
δ_t	Thrust command
ε	Simulated neural network error signal
ζ	Damping ratio
Θ	Airplane pitch attitude angle in Earth axis system
ρ	Density of air
Φ	Airplane bank angle attitude in Earth axis system
Ψ	Airplane heading angle in Earth axis system
ω	Frequency
ω_n	Natural frequency

Subscripts

A	Approximation
b	Body axis
G	Relating to system G(s)

LIST OF SYMBOLS (continued)

I	Inverse
P	Physical plant
s	Stability axis

CHAPTER 1

INTRODUCTION

1.1 Motivation

Data from General Aviation manufacturers show that deliveries of new aircraft have significantly declined since 1978 [1]. This decline has been attributed to a number of factors: the price of fuel, product liability, the declining number of military veteran pilots, competing modes of transportation, the high cost of learning to fly, the high cost of maintaining pilot proficiency, and perceptions of safety.

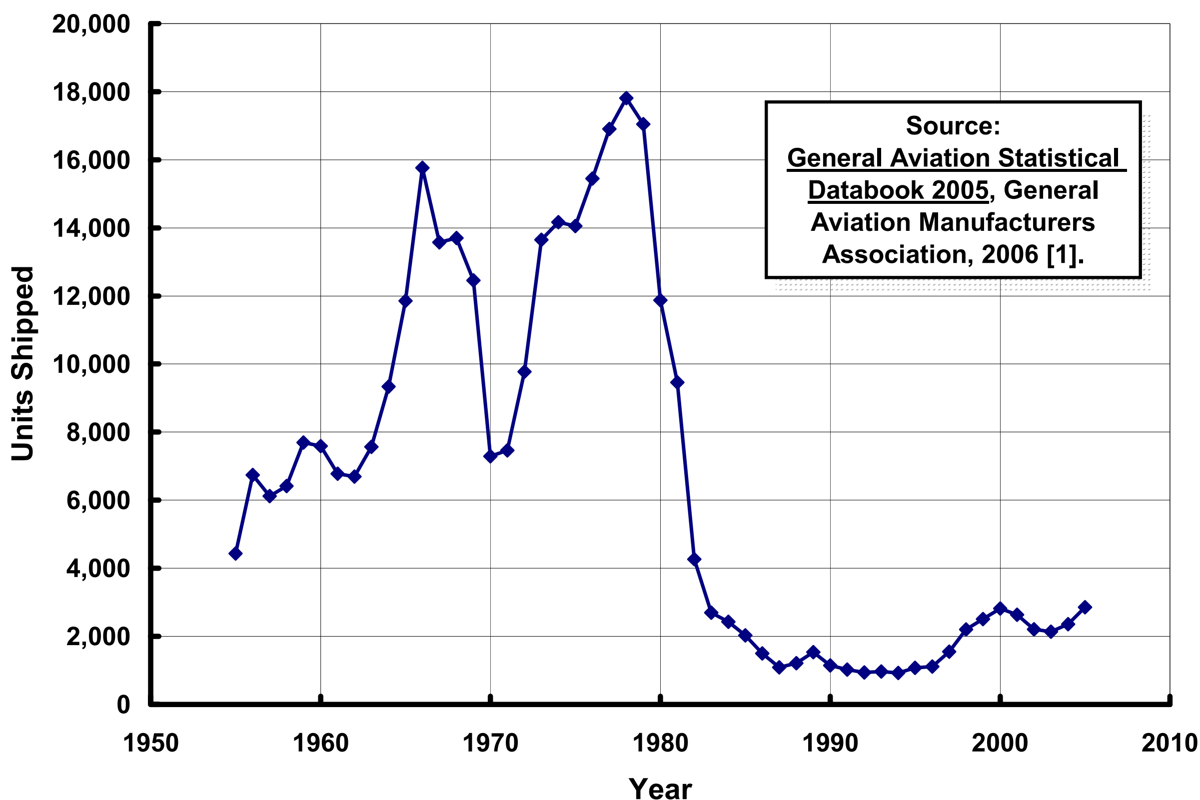


Figure 1: The decline of General Aviation deliveries since 1978

NASA speculates that a renewed interest in personal aviation might be stimulated if these last three factors are addressed [2]. Aircraft that are easier to fly would reduce the associated cost of learning to fly and, additionally, eliminate the burdensome requirements of maintaining instrument proficiency. Furthermore, safety would be increased if consistent predictable flying qualities could be maintained, regardless of the aircraft failure state. These speculations by NASA suggest that a resurgence in personal flying may be possible if aircraft can be built with easy-to-use intuitive flight controls that adapt to changes in the aircraft operating state.

The research described herein is motivated by the desire to make aviation more accessible to the masses who have the desire for personal air transportation but do not have the free time available for learning the traditional piloting technique.

1.2 Research Objectives

Traditional aircraft controls have a one-to-one correspondence with the control surfaces and the throttle [3]. In this "stick and rudder" mode of flying, the pilot does not control the end-goal states of the aircraft: typically speed, altitude, and heading. The pilot serves as, at least, a second-order integrator as he "closes the loop" between aircraft accelerations and desired end-states. This task is further complicated by the coupled oscillatory modes typically found in General Aviation aircraft. Workload, accuracy, and safety are improved if the pilot is able to more directly command the desired end-state objectives.

The objectives of this research are intended to address these issues in order to enable more direct control of the aircraft. To achieve this, a controller architecture will be derived that does the following:

1. Separates the normally coupled modes of the aircraft.
2. Permits direct control of flight-path angle, γ, and speed, $V_{p.}$

3. Achieves prescribed overall system dynamic characteristics.
4. Adapts to uncertainties in the system, including real-time changes to the physical plant, such as failures.
5. Is computationally efficient.

The overarching goal is to strive for a system that is simple, robust, and relevant to General Aviation aircraft.

This research will address only the longitudinal characteristics of the aircraft. It is expected that the same method can be applied to the lateral-directional modes.

1.3 Literature Review

Significant work has taken place over the past decade with respect to advanced flight controls. Beringer [3] looked at the application of "Performance Control Systems" (PCS) to General Aviation aircraft. Significant reductions in workload and piloting error were found if the pilot directly controlled "higher level performance variables," such as rate of climb or airspeed. Piloted simulations were conducted that permitted back-to-back evaluation of conventional controls versus decoupled fly-by-wire controls. Decoupled flight controls showed about one-third the control activity (reduced workload) and one-third the tracking error (reduced piloting error), when compared to conventional flight controls.

Duerksen examined the application of adaptive flight controls on General Aviation aircraft [4]. Duerksen accomplished his adaptation using fuzzy logic. Decoupled flight controls were implemented for speed and flight-path angle control. The primary objective for this effort was to develop a generic controller that could be independent of aircraft type. To illustrate the concept, the technique was applied to simulations of two disparate aircraft types: a Beechjet 400A and Beechcraft Bonanza. One aircraft is a jet, and the other a piston-powered single-

engine aircraft. Using fuzzy logic, the two aircraft were made to handle similarly. The fuzzy logic algorithms developed by Duerksen are the algorithms later implemented by Beringer in his work on PCS controllers [3].

Calise, Rysdyk et al. have accomplished significant work in the control of nonlinear systems using a technique called "dynamic inversion" [5, 6, 7, 8, 9]. This control architecture is illustrated in Figure 2. For any physical system described by the transfer function f, the key element of this technique is to find an approximate inverse transfer function $\hat{f}^{-1}$. A "pseudo control variable," ν, is calculated that closely represents the desired rates of change of the state variable x. This pseudo control variable is fed into the inverse transfer function to create control inputs, δ, to the physical system. If $\hat{f}^{-1}$ is a perfect inverse of f, then $\hat{f}^{-1} f = \mathbf{I}$, and the output from f would be precisely ν. The integration of ν would give x. However, it is stated that $\hat{f}^{-1}$ is only an approximation of a perfect inverse, and therefore there is some inversion error. This inversion error might arise from modeling error or uncertainties in the physical system. To compensate for this inversion error, an adaptive neural network is used to "learn" what the error is and, in turn, provide a compensating signal back into the controller. The pseudo control variable, ν, has the beneficial addition of this compensating signal. The intent is to null out the inversion error. The handling qualities and linear control blocks upstream convert the pilot control inputs into the desired rates, as required to achieve the desired handling characteristics.

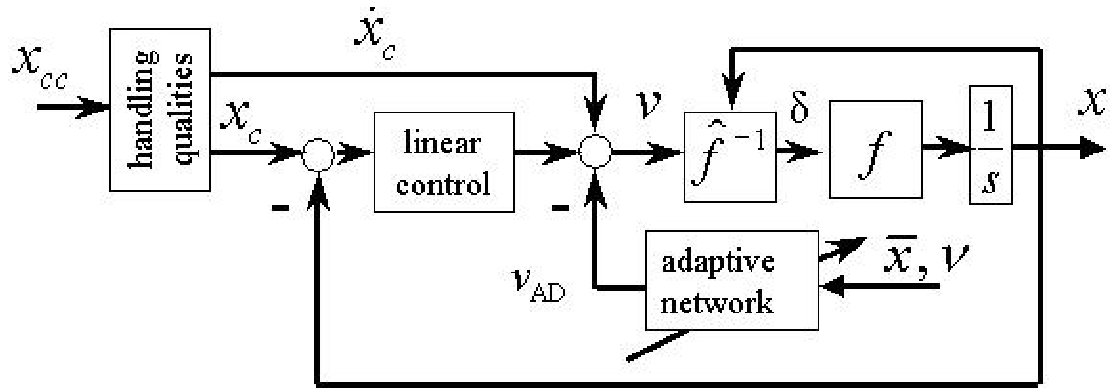


Figure 2: Typical configuration of dynamic inversion controller [9]

Various applications of this technique have been demonstrated in simulation. A rate and attitude command system was demonstrated by Calise and Rysdyk for the highly nonlinear XV-15 tilt-rotor aircraft [5]. MacFarland and Calise demonstrated angle of attack and bank angle control for an agile anti-air missile in regions of significant aerodynamic nonlinearity [6]. Hovakimyan et al. used dynamic inversion techniques for the control of an R-50 helicopter [7]. Hovakimyan's application is interesting in that an observer was included as part of the controller to estimate plant states not directly measurable. In this instance, the neural network learned from the observer outputs, not from direct measurement of the states. Rysdyk expanded the XV-15 controller to include lateral directional control [8]. This work included an expansion of the work by Calise and Rysdyk [5] but, more importantly, included a proof that showed the outputs of a neural network are bounded. Rysdyk and Agarwal applied the technique to implementation of a "Total Energy Control System" [9], as originally proposed by Lambregts [10]. In this work, the concept of "pseudo control hedging" was proposed as a means for "anti-windup protection" in the neural network.

Research at Wichita State University (WSU) by Steck, Rokhsaz et al., has involved integration of the Calise methods with the works of Lambregts and Duerksen to implement adaptive feedback control into General Aviation aircraft [11, 12, 13, 14]. Investigations have been made into the FAA certification of neural networks in General Aviation aircraft [13]. Considerable effort has been invested in simulations, both desktop simulations with MATLAB™ and pilot-in-the loop simulations [14]. These simulations investigated the ability of the WSU algorithms to cope with aircraft failures involving both stability ("A matrix" failures) and/or controllability ("B matrix" failures). Work is ongoing to flight-test an adaptive controller in a fly-by-wire test-bed aircraft owned by Hawker Beechcraft Corporation.

While not directly related to aircraft, another work of interest is that conducted by Hess [15]. These investigations looked at the application of recursive neural networks (RNN) in the control of large warships. Recursion is of interest because it allows for dynamics to be included in the neural network. This suggests that a better representation of modeling error, particularly dynamic errors, may be obtained with a recursive neural network. Inputs to the neural network are taken from a number of prior time steps. Finite differences between time steps enable the network to approximate first-order derivatives, or higher, depending on the number of preceding time steps included. This work also provides good examples of substantial pre- and post-processing of network parameters in generalizing the complexity of maneuvering a large ship in extreme sea-states. The final neural network for this application involved 88 inputs, two hidden layers, and 134 computational nodes.

Many observations were gleaned from these past works that are applicable to the current topic.

First, the work of Calise et al. was applied to highly nonlinear systems, where the inverse transfer function depended on dynamic state feedback to adequately characterize the inverse. For General Aviation aircraft, the problem is generally linear. Assuming linearity suggests that a simple inverse transfer function can be formulated to enable strictly feed-forward operations. Where this assumption of linearity may not be valid, it is proposed that a greater emphasis can be placed on the neural network to address residual nonlinearities.

Second, once again taking credit for largely linear aerodynamics, it is proposed that the algebraic expressions for the inverse transfer function developed by Steck et al. can be replaced by a linear time invariant (LTI) state space representation of the inverse transfer function. Linear algebra should provide for faster execution within the controller. Furthermore, opportunities are identified to improve upon the flight-path control laws by controlling normal load factor, n_z, rather than pitch rate acceleration, $\ddot{\theta}$, as the means to achieve a desired flight-path. Steck addressed the entire six degree of freedom problem. This work only addresses the longitudinal case.

Finally, it is proposed that since the inverse transfer function is a dynamic system, it is safe to assume that the inversion error is likely to involve dynamics also. Consequently, the recursive neural networks discussed by Hess may provide benefit in accurately capturing any dynamics present in the inversion error.

With these observations in mind, the following items are addressed in this dissertation:

1. An adaptive inverse controller similar to those presented by Calise et al. will be used as a starting point.
2. A closed form of the inverse transfer function will be implemented, not dependent on dynamic feedback.

3. A refined control law for flight-path angle will be derived using normal load factor, n_z, as the key variable.
4. Inversion errors will be cancelled using a recursive neural network.

This approach will be used to derive a control architecture that satisfies the objectives identified in subchapter 1.2.

CHAPTER 2

THEORETICAL DEVELOPMENT OF CONTROLLER

2.1 Overview

The functional objectives of the controller are to enable the pilot to directly control the aircraft end-states and to have the overall system react with the desired dynamic response. These objectives are decomposed in Figure 3 into a number of subfunctions. The pilot commands his desired end-state. These commands are interpreted in the context of the desired overall system dynamics. Once the path from the current-state to the end-state is determined, aircraft control movements are computed and fed into the aircraft. The aircraft will react in response to these control inputs. Feedback from the airplane response is used to assure consistency with the desired dynamic characteristics. Furthermore, feedback from the airplane response may lead to modification of the control inputs to achieve better overall system performance.

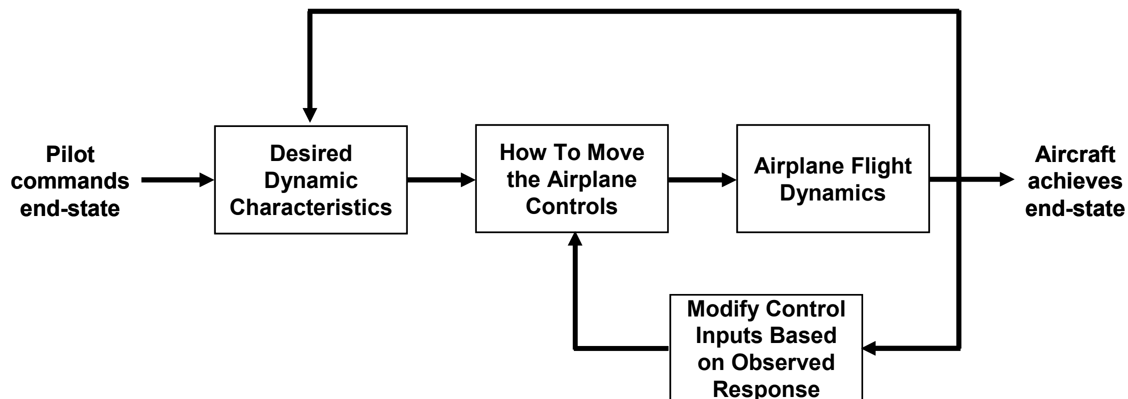


Figure 3: Functional decomposition of the controller

These subfunctions map one-for-one onto the architecture of the overall system. The controller consists of three major elements: a linear model prefilter, a linear inverse transfer function, and an adaptive neural network compensator. A block diagram showing the

configuration of the controller, plus the physical plant, is shown in Figure 4. In summary, the linear prefilter computes the rates "desired" of the system to achieve the commanded values of speed and flight-path angle. These desired rates are combined with the compensating output of the neural network to compute the "required" rates. The adaptive neural network compensates for modeling error or changes in the physical plant that may occur during operation. Given these rates, the inverse transfer function then computes the control inputs to the physical plant necessary to achieve the desired rates and, as a consequence of integration, to achieve the commanded speed and flight-path angle. A detailed description of each element follows.

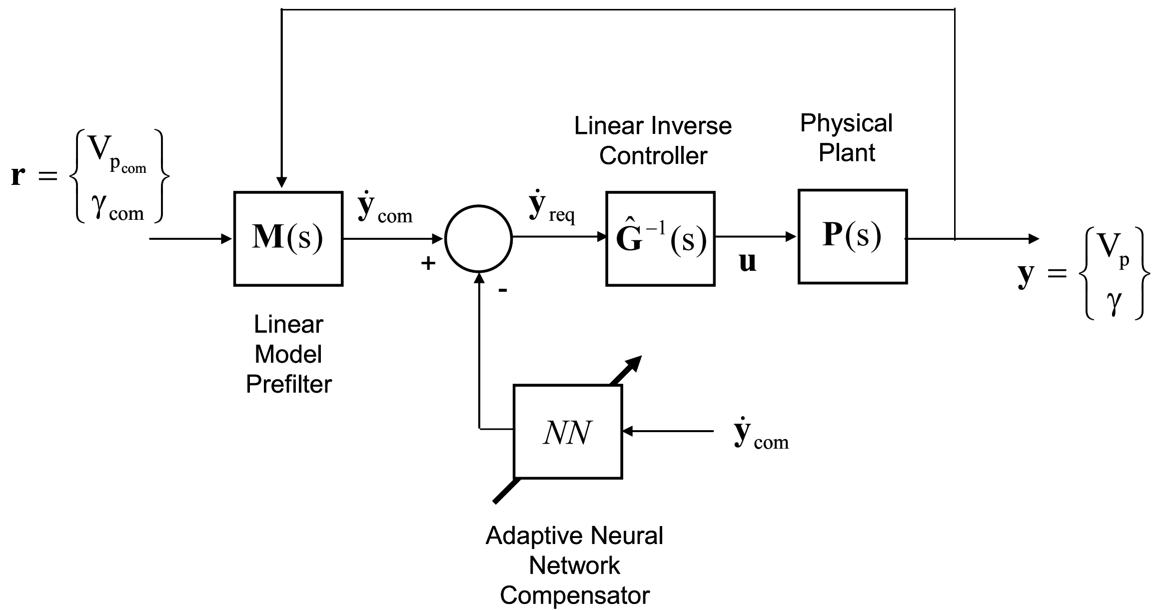


Figure 4: Summary architecture of adaptive inverse controller

2.2 Physical Plant

The physical plant, $P(s)$, may be any physical system. For this research, the physical plant is assumed to be a Beechcraft F-33C Bonanza. The use of this subject aircraft is consistent with previous work accomplished at Wichita State University [11, 12, 13, 14] and complements

ongoing flight research being conducted jointly by Wichita State University and Hawker Beechcraft Corporation. The F-33C Bonanza is a four-passenger, low-wing monoplane, typical of high-performance General Aviation airplanes [16]. The technical specifications of the aircraft are summarized in Table 1. A three-view illustration of the aircraft is shown in Figure 5.

TABLE 1

BEECHCRAFT MODEL F33C SUMMARY CHARACTERISTICS

Wing Area	181 square feet
Wing Span	33 feet 6 inches
Engine	One Teledyne Continental Motors Corporation IO-520, rated at 285 horsepower
Maximum Takeoff Weight	3400 pounds
Maximum Structural Cruising Speed	167 KIAS
Stall Speed (Max Takeoff Weight, Landing Flaps)	52 KIAS

The controls available for the aircraft are elevator deflection, δ_e, aileron deflection, δ_a, rudder deflection, δ_r, and thrust command, δ_t. Since this research is limited to the longitudinal mode, only elevator and thrust commands are utilized here.

The aerodynamics of this airplane are essentially linear up to the angle of attack for aerodynamic stall. However, the drag characteristics are nonlinear, exhibiting a typical parabolic drag polar. The validity of the proposed controller is not limited to physical plants that are linear in behavior. In fact, it will be shown that the controller shows satisfactory performance even when artificial nonlinearities are introduced into the Bonanza aerodynamic characteristics.

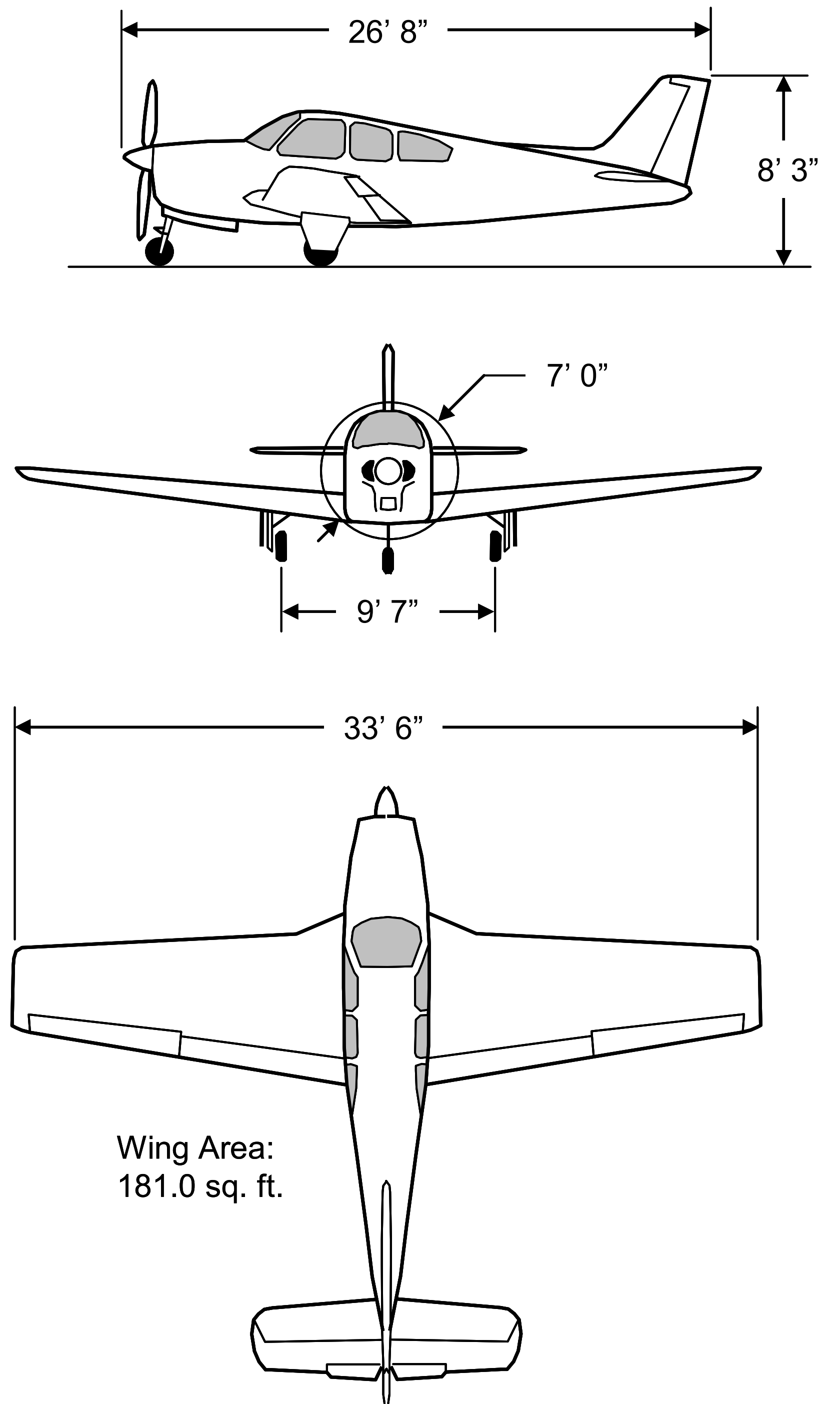


Figure 5: Three-view drawing of the Beechcraft F33C Bonanza [16]

For some purposes, it is convenient to invoke a strictly linear approximation of the physical plant. This linear approximation is noted as $\mathbf{P}_A(s)$. Furthermore, it is convenient to express $\mathbf{P}_A(s)$ as an LTI state space model, such that

$$\dot{\mathbf{x}}_A = \mathbf{A}_A \mathbf{x}_A + \mathbf{B}_A \mathbf{u} \tag{2.1}$$

$$\mathbf{y}_A = \mathbf{C}_A \mathbf{x}_A + \mathbf{D}_A \mathbf{u} \tag{2.2}$$

where

$$\mathbf{u} = [\delta_e \quad \delta_t]^T \tag{2.3}$$

This state space model can be represented by the notation

$$\mathbf{P}_A(s) = \left[\begin{array}{c|c} \mathbf{A}_A & \mathbf{B}_A \\ \hline \mathbf{C}_A & \mathbf{D}_A \end{array}\right] \tag{2.4}$$

Derivation of the state space model for a typical General Aviation aircraft with conventional controls, such as the Bonanza, is shown in Appendix A. This state space model will be used in the derivation of the controller. It is shown that

$$\mathbf{x}_A = \begin{Bmatrix} u_b \\ \alpha \\ q \\ \theta \end{Bmatrix}, \text{ and } \mathbf{y}_A = \begin{Bmatrix} u_b \\ \gamma \end{Bmatrix} \tag{2.5}$$

It is important to note at this point that $\mathbf{D}_A = 0$ for such aircraft.

2.3 Linear Inverse Transfer Function

The transfer function of a physical system calculates system outputs, **y**, based on given inputs, **u**. The inverse transfer function does the inverse, as the name implies. The inverse transfer function calculates the system inputs, **u**, necessary to achieve a given system output, **y**. In the Laplace domain, the transfer function is represented as

$$\mathbf{Y}(s) = \mathbf{G}(s)\mathbf{U}(s) \tag{2.6}$$

The inverse transfer function is represented as

$$\mathbf{U}(s) = \mathbf{G}^{-1}(s)\,\mathbf{Y}(s) \tag{2.7}$$

Inserting equation (2.6) into equation (2.7), it is seen that

$$\mathbf{G}^{-1}(s)\,\mathbf{G}(s) = \mathbf{I} \tag{2.8}$$

where I is the identity matrix. It is important to note that **G**(s) is an operator in the Laplace domain. Consequently, its multiplicative inverse, $\mathbf{G}^{-1}(s)$, exists only in the Laplace domain. If the choice is to work in the time domain, **G**(s) is represented as an LTI system **G**(t). Another LTI system may be found that is the inverse LTI operator $\mathbf{G}^{-1}(t)$ such that

$$\mathbf{G}^{-1}(\mathbf{G}(t)) = \mathbf{I} \tag{2.9}$$

If $\mathbf{G}(s)$ is realized as an LTI system, such as $\mathbf{P}_A(s)$, described in state space format, the existence of $\mathbf{G}^{-1}(s)$ is dependent on satisfying both of the following two conditions [17]:

1. **D** must be square.
2. **D** must be invertible.

A consequence of these two conditions is that **D** must be a square matrix of rank N, where N is the number of controls. In the case of typical General Aviation aircraft, such as the subject aircraft, there are two longitudinal controls. Therefore, N equals two: one for the elevator and one for the throttle. Consequently, only N output variables can be controlled. In the case of the subject aircraft, two output variables can be controlled.

If $\mathbf{G}(s)$ is realized as a state space system

$$\mathbf{G}(s) = \left[\begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline \mathbf{C} & \mathbf{D} \end{array}\right] \tag{2.10}$$

it can be shown [17] that the inverse transfer function is

$$\mathbf{G}^{-1}(s) \equiv \left[\begin{array}{c|c} \mathbf{A}_I & \mathbf{B}_I \\ \hline \mathbf{C}_I & \mathbf{D}_I \end{array}\right] = \left[\begin{array}{c|c} \mathbf{A}-\mathbf{B}\mathbf{D}^{-1}\mathbf{C} & -\mathbf{B}\mathbf{D}^{-1} \\ \hline \mathbf{D}^{-1}\mathbf{C} & \mathbf{D}^{-1} \end{array}\right] \tag{2.11}$$

At this point, a judicious choice is required for $\mathbf{G}(s)$. A first reaction would be to choose $\mathbf{G}(s) = \mathbf{P}_A(s)$. This choice would suggest direct control of the end-states via the inverse transfer function. However, this will not work here because $\mathbf{D}_A = \mathbf{0}$ and therefore violates condition (2) above. A different transfer function $\mathbf{G}(s)$ must be found. A function that produces $\mathbf{y} = [\dot{u}_b \quad \dot{\alpha}]^T$ satisfies the conditions. It will be shown in subchapter 2.4 that controlling $\dot{u}_b$ and $\dot{\alpha}$ enables control of the end-states of V_p and γ as well. For this case the state space matrices are constructed like this:

1. $\mathbf{A}$ and $\mathbf{B}$ are the same as $\mathbf{A}_A$ and $\mathbf{B}_A$.
2. $\mathbf{C}$ is constructed of the first two rows of $\mathbf{A}_A$ (the $\dot{u}_b$ and $\dot{\alpha}$ rows).
3. $\mathbf{D}$ is similarly constructed of the first two rows of $\mathbf{B}_A$ (the $\dot{u}_b$ and $\dot{\alpha}$ rows).

$\mathbf{D}$ now satisfies conditions (1) and (2), and therefore an inverse of this transfer function exists. Specifically,

$$\mathbf{G}(s) = \left[\begin{array}{c|c} \mathbf{A}_G & \mathbf{B}_G \\ \hline \mathbf{C}_G & \mathbf{D}_G \end{array}\right] \tag{2.12}$$

After making the substitutions as noted above,

$$\mathbf{G}(s) = \left[\begin{array}{cccc|cc} & & \mathbf{A}_A & & & \mathbf{B}_A \\ \hline A_A(1,1) & A_A(1,2) & A_A(1,3) & A_A(1,4) & B_A(1,1) & B_A(1,2) \\ A_A(2,1) & A_A(2,2) & A_A(2,3) & A_A(2,4) & B_A(2,1) & B_A(2,2) \end{array}\right] \tag{2.13}$$

From the definition of the inverse transfer function in equation (2.11),

$$\mathbf{G}^{-1}(s) = \left[\begin{array}{c|c} \mathbf{A}_G - \mathbf{B}_G\mathbf{D}_G^{-1}\mathbf{C}_G & -\mathbf{B}_G\mathbf{D}_G^{-1} \\ \hline \mathbf{D}_G^{-1}\mathbf{C}_G & \mathbf{D}_G^{-1} \end{array}\right] \tag{2.14}$$

$$\mathbf{G}^{-1}(s) = \left[\begin{array}{c|c} \mathbf{A}_I & \mathbf{B}_I \\ \hline \mathbf{C}_I & \mathbf{D}_I \end{array}\right] \tag{2.15}$$

$$\dot{\mathbf{x}}_I = \mathbf{A}_I \mathbf{x}_I + \mathbf{B}_I \mathbf{u} \tag{2.16}$$

$$\mathbf{y}_I = \mathbf{C}_I \mathbf{x}_I + \mathbf{D}_I \mathbf{u} \tag{2.17}$$

and

$$\mathbf{u} = [\dot{u}_b \quad \dot{\alpha}]^T_{req} \tag{2.18}$$

$$\mathbf{y}_I = [\delta_e \quad \delta_t]^T \tag{2.19}$$

While not obvious, it should be pointed out that $\mathbf{x}_I = -\mathbf{x}_A$. At the same time the LTI system $\mathbf{G}^{-1}(s)$ is computing the necessary control inputs, it is calculating its own internal representation of the physical plant state variables. The dynamics of the physical plant enter into the calculations via this internal representation. This is in contrast with the dynamic inverse methods used by Calise or Steck where the airplane state is explicitly fed back for purposes of calculating the dynamic inverse.

$\mathbf{G}^{-1}(s)$ acts upon $[\dot{u}_b \quad \dot{\alpha}]^T_{req}$ to produce a set of control inputs $[\delta_e \quad \delta_t]^T$. When fed into $\mathbf{G}(s)$, these control inputs will produce $[\dot{u}_b \quad \dot{\alpha}]^T$. A block diagram of this sequence of operations is illustrated in Figure 6.

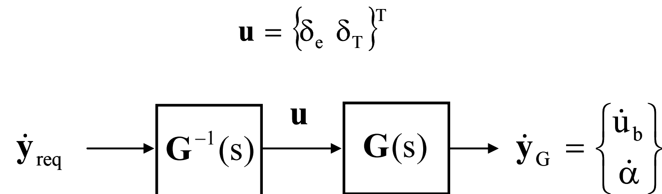


Figure 6: Pass-through function of $\mathbf{G}^{-1}(s)\mathbf{G}(s)$

By a change to the $\mathbf{C}_A$ matrix (refer to Appendix A, equation (A23)), $\mathbf{P}_A(s)$ can also produce $\mathbf{y} = \begin{bmatrix} \dot{u}_b & \dot{\alpha} \end{bmatrix}^T$. With this context in mind, a parallel set of transfer functions may be envisioned, as shown in Figure 7.

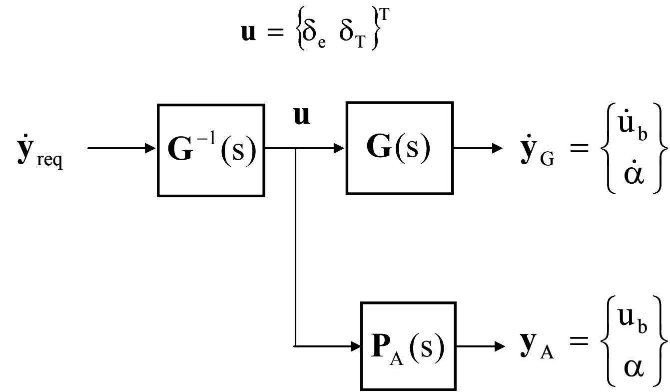


Figure 7: Parallel operation of $\mathbf{G}(s)$ and $\mathbf{P}_A(s)$.

A vector of required rates, $\dot{\mathbf{y}}_{req}$, is input into $\mathbf{G}^{-1}(s)$. The inverse transfer function produces control inputs which, in turn, go to both $\mathbf{G}(s)$ and $\mathbf{P}_A(s)$. $\mathbf{G}(s)$ produces the rates, while $\mathbf{P}_A(s)$ produces u_b and α. This parallel structure is effectively an integrator, as shown in Figure 8. It should be noted that $\mathbf{G}(s)$ is only an abstraction used in formulating $\mathbf{G}^{-1}(s)$. No physical instance of $\mathbf{G}(s)$ is found in the system.

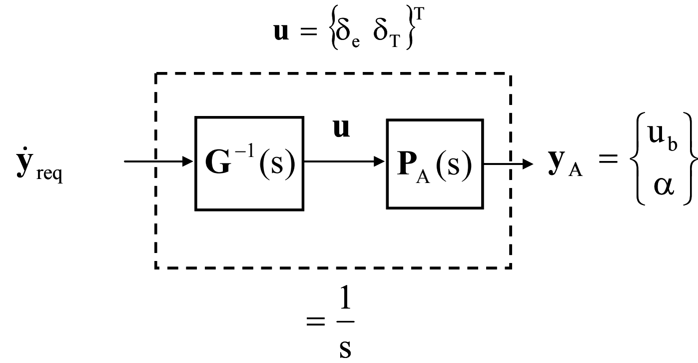


Figure 8: The combination of $\mathbf{G}(s)$ and $\mathbf{P}_A(s)$ acting as an integrator

The equivalency of $\mathbf{G}^{-1}(s)\mathbf{P}_A(s)$ to an integrator is a key concept in the development of this controller. This equivalency is illustrated in the examples below.

A MATLAB™ model was constructed for the transfer function $\mathbf{G}^{-1}(s)\mathbf{P}_A(s)$, as shown in Figure 9. The inputs to the transfer function are a $\dot{u}$ command and an $\dot{\alpha}$ command. The command for $\dot{u}$ is a repeating pulse, 10 feet/sec^2 in amplitude and two seconds wide. The command for $\dot{\alpha}$ is zero. These inputs are plotted in Figure 10, as are the outputs from the system. It is obvious to see that the $\dot{u}$ pulses, 10 feet/sec^2 wide and 2 seconds wide, integrate into changes in velocity of 20 feet/sec with each pulse. Since no $\dot{\alpha}$ is commanded, α remains constant.

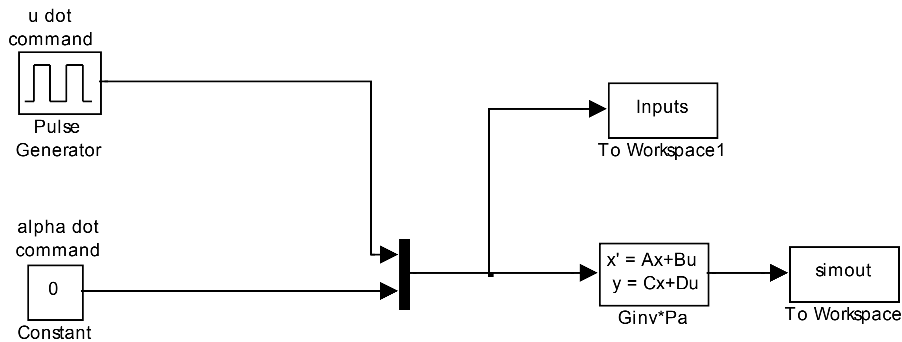


Figure 9: MATLAB™ block diagram illustrating integration of $\dot{u}$ commands

Figure 11 shows all the state variables for the airplane system, $\mathbf{x} = \begin{bmatrix} u & \alpha & q & \theta \end{bmatrix}^T$. It is important to note that only the "commanded" variables, u and α, are "under control." The other state variables will take on whatever values come as a consequence of achieving the commanded values of $\dot{u}$ and $\dot{\alpha}$. For this example, where speed is increased at a fixed angle of attack, lift is no longer in equilibrium with weight. A vertical acceleration normal to the flight-path occurs, resulting in a curving flight-path as evident in the pitch rate and pitch angle traces. The mathematical relationships between these parameters will be examined in the following subchapter. As a reminder, flight-path angle, γ, is not a state variable but rather a linear combination of state variables α and θ.

Figure 12 shows a similar model but with $\dot{\alpha}$ being stimulated with a pulse .05 radians/sec in amplitude and two seconds wide. The command for $\dot{u}$ is zero. The inputs and corresponding outputs are shown in Figure 13. Once again, it can be seen that the integration of an $\dot{\alpha}$ pulse .05 radians/sec by two seconds results in a change in α of .10 radians with each pulse. Velocity stays constant at zero. Figure 14 shows how all of the state variables change in response. Similar to the preceding example, a change in lift occurs, but as a result of changing the angle of attack in this case. An acceleration normal to the flight-path occurs, resulting in a similar but somewhat different curvilinear flight-path.

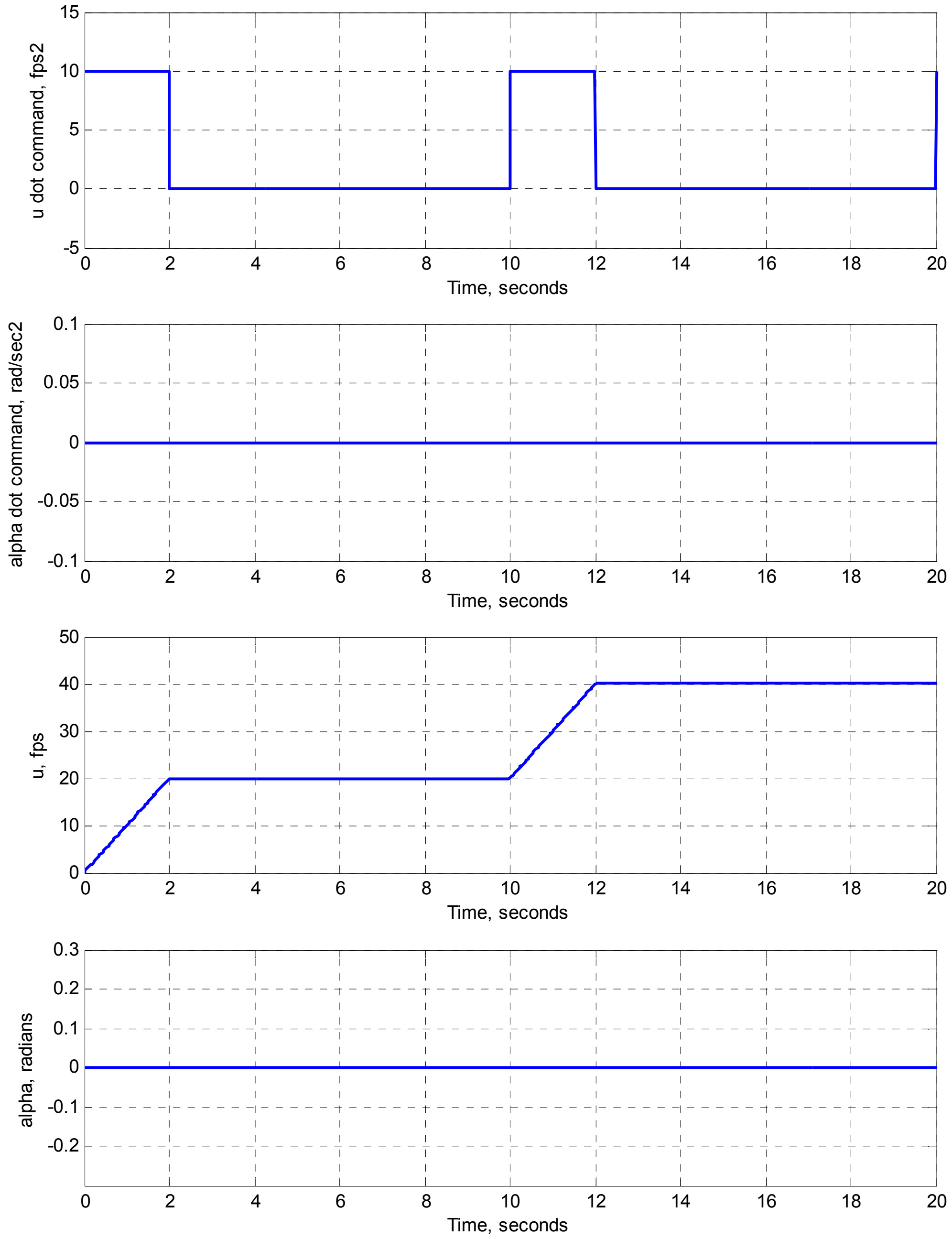


Figure 10: Time history results of integration of $\dot{u}$ commands via $\mathbf{G}^{-1}(s)\mathbf{P}_A(s)$

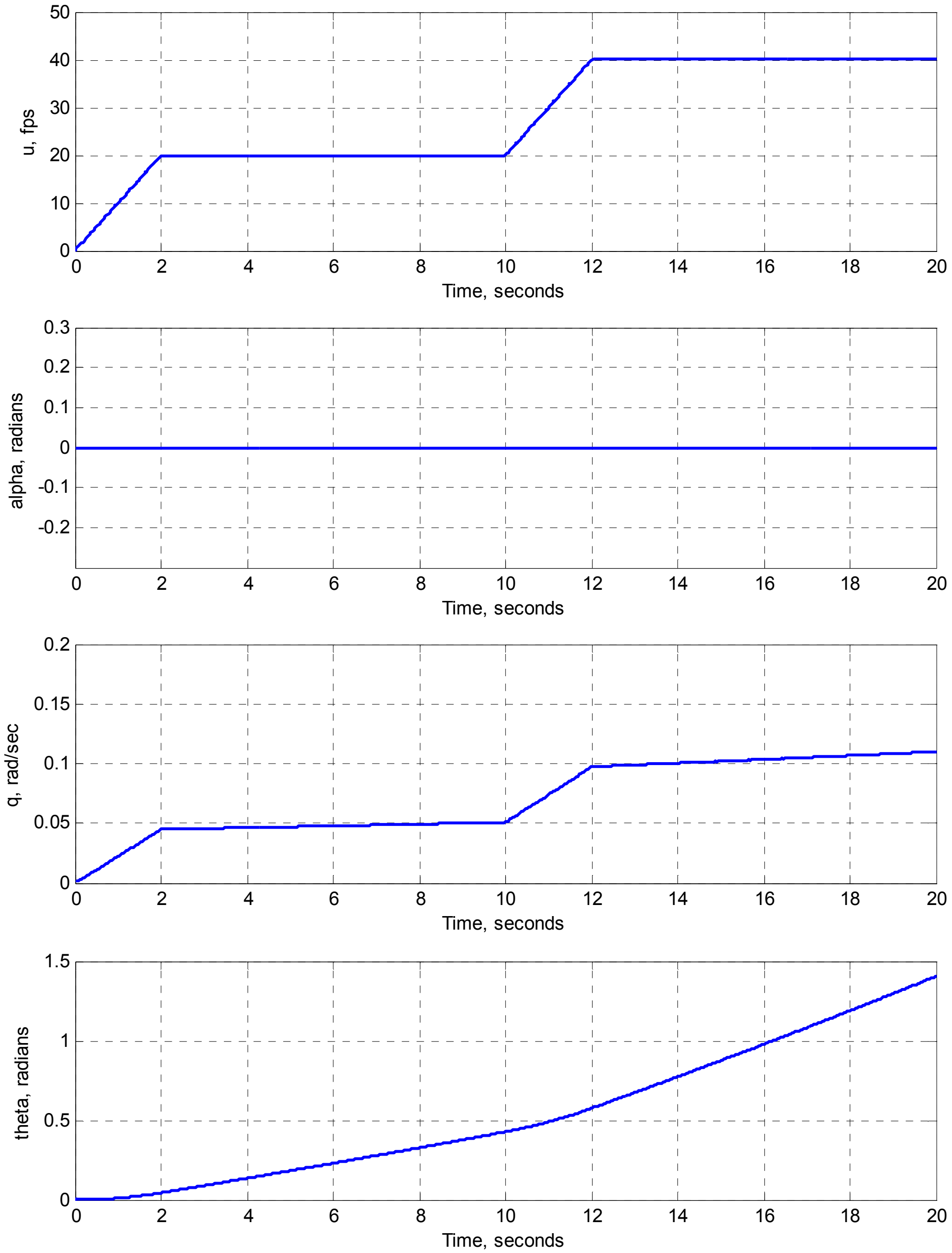


Figure 11: Time histories of state variables resulting from $\dot{u}$ commands

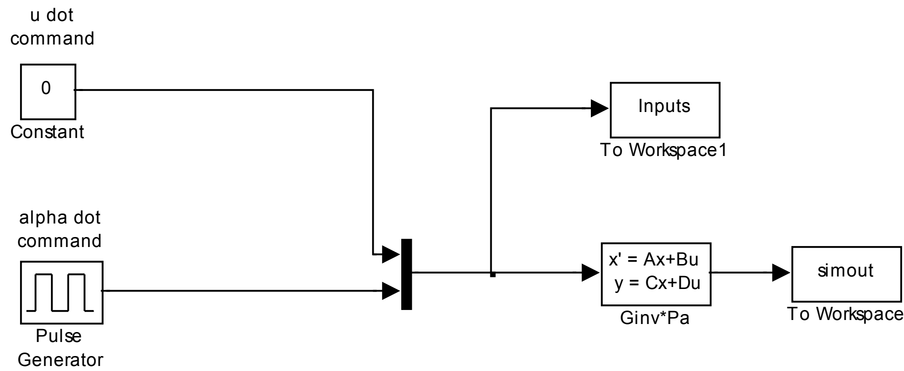


Figure 12: MATLAB™ block diagram illustrating integration of $\dot{\alpha}$ commands

The important point to be learned from these examples is that the linear inverse transfer function, combined with the physical plant, achieves exactly the integral of the input parameters: no more, no less. No direct control of the other state variables is implied or expected. Some other control logic must be applied if the objective is to control the other state variables or, in the case of flight-path angle, a linear combination of the state variables. Hence, the role of the next element of the controller: the linear prefilter.

.

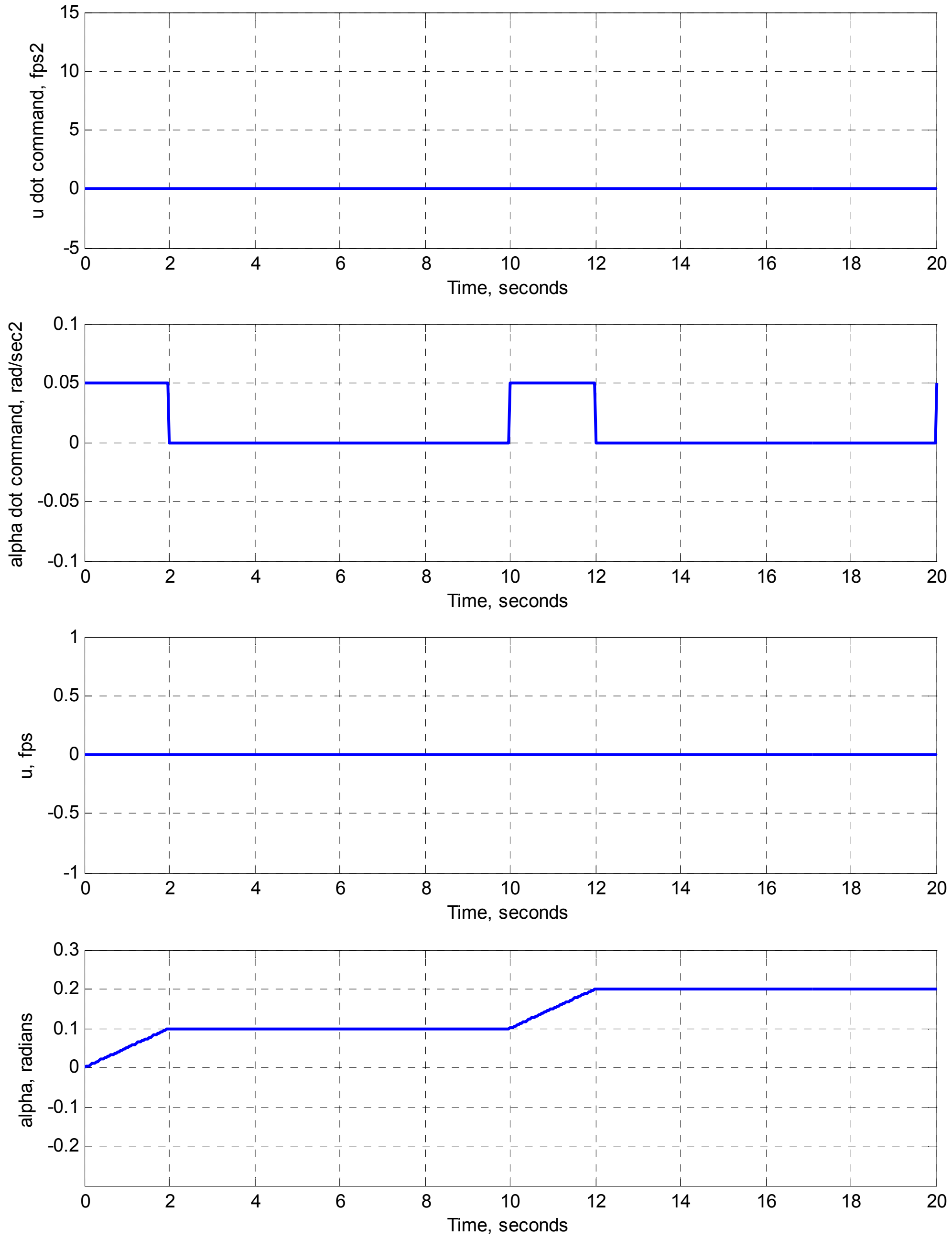


Figure 13: Time history results of integration of $\dot{\alpha}$ commands via $\mathbf{G}^{-1}(s)\mathbf{P}_A(s)$

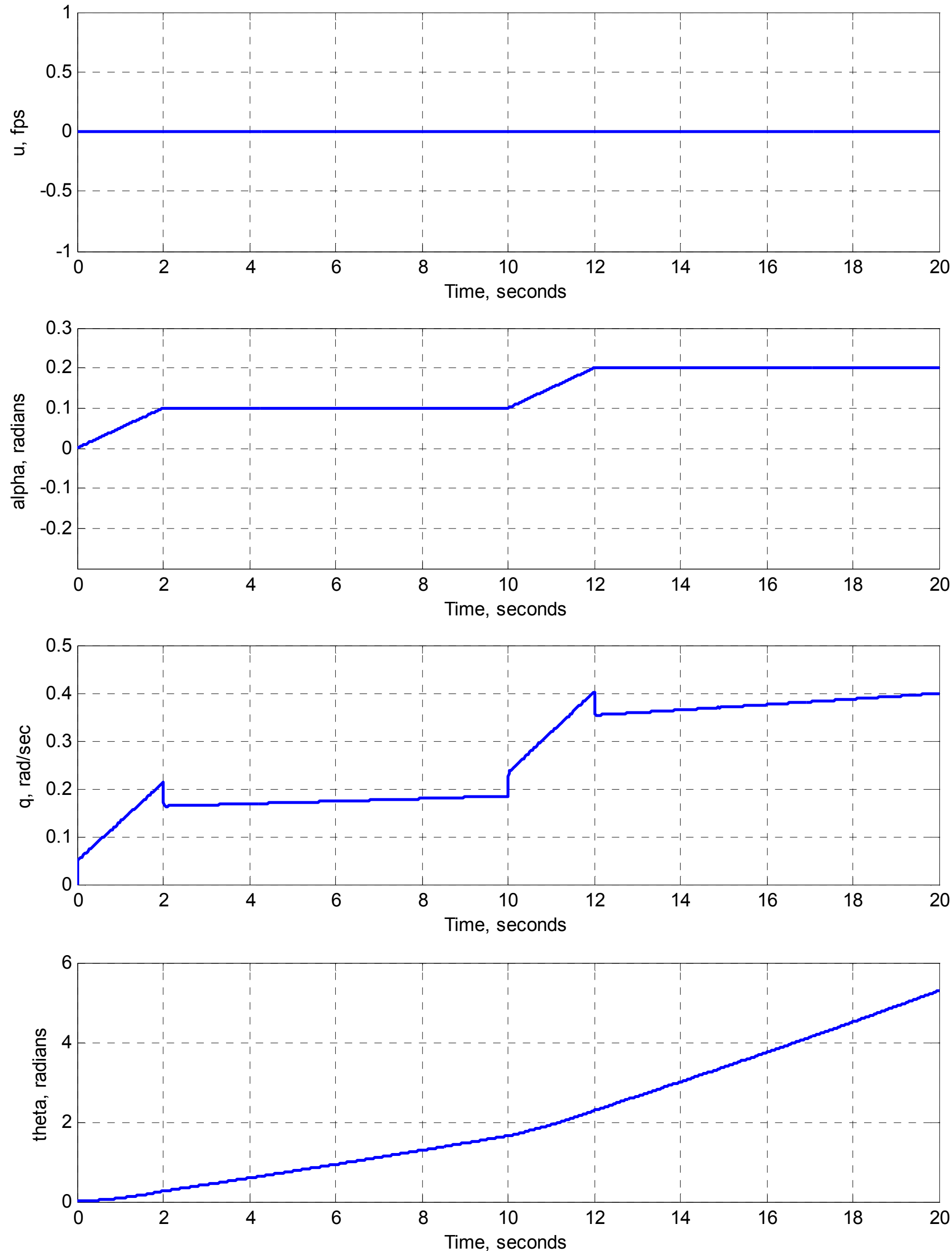


Figure 14: Time histories of state variables resulting from $\dot{\alpha}$ commands

2.4 Linear Prefilter

The linear prefilter computes the rates desired of the overall system to achieve the tracking characteristics of any arbitrary linear system. This prefilter performs a similar function to the command filter and handling qualities filter used by Calise and Rysdyk [6] and illustrated

in Figure 2. These desired rates from this prefilter will be shown to be an input into the inverse linear transfer function. In essence, the overall controller becomes a model-following controller, following the characteristics embedded in the linear prefilter. For this work, it is assumed that the system is to have the characteristics of a second-order system with respect to tracking speed and flight-path angle, as described in equations (2.20) and (2.21).

$$\ddot{V}_p + 2\varsigma_u \omega_{n_u} \dot{V}_p + \omega_{n_u}^2 V_p = \omega_{n_u}^2 V_{p_{com}} \tag{2.20}$$

$$\ddot{\gamma} + 2\varsigma_\gamma \omega_{n_\gamma} \dot{\gamma} + \omega_{n_\gamma}^2 \gamma = \omega_{n_\gamma}^2 \gamma_{com} \tag{2.21}$$

These damping ratios and natural frequencies are totally sufficient to describe the dynamic characteristics of the overall system and can be arbitrarily chosen in order to have the desired overall system characteristics.

A block diagram for a simple second-order system is shown in Figure 15. This will be used as a starting point for the development of both the speed and flight-path prefilters.

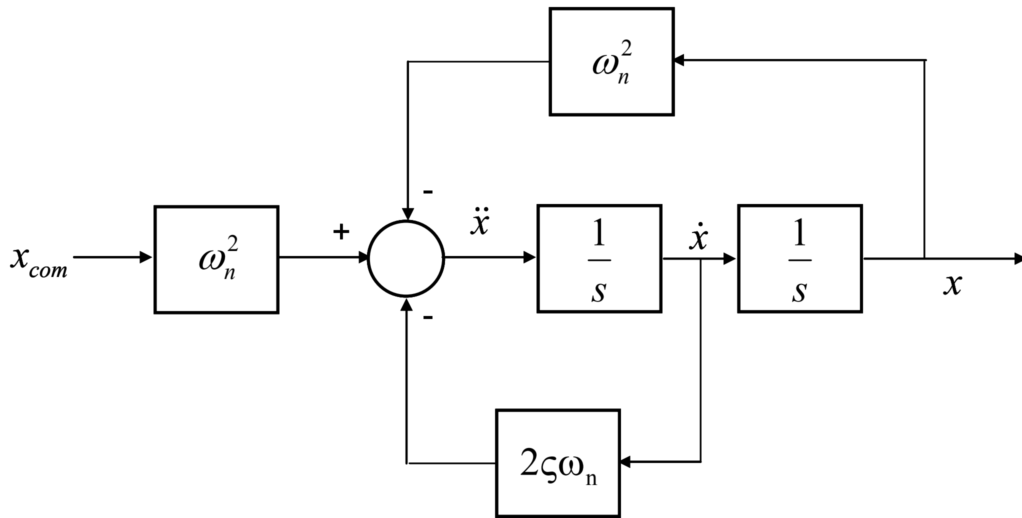


Figure 15: Simple second-order system

The speed prefilter is considered first. The block diagram in Figure 15 may be rearranged, but with no change in connectivity, as shown in Figure 16.

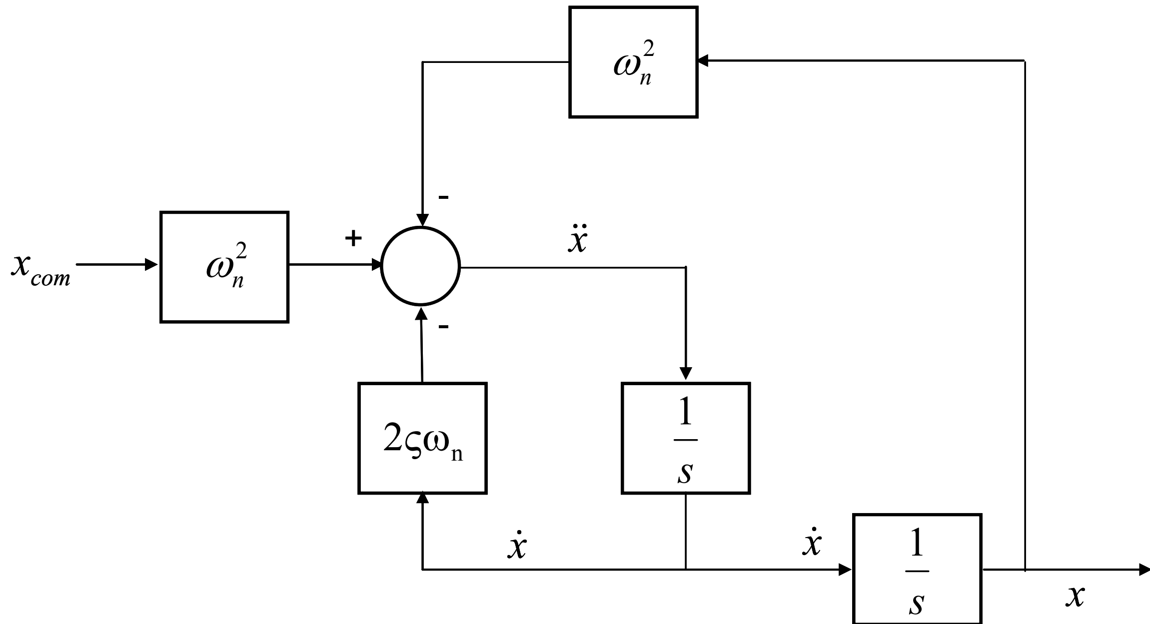


Figure 16: Rearrangement of second-order system, with no change in connectivity

Furthermore, this block diagram may be made specific to speed control by substituting u_b for x, introduce the variable, u_{b_m}, and invoke the equivalency of $\mathbf{G}^{-1}(s)\,\mathbf{P}_A(s)$ as an integrator. The results of this are shown in Figure 17.

Finally, the elements within this block diagram can be grouped by function, as shown in Figure 18. At a still higher level of abstraction, the block diagram for the speed prefilter can be viewed in Figure 19.

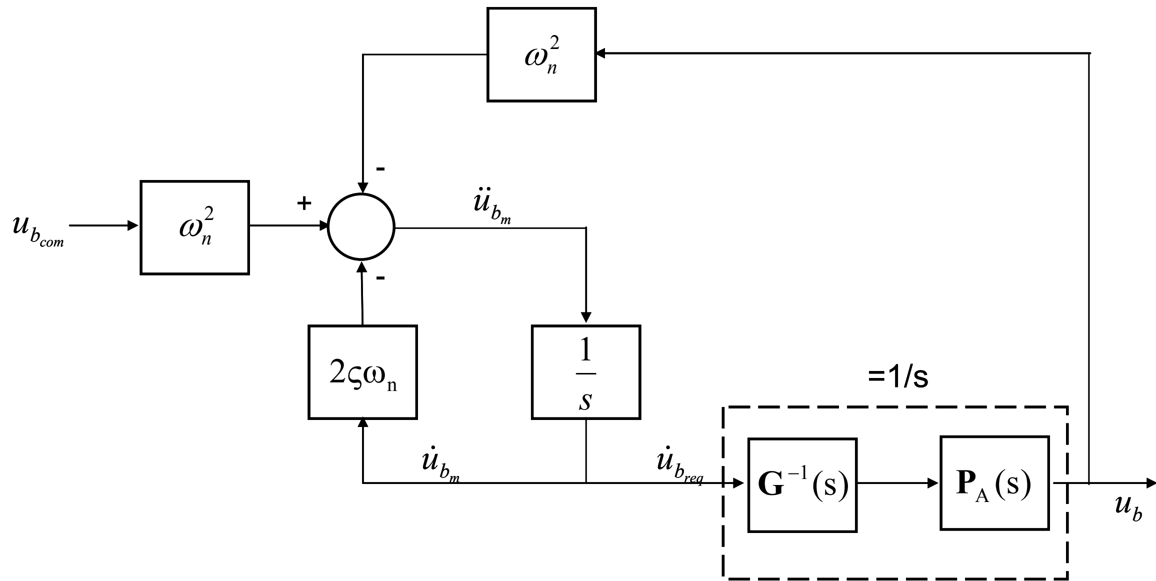


Figure 17: Customizing second-order system for speed control

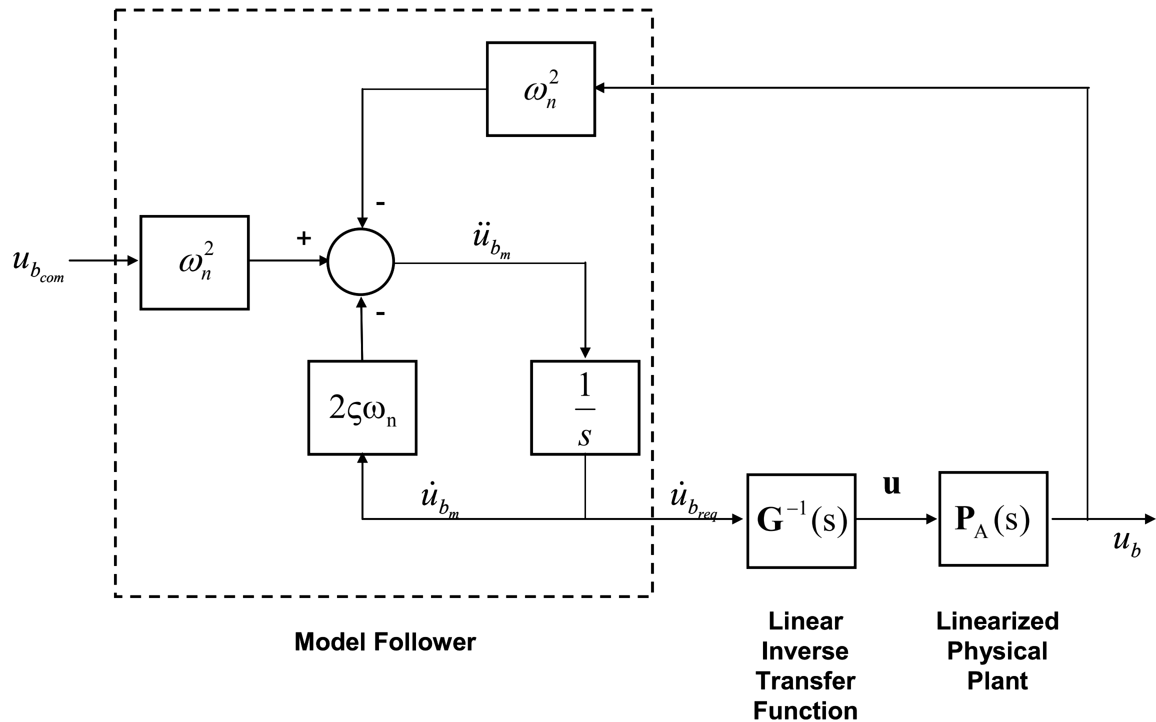


Figure 18: Functional grouping of elements within speed controller

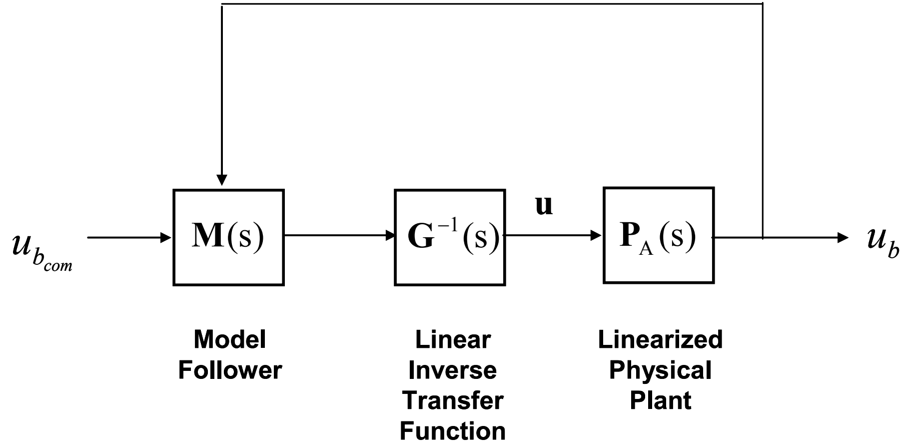


Figure 19: High-level arrangement of speed controller

The flight-path angle prefilter is more complex because flight-path angle is not controlled directly. Only $\dot{u}_b$ and $\dot{\alpha}$ are directly controlled by the inverse transfer function. To derive the control law for flight-path angle, it is necessary to first consider how flight-path angle is changed. If a pilot were flying the airplane, he would "pull g's." That is to say, the pilot would increase the angle of attack in order to increase the normal load factor, n_z, and thereby produce an imbalance in forces normal to the flight-path. This force imbalance causes an acceleration normal to the flight-path, thereby changing the flight-path angle. Mathematically,

$$mV_p\dot{\gamma} = \sum F_{\substack{\text{normal} \\ \text{to Vp}}} \tag{2.22}$$

$$mV_p\dot{\gamma} = \bar{q}SC_L - W\cos\gamma \tag{2.23}$$

Taking the derivative of both sides with respect to time yields:

$$\underline{m\dot{V}_p\dot{\gamma}} + mV_p\ddot{\gamma} = \dot{\bar{q}}SC_L + \bar{q}S\dot{C}_L + \underline{W\sin\gamma\dot{\gamma}} \tag{2.24}$$

If an order-of-magnitude analysis is performed on equation (2.24), with the flight conditions likely for the example aircraft, it can be shown that:

$m\dot{V}_p\dot{\gamma}$ is on the order of 10^0 lbs/sec.

$mV_p\ddot{\gamma}$ is on the order of 10^4 lbs/sec.

$\dot{\bar{q}}SC_L$ is on the order of 10^2 lbs/sec.

$\bar{q}S\dot{C}_L$ is on the order of 10^4 lbs/sec.

$W\sin\gamma\,\dot{\gamma}$ is zero, if the equation is evaluated at $\gamma = 0$.

The underlined terms in equation (2.24) are thus considered to be "small," and can be dismissed.

The time derivative of dynamic pressure can be decomposed into

$$\dot{\bar{q}} = \frac{d}{dt}\left(\frac{1}{2}\rho V_p^2\right) \tag{2.25}$$

$$\dot{\bar{q}} = \rho V_p \dot{V}_p \tag{2.26}$$

Making the simplification that the lift coefficient is primarily a function of angle of attack, the time derivative of lift coefficient can be similarly decomposed as

$$\dot{C}_L \approx \frac{d}{dt}\left(C_{L_0} + C_{L_\alpha}\alpha\right) \tag{2.27}$$

$$\dot{C}_L = C_{L_\alpha}\dot{\alpha} \tag{2.28}$$

Substituting equations (2.26) and (2.28) into equation (2.24) and dropping the underlined terms, results in

$$mV_p\ddot{\gamma} = \rho V_p \dot{V}_p S C_L + \bar{q}SC_{L_\alpha}\dot{\alpha} \tag{2.29}$$

Rearranging, this becomes

$$\bar{q}SC_{L_\alpha}\dot{\alpha} = mV_p\ddot{\gamma} - \rho V_p \dot{V}_p S C_L \tag{2.30}$$

$$\bar{q}SC_{L_\alpha}\dot{\alpha} = mV_p\ddot{\gamma} - 2\left(\frac{\rho V_p^2 SC_L}{2V_p}\right)\dot{V}_p \tag{2.31}$$

$$\bar{q}SC_{L_\alpha}\dot{\alpha} = mV_p\ddot{\gamma} + \left(\frac{-2n_zW}{V_p}\right)\dot{V}_p \tag{2.32}$$

$$\dot{\alpha} = \left(\frac{mV_p}{\bar{q}SC_{L_\alpha}}\right)\ddot{\gamma} + \left(\frac{-2n_zW}{\bar{q}SC_{L_\alpha}V_p}\right)\dot{V}_p \tag{2.33}$$

$$\dot{\alpha} = \Lambda_1\ddot{\gamma} + \Lambda_2\dot{V}_p \tag{2.34}$$

where

$$\Lambda_1 \equiv \frac{mV_p}{\bar{q}SC_{L_\alpha}} \tag{2.35}$$

$$\Lambda_2 \equiv \frac{-2n_zW}{\bar{q}SC_{L_\alpha}V_p} \tag{2.36}$$

Equation (2.34) provides the necessary link connecting angle of attack (which can be controlled) with flight-path angle (the desired variable to track). Recall that the objective is to have $\ddot{\gamma} = -2\varsigma_\gamma\omega_{n_\gamma}\dot{\gamma} - \omega_{n_\gamma}^2\gamma + \omega_{n_\gamma}^2\gamma_{com}$, as noted in equation (2.21). Making this substitution into equation (2.34)

$$\dot{\alpha} = \Lambda_1\left(-2\varsigma_\gamma\omega_{n_\gamma}\dot{\gamma} - \omega_{n_\gamma}^2\gamma + \omega_{n_\gamma}^2\gamma_{com}\right) + \Lambda_2\dot{V}_p \tag{2.37}$$

$$\dot{\alpha} = \Lambda_2\dot{V}_p - 2\Lambda_1\varsigma_\gamma\omega_{n_\gamma}\dot{\gamma} - \Lambda_1\omega_{n_\gamma}^2\gamma + \Lambda_1\omega_{n_\gamma}^2\gamma_{com} \tag{2.38}$$

In the event it is desired to match the system response to some other linear system response, such as a first-order system, a different but appropriate substitution could be made into equation (2.34).

Equation (2.38) now forms the basis of the flight-path angle control law. Observe that the last two terms are essentially an error term, suggesting that $\dot{\alpha}$ follows the error in flight-path angle, with all other parameters held constant.

A problematic aspect of equation (2.38) is that it involves the derivative of the flight-path angle, $\dot{\gamma}$. This derivative is not available as an output from the physical plant. Taking the derivative of the flight-path angle directly is not good practice, since signal noise may make the derivative unreliable. To overcome this, a second-order filter is applied to the flight-path angle to produce a filtered flight-path angle, γ_f, and its derivative, $\dot{\gamma}_f$. This second-order filter is implemented using an LTI state space model of the form

$$\mathbf{F}(s) = \left[\begin{array}{c|c} \mathbf{A}_F & \mathbf{B}_F \\ \hline \mathbf{C}_F & \mathbf{D}_F \end{array}\right] \tag{2.39}$$

where

$$\mathbf{A} = \begin{bmatrix} -2\varsigma_f \omega_{n_f} & -\omega_{n_f}^2 \\ 1 & 0 \end{bmatrix} \tag{2.40}$$

$$\mathbf{B} = \begin{bmatrix} \omega_{n_f}^2 \\ 0 \end{bmatrix} \tag{2.41}$$

$$\mathbf{C} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{2.42}$$

$$\mathbf{D} = \mathbf{0} \tag{2.43}$$

$$\mathbf{u} = \gamma \tag{2.44}$$

$$\mathbf{y} = \begin{bmatrix} \dot{\gamma}_f \\ \gamma_f \end{bmatrix} \tag{2.45}$$

Combining the gamma filter and the control law, as described in equations (2.29) through (2.45), completes the flight-path prefilter. A block diagram of the prefilter is illustrated in Figures 20 and 21.

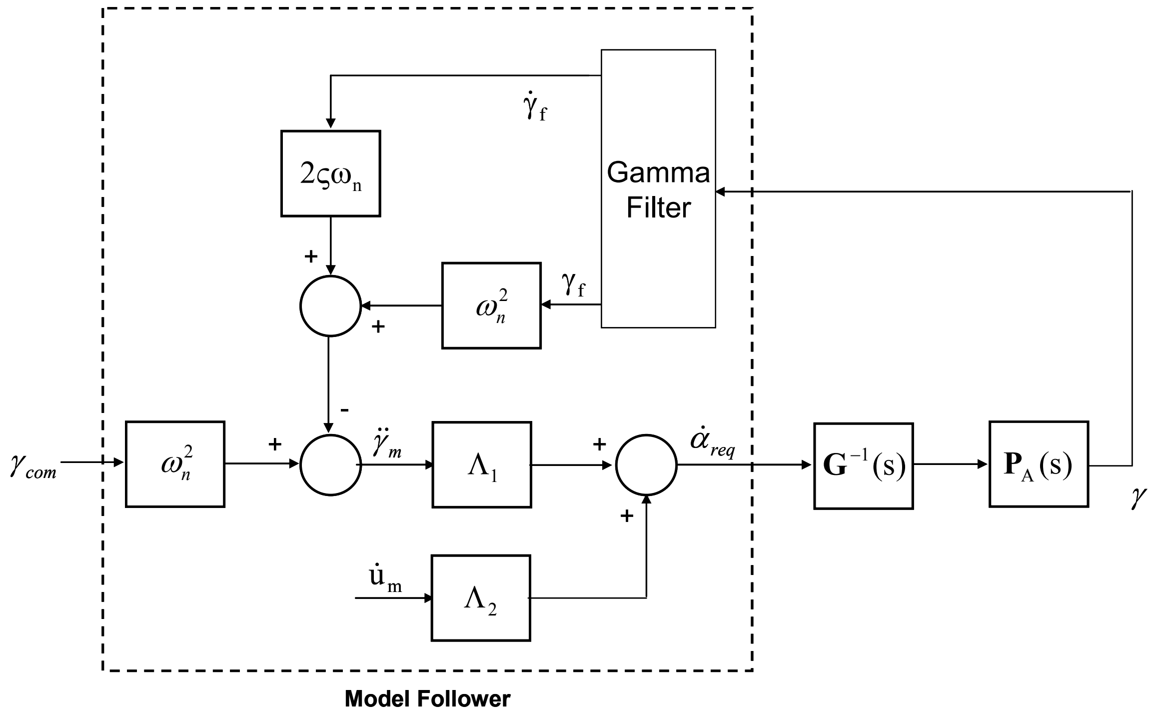


Figure 20: Block diagram of flight-path angle controller

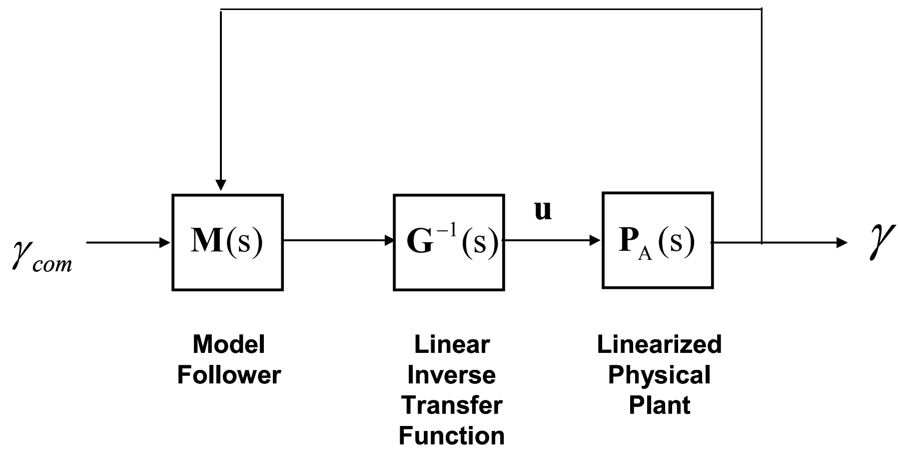


Figure 21: High-level arrangement of flight-path angle controller

The speed and flight-path channels of the linear prefilter can now be assembled into a single state space representation

$$\mathbf{M}(s) = \left[\begin{array}{c|c} \mathbf{A}_M & \mathbf{B}_M \\ \hline \mathbf{C}_M & \mathbf{D}_M \end{array}\right] \tag{2.46}$$

where

$$\mathbf{A}_M = \begin{bmatrix} -2\varsigma_u\omega_{n_u} & -\omega_{n_u}^2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -2\varsigma_f\omega_{n_f} & -\omega_{n_f}^2 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.47}$$

$$\mathbf{B}_M = \begin{bmatrix} \omega_{n_u}^2 & 0 & -\omega_{n_u}^2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \omega_{n_f}^2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.48}$$

$$\mathbf{C}_M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \Lambda_2 & 0 & -2\varsigma_\gamma\omega_{n_\gamma}\Lambda_1 & -\omega_{n_\lambda}^2\Lambda_1 \end{bmatrix} \tag{2.49}$$

$$\mathbf{D}_M = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \Lambda_1\omega_{n_\gamma}^2 & 0 & 0 \end{bmatrix} \tag{2.50}$$

$$\mathbf{u} = \begin{bmatrix} V_{p_{com}} \\ \gamma_{com} \\ u \\ \gamma \end{bmatrix} \tag{2.51}$$

$$\mathbf{y}_M = \begin{bmatrix} \dot{u}_{b_{req}} \\ \dot{\gamma}_{req} \end{bmatrix} \tag{2.52}$$

Not withstanding any modeling errors or inversion errors, a controller could be assembled from the elements already described. The controller would take the form as shown in Figure 22. Commands from the pilot, plus feedback from the physical plant are received by the linear model prefilter. The prefilter calculates the rates required of the system in order to drive the tracking error to zero in a fashion consistent with the second-order system characteristics

imbedded in the prefilter. The linear inverse transfer function computes control inputs to the physical plant necessary to achieve the desired rates. The physical plant responds accordingly.

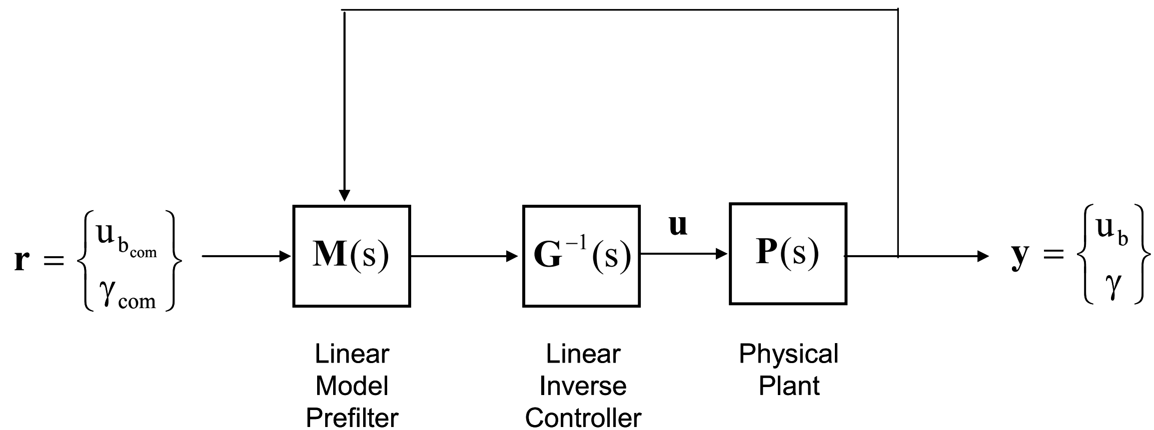
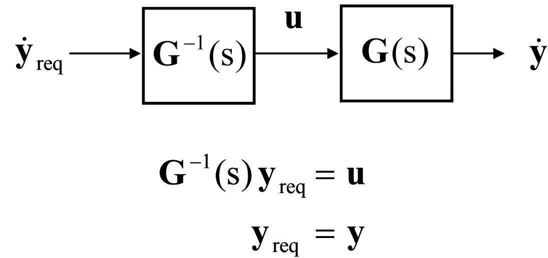


Figure 22: A linear inverse controller with model following, but no adaptation

There are two significant shortcomings in the system described in Figure 22. First, the inverse transfer function is based on a linear approximation of the physical plant. The degree of nonlinearity present in the real plant will influence the precision of tracking. Second, the controller is not adaptive to changes in the physical plant during operation. Even if the physical plant was linear, unforeseen changes may occur within the plant (wear, failures, or new operating modes) that cause the physical plant to operate differently than the characteristics assumed when creating the inverse transfer function. Both of these issues are satisfactorily addressed by the addition of the third element into the controller: the adaptive neural network compensator.

2.5 Adaptive Neural Network Compensator

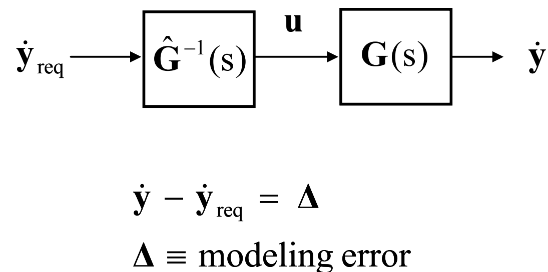
The existence of an inverse transfer function was discussed in subchapter 2.3. Under certain conditions, a linear inverse transfer function exists such that $\mathbf{G}^{-1}(s)\mathbf{G}(s) = \mathbf{I}$. A block diagram illustrating this is shown in Figure 23.



$$\mathbf{G}^{-1}(s)\mathbf{y}_{req} = \mathbf{u}$$
$$\mathbf{y}_{req} = \mathbf{y}$$

Figure 23: Function of an inverse transfer function

In practice, an exact inverse may not be realizable. This may be the result of approximating what is really a nonlinear function, or perhaps there has been some change in behavior of the physical plant. In cases such as this, the best that can be achieved is an approximate inverse function, $\widehat{\mathbf{G}}^{-1}(s)$, illustrated in Figure 24. Note that there is some modeling error, denoted as Δ.



$$\dot{\mathbf{y}} - \dot{\mathbf{y}}_{req} = \mathbf{\Delta}$$
$$\mathbf{\Delta} \equiv \text{modeling error}$$

Figure 24: An approximate inverse transfer function with modeling error

In a discretized computation environment, the definition for Δ is

$$\mathbf{\Delta}(k) = \dot{\mathbf{y}}(k+1) - \dot{\mathbf{y}}_{req}(k), \text{ or} \tag{2.53}$$

$$\dot{\mathbf{y}}(k+1) - \mathbf{\Delta}(k) = \dot{\mathbf{y}}_{req}(k) \tag{2.54}$$

The intent of the controller is to produce a result $\dot{\mathbf{y}}(k+1)$ that matches the command of $\dot{\mathbf{y}}_{com}(k)$, such that

$$\dot{\mathbf{y}}_{com}(k) = \dot{\mathbf{y}}(k+1) \tag{2.55}$$

After substitution of equation (2.55) into equation (2.54), the result is

$$\dot{\mathbf{y}}_{com}(k) - \mathbf{\Delta}(k) = \dot{\mathbf{y}}_{req}(k) \tag{2.56}$$

Figure 25 depicts this in block diagram form. It must be noted that $\mathbf{\Delta}(k)$ anticipates the modeling error and adjusts the commanded rate appropriately.

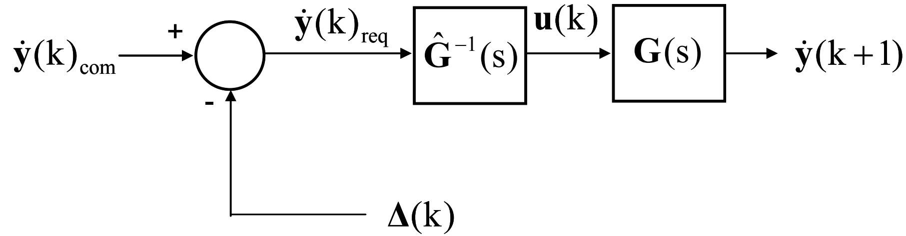


Figure 25: Incorporation of anticipated modeling error into controller

A neural network can be used to compute an estimate of $\mathbf{\Delta}(k)$ based on past observations of the modeling error. This estimate is referred to as $\hat{\mathbf{\Delta}}(k)$. The difference between $\dot{y}_{req}(k)$ and $\dot{y}(k+1)$, the actual rate realized at one computation step ahead, is computed and fed into the network to be learned. This $\mathbf{\Delta}(k)$ is learned as a function of $\dot{y}_{req}(k)$ and a variety of other training data. When used to estimate $\hat{\mathbf{\Delta}}(k)$, the input to the network is $\dot{y}_{com}(k)$ plus these same

other data, as shown in Figure 26. These other data are selected to best represent the dynamics of the modeling error. Selection of these other data is discussed in subchapter 3.3.

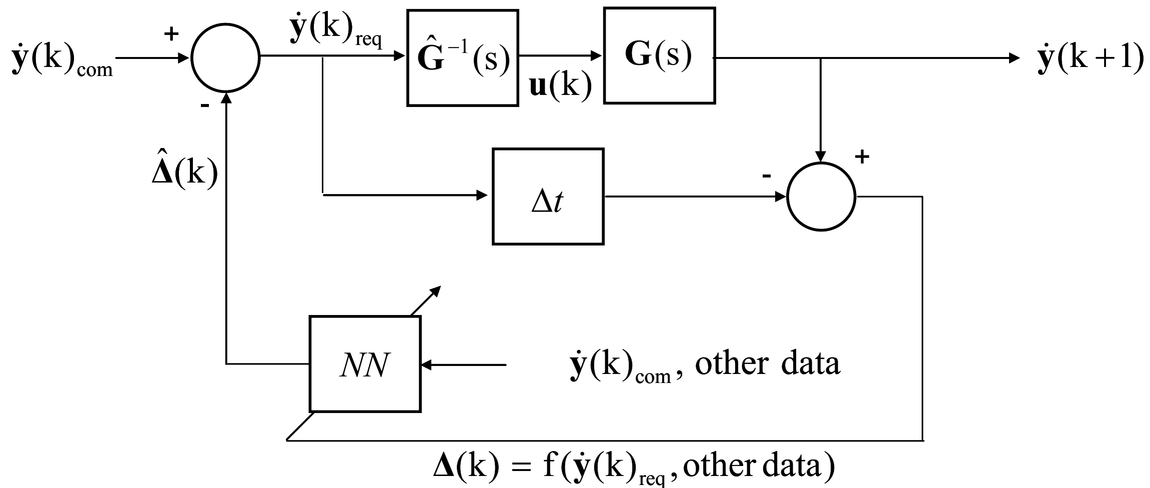


Figure 26: Incorporation of adaptive neural network to approximate modeling error

This entire structure can be viewed as an integrator, similar to the linear inverse transfer function in parallel with the physical plant. Commanded rates come into the system, and their integrals are produced as outputs. See Figure 27.

Online learning allows the neural network to adapt to new operating characteristics in the physical plant. Online learning can be accomplished by the network, whether or not it is actively engaged for compensation. The controller can operate without the benefit of the adaptive compensation, with some degradation in performance, for cases where the physical plant is closely matched to the inverse transfer function. Such operation is demonstrated in subchapter 4.3, example Case 7. While not typical, operation without compensation provides a mechanism for the network to be trained before being put into operation. A preferred method for initial training, however, is to pre-train the network offline to learn the nominal characteristics of the physical plant.

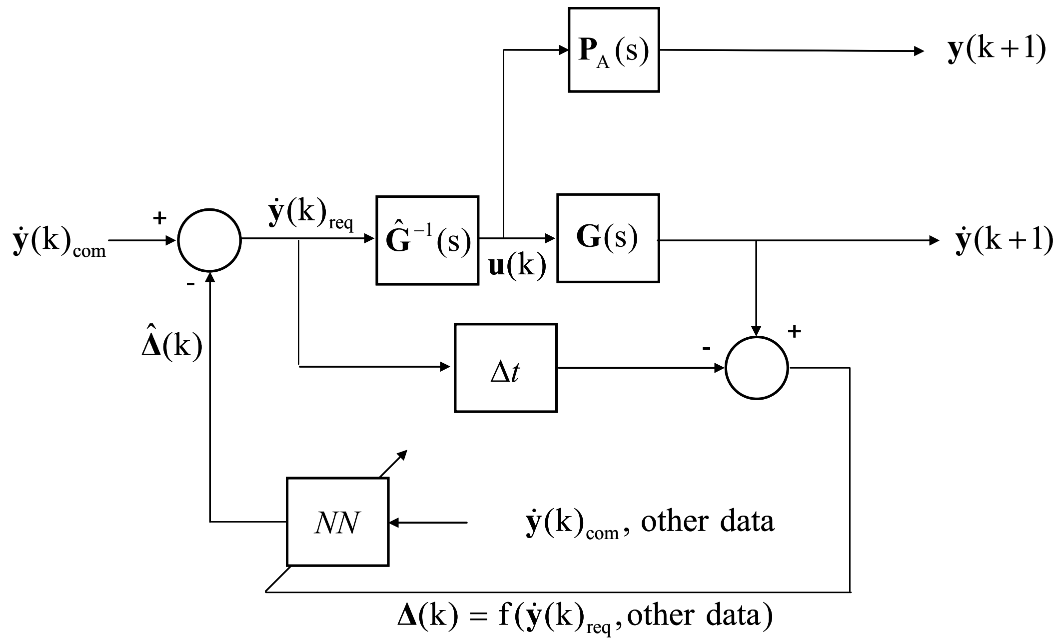


Figure 27: Addition of neural network preserves integration function of controller

At this point, the theory of all major elements of the controller has been discussed. The controller architecture is summarized once again in Figure 28. Now it is time to discuss the mechanics of implementing the controller.

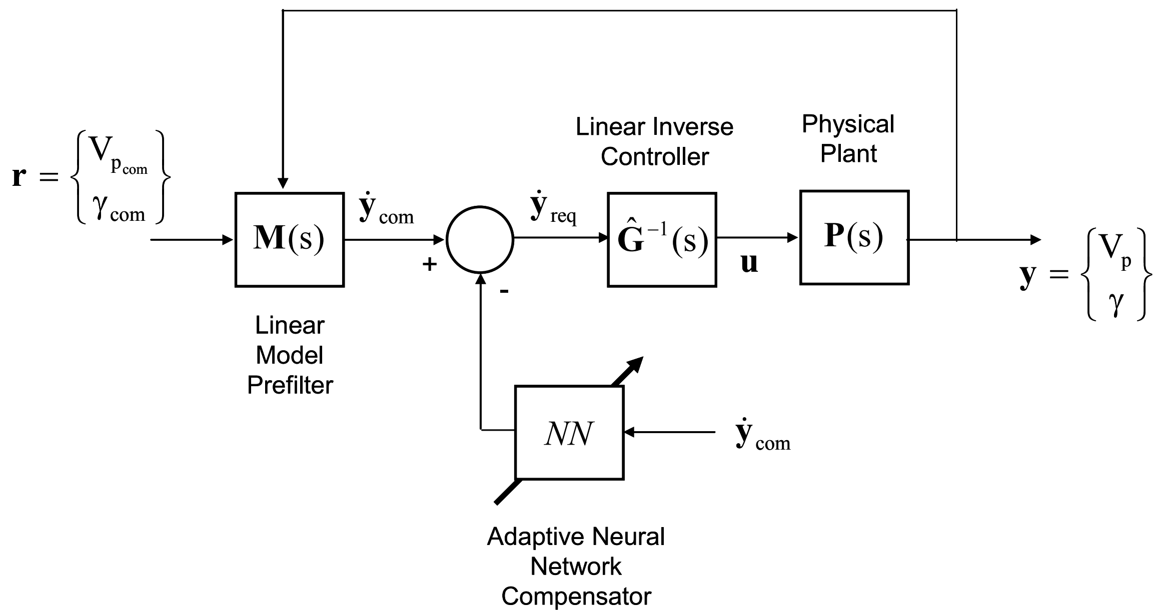


Figure 28: Summary of controller architecture

CHAPTER 3

IMPLEMENTATION

3.1 Bonanza Sample Aircraft

The Beechcraft Bonanza is a low-wing piston-powered monoplane typical of high-performance General Aviation aircraft. Wichita State University and Hawker Beechcraft Corporation have collaborated in flight research on advanced flight controls using a specially equipped Bonanza, serial number CJ-144. This same aircraft has been used successfully in NASA-sponsored research as part of the AGATE and SATS programs. CJ-144 has been modified to have a fly-by-wire control system on the pilot's side of the aircraft, while retaining traditional mechanical flight controls on the copilot's side. This architecture allows a subject pilot to evaluate flight control algorithms on the left side, while a safety pilot is standing by on the right side to take control in the event of a problem. Illustrations of CJ-144 and its cockpit arrangement are shown in Figures 29 and 30.

The pilot's controls in CJ-144 consist of a "sidestick" controller and a speed-command lever. In the CJ-144 configuration, the sidestick controller uses fore-and-aft motion to command the vertical flight-path angle, while side-to-side motion controls bank angle. The sidestick is spring-loaded to center, which in turn, commands straight and level flight. The speed command lever is simply an input device to command speed. Baseline CJ-144 control laws command elevator deflection and engine power as necessary to maintain the commanded flight-path angle (via the sidestick) and airspeed (via the speed-command lever). For this research, a similar control scheme is assumed. The pilot controls flight-path angle and airspeed as two independent inputs. Controlling the lateral-directional modes of the aircraft is outside the scope of this effort.

Figure 29: In-flight photo of Hawker Beechcraft Corporation's fly-by-wire Bonanza, CJ-144 (courtesy of Hawker Beechcraft Corporation)

Figure 30: Cockpit of Hawker Beechcraft Corporation's fly-by-wire Bonanza, CJ-144 (courtesy of Hawker Beechcraft Corporation)

The aerodynamic coefficients and mass properties of the aircraft are provided to Wichita State University for academic use only under license by Hawker Beechcraft Corporation. The data are still considered proprietary data by Hawker Beechcraft Corporation and will not be disclosed in this dissertation.

It should be noted that no flight test evaluation of this controller has taken place on the Bonanza. The purpose of the airplane is simply to serve as a "real-world" candidate platform for simulated evaluation of this controller.

3.2 Aircraft Simulation

The aircraft is simulated in a six degree of freedom simulation written in ANSI C. Even though the simulation is capable of running all six degrees of freedom, only the longitudinal modes are utilized for this effort. The equations of motion are largely taken from Roskam [18] and are described in detail in Appendix B. The simulation uses a fourth-order Adams predictor-corrector algorithm for solving the differential equations of motion. The dynamics of the simulation were validated by inserting the data for Roskam's sample "typical business jet." Program function "force_Roskam" inserts this sample airplane data into the simulation via hard code. Results from the simulation were compared with the results shown in Roskam's Table 5.4 and found to be in agreement. The static and trim solutions have been checked by hand. The simulation runs at 50 computation cycles per second of simulated time. On a PC with an Intel 3.2 GHz Pentium 4 processor, the simulation runs approximately 1000 times faster than real time.

Some key features of the code are as follows:

1. Batch mode operation for stacking of multiple cases.
2. Online training of the adaptive neural network.

3. Creation of training data files for offline training, if desired.
4. Creation of output files for plotting and analysis of simulation results.

A complete source code listing, except for proprietary Hawker Beechcraft Corporation data, can be found in Appendix C.

3.3 Neural Network Configuration

The neural network used is a feed forward architecture using backpropagation for learning. The theory of neural networks is described in detail in Fausett's textbook [19]. The value of neural networks in this problem is their ability to approximate any mathematical function, their ability to model a phenomenon with no *a priori* knowledge of the physics, and their ability to learn in real time. All of these are essential attributes when the controller is tasked with responding quickly to an unspecified failure that may be unknown or not identified.

Defining the exact topology of the network involves some educated guesswork, along with some trial and error. The baseline inspiration for the topology is the state space model for the inverse transfer function. The reasoning is that if the intent of the neural network is to capture the modeling error, the modeling error should, at a minimum, be somewhat similar in functional dependence to the linear inverse transfer function.

The inverse transfer function is described as a state space model. While typically computed using matrices, this same state space model can be implemented exactly as a neural network, with one hidden layer and linear activation functions on each neuron. The weights between the fully connected input and hidden layers represent the coefficients of matrices **A** and **B**. Weights between the fully connected hidden and output layers represent the coefficients of matrices **C** and **D**. This is illustrated in Figure 31. This reasoning immediately suggests that a three-layer network will be required for this application, with a matching number of neurons on

the first and second layers. The output layer will, of course, have two output neurons: one for the modeling error on $\dot{V}_p$ and one for the modeling error on $\dot{\alpha}$.

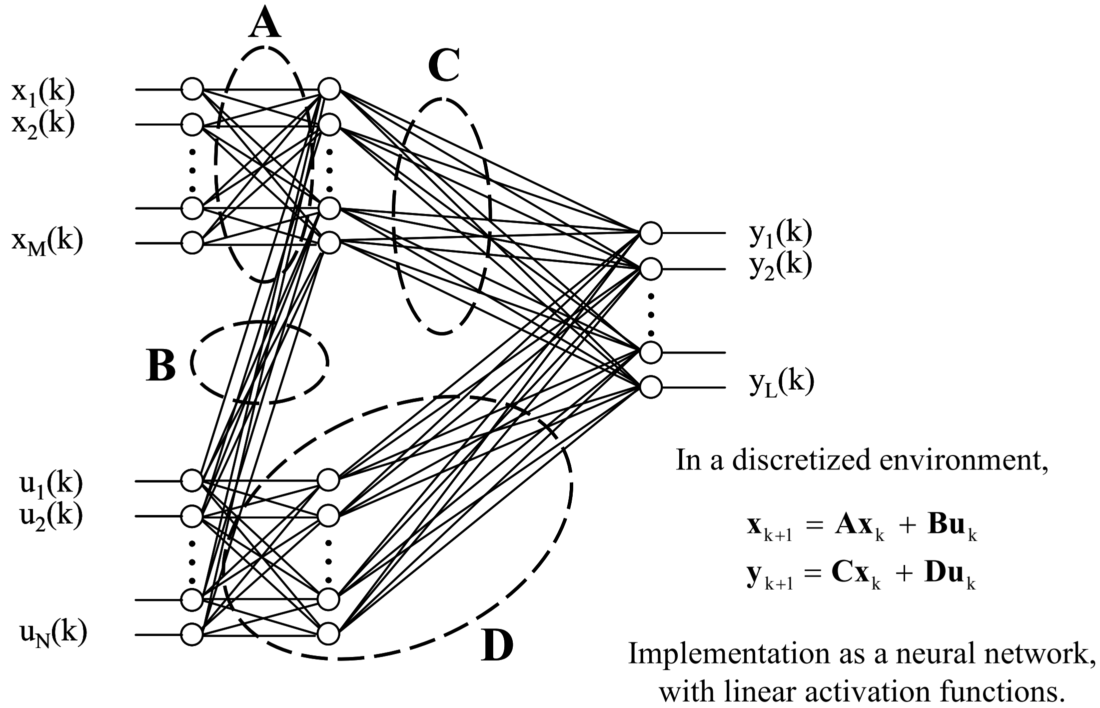


Figure 31: Equivalency of neural network and state space matrix

Anticipating nonlinearities in the modeling error, nonlinear activation functions are assumed for all neurons. A hyperbolic tangent (tanh) function is used.

Furthermore, it is anticipated that the modeling error will be a function of angle of attack (such as nonlinear stall characteristics) and airspeed (such as dynamic pressure, Reynolds number, or Mach effects). In fact, it can be hypothesized that, in general, modeling error could be a function of any of the aircraft state variables: V_p, α, q, or θ. Consequently, these state variables should be included as inputs to the network.

It is anticipated that the modeling error will involve dynamics of the aircraft states. A recursive neural network, as employed by Hess [15], is utilized to provide a mechanism to

capture these dynamics. Recursion is accomplished externally to maintain a feed-forward network structure. It cannot be known *a priori* what the modeling error may be caused by, particularly in the case of system failures within the physical plant. These errors may involve first and second derivatives of the state variables or control inputs. Consequently, not only must the current values of the state variables and inputs be included, but the proceeding two time steps must be included. The network will have current values, plus first and second derivatives by means of finite differences, available for its computations. These could be important for modeling error resulting from hysteresis, computational delays, or backlash.

Trim speed and trim altitude are also likely candidates for inputs to the network, since the aerodynamic characteristics that are the building blocks for the inverse transfer function are dependent on these data.

With these concepts in mind, an iterative process begins to pre-train the neural network. Pre-training the network is analogous to teaching a human to fly. Pre-training instills initial intelligence into the controller. When humans fly an airplane, they generally take control prepared with previously learned flying skills. It would be unreasonable to expect a human to cope with complex failure scenarios in an airplane, and manage well, with no prior experience. So it is true with the adaptive inverse controller. It is unreasonable to expect good performance without pre-training. During the pre-training process, flying experiences are repetitively presented to the neural network, and over time, the network learns. This is not unlike the repetitive process humans go through in learning to fly.

A companion program was written to pre-train the network “offline.” Offline implies post-processing the results of the airplane simulation. The flight dynamic modules are absent in the offline training program, and are replaced with file I/O routines to handle large training data

files. All neural network computational modules are the same in both the flight simulation and offline training programs. This offline process facilitates the iterative process of pre-training the network. During this pre-training process, the network architecture evolves. The ability to deal with unanticipated scenarios drives to a larger network configuration than would be required for a small bounded set of failure scenarios. As will be shown later, the final architecture proved to be robust and responsive to all nonlinearities and failures examined.

The pre-training process is shown graphically in Figure 32. The process follows this sequence:

1. The aircraft simulation is exercised over a range of speed and altitudes that cover the aircraft flight envelope. Training maneuvers, consisting of random commands for speed and flight-path, are flown by the simulation. The airplane is in its nominal no-failure-state condition with no neural network compensation. The necessary data for training are recorded in a data file. There are 11,550 training pairs in the final training data set.
2. Recorded data is divided into a training data set and test data set. The training data set is used for offline training of the network. The trained network is then exercised offline with the test data to see if the network can replicate the test outputs. The goal for training is to produce a network that works equally well on both the training data set and the test data set.
3. Candidate architectures are implemented into the simulation. The offline training program creates C code that can be "cut and paste" into the simulation code. This coding contains all the network parameters necessary to fully describe the network; such as the network topology, the value of all network weights, and all network input and output scaling factors.

4. A series of validation maneuvers are flown with the simulation to evaluate the performance of the controller with the adaptive neural network compensator turned on. Unlike the random training maneuvers, these are a series of well-defined piloting tasks similar to the sample cases to be described in Chapter 4.
5. The training iteration is continued until the results of the validation maneuvers are satisfactory. If the results are not satisfactory, adjustments are made to either the network architecture or to the training experience, and then the process is repeated.

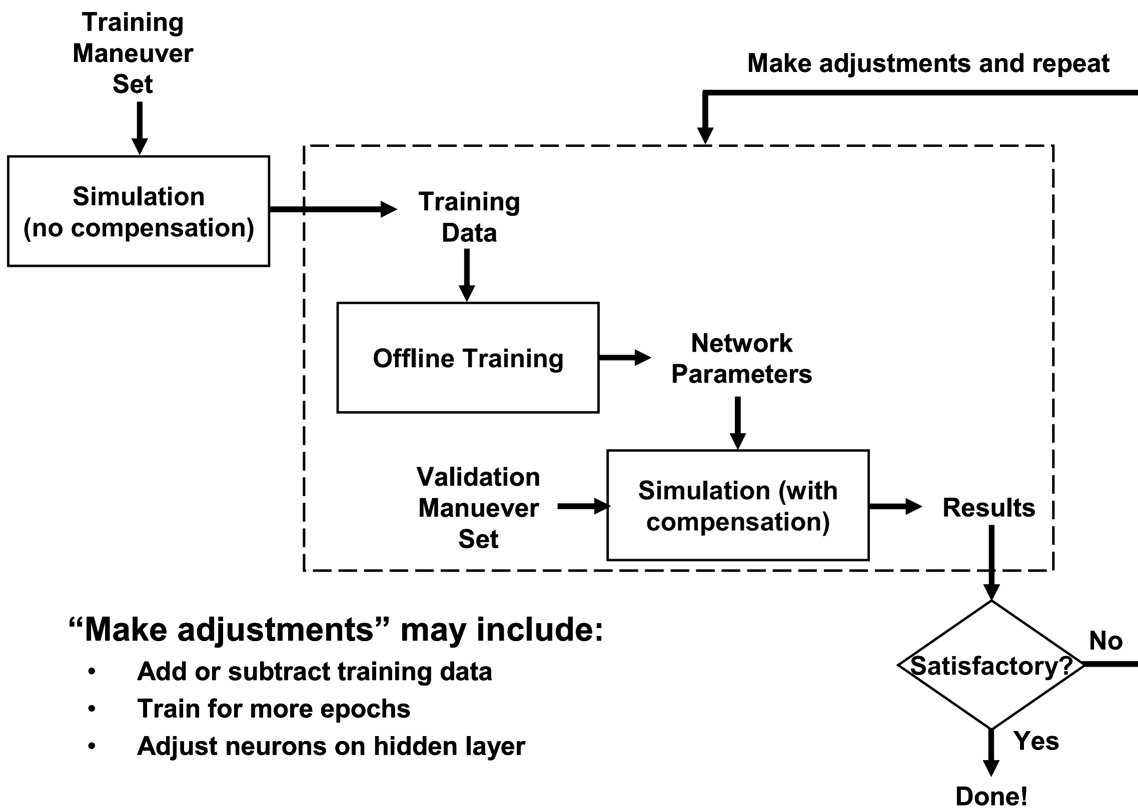


Figure 32: Neural network training process

Figure 33 characterizes the training data before the training process begins. The purpose of the training is to find some correlation between $\Delta\dot{u}_b$ and $\Delta\dot{\alpha}$ with the required values of

$\dot{u}_b$ and $\dot{\alpha}$, respectively, plus whatever other data may be required to achieve a correlation. The magnitude of this task is evident. There is a weak correlation evident between $\Delta\dot{u}_b$ and the required $\dot{u}_b$, but considerable scatter remains. Other data must be involved to explain this scatter. On the other hand, there is no correlation evident between $\Delta\dot{\alpha}$ and the required $\dot{\alpha}$. An absolutely random scattering of $\Delta\dot{\alpha}$ points is apparent. The challenge for the offline training is to find a correlation such that a good estimate for $\Delta(k)$ may be obtained from some combination of available data.

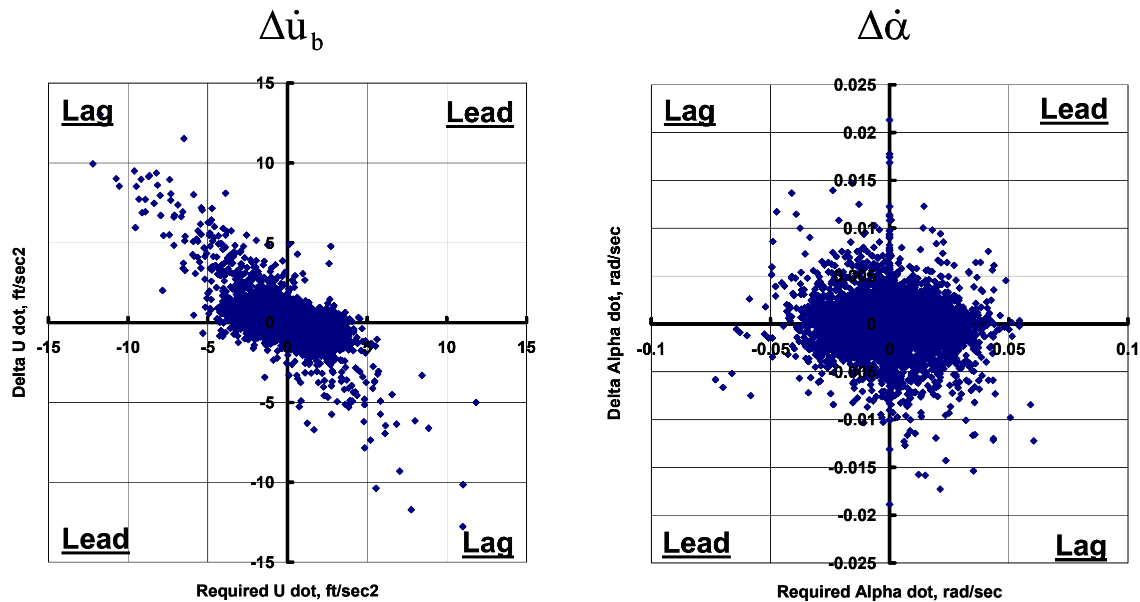


Figure 33: Data before training

Figures 34 and 35 show the final training results and the test results from the offline training. The known system outputs are plotted on the x-axis, and the estimated output by the network is plotted on the y-axis. Ideally, all points should lie along the "45 degree line of correlation." While not perfect, this fit is quite good. The final RMS error for all data points is

.030%. The Pearson product moment correlation coefficient (R^2) commonly used in Excel™ trend analysis is seen to be very close to one, also indicating a good correlation with the data.

The final architecture chosen is illustrated in Figure 36. There are 64 computational nodes in the network. The external recursion of the output can be seen.

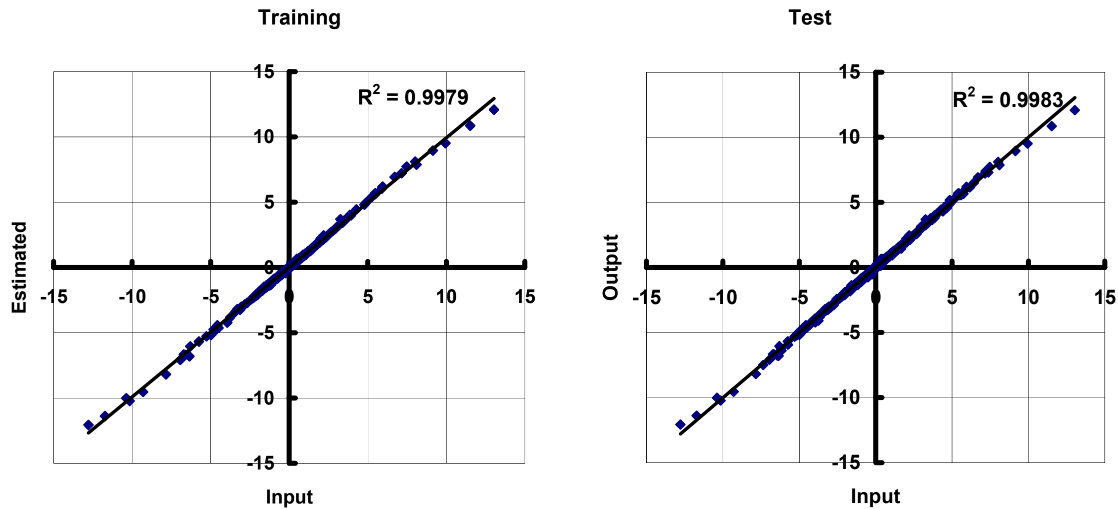


Figure 34: Results of training network on $\dot{V}_p$ modeling error

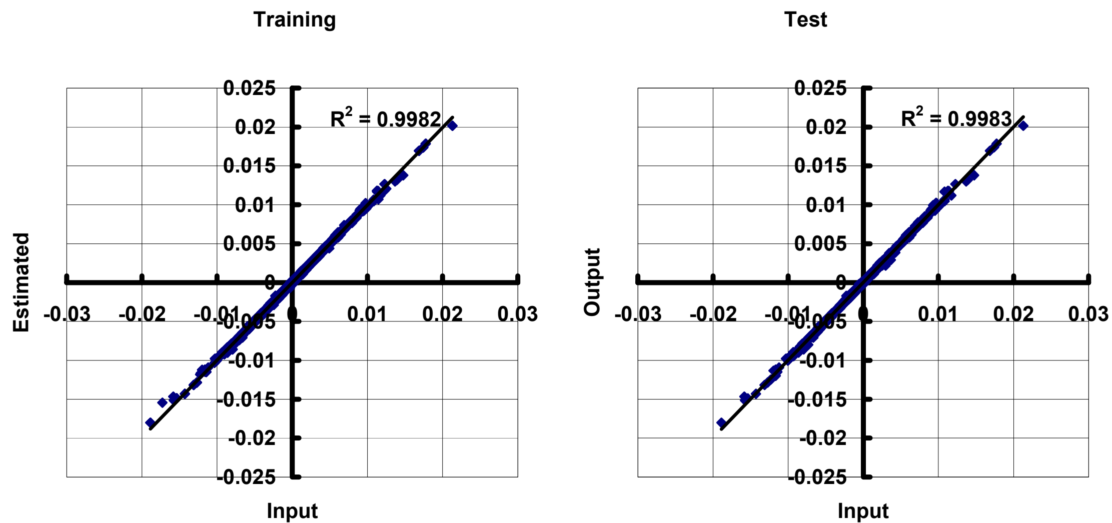


Figure 35: Results of training network on $\dot{\alpha}$ modeling error

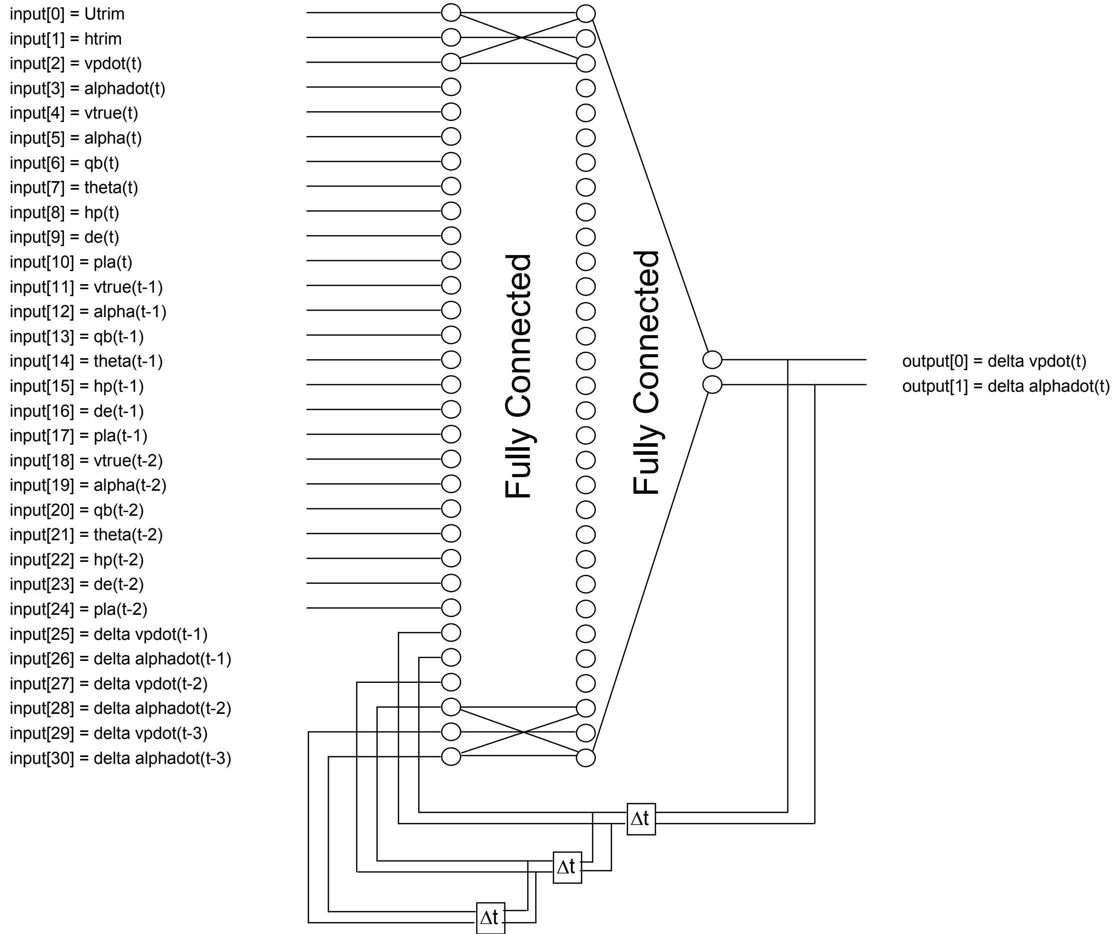


Figure 36: Topology of final network architecture

3.4 Controller Coding

The controller is coded in ANSI C for future compatibility with the flight computer in Hawker Beechcraft Corporation's Bonanza CJ-144. The code is modularized to facilitate rapid integration into CJ-144 if the opportunity becomes available.

The linear prefilter is implemented in function "linear_prefilter_l2." The code may be found in Appendix C. This function implements the state space model of the prefilter, as described in equations (2.46) through (2.52). All operations are implemented using linear algebra. The **A** and **B** matrices of the prefilter are static after initialization, since they are functions only of constant damping ratios and natural frequencies. The **C** and **D** matrices are sparse, and only the non-zero coefficients are explicitly coded. These non-zero coefficients are updated on each computation cycle to reflect the current values of true airspeed and atmospheric density. The inputs to this routine are commands for airspeed and flight-path angle, and the current values of airspeed and flight-path angle. The outputs are the commanded values of $\dot{V}_p$ and $\dot{\gamma}$. There are 25 lines of executable code in this function.

The inverse transfer function is implemented in function "inverse_xfunction." The code may be found in Appendix C. This function implements the state space model of the linear inverse transfer function, as described in equations (2.12) through (2.19). All operations are implemented using linear algebra. The **A**, **B**, **C**, and **D** matrices are all static once initialized. They are a function of the initial trim true airspeed only, and consequently do not change throughout a simulation run. The inputs to this routine are the required values of $\dot{V}_p$ and $\dot{\gamma}$. The outputs are elevator position and power-lever angle, PLA. "Power-lever angle" is an anachronism in this application, but refers to the historical use of "power levers" to control engine power. In this controller, power-lever angle is used to command a percent of maximum

available thrust for the given altitude and airspeed. PLA of 100 equates to 100 percent of available thrust. There are 20 lines of executable code in this function.

The **A**, **B**, **C**, and **D** matrices for the inverse transfer function are scheduled as functions of the trim airspeed. Scheduling is accomplished by explicitly calculating the coefficients for each matrix for ten different true airspeeds, from 100 feet/sec to 260 feet/sec, at an altitude of 3000 feet. The assumption of 3000 feet highlights that this inverse transfer function is approximate at other altitudes. These matrices were calculated using a MATLAB™ script. Internal MATLAB™ routines were used to convert the state space model from the continuous time domain to the discrete time domain. Each coefficient of each matrix was curve fit against a polynomial in speed. These resulting polynomials are coded into the C program. Function "update_inv_state_space" expands these polynomials once at the beginning of a run. While the calculations sound daunting, this function requires only 25 lines of code.

The adaptive neural network compensator is implemented through three C functions. Function "initialize_by_code" initializes the network configuration only once at the beginning of each simulation. Functions "forward_pass" and "backprop" accomplish the forward pass and backpropagation computations for the neural network. Online learning occurs during backpropagation. Function forward_pass has seven lines of executable code. Function backprop has 21 lines of executable code.

Function "DEF_JRcontrol" assembles all components into the functional controller. The computational sequence is as follows:

1. Compute Δ based on results from the inputs and outputs to the last computation cycle.
2. If requested, write training data to a scratch file for later formatting.
3. If learning is enabled, back propagate to learn from Δ calculated above.

4. Use the linear prefilter to compute the desired values of $\dot{V}_p$ and $\dot{\alpha}$.

5. Invoke angle of attack limiting, if requested.

6. Assemble the inputs to the network for this computation cycle.

7. If the adaptive neural network compensator is active, calculate values for $\hat{\Delta}\dot{V}_p$ and $\hat{\Delta}\dot{\alpha}$ and add to desired $\dot{V}_p$ and $\dot{\alpha}$ to get required $\dot{V}_p$ and $\dot{\alpha}$. Otherwise, set the required $\dot{V}_p$ and $\dot{\alpha}$ values to be the same as the desired $\dot{V}_p$ and $\dot{\alpha}$.

8. Use the linear inverse transfer function to calculate the control inputs for elevator and power-lever angle.

   The resulting control inputs are then returned to the simulation to fly the airplane.

CHAPTER 4

RESULTS

4.1 Introduction

Twenty-three simulations are presented to demonstrate the controller's ability to meet the objectives outlined in subchapter 1.2. The scenarios examined in each of these cases are summarized in Table 2. Each case is addressed by a series of three pages of time histories. Page one illustrates the perturbations in speed, flight-path angle, and the controls (elevator and power-lever angle) from the trim values. The command inputs are over-plotted on speed and flight-path angle. Pages two and three show the full values of the flight parameters in common engineering units. It is important to note that the simulation works with the full values, and the perturbation quantities are simply shown for convenience when comparing back to an ideal second-order system. Page three also contains a running average of the network RMS error. This parameter is an indicator of the learning achieved by the network. Small values of this parameter indicate that the network is well trained and little additional learning is taking place. RMS errors on the order of one suggest that the network is experiencing significant new information and that significant learning is taking place. This organization applies to all example cases that follow.

As previously discussed, the current controller permits arbitrary values to be prescribed for the dynamic characteristics of the overall system. For the cases examined, the following characteristics are assigned: $\zeta_u = .8$, $\omega_{nu} = 1$ rad/sec, $\zeta_\gamma = .7$, and $\omega_{nu} = 2.5$ rad/sec. These values produce a rise time of 3.12 seconds for speed and .856 seconds for flight-path angle. These characteristics are chosen for illustrating the generality of the current controller but may not be the optimum for handling qualities.

The controller as implemented in this work is intended to demonstrate that the academic objectives of the controller are satisfied: obtaining prescribed system performance and adaptation to changes in the physical plant. To make these objectives evident, some "real-world" considerations have been purposely neglected. No limits are imposed on elevator or thrust commands. No limits are placed on normal load factor or speed. Such limits would obscure the performance of the system toward achievement of the characteristics prescribed. A simple limiter is imposed on angle of attack at fourteen degrees. This angle of attack limiter is exercised in some of the simulations examined to demonstrate that the controller architecture is compatible with the types of limits that would need to be imposed in a real application.

4.2 Nominal Controller Performance

Six cases follow that illustrate the performance of the controller for the nominal physical plant. These cases span a range of altitude and speed and demonstrate the controller's performance across the aircraft flight envelope. The command inputs are a series of step inputs in speed or flight-path angle. In addition to the input commands, a reference response is over-plotted on speed and flight-path. This reference response is the response that would be expected from an ideal second-order linear system with the specified characteristics. Cases 1 through 6 (Figures 37 through 42) demonstrate several important characteristics of the controller. First, the controller closely approximates the second-order performance desired. Examination of the perturbation velocity graph shows that the simulation almost perfectly overlays the reference. Examining the perturbation gamma graph shows a similar comparison. The flight-path angle is still good but does show some very slight tendency to overshoot the reference. This overshoot suggests that a somewhat lower damping ratio is achieved than is targeted. This small loss of damping can be traced to the gamma filter within the linear prefilter, and the phase shift it

produces in flight-path angle feedback. If exact damping performance is required, the target value for damping ratio can be adjusted to compensate. Second, comparing the results of Cases 1 through 6 shows that controller performance is consistent across all speeds and altitudes. Third, little learning is occurring. The RMS error graph shows a low level of learning for all cases. This is to be expected since the adaptive neural network was pre-trained using the flight characteristics of the nominal aircraft. The step commands in flight-path angle show higher levels of learning taking place, since step inputs were not part of the training maneuvers and these are "new" to the neural network. Finally, it should be observed that there is no evidence of the phugoid or short period modes associated with typical General Aviation airplanes. These classically coupled modes have been totally suppressed.

TABLE 2

SUMMARY OF SIMULATION CASES EXAMINED

Case #	Description	NNet Compensation	Failure State	Speed	Altitude
1	Square Wave on Speed	On	None	80	3000
2	Square Wave on Speed	On	None	100	3000
3	Square Wave on Speed	On	None	150	3000
4	Square Wave on Gamma	On	None	80	5000
5	Square Wave on Gamma	On	None	100	5000
6	Square Wave on Gamma	On	None	150	5000
7	Square Wave on Speed, with Gamma Step at t=2	Off	None	120	6000
8	Square Wave on Speed, with Gamma Step at t=3	On	None	120	6000
9	Square Wave on Speed, with Gamma Step at t=4	Off	Halve CMA at t=6	120	6000
10	Square Wave on Speed, with Gamma Step at t=5	On	Halve CMA at t=6	120	6000
11	Steck Case 1	On	As Noted	100	3000
12	Steck Case 2	On	As Noted	100	3000
13	Steck Case 3	On	As Noted	100	3000
14	Steck Case 4	On	As Noted	100	3000
15	Long Duration Manuever	On	None	100	3000
16	Long Duration Manuever	On	CMA = 0 after t=400	100	3000
17	Non-Linear CLA and CMA	On	None	80	3000
18	Non-Linear CLA and CMA	On	Halve CLDE and CMDE at t=400	80	3000
19	Sine Wave on Speed	On	None	100	3000
20	Sine Wave on Speed	On	Neural Network Udot Hardover	100	3000
21	Sine Wave on Speed	On	Neural Network Udot Oscillatory Failure	100	3000
22	Sine Wave on Speed	On	Neural Network Alphadot Hardover	100	3000
23	Sine Wave on Speed	On	Neural Network Alphadot Oscillatory Failure	100	3000

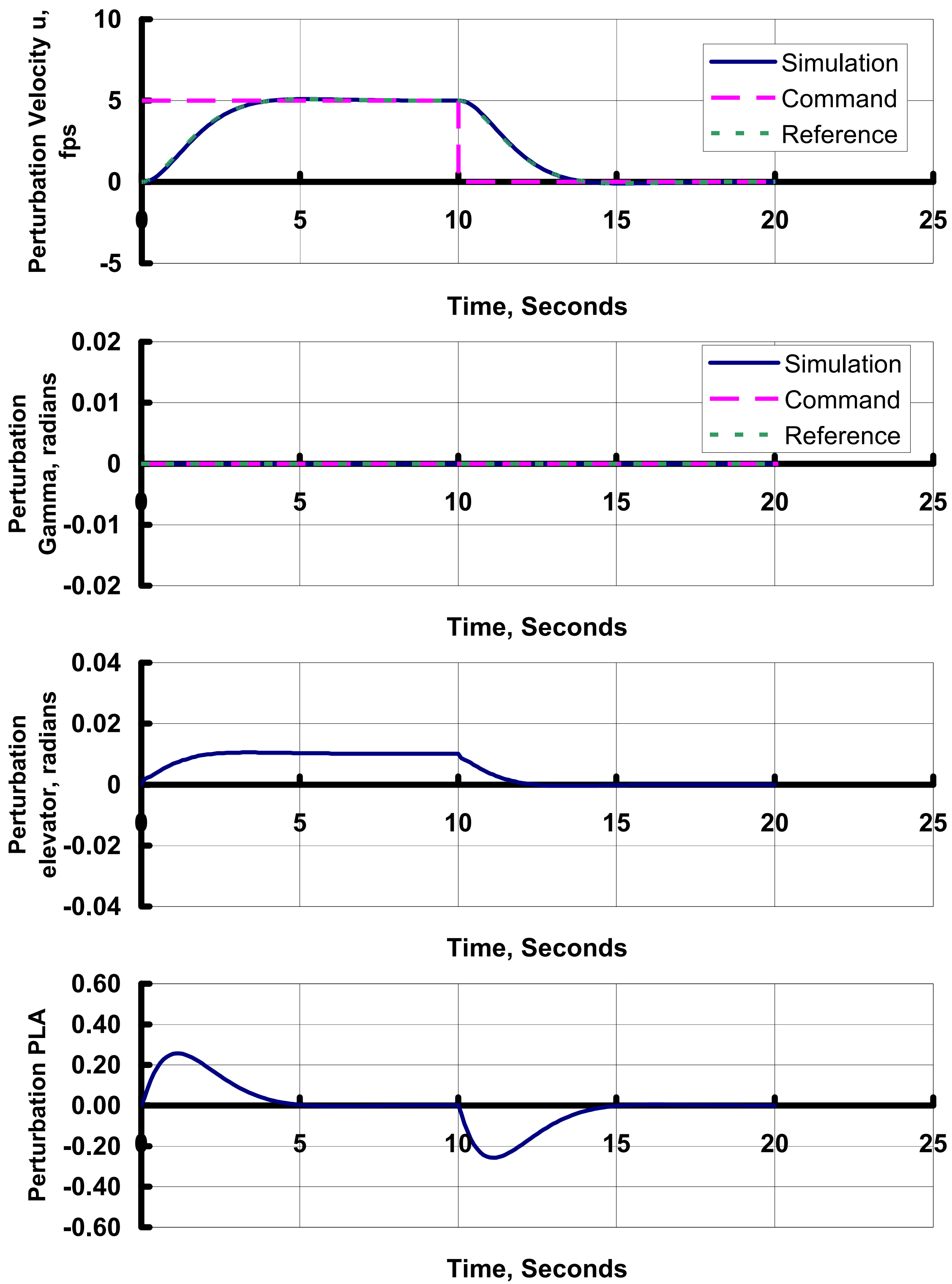


Figure 37 (a): Case 1 time history

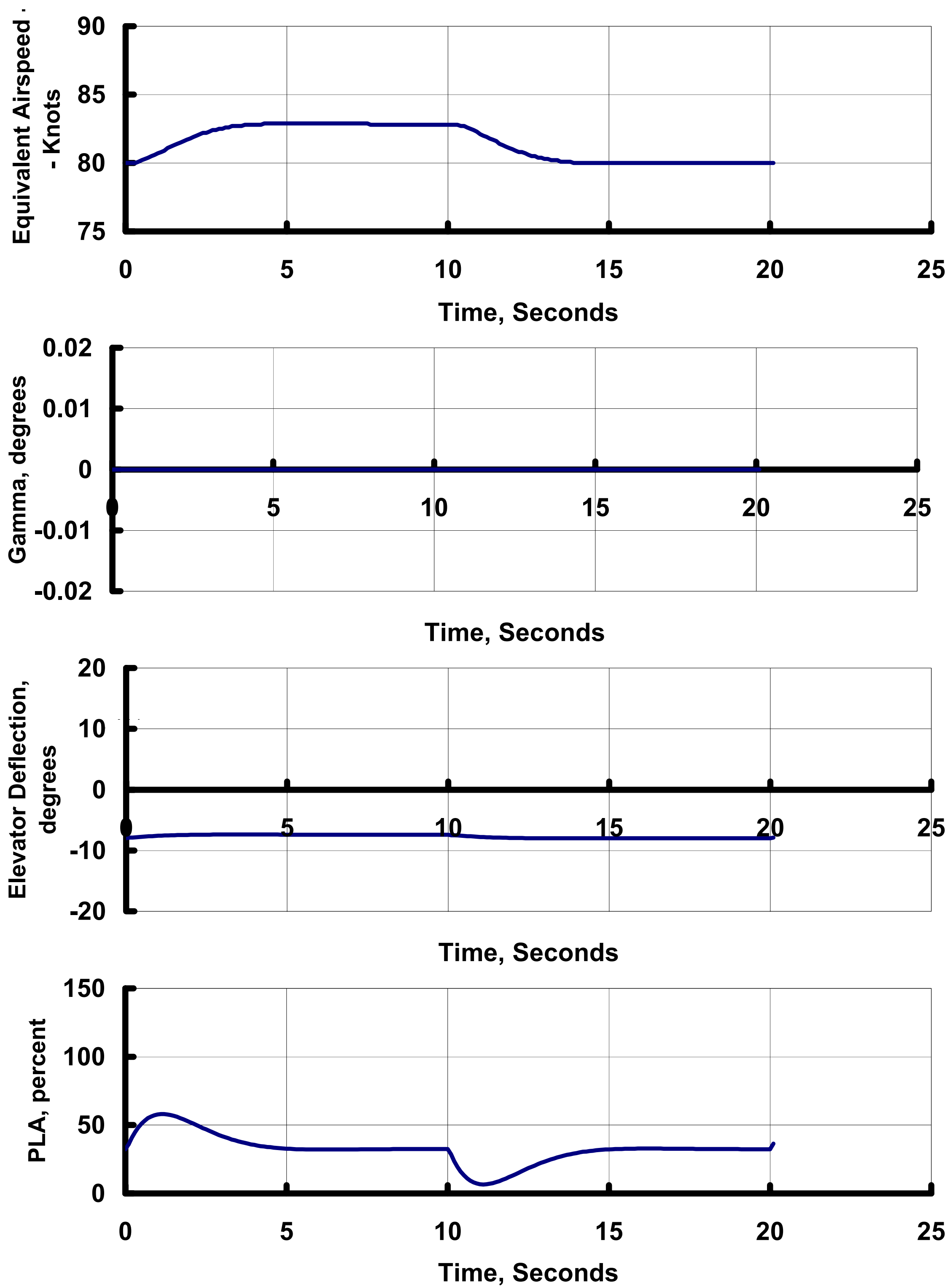


Figure 37 (b): Case 1 time history

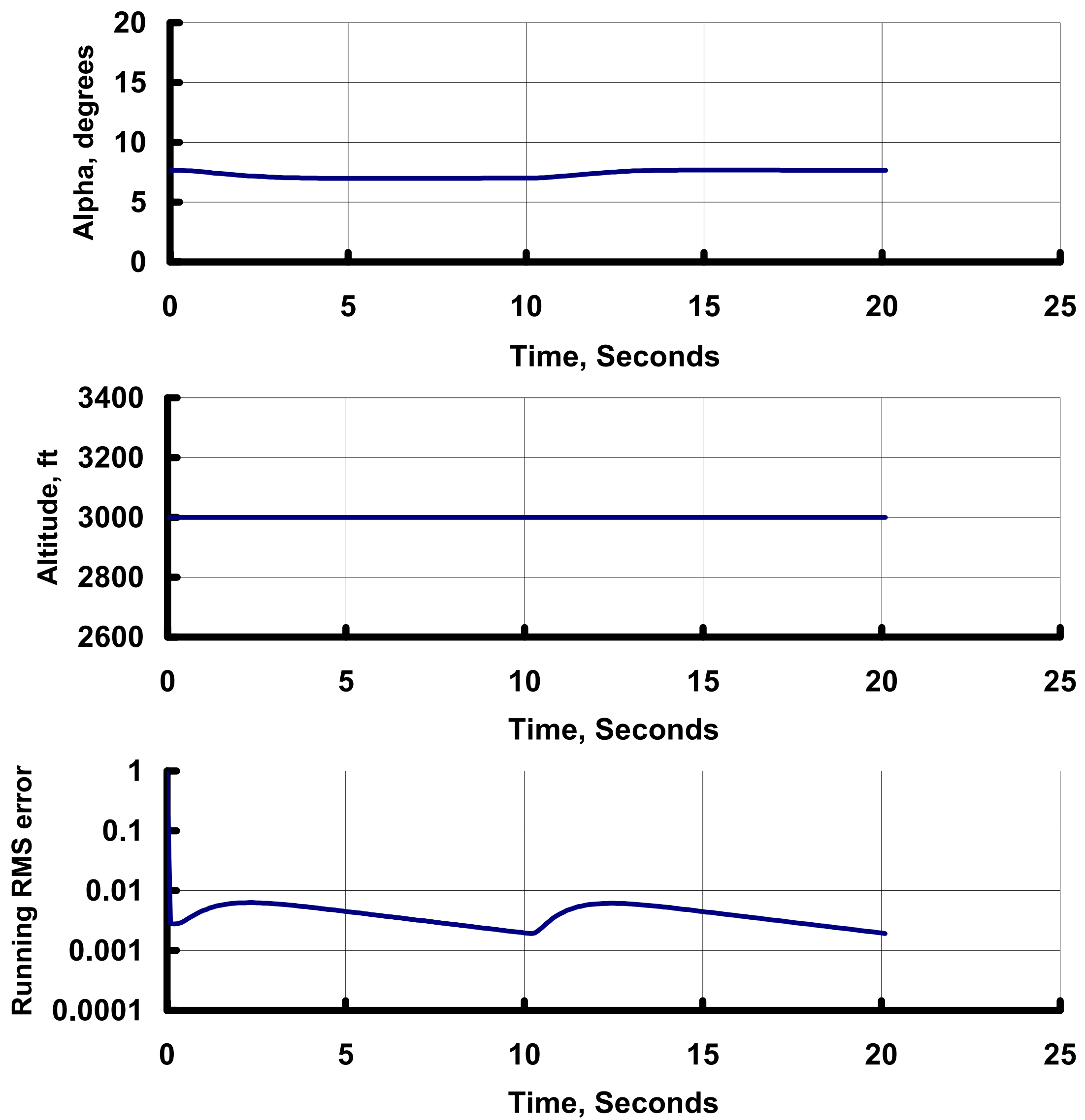


Figure 37 (c): Case 1 time history

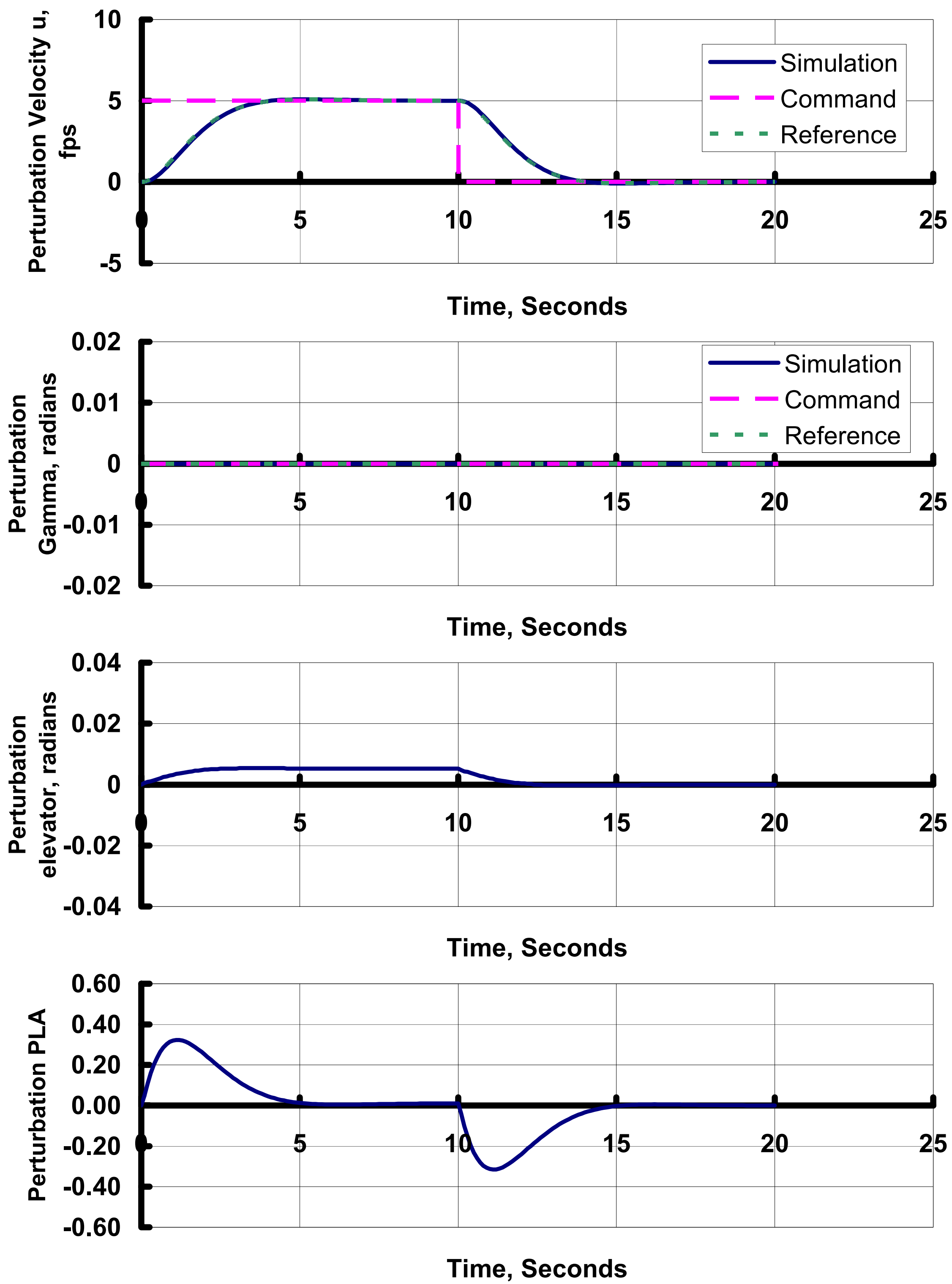


Figure 38 (a): Case 2 time history

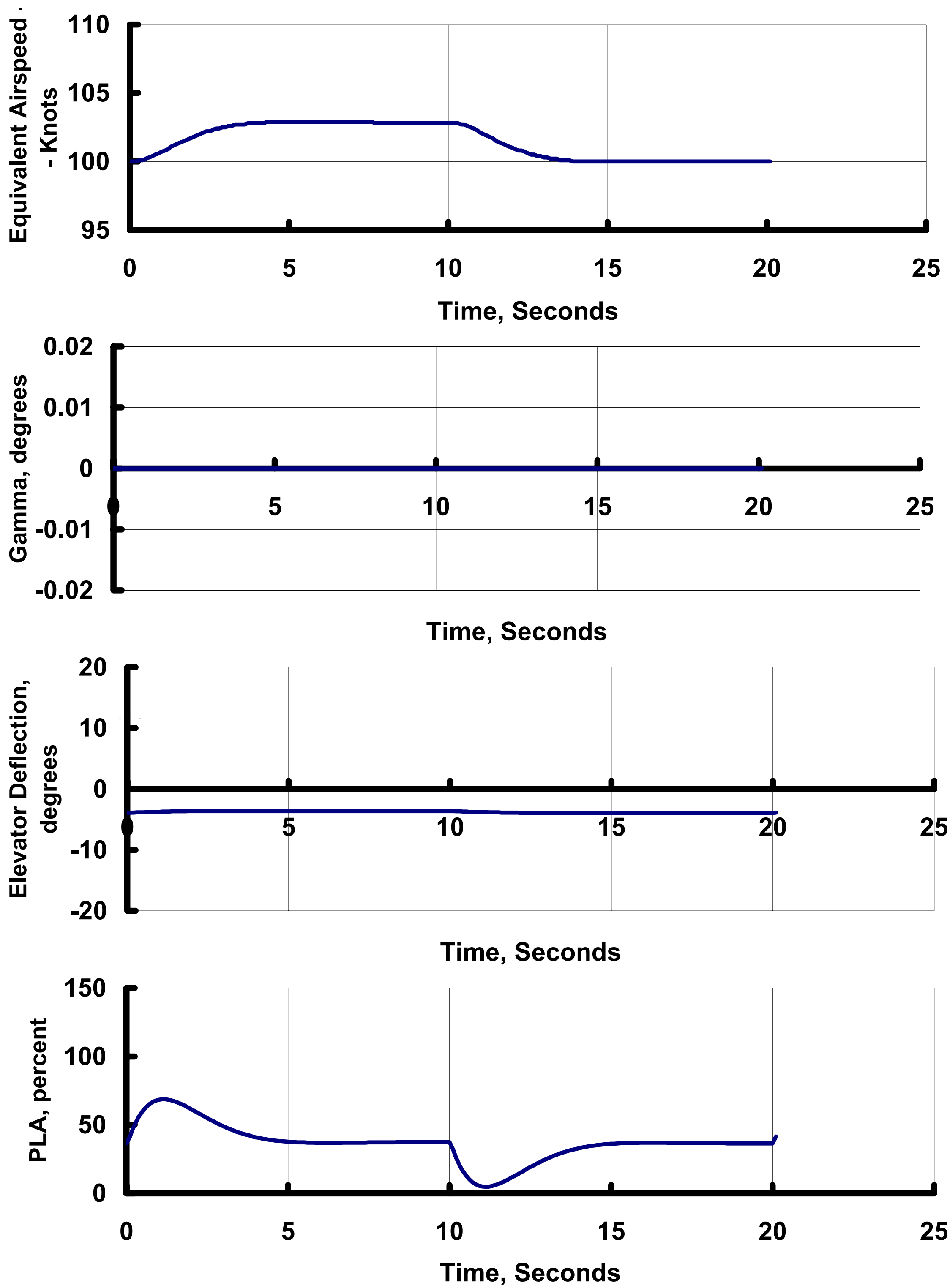


Figure 38 (b): Case 2 time history

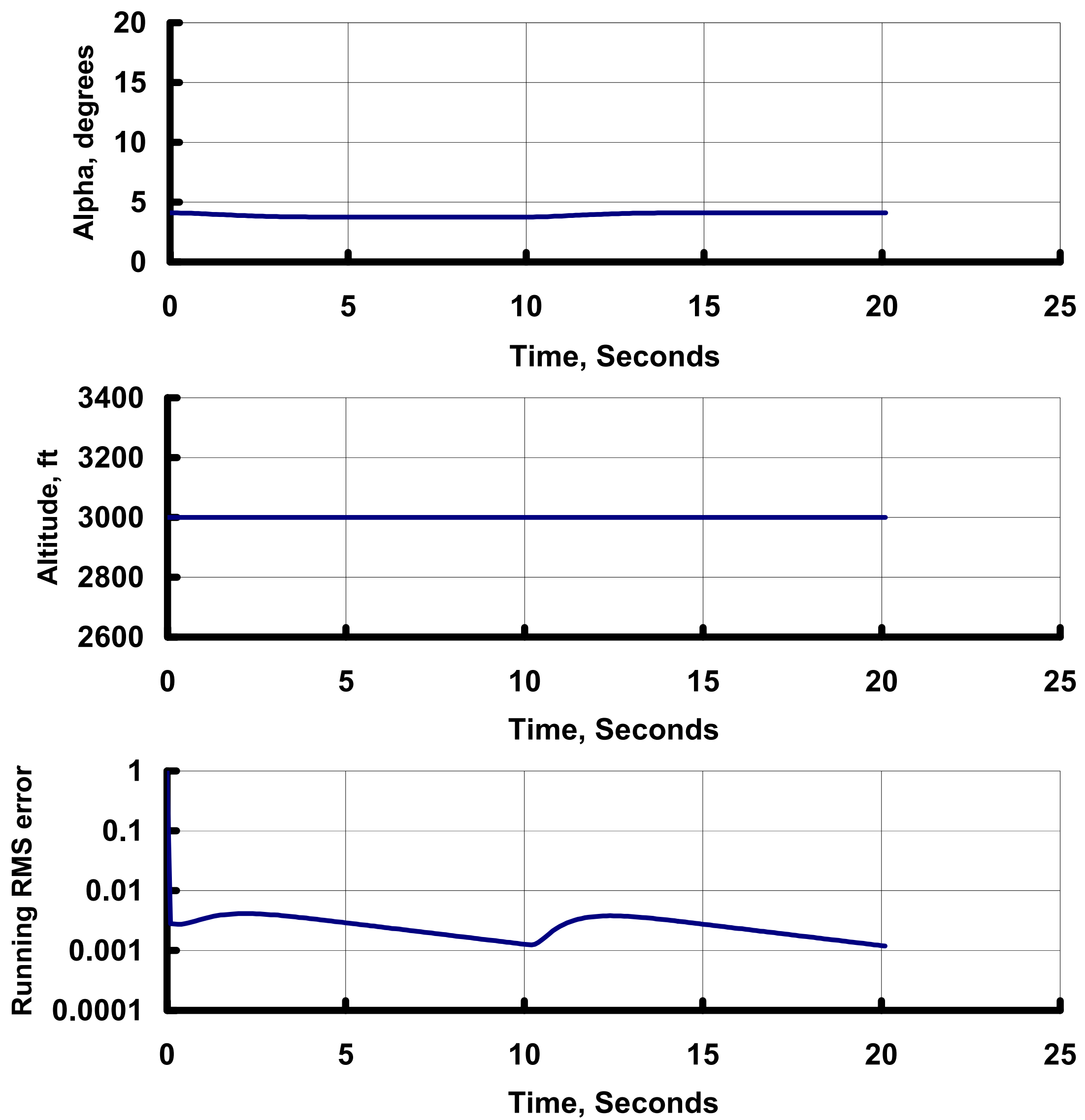


Figure 38 (c): Case 2 time history

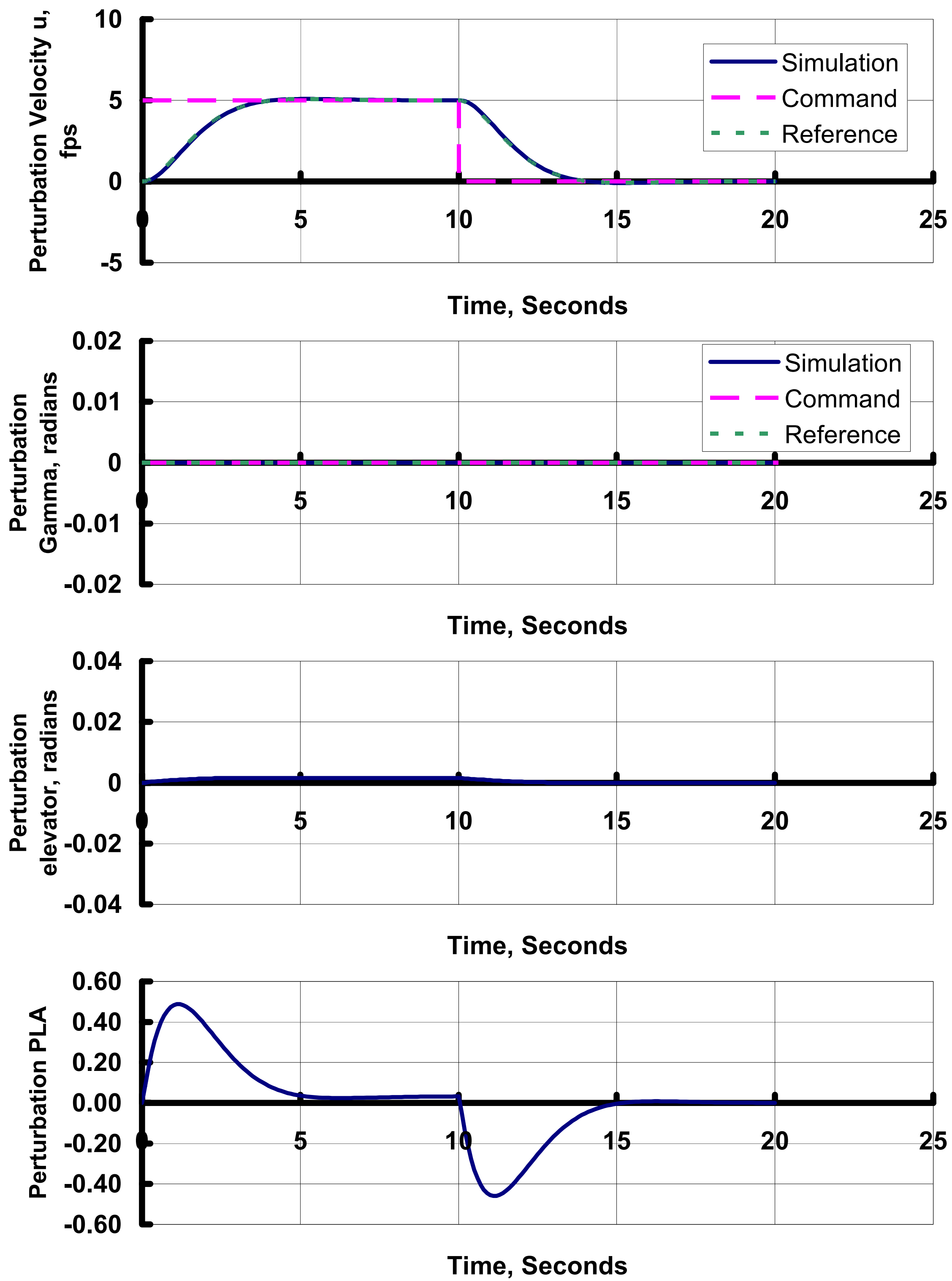


Figure 39 (a): Case 3 time history

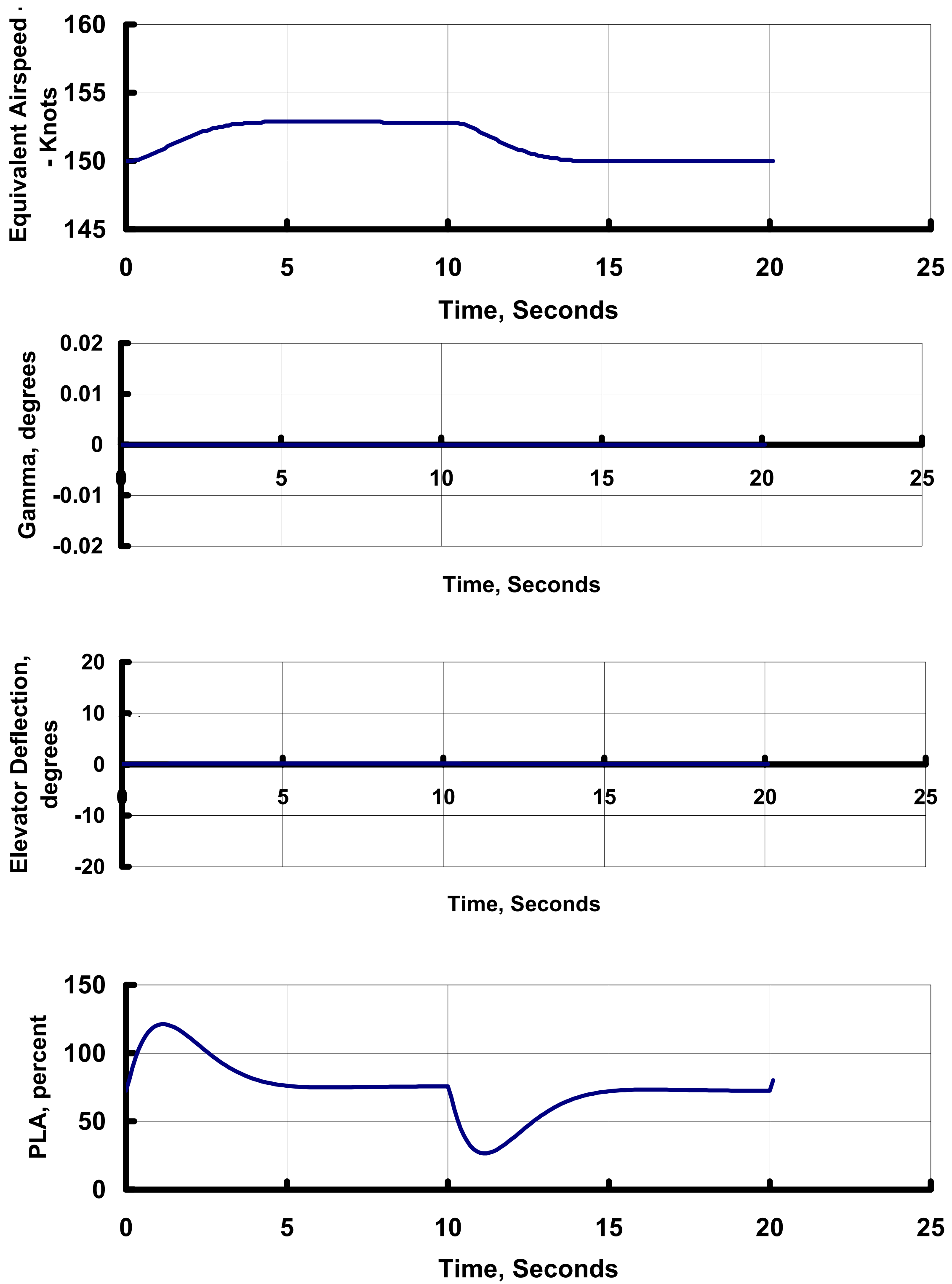


Figure 39 (b): Case 3 time history

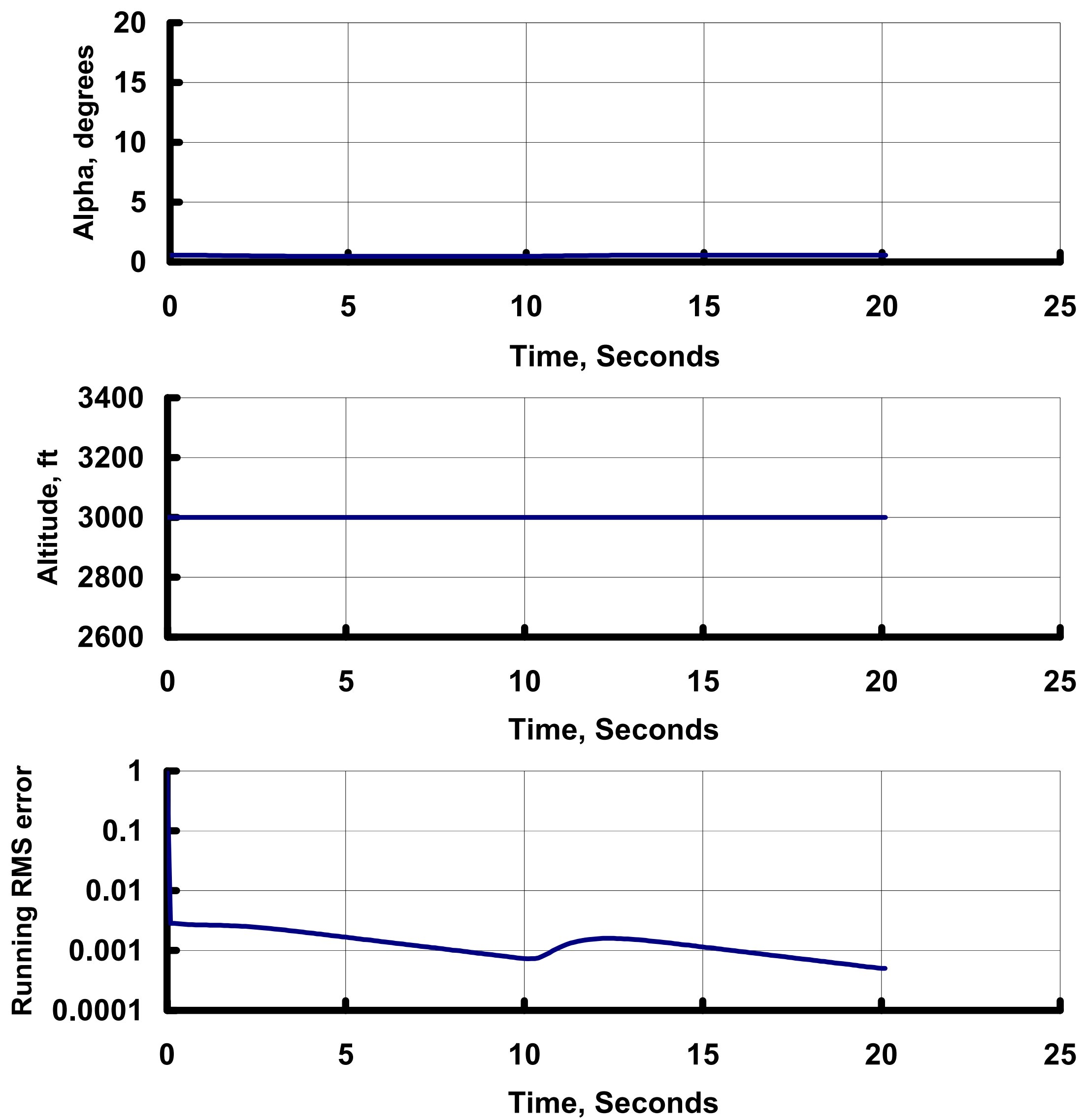


Figure 39 (c): Case 3 time history

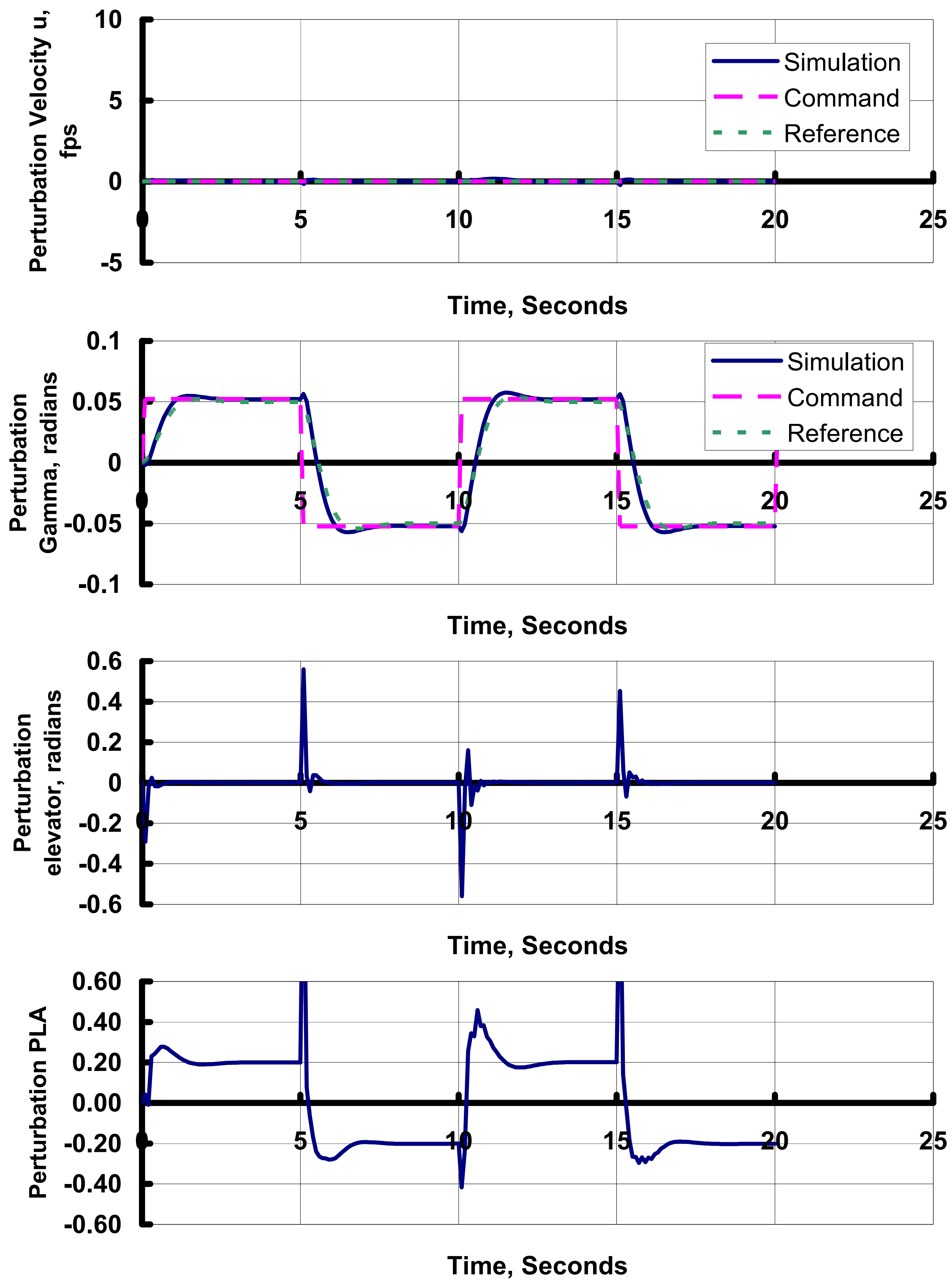


Figure 40 (a): Case 4 time history

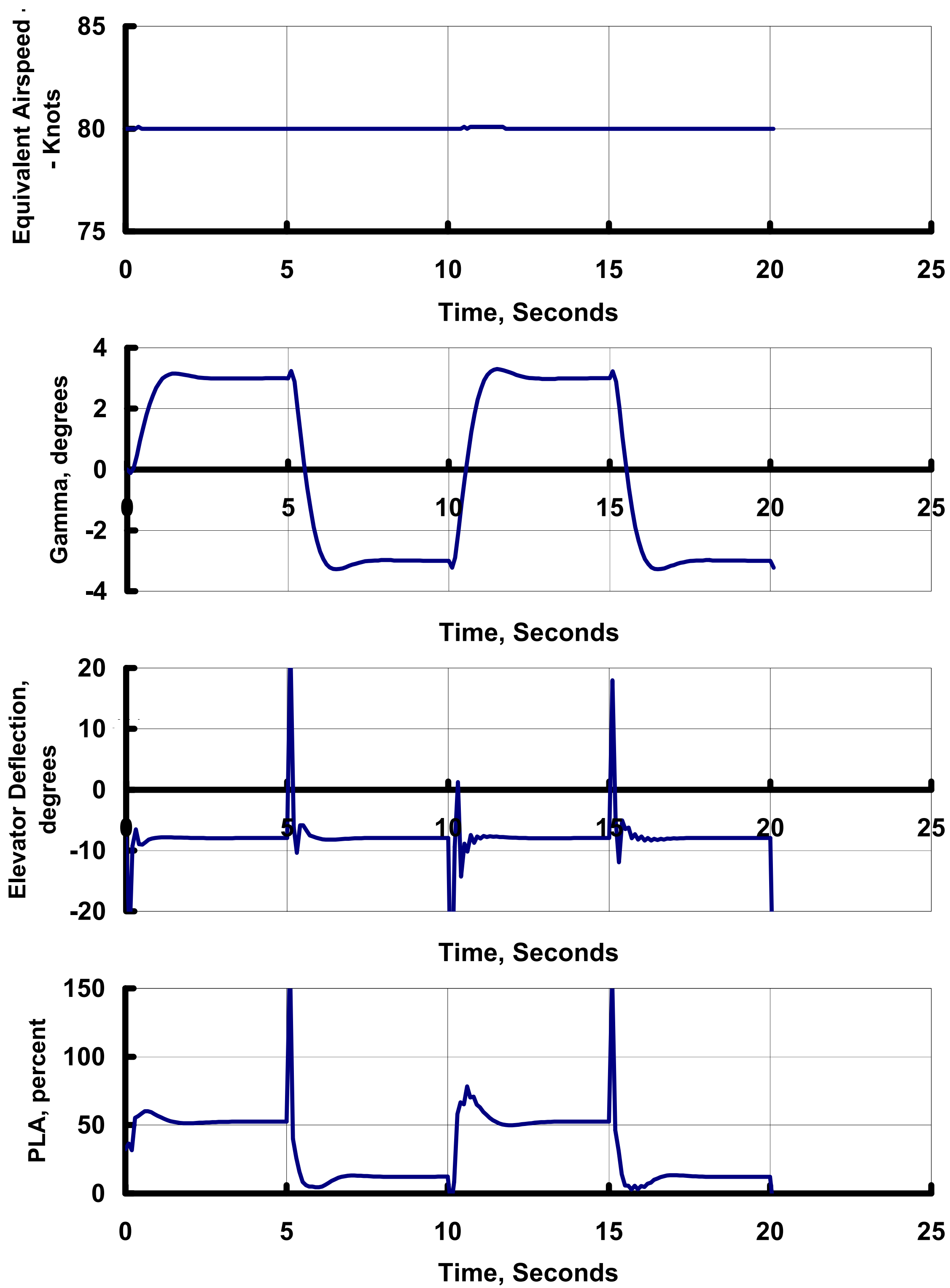


Figure 40 (b): Case 4 time history

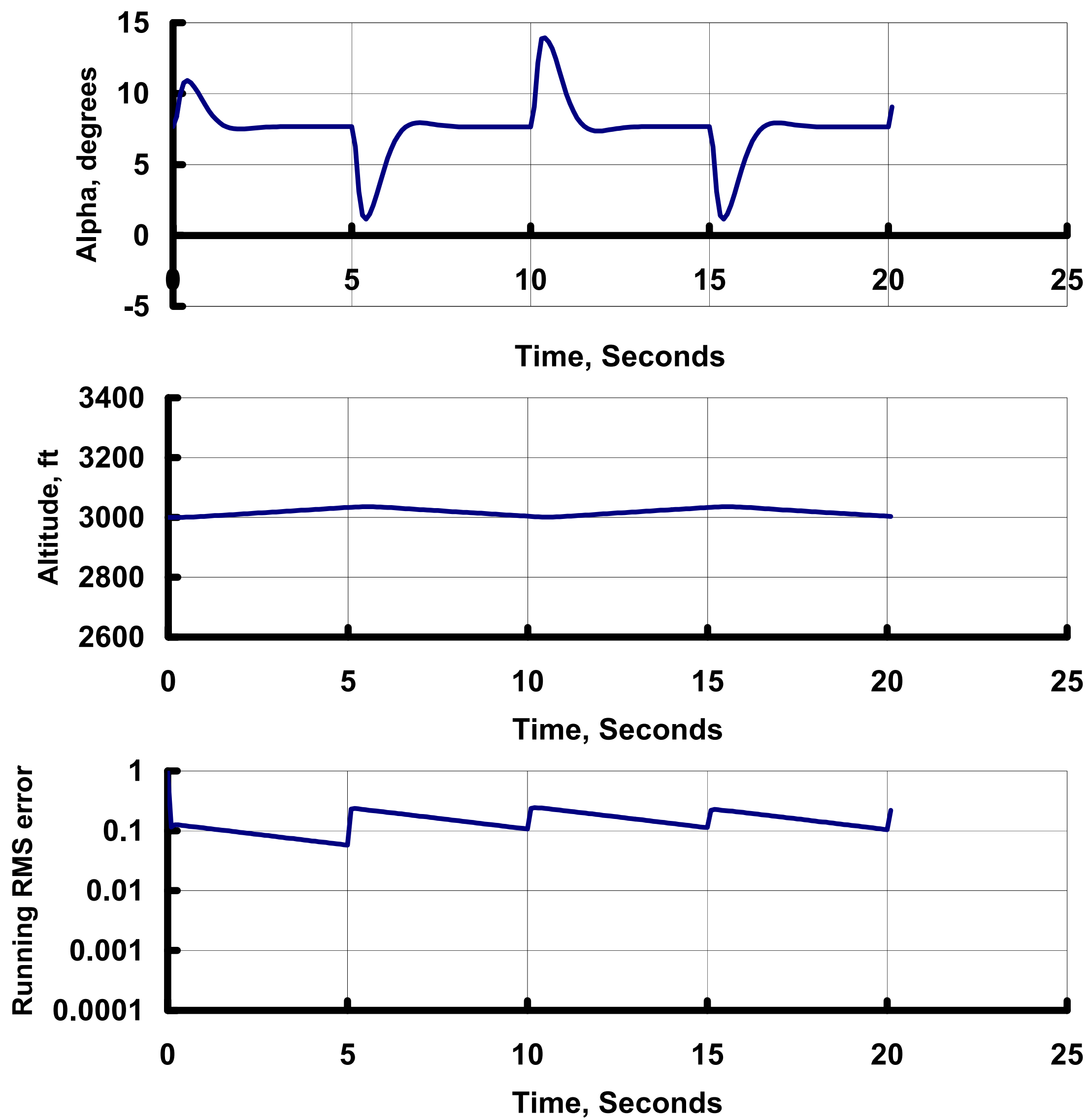


Figure 40 (c): Case 4 time history

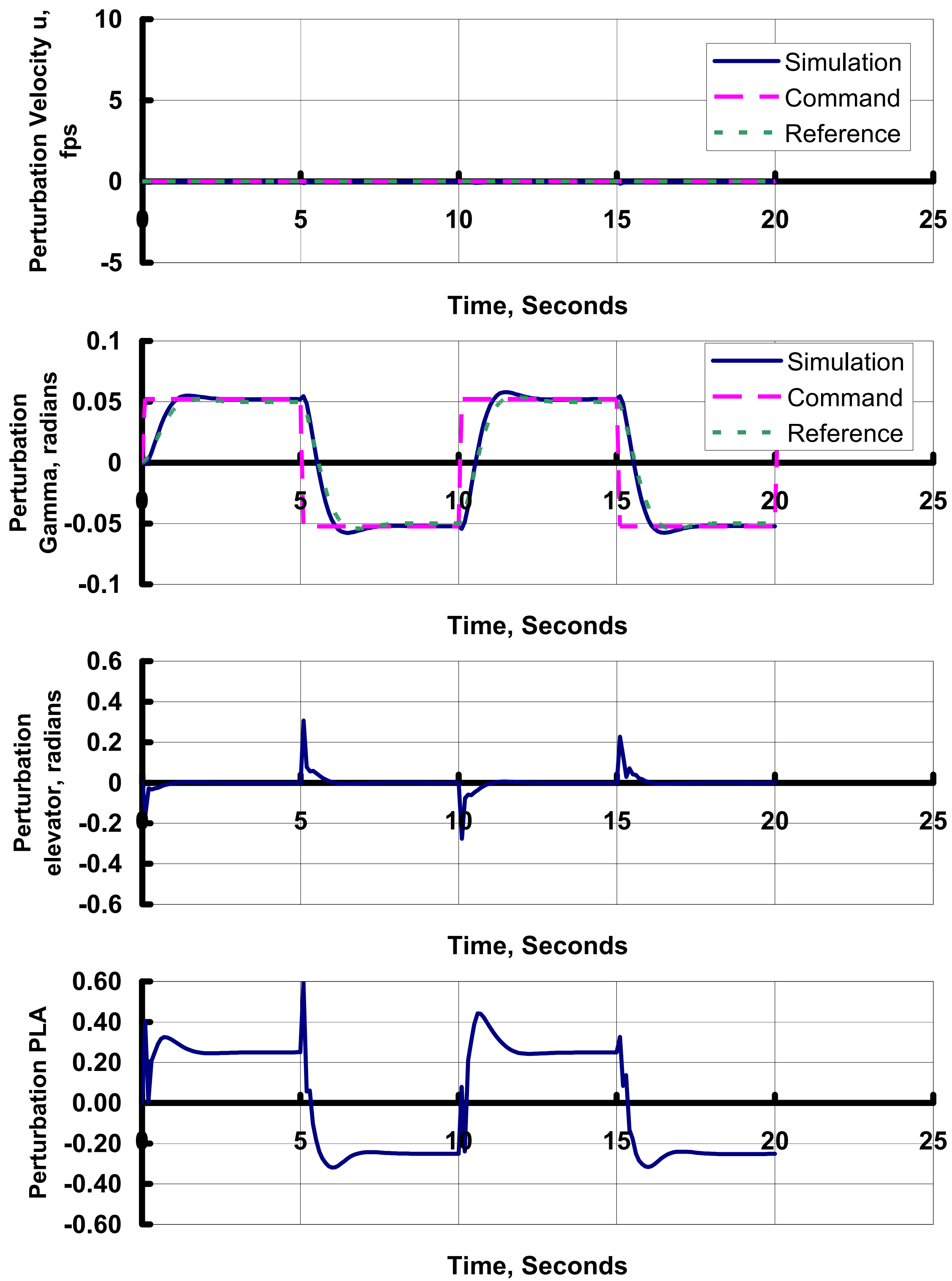


Figure 41 (a): Case 5 time history

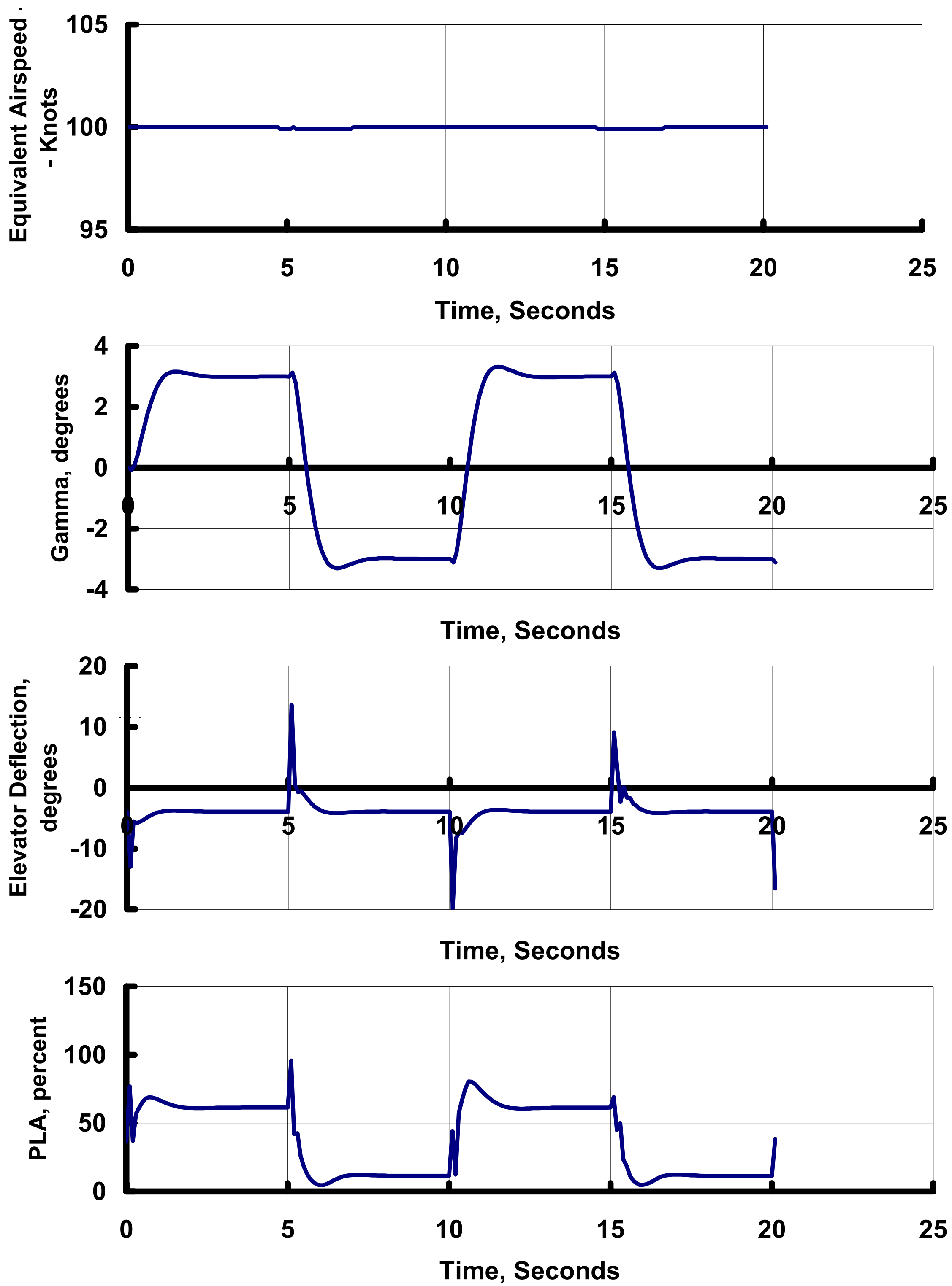


Figure 41 (b): Case 5 time history

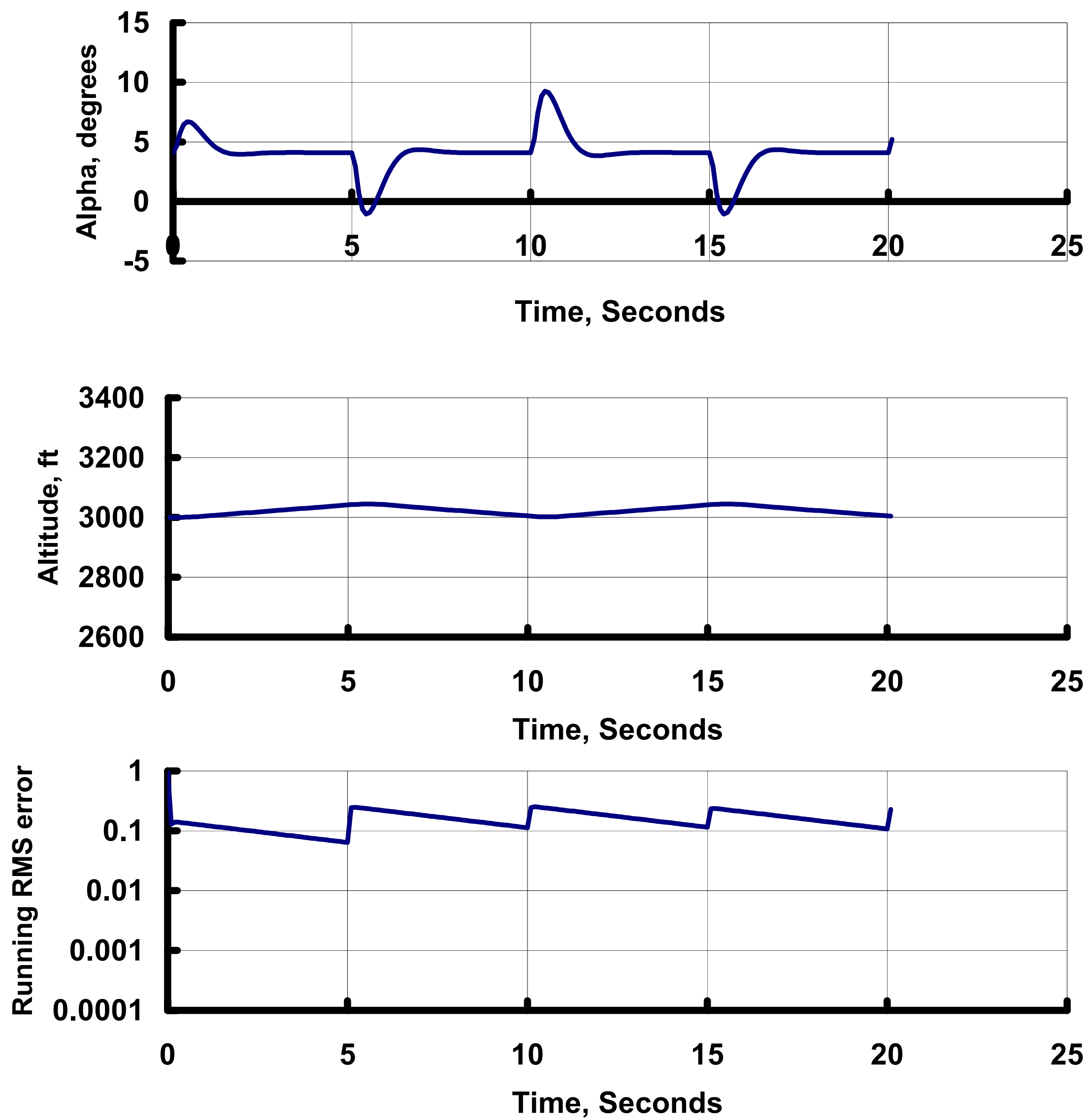


Figure 41 (c): Case 5 time history

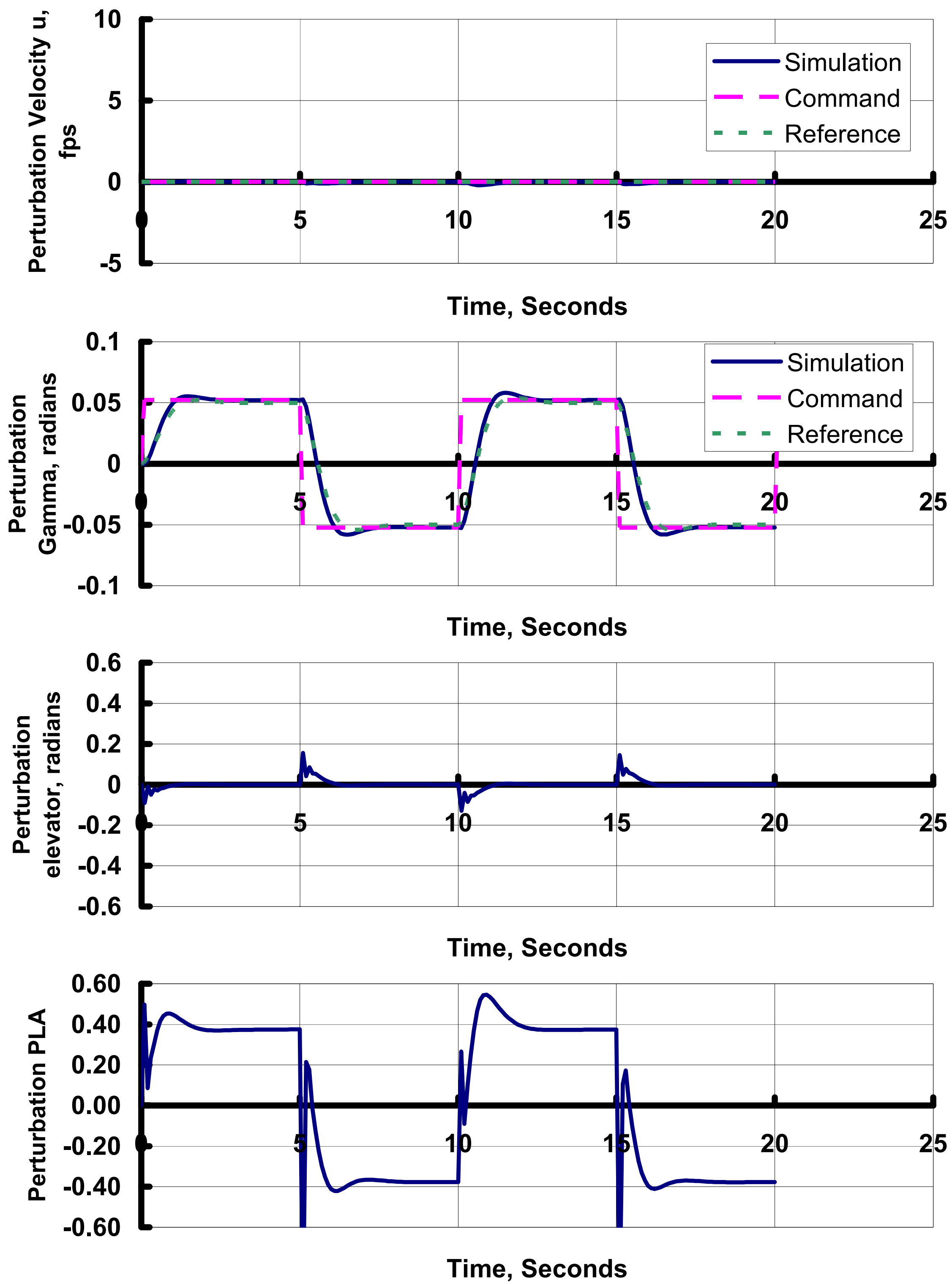


Figure 42 (a): Case 6 time history

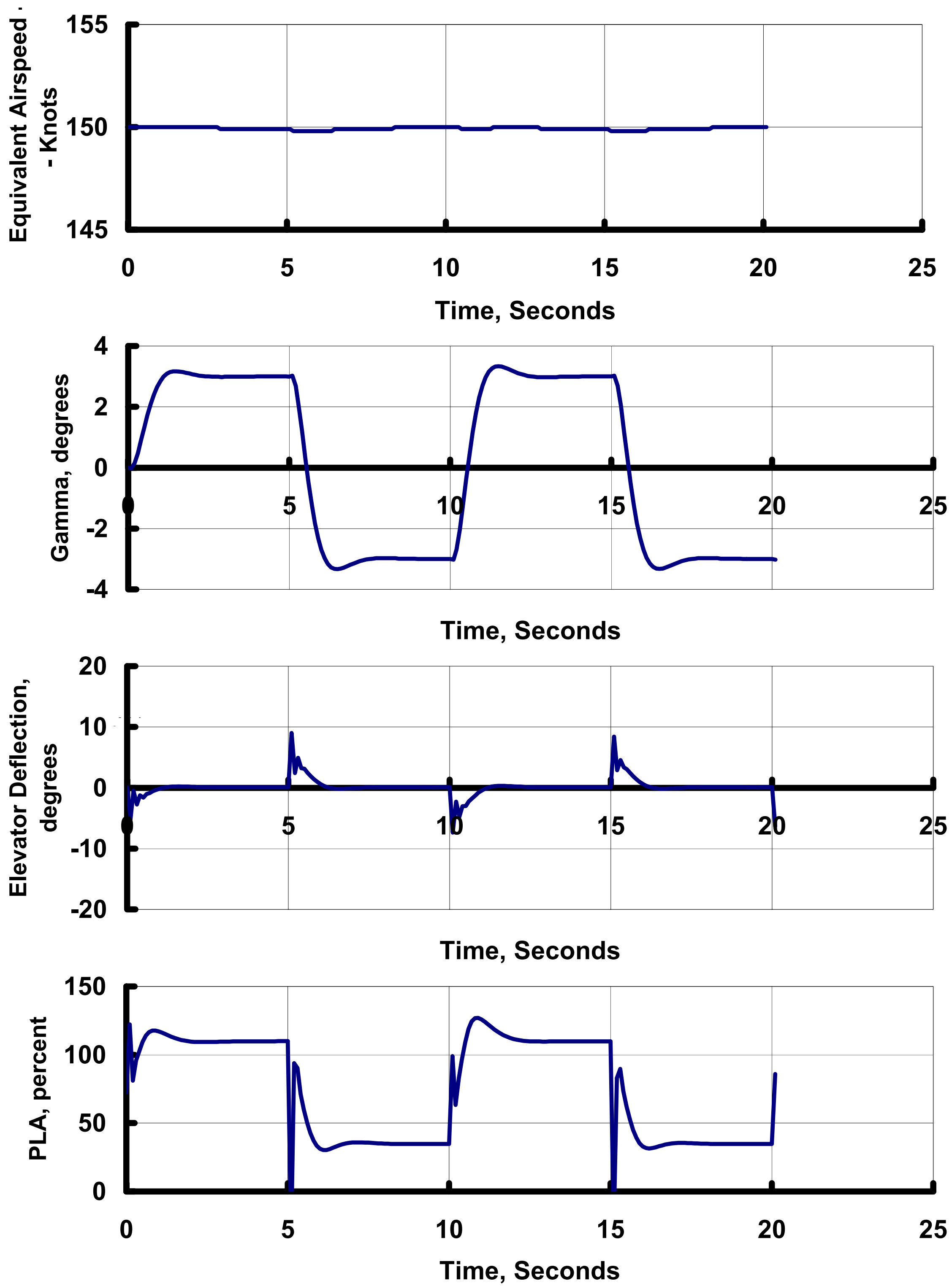


Figure 42 (b): Case 6 time history

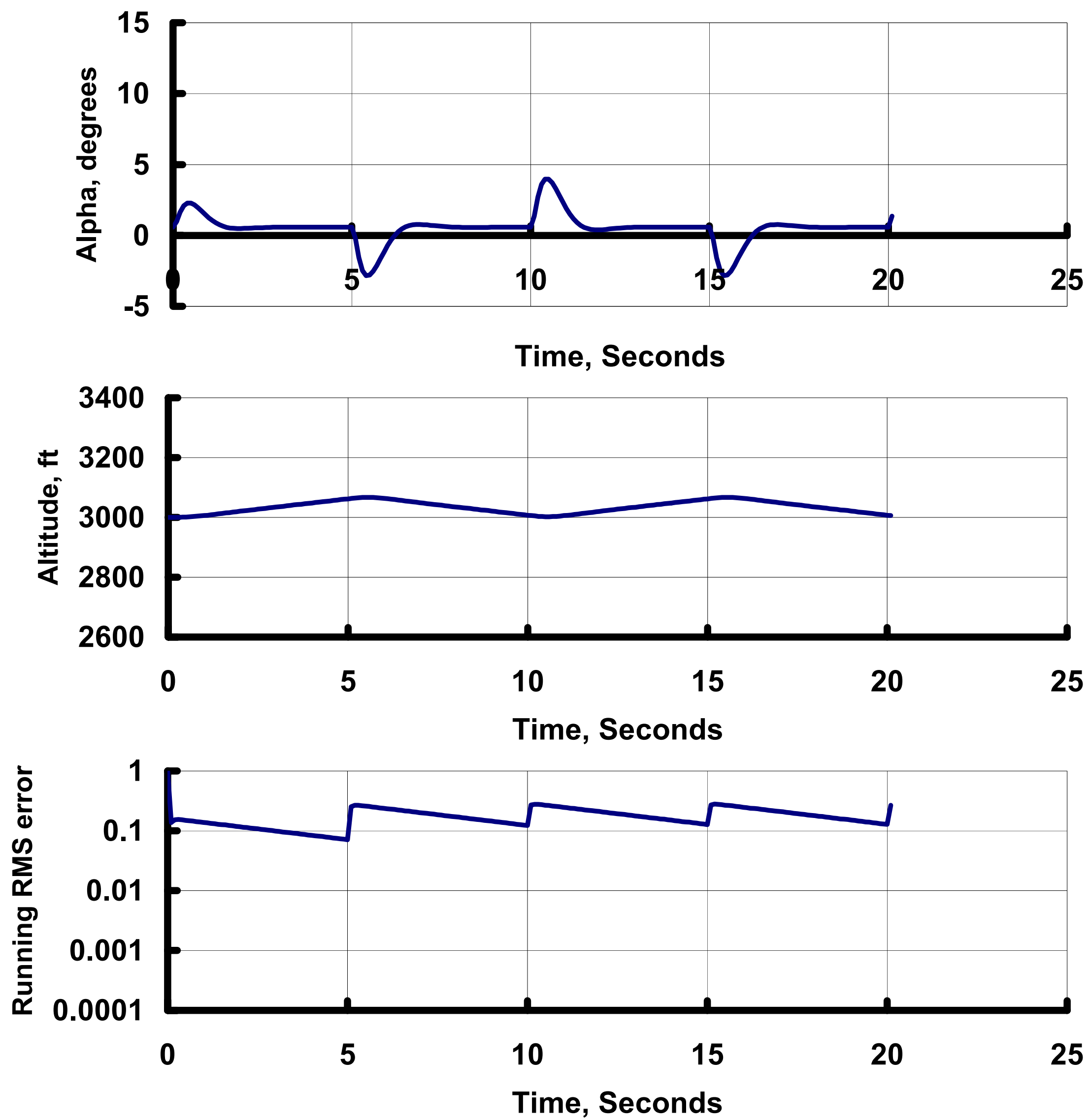


Figure 42 (c): Case 6 time history

4.3 Failures Involving Stability and Controllability of the Physical Plant

Cases 7 through 14 examine the controller's ability to adapt to failures within the physical plant. Good performance will be evaluated by the controller's ability to maintain the desired second-order characteristics with a minimal transient between the nominal and the failure state.

Cases 7 through 10 (Figures 43 through 46) all address the same commanded maneuver. The airplane is commanded to fly a speed square wave, with speed steps of five feet/second. At two seconds into the simulation, a step change in flight-path is commanded equal to eight degrees.

Case 7 demonstrates the controller's performance for the nominal physical plant and no neural network compensation. This case can be compared with Case 8 where the neural network actively compensates for modeling errors. At first glance, it appears that whether the network is on or off makes little appreciable difference for these two cases. This is to be expected, since the linear inverse transfer function is built around the nominal aircraft state, therefore there should be little modeling or inversion error. However, on close inspection of the perturbation velocity trace it is possible to see that speed tracking accuracy is somewhat degraded in Case 7, without the neural network compensation. Furthermore, without the neural network compensation in Case 7, a defect in speed, relative to the reference, is apparent between about two seconds and five seconds. The neural network corrects this defect, and all "humps" in Case 8 are similar. Even with a nominal system, an improvement in tracking is available with the neural network on.

The neural network compensation is not involved in flying the aircraft in Case 7. However, the neural network is continuing to learn. This is apparent in the RMS error time history. This will be seen in all cases with the neural network compensation off.

The time history for equivalent airspeed in Case 8 shows what might appear to be a drift in equivalent airspeed. This is not a drift, but rather reflects the general decrease in equivalent airspeed with increasing altitude as the controller tracks to true airspeed commands. The change in altitude is evident in the altitude trace.

Case 9 begins the same as Case 7. The airplane is initially in its nominal state with no neural network compensation. However, an anomaly of some kind is assumed to occur in Case 9 at t = 6 seconds that reduces the stability of the airplane. The derivative C_{M_α} is cut in half. A vertical dashed line in Figure 45 graphically indicates the time of the failure. Since the linear transfer function incorrectly assumes nominal aircraft stability, and with the no neural network compensation, the airplane receives erroneous elevator commands and deviates from the commanded maneuver. Speed tracking is the most effected without the benefit of the neural network. Speed begins an apparent divergence from commanded speed, at least within the time window examined. Flight-path angle is also somewhat degraded. Comparison of the gamma perturbation plot with Case 8 shows a transient on the commanded flight-path angle after the failure.

Case 10 repeats the scenario of Case 9, only now the adaptive neural network compensation is active. With the adaptive neural network compensation, there is no appreciable deviation from the commanded maneuver when the failure occurs at six seconds. Comparing speed and flight-path angle tracking for Case 10 with Case 8 shows them to be virtually the same, even with the failure present in Case 10. From the perturbation elevator trace, it is apparent that a change occurs at six seconds. A change in elevator deflection for Case 10 reflects what is essentially a new trim state based on the new value for C_{M_α}. The neural network compensator quickly adapts to the failure and continues the maneuver with little interruption.

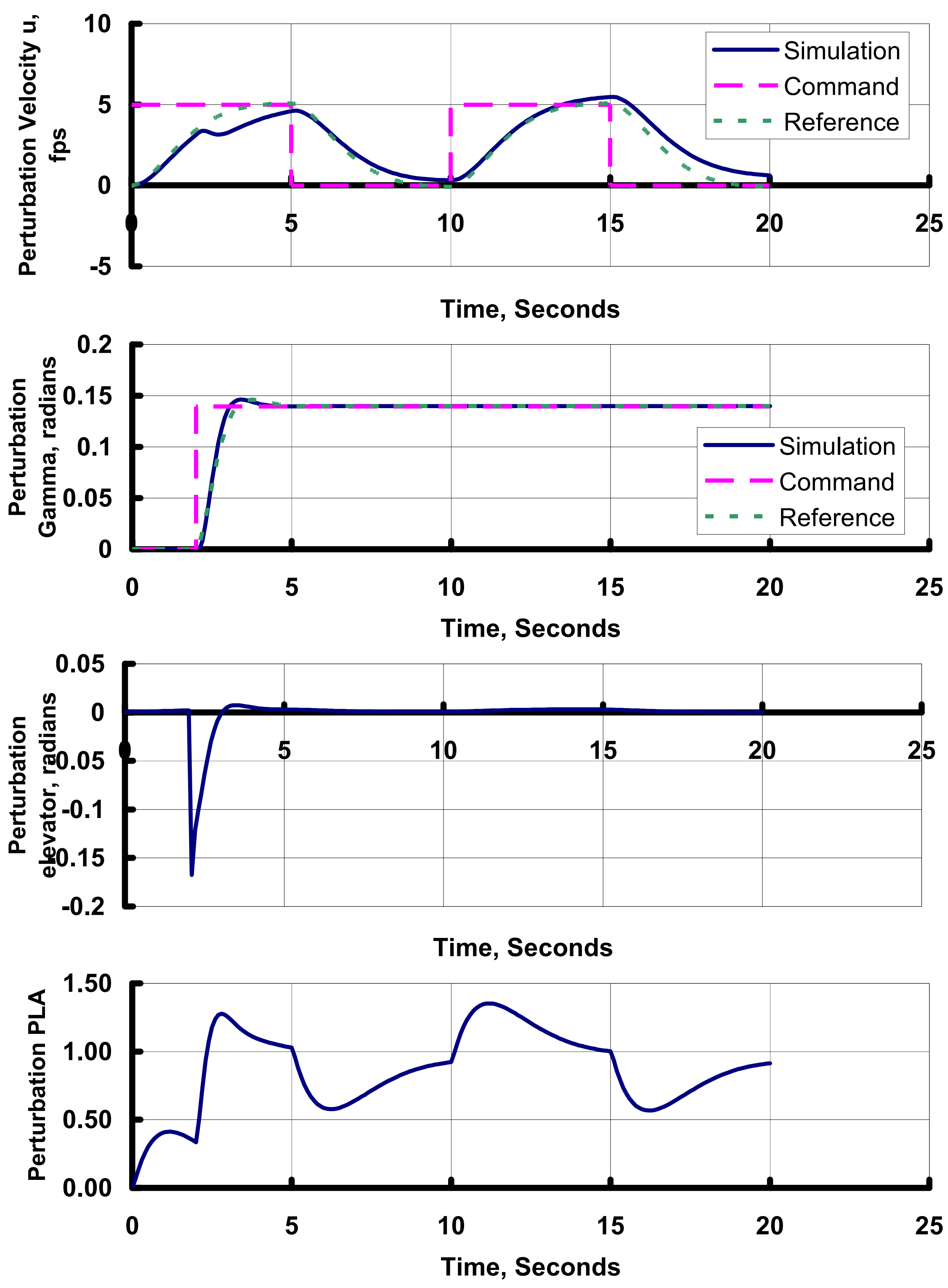


Figure 43 (a): Case 7 time history

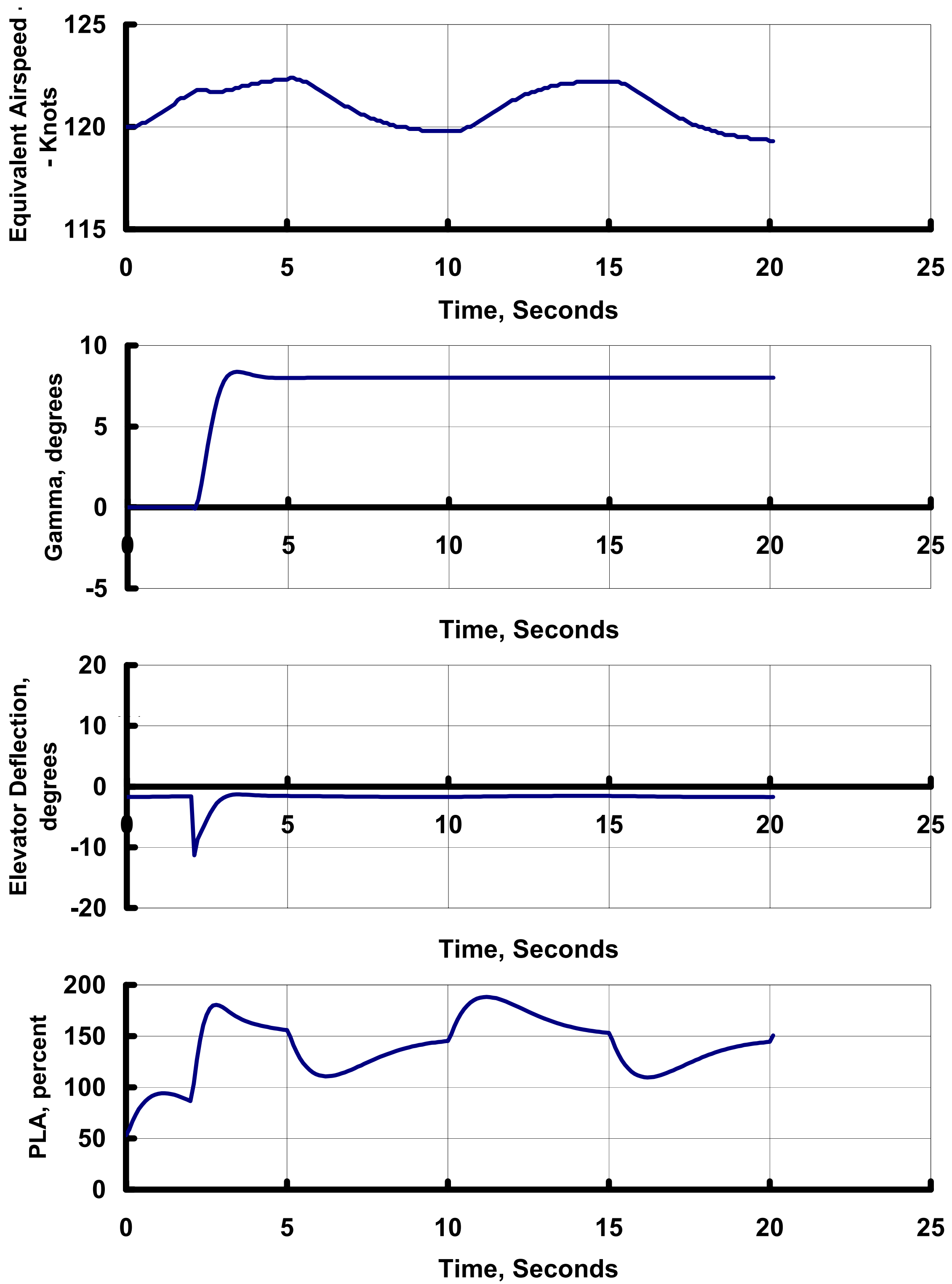


Figure 43 (b): Case 7 time history

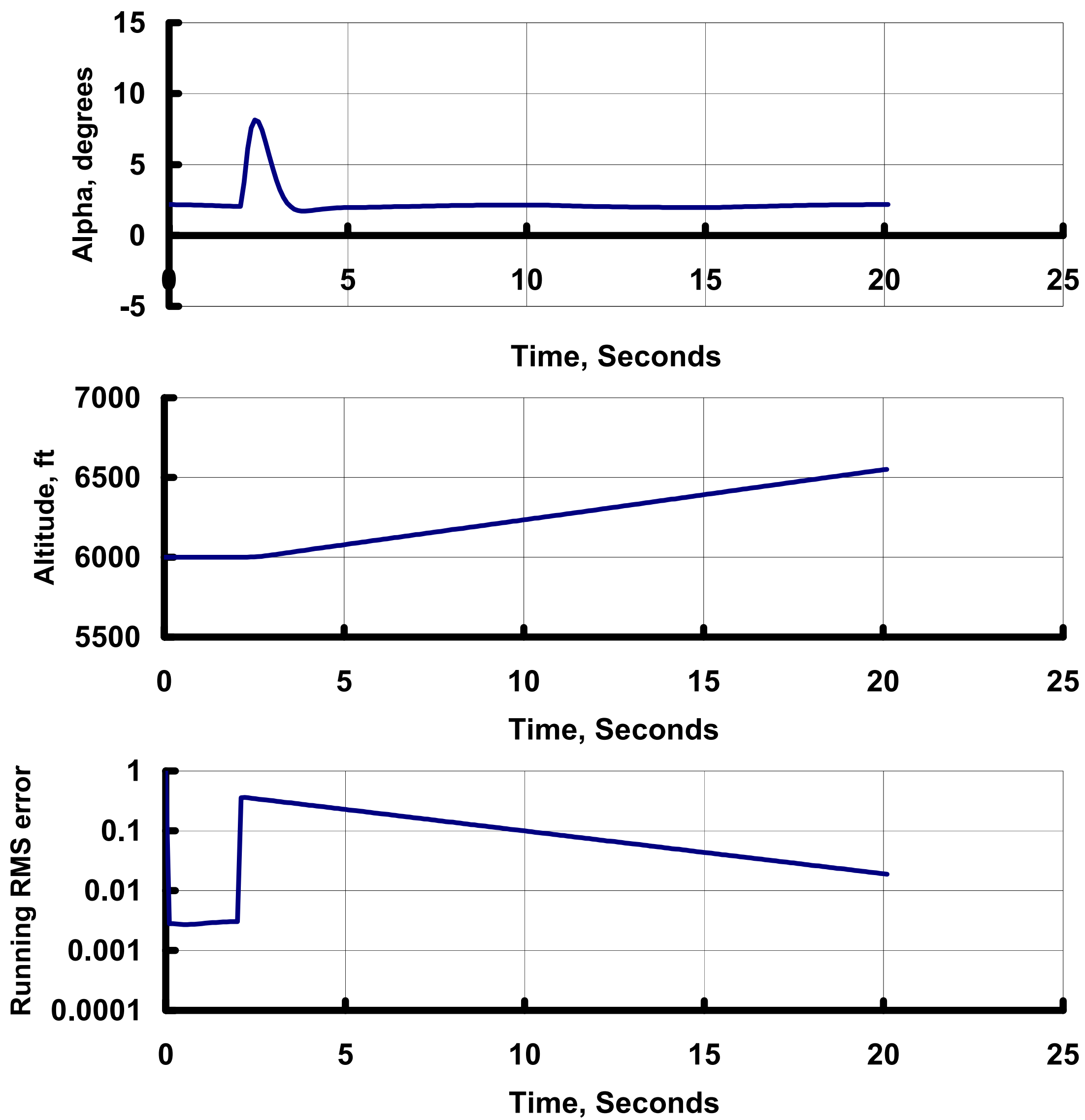


Figure 43 (c): Case 7 time history

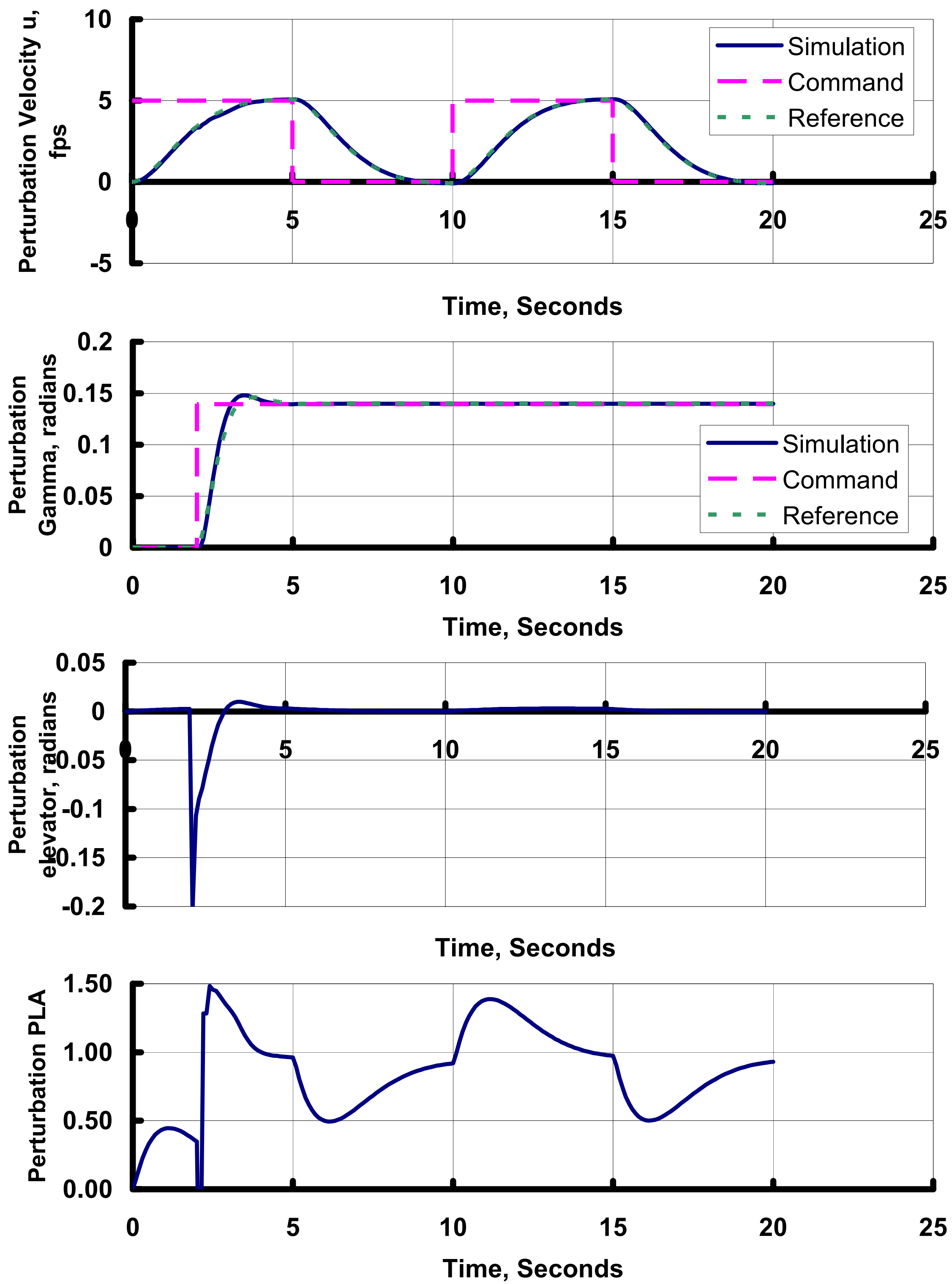


Figure 44 (a): Case 8 time history

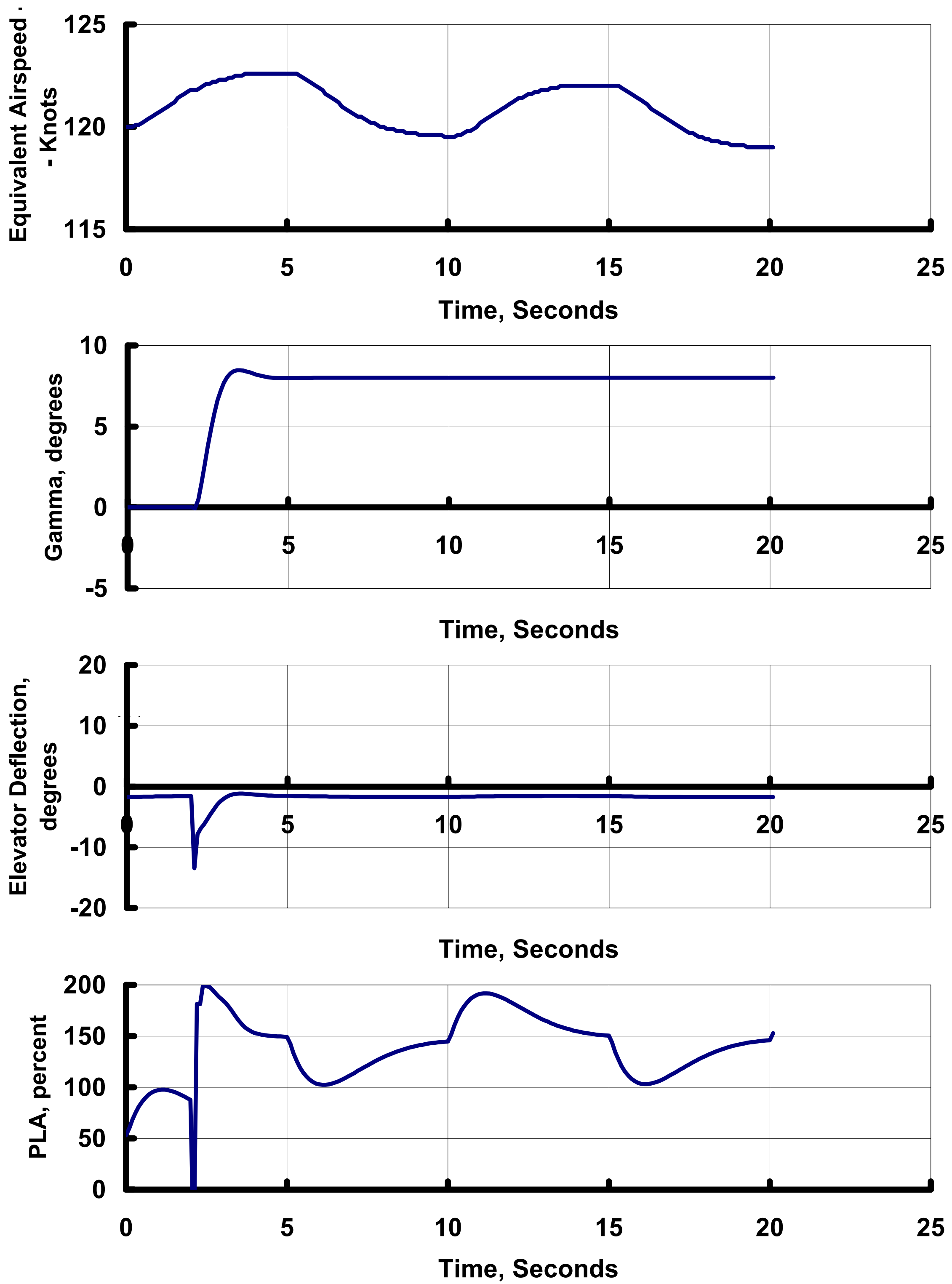


Figure 44 (b): Case 8 time history

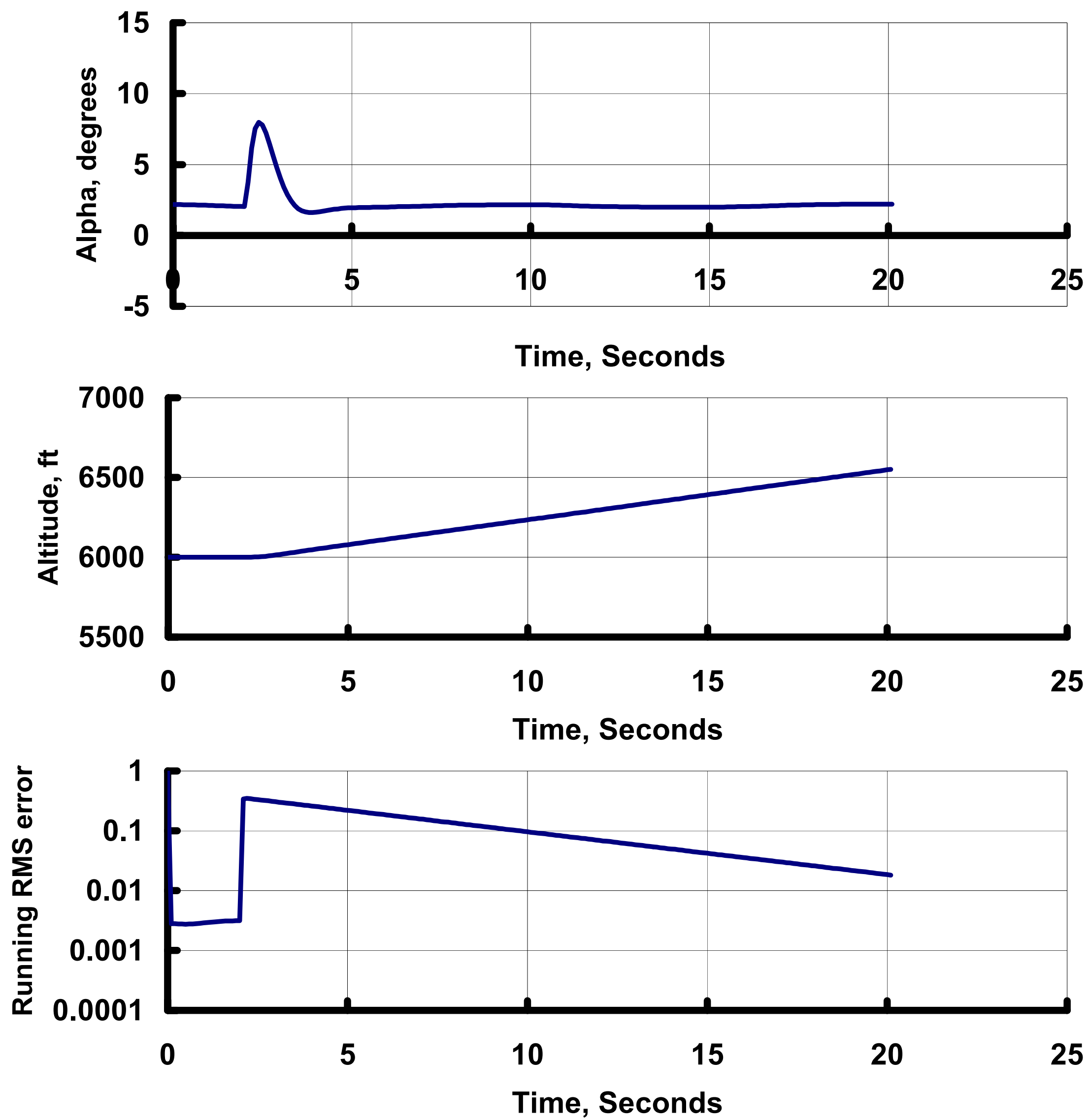


Figure 44 (c): Case 8 time history

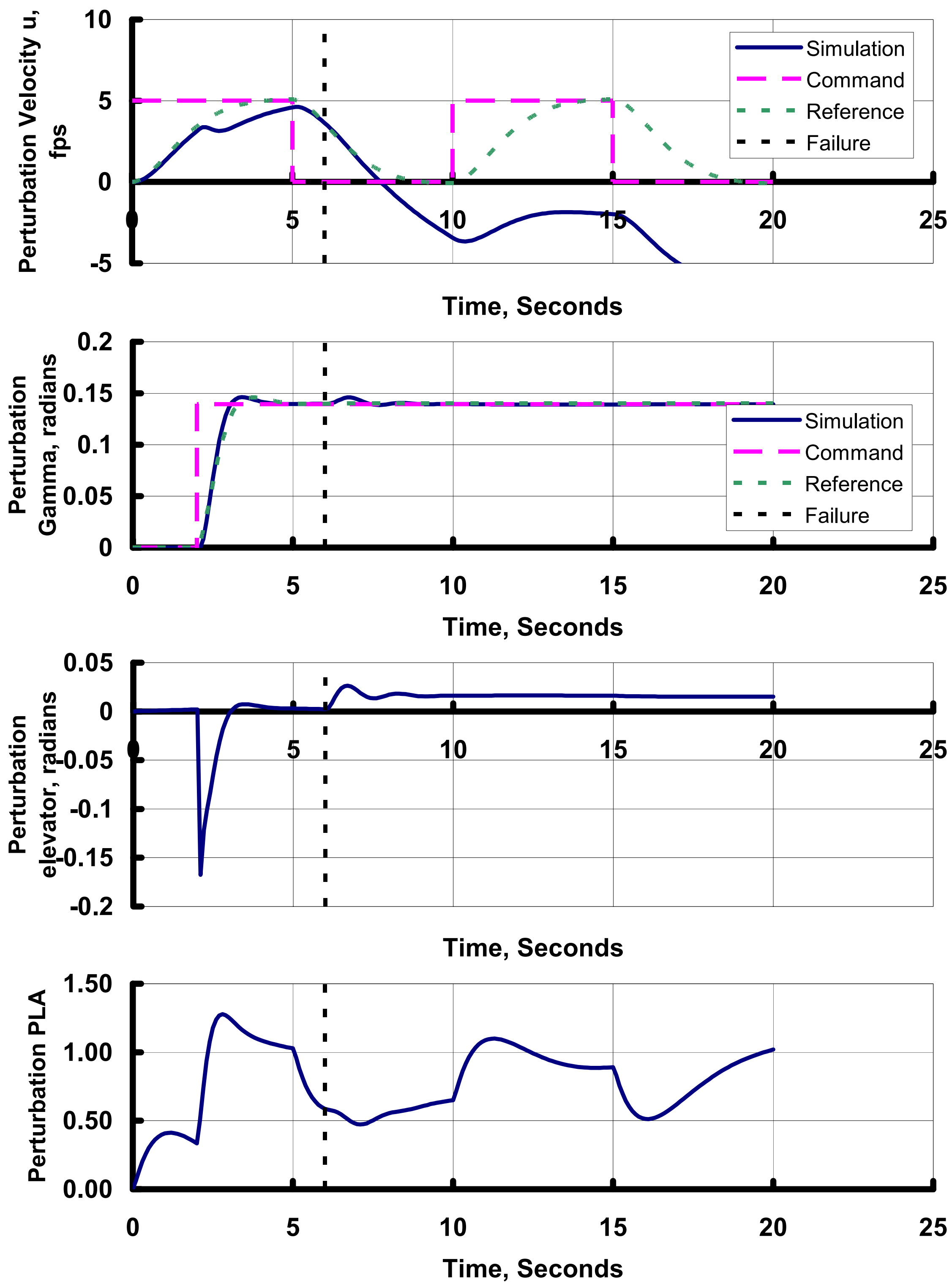


Figure 45 (a): Case 9 time history

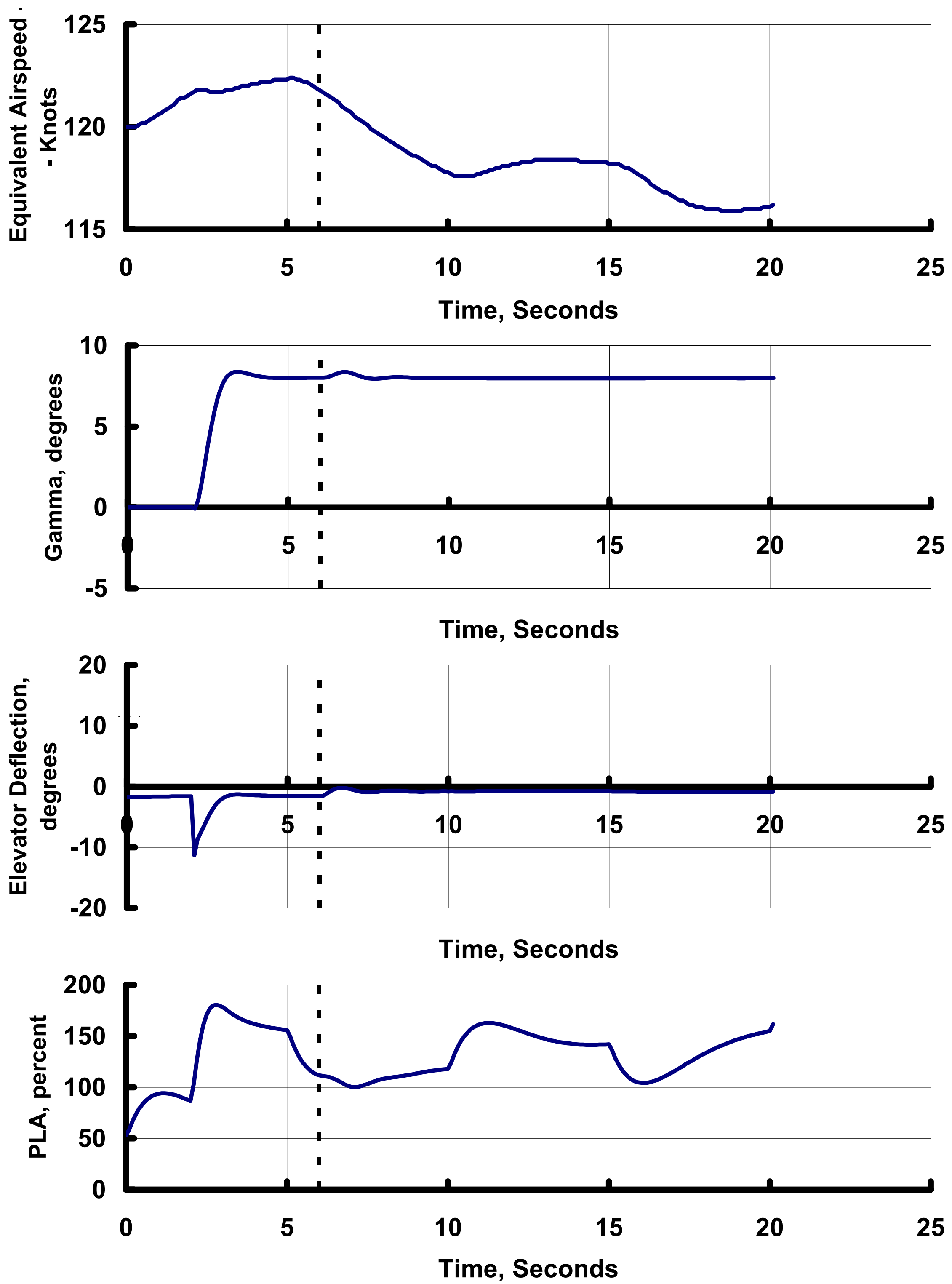


Figure 45 (b): Case 9 time history

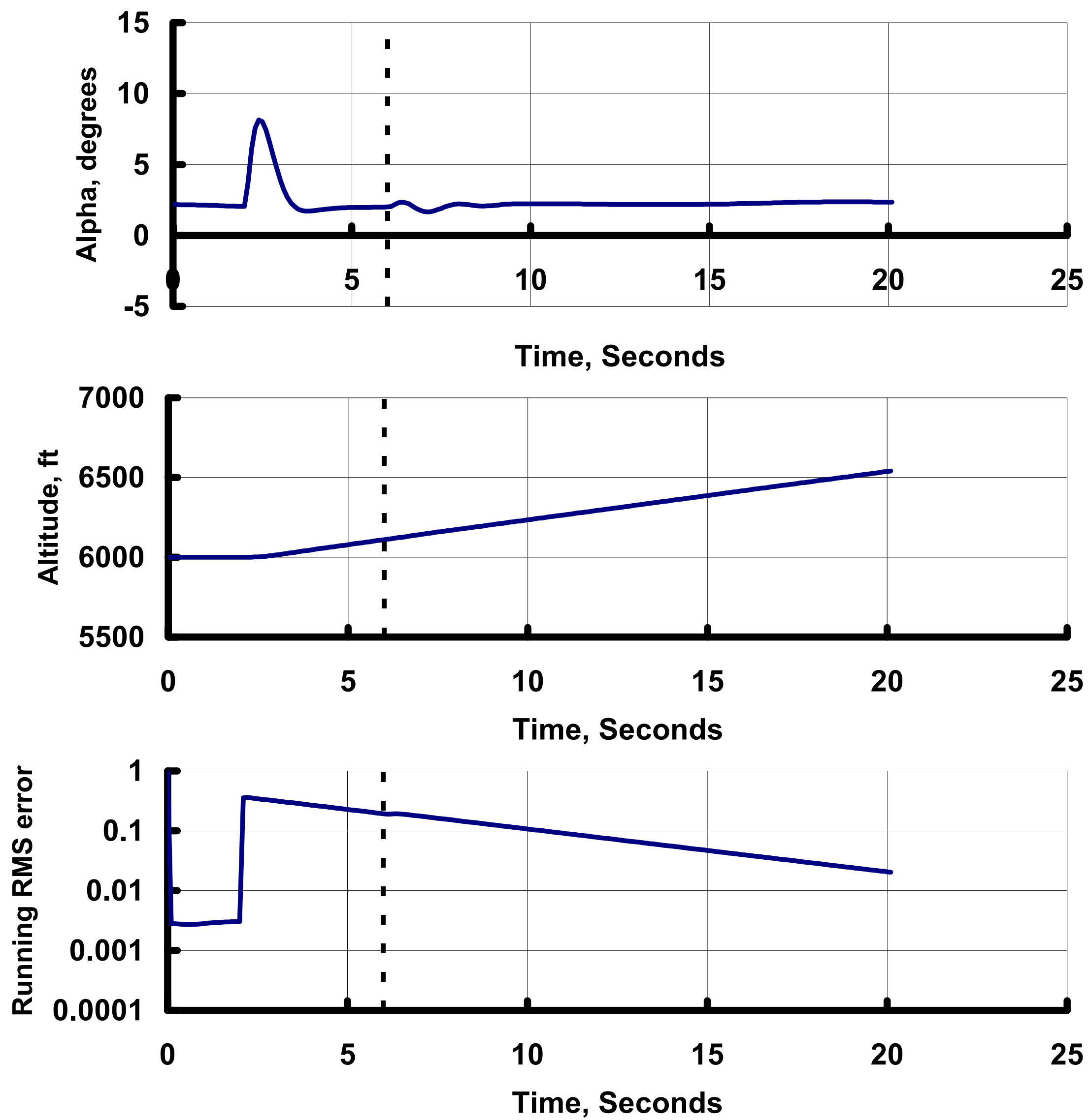


Figure 45 (c): Case 9 time history

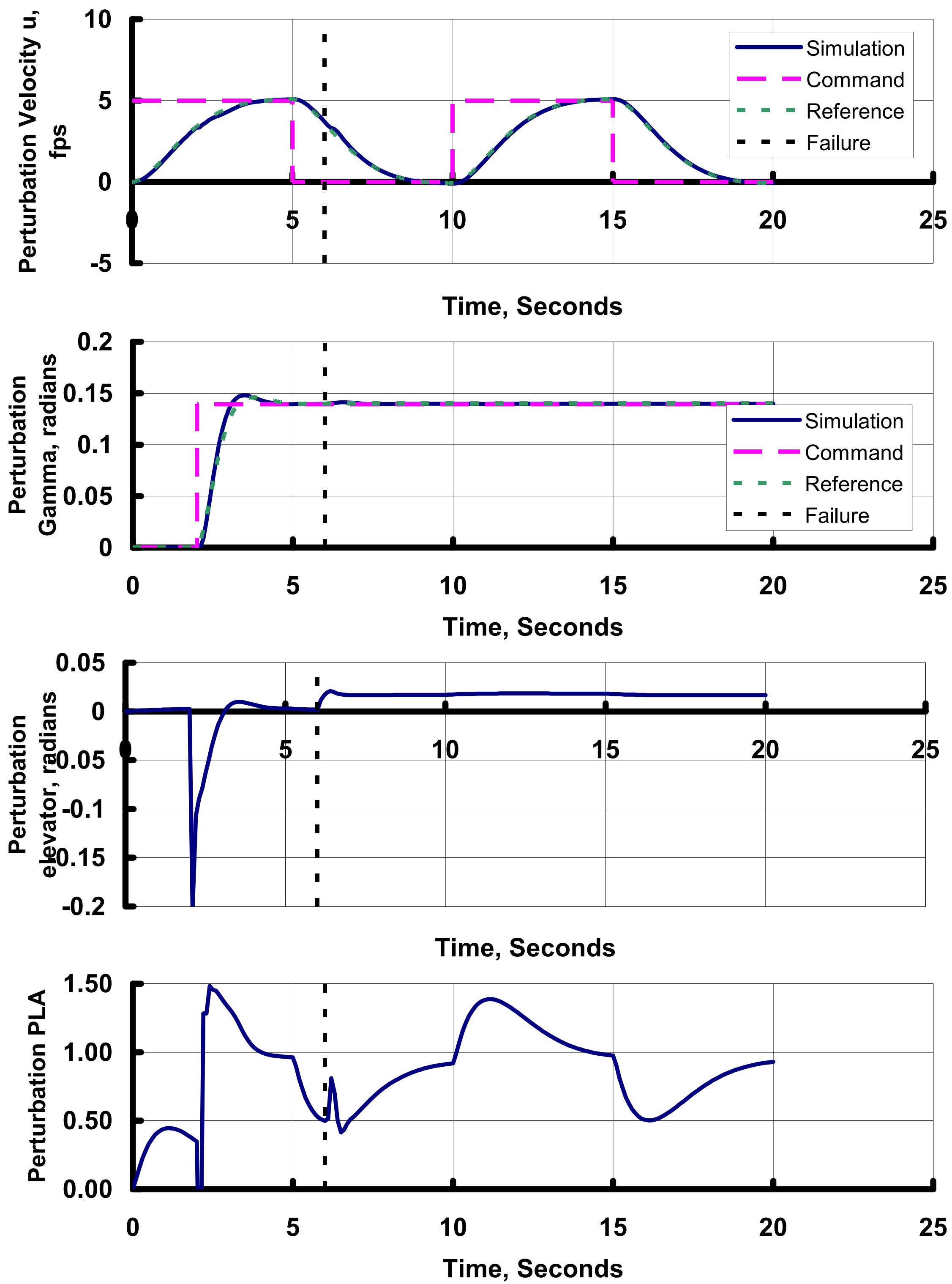


Figure 46 (a): Case 10 time history

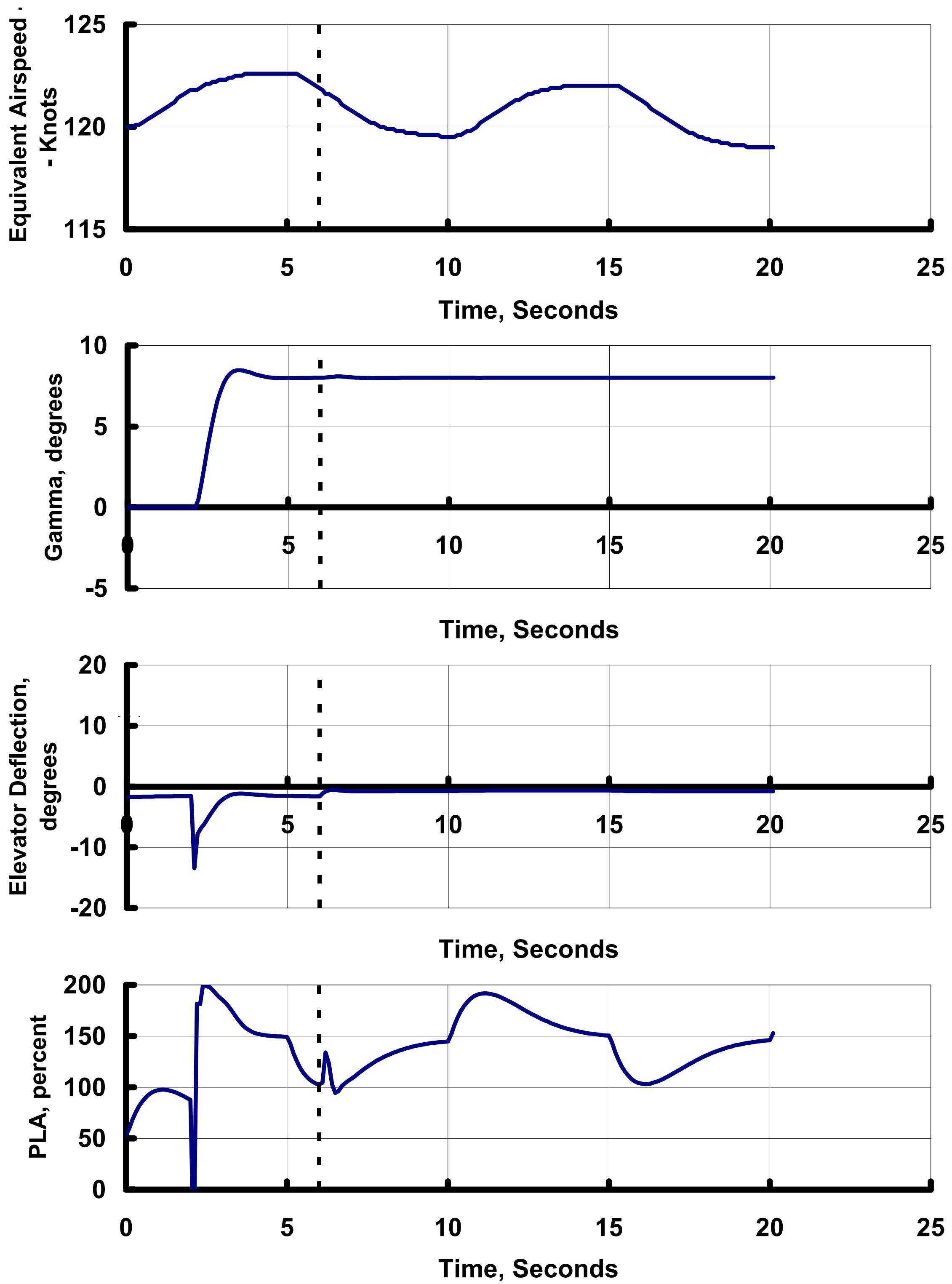


Figure 46 (b): Case 10 time history

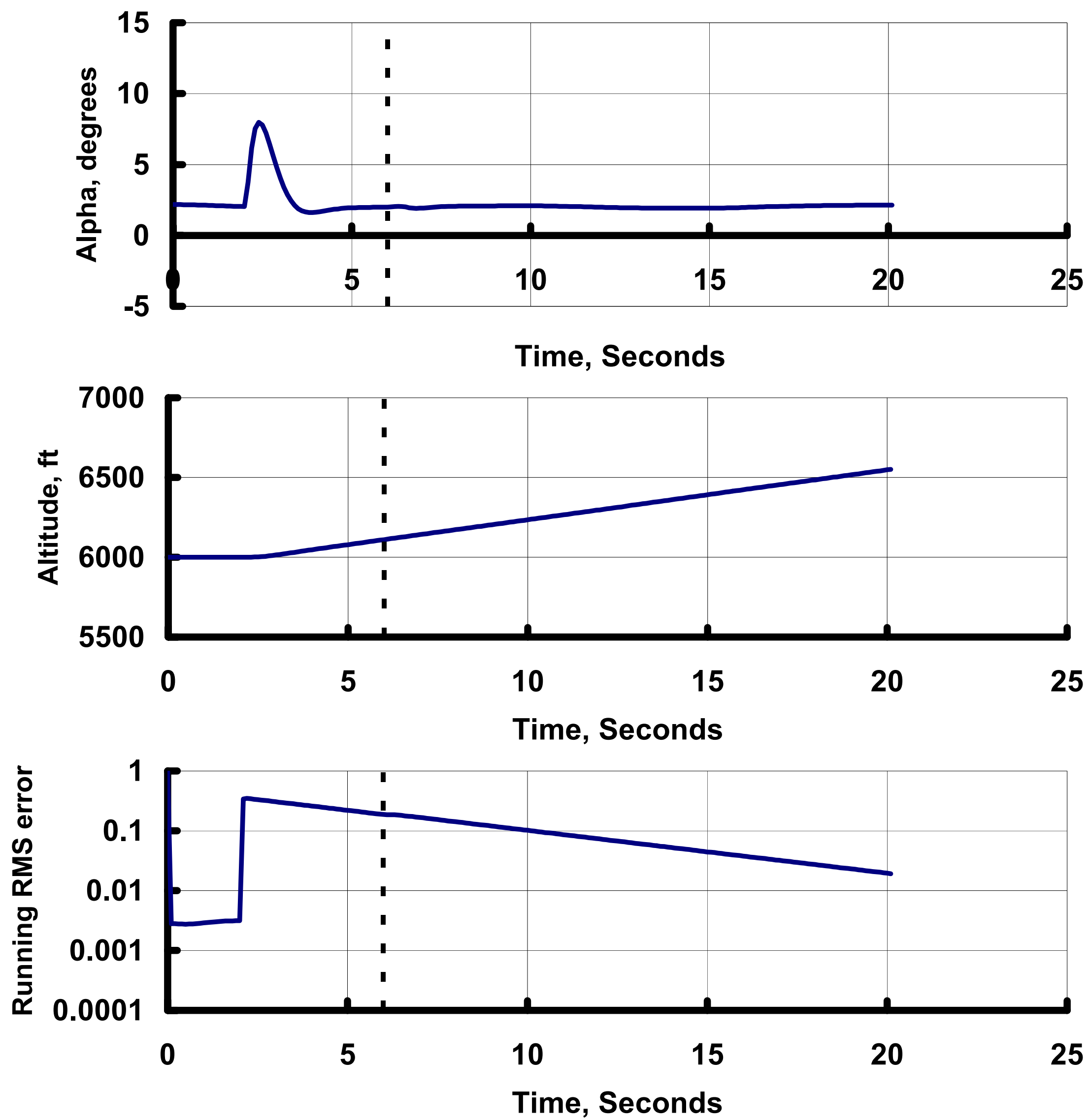


Figure 46 (c): Case 10 time history

Cases 11 through 14 (Figures 47 through 50) reproduce failure scenarios examined by Steck [13]. The neural network is on at all times during these four cases to compensate for failures.

Case 11 represents maintaining level flight with the insertion of two failures. At five seconds into the simulation, C_{M_α} is reduced by ten percent. Ten seconds later, throttle gain is reduced by 25 percent. Vertical dashed lines in Figure 47 mark the time of the failures. The current controller adapts to these failures with very little transient behavior. Speed variations are less than one knot, and gamma variations are less than one degree. A small change in trim elevator setting is evident when the stability change occurs at six seconds. An increase in PLA is observed at fifteen seconds to offset the reduction in throttle gain. Learning is evident from the RMS error chart. Step changes in learning occur at the first and second failure.

Case 12 involves a pitch-up command at eight seconds to a five-degree flight-path angle, followed by a 20 feet/sec step command in speed at 20 seconds. The same failure sequence occurs as in Case 11. Once again, no significant deviations from the commanded speed and flight-path angle are observed. Similar behaviors for elevator and PLA are seen as in Case 11 as the system adapts to the failure. Learning is evident at the time of the failures based on the step changes in RMS error.

Case 13 starts with an immediate failure at time zero that reduces both C_{L_α} and C_{M_α} by 50 percent. The objective is to maintain steady level flight. Significant learning at time zero is evident in the RMS error trace. The transient errors after this immediate failure are somewhat larger with this scenario but still less than one knot in airspeed and less than one degree in flight-path angle.

Case 14 (Figure 50) is a somewhat extreme scenario. A step decrease in speed of 70 feet/sec is commanded at three seconds. Failures involve a ten percent reduction in C_{M_α} at twenty seconds and throttle gain reduction of 25 percent at 25 seconds. This is an unrealistic scenario in that the final commanded speed is less than the stall speed for the real aircraft. None the less, the controller is able to achieve the desired speed reduction. This is an interesting case in that the maneuver causes the airplane to reach the angle of attack limiter implemented in the simulation. Almost simultaneously with reaching the target speed reduction at about eight seconds, the angle of attack reaches the 14 degree limit. Elevator and PLA traces are seen to have a change in character when the controller switches to compliance with the 14 degree angle of attack limit. Examination of the angle of attack trace shows good adherence to the 14 degree limit. The available lift coefficient at 14 degrees angle of attack is not quite sufficient to maintain equilibrium in lift, and the airplane is observed to begin a descent. Flight-path angle increases negatively, and the altitude begins to decrease. The failures seem almost anti-climactic after the speed reduction. However, the controller is observed to achieve a new steady-state elevator solution after the stability failure and a new PLA setting after the throttle gain failure. Even with the controller "distracted" by observing the angle of attack limit, it does a reasonable job of tracking speed in this very low-speed maneuver.

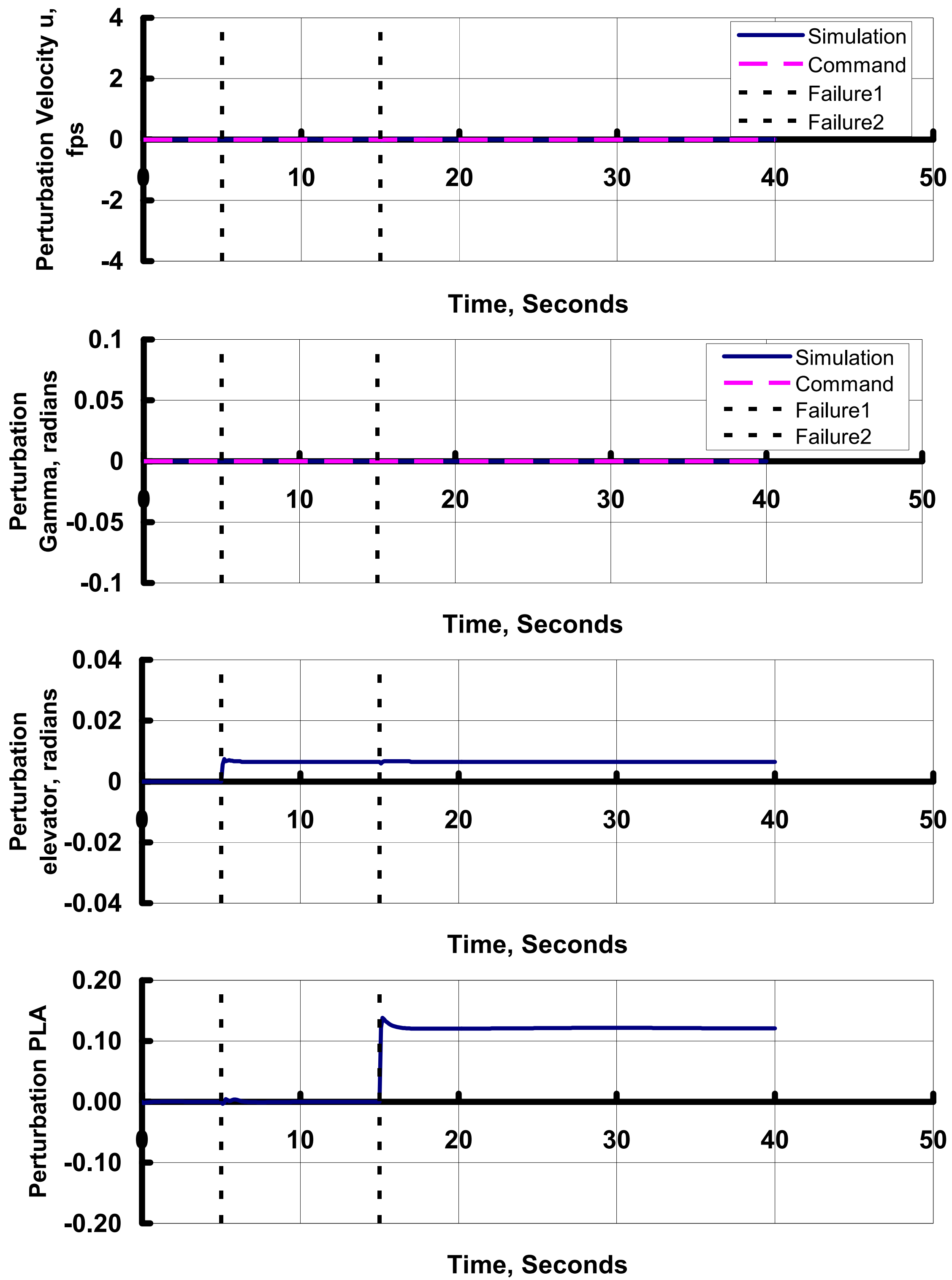


Figure 47 (a): Case 11 time history

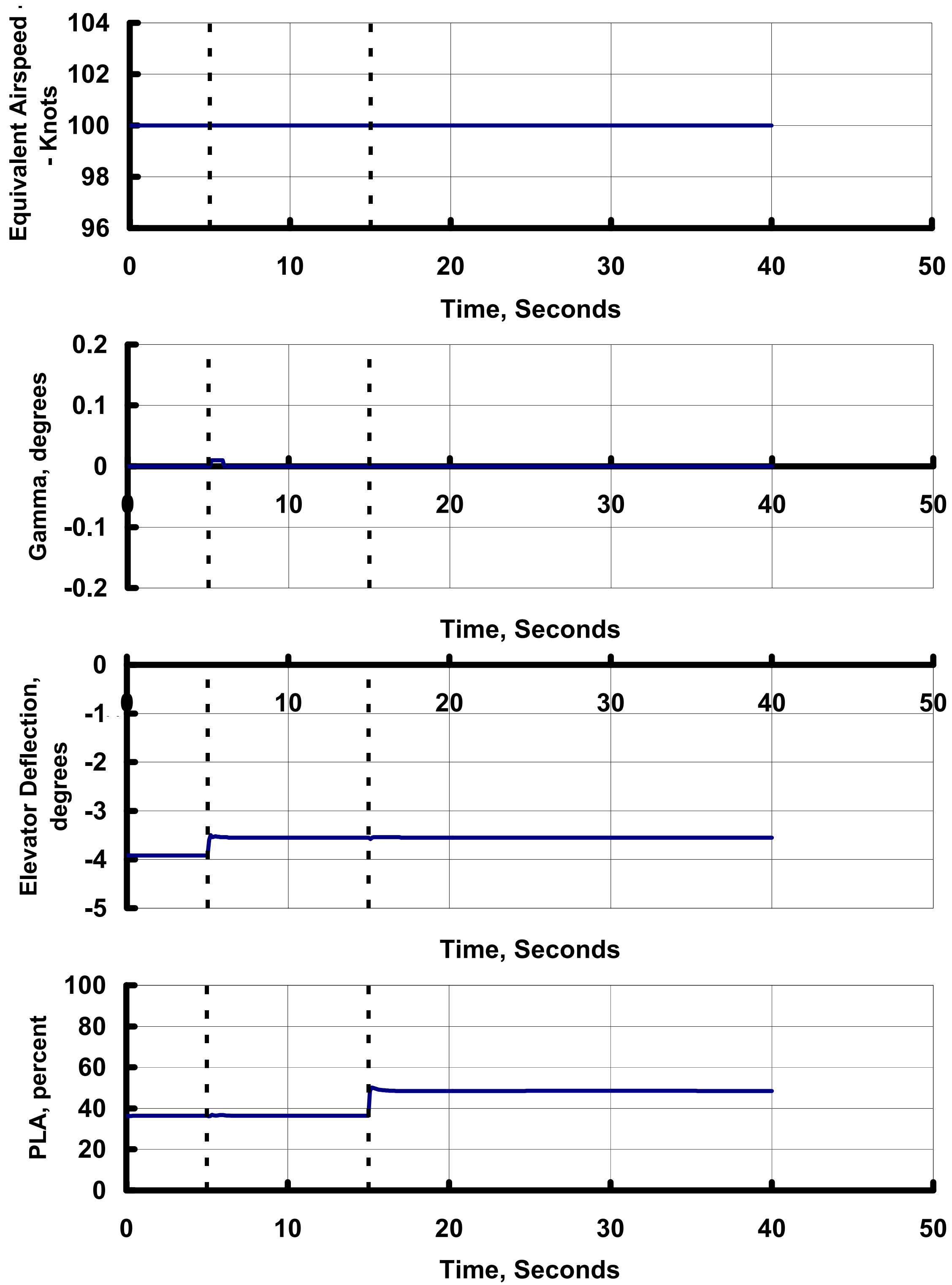


Figure 47 (b): Case 11 time history

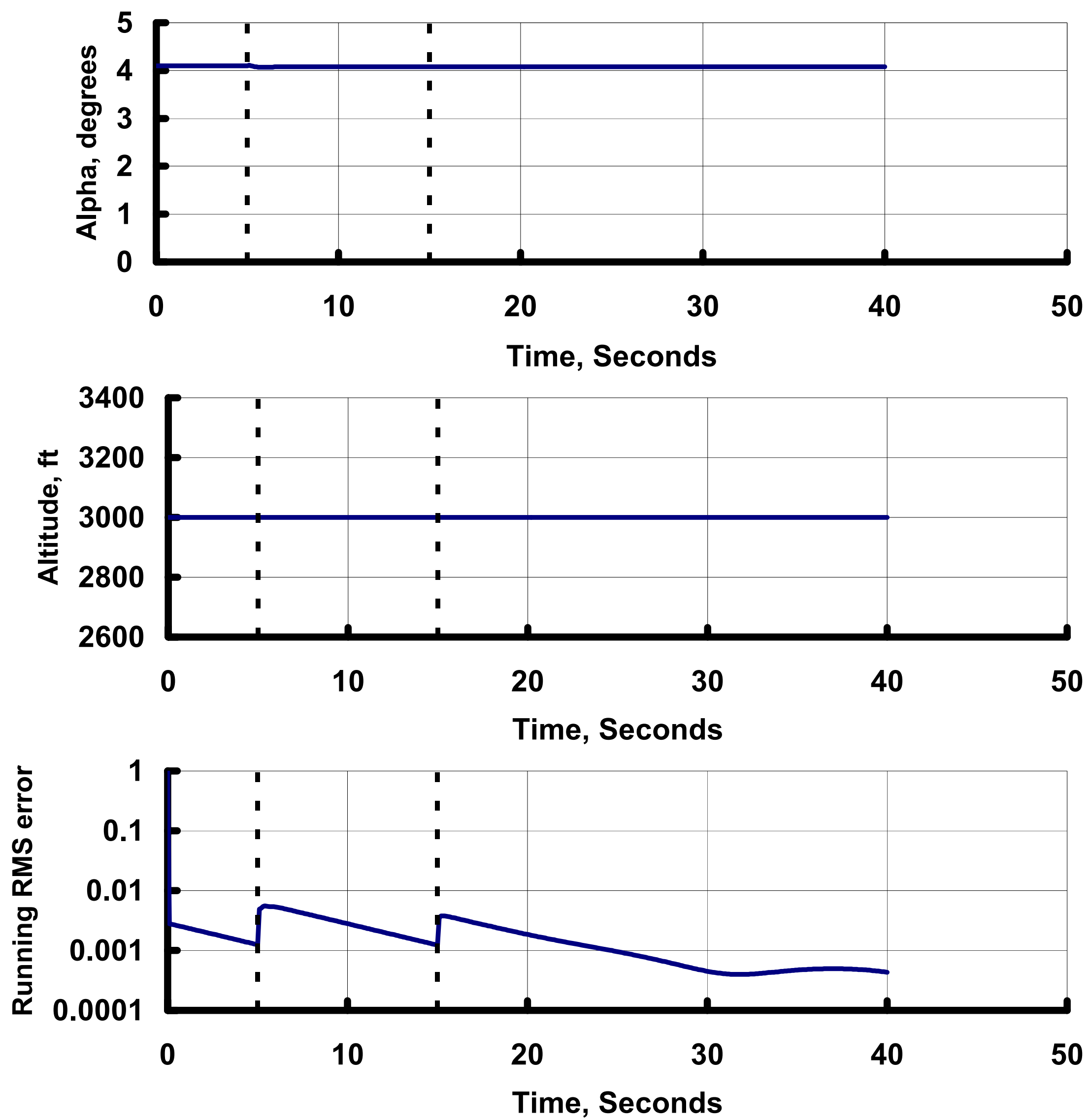


Figure 47 (c): Case 11 time history

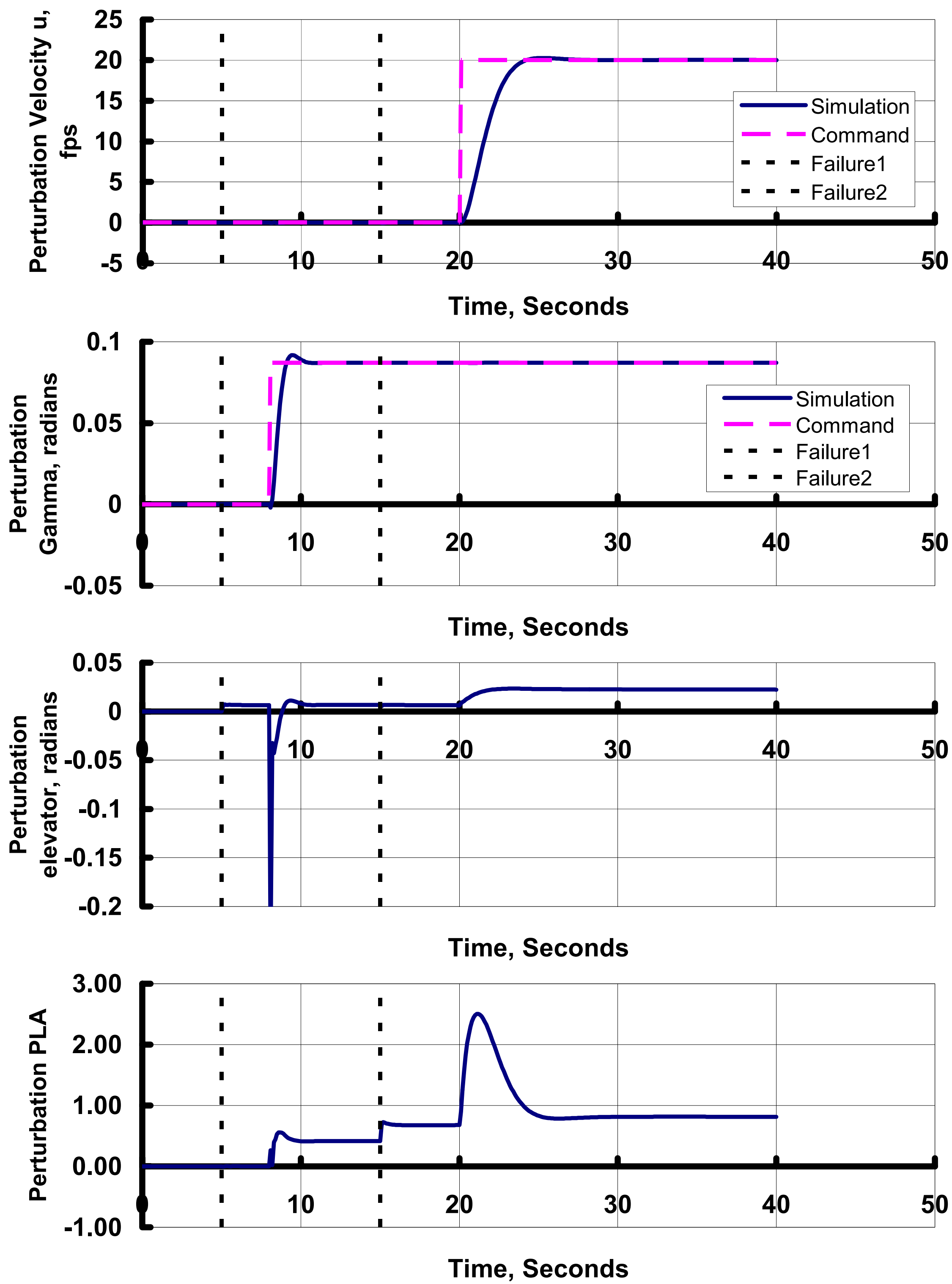


Figure 48 (a): Case 12 time history

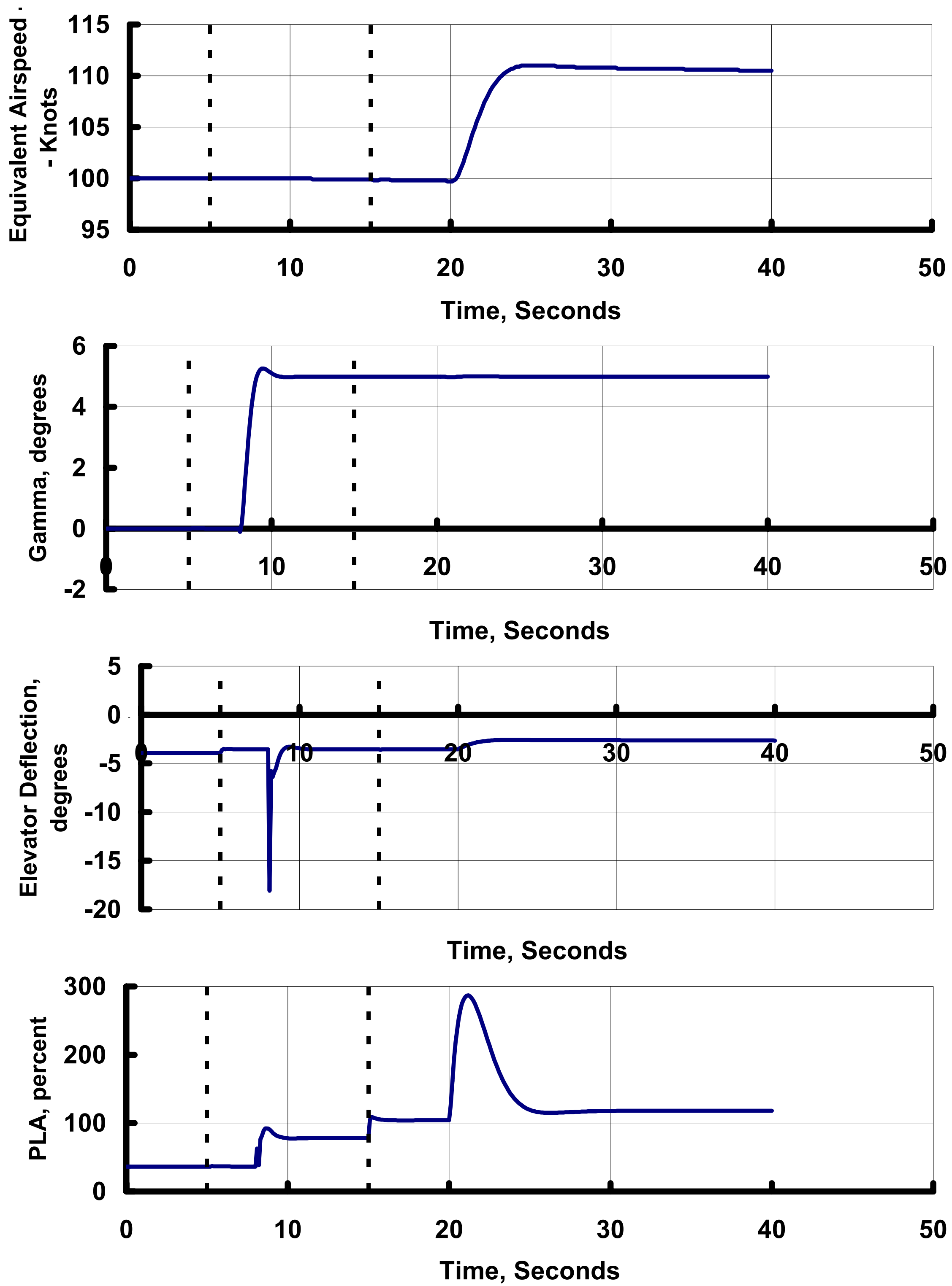


Figure 48 (b): Case 12 time history

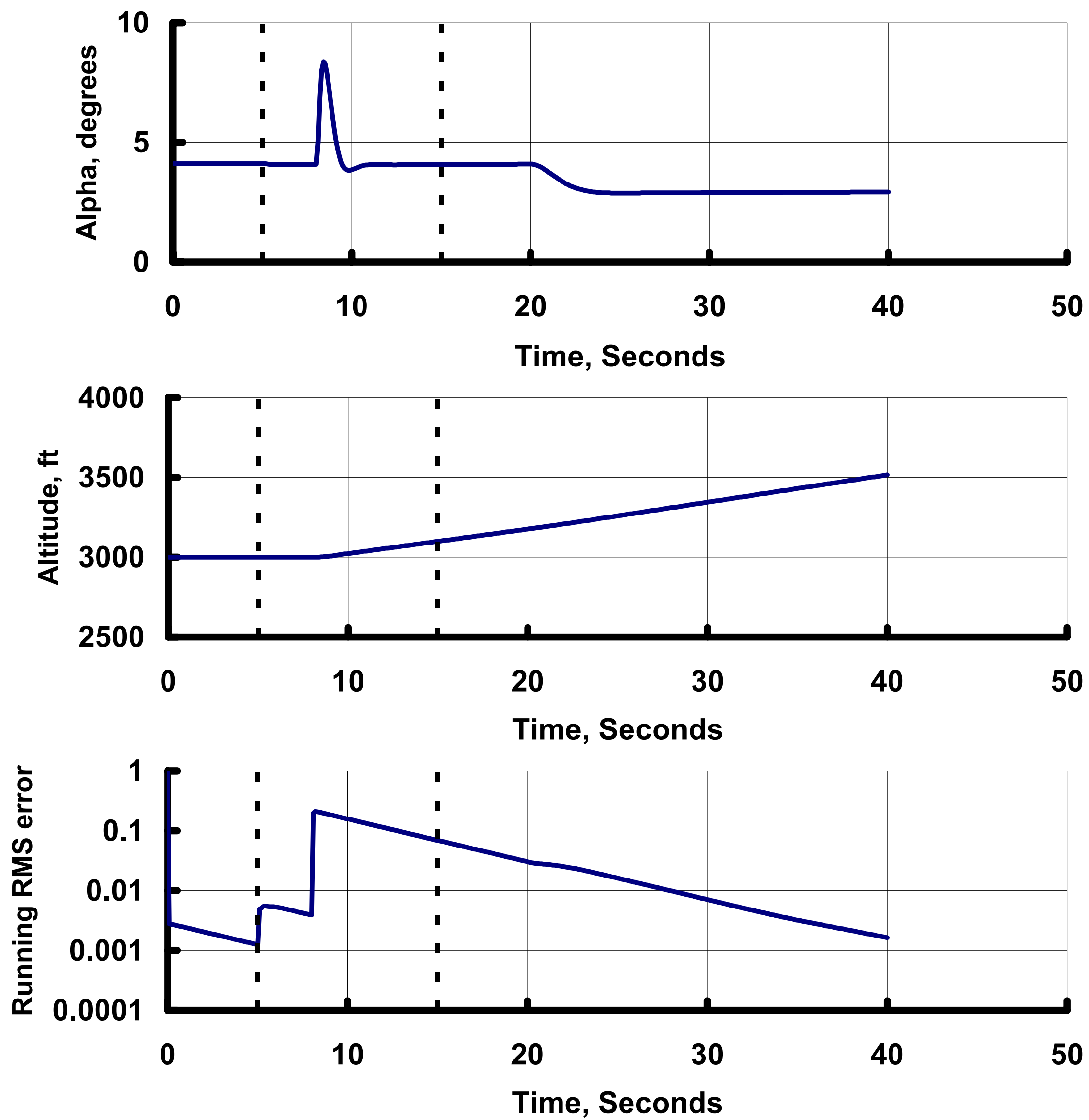


Figure 48 (c): Case 12 time history

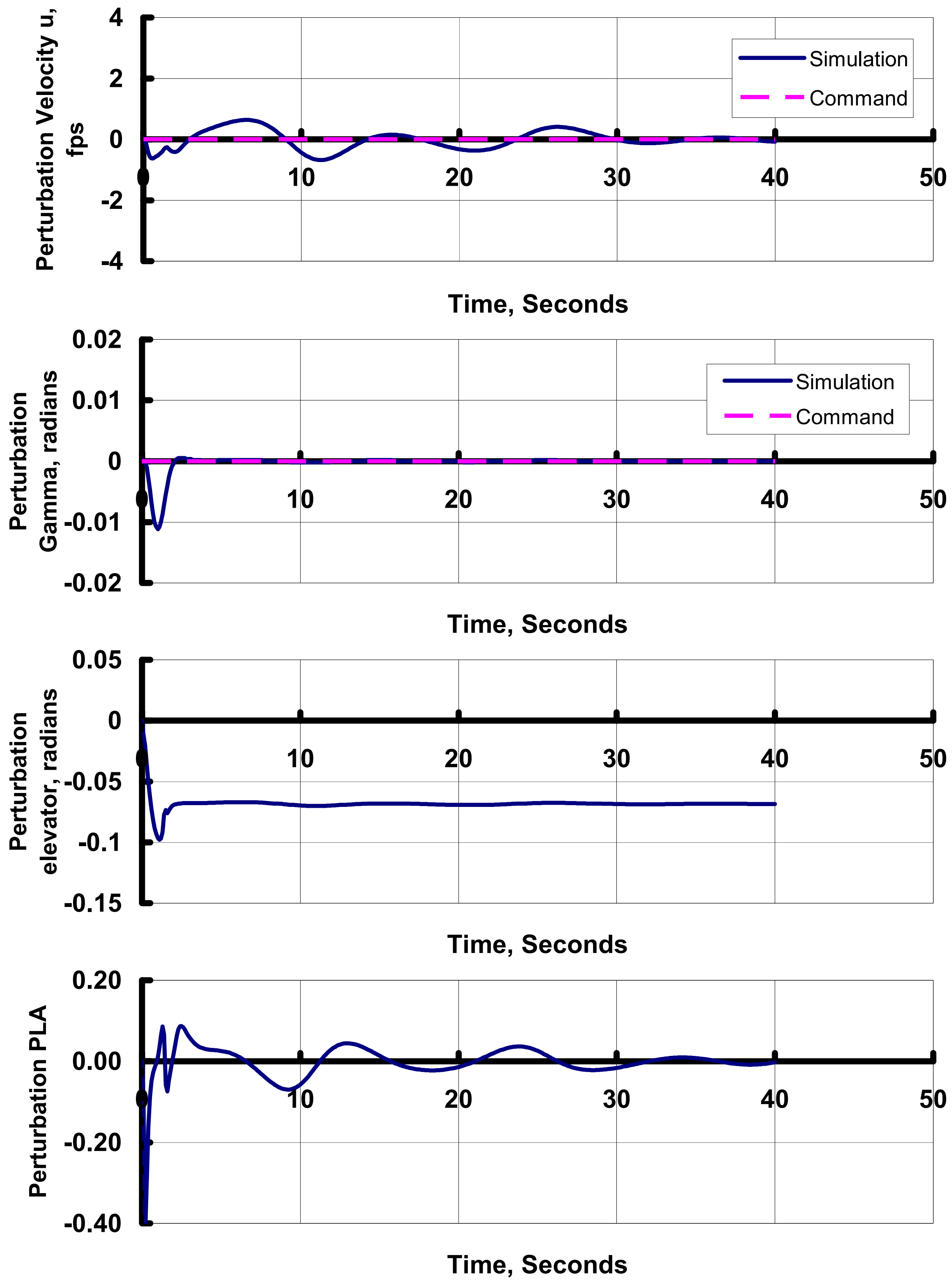


Figure 49 (a): Case 13 time history

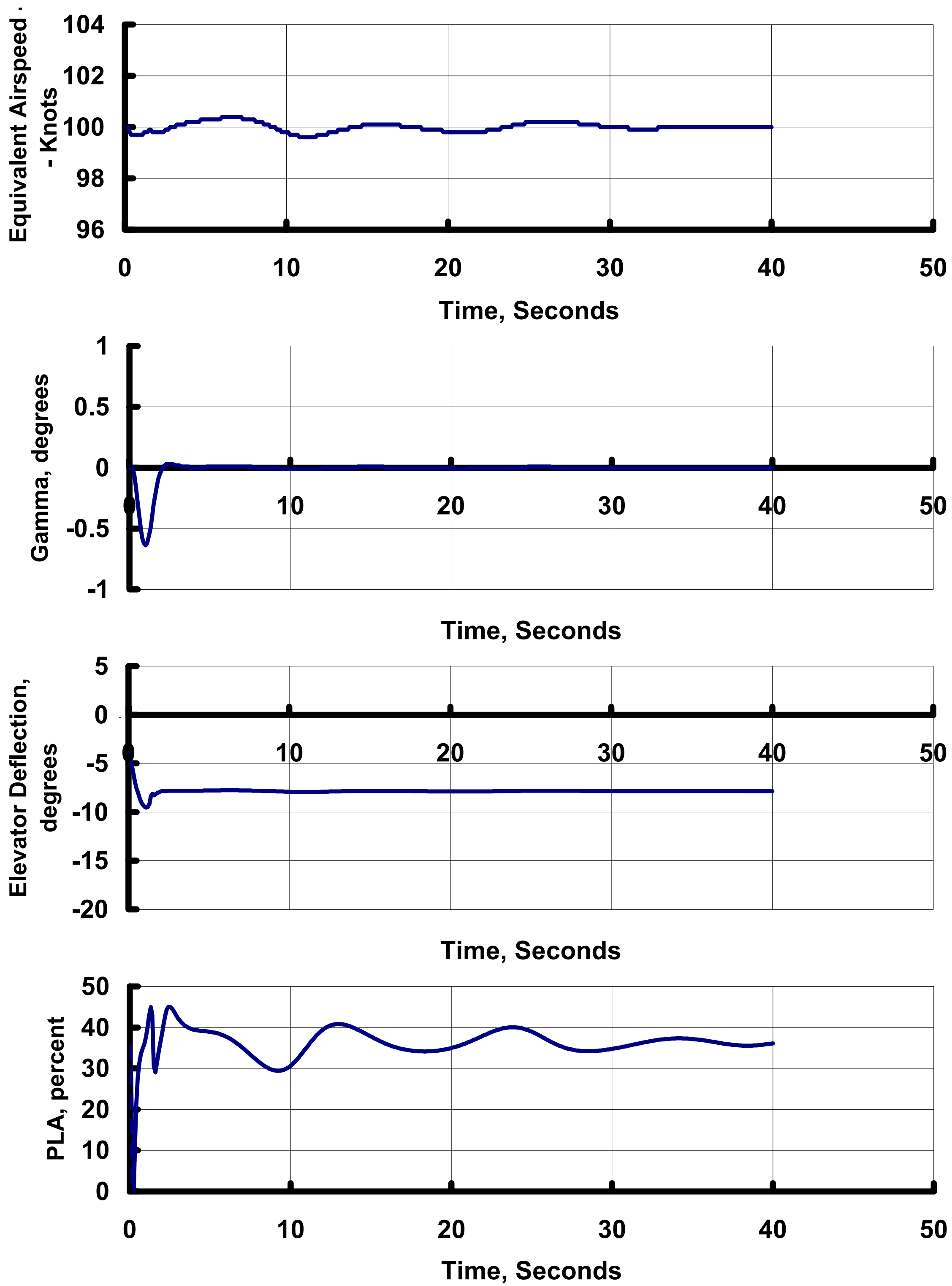


Figure 49 (b): Case 13 time history

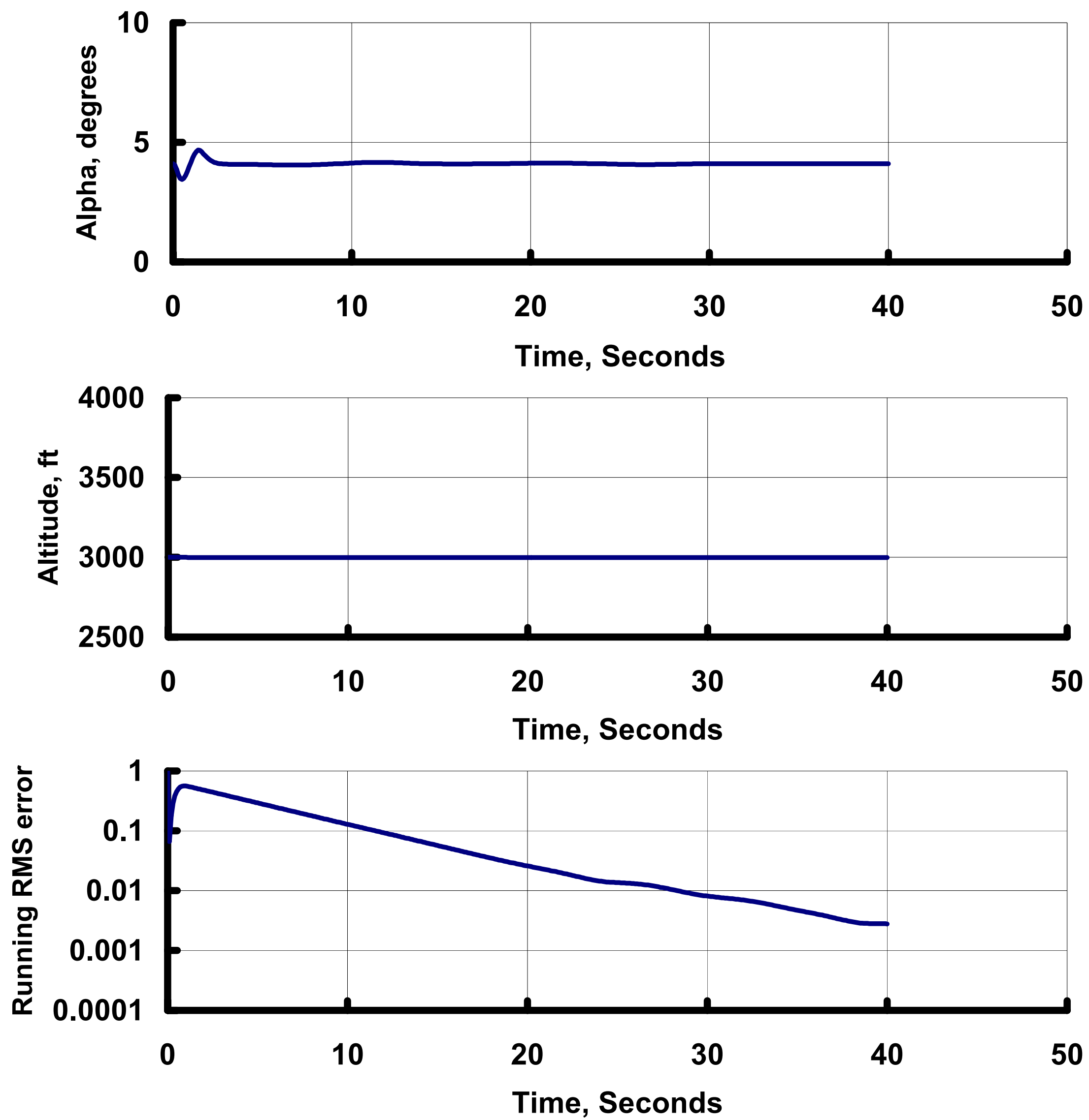


Figure 49 (c): Case 13 time history

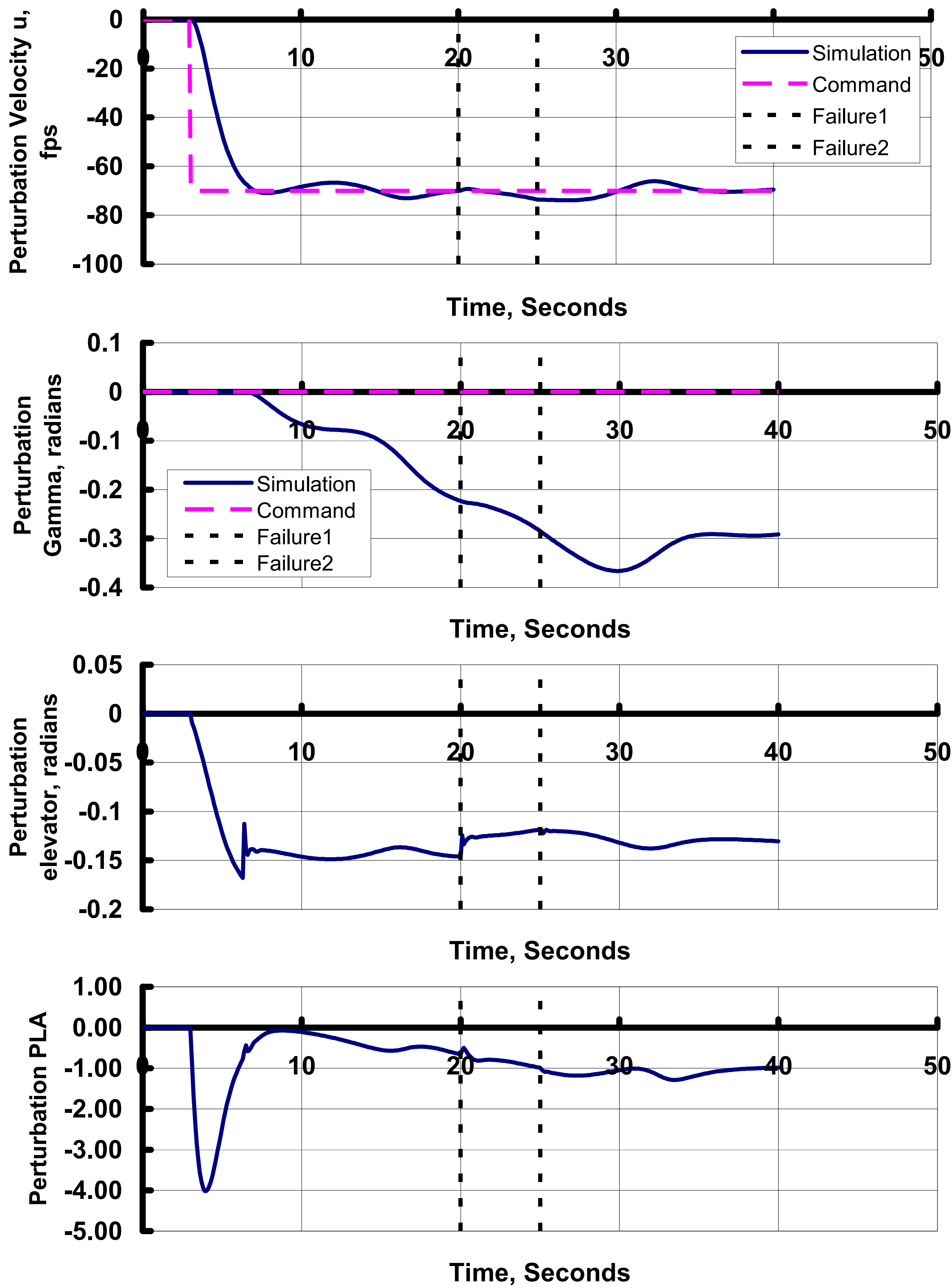


Figure 50 (a): Case 14 time history

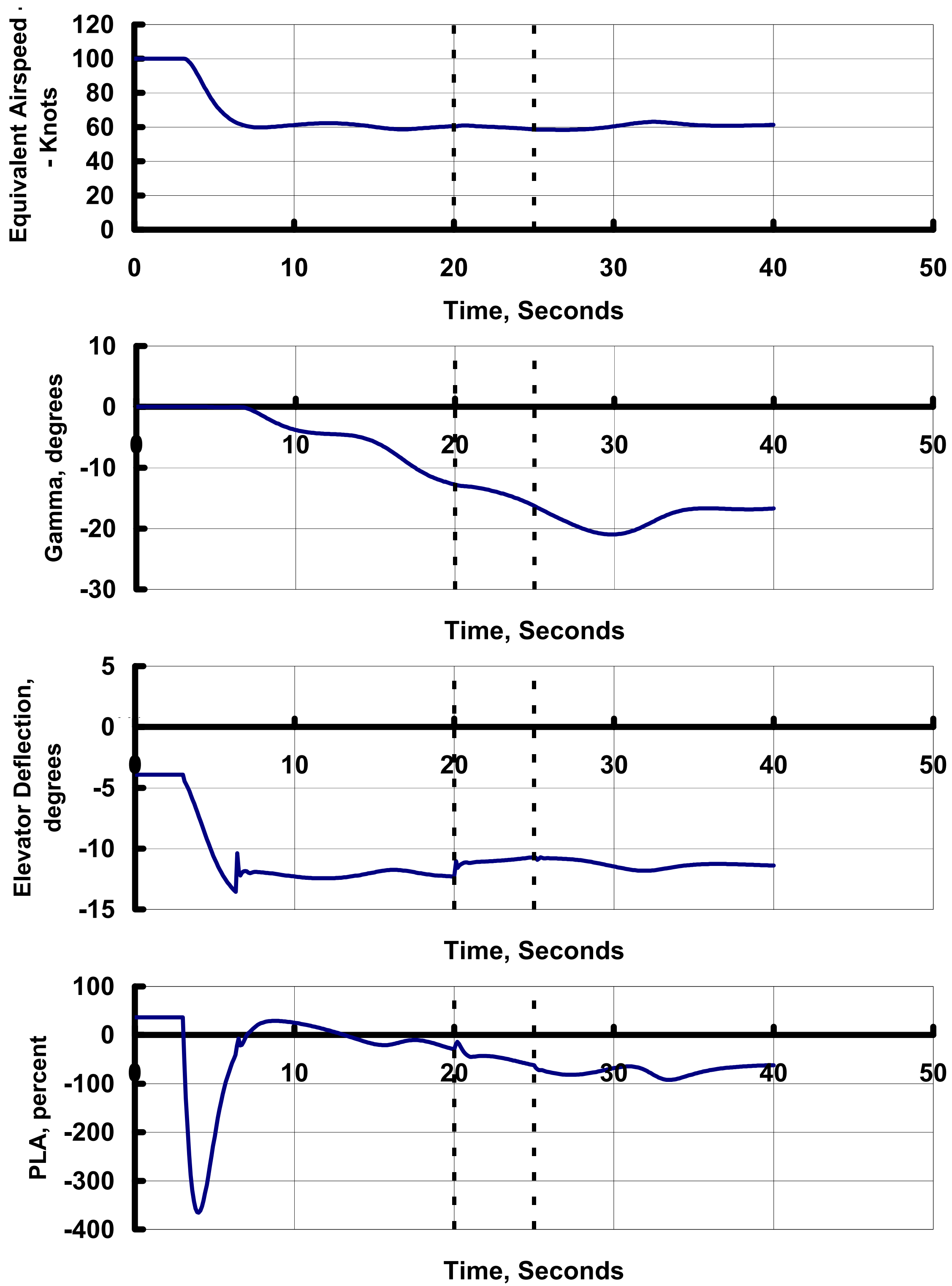


Figure 50 (b): Case 14 time history

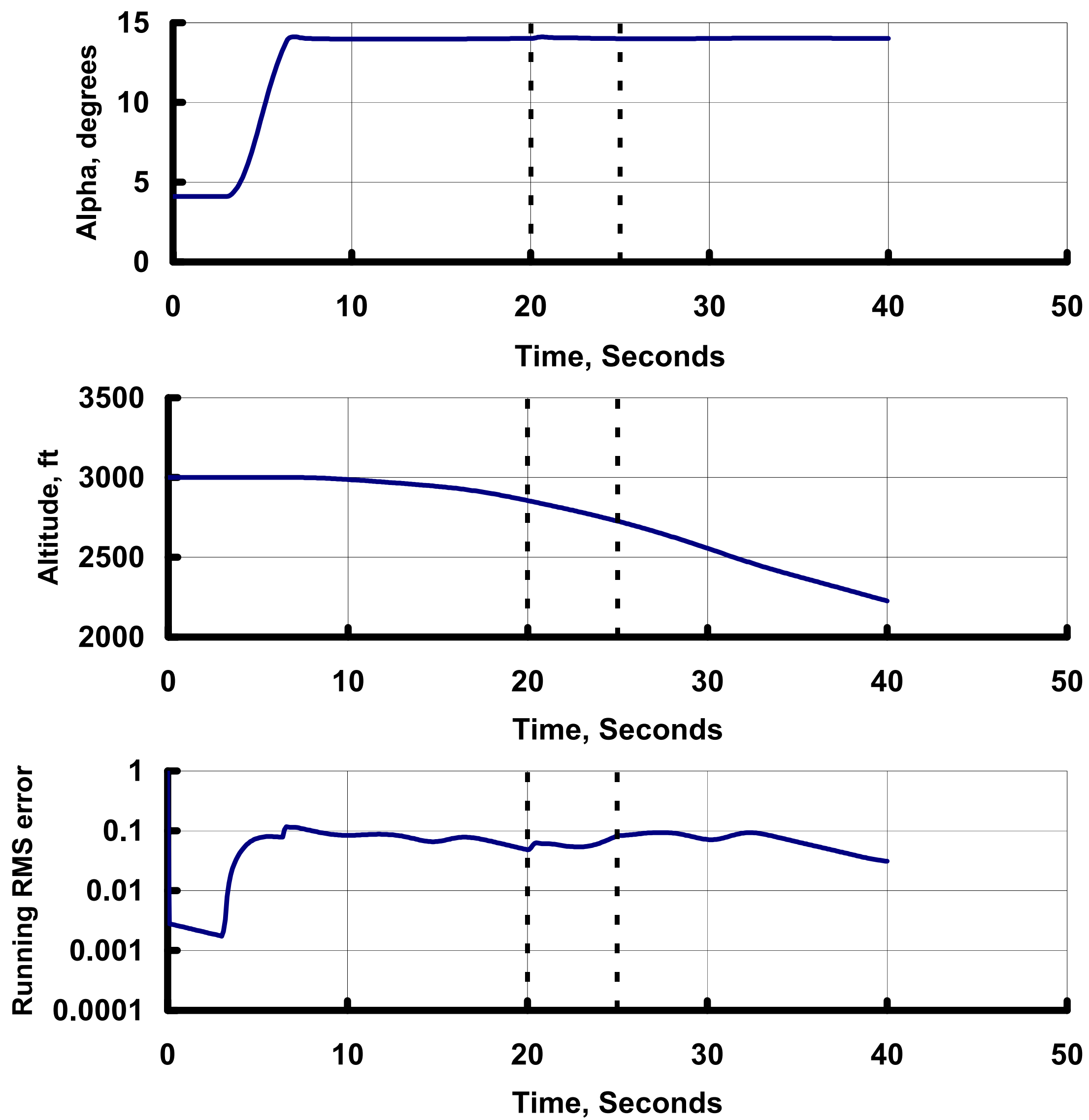
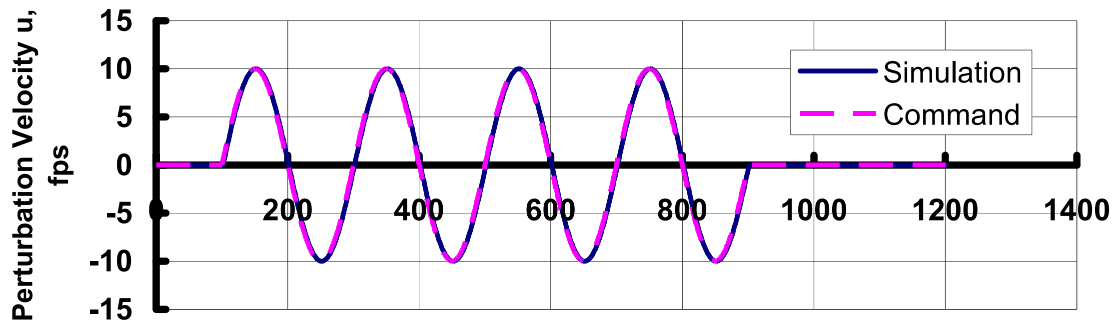
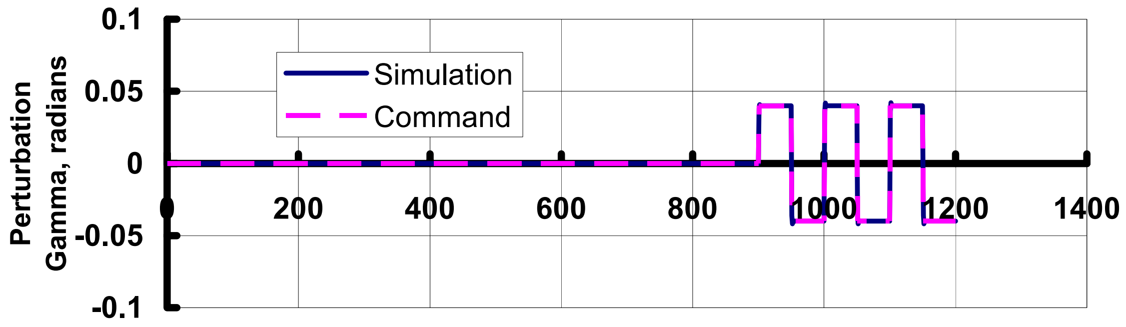


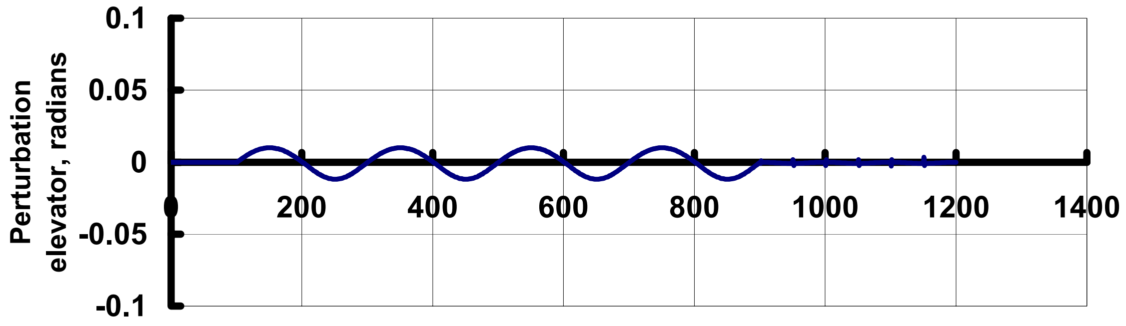
Figure 50 (c): Case 14 time history

Case 15 (Figure 51) presents a long duration maneuver to illustrate the time stability of the controller. This case is also the baseline for the cases that follow. In Case 15, the aircraft is in its nominal state with the adaptive neural network compensation on. The commanded maneuver involves an 800 second sine wave command on airspeed, with an amplitude of ten feet/sec, followed by 300 seconds of a square wave command on flight-path angle, with an amplitude of .04 radians (2.3 degrees). The nominal airplane shows good tracking behavior and crisp response to commands. The simulation speed and flight-path angle can be seen to virtually overlay the commands. As noted in prior cases, the RMS error increases when step commands in flight-path angle are received, reflecting the fact that step changes were not part of the training data used to pre-train the neural network.

Case 16 (Figure 52) is the same command set as Case 15. However, in Case 16 the airplane loses all static stability ($C_{M_\alpha} = 0$) at 400 seconds. In spite of this significant change in stability, the controller quickly adapts, stabilizes the neutrally stable airplane, and maintains the commanded airspeed and flight-path angle. There are no visible transients in airspeed or flight-path angle. However, the failure is quite evident by examination of the elevator time history. Once again a significant change in the trim elevator setting is visible, as would be expected with a change in C_{M_α}. A spike in learning occurs subsequent to the failure, but the controller learns quickly and learning returns to a normal level within 50 seconds. A small transient in angle of attack is also evident at the time of failure.







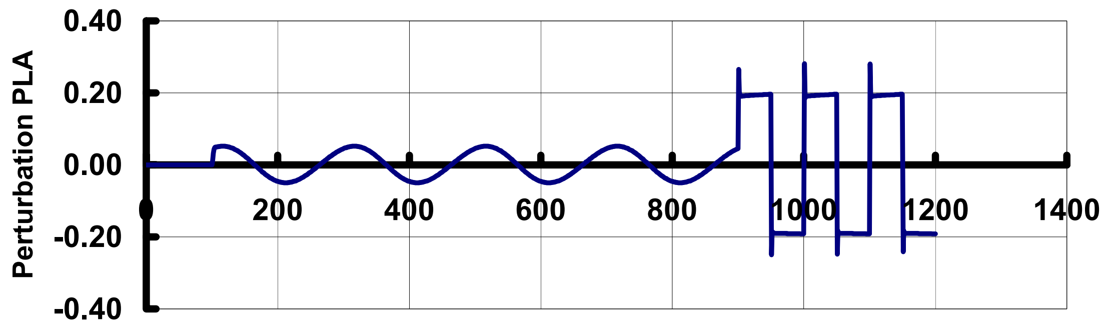


Figure 51 (a): Case 15 time history

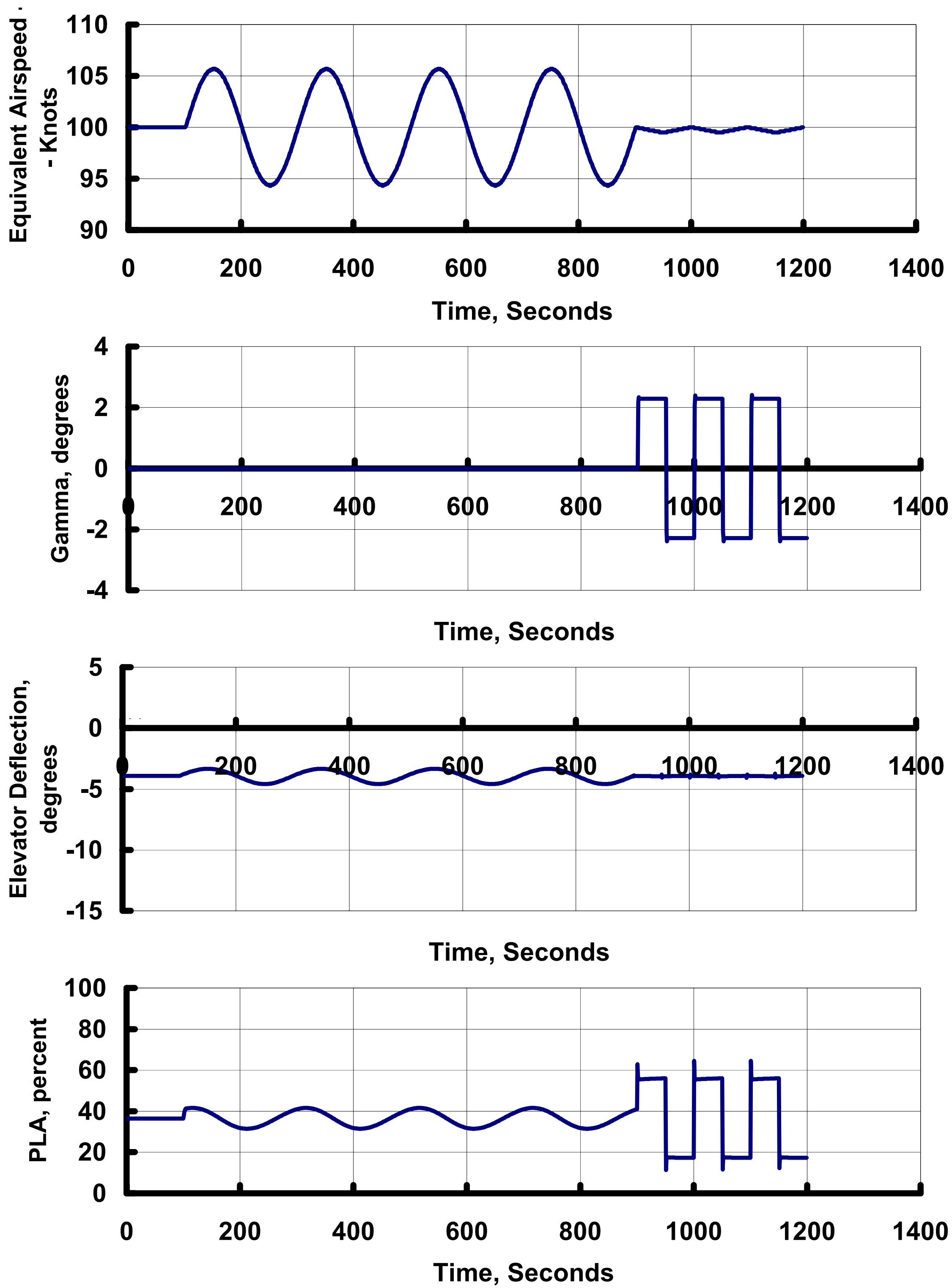


Figure 51 (b): Case 15 time history

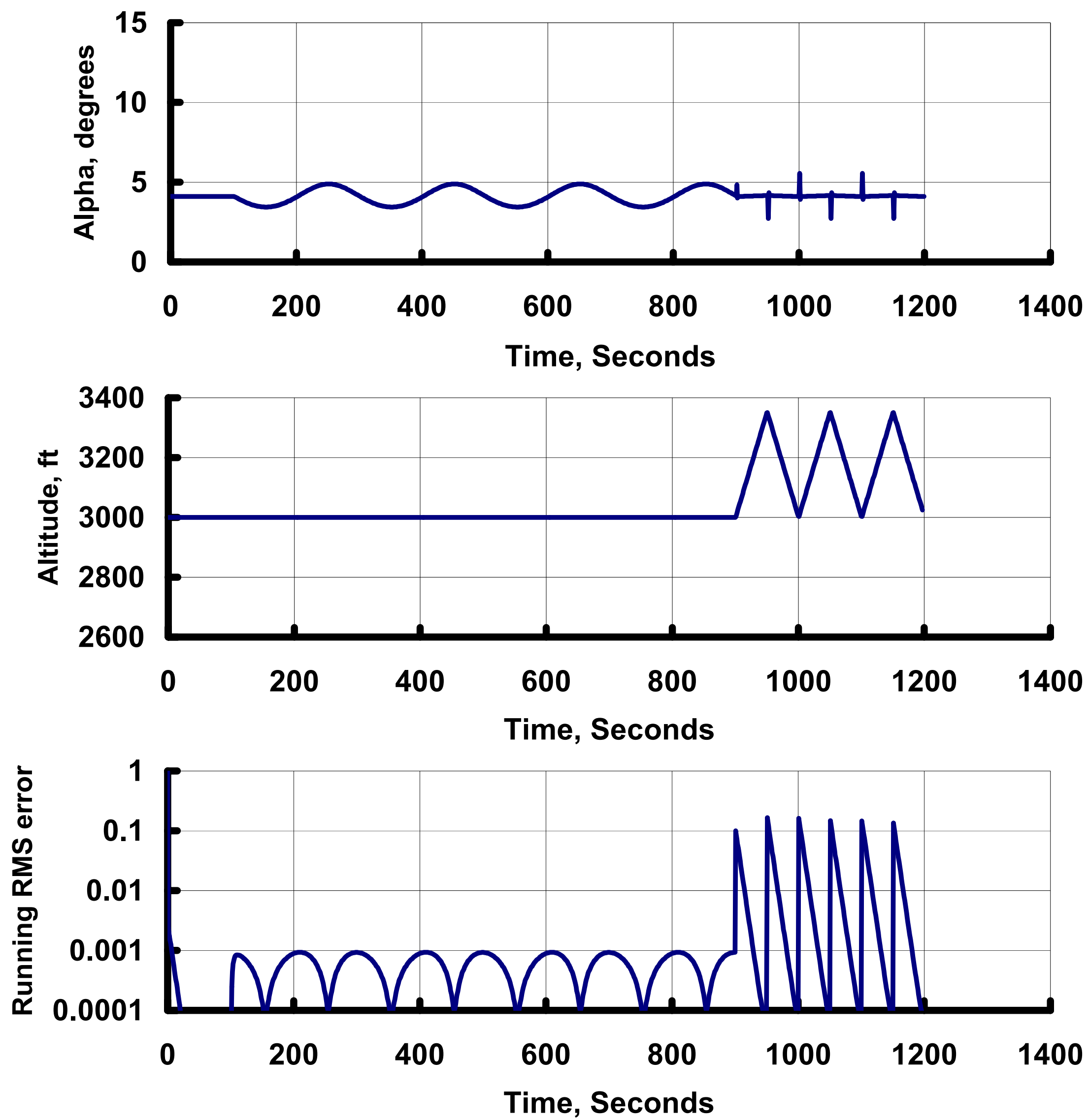


Figure 51 (c): Case 15 time history

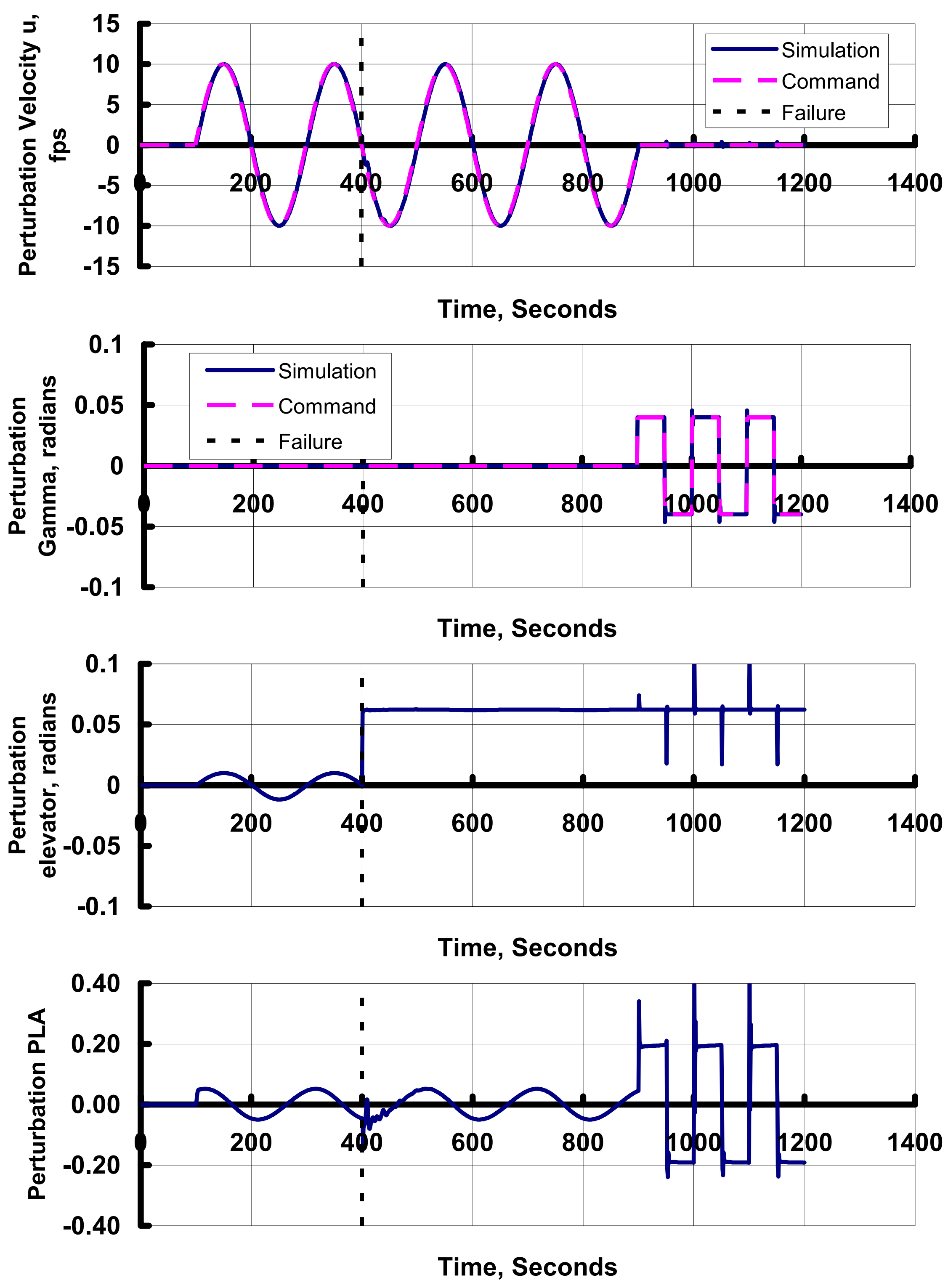


Figure 52 (a): Case 16 time history

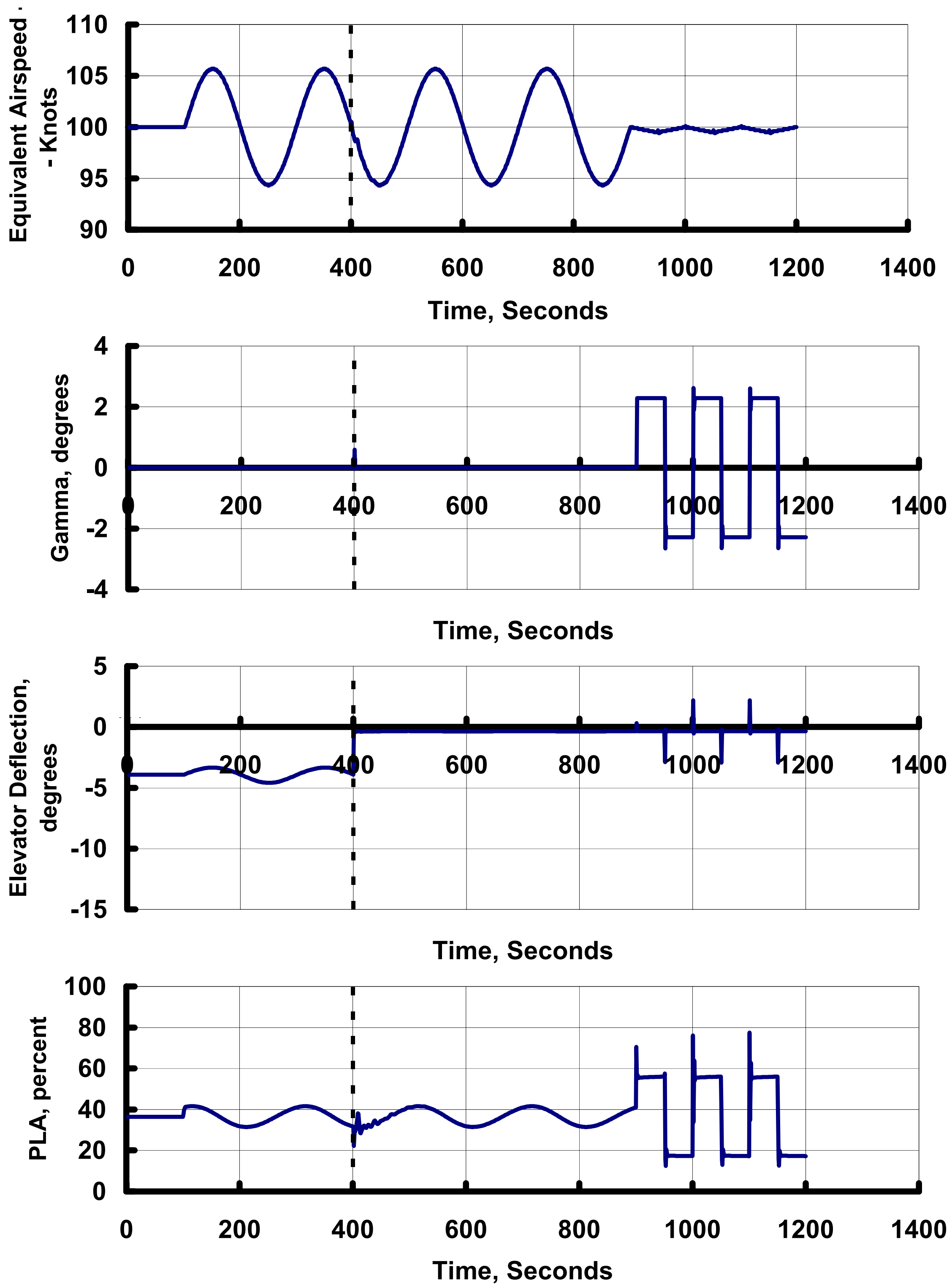


Figure 52 (b): Case 16 time history

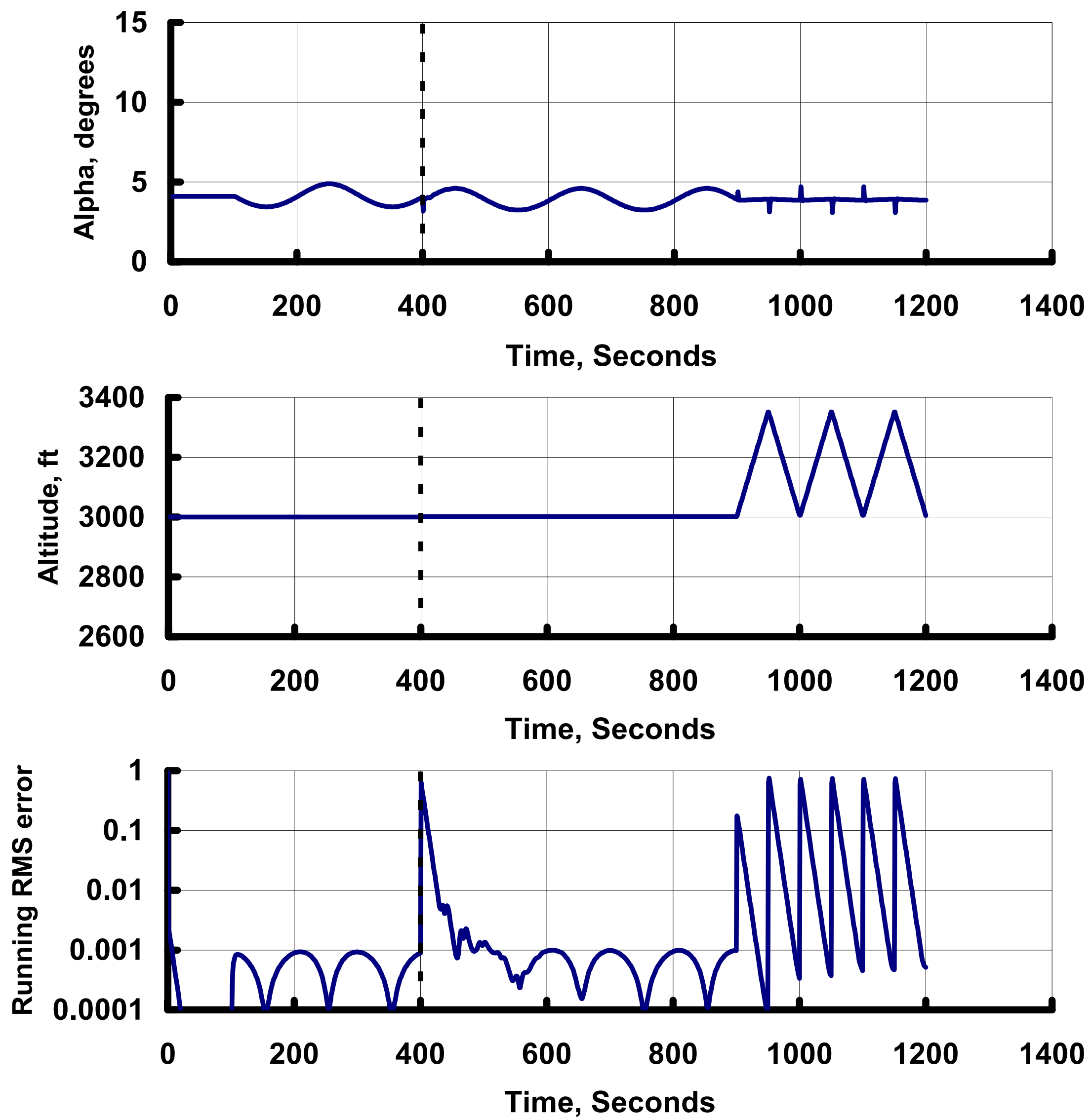


Figure 52 (c): Case 16 time history

4.4 Nonlinear Physical Plant Characteristics

Cases 17 and 18 address nonlinear characteristics in the physical plant. All previous cases assumed linear aerodynamic characteristics. For these next two cases, the lift and pitching moment coefficients vary in an arbitrarily nonlinear fashion, as illustrated in Figures 53 and 54. However, the adaptive neural network is still pre-trained with linear aerodynamic characteristics.

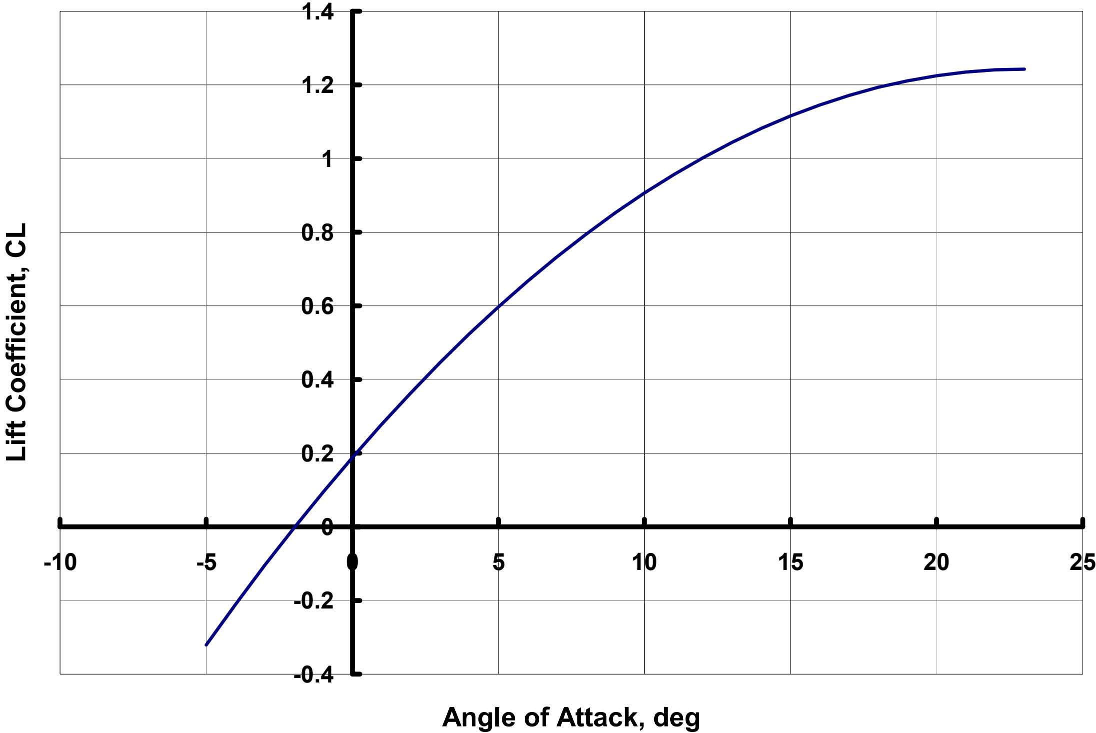


Figure 53: Simulated nonlinear lift coefficient characteristics

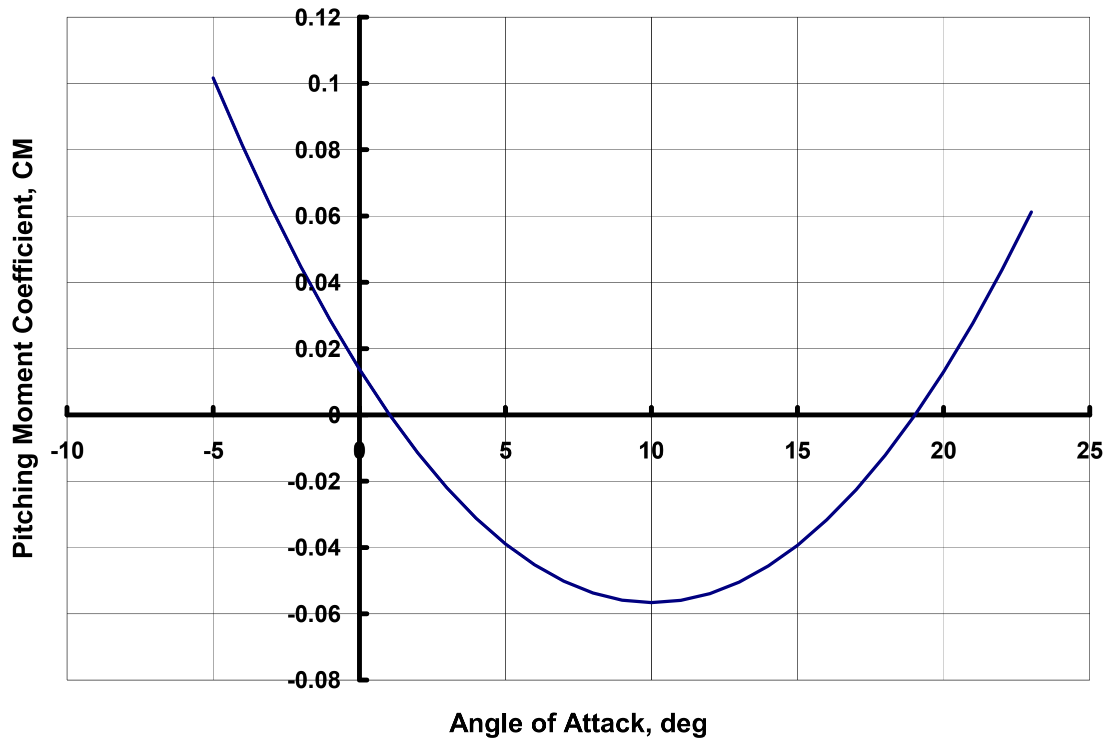
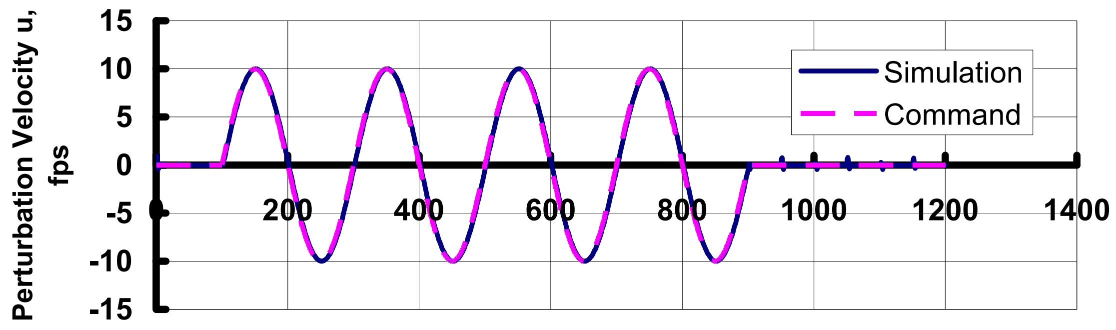


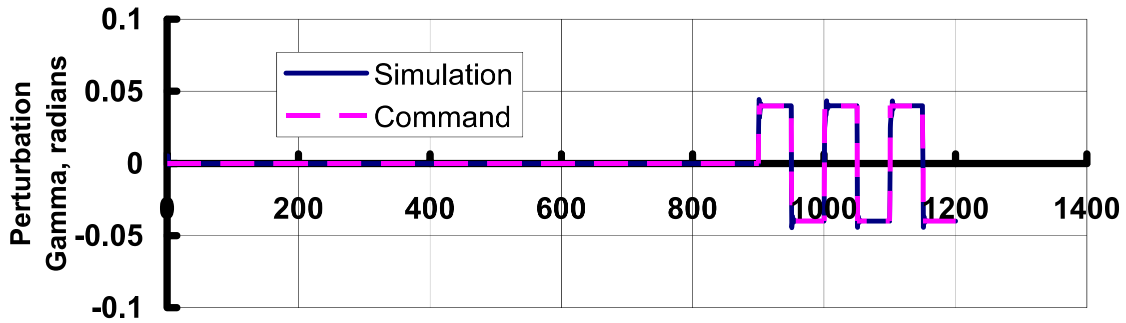
Figure 54: Simulated nonlinear pitching moment coefficient characteristics

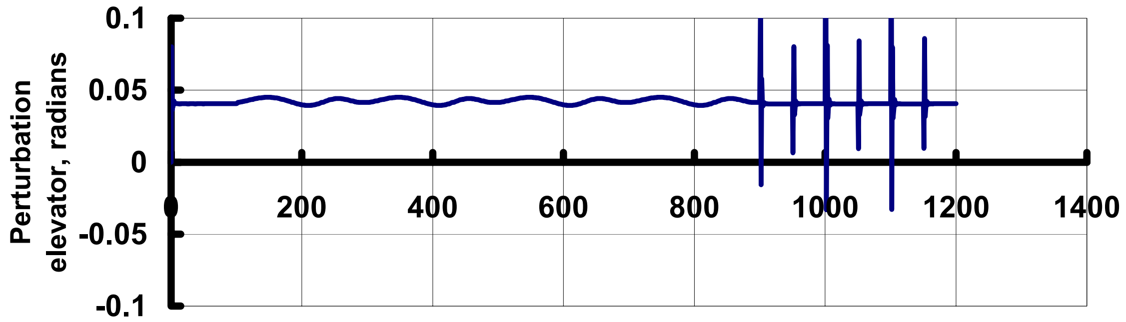
Case 17 (Figure 55) is commanded with the same variations in speed and flight-path as Case 15. However, these variations are about a new trim speed of 80 KEAS. This new trim speed is chosen such that the trim angle of attack of 9.47 degrees is at the threshold of the significant nonlinear behavior. As the angle of attack varies, the aircraft experiences significant variations in C_{L_α} and C_{M_α}. No apparent difference in controller performance is evident from Case 15. This is a significant accomplishment considering that the adaptive neural network was pre-trained on the linear aircraft response, and it is adapting to the nonlinear aerodynamics in real-time.

Case 18 (Figure 56) is the same as Case 17, except that both $C_{L_{\delta e}}$ and $C_{M_{\delta e}}$ are reduced by 50 percent at 400 seconds. A very small transient is evident at the time of the failure, but the

controller maintains speed tracking accuracy well. This transient manifests itself as small variations in speed and PLA, but primarily in elevator deflection. The controller detects the need for a new trim elevator setting, in light of the reduced elevator authority, and smoothly adjusts. Later however, flight-path accuracy is somewhat degraded during the abrupt step changes in commanded flight-path. Some loss of precision is evident in the initial capture of the target flight-path angle. In the second period of the flight-path command square wave, the controller is unable to keep up, and control is lost. This is the only case tested where the controller was not capable of managing the airplane. As in Case 17, the network has been pre-trained with the linear aircraft response. It is speculated that if a rigorous pre-training of the network was undertaken using the nonlinear characteristics, this loss of control would be avoided.







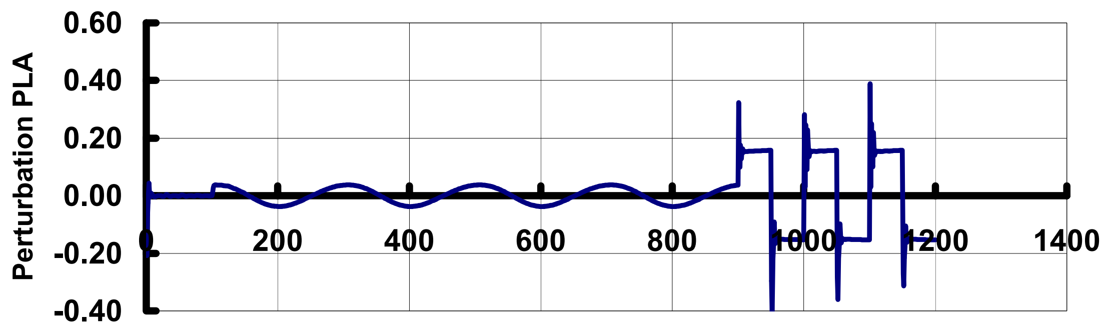


Figure 55 (a): Case 17 time history

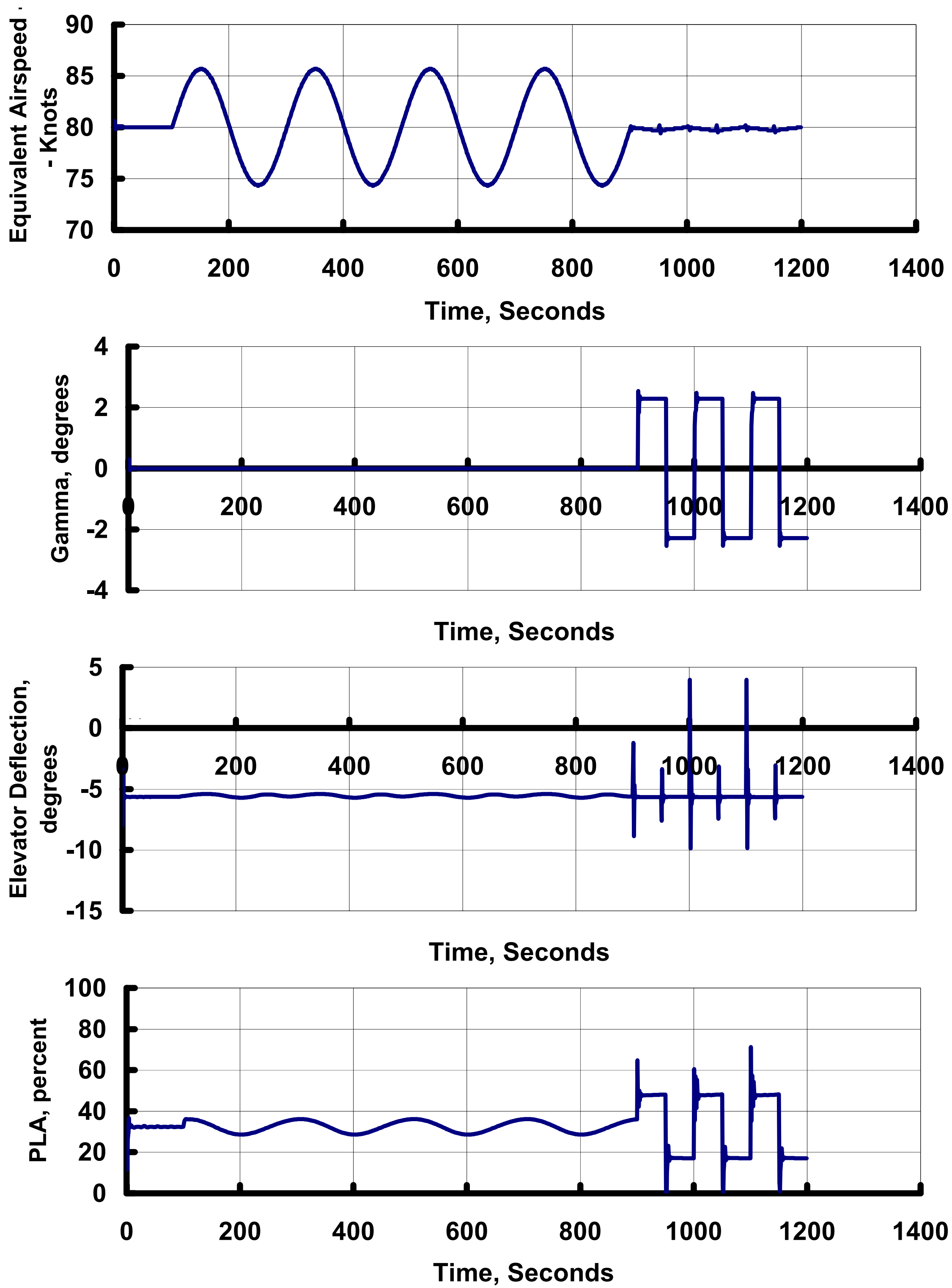


Figure 55 (b): Case 17 time history

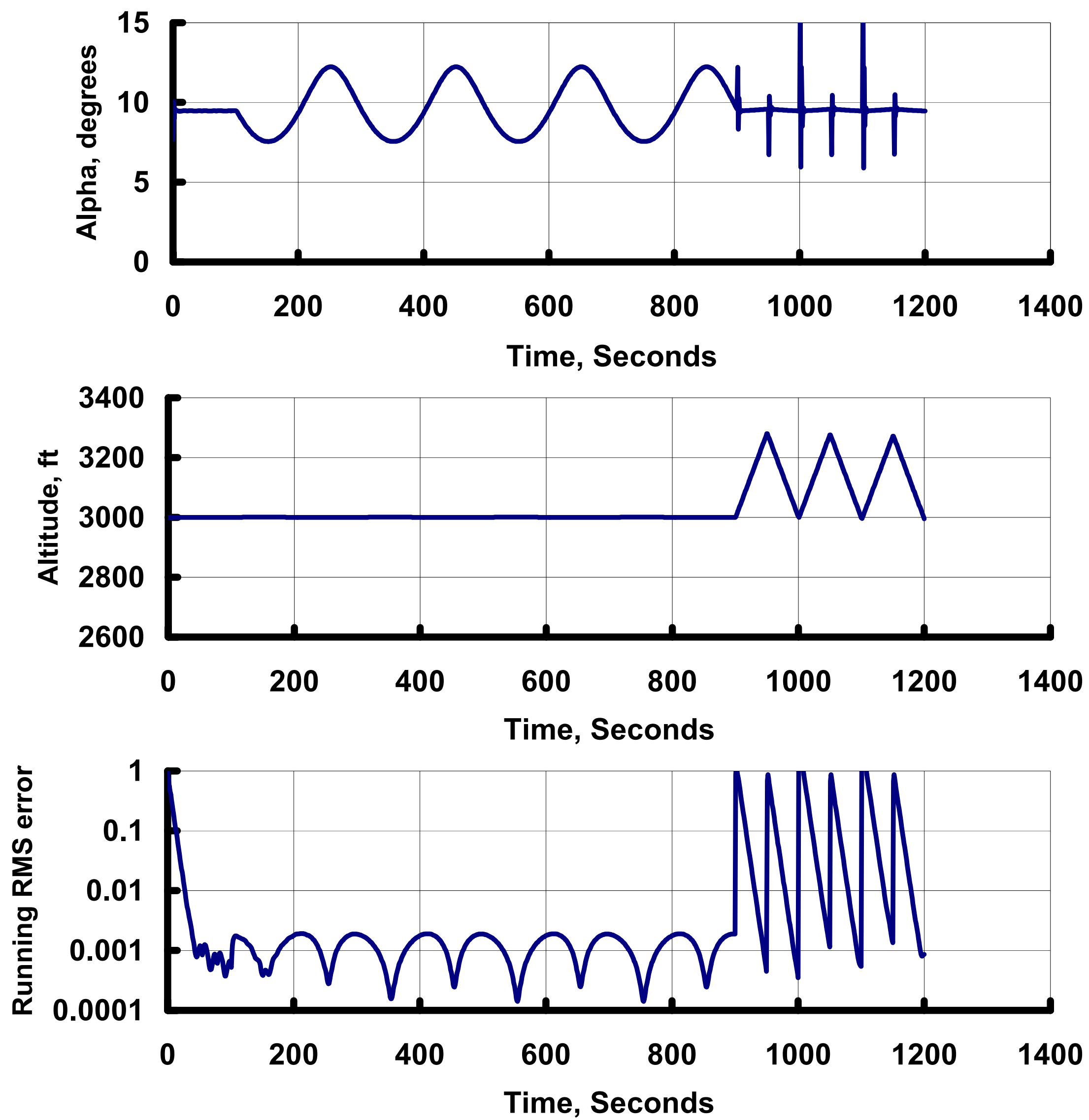


Figure 55 (c): Case 17 time history

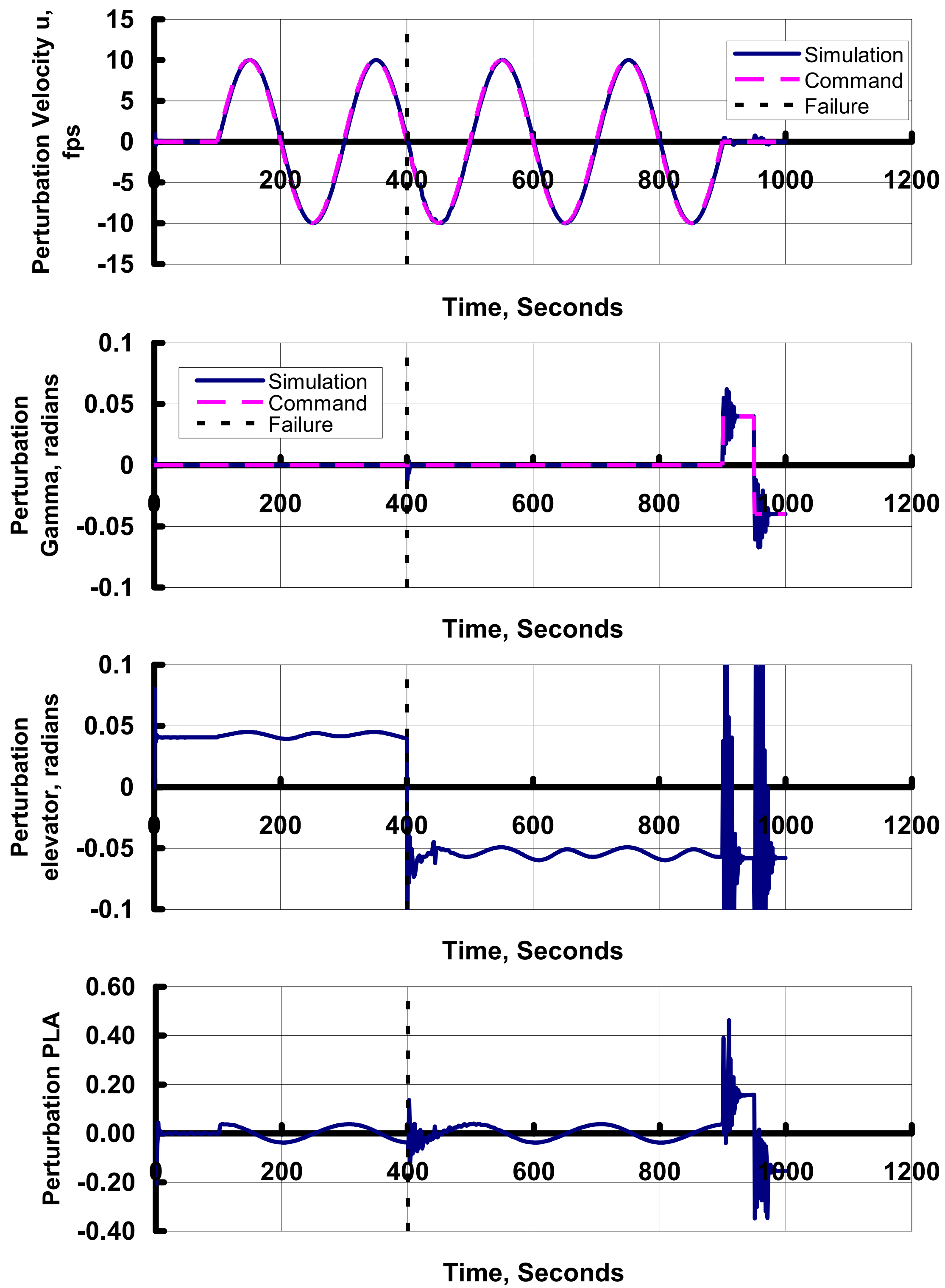


Figure 56 (a): Case 18 time history

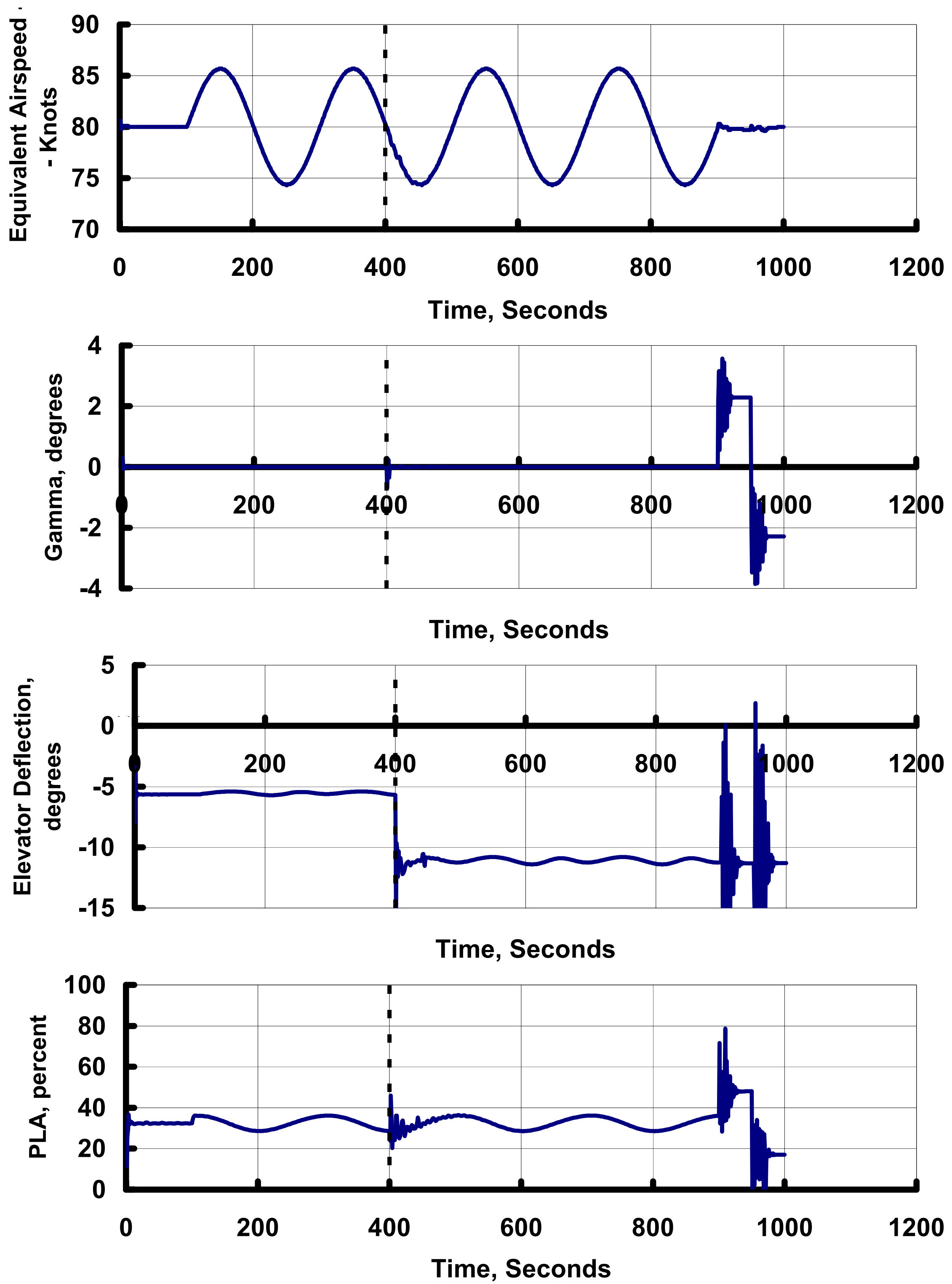


Figure 56 (b): Case 18 time history

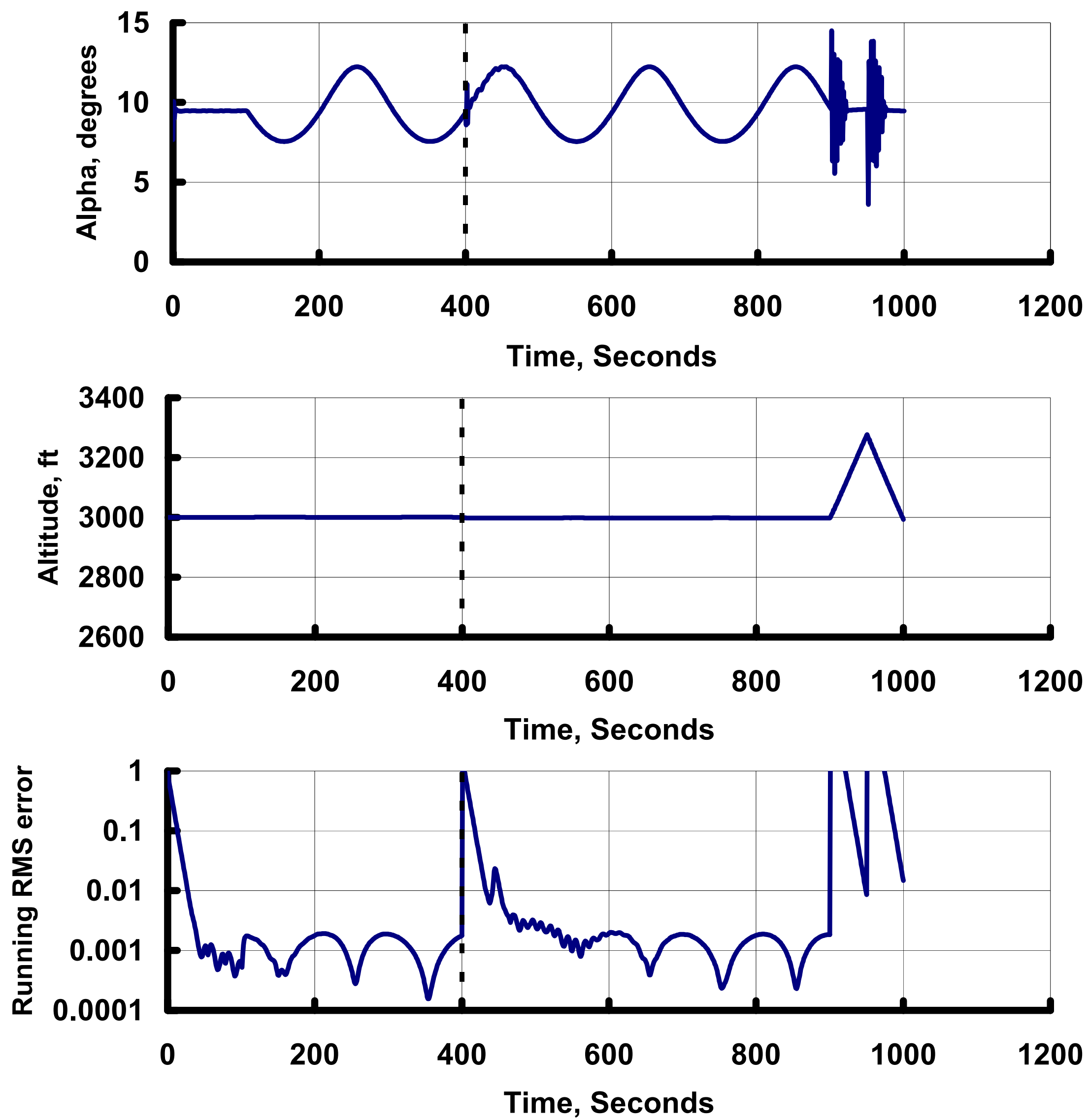


Figure 56 (c): Case 18 time history

4.5 Neural Network Failures

A brief examination is undertaken of failures of the neural network itself. This discussion is outside the original scope of the research proposed for this dissertation but is a worthwhile addition nonetheless. Insights are obtained for design and potential certification of future control systems.

Failures in typical flight control systems consist of hardovers, oscillatory failures, or runaways resulting from integrator "wind-up." A beneficial feature of neural networks is that many employ activation functions that are self-limiting. In the current research, as an example, neurons use a hyperbolic tangent (tanh) activation function. The unscaled output of the neurons asymptotically approach plus-or-minus one as the sum of their inputs goes to plus-or-minus infinity. This is illustrated in Figure 57. Most importantly, the output neurons are likewise bounded. As a consequence of this, the output of the network is bounded. An unbounded wind-up failure is impossible. Hardovers will be limited in magnitude by the network output scaling. Oscillatory failures will likewise be bounded in amplitude by the output scaling.

Case 19 is the baseline scenario for the network failures examined. All controller functions operate normally. The command for this case is a long period (T = 200 seconds) speed command of 10 feet/sec amplitude. Flight-path angle is commanded to be zero throughout. From examination of the perturbation velocity and perturbation gamma traces, it is seen that the output almost tracks the command perfectly. This should be expected since the frequency of the command is far less than the resonant frequency of the overall system.

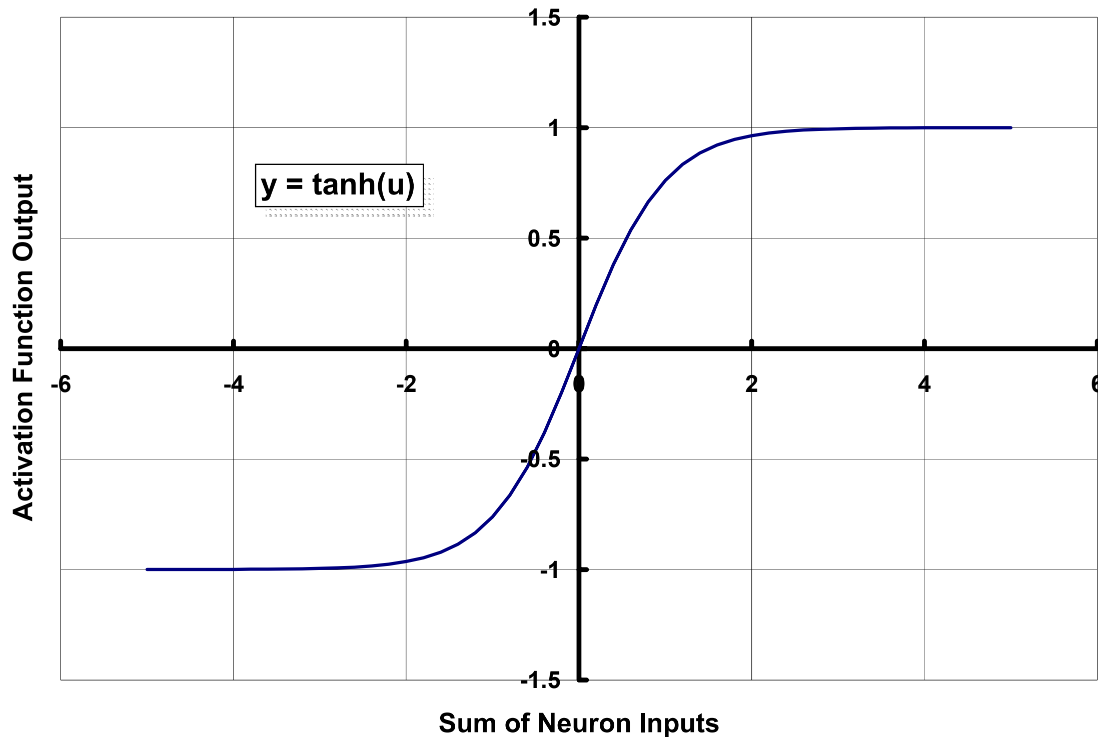


Figure 57: Neuron activation function self-limits output

Case 20 examines a neural network hardover on the $\Delta\dot{u}$ channel that occurs at 6 seconds. This hardover is assumed to go to the maximum output value permitted by the output scaling of the network. The $\Delta\dot{\alpha}$ signal remains functional. Figure 59 shows the resulting response. The simulation is run for 500 seconds to demonstrate there are no transients with long time constants. A steady-state error of approximately 26 feet/sec is visible in the perturbation velocity trace. While not trivial, this error could be trapped by software monitors that either enable a back-up algorithm for mitigation and/or annunciate the failure for pilot corrective action. Flight-path angle tracking error is less than one degree.

The RMS error time history for these failure cases should be interpreted with care. For Case 20, the $\Delta\dot{\alpha}$ channel is learning and participating in control of the system normally. While a

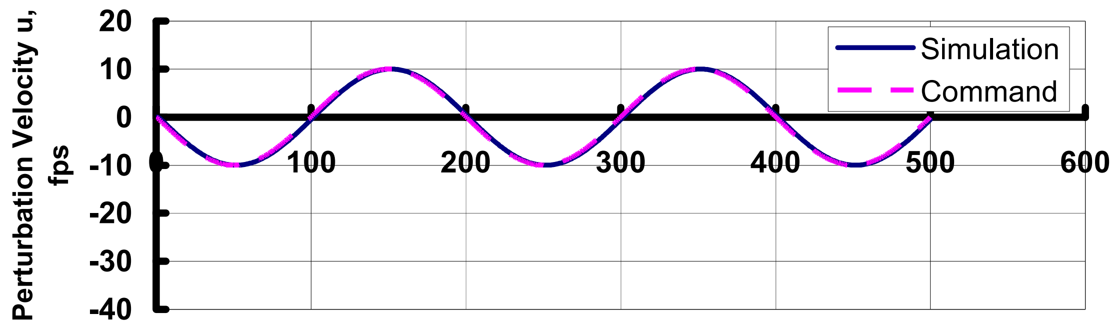
failure on the $\Delta\dot{u}$ channel is simulated, the method of simulating the failure does not precluded that channel from learning normally. This is a consequence of how the failure is simulated. In practice, it is expected that the learning on a failed channel will be unpredictable and will depend on the nature of the failure.

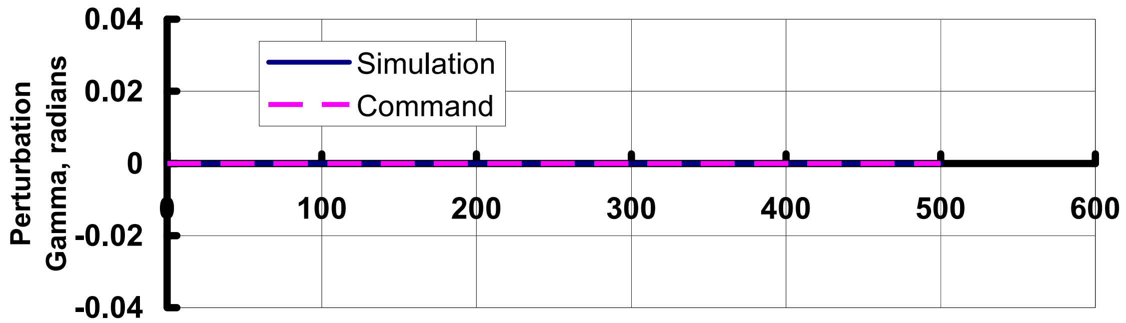
Case 21 examines a $\Delta\dot{u}$ oscillatory failure at 6 seconds occurring at the overall system resonant frequency of .60 rad/sec. Once again, the amplitude of the error signal is assumed to be the maximum permitted by the output scaling. Figure 60 shows the resulting time histories. An oscillatory speed tracking error occurs at an amplitude of about 24 feet/sec. A small amplitude flight-path error occurs on the order of one-half degree.

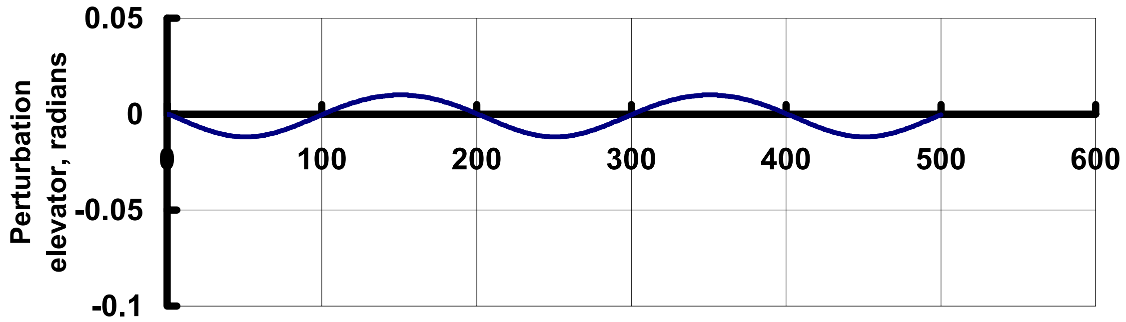
Case 22 examines a neural network hardover on the $\Delta\dot{\alpha}$ channel that occurs at 6 seconds. This signal is assumed to go to the maximum output value permitted by output scaling of the network. The $\Delta\dot{u}$ signal remains functional. Figure 61 shows the resulting response. For this case, speed is unaffected while flight-path angle experiences a steady-state error of only -.01 radians.

Case 23 is illustrated in Figure 62. This case looks at an oscillatory failure of the $\Delta\dot{\alpha}$ channel at 6 seconds. The failure has a frequency of 1.785 rand/sec to match the resonant frequency of the chosen flight-path angle characteristics. As seen in the $\Delta\dot{\alpha}$ hardover, no influence is detected in the speed tracking, and only a small periodic error less than one degree is noted in the flight-path angle tracking.

With these observations in place, questions arise to explain the differences in the effects of failures between $\Delta\dot{u}$ and $\Delta\dot{\alpha}$.







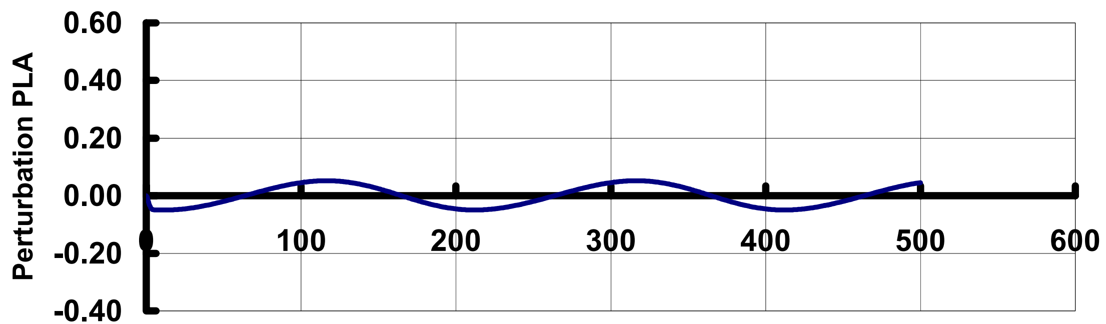


Figure 58 (a): Case 19 time history

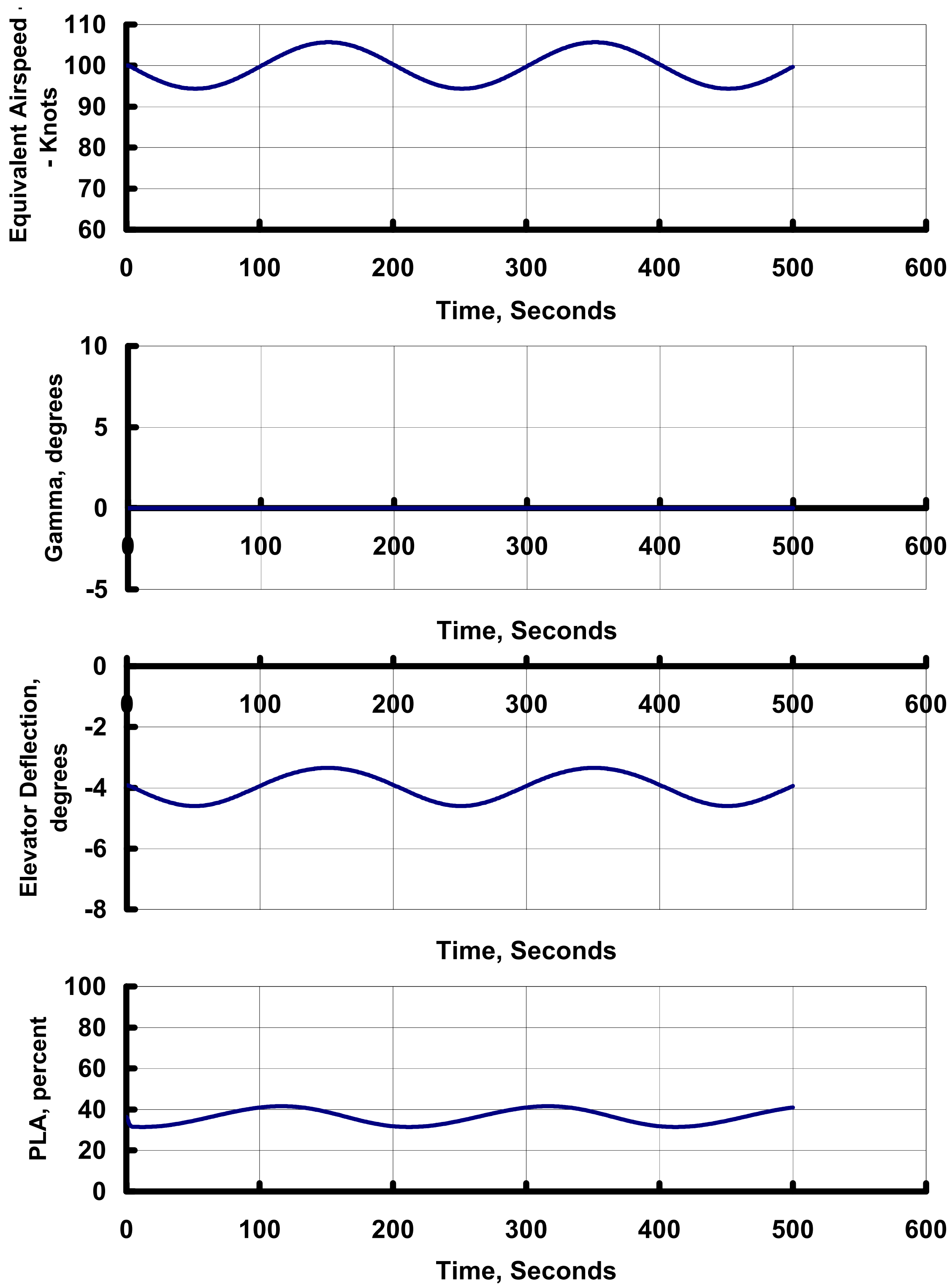


Figure 58 (b): Case 19 time history

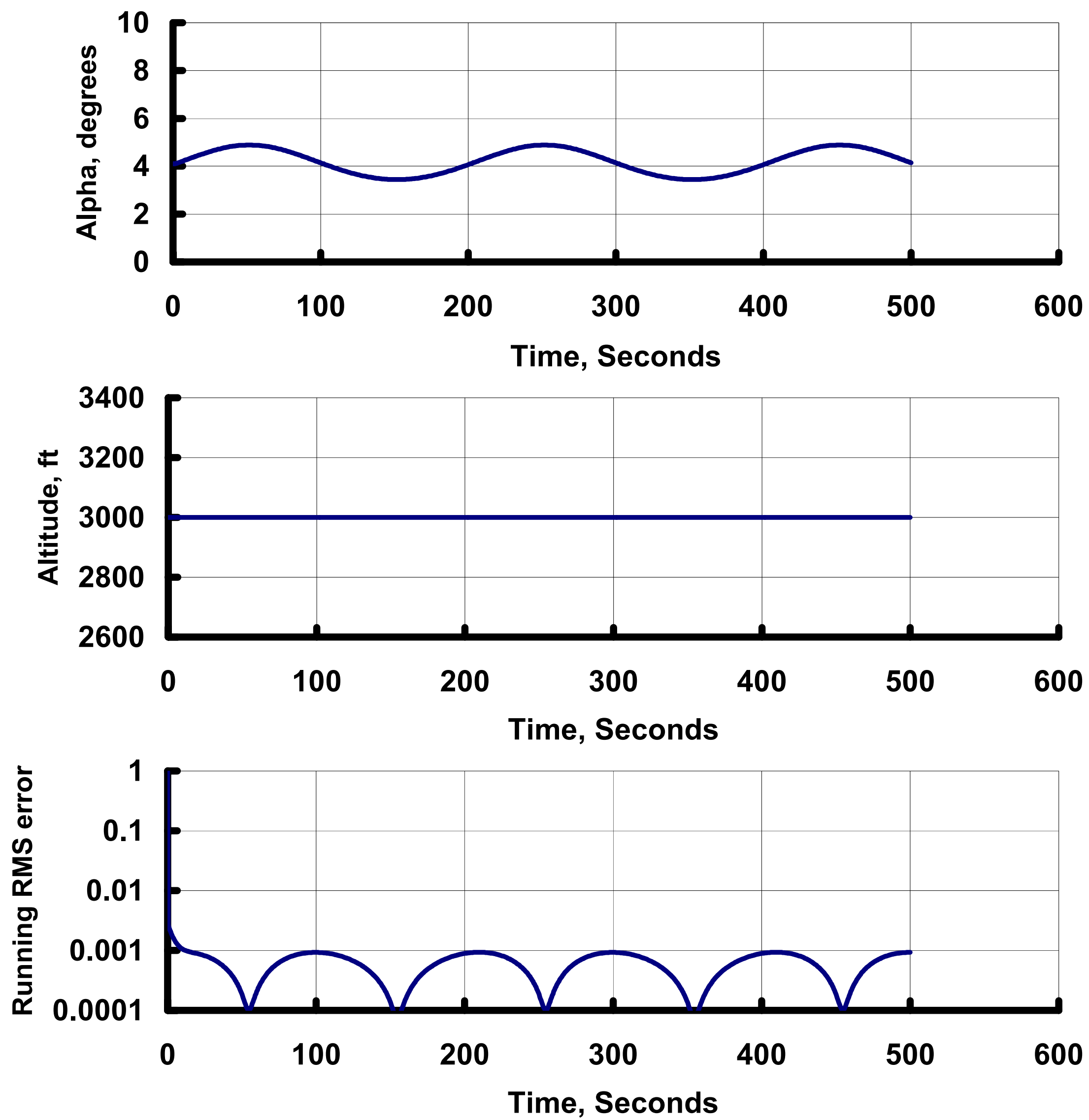
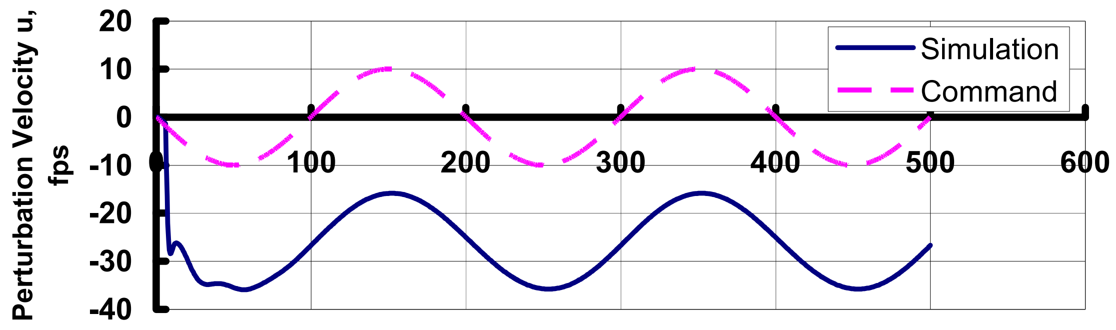
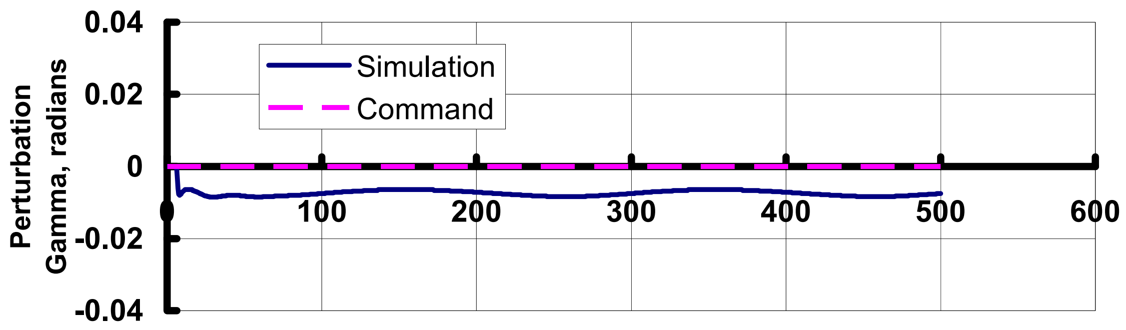
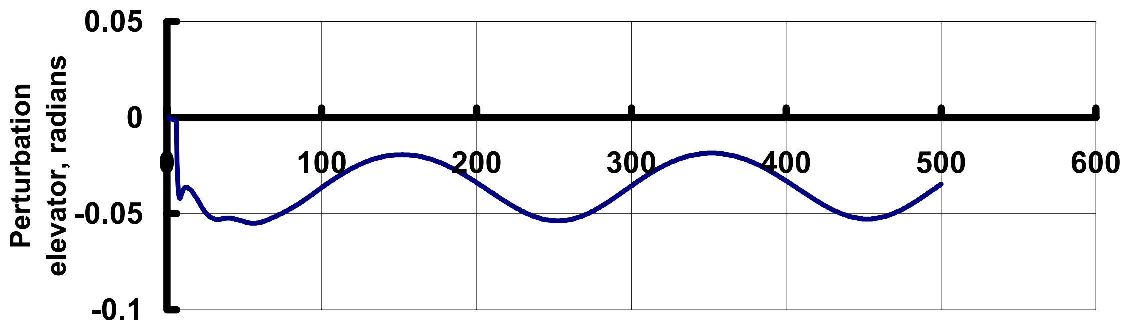


Figure 58 (c): Case 19 time history







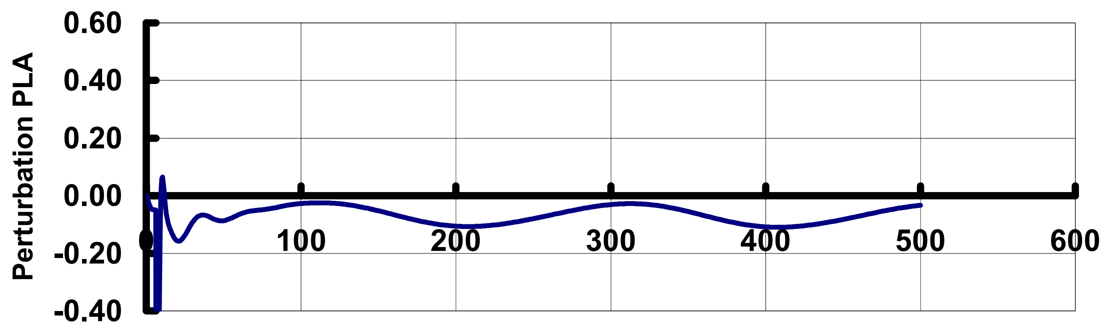


Figure 59 (a): Case 20 time history

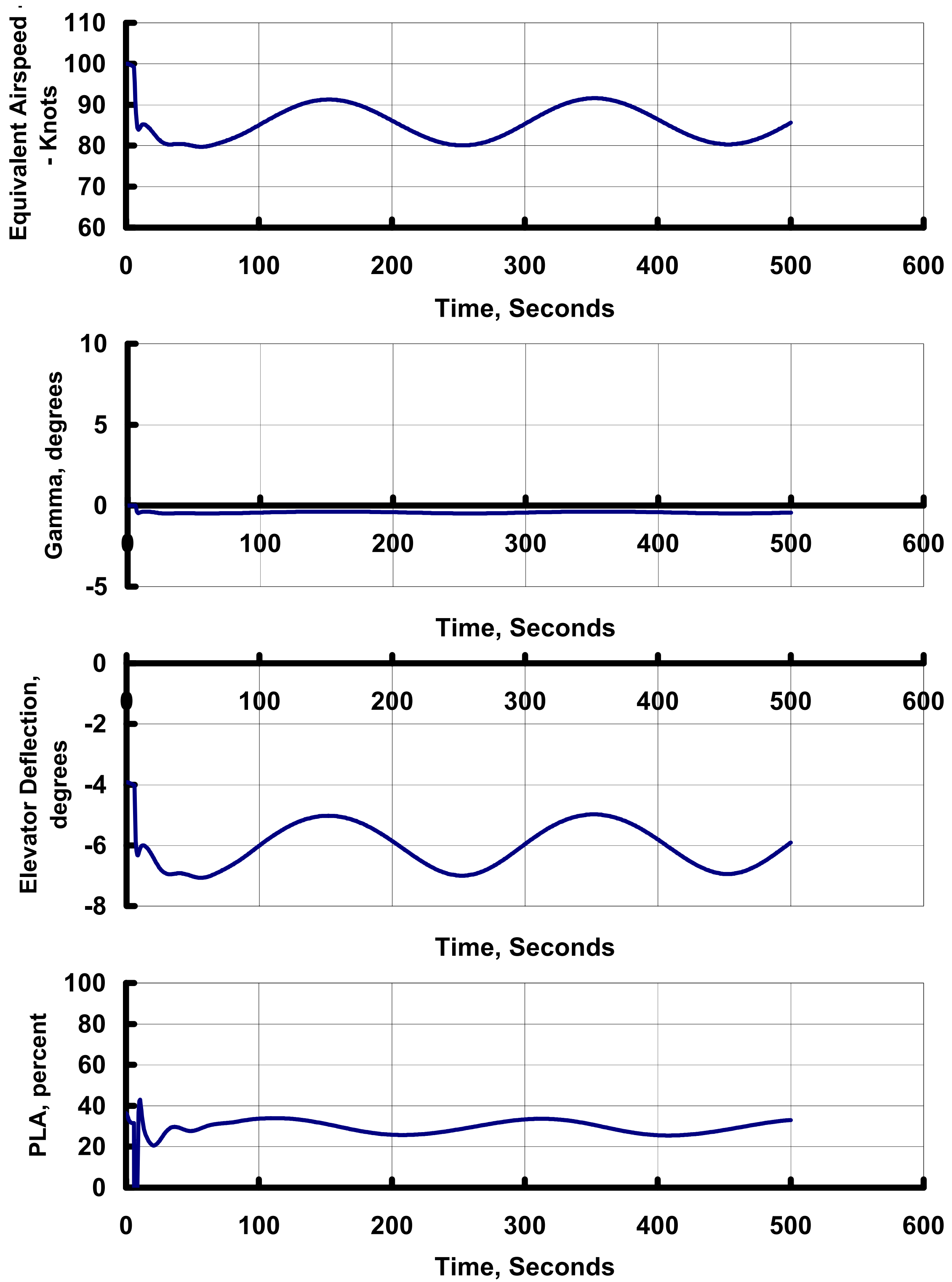


Figure 59 (b): Case 20 time history

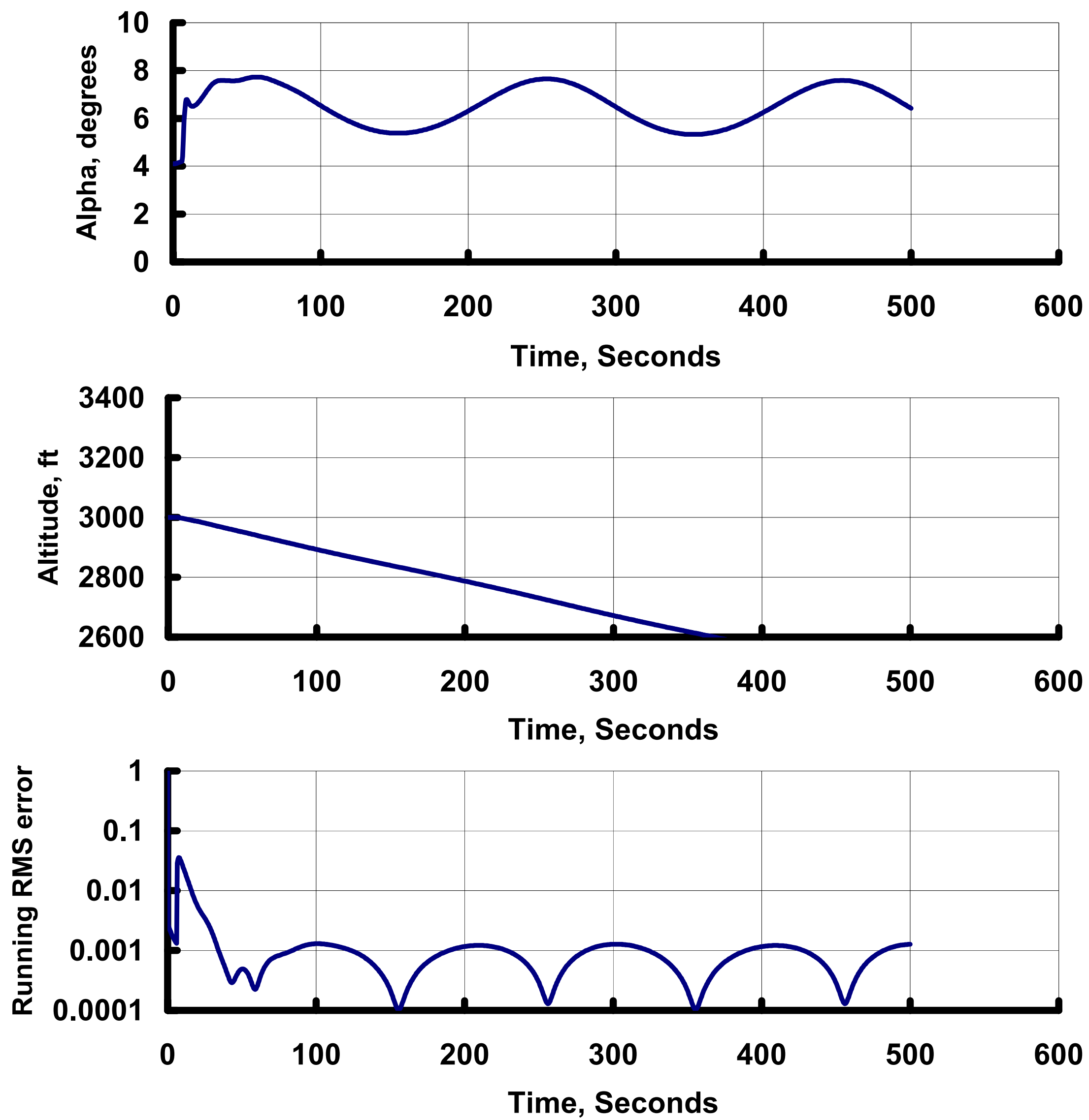
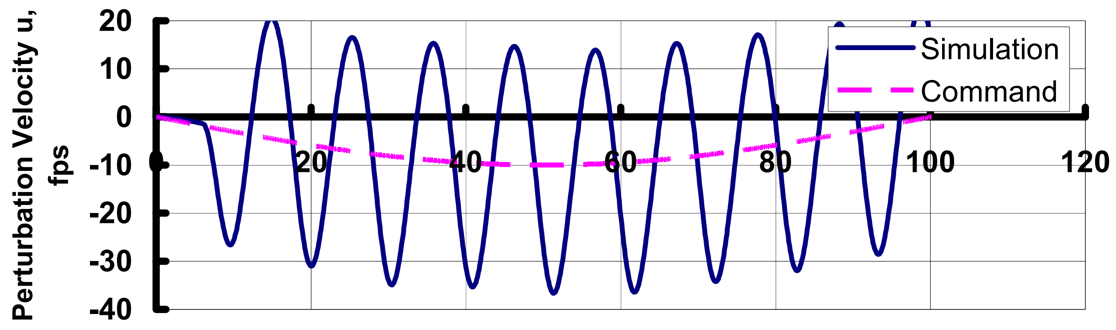
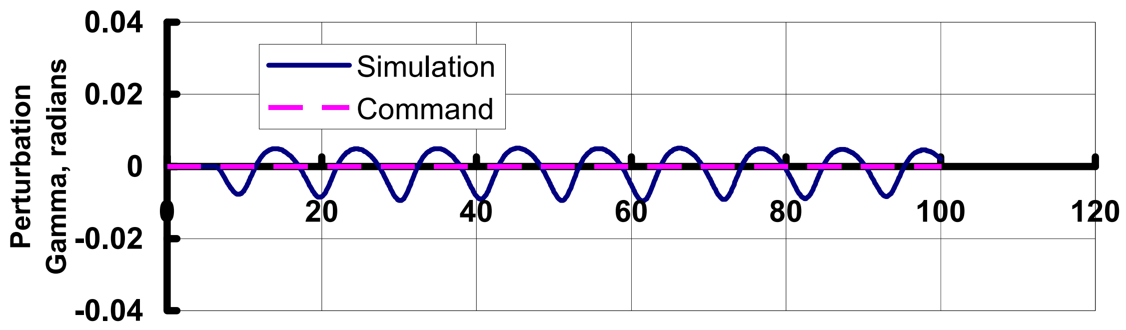


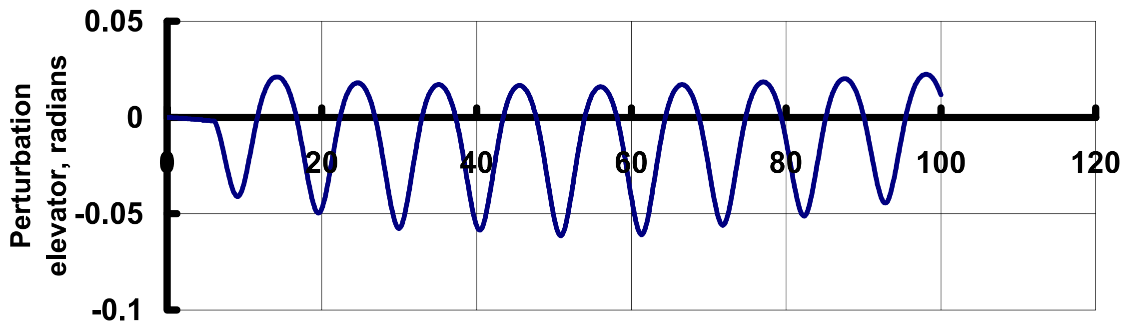
Figure 59 (c): Case 20 time history

First and most easily explained is why $\Delta\dot{u}$ failures influence speed and flight-path tracking, whereas $\Delta\dot{\alpha}$ failures only affect flight-path tracking. By referring back to the flight-path angle block diagram of Figure 20, it can be seen that the $\Delta\dot{\alpha}$ required to track the desired flight-path angle involves speed perturbations in the calculations. Errors in speed tracking will propagate into this function and create errors in flight-path tracking. The influence of this speed error is of the second-order when compared to flight-path errors and, consequently only a small error in flight-path angle results. In contrast, the speed control block diagram of Figure 18 has no angle of attack participation, and as a result, $\Delta\dot{\alpha}$ errors do not influence speed tracking. Of course, $\Delta\dot{u}$ and $\Delta\dot{\alpha}$ failures each influence their respective tracking accuracy.

The question remains why the speed errors from a $\Delta\dot{u}$ failure are significantly larger than the flight-path errors resulting from the $\Delta\dot{\alpha}$ failures, in relative terms. The answer is rather straight forward. The output scaling on $\Delta\dot{u}$ is relatively large, approximately 16 feet/sec^2 for the training data used in this investigation, or almost a half g. The influence of this scaling factor may be assessed by examining the dynamics of the speed controller. Figure 63 illustrates the speed controller with a neural network error, $\varepsilon(s)$, injected. Because of the equivalency of $\mathbf{G}^{-1}(s)\mathbf{P}_A(s)$ to an integrator, this system can be viewed as a single-input single-output (SISO) system. The system transfer function can be readily derived.







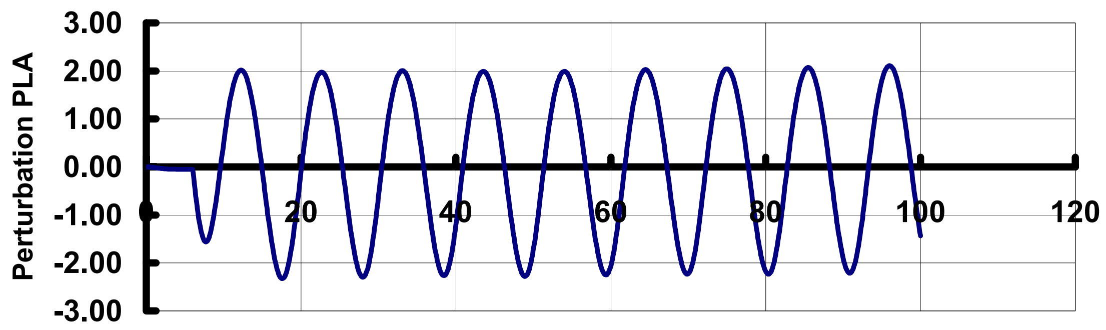


Figure 60 (a): Case 21 time history

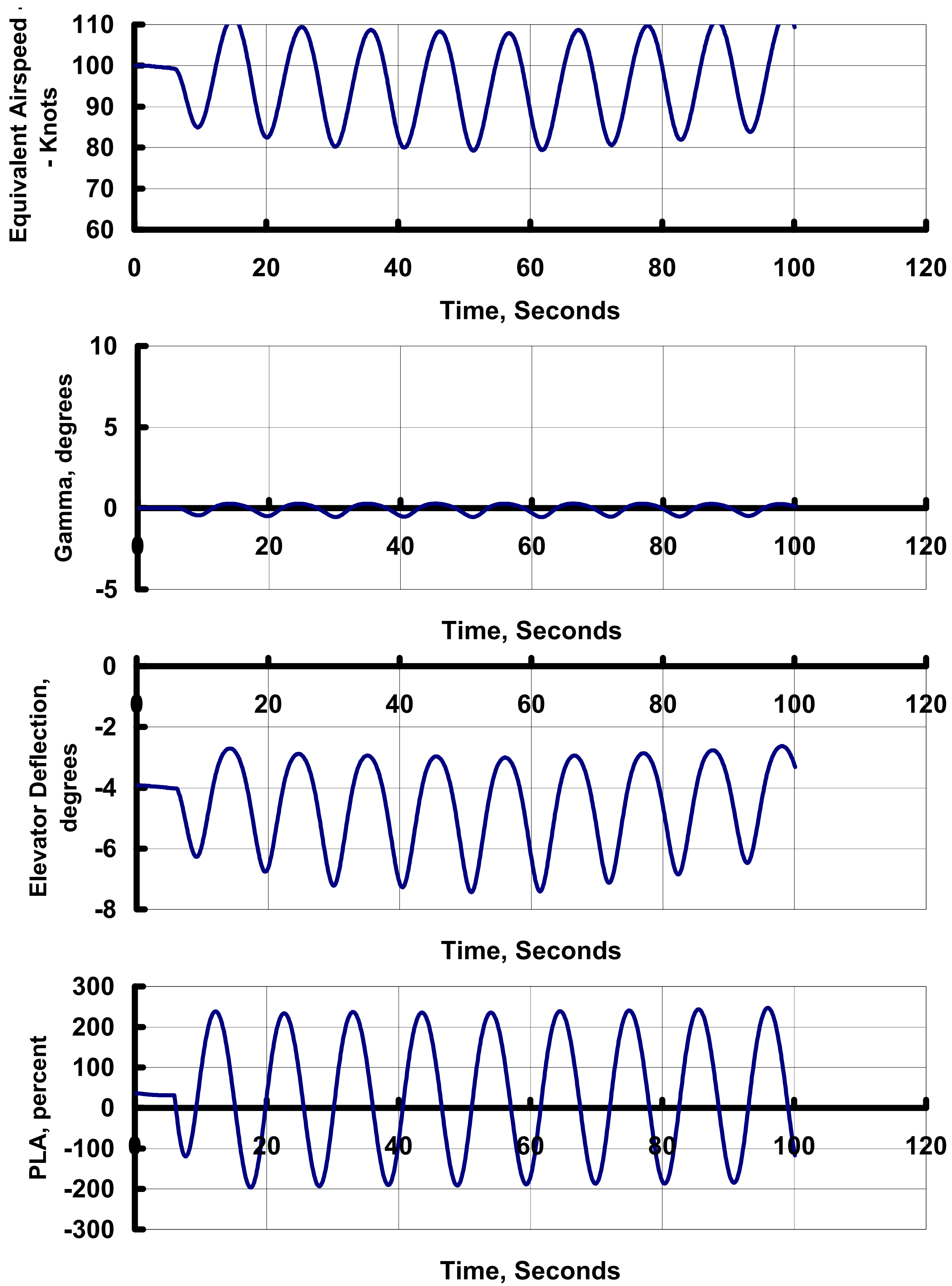


Figure 60 (b): Case 21 time history

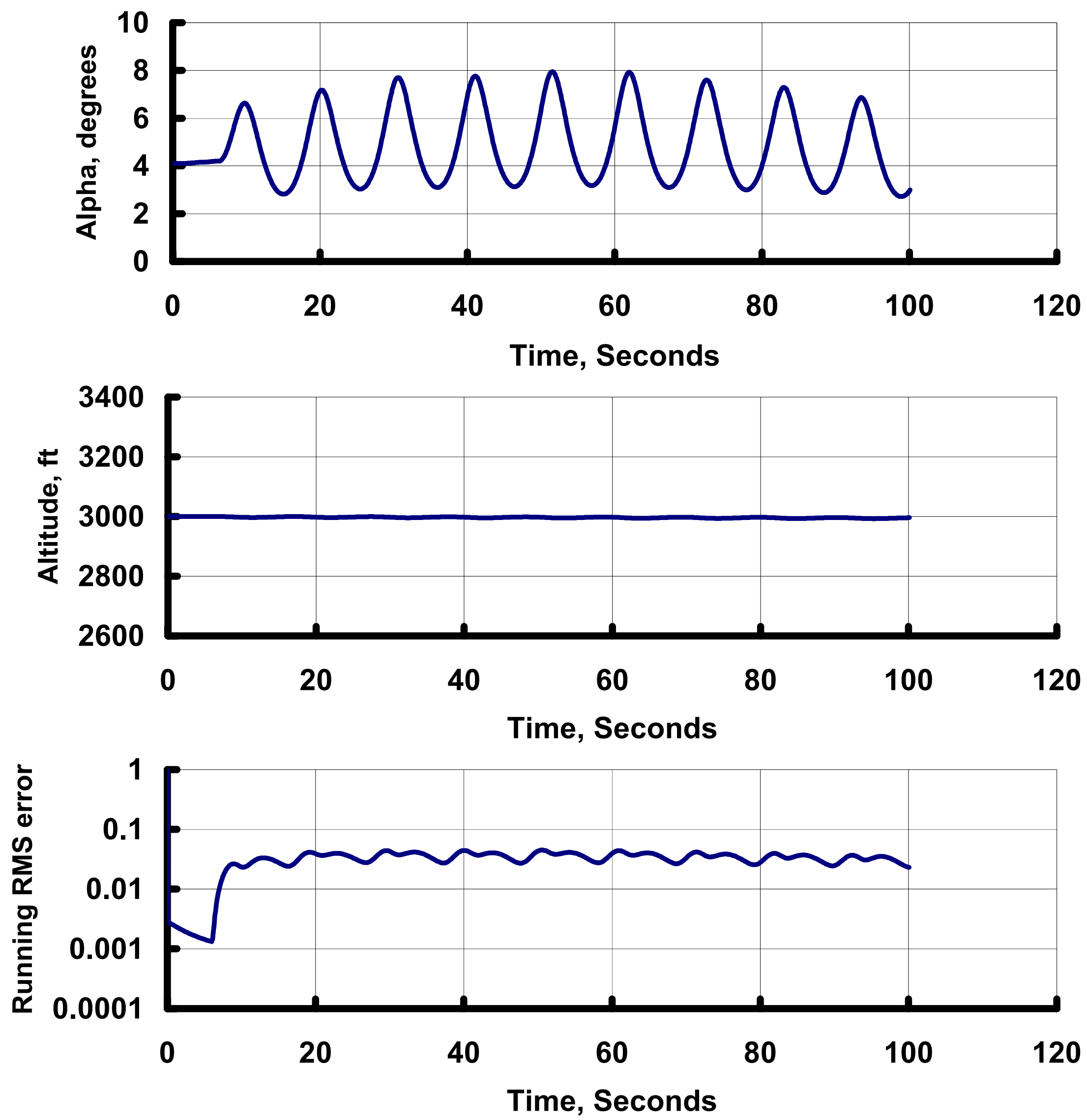
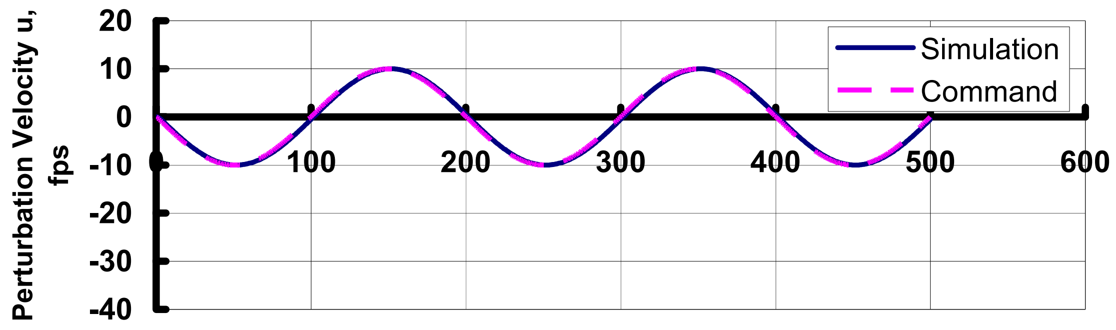
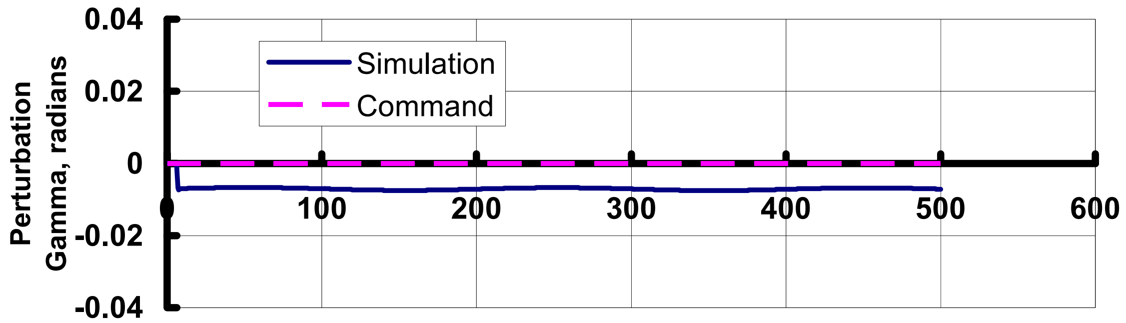
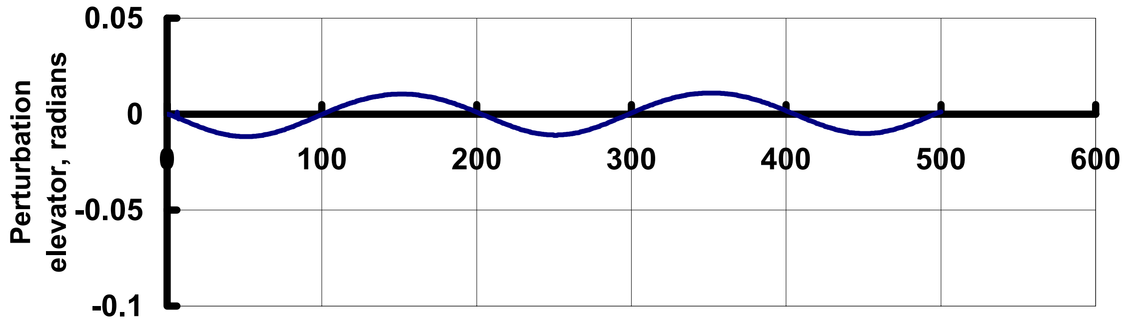


Figure 60 (c): Case 21 time history







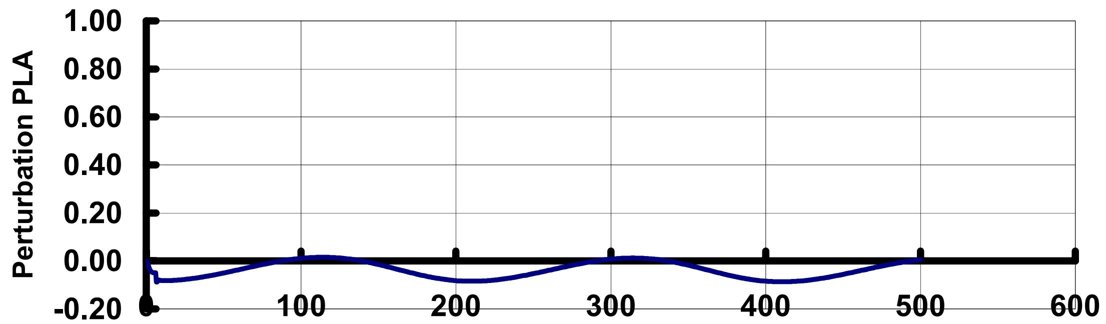


Figure 61 (a): Case 22 time history

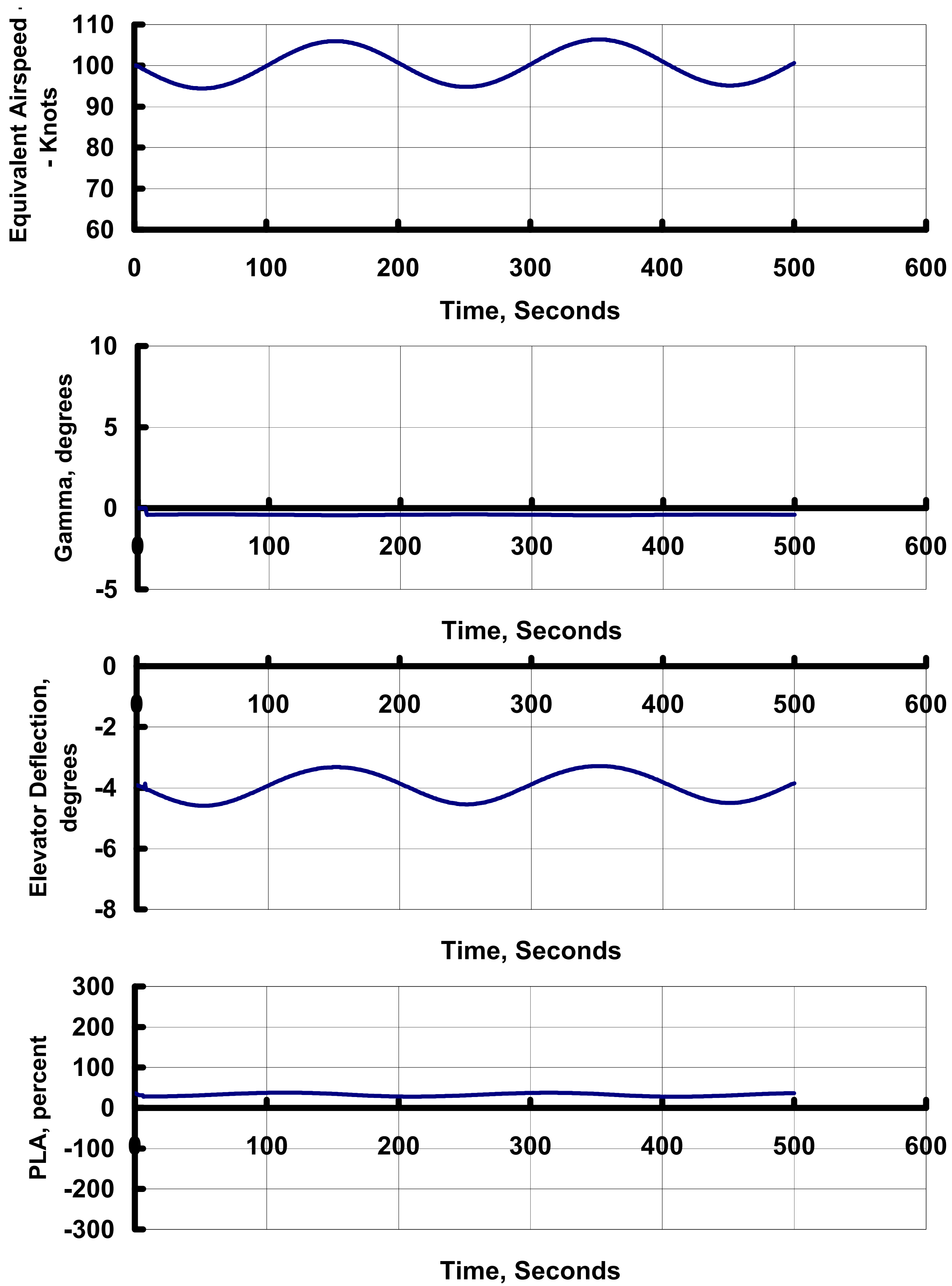


Figure 61 (b): Case 22 time history

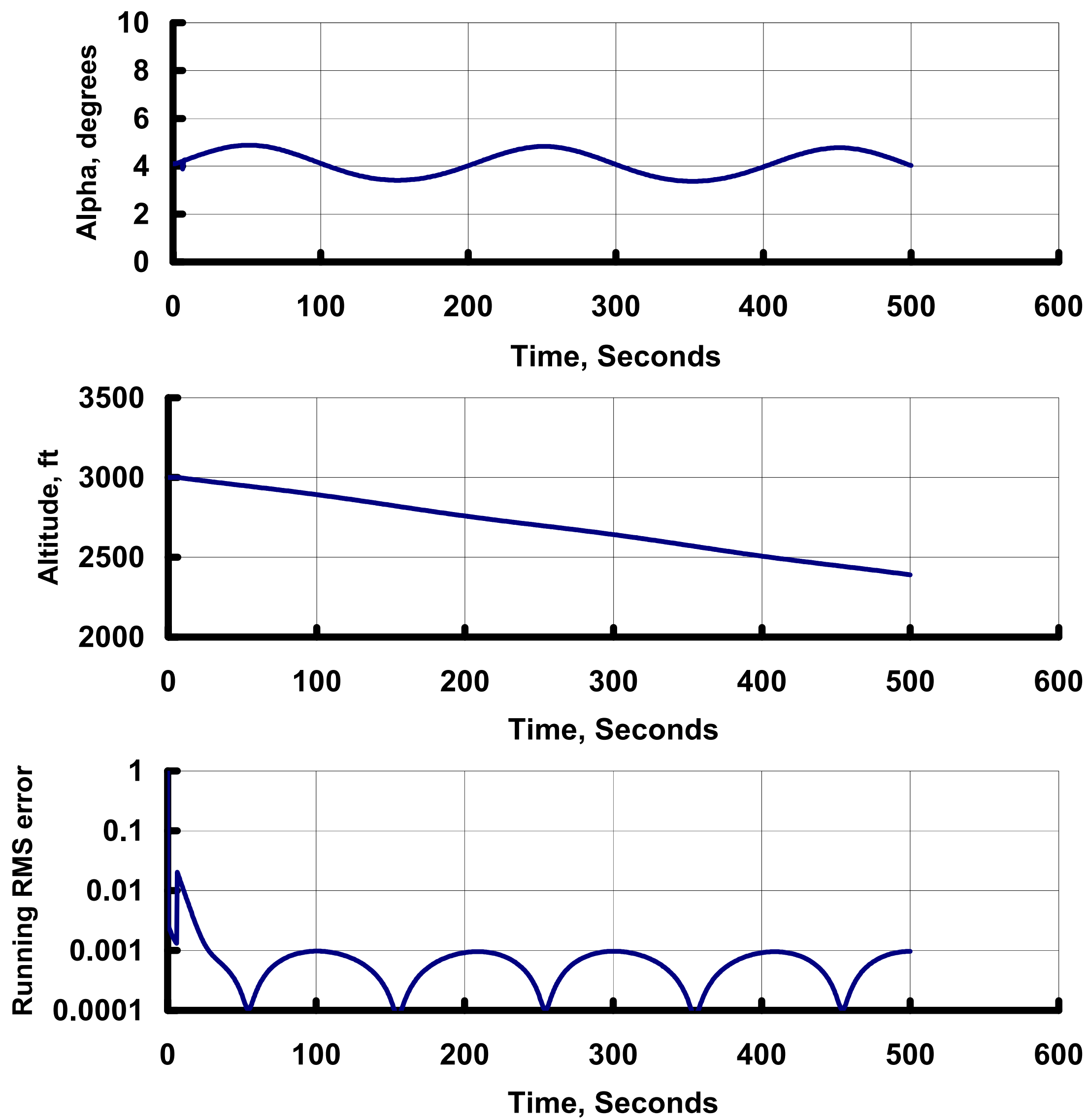


Figure 61 (c): Case 22 time history

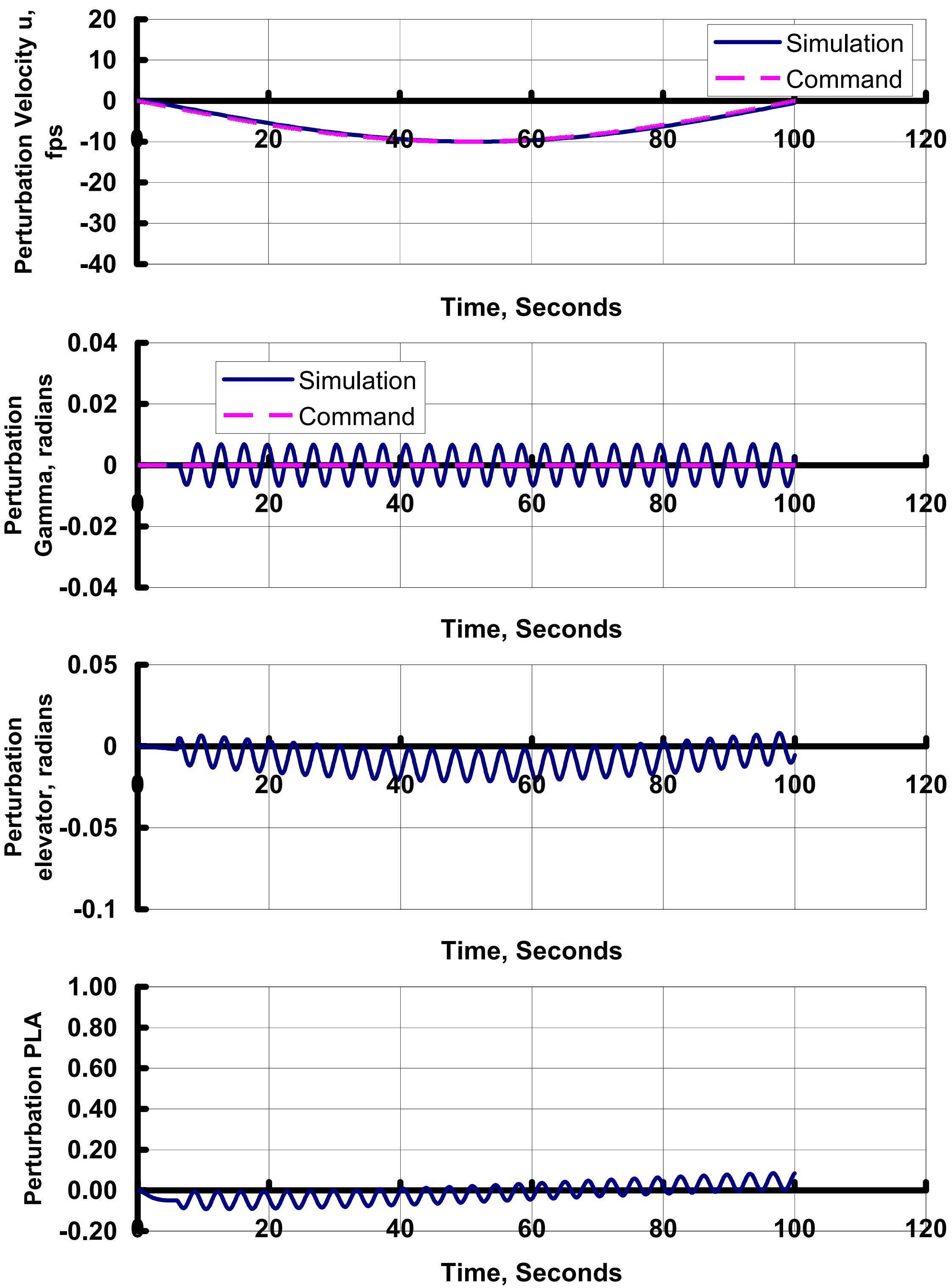


Figure 62 (a): Case 23 time history

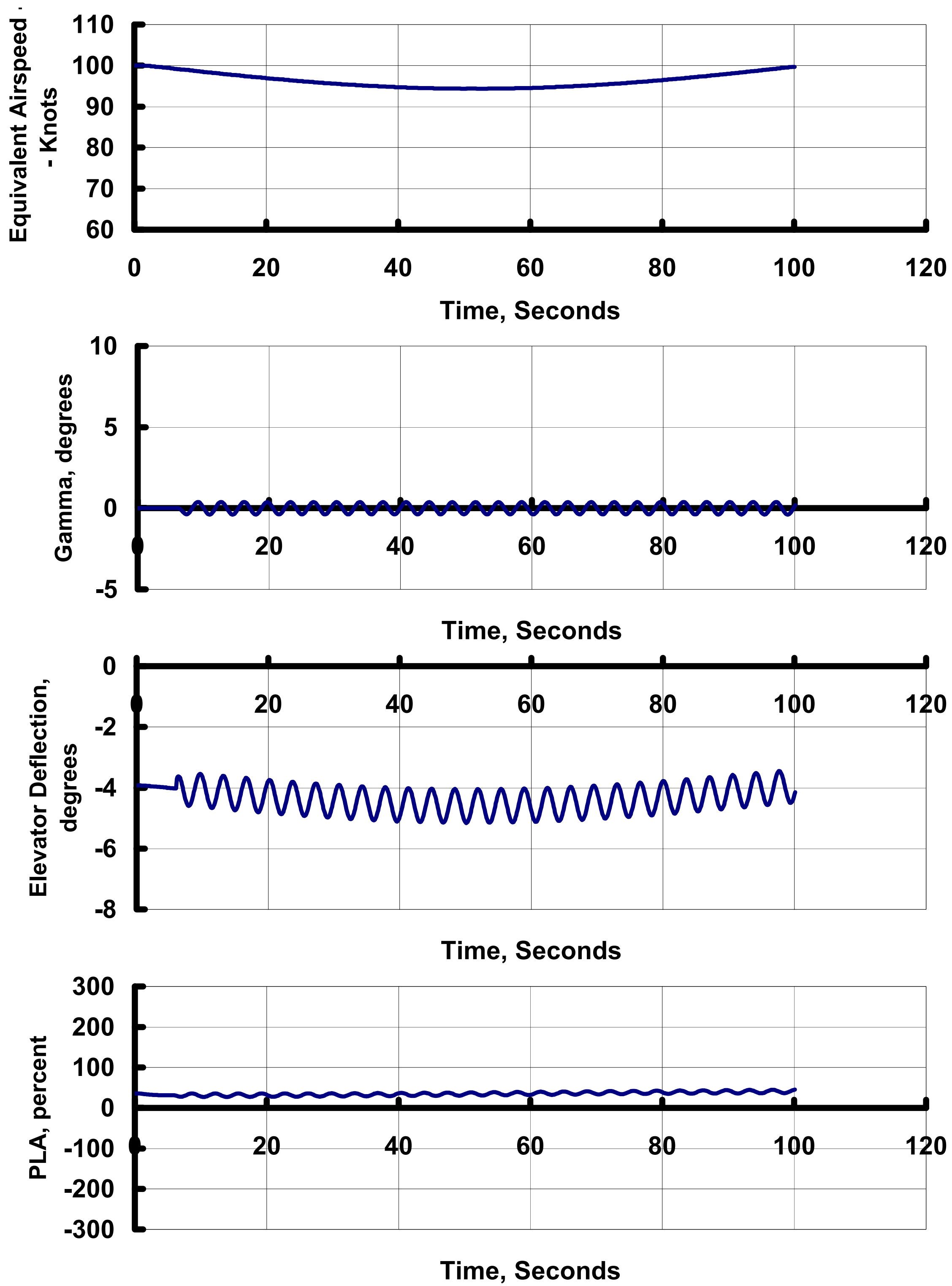


Figure 62 (b): Case 23 time history

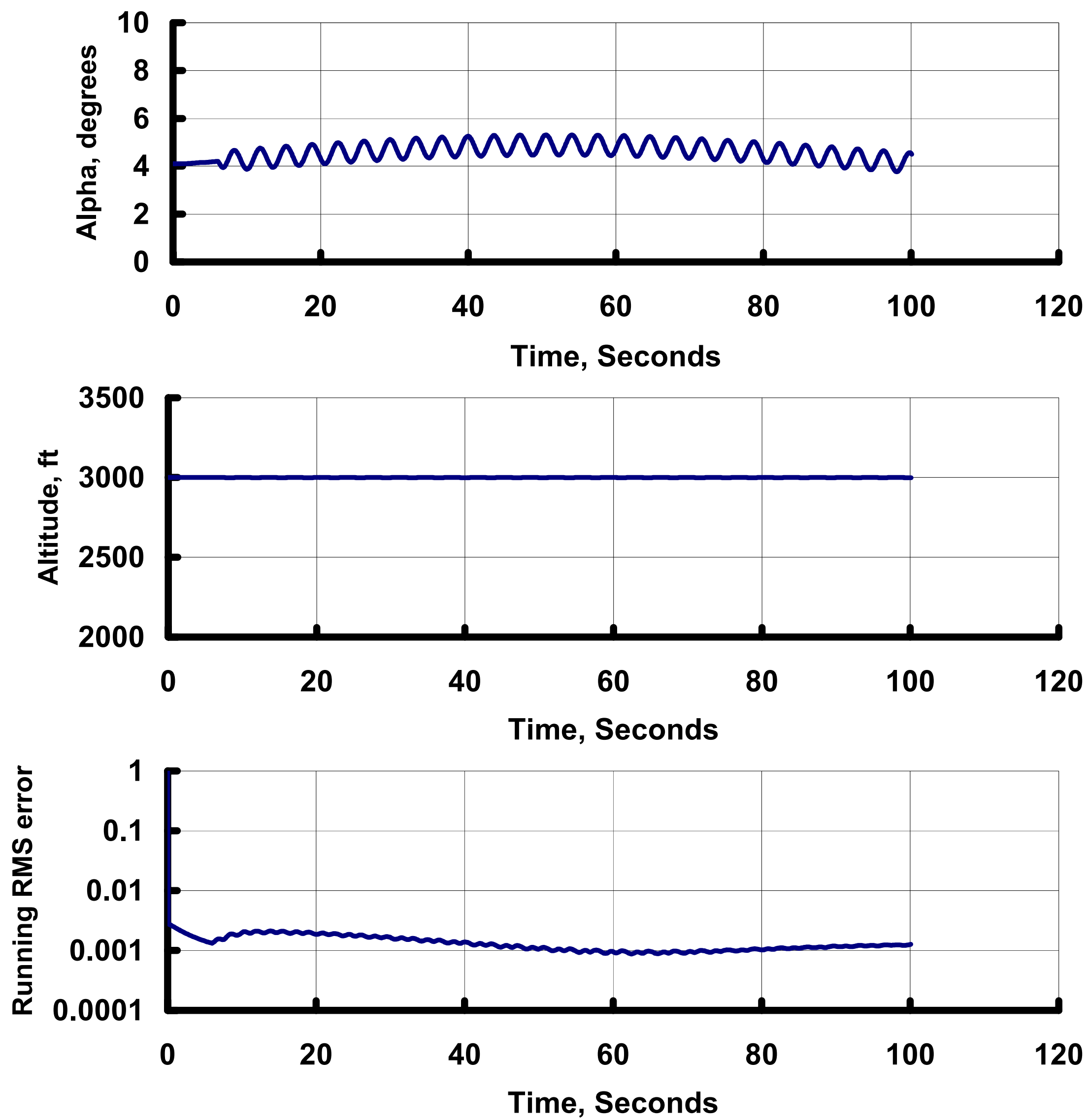


Figure 62 (c): Case 23 time history

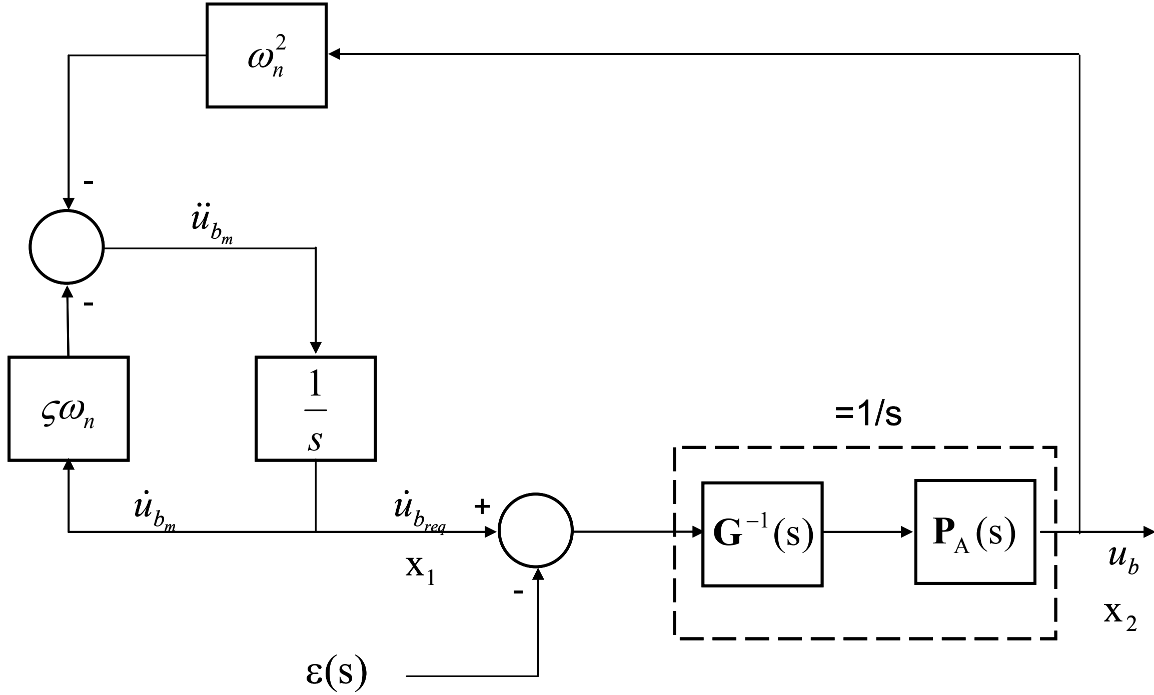


Figure 63: Speed controller block diagram with erroneous neural network signal

The state space representation of this system is

$$\begin{Bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{Bmatrix} = \begin{bmatrix} -2\varsigma\omega_n & -\omega_n^2 \\ 1 & 0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} \varepsilon(t) \tag{4.1}$$

$$u_b = x_2 \tag{4.2}$$

The A matrix of this system is exactly the same as the A matrix for the u channel of the linear prefilter in normal operation. The eigenvalues are determined by the characteristic equation

$$s^2 + 2\varsigma\omega_n s + \omega_n^2 = 0\,, \tag{4.3}$$

which are similar to the nominal controller.

The steady-state response of the system to a constant input ε is

$$u_b = -\frac{2\varsigma}{\omega_n}\varepsilon \tag{4.4}$$

The steady-state response for a sinusoidal input $\varepsilon(t) = \sin \omega t$ is

$$|u_b| = |\varepsilon|\sqrt{\frac{\omega^2 + (2\varsigma\omega_n)^2}{(\omega^2 - \omega_n^2)^2 + (2\varsigma\omega_n\,\omega)^2}} \tag{4.5}$$

For all sample cases, $\zeta = .80$ and $\omega_n = 1.0$. For the hardover case, ε = 16 and the resulting steady-state error on u_b would be 25.6 feet/sec. This theoretical error agrees quite well with the simulation results shown in Figure 59. For a sinusoidal input where ε(t) = 16 sin(.6t), the steady-state speed error amplitude would be 23.7 feet/sec, which also agrees well with the simulation results shown in Figure 60. A Bode chart illustrating the speed sensitivity of the system response to $\Delta\dot{u}$ failures is shown in Figure 64. It is common to associate critical oscillatory failures with the system resonant frequency; however, in this case the damping ratio is so high that for any frequency above zero, the system gain is decreasing.

A similar analysis can be undertaken for the flight-path angle control. However, this exercise is more complex because this channel of the control is not a single-input single-output system and is not conducive to a closed form expression for the errors. MATLAB™ is used to compute the frequency response. Figure 65 is a Bode chart showing the flight-path sensitivity to a $\Delta\dot{u}$ oscillatory failure.

The flight-path sensitivity to $\Delta\dot{\alpha}$ failures is calculated with MATLAB™ in the same manner. The flight-path angle sensitivity to oscillatory $\Delta\dot{\alpha}$ failures is shown in Figure 66. As previously observed, $\Delta\dot{\alpha}$ failures have no effect on speed tracking.

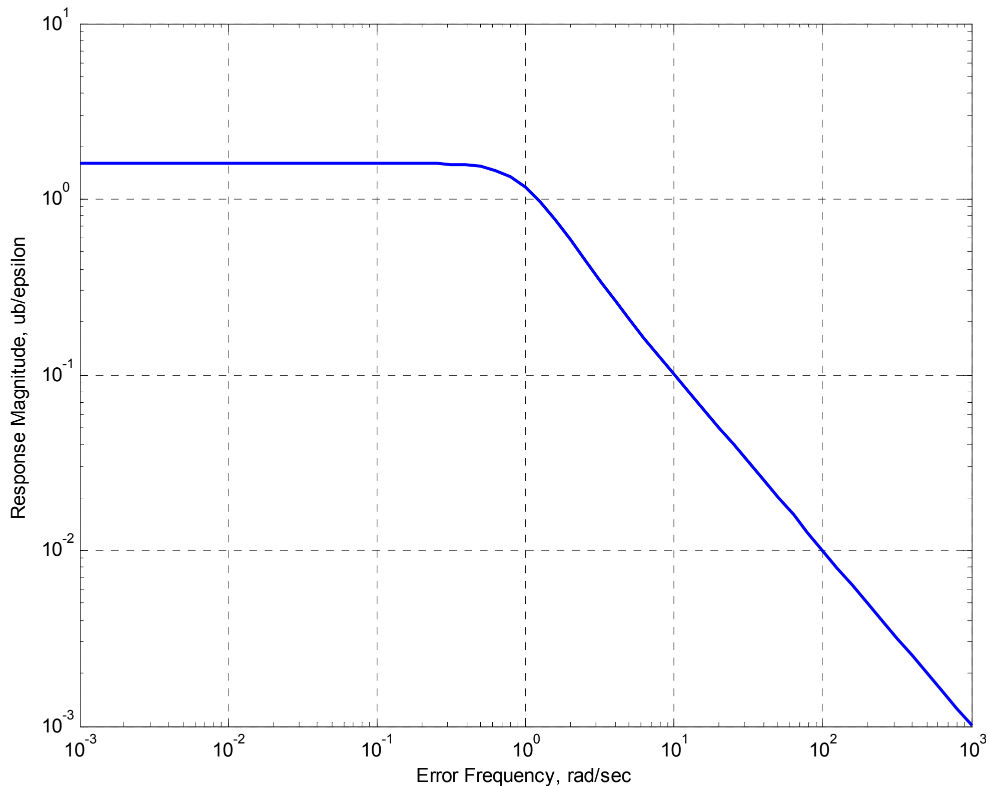


Figure 64: Speed sensitivity to oscillatory $\Delta\dot{u}$ errors from the neural network

The key thought to take away from this discussion is that the overall system response to a network failure is proportional to the output scaling. The output scaling is derived from the range of Δ's accommodated during training. A network of this architecture designed to accommodate aggressive maneuvering will be susceptible to large bounded speed errors in the event of an undetected uncorrected network malfunction. A corollary to this is that the designer can make the response to undetected network benign if the control system is designed for benign maneuvers. It must be reiterated that the training maneuvers used throughout all of these sample cases were specifically chosen to be aggressive for the purposes of exploring the boundaries of this design paradigm. A network trained specifically for an easy-to-fly General Aviation aircraft

would likely be designed for docile flight, with lower values experienced for $\Delta\dot{u}$ and $\Delta\dot{\alpha}$. This sort of flight would permit smaller scaling factors for $\Delta\dot{u}$ and $\Delta\dot{\alpha}$, and consequently the controller would experience less susceptibility to speed tracking errors in the event of a neural network failure.

Thus, two new design parameters emerge for the design of the control system: scale factors for $\Delta\dot{u}$ and scale factors for $\Delta\dot{\alpha}$. These parameters determine the aggressiveness accommodated by the neural network, with the tradeoff that large scaling factors for $\Delta\dot{u}$ will result in speed excursions proportional to $\Delta\dot{u}$ in the event of a neural network failure of the $\Delta\dot{u}$ channel. As already discussed, the parameters ς_u, ω_{n_u}, ς_γ, and ω_{n_γ} determine the responsiveness of the system.

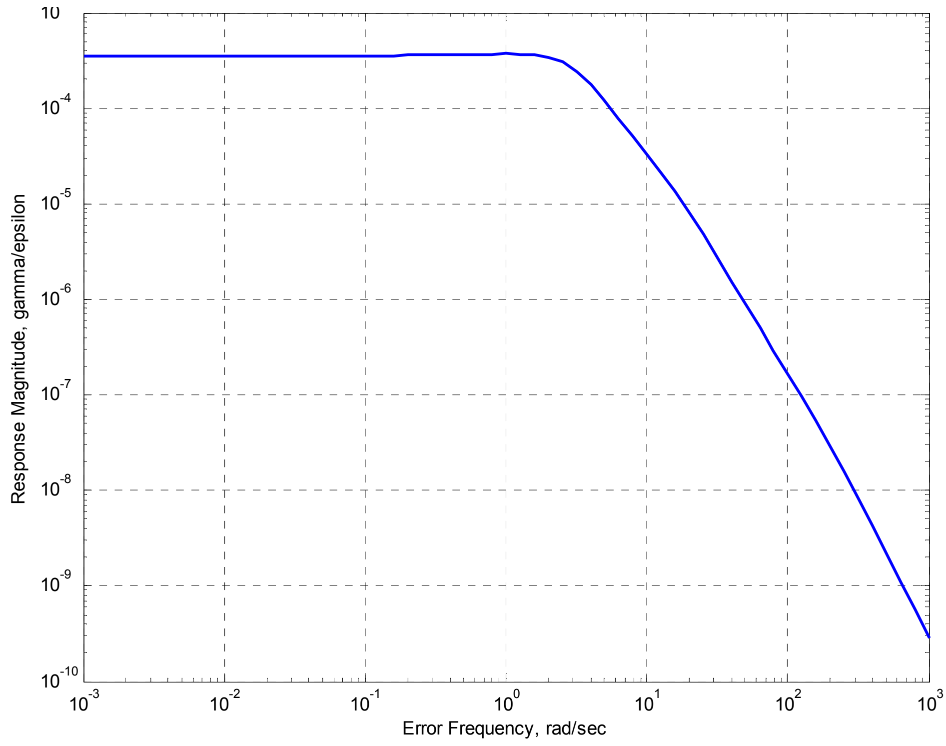


Figure 65: Flight-path sensitivity to oscillatory $\Delta\dot{u}$ errors from the neural network

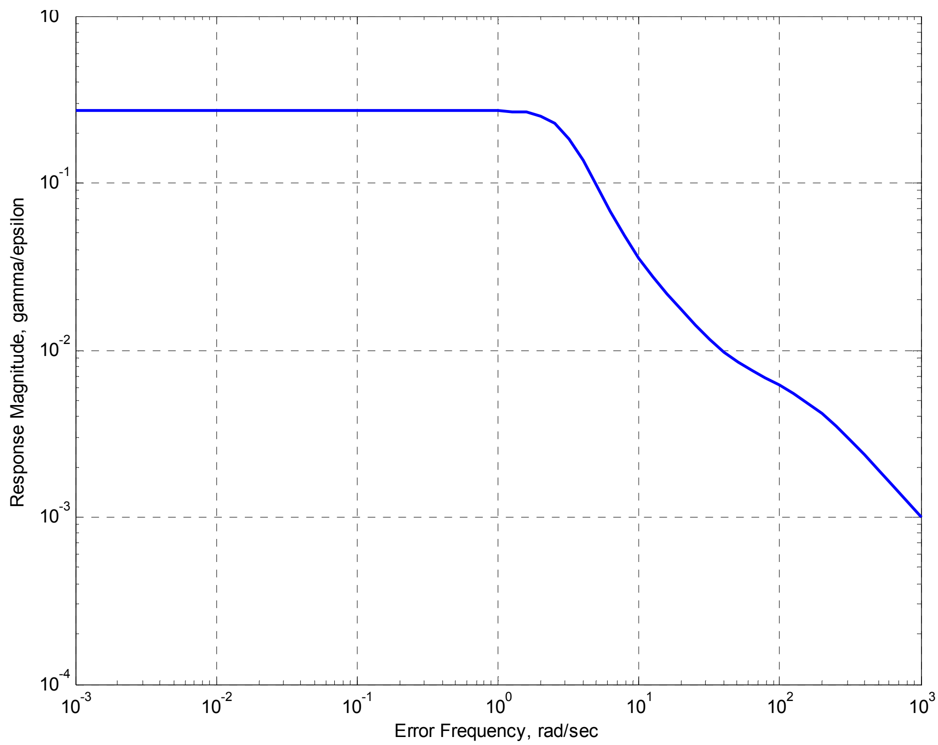


Figure 66: Flight-path sensitivity to oscillatory $\Delta\dot{\alpha}$ errors from the neural network

CHAPTER 5

CONCLUSIONS

5.1 Summary of Results

An adaptive controller is developed that meets the research objectives identified in subchapter 1.2. By building upon the work done by Calise, Rysdyk, Steck, Rokhsaz, and others, the present work extends the state of the art by implementing a closed-form expression for the inverse transfer function and using a recursive neural network to account for dynamics in the modeling and inversion errors.

Simulation results are shown to substantiate the current controller does the following:

1. Separates the normally coupled modes of the aircraft. The classic phugoid and short period modes typical of General Aviation aircraft are eliminated.
2. Enables independent control of speed and flight-path angle. Abrupt commands in one parameter create only minor transients in the other.
3. Produces an overall system response that matches the desired characteristics of an arbitrary linear system.
4. Quickly adapts to unexpected changes in the physical plant. Changes in plant stability and controllability are accommodated equally well.
5. Executes quickly, enabling the overall simulation to run 1000 times faster than real time.

All sample cases show that the controller performs its intended function. Only one case indicates any degradation from the desired performance. This exception is a challenging case involving nonlinear aerodynamics where linear aerodynamics is expected, reduced control authority, and abrupt square-wave inputs.

A brief examination of the consequences of failures in the adaptive neural network is undertaken. All cases examined are bounded. No cases are seen where total loss of control occurs. Speed excursions are seen to be the primary outcome from neural network failures. These speed excursions may be a simple offset from commanded speed or an oscillatory variation, depending on the nature of the failure. Flight-path angle is shown to be more robust to network failures.

The overall objectives for a General Aviation controller are satisfied as well. Coupled with a cockpit controls configuration such as in Hawker Beechcraft Corporation's CJ-144, the current controller will permit the pilot to directly control the high-level performance variables, speed and flight-path angle, in an intuitive fashion. Training time and time spent on maintaining proficiency should be reduced. Safety is enhanced by enabling predictable flying characteristics regardless of the aircraft failure state. These attributes should contribute to encouraging more people to enter General Aviation by reducing some of the significant barriers to personal flight.

5.2 Recommendations for Future Study

The objectives for this research are met. However, several areas of interest surfaced in the course of this research. While outside the scope of this effort, investigation of the following topics might be of interest to others:

1. Implement the current controller on CJ-144. Funding for this effort will be required. It would be of interest to compare the current controller in back-to-back flights with the baseline CJ-144 controller and the present WSU Adaptive Bias Controller derived by Steck and Rokhsaz [12].

2. Develop an improved linear prefilter using robust control theory. It would be interesting to develop an H_2 or H_∞ controller to mitigate some resonance evident arising from the gamma filter within the linear prefilter.
3. Extend the current method to include lateral-directional modes of the aircraft.
4. Investigate the resilience to other nonlinear effects known to be present in real airplane implementations. Of particular interest would be the effects of time delays in aircraft actuators (both first-order lags and pure time delays) and mechanical backlash.
5. Investigate the use of observers to synthesize certain aircraft state variables, with the goal of reducing the sensor suite required on the airplane.
6. Consider a committee network to provide redundancy in the neural network.

While not an exhaustive list, these are the topics that are most evident from the current research.

REFERENCES

REFERENCES

[1] *General Aviation Statistical Databook 2005*, General Aviation Manufacturers Association, 2006.

[2] Holmes, B.J., Durham, M.H., and Tarry, S., "The Small Aircraft Transportation System Concept and Technologies," AIAA 2003-2510, 2003.

[3] Beringer, D.B., "Applying Performance-Controlled Systems, Fuzzy Logic, and Fly-by-Wire Controls to General Aviation," DOT/FAA/AM-02/7, 2002.

[4] Duerksen, N., "Fuzzy Logic Adaptive Decoupled Flight Controls for General Aviation Airplane", Doctoral Dissertation, Department of Aerospace Engineering, Wichita State University, Spring 1996.

[5] Calise, A.J,. and Rysdyk, R.T., "Nonlinear Adaptive Flight Control Using Neural Networks," *IEEE Control Systems*, December 1998, pp. 14-25.

[6] MacFarland, M.B., and Calise, A.J., "Adaptive Nonlinear Control of Agile Antiair Missiles Using Neural Networks," *IEEE Transactions On Control Systems Technology*, Vol. 8, No. 5, September 2000, pp. 749-756.

[7] Hovakimyan, N., Nardi, F. Calise, A.J., and Kim N., "Adaptive Output Feedback Control of Uncertain Nonlinear Systems Using Single-Hidden-Layer Neural Networks," *IEEE Transactions On Neural Networks*, Vol. 13, No. 6, November 2002, pp 1420-1431.

[8] Rysdyk, R., and Calise, A.J., "Robust Nonlinear Adaptive Flight Controls for Consistent Handling Qualities," *IEEE Transactions On Control Systems Technology*, Vol. 13, No. 6, November 2005, pp. 896-910.

[9] Rysdyk, R., and Agarwal, R.K., "Nonlinear Adaptive Flight-path and Speed Control Using Energy Principles," AIAA 2002-4440, August 2002.

[10] Lambregts, A., "Vertical flight-path and Speed Autopilot Design using Total Energy Principles," *AIAA Conference Proceedings*, AIAA Paper 83-2239CP, August 1983.

[11] Steck, J., and Rokhsaz, K., "Simulation and Flight Test Assessment of Safety Benefits and Certification Aspects of Advanced Flight Control Systems," Briefing paper for FAA Project Restart Meeting, November 2001.

[12] Pesonen, U.J. et al., "Adaptive Neural Network Inverse Controller for General Aviation Safety," *Journal Of Guidance, Control, And Dynamics*, Vol. 27, No. 3, May-June 2004, pp. 434-443.

[13] Steck, J. et al., "Simulation and Flight Test Assessment of Safety Benefits and Certification Aspects of Advanced Flight Control Systems," U.S. Department of Transportation Report DOT/FAA/AR-00-C-WSU-00-14, December 2002.

[14] Steck, J.E., Rokhsaz, K., Pesonen, U.J., Mochrie, S., and Maxfield, M. "Pilot Evaluation of an Adaptive Controller on a General Aviation SATS Testbed Aircraft," AIAA 2004-5239, August 2004.

[15] Hess, D.E., Faller, W.E., Fu, T.C., and Ammeen, E.S., "Improved Simulation of Ship Maneuvers Using Recursive Neural Networks," AIAA 2006-1481, 2006.

[16] Anon., *Beechcraft Bonanza F33A/F33C Acrobatic Pilot's Operating Handbook*, Beech Aircraft Company, 1994.

[17] Zhou, K., *Essentials of Robust Control*, Prentice Hall, New Jersey, 1998.

[18] Roskam, J., *Airplane Flight Dynamics and Automatic Flight Controls*, DAR Corporation, Lawrence, Kansas, 2001.

[19] Fausett, L., *Fundamentals of Neural Networks*, Prentice-Hall, Upper Saddle River, New Jersey, 1994.

[20] Deitel, H.M., *C: How to Program*, Pearson Education Inc., New Jersey, 2004.

[21] Burl, J.B., *Linear Optimal Control*, Addison-Wesley, Menlo Park, California, 1999.

[22] Brogan, W., *Modern Control Theory*, Prentice-Hall, New Jersey, 1991.

[23] Kreyszig, E., *Advanced Engineering Mathematics*, John Wiley and Sons, New York, 1999.

[24] Deng, H., and Li, H., "A Novel Neural Approximate Inverse Control for Unknown Nonlinear Discrete Dynamical Systems," *IEEE Transactions on Systems, Man and Cybernetics -- Part B: Cybernetics*, Vol. 35, No. 1, February 2005, pp. 115-123.

[25] Davis, C.J., "Solving and Inverse Control Problem using Predictive Methods with Application to an Aircraft Accident," AIAA-99-4220, 1998.

[26] Deb, D., Tao, G., Burkholder, J.O., and Smith, D., "An Adaptive Inverse Control Scheme for Synthetic Jet Actuator Arrays," AIAA 2005-7170, 2005.

[27] Lou, K.Y., and Bryson, A.E., "Inverse and Optimal Control for Precision Aerobatic Maneuvers," *Journal of Guidance, Control, and Dynamics*, Vol. 19, No. 2, March-April 1996, pp. 483-488.

[28] Carberra, J.B.D., and Nerendra, K.S., "Issues in the Application of Neural Networks for Tracking Based on Inverse Control," *IEEE Transactions On Automatic Control*, Vol. 44, No. 11, pp. 2007-2027.

[29] de Rivals-Mazeres, G., Yim, W., Mora-Camino, F., and Singh, S.N., "Dual Mode Inverse Control and Stabilization of Flexible Space Robots," AIAA-97-3559, 1997.

[30] Bruce, K.R., "Integrated Autopilot/Autothrottle Based on a Total Energy Control Concept: Design and Evaluation of Additional Autopilot Modes," NASA Contractor Report 4131, 1988.

[31] Kurdjukov, A.P., Natchinkina, G.N., and Shevtchenko, A.M., "Energy Approach to Flight Control," AIAA-98-4211, 1998.

APPENDICES

APPENDIX A

STATE SPACE REPRESENTATION FOR THE LINEARIZED AIRCRAFT MODEL

The state space equations for the linear representation of the airplane, $\mathbf{P}_A(s)$, are derived in this appendix. The state space representation is an essential ingredient in solving for the inverse transfer function, $\mathbf{G}^{-1}(s)$.

The starting point for this derivation is the perturbed longitudinal equations of motion, found in equations (5.30a), (5.30b), and (5.30c) of Roskam [18]:

$$\dot{u} = -g\theta\cos\theta_1 + X_u u + X_{T_u} u + X_\alpha \alpha + X_{\delta_e}\delta_e \tag{A1}$$

$$U_1\dot{\alpha} - U_1\dot{\theta} = -g\theta\sin\theta_1 + Z_u u + Z_\alpha \alpha + Z_{\dot{\alpha}}\dot{\alpha} + Z_q\dot{\theta} + Z_{\delta_e}\delta_e \tag{A2}$$

$$\ddot{\theta} = M_u u + M_{T_u} u + M_\alpha \alpha + M_{\dot{\alpha}}\dot{\alpha} + M_q\dot{\theta} + M_{\delta_e}\delta_e \tag{A3}$$

The definition of the dimensional stability derivatives can be found in Roskam, Table 5.1.

Equations (A1), (A2), and (A3) can be rearranged to move all terms involving accelerations to the left-hand side. Control derivatives related to thrust are also incorporated.

$$\dot{u} = \left(X_u + X_{T_u}\right)u + X_\alpha \alpha - g\theta\cos\theta_1 + X_{\delta_e}\delta_e + X_{\delta_t}\delta_t \tag{A4}$$

$$\left(U_1 - Z_{\dot{\alpha}}\right)\dot{\alpha} = Z_u u + Z_\alpha \alpha + \left(Z_q + U_1\right)\dot{\theta} - g\theta\sin\theta_1 + Z_{\delta_e}\delta_e + Z_{\delta_t}\delta_t \tag{A5}$$

$$\ddot{\theta} = M_u u + M_{T_u} u + M_\alpha \alpha + M_{\dot{\alpha}}\dot{\alpha} + M_q\dot{\theta} + M_{\delta_e}\delta_e + M_{\delta_t}\delta_t \tag{A6}$$

Equations (A4), (A5), and (A6) can now be rewritten in matrix format. Note that row four in equation (A7) simply expresses the relationship that $q = \dot{\theta}$.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & (U_1 - Z_{\dot{\alpha}}) & 0 & 0 \\ 0 & -M_{\dot{\alpha}} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \frac{d}{dt} \begin{Bmatrix} u \\ \alpha \\ q \\ \theta \end{Bmatrix} = \begin{bmatrix} (X_u + X_{T_u}) & X_\alpha & 0 & -g\cos\theta_1 \\ Z_u & Z_\alpha & (Z_q + U_1) & -g\sin\theta_1 \\ (M_u + M_{T_u}) & (M_\alpha + M_{T_\alpha}) & M_q & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{Bmatrix} u \\ \alpha \\ q \\ \theta \end{Bmatrix}$$

$$+ \begin{bmatrix} X_{\delta_e} & X_{\delta_t} \\ Z_{\delta_e} & Z_{\delta_e} \\ M_{\delta_e} & M_{\delta_t} \\ 0 & 0 \end{bmatrix} \begin{Bmatrix} \delta_e \\ \delta_t \end{Bmatrix} \tag{A7}$$

$$\text{Let } \mathbf{M}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & (U_1 - Z_{\dot{\alpha}}) & 0 & 0 \\ 0 & -M_{\dot{\alpha}} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{A8}$$

$$\mathbf{A}_1 = \begin{bmatrix} (X_u + X_{T_u}) & X_\alpha & 0 & -g\cos\theta_1 \\ Z_u & Z_\alpha & (Z_q + U_1) & -g\sin\theta_1 \\ (M_u + M_{T_u}) & (M_\alpha + M_{T_\alpha}) & M_q & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \tag{A9}$$

$$\mathbf{B}_1 = \begin{bmatrix} X_{\delta_e} & X_{\delta_t} \\ Z_{\delta_e} & Z_{\delta_e} \\ M_{\delta_e} & M_{\delta_t} \\ 0 & 0 \end{bmatrix}, \tag{A10}$$

$$\mathbf{x} = \begin{Bmatrix} u \\ \alpha \\ q \\ \theta \end{Bmatrix}, \tag{A10}$$

$$\text{and } \mathbf{u} = \begin{Bmatrix} \delta_e \\ \delta_t \end{Bmatrix}. \tag{A11}$$

Using this notation,

$$\mathbf{M}_1 \dot{\mathbf{x}} = \mathbf{A}_1 \mathbf{x} + \mathbf{B}_1 \mathbf{u} \tag{A12}$$

Both sides of equation (A12) can be pre-multiplied by $\mathbf{M}_1^{-1}$.

$$\mathbf{M}_1^{-1}\mathbf{M}_1\,\dot{\mathbf{x}} = \mathbf{M}_1^{-1}\mathbf{A}_1\,\mathbf{x} + \mathbf{M}_1^{-1}\mathbf{B}_1\,\mathbf{u} \tag{A13}$$

Using the following definitions

$$\mathbf{A}_{\mathrm{A}} \equiv \mathbf{M}_1^{-1}\mathbf{A}_1 \tag{A14}$$

$$\mathbf{B}_{\mathrm{A}} \equiv \mathbf{M}_1^{-1}\mathbf{B}_1 \tag{A15}$$

equation (A13) can be rewritten as

$$\dot{\mathbf{x}} = \mathbf{A}_{\mathrm{A}}\,\mathbf{x} + \mathbf{B}_{\mathrm{A}}\,\mathbf{u} \tag{A16}$$

Equation (A16) is the desired form for the first part of the state space representation. To complete the state space representation, an expression for **y** is required in the form of

$$\mathbf{y} = \mathbf{C}_{\mathrm{A}}\,\mathbf{x} + \mathbf{D}_{\mathrm{A}}\,\mathbf{u} \tag{A17}$$

For the case where it is desired for **y** to be the state variables u, α, q, and θ:

$$\mathbf{C}_{\mathrm{A}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{I} \tag{A18}$$

and

$$\mathbf{D}_{\mathrm{A}} = \mathbf{0} \tag{A19}$$

For the case where it is desired for **y** to be only u and γ:

$$\mathbf{C}_{\mathrm{A}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \tag{A20}$$

and

$$\mathbf{D}_{\mathrm{A}} = \mathbf{0} \tag{A21}$$

For the case where it is desired for **y** to be u and α:

$$\mathbf{C}_A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad \text{(A22)}$$

and

$$\mathbf{D}_A = \mathbf{0} \quad \text{(A23)}$$

An augmented matrix can also be constructed that includes the state variables and γ:

$$\mathbf{C}_A = \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \hdashline 0 & -1 & 0 & 1 \end{array}\right] \quad \text{(A24)}$$

and

$$\mathbf{D}_A = \mathbf{0} \quad \text{(A25)}$$

In any of these cases, a complete state space representation of the linearized aircraft model is available. This state space model can be abbreviated with the notation

$$\mathbf{P}_A(s) = \left[\begin{array}{c|c} \mathbf{A}_A & \mathbf{B}_A \\ \hline \mathbf{C}_A & \mathbf{D}_A \end{array}\right] \quad \text{(A24)}$$

APPENDIX B

EQUATIONS OF MOTION USED IN MODELING AIRCRAFT FLIGHT DYNAMICS

Equations (B1) through (B37) summarize the equations utilized in the simulation to solve the flight dynamics of the aircraft. Note that full coupling between axes is included. These equations are adapted from those found in Roskam [18]. Subscript s denotes a stability axis property. Subscript b denotes a body axis property.

$$\alpha = \tan^{-1}\left(\frac{W_b}{U_b}\right) \tag{B1}$$

$$\beta = \tan^{-1}\left(\frac{V_b}{U_b}\right) \tag{B2}$$

$$V_p = \sqrt{U_b^2 + V_b^2 + W_b^2} \tag{B3}$$

$$P_s = P_b \cos\alpha + P_b \sin\alpha \tag{B4}$$

$$Q_s = Q_b \tag{B5}$$

$$R_s = -R_b \sin\alpha + R_b \cos\alpha \tag{B6}$$

$$\dot{\alpha} = \frac{\left(\dot{W}_b U_b - W_b \dot{U}_b\right)}{U_b^2} \tag{B7}$$

$$\hat{\dot{\alpha}} = \frac{\dot{\alpha}\,\bar{c}}{2V_p} \tag{B8}$$

$$\hat{p} = \frac{P_s\, b}{2V_p} \tag{B9}$$

$$\hat{q} = \frac{Q_s\, \bar{c}}{2V_p} \tag{B10}$$

$$\hat{r} = \frac{R_s\, b}{2\, V_p} \tag{B11}$$

$$C_L = C_{L_0} + C_{L_\alpha}\, \alpha + C_{L_{\delta_e}}\, \delta_e + C_{L_{\delta tab}}\, \delta_{tab} + C_{L_{\dot{\alpha}}}\, \hat{\dot{\alpha}} + C_{L_q}\, \hat{q} \tag{B12}$$

$$C_M = C_{M_0} + C_{M_\alpha}\, \alpha + C_{M_{\delta_e}}\, \delta_e + C_{M_{\delta tab}}\, \delta_{tab} + C_{M_{\dot{\alpha}}}\, \hat{\dot{\alpha}} + C_{M_q}\, \hat{q} \tag{B13}$$

$$C_D = C_{D_0} + C_{D_K}\, C_L^2 \tag{B14}$$

$$T = f(PLA, V_p, \rho) \tag{B15}$$

$$C_y = C_{y_0} + C_{y_\beta}\, \beta + C_{y_{\delta_r}}\, \delta_r + C_{y_{\delta_a}}\, \delta_a + C_{y_p}\, \hat{p} + C_{y_r}\, \hat{r} \tag{B16}$$

$$C_l = C_{l_0} + C_{l_\beta}\, \beta + C_{l_{\delta_r}}\, \delta_r + C_{l_{\delta_a}}\, \delta_a + C_{l_p}\, \hat{p} + C_{l_r}\, \hat{r} \tag{B17}$$

$$C_n = C_{n_0} + C_{n_\beta}\, \beta + C_{n_{\delta_r}}\, \delta_r + C_{n_{\delta_a}}\, \delta_a + C_{n_p}\, \hat{p} + C_{n_r}\, \hat{r} \tag{B18}$$

$$F_{x_b} = \overline{q} S (C_L \sin\alpha - C_D \cos\alpha) - W \sin\Theta + T \cos\alpha \tag{B19}$$

$$F_{y_b} = \overline{q} S C_y + W \sin\Phi \cos\Theta \tag{B20}$$

$$F_{z_b} = \overline{q} S \left(-C_L \cos\alpha - C_D \sin\alpha\right) + W \cos\Theta \cos\Phi + T \sin\alpha \tag{B21}$$

$$L_s = \overline{q} S b C_l \tag{B22}$$

$$M_s = \overline{q} S \overline{c} C_M \tag{B23}$$

$$N_s = \overline{q} S b C_n \tag{B24}$$

$$L_b = L_s \cos\alpha - N_s \sin\alpha \tag{B25}$$

$$M_b = M_s \tag{B26}$$

$$N_b = L_s \sin\alpha + N_s \cos\alpha \tag{B27}$$

$$\dot{U}_b = F_{x_b} / m + V_b R_b - W_b Q_b \tag{B28}$$

$$\dot{V}_b = F_{y_b} / m - U_b R_b + W_b P_b \tag{B29}$$

$$\dot{W}_b = F_{z_b} / m + U_b Q_b - V_b P_b \tag{B30}$$

$$\dot{P}_b = \frac{\left(\begin{array}{c} I_{zz_b} L_b + I_{xz_b} N_b + \left(I_{xz} I_{zz} - I_{yy} I_{xz} + I_{xz}^2\right) P_b Q_b + \\ \left(I_{yy} I_{zz} - I_{zz}^2 - I_{xz}^2\right) R_b Q_b \end{array} \right)}{\left(I_{xx_b} I_{zz_b} - I_{xz_b}^2\right)} \tag{B31}$$

$$\dot{Q}_b = \frac{M_s - \left(I_{xx_b} - I_{zz_b}\right) P_b R_b - I_{xz_b} \left(P_b^2 - R_b^2\right)}{I_{yy_b}} \tag{B32}$$

$$\dot{R}_b = \frac{\left(\begin{array}{c} I_{xz_b} L_b + I_{xx_b} N_b + \left(I_{xz}^2 - I_{yy} I_{xx} + I_{xx}^2\right) P_b Q_b + \\ \left(I_{yy} I_{xz} - I_{zz} I_{xz} - I_{xz} I_{xx}\right) R_b Q_b \end{array} \right)}{\left(I_{xx_b} I_{zz_b} - I_{xz_b}^2\right)} \tag{B33}$$

$$\dot{\Phi} = P_b + Q_b \sin\Phi \tan\Theta + R_b \cos\Phi \tan\Theta \tag{B34}$$

$$\dot{\Theta} = Q_b \cos\Phi - R_b \sin\Phi \tag{B35}$$

$$\dot{\Psi} = \left(Q_b \sin\Phi + R_b \cos\Phi\right) / \cos\Theta \tag{B36}$$

$$\dot{h} = U_b \sin\Theta - V_b \cos\Theta \sin\Phi - W_b \cos\Theta \cos\Phi \tag{B37}$$

An Adams predictor-corrector method is used for solving these equations. For each computation cycle, the values of the state variables are predicted using four prior timeframes. These predicted values are then used to start an iterative calculation of the equations of motion for this timeframe. Iteration is required because, in general, all state variables are functions of one another. The iteration continues until the value of each state variable converges. Practice shows that four iterations are sufficient. After four iterations, a final correction step takes place.

The predictor-corrector method is not self-starting. However, it can be started by assuming that the system is in equilibrium, and all accelerations are zero. Initial conditions for rates and displacements are determined from the solution for the initial trim state. These initial

conditions are used to preload the predictor-corrector, so that the algorithm has values for four "fictitious" time states prior to the starting time, all with the same equilibrium values.

Appendix C

Program Listing

A complete listing of the simulation computer code follows. This program is written in ANSI C for portability. The individual functions are listed alphabetically.

Function "alpha_limiter3.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"


double alpha_limiter3(double alphadot_command,double alpha,double alphamin,
double alphamax,
                          double alphadot_current,double dt, double mytime,
FILE *logfilePtr)
{
/*          Copyright 2007 Hugh S. Bruner             */
      double projected_alpha;
      double alphadot_limited;
      static int latched=0;
      static double x,xold = 0., xdot,xdotold=0.;
      double alphamax_threshold,alphamin_threshold;
      double A,B,C1,C2;
      double zeta = .7, omegan = 20.;

      A = dt*(1. - 2.*zeta*omegan*dt);
      B = (1. - dt*dt*omegan*omegan);
      C1 = dt*dt*omegan*omegan*alphamax;
      C2 = dt*dt*omegan*omegan*alphamin;

      alphamax_threshold = A*alphadot_current + B*alpha + C1;
      alphamin_threshold = A*alphadot_current + B*alpha + C2;

      if (!latched)
      {
            projected_alpha = alpha + alphadot_command*dt;
      }
      else
      {
            projected_alpha = alpha + 2.*alphadot_command*dt;
      }



      if ((projected_alpha > alphamax_threshold) && (alphadot_command > 0.))
                  //  Positive limits  //
      {
```

```
		if (!latched)
		{
			xold = alpha;
			xdotold = alphadot_current;
			latched = 1;
//			fprintf(logfilePtr,"\nLatched at t = %f",mytime);
			fprintf(logfilePtr,"\nalphamax = %f\talpha_threshold = 
%f",alphamax,alphamax_threshold);
		}

		xdot = (1. - 2.*zeta*omegan*dt)*xdotold - omegan*omegan*dt*xold + 
omegan*omegan*dt*alphamax;
		x = xdotold*dt + xold;
		alphadot_limited = xdot;
		xold = x;
		xdotold = xdot;

//		fprintf(logfilePtr,"\nLimiting alphadot at t = %f",mytime);
	}
	else if ((projected_alpha < alphamin_threshold) && (alphadot_command < 
0.))				//  Negative limits  //
	{
		if (!latched)
		{
			xold = alpha;
			xdotold = alphadot_current;
			latched = 1;
//			fprintf(logfilePtr,"\nLatched at t = %f",mytime);
		}

		xdot = (1. - 2.*zeta*omegan*dt)*xdotold - omegan*omegan*dt*xold + 
omegan*omegan*dt*alphamin;
		x = xdotold*dt + xold;
		alphadot_limited = xdot;
		xold = x;
		xdotold = xdot;

//		fprintf(logfilePtr,"\nLimiting alphadot at t = %f",mytime);
	}

	else										// No 
limits at all   //
	{
		alphadot_limited = alphadot_command;
		latched = 0;
		xold = 0;
		xdotold = 0;
	}

	return alphadot_limited;

}
```

Function "autopilot.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Autopilot function
                                    */
/*
                  Changed 12/5/2006 to include pla
                  Changed 5/1/2006 to provide random command generation
                  Updated 12/19/2006 to move command generation to a stand-
alone function
                  Copyright 2007 Hugh S. Bruner

                                          */

int autopilot(Network *myNetPtr, const double *mytimePtr, const double dt,
const double vkeastrim, double utrimfps,
                    const int manuever, double de1, double da1, double dr1,
double pla1,
                    double *dePtr, double *daPtr, double *drPtr, double
*plaPtr, double *ucommandPtr, double *gammacommandPtr,
                    FILE *filePtr, FILE *errFilePtr, FILE *modeloutPtr,
double time_to_learn, double time_to_control)
{

//    Declarations that are basic to autopilot
      float ucom;
      float gammacom;
      double dde;                               // Incremental elevator
deflection computed by autopilot
      double dpla;                              // Incremental power lever
angle computed by autopilot

/*
      The function command_input below returns u commands (ucom) and gamma
commands (gammacom) based
      on a predefined set of manuevers.  The manuevers are numbered and the
value of variable manuever
      determines which manuever is activated.
*/
      command_input(mytimePtr, dt, &ucom, &gammacom, manuever, errFilePtr);

      *ucommandPtr = ucom;
      *gammacommandPtr = gammacom;

/*
      Function DEF_JRcontrol contains the essence of the inverse controller
with adaptive compensation.
      Calling the function below computes increments to elevator (dde) and
PLA (dpla) that in turn are
      added to the trim values for elevator (de1) and PLA (pla1).  The inputs
to the function include
      speed perturbation commanded off of trim (ucom) and the gamma commanded
(gammacom).  The terms
```

```
      (vtrue - U1) and (theta - alpha) are the speed and gamma feedback,
respectively
*/
      DEF_JRcontrol(myNetPtr,mytimePtr,dt,time_to_learn,time_to_control,ucom,
gammacom, (float)(vtrue - U1),
                  (float)(theta - alpha), &dde, &dpla, filePtr,errFilePtr,
modeloutPtr);


            *dePtr = de1 + dde;                              // add increment
in elevator to trim elevator position
            *plaPtr = pla1 + dpla;                     // add increment in PLA
to trim PLA setting
            *drPtr = (double)0;                              // rudder is not
used
            *daPtr = (double)0;                              // aileron is not
used

//    Apply elevator and PLA limits here, if desired.  For dissertation, they
are not.


      return 0;
}
```

Function "backprop.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

float backprop(Network *netPtr, TrainingPair *dataPtr, float learning_rate)
{
/*
            This function does the calculations involved in backpropagating
            through the neural network.

            Copyright 2007 Hugh S. Bruner
*/
      short k;
      short l;
      short m;
      short n;
      float tout;
      float sum;
      float errsum = 0;
      Neuron *neuronPtr;
      Neuron *otherneuronPtr;

/*          Calculate the deltas for the net first:
      */

//    For each layer, k, starting at the end and going backwards
      for (k=netPtr->K; k > 1; k--)
      {
//          For each neuron in the layer, n
            for (n=1; n <= netPtr->nk[k]; n++)
            {
                  neuronPtr = netPtr->layerPtr[k]->neuronPtr[n];
//                Calculate the delta
//                If  it is the last layer,
                  if (k == netPtr->K)
                  {

//                      Scale the training output
                        tout = (dataPtr->output[n-1] - netPtr->w0[n])/netPtr-
>w1[n];
//                      printf("\ntout = %f",tout);
//                      Calculate the delta
                        neuronPtr->delta = (tout - neuronPtr->y)*neuronPtr-
>yprime;
//                      Calculate contribution to the sum of the errors
squared
                        errsum = errsum + (tout - neuronPtr->y)*(tout -
neuronPtr->y);
                  }
//                Otherwise,
                  else
                  {
                        sum = (float)0.;
                        for (l=1; l <=netPtr->nk[k+1]; l++)
                        {
```

```
                                otherneuronPtr = netPtr->layerPtr[k+1]-
>neuronPtr[l];
                                sum += otherneuronPtr->delta*otherneuronPtr-
>w[n];
                          }
                          neuronPtr->delta = sum*neuronPtr->yprime;
                    }
//                  End-if on layers

             }
//           End-for on neurons
      }
//    End-for on layers

//    Calculate the average squared-error metric
      errsum = errsum/(float)netPtr->N;


/*                  Now adjust the network weights
      */

//    Again, For each layer, k, starting at the end and going backwards
      for (k=netPtr->K; k > 1; k--)
      {
//           For each neuron in the layer, n
             for (n=1; n <= netPtr->nk[k]; n++)
             {
                    neuronPtr = netPtr->layerPtr[k]->neuronPtr[n];
//                  Change the bias neuron
                    neuronPtr->w[0] = neuronPtr->w[0] +
learning_rate*neuronPtr->delta;

//                  For each connection, m
                    for (m=1; m <= neuronPtr->M; m++)
                    {
//                        ~~~~~~~~
                          otherneuronPtr = netPtr->layerPtr[k-1]->neuronPtr[m];
                          neuronPtr->w[m] = neuronPtr->w[m] +
learning_rate*neuronPtr->delta*otherneuronPtr->y;

                    }
//                  End-for on conections
             }
//           End-for on neurons
      }
//    End-for on layers

      return errsum;
}

/*           End function backprop
      */
```

Function "case_data_write.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"


int case_data_write(FILE *ffilePtr, char *descriptorPtr)
{
/*          Function to print out inputs for verification
            Copyright 2007 Hugh S. Bruner
                                */
//    Declarations
//                              1         2         3         4         5
//                     12345678901234567890123456789012345678901234567890
      char format1[50] = "     %10s = %f + %f*alpha + %f*tcp + %f*alpha*tcp";
      char format2[25] = "     %10s = %10f %-10s";
      char format3[40] = "     %10s = %10f + %-10f*(CL - %f)^2";
      char format4[45] = "     %10s = %5.2f + %6.3f/deg elevator";
      char format5[30] = "     %10s = %f + %f*tcp";

//    Write out the case descriptor
      fprintf(ffilePtr,"\n      %s",descriptorPtr);

//    Write out flight conditions
      fprintf(ffilePtr,"\n");

      fprintf(ffilePtr,format2,"Vkeas",vkeas,"kts");
      fprintf(ffilePtr,format2,"alt",h,"ft");
      fprintf(ffilePtr,format2,"C.G.",xcg,"cbar");

//    Write out mass properties
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format2,"Weight",weight,"lbs");
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format2,"Ixxb",ixxb,"slug ft2");
      fprintf(ffilePtr,format2,"Iyyb",iyyb,"slug ft2");
      fprintf(ffilePtr,format2,"Izzb",izzb,"slug ft2");
      fprintf(ffilePtr,format2,"Ixzb",ixzb,"slug ft2");
      fprintf(ffilePtr,"\n");

//    Write out geometry
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format2,"S ref",sref,"sq. ft.");
      fprintf(ffilePtr,format2,"c ref",cref,"feet");
      fprintf(ffilePtr,format2,"b ref",bref,"feet");
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format2,"Se",se,"sq. ft.");
      fprintf(ffilePtr,format2,"ce",ce,"feet");
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format2,"ref cg",cgref,"cbar");

//    Write out mechanical data

      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format4,"bob wt",bw0,dbwdde);
      fprintf(ffilePtr,"\n");
```

```
      fprintf(ffilePtr,format4,"dn spring",ds0,ddsdde);

//    Write out longitudinal stability data

      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"clo",clov.constant_term,clov.alpha_term,clov.
tcp_term,clov.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cla",clav.constant_term,clav.alpha_term,clav.
tcp_term,clav.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"clde",cldev.constant_term,cldev.alpha_term,cl
dev.tcp_term,cldev.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cldetab",cldetabv.constant_term,cldetabv.alph
a_term,cldetabv.tcp_term,cldetabv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cladot",cladotv.constant_term,cladotv.alpha_t
erm,cladotv.tcp_term,cladotv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"clq",clqv.constant_term,clqv.alpha_term,clqv.
tcp_term,clqv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");

      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cmo",cmov.constant_term,cmov.alpha_term,cmov.
tcp_term,cmov.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cma",cmav.constant_term,cmav.alpha_term,cmav.
tcp_term,cmav.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cmde",cmdev.constant_term,cmdev.alpha_term,cm
dev.tcp_term,cmdev.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cmdetab",cmdetabv.constant_term,cmdetabv.alph
a_term,cmdetabv.tcp_term,cmdetabv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cmadot",cmadotv.constant_term,cmadotv.alpha_t
erm,cmadotv.tcp_term,cmadotv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cmq",cmqv.constant_term,cmqv.alpha_term,cmqv.
tcp_term,cmqv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");

      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format3,"cd",cdo,cdk,cli);
      fprintf(ffilePtr,"\n");

      fprintf(ffilePtr,format5,"eps0",epsv.constant_term,epsv.tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format5,"epsa",epsv.alpha_term,epsv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");

      fprintf(ffilePtr,format2,"cheo",cheo,"/rad");
      fprintf(ffilePtr,format2,"chea",chea,"/rad");
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format2,"chede",chede,"/rad");
```

```
      fprintf(ffilePtr,format2,"chedetab",chedetab,"/rad");
      fprintf(ffilePtr,"\n");

//    Write out lateral-directional stability data

      fprintf(ffilePtr,"\n");

      fprintf(ffilePtr,format1,"cyb",cybv.constant_term,cybv.alpha_term,cybv.
tcp_term,cybv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cydr",cydrv.constant_term,cydrv.alpha_term,cy
drv.tcp_term,cydrv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cyda",cybv.constant_term,cydav.alpha_term,cyd
av.tcp_term,cydav.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cyp",cypv.constant_term,cypv.alpha_term,cypv.
tcp_term,cypv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cyr",cyrv.constant_term,cyrv.alpha_term,cyrv.
tcp_term,cyrv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");

      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cnb",cnbv.constant_term,cnbv.alpha_term,cnbv.
tcp_term,cnbv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cndr",cndrv.constant_term,cndrv.alpha_term,cn
drv.tcp_term,cndrv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cnda",cnbv.constant_term,cndav.alpha_term,cnd
av.tcp_term,cndav.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cnp",cnpv.constant_term,cnpv.alpha_term,cnpv.
tcp_term,cnpv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"cnr",cnrv.constant_term,cnrv.alpha_term,cnrv.
tcp_term,cnrv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");

      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"crb",crbv.constant_term,crbv.alpha_term,crbv.
tcp_term,crbv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"crdr",crdrv.constant_term,crdrv.alpha_term,cr
drv.tcp_term,crdrv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"crda",crbv.constant_term,crdav.alpha_term,crd
av.tcp_term,crdav.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"crp",crpv.constant_term,crpv.alpha_term,crpv.
tcp_term,crpv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
      fprintf(ffilePtr,format1,"crr",crrv.constant_term,crrv.alpha_term,crrv.
tcp_term,crrv.alpha_tcp_term);
      fprintf(ffilePtr,"\n");
```

```
	if (control_simfailure)
	{
		fprintf(ffilePtr,"\n\nFailures are enabled.");
		printf("\n\nFailures are enabled.");
	}
	else
	{
		fprintf(ffilePtr,"\n\nFailures are not enabled.");
		printf("\n\nFailures are not enabled.");
	}



	return 0;
}
```

Function "CJ144_data.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Function to load CJ-144 Data
                  */
int CJ144_data(void)
{

/*          This function contains the basic data on a Beechcraft Bonanza F-
33C.
            This data has been provided to Wichita State University under a
license
            granted by Hawker Beechcraft Corporation.  This infomation is to
be used for
            designated research purposes only, and is not to be distributed
to any
            unauthorized individual(s).
                        */

//    Declarations
      double rho;                   // density of the air

//    Mass Properties

      weight = 3300.0;
      xcg = /*  This data is proprietary to Hawker Beechcraft Corporation */
      ixxb = /*  This data is proprietary to Hawker Beechcraft Corporation */
      iyyb = /*  This data is proprietary to Hawker Beechcraft Corporation */
      izzb = /*  This data is proprietary to Hawker Beechcraft Corporation */
      ixzb = /*  This data is proprietary to Hawker Beechcraft Corporation */

//    Geometry

      sref = 177.6;
      cref = /*  This data is proprietary to Hawker Beechcraft Corporation */
      bref = 32.819;
      cgref = /*  This data is proprietary to Hawker Beechcraft Corporation
*/
      se = /*  This data is proprietary to Hawker Beechcraft Corporation */
      ce = /*  This data is proprietary to Hawker Beechcraft Corporation */

//    Mechanical

      bw0 = /*  This data is proprietary to Hawker Beechcraft Corporation */
      ds0 = /*  This data is proprietary to Hawker Beechcraft Corporation */
      gev[0] = /*  This data is proprietary to Hawker Beechcraft Corporation
*/
//    Flight Condition

      h = (float)3000.;
      rho = std_density(h);
      vkeas = 100.;
      xcg = .32;
      dx = (xcg - cgref)*cref;
```

```
//    Derivatives

      clov.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      clav.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cldev.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cldetabv.constant_term = /*  This data is proprietary to Hawker
Beechcraft Corporation */
      clqv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cladotv.constant_term = /*  This data is proprietary to Hawker
Beechcraft Corporation */

      cmov.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cmav.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cmdev.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cmdetabv.constant_term = /*  This data is proprietary to Hawker
Beechcraft Corporation */
      cmqv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cmadotv.constant_term = /*  This data is proprietary to Hawker
Beechcraft Corporation */

      cdo = /*  This data is proprietary to Hawker Beechcraft Corporation */
      cdk = /*  This data is proprietary to Hawker Beechcraft Corporation */
      cli = /*  This data is proprietary to Hawker Beechcraft Corporation */

      epsv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      epsv.alpha_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */


      crbv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      crpv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      crrv.constant_term = (/*  This data is proprietary to Hawker Beechcraft
Corporation */
      crdrv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      crdav.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cnbv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cnpv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cnrv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cybv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
```

```
      cypv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cyrv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      cydrv.constant_term = /*  This data is proprietary to Hawker Beechcraft
Corporation */

//    Hinge moments

      cheo = /*  This data is proprietary to Hawker Beechcraft Corporation */
      chede = /*  This data is proprietary to Hawker Beechcraft Corporation
*/
      chedetab = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      chea = /*  This data is proprietary to Hawker Beechcraft Corporation */

//    Limits

      max_alpha = /*  This data is proprietary to Hawker Beechcraft
Corporation */
//    max_alpha = /*  This data is proprietary to Hawker Beechcraft
Corporation */
      min_alpha = /*  This data is proprietary to Hawker Beechcraft
Corporation */

      kthrust = 1.;

      return 0;

}
/*          End of function to load CJ-144       data
                  */
```

Function “clear_the_network.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int clear_the_network(Network *myNetworkPtr)
{
/*          This function clears all the attributes associated with the
network
            but preserves the network itself.
            In contrast, function erase network clears all the attributes
plus eliminates
            the network from memory

            Copyright 2007 Hugh S. Bruner

                                                      */

//    If the network exists
      if (myNetworkPtr != NULL)
      {

//          Delete network connections
            delete_net_connections(myNetworkPtr);
//          Delete training data
            delete_training_data(myNetworkPtr);
      }
//    End-if on existence of network

      return 0;
}
/*          End function clear_the_network
                                                */
```

Function "command_input.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"



int command_input(const double *mytimePtr, const double dt, float
*ucommandPtr, float *gammacommandPtr,
                          const int manuever, FILE *errFilePtr)
{
/*          Function command_input
            Created by H.S. Bruner
            Copyright 2007 Hugh S. Bruner

            This function synthesizes command inputs in batch mode that would
normally come from
            some sort of cockpit input device, such as a stick and/or speed
control lever

*/

//    Declarations needed for command transfer function
      float au, bu, ag, bg;
      float yinu, ying;
      float denomu, denomg;
      float xnow=0;
      float Ku, Kg;
      static float xuk = 0, xuk1 = 0, xuk2= 0;
      static float xgk = 0, xgk1 = 0, xgk2= 0;
      static int firsttime = 1;                        // integer code on
first time through
      float tau1=10, tau2 = 1;
      float tau3=10, tau4 = 1.;
//    Declarations that are basic to autopilot
      float ucom=0.;
      float gammacom=0.;
      static float gammacomold = -1.;
      static int gammacomcount=0;
//    int manuever=3;
      const double pi = 3.14159;

//    If this is the first time through...
      if ((*mytimePtr-dt) < .001)
      {
            fprintf(errFilePtr,"\nAutopilot will fly manuever %d",manuever);
            printf("\nAutopilot will fly manuever %d",manuever);
            if (manuever == 6) srand((int)(h + U1)/10);
            xuk = (float)0.;              // Reinitialize on first pass
through of a simulation
            xuk1 = (float)0.;             // Reinitialize on first pass
through of a simulation
            xuk2= (float)0.;              // Reinitialize on first pass
through of a simulation
            xgk = (float)0.;              // Reinitialize on first pass
through of a simulation
```

```
            xgk1 = (float)0.;              // Reinitialize on first pass
through of a simulation
            xgk2= (float)0.;               // Reinitialize on first pass
through of a simulation
            firsttime = (float)0;
            gammacomold = -1.;
            gammacomcount = 0;
      }

//    User may enable any autopilot algorithm in this function.
//    Function returns via pointers the incremental values of elevator (dde),
//    aileron (dda), and rudder that are to be ADDED to the steady-state trim
//    values for the control deflections.


            switch(manuever)
            {

            case 1:                                            // Do nothing
scenario.  Also Steck cases 1 and 3
            case 3:

                  ucom = (float)0;
                  gammacom = 0;
                  break;

            case 2:                                            // Steck case 2

                  if (*mytimePtr <= 8.)
                  {
                        gammacom = (float)0.;
                  }
                  else
                  {
                        gammacom = (float).087266;
                  }
                  if (*mytimePtr <= 20.)
                  {
                        ucom = (float)0.;
                  }
                  else
                  {
                        ucom = (float)20.;
                  }
                  break;


            case 4:                                            // Steck case 4

                  gammacom = 0.;
                  if (*mytimePtr <= 3.)
                  {
                        ucom = (float)0.;
                  }
                  else
                  {
                        ucom = (float)-70.;
```

```
                }
                break;

            case 5:                                             // Hard pull up

                ucom = (float)0.;
                if (*mytimePtr < 5.)
                {
                      gammacom = 0;
                }
                else
                {
                      gammacom = .174;
                }
                break;

            case 6:                                             // Moderately
random inputs

                if (*mytimePtr < 5.)
                {
                      ucom = 0.;
                      gammacom = 0.;
                }
                else
                {
                      au = 1/tau1 + 1/tau2;
                      bu = 1/(tau1*tau2);
                      ag = 1/tau3 + 1/tau4;
                      bg = 1/(tau3*tau4);
                      denomu = 1. + (au + bu*dt)*dt;
                      denomg = 1. + (ag + bg*dt)*dt;
//                    pick a random number between -1 and 1
                      yinu = (float)((rand() % 200)- 100)/100;
                      ying = (float)((rand() % 200)- 100)/100;
//                    pass it through the transfer function
                      xuk = (dt*dt*yinu + xuk1*((float)2. + au*dt) -
xuk2)/denomu;
                      xgk = (dt*dt*ying + xgk1*((float)2. + ag*dt) -
xgk2)/denomg;

                      if (Utrim > 170)
                      {
                            Ku = 400.;
                            Kg = 2.5;
                      }
                      else
                      {
                            Ku = .0138*Utrim*Utrim;
                            Kg = .0147*Utrim;
                      }
                      ucom = Ku*xuk*bu;
                      gammacom = Kg*xgk*bg;
//                    preserve old values for next pass
                      xuk2 = xuk1;
                      xuk1 = xuk;
                      xgk2 = xgk1;
```

```
                        xgk1 = xgk;
                  }

                  break;
            case 7:                                         //    Increasing
step functions gamma then speed

                  if (*mytimePtr <= 100.) gammacom = 0.;
                  else if (*mytimePtr <= 120.) gammacom = .02;
                  else if (*mytimePtr <=140.) gammacom = -.03;
                  else if (*mytimePtr <=160.)  gammacom =  .04;
                  else if (*mytimePtr <=180.) gammacom = -.05;
                  else if (*mytimePtr <=200.) gammacom = 0.06;
                  else if (*mytimePtr <=220.) gammacom = -.07;
                  else if (*mytimePtr <=240.) gammacom =  .08;
                  else if (*mytimePtr <=260.) gammacom = -.09;
                  else gammacom = 0.;

                  if (*mytimePtr < 280.) ucom = 0.;
                  else if (*mytimePtr < 300.) ucom = 5.;
                  else if (*mytimePtr < 320.) ucom = -10.;
                  else if (*mytimePtr < 340.) ucom =  15.;
                  else if (*mytimePtr < 360.) ucom = -20.;
                  else if (*mytimePtr < 380.) ucom =  30.;
                  else if (*mytimePtr < 400.) ucom = -40.;
                  else if (*mytimePtr < 420.) ucom =  50.;
                  else ucom = 0.;

                  break;


            case 8:

                  if (*mytimePtr < 10.)
                  {
                        ucom = 0.;
                        gammacom = 0.;
                  }
                  else if (*mytimePtr < 110.)
                  {
                        ucom = (*mytimePtr - 10)*.40;
                        gammacom = 0.;
                  }
                  else if (*mytimePtr < 200.)
                  {
                        ucom = 40.;
                        gammacom = 0.;
                  }
                  else if (*mytimePtr < 300.)
                  {
                        ucom = 40.;
                        gammacom = -.0872;
                  }
                  else if (*mytimePtr < 400.)
                  {
                        ucom = (400. - *mytimePtr)*.40;
                        gammacom = 0.;
```

```
			}
			else
			{
				ucom = 0.;
				gammacom = 0.;
			}

			break;

		case 9:								// MATLAB
check

			ucom = sin(*mytimePtr*2.*pi/10.);
			if (ucom > 0.)
			{
				ucom = (float)5.;
			}
			else
			{
				ucom = (float)0.;
			}
			if (*mytimePtr < 2.)
			{
				gammacom = (float)0.;
			}
			else
			{
				gammacom = (float).14;
			}

			break;

		case 10:								// Square
wave on speed only

			ucom = sin(*mytimePtr*2.*pi/20.);
			if (ucom > 0.)
			{
				ucom = (float)5.;
			}
			else
			{
				ucom = (float)0.;
			}
			gammacom = (float)0.;

			break;

		case 11:								// Square
wave on gamma only

			gammacom = sin(*mytimePtr*2.*pi/10.);
			if (gammacom > 0.)
			{
				gammacom = (float).05236;		//  3.00 degrees
			}
			else
```

```
                {
                      gammacom = (float)-.05236;          // -3.00 degrees
                }
                ucom = (float)0.;

                break;

          case 12:                                              // Combined
square wave on speed and gamma

                ucom = sin(*mytimePtr*2.*pi/20.);
                if (ucom > 0.)
                {
                      ucom = (float)5.;
                }
                else
                {
                      ucom = (float)0.;
                }
                gammacom = sin(*mytimePtr*2.*pi/10.);
                if (gammacom > 0.)
                {
                      gammacom = (float).08726;
                }
                else
                {
                      gammacom = (float)-.08726;;
                }

                break;

          case 15:

                ucom = (float)0.;
                gammacom = (float)0.;

                if (*mytimePtr > 40.)
                {
                      gammacom = (float)0.;
                }
                else if(*mytimePtr > 35.)
                {
                      gammacom = (float)-.12;
                }
                else if (*mytimePtr > 30.)
                {
                      gammacom = (float)0.;
                }
                else if(*mytimePtr > 25.)
                {
                      gammacom = (float).09;
                }
                else if (*mytimePtr > 20.)
                {
                      gammacom = (float)0.;
                }
                else if(*mytimePtr > 15.)
```

```
                {
                      gammacom = (float)-.06;
                }
                else if (*mytimePtr > 10.)
                {
                      gammacom = (float)0.;
                }
                else if(*mytimePtr > 5.)
                {
                      gammacom = (float).03;
                }

                break;

          case 16:                                              // Square
wave on gamma only

                gammacom = sin(*mytimePtr*2.*pi/10.);
                if ((gammacom*gammacomold) <= 0.)
                {
                      gammacomcount++;
                      /* printf("\ngammacom = %f",gammacom);
                      printf("\ngammacomold = %f",gammacomold);
                      printf("\nproduct = %f",gammacom*gammacomold);
                      printf("\ngammacomcount = %d",gammacomcount); */
                }

                if (gammacom > 0.)
                {
                      gammacom = (float).0175*(float)gammacomcount;
                }
                else
                {
                      gammacom = (float)-.0175*(float)gammacomcount;
                }
                gammacomold = gammacom;
                ucom = (float)0.;

                break;

          case 51:                      // 100 second period sine wave on
speed.  Maintain level flight

                ucom = (float)(-10.*sin(pi*(*mytimePtr)/100.));
                gammacom = (float)0.;

                break;



          default:
// default case, as used in proposal

//              In this case, no inputs for first five seconds

                if (*mytimePtr <= 5.)
                {
```

```
                        ucom = (float)0.;
                        gammacom = (float)0.;
                  }
                  else if (*mytimePtr <100)//   followed by random commands
afterwards
                  {
                        ucom = 0.;
                        gammacom = 0.;
//                      randomCom(&ucom, &gammacom);
                  }
                  else        // What happens after 100 is determined by the
following switch
                  {

                        if (100. <= *mytimePtr && *mytimePtr <=900.)
                              {
                                    ucom = (float)(10.*sin(pi*(*mytimePtr -
100.)/100.));
                                    gammacom = (float) 0.;
                              }
                              else
                              {
                                    gammacom =(float)(.04*sin(pi*(*mytimePtr
- 900.)/50.));
                                    ucom = (float)0;
                                    if (gammacom > 0)
                                    {
                                          gammacom = (float).04;
                                    }
                                    else
                                    {
                                          gammacom = (float)-.04;
                                    }

                              }
                  }
//                      End-if
                        break;
            }
//    }

      *ucommandPtr = ucom;
      *gammacommandPtr = gammacom;

      return 0;

}
```

Function “correct.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"


/*          Function to correct a state variable
            Copyright 2007 Hugh S. Bruner
      */
double correct(const double *ydotPtr, const double *yoldPtr, const double
*dtPtr)
{
//    Declarations
      double ynew;

//    Apply fourth-order Adams predictor formula
//    correct = yold + dt * (ydot(1) * b(1) + ydot(2) * b(2) - ydot(3) * b(3)
+ ydot(4) * b(4))
      ynew = *yoldPtr + (*dtPtr) * (ydotPtr[0] * 0.375
            + ydotPtr[1] * 0.79167 - ydotPtr[2] * 0.20833 + ydotPtr[3] *
0.041667);
      return ynew;
}
/*          End of function
                        */
```

Function "create_net_connections.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int create_net_connections(Network *networkPtr, float xinputs[])
{
      /*                Copyright 2007 Hugh S. Bruner             */
      short int m;
      short int k;
      short int n;
      Layer *newLayer;

      #ifdef VERBOSE
      printf("\nEntering create_net_connections");
#endif
//    Insure number of neurons specified for first and last layer
//    are compatible with network inputs and outputs.
      networkPtr->nk[1] = networkPtr->M;
      networkPtr->nk[networkPtr->K] = networkPtr->N;

//  For each input, m
      for (m=0; m<networkPtr->M; m++)
      {
//          Assign its address to xPtr[m]
            (networkPtr->xPtr[m+1])=&xinputs[m];
#ifdef VERBOSE
            printf("\nX[%d] is at address %p",m,&xinputs[m]);
#endif
      }
//    End-for

//    For each layer, k
      for (k=1; k<=(networkPtr->K); k++)
      {
//          Create layer
#ifdef VERBOSE
            printf("\n");
            printf("\n");
            printf("\nCreating layer %d",k);
#endif
            newLayer = (Layer *)malloc(sizeof(Layer));
//          Assign layer address to layer pointer[k]
            networkPtr->layerPtr[k]=newLayer;
//          The layer inherits the number of neurons specified for this layer
in the network.
//          This may be overridden for the first and last layers
            newLayer->N = networkPtr->nk[k];
#ifdef VERBOSE
            printf("  Memory address = %p",newLayer);
#endif
//          If layer is the first layer (k=1)
            if (k==1)
            {
//                This layer will have the same number of inputs as the
network has inputs
```

```
                  newLayer->M = networkPtr->M;
//                Connectivity is LINEAR
                  newLayer->connect = LINEAR;
//                Assign address contained in each network input pointer to
input pointers in this layer
                  for (m=1;m<=networkPtr->M;m++)
                  {
                        newLayer->xPtr[m] = networkPtr->xPtr[m];
#ifdef VERBOSE
                        printf("\nNew layer input[%d] connects to address
%p",m,newLayer->xPtr[m]);
#endif
                  }
            }
//          Otherwise, for other layers
            else
            {
//                This layer has the same number of inputs as the prior layer
had outputs
                  newLayer->M = networkPtr->layerPtr[k-1]->N;
//                Connectivity is FULLY
                  newLayer->connect = FULLY;
//                Assign address contained in each output pointer of previous
layer to input pointers of this layer
                  for (m=1;m<=networkPtr->layerPtr[k-1]->N;m++)
                  {
                        newLayer->xPtr[m] = networkPtr->layerPtr[k-1]-
>yPtr[m];
#ifdef VERBOSE
                        printf("\nNew layer input[%d] connects to address
%p",m,newLayer->xPtr[m]);
#endif
                  }
            }
//          End-if

//          Populate layer

#ifdef VERBOSE
            printf("\n");
            printf("\nLayer %d Neurons:",k);
            printf("\n================");
#endif
            populate_layer(newLayer);

//          If this is the last layer (k = K)
            if (k==(networkPtr->K))
            {
//                Assign addresses contained in each output pointer of this
layer to the output pointers for the network
                  for (n=1; n<=networkPtr->N;n++)
                  {
                        networkPtr->yPtr[n] = newLayer->yPtr[n];
#ifdef VERBOSE
                        printf("\nNetwork output[%d] connects to address
%p",n,networkPtr->yPtr[n]);
#endif
```

```
                    }
              }
//            Endif
        }
//      end-for
        return 0;
}
/*                 End of function create_net_connections
                                      */
```

Function "create_new_network.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int create_new__network(Network *netPtr, FILE *filePtr, float xinputs[])
{
	/*		Copyright 2007 Hugh S. Bruner		*/
	char data_record[READRECLEN] = "";
	int tokenized = FALSE;
	int xinteger;
	int ierror;
	short int k;
	short int m;
	short int n;
#ifdef VERBOSE
	printf("\nEntering create_new_network");
	printf("\nNetwork address: %p",netPtr);
#endif
//	Read in number of network inputs, M. Print error if M > MMAX
	if ((ierror =
read_item(NULL,&xinteger,NULL,filePtr,data_record,&tokenized)) != 0)
	{
		error_handler(ierror,"reading M, create_new_network");
		return ierror;
	}
	if (xinteger > MMAX)
	{
		fprintf(stderr,"Value of %d exceeds allowable number of network
inputs",xinteger);
		return -3;
	}
	printf("\nNetwork will have %d inputs",xinteger);
	netPtr->M = xinteger;
	netPtr->nk[1] = netPtr->M;		// by convention, first layer
matches number of inputs
//	Read in number of network outputs, N.  Print error if N > NMAX.
	if ((ierror =
read_item(NULL,&xinteger,NULL,filePtr,data_record,&tokenized)) != 0)
	{
		error_handler(ierror,"reading N, create_new_network");
		return ierror;
	}
	if (xinteger > NMAX)
	{
		fprintf(stderr,"Value of %d exceeds allowable number of network
outputs",xinteger);
		return -4;
	}
	printf("\nNetwork will have %d outputs",xinteger);
	netPtr->N = xinteger;
//	Read in number of network layers, K
	if ((ierror =
read_item(NULL,&xinteger,NULL,filePtr,data_record,&tokenized)) != 0)
	{
		error_handler(ierror,"reading K, create_new_network");
```

```
            return ierror;
      }
      if (xinteger > KMAX)
      {
            fprintf(stderr,"Value of %d exceeds allowable number of network
layers",xinteger);
            return -5;
      }
      printf("\nNetwork will have %d layers",xinteger);
      netPtr->K = xinteger;
      netPtr->nk[netPtr->K] = netPtr->N;        // by convention, last layer
matches number of outputs
//    For each hidden layer
      for (k=2; k <= (netPtr->K - 1); k++)
      {
//          Read in the number of neurons
            if ((ierror =
read_item(NULL,&xinteger,NULL,filePtr,data_record,&tokenized)) != 0)
      {
            error_handler(ierror,"reading number of neurons,
create_new_network");
            return ierror;
      }
            netPtr->nk[k] = xinteger;
            printf("\nLayer[%d] will have %d neurons.",k,xinteger);
      }
//    End-for on hidden layers

//    OK.  Now make all the network connections
#ifdef VERBOSE
      printf("\nCalling create_new_network");
#endif
      create_net_connections(netPtr,xinputs);

//    For each layer, k
      for (k=1; k <= netPtr->K; k++)
      {
//          For each neuron on the layer, n
            for (n=1; n <= netPtr->layerPtr[k]->N ; n++)
            {
//                Set the transfer function type
                  if (k == 1) xinteger = 0;
                  else xinteger = 1;
                  netPtr->layerPtr[k]->neuronPtr[n]->tf =
(transfer_function_type)xinteger;
//                Set the threshold value to zero
                  netPtr->layerPtr[k]->neuronPtr[n]->threshold = (float)0;
//                Set the bias weight to zero
                  netPtr->layerPtr[k]->neuronPtr[n]->w[0] = (float)0;
//                For each neuron input, m
                  for (m=1; m <= netPtr->layerPtr[k]->neuronPtr[n]->M; m++)
                  {
//                      Set the weight to zero
                        netPtr->layerPtr[k]->neuronPtr[n]->w[m] = (float)0;
                  }
//                End-for on inputs
            }
```

```
//          End-for on neurons
      }
//    End-for on layers

//    For each network output, n
      for (n=1; n <= netPtr->N; n++)
      {
//          Set its scale factors
            netPtr->w0[n] = (float)0;
            netPtr->w1[n] = (float)1;
      }
//    End-for on network outputs


      return 0;
}
```

Function "DEF_JRl2_control.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int DEF_JRcontrol(Network *myNetPtr, const double *mytimePtr, const double
dt, double time_to_learn, double time_to_control,
                          float ucom,float gammacom,float ux, float gammax,
double *ddePtr,double *dplaPtr, FILE *filePtr,
                          FILE *errFilePtr, FILE *modeloutPtr)
/*
            Function to implement inverse control with adaptive compensation,
"knicknamed" DEF_JR
            Equations numbers refer back to H.S. Bruner's dissertation
            Copyright 2007 Hugh S. Bruner


              Arguement List:
              --------------

                  *mytimePtr = a pointer back to the simulation time
                  ucom = the commanded perturbation velocity = commanded
speed - commanded speed at engagement, feet/second
                  gammacom = the commanded perturbation gamma
                  ux = airplane current true airspeed in feet/second
                  gammax - airplane current flight path angle, in radians
                  *ddePtr = a pointer back to the perturbation elevator
position, measured in radians.  To be added to equilibrium de
                  *dplaPtr = a pointer back to the perturbation PLA, measured
in radians.  To be added to equilibrium PLA.
                  *filePtr = a pointer back to the output device for
formatted I/O.

              H.S. Bruner
              December 21, 2006

*/

{

      static double ucomold = 0.,gcomold = 0.;  // Old values of command
variables, retained as static variables
      static double sum_of_errors2 = 0.;              // weighted sum of
errors squared.
      const double weighting_factor = .0066;          // weighting factor
      float learning_rate = (float).005;              // learning rate
      float udot_req_linear, alphadot_req_linear;     // approximat
accelerations, resulting from the linear inverse
      float udot_req, alphadot_req;                   // Commanded
accelerations, resulting from model control
      float delta_udot;                                     // incremental
udot to be calculated by neural network
      float delta_alphadot;                                 // incremental
alphadot to be calculated by neural network
      double current_alphadot;
```

```
	static TrainingPair myTrainingData={0.};  // structure for passing
training data to the neural network
	static int reset;


// RESET.  Several items are reset at the beginning each simulation


	if ((*mytimePtr - dt) < .001)       // This catches the first pass
through
	{
		printf("\nReset in DEF_JR");
		ucomold = 0.;                              // initialize old value
of ucom
		gcomold = 0.;                              // initialize old value
of gcom
		sum_of_errors2 = 0.;                 // reset the sum of errors-
squared
		reset = 1;                                 // set reset flag to
TRUE
		initialize_training_pair(&myTrainingData);
	}


//    ----------  Network training ------------------------------------------
---------------------------
//
//          Find current resulting accelerations, compare them with what was
commanded last time,
//          then store that in the training data.  Refer to equation 2.53

//                NOTE:  If you change the network configuration, the
following two lines may change

	myTrainingData.output[0] = (float)((ub*ubd[0] + wb*wbd[0])/vtrue) -
myTrainingData.input[2];
	myTrainingData.output[1] = (float)((ub*wbd[0] -
wb*ubd[0])/(vtrue*vtrue)) - myTrainingData.input[3];
	current_alphadot = (ub*wbd[0] - wb*ubd[0])/(vtrue*vtrue);

//          Output the training data to a file

	if (log_training_data == 1)
	{
		fwrite(&myTrainingData,sizeof(TrainingPair),1,filePtr);
	}


//    If you are using the network, backpropogate here....

	if (*mytimePtr >= time_to_learn)
	{
		sum_of_errors2 = sum_of_errors2*(1.-weighting_factor) +
weighting_factor*backprop(myNetPtr, &myTrainingData, learning_rate);
	}
	nnrms = sum_of_errors2;
```

```
// --------------------------------------------------------------------------------
-------------------

//    Compute linear prefilter outputs.  Note the need to use the "old"
values for commanded u and gamma

      linear_prefilter(reset, dt, ucomold, gcomold,ux, gammax,
&udot_req_linear, &alphadot_req_linear, errFilePtr);


//    The following line is where alpha limiting is applied.  Alpha dot
required passes through unless
//    limiting is required.  If limiting is required, alpha dot is
manipulated to stay within limits

      alphadot_req_linear =
(float)alpha_limiter3(alphadot_req_linear,alpha,min_alpha,max_alpha,
current_alphadot,
            (double).02,*mytimePtr,stderr);


//    ---------------------  Network useage ----------------------------------
--------------------
//    With the required linear accelerations known, update the training pair
with the current values...

      pop_training_pair(&myTrainingData, udot_req_linear,
alphadot_req_linear);         // populate the training pair


//    Do a forward pass of the network if you are learning or controlling

      if ((time_to_learn < *mytimePtr)  || (time_to_control < *mytimePtr))
      {
            forward_pass(myNetPtr, &myTrainingData);                    // do
a forward pass with the network
            input_excedance_alarm(myNetPtr,errFilePtr,mytimePtr); // write an
error message if you go out of bounds
      }

//    If you are using the newtwork for control, calculate the required
deltas for udot and alphadot

      if (*mytimePtr >= time_to_control)
      {
            delta_udot = myNetPtr->Y[1];                    // delta_udot
comes from the output of the network
            delta_alphadot = myNetPtr->Y[2];                // delta_alphadot
comes from the other output of the network
            if (control_simfailure)                               // If
simulating aircraft or system errors...
            {
                  /*    The function below is used to simulate errors in the
neural net.
                        In line below, input 3 controls udot, input 4
controls alphadot.
```

```
                        Keep these at zero unless specifically running
network failures!!
                        There is no control of these from the input data set
                                          */
                  netfailure(*mytimePtr, 0, 0, &delta_udot,
&delta_alphadot,myNetPtr->w1[1],myNetPtr->w1[2]);
            }
            udot_req = udot_req_linear - delta_udot;
      /*  Equation 2.55 for udot          */
            alphadot_req = alphadot_req_linear - delta_alphadot;        /*
Equation 2.55 for alpha dot   */
            myTrainingData.input[2] = udot_req;
      /*  This overwrites what is in the training pair */
            myTrainingData.input[3] = alphadot_req;
      /*  This overwrites what is in the training pair */
      }
      else                  // if the network is not involved in control,
the deltas are not used
      {
            udot_req = udot_req_linear;
            alphadot_req = alphadot_req_linear;
      }


//    Compute the inverse transfer function output

      inverse_xfunction(reset, udot_req, alphadot_req, ddePtr, dplaPtr);

//    retain old values for ucom and gammacom

      ucomold = ucom;
      gcomold = gammacom;

      reset = 0;                        // turn off reset flag when leaving
the program

      return 0;
}
/*          End of DEF Control function                     */
```

Function “delete_net_connections.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int delete_net_connections(Network *myNetwork)
{
	/*		Copyright 2007 Hugh S. Bruner		*/
	short int k;
	short int n;
	fprintf(stderr,"\n\n=============================================");
	fprintf(stderr,"\n\tDeleting Network connections");
	fprintf(stderr,"\n=============================================\n");
//	For each layer, k
	for (k=1; k<=myNetwork->K; k++)
	{
//		For each neuron, n, in this layer
		for (n=1; n<=myNetwork->layerPtr[k]->N; n++)
		{
//			Free the memory for this neuron
			free(myNetwork->layerPtr[k]->neuronPtr[n]);
#ifdef VERBOSE
		printf("\nDeleting Layer[%d], Neuron[%d].",k,n);
#endif
		}
//		End-for
//		Free the memory for this layer
		free(myNetwork->layerPtr[k]);
#ifdef VERBOSE
		printf("\nDeleting remainder of Layer[%d].",k);
#endif
	}
//	end-for

//	Zero-out number of layers, inputs, and outputs

	myNetwork->K = 0;
	myNetwork->M = 0;
	myNetwork->N = 0;

	return 0;
}
/*		End function delete_network
						*/
```

Function "delete_training_data.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int delete_training_data(Network *netPtr)
{
	/*		Copyright 2007 Hugh S. Bruner		*/
	TrainingPair *currentPtr;

//
//	If there is a training dataset to delete
	if (netPtr->trainingdataPtr != NULL)
	{

		printf("\n\n================================================");
		printf("\n\tDeleting Training Data");
		printf("\n================================================\n");

//		We start at the bottom of the dataset, and work our way up.
//		While bottom pointer is not NULL
		while (netPtr->trainingdataPtr->bottomPtr != NULL)
		{
//			The current address is where the bottom pointer points
			currentPtr = netPtr->trainingdataPtr->bottomPtr;
//			Assign the address at the  "previous pointer" in the
current training pair to the bottom pointer
			netPtr->trainingdataPtr->bottomPtr = currentPtr->prevPtr;
#ifdef VERBOSE
			printf("\nReleasing training pair memory");
#endif

//			Free the memory at the current address
			free(currentPtr);
		}
//		End-while

//		Free the memory associated with the training data structure
		free(netPtr->trainingdataPtr);
//		Set the training data pointer to NULL to indicate there is no
training data
		netPtr->trainingdataPtr = NULL;
	}

//	End-while on there is a dataset

	return 0;
}
/*		End of function delete_training_data
				*/
```

Function “erase_network.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"


int erase_network(Network *networkPtr)
{
      /*                Copyright 2007 Hugh S. Bruner                     */
#ifdef VERBOSE
      printf("\nEntering erase_network");
#endif
//    If the network exists
      if (networkPtr != NULL)
      {

//          Delete network connections
            delete_net_connections(networkPtr);
//          Delete training data
            delete_training_data(networkPtr);
//          Delete the network structure itself
            free(networkPtr);
//          Set network pointer to null
            networkPtr = NULL;
      }
//    End-if on existence of network

      return 0;
}

/*          End of function erase_network
            */
```

Function "error_handler.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int error_handler(int error_code, char *descr)
{
      if (error_code != 0)
      {
            fprintf(stderr,"\nError code = %d",error_code);
            fprintf(stderr,"\n%s", *descr);
            return -99;
      }
      else
      {
            return 0;
      }

}
```

Function “eval.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Function to evaluate a stability derivative value for the
            given angle of attack and thrust coefficient value
            Copyright 2007 Hugh S. Bruner
                       */
double eval(DerivPolynomial *vectorPtr, const double *alphaPtr, const double
*tcpPtr)
{

//    Declarations
      double derivative;

//    Expand the polynomial
      derivative = (double)vectorPtr->constant_term
            + (double)vectorPtr->alpha_term*(*alphaPtr)
            + (double)vectorPtr->tcp_term*(*tcpPtr)
            + (double)vectorPtr->alpha_tcp_term*(*alphaPtr)*(*tcpPtr);

//    Return the value of the stability derivative, per radian
      return derivative;
}
/*          End of evaluation function
                  */
```

Function “evaluate_neuron.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

float evaluate_neuron(Neuron *current)
{
	/*		Copyright 2007 Hugh S. Bruner			*/
	short int m;
	float sum;

//	Initialize summing junction with the bias value
	sum = current->w[0];

//	For each input, m
	for (m=1; m<=current->M;m++)
	{
//		Add the product of the input and its respective weight to the 
summing junction
		sum +=(*current->xPtr[m])*current->w[m];
	}
//	End for
//	Apply the appropriate transfer function to the total at the summing 
junction
//	Reference:  Fausett, pages 17 through 19
	switch (current->tf)
	{
	case IDENTITY:

		current->y = sum;
		current->yprime = (float)1.;
		break;

	case TANH:

		current->y = (float)tanh(sum);
		current->yprime = (float)((1. + current->y)*(1.-current->y));
		break;

	case SIGMOID:

		current->y = (float)(1./(1.+ (float)exp(sum)));
		current->yprime = current->y/(1-current->y);
		break;

	case BINARY:

		if (sum < current->threshold)
		{
			current->y = 0;
		}
		else
		{
			current->y = 1;
		}
		current->yprime = 1;
```

```
			break;
		}
		return current->y;
}
/*			End function evaluate_neuron
							*/
```

Function "execute_failure.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int execute_failure(double mytime, int manuever)
/*

            Function to execute predetermined failure scenarios, based on
time and manuever code

            Written by H.S. Bruner
            Copyright 2007 Hugh S. Bruner

*/
{
      switch(manuever)
      {

      case 1:                                         // Steck cases 1 and 2
      case 2:

            if (mytime > 5.) cma = .90*cma;
            if (mytime > 15) kthrust = .75;

            break;

      case 3:                                               // Steck case 3

            if (mytime > 0.)
            {
                  clde = .5*clde;
                  cmde = .5*cmde;
            }

            break;

      case 4:                                               // Steck case 4

            if (mytime > 20.) cma = .9*cma;
            if (mytime > 25.) kthrust = .75;

            break;

      case 10:                                        // stability failure at
t=6
      case 9:
      case 11:

            if (mytime >=6.)
            {
                  cma = cma/2.;
            }

            break;
```

```
	case 12:					// Control failure at
t=6

		if (mytime >=6.)
		{
			clde = clde/2.;
			cmde = cmde/2.;
		}

		break;

	case 13:					// Non-linear cl and cm

		clav.alpha_term = (float)-6.56;
		cmav.alpha_term = (float)2.3;

		break;

	case 14:					// Non-linear cl and
cm, and cmde failure

		clav.alpha_term = (float)-6.56;
		cmav.alpha_term = (float)2.3;
		if (mytime >=400.)
		{
			clde = clde/2.;
			cmde = cmde/2.;
		}

		break;

	case 27:					// CMA goes to zero at
t=400

		if (mytime >= 400.)
		{
			cma = 0.;
		}

		break;

	}

	return 0;

}
```

Function "extract_line.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

char * extract_line(char * sourcePtr, char * var_namePtr, char * op_namePtr,
char * value_stringPtr)
/*
            Copyright 2007 Hugh S. Bruner

            Function to find three things from a string:  A variable name, an
operator, and a value for
            the operation.

            The function looks through the source string (pointed to by
*sourcePtr) up to the point it
            finds a comma or the end of the source string.

            The function returns a pointer to the next point in the source
string after the comma, or
            a NULL pointer if the source string ended without a comma.

            Arguments:
            ==========

            *sourcePtr              pointer to the source string

            *var_namePtr            pointer to the string in the calling
program that will
                                          receive the variable name

            *op_namePtr             pointer to the string in the calling
program that will receive
                                          the operator

            *value_stringPtr  pointer to the string in the calling program
that will receive the string
                                          value found after the operator.
The calling program is responsible for
                                          interpretation of this string.


                                                                              */
{
      char * end_Ptr;                           // Pointer to the end of
where this call to the function processed text
      char * operatorPtr;                       // Will point to the place in
the source string where the operator is found
      unsigned int len_var_name;          // Will contain the length of the
variable name string
      unsigned int len_value_string;      // Will contain the length of the
value string
      char scratch[100];                        // scratch string

      end_Ptr = NULL;                           // assume the end_Ptr will
say no more processing can be done on this string.
```

```
                                                                    // This will change if
something contradicts this assumption


/*          If the source string has some non-zero length, process the string
            */
      if (strlen(sourcePtr) > 0)
      {
            end_Ptr = strchr(sourcePtr,',');          // if a comma is in the
source string, that will be our end point
/*
            Next look for operators, commas or colons, and capture their
location in *operatorPtr.
            *operatorPtr will be NULL if no operators are found

                                                      */
            if ((int)(operatorPtr = strchr(sourcePtr,'='))==NULL)
            {
                  operatorPtr = strchr(sourcePtr,':');
            }
/*
            If operators were found: i.e. operatorPtr is not NULL

                                                      */
            if (operatorPtr != NULL)

            {
                  strncpy(op_namePtr,operatorPtr,1);                   //
Copy the operator string to *op_namePtr
                  *(op_namePtr + 1) = '\0';
      // Null terminate the string
//                printf("\nOperator = %s",op_namePtr);
                  len_var_name = (operatorPtr - sourcePtr);       //
Calculate the length of the variable string
                  strncpy(var_namePtr,sourcePtr,len_var_name);    // Copy the
variable name from the source string
                  *(var_namePtr + len_var_name) = '\0';                //
Null terminate the string
//                printf("\nVar_name = %s",var_namePtr);
/*
                  Calculate the length of the value string.  If no commas are
found (end_Ptr == NULL),
                  the length is determined as the difference between the
length of the string and the
                  variable name string, less one for the place taken by the
operator and one more for
                  the linefeed at the end of the record.  If commas are
found,
                  the length is the difference between the end_Ptr and the
operator pointer, less one for
                  the space taken by the comma.

                                                      */
                  if (end_Ptr == NULL)
                  {
                        len_value_string = strlen(sourcePtr) - len_var_name -
2;
```

```
                  }
                  else
                  {
                        len_value_string = end_Ptr - operatorPtr - 1;
                  }
//                printf("\nlen_value_string = %d",len_value_string);
                  strncpy(value_stringPtr,(operatorPtr +
1),len_value_string);   // Copy the value string from the

                                    // source string, starting at the

                                    // the place pointed to by operatorPtr

                                    // plus one.
                  *(value_stringPtr + len_value_string) = '\0';
            // Null terminate the string


//                Remove any extraneous white space from the strings
                  no_white_space(var_namePtr,scratch);
                  strcpy(var_namePtr,scratch);
                  no_white_space(op_namePtr,scratch);
                  strcpy(op_namePtr,scratch);
                  no_white_space(value_stringPtr,scratch);
                  strcpy(value_stringPtr,scratch);

            }
            else        // you found no operators within the string.  Set
things as "unspecified"
            {
                  strcpy(var_namePtr,"unspecified");
                  strcpy(op_namePtr,"u");
                  strcpy(value_stringPtr,"unspecified");
            }
            if (end_Ptr != NULL) ++end_Ptr;  // Add one to the value of
end_Ptr to put the pointer beyond the comma

      }           // end if on strlen > 0

    return end_Ptr;                                 // Return the value of
end_Ptr to the calling program.
}
/*                      End of function read_line
                                    */
```

Function "fast.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Function to compute high frequency part of simulation
      */
int fast(double *sim_timePtr, double dt)
{
/*
      This function computes the dynamics of motion of the aircraft.
      This is intended to operatate at fairly "high" frequency, > 20 Hz.
      Equations referenced may be found in Appendix B of H.S. Bruner's
      PhD dissertation.

      Copyright 2007 Hugh S. Bruner

                              */
//    Declarations
      int iterations;                               // iteration counter
      double ps, rs;                                // stability axis
rotation rates
      double c2v, b2v;                        // used in computing helix
angles
      double alphadot;                        // rate of change of angle of
attack
      double phat,qhat,rhat,alphadothat;  // helix angles
      double cl, cm, cd, cy, cn, cr;          // aero coefficients
      double thrust;                                // engine thrust
      double fxb, fyb, fzb;                   // body axis forces
      double pm, rms, yms;                    // stability axis moments
      double rm, ym;                                // body axis moments

      *sim_timePtr += dt;                           // increment simulation
time by dt

      if (real_time) simulation_timer(20);//    Simulation timer paces
execution of fast

//    Predict state variables

      ub = predict(ubd,&ub0,&dt);
      vb = predict(vbd,&vb0,&dt);
      wb = predict(wbd,&wb0,&dt);
      pb = predict(pbd,&pb0,&dt);
      qb = predict(qbd,&qb0,&dt);
      rb = predict(rbd,&rb0,&dt);
      phi = predict(phid,&phi0,&dt);
      theta = predict(thetad,&theta0,&dt);
      psi = predict(psid,&psi0,&dt);
      h = predict(hd,&h0,&dt);
      thrust = thrust_model(pla,vtrue,rho);


//    For four iterations, refine the prediction
```

```
      for (iterations = 0; iterations <4; iterations++)
      {
//          Compute stability axis velocities and rates from body axes

            alpha = atan(wb/ub);                            /* Equation B1
*/
            beta = atan(vb/ub);                                   /* Equation
B2  */
            cosa = cos(alpha);
            sina = sin(alpha);
            sinth = sin(theta);
            costh = cos(theta);
            tanth = tan(theta);
            sinphi = sin(phi);
            cosphi = cos(phi);
            vtrue = sqrt(ub*ub + vb*vb + wb*wb);      /*  Equation B3  */
            c2v = cref/(2.*vtrue);
            b2v = bref/(2.*vtrue);

            ps = pb * cosa + rb * sina;                     /*  Equation B4
*/
            rs = -pb * sina + rb * cosa;              /*  Equation B6  */

            phat = ps * b2v;                                /*  Equation B9
*/
            qhat = qb*c2v;                                        /*
Equation B10 */
            rhat = rs * b2v;                                /*  Equation B11
*/

            alphadot = wbd[0]/ub - wb*ubd[0]/(ub*ub); /*  Equation B7  */
            alphadothat = alphadot*c2v;                           /*
Equation B8  */

//          Compute stability axis force and moment coefficients

            cl = clo + cla * alpha + clde * de + cldetab * detab + cladot *
alphadothat + clq * qhat;     /*  Equation B12 */
            cm = cmo + cma * alpha + cmde * de + cmdetab * detab + cmadot *
alphadothat + cmq * qhat;     /*  Equation B13 */
            cd = cdo + cdk*cl*cl;                                       /*
Equation B14 */
            thrust = kthrust*thrust_model(pla,vtrue,rho);   /*  Equation B15
*/

            cy = cyo + cyb*beta + cydr*dr + cyda*da + cyp*phat + cyr*rhat;
      /*  Equation B16 */
            cn = cno + cnb*beta + cndr*dr + cnda*da + cnp*phat + cnr*rhat;
      /*  Equation B17 */
            cr = cro + crb*beta + crdr*dr + crda*da + crp*phat + crr*rhat;
      /*  Equation B18 */

//          Compute body axis forces

            fxb = q1sw * (cl * sina - cd * cosa) - weight * sinth + thrust *
cosa;                   /*  Equation B19 */
```

```
            fyb = q1sw * cy + weight * sinphi * costh;
                                           /*  Equation B20 */
            fzb = q1sw * (-cl * cosa - cd * sina) + weight * costh * cosphi +
thrust * sina;    /*  Equation B21 */

//          Compute stability axis moments

            pm = q1swcw*cm;                                         /*
Equation B22 */
            rms = q1swbw*cr;                                  /*  Equation B23
*/
            yms = q1swcw*cn;                                  /*  Equation B24
*/

//          Transform moments in stability axis into body axis

            rm = rms * cosa - yms * sina;             /*  Equation B25 */
            ym = rms * sina + yms * cosa;             /*  Equation B27 */

//          Compute linear and rotational accelerations

      ubd[0] = fxb / mass + vb * rb - wb * qb;
      /*  Equation B28 */
      vbd[0] = fyb / mass - ub * rb + wb * pb;
      /*  Equation B29 */
      wbd[0] = fzb / mass + ub * qb - vb * pb;
      /*  Equation B30 */
      qbd[0] = pm / iyyb - (ixxb - izzb) * pb * rb / iyyb - ixzb * (pb * pb -
rb * rb) / iyyb;  /*  Equation B32 */
      pbd[0] = (izzb * rm + ixzb * ym) / idivisor;
      /*  Equation B31, abbreviated */
      rbd[0] = (ixzb * rm + ixxb * ym) / idivisor;
      /*  Equation B32, abbreviated */

      phid[0] = pb + qb * sinphi * tanth + rb * cosphi * tanth;
      /*  Equation B34 */
      thetad[0] = qb * cosphi - rb * sinphi;
      /*  Equation B35 */
      psid[0] = (qb * sinphi + rb * cosphi) / costh;
      /*  Equation B36 */
      hd[0] = ub * sinth - vb * costh * sinphi - wb * costh * cosphi;   /*
Equation B37 */

//          Correct estimates

            ub = correct(ubd,&ub0,&dt);
            vb = correct(vbd,&vb0,&dt);
            wb = correct(wbd,&wb0,&dt);
            pb = correct(pbd,&pb0,&dt);
            qb = correct(qbd,&qb0,&dt);
            rb = correct(rbd,&rb0,&dt);
            phi = correct(phid,&phi0,&dt);
            theta = correct(thetad,&theta0,&dt);
            psi = correct(psid,&psi0,&dt);
            h = correct(hd,&h0,&dt);

//    End of iteration
```

```
      }
//    Shuffle down history arrays

      push_down();

//    Establish new "old" values

      ub0 = ub;
    vb0 = vb;
    wb0 = wb;
    pb0 = pb;
    qb0 = qb;
    rb0 = rb;
    theta0 = theta;
    phi0 = phi;
    psi0 = psi;
    h0 = h;

      return 0;

}
/*          End of fast
                        */
```

Function "fetrim.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*                 Function to calculate Fe = 0 trim conditon
                         */
/*

                   Copyright 2007 Hugh S. Bruner

                                          */
int fetrim(double *alpha1Ptr, double *de1Ptr, double *detab1Ptr, double
*pla1Ptr)
{
//    Declarations
      short int iterations;
//    short int i,j;
      float eps=(float)1e-7;
      double cl1, cd1;
      double a[3][3]={0.}, x[3]={0.}, y[3]={0.};
      double thrust, de1, alpha1, detab1, change,epsa,eps0;

//    Calculate CL and CD
      cl1 = weight/(q1sw);
      cd1 = cdo + cdk*(cl1-cli)*(cl1-cli);

//    Thrust = drag
      tcp = cd1;
      thrust = tcp*q1sw;

//    Make some assumptions to start the iteration

      de1 = 0;
      change = 1;
      alpha = 0.;

      clo = eval(&clov,&alpha,&tcp);                    // evaluate parameter
for assumed alpha and tcp
      cla = eval(&clav,&alpha,&tcp);
      clde = eval(&cldev,&alpha,&tcp);
      cldetab = eval(&cldetabv,&alpha,&tcp);

      cmo = eval(&cmov,&alpha,&tcp);
      cma = eval(&cmav,&alpha,&tcp);
      cmde = eval(&cmdev,&alpha,&tcp);
      cmdetab = eval(&cmdetabv,&alpha,&tcp);

#ifdef DEBUG
            printf("\nIn function fetrim.");

            printf("\nTrim cl1 = %f",cl1);
            printf("\nTrim cd1 = %f",cd1);

#endif
```

```
//    Iterate until the change in each iteration is within tolerance
      iterations = 0;
      while (change > eps)
      {
            iterations ++;

//          Compute things that are functions of de and alpha
            fbw = bw0 + dbwdde*de1;
            fds = ds0 + ddsdde*de1;
            ge = gev[0] + de1*(gev[1] + de1*(gev[2] + de1*gev[3]));

            clo = eval(&clov,&alpha1,&tcp);           // update parameters
with new alpha and tcp
            cla = eval(&clav,&alpha1,&tcp);
            clde = eval(&cldev,&alpha1,&tcp);
            cldetab = eval(&cldetabv,&alpha1,&tcp);

            cmo = eval(&cmov,&alpha1,&tcp);
            cma = eval(&cmav,&alpha1,&tcp);
            cmde = eval(&cmdev,&alpha1,&tcp);
            cmdetab = eval(&cmdetabv,&alpha1,&tcp);

            eps0 = epsv.constant_term + epsv.tcp_term*tcp;
            epsa = epsv.alpha_term + epsv.alpha_tcp_term*tcp;
//          tcp = cd1/cos(alpha1);
//          thrust = tcp*q1sw;

//          Load array that represent the matrix equation for trim
            a[0][0] = cla;
            a[0][1] = clde;
            a[0][2] = cldetab;
            a[1][0] = cma;
            a[1][1] = cmde;
            a[1][2]     = cmdetab;
            a[2][0] = chea*(1.-epsa);
            a[2][1] = chede;
            a[2][2] = chedetab;

            y[0] = cl1 - clo;
            y[1] = -cmo - dx*cl1;
            y[2] = -cheo + chea*eps0 - (fbw + fds)/(q1*se*ce*ge);

            solve3by3(a,x,y);

            change = fabs(x[1]-de1);            // change is defined by the
absolute
                                                            // value of
change in elevator during this iteration
            alpha1 = x[0];
            de1 = x[1];
            detab1 = x[2];
      }

      de = de1;
      detab = detab1;
      alpha = alpha1;
      pla = (double)100*thrust/thrust_model((double)100., vtrue, rho);
```

```
	ub = vtrue*cos(alpha);
	wb = vtrue*sin(alpha);
	vb = 0;
	theta = alpha1;

	*alpha1Ptr = alpha1;
	*de1Ptr = de1;
	*detab1Ptr = detab;
	*pla1Ptr = pla;

//	That is all!
	return 0;
}
/*		End of trim function
							*/
```

Function "force_roskam.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Function to "jam" in Roskam test case data
            Copyright 2007 Hugh S. Bruner

            Reference:  Lear Jet sample case from Roskam, Appendix A, p. 617
                  */
int force_roskam(void)
{
//    Declarations
      double rho;                   // density of the air

#ifdef DEBUG
            printf("\nfunction force_roskam.");
#endif
//    Mass Properties

      weight = 13000.;
      xcg = .32;
      ixxb = 28000.;
      iyyb = 18800.;
      izzb = 47000.;
      ixzb = 13000.;

//    Geometry

      sref = 230.;
      cref = 7.;
      bref = 34.;
      cgref = .32;
      se = 13.8;
      ce = 1.;

//    Mechanical (made up.  Not provided in Roskam example)

      bw0 = 5;
      ds0 = 5;
      gev[0] = 1.;

//    Flight Condition

      h = 40000.;
      rho = std_density(h);
      vkeas = 400.*sqrt(rho/.002378);
      xcg = .32;
      dx = (xcg - cgref)*cref;

//    Derivatives

      clov.constant_term = (float).135;
      clav.constant_term = (float).1019*(float)rtod;
      cldev.constant_term = (float).00803*(float)rtod;
      cldetabv.constant_term = (float).001*(float)rtod;
```

```
      clqv.constant_term = (float)4.7;
      cladotv.constant_term = (float)2.2;

      cmov.constant_term = (float).02;
      cmav.constant_term = (float)-.0112*(float)rtod;
      cmdev.constant_term = (float)-.0216*(float)rtod;
      cmdetabv.constant_term = (float)-.002*(float)rtod;
      cmqv.constant_term = (float)-15.5;
      cmadotv.constant_term = (float)-6.7;

      cdo = (float).0217;
      cdk = (float).0704;
      cli = (float)0;

      epsv.constant_term = (float)1.5/(float)rtod;
      epsv.alpha_term = (float).25;


      crbv.constant_term = (float)-.00192*(float)rtod;
      crpv.constant_term = (float)-.45;
      crrv.constant_term = (float).16;
      crdrv.constant_term = (float).00033*(float)rtod;
      cnbv.constant_term = (float).0022*(float)rtod;
      cnpv.constant_term = (float)-.008;
      cnrv.constant_term = (float)-.2;
      cybv.constant_term = (float)-.01274*(float)rtod;
      cypv.constant_term = (float)0.;
      cyrv.constant_term = (float).4;
      cydrv.constant_term = (float)0;

//    Hinge moments

      cheo = (float)0.;
      chede = (float)-.0066*(float)rtod;
      chedetab = chede;
      chea = (float)-.001*(float)rtod;

      return 0;


}
/*          End of function to force in Roskam data
                  */
```

Function "format_filles.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Function to create formatted output files
            Copyright 2007 Hugh S. Bruner
                                    */
int format_files(FILE *ffilePtr, int isims, char *descriptor)
{
      BinaryRecord time_slice;            // To catch data at each time slice
      FILE *csvPtr;                             // File pointer for the csv
output file
      FILE *bfilePtr;                           // File pointer for the
unformatted output
      char csvfilename[20];               // File name for the csv file

//    Use data from scratch file to create a neatly formatted output file and
//    a similar csv formatted file for excel.
//    First synthesize a file name

      sprintf(csvfilename,"output%02d.csv",isims);

//    Open the binary file  again for reading

      if((bfilePtr = fopen(SCRATCH,"rb")) == NULL)
      {
            printf("\nCould not open binary file for read-only");  // File
would not open
            return -2;
      }
//    Open the csv target file for writing

      else if((csvPtr = fopen(csvfilename,"w")) == NULL)
      {
            printf("\nCould not open .csv file for writing.");  // File would
not open
            return -3;
      }
      else                                // File opened correctly
      {

//          Write banner fields to the files.  It looks messy, but it is just
a long format descriptor...
//          For the text file...
            fprintf(ffilePtr,"\n%s\n",descriptor);

            fprintf(ffilePtr,"\n%6s %6s %7s %6s %6s %7s %7s %7s %7s %6s %6s
%6s %6s %7s %6s %6s %6s %7s %8s %8s %8s %8s %8s %8s",

      "Time","Vkeas","h","Alpha","Beta","Phi","Theta","Psi","Gamma","de","det
ab","dr","da","pla",
                  "dde,ap","ddr, ap","dda, ap","dpla,ap",
                  "Xearth","Yearth", "RMS err", "ucom", "gammacom", "vtrue");
            fprintf(ffilePtr,"\n%6s %6s %7s %6s %6s %7s %7s %7s %7s %6s %6s
%6s %6s %7s %6s %6s %6s %7s %8s %8s %8s %8s %8s %8s",
```

```
      "(sec)","(kts)","(ft)","(deg)","(deg)","(deg)","(deg)","(deg)","(deg)",
"(deg)","(deg)","(deg)","(deg)",
                  "(pct)","(deg)","(deg)","(deg)","(pct)","(ft)","(ft)", "(-
)", "(fps)", "(deg)", "(fps)");

//          For the csv file...
            fprintf(csvPtr,"%s",descriptor);
            fprintf(csvPtr,"U1 = %.1f fps, hp = %.0f ft",Utrim,htrim);

            fprintf(csvPtr,"\n%6s, %6s, %7s, %6s, %6s, %7s, %7s, %7s, %7s,
%6s, %6s, %6s, %6s, %7s, %6s, %6s, %6s, %7s, %8s, %8s, %8s, %8s, %8s, %8s",

      "Time","Vkeas","h","Alpha","Beta","Phi","Theta","Psi","Gamma","de","det
ab","dr","da", "pla",
                  "dde_ap","ddr_ap","dda_ap","dpla_ap",
                  "Xearth","Yearth", "RMS err", "ucom", "gammacom", "vtrre");
            fprintf(csvPtr,"\n%6s, %6s, %7s, %6s, %6s, %7s, %7s, %7s, %7s,
%6s, %6s, %6s, %6s, %7s, %6s, %6s, %6s, %7s,%8s, %8s, %8s, %8s, %8s, %8s",

      "(sec)","(kts)","(ft)","(deg)","(deg)","(deg)","(deg)","(deg)","(deg)",
"(deg)","(deg)","(deg)","(deg)",
                  "(pct)","(deg)","(deg)","(deg)","(pct)","(ft)","(ft)", "(-
)", "(fps)", "(deg)","(fps)");

//          OK.  We read a record from the binary file, then write it
formatted to the formatted file
//          until we run out of records

            fread(&time_slice,sizeof(BinaryRecord),1,bfilePtr);
            while (!feof(bfilePtr))
            {
//                      First write a formatted record to the text file
                  fprintf(ffilePtr,"\n");
                  fprintf(ffilePtr,"%6.2f ",time_slice.time);
                  fprintf(ffilePtr,"%6.1f ",time_slice.vkeas);
                  fprintf(ffilePtr,"%7.1f ",time_slice.h);
                  fprintf(ffilePtr,"%6.2f ",time_slice.alpha*rtod);
                  fprintf(ffilePtr,"%6.2f ",time_slice.beta*rtod);
                  fprintf(ffilePtr,"%7.2f ",time_slice.phi*rtod);
                  fprintf(ffilePtr,"%7.2f ",time_slice.theta*rtod);
                  fprintf(ffilePtr,"%7.2f ",time_slice.psi*rtod);
                  fprintf(ffilePtr,"%7.2f ",time_slice.gamma*rtod);
                  fprintf(ffilePtr,"%6.2f ",time_slice.de*rtod);
                  fprintf(ffilePtr,"%6.2f ",time_slice.detab*rtod);
                  fprintf(ffilePtr,"%6.2f ",time_slice.dr*rtod);
                  fprintf(ffilePtr,"%6.2f ",time_slice.da*rtod);
                  fprintf(ffilePtr,"%7.2f ",time_slice.pla);
                  fprintf(ffilePtr,"%6.2f ",time_slice.dde*rtod);
                  fprintf(ffilePtr,"%6.2f ",time_slice.ddr*rtod);
                  fprintf(ffilePtr,"%6.2f ",time_slice.dda*rtod);
                  fprintf(ffilePtr,"%7.2f ",time_slice.dpla);
                  fprintf(ffilePtr,"%8.1f ",time_slice.xearth);
                  fprintf(ffilePtr,"%8.1f ",time_slice.yearth);
                  fprintf(ffilePtr,"%8.6f ",time_slice.rmserror);
                  fprintf(ffilePtr,"%8.3f ",time_slice.ucom);
                  fprintf(ffilePtr,"%8.4f ",time_slice.gammacom);
```

```
                    fprintf(ffilePtr,"%8.3f ",time_slice.vtrue);

//                        Then write a formatted record to the csv file
                    fprintf(csvPtr,"\n");
                    fprintf(csvPtr,"%6.2f, ",time_slice.time);
                    fprintf(csvPtr,"%6.1f, ",time_slice.vkeas);
                    fprintf(csvPtr,"%7.1f, ",time_slice.h);
                    fprintf(csvPtr,"%6.2f, ",time_slice.alpha*rtod);
                    fprintf(csvPtr,"%6.2f, ",time_slice.beta*rtod);
                    fprintf(csvPtr,"%7.2f, ",time_slice.phi*rtod);
                    fprintf(csvPtr,"%7.2f, ",time_slice.theta*rtod);
                    fprintf(csvPtr,"%7.2f, ",time_slice.psi*rtod);
                    fprintf(csvPtr,"%7.2f, ",time_slice.gamma*rtod);
                    fprintf(csvPtr,"%6.2f, ",time_slice.de*rtod);
                    fprintf(csvPtr,"%6.2f, ",time_slice.detab*rtod);
                    fprintf(csvPtr,"%6.2f, ",time_slice.dr*rtod);
                    fprintf(csvPtr,"%6.2f, ",time_slice.da*rtod);
                    fprintf(csvPtr,"%7.2f, ",time_slice.pla);
                    fprintf(csvPtr,"%6.2f, ",time_slice.dde*rtod);
                    fprintf(csvPtr,"%6.2f, ",time_slice.ddr*rtod);
                    fprintf(csvPtr,"%6.2f, ",time_slice.dda*rtod);
                    fprintf(csvPtr,"%7.2f, ",time_slice.dpla);
                    fprintf(csvPtr,"%8.1f, ",time_slice.xearth);
                    fprintf(csvPtr,"%8.1f, ",time_slice.yearth);
                    fprintf(csvPtr,"%8.6f, ",time_slice.rmserror);
                    fprintf(csvPtr,"%8.3f, ",time_slice.ucom);
                    fprintf(csvPtr,"%8.4f, ",time_slice.gammacom);
                    fprintf(csvPtr,"%8.3f, ",time_slice.vtrue);

//                        Try to read another binary record
                    fread(&time_slice,sizeof(BinaryRecord),1,bfilePtr);

//                  End of while loop on reading records
              }

//                  Close output files
              fclose(bfilePtr);
              fclose(csvPtr);
              return 0;
       }                                       //    End of if-then-else
clause on file opens
}
/*                  End of function to output formatted files
              */
```

Function "format_training_file.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int format_training_data(FILE *outfilePtr,int pfreq)
{
/*          Function to read the binary scratch file containing the selected
training data, and write it out as
                  a formatted file.
            Copyright 2007 Hugh S. Bruner
*/

      FILE *infilePtr;
      short int i;
      short jfreq;                  // print counter
      infilePtr = fopen(SCRATCH2,"rb");
      TrainingPair binout;

//          printf("\nInside function format_training_data.");

//          Print banner on file

            fprintf(outfilePtr,"%c Print frequency is ever %d
frames.",'%',pfreq);
            fprintf(outfilePtr,"\n%c",'%');

            fprintf(outfilePtr,"%s,","Utrim(t)");
            fprintf(outfilePtr,"%s,","htrim(t)");
            fprintf(outfilePtr,"%s,","vpdot req(t)");
            fprintf(outfilePtr,"%s,","alphadot req(t)");
            fprintf(outfilePtr,"%s,","vp(t)");
            fprintf(outfilePtr,"%s,","alpha(t)");
            fprintf(outfilePtr,"%s,","q(t)");
            fprintf(outfilePtr,"%s,","theta(t)");
            fprintf(outfilePtr,"%s,","hp(t)");
            fprintf(outfilePtr,"%s,","dde(t)");
            fprintf(outfilePtr,"%s,","dpla(t)");

            fprintf(outfilePtr,"%s,","vp(t-1)");
            fprintf(outfilePtr,"%s,","alpha(t-1)");
            fprintf(outfilePtr,"%s,","q(t-1)");
            fprintf(outfilePtr,"%s,","theta(t-1)");
            fprintf(outfilePtr,"%s,","hp(t-1)");
            fprintf(outfilePtr,"%s,","dde(t-1)");
            fprintf(outfilePtr,"%s,","dpla(t-1)");

            fprintf(outfilePtr,"%s,","vp(t-2)");
            fprintf(outfilePtr,"%s,","alpha(t-2)");
            fprintf(outfilePtr,"%s,","q(t-2)");
            fprintf(outfilePtr,"%s,","theta(t-2)");
            fprintf(outfilePtr,"%s,","hp(t-2)");
            fprintf(outfilePtr,"%s,","dde(t-2)");
            fprintf(outfilePtr,"%s,","dpla(t-2)");

            fprintf(outfilePtr,"%s,","delta vpdot req(t-1)");
```

```
            fprintf(outfilePtr,"%s,","delta alphadot req(t-1)");

            fprintf(outfilePtr,"%s,","delta vpdot req(t-2)");
            fprintf(outfilePtr,"%s,","delta alphadot req(t-2)");

            fprintf(outfilePtr,"%s,","delta vpdot req(t-3)");
            fprintf(outfilePtr,"%s,","delta alphadot req(t-3)");

            fprintf(outfilePtr,"%s,","delta vpdot req(t)");
            fprintf(outfilePtr,"%s,","delta alphadot req(t)");

            fprintf(outfilePtr,"\n");

//          OK.  We read a record from the binary file, then write it
formatted to the formatted file
//          until we run out of records

            jfreq = 0;

            fread(&binout,sizeof(TrainingPair),1,infilePtr);
            while (!feof(infilePtr))
            {
//                Count the record read, if you have reached the print
frequency, then print it

                  if (++jfreq >= pfreq)
                  {

//                      First write a formatted record to the text file

                        for (i=0; i<=30; i++)
                        {
                              fprintf(outfilePtr,"%.6g,",binout.input[i]);
                        }
                        for (i=0; i<=1;   i++)
                        {
                              fprintf(outfilePtr,"%.6g,",binout.output[i]);
                        }
                        fprintf(outfilePtr,"\n");

                        jfreq = 0;        // reset print counter to 0

                  }

//                Read the next record
                  fread(&binout,sizeof(TrainingPair),1,infilePtr);
            }

            fclose(infilePtr);

            return 0;
}
/*                End of function format_training_data
      */
```

Function "forward_pass.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int forward_pass(Network *currentNet, TrainingPair *currentpairPtr)
{
	/*		Copyright 2007 Hugh S. Bruner		*/
	short int k;
	short int m;
	short int n;
	float result;
//
//		Move inputs on training pair to the input neurons of the network

	for (m=0; m<currentNet->M;m++)
	{
		*(currentNet->xPtr[m+1]) = currentpairPtr->input[m];  //assign
value of training pair input to net input
	}

//	For each layer, k
	for (k=1; k <= currentNet->K; k++)
	{
//		For each neuron in the layer, n
		for (n=1; n <= currentNet->nk[k]; n++)
		{
//			Evaluate the neuron output
			result = evaluate_neuron(currentNet->layerPtr[k]-
>neuronPtr[n]);
#ifdef SHOWNEURONS
		printf("\n\n\nLayer[%d], Neuron[%d] value is %f",k,n,result);
		dump_neuron(currentNet->layerPtr[k]->neuronPtr[n]);
#endif
		}
//		End-for
	}
//	End for

//	For each output neuron, n
	for (n=1; n <= currentNet->N; n++)
	{
//		Apply scale factors to get Network output in user units
		currentNet->Y[n] = *currentNet->yPtr[n]*currentNet->w1[n] +
currentNet->w0[n];
	}
//	End for on output neurons

	return 0;
}
/*		End of function forward_pass
							*/
```

Function "get_train_options.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int get_train_options(char *sourcePtr, double *start_learn, double
*start_control)
{
	/*		Copyright 2007 Hugh S. Bruner		*/
	char delims[]=" ,;|\t\n";
	int tokenized = FALSE;
	char *tokenPtr;
	char **PtrPtr = NULL;
	short count = 1;

//	Assumed values

	*start_learn = 0.;
	*start_control = 0;
	tokenPtr = sourcePtr;

	while (tokenPtr != NULL)
	{

		if (!tokenized)
		{
			tokenPtr = strtok(sourcePtr,delims);
			tokenized = TRUE;
		}
		else
		{
			tokenPtr = strtok(NULL,delims);
		}
		if (tokenPtr != NULL)
		{
			if (count == 1) *start_learn = strtod(tokenPtr,PtrPtr);
				else if (count == 2) *start_control =
strtod(tokenPtr,PtrPtr);
			count++;
		}
	}

	return 0;


}
```

Function "globals.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"

/*          Globally used enumerations
            Copyright 2007 Hugh S. Bruner
                                                             */


//    Global declarations

FILE *testoutPtr;

short int real_time = 0;                                     // Controls
whether running realtime or not
short int log_training_data = 1;                        // Controls whether or
not to log training data
int control_printfreq=1;                                           // Controls
frequency of printing output during sim
int control_ptrainfreq=100;                                        // Controls
frequency of printing training data during sim
int control_simfailure=0;                                          // Controls
whether or not to execute failures during sim
double nnrms;                                           // neural network rms
error.


//  Global variables


//          Simulation:

DerivPolynomial clov={0.}, clav={0.}, cldev={0.};
DerivPolynomial cldetabv={0.}, cladotv={0.}, clqv={0.};
DerivPolynomial cmov={0.}, cmav={0.}, cmdev={0.};
DerivPolynomial cmdetabv={0.}, cmadotv={0.}, cmqv={0.};
DerivPolynomial cyov={0.}, cybv={0.}, cydrv={0.};
DerivPolynomial cydav={0.}, cypv={0.}, cyrv={0.};
DerivPolynomial cnov={0.}, cnbv={0.}, cndrv={0.};
DerivPolynomial cndav={0.}, cnpv={0.}, cnrv={0.};
DerivPolynomial crov={0.}, crbv={0.}, crdrv={0.};
DerivPolynomial crdav={0.}, crpv={0.}, crrv={0.};
DerivPolynomial epsv={0.};
double cheo, chea, chede, chedetab;                          // elevator hinge
moment characteristics
double max_alpha, min_alpha;

double weight, xcg, dx, mass, ixxb,
      iyyb, izzb, ixzb, idivisor;                                  // Mass
properties
double sref, cref, bref, cgref;                                    //
Reference geometry
double se, ce;                                                           //
elevator area and elevator chord
double fds, fbw, ds0, ddsdde, bw0, dbwdde;                   // bobweight and
downspring values
```

```
double ge, gev[4];                                                          //
Elevator gearing curve

double clo, cla, clde, cldetab, cladot, clq;               // lift
coefficients and derivatives
double cmo, cma, cmde, cmdetab, cmadot, cmq;               // pitching
moment coefficients and derivatives
double cdo, cdk, cli;                                            // drag
polar
double cyo, cyb, cydr, cyda, cyp, cyr;                     // side force
coefficients and derivatives
double cno, cnb, cndr, cnda, cnp, cnr;                     // yawing moment
coefficients and derivatives
double cro, crb, crdr, crda, crp, crr;                     // rolling moment
coefficients and derivatives

double ub, vb, wb;                  // Body axis linear velocities
double us, vs, ws;                  // Stability axis linear velocities
double pb, qb, rb;                  // Body exis rotational velocities
double ps, qs, rs;                  // Stability axis rotational velocities
double psi, theta, phi;       // Euler angles
double ub0, ubd[5];                 // U body-axis time-derivative and
history vector
double vb0, vbd[5];                 // V body-axis time-derivative and
history vector
double wb0, wbd[5];                 // W body-axis time-derivative and
history vector
double pb0, pbd[5];                 // P body-axis time-derivative and
history vector
double qb0, qbd[5];                 // Q body-axis time-derivative and
history vector
double rb0, rbd[5];                 // R body-axis time-derivative and
history vector
double phi0, phid[5];         // Phi time-derivative and history vector
double theta0, thetad[5];     // Theta time-derivative and history vector
double psi0, psid[5];         // Psi time-derivative and history vector
double h0, hd[5];             // Altitude time-derivative and history vector
double xearth,yearth;         // Earth coordinates
double mach, vkeas, h, vtrue, rho, q1, U1;
double htrim, Utrim;          // trim values for altitude and speed
double q1sw, q1swbw, q1swcw;

double alpha, beta;
double sina, cosa, sinth, costh, tanth, sinphi, cosphi;     // trig quatities
of alpha and theta
double pla, tcp;              // power lever angle, thrust coefficent
double kthrust = 1.;          // thrust multiplier
double de,detab,da,dr;        // Control deflectins: elevator, elevator tab,
aileron, and rudder
unsigned int start_time, stop_time, elapsed_time;

double g=32.2;                                // Gravitational constant
double rtod= 57.2;                            // Radians to degrees
conversion
```

Function "initial.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Function to initialize simulation variables
            Copyright 2007 Hugh S. Bruner
      */
int initialize_sim(void)
{
//    Declarations
      int i;                                    // Loop counter

#ifdef DEBUG
            printf("\nfunction initalize sim.");
#endif
//    Initialize constants
      pb0 = 0;
      qb0 = 0;
      rb0 = 0;
      ub0 = ub;
      wb0 = wb;
      vb0 = vb;
      phi0 = 0;
      theta0 = theta;
      psi0 = 0;
      h0 = h;


//    Initialize arrays
      for (i=0; i<5; i++)
      {
            ubd[i] = 0;
            vbd[i] = 0;
            wbd[i]= 0;
            pbd[i] = 0;
            qbd[i] = 0;
            rbd[i] = 0;
            phid[i] = 0;
            thetad[i] = 0;
            psid[i] = 0;
            hd[i] = 0;
      }
//    Return to calling program

      return 0;
}
/*          End of initialization function
                        */
```

Function "initialize_by_code.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int initialize_by_code(Network *netPtr, float xinputs[])
/*          Function to initialize neural network programmatically
            Copyright 2007 Hugh S. Bruner
            Created 11:29:58                        01/28/2007

            Created from g6070128_2.csv  RMS = .002703
      */
{

            netPtr->M = 31;
            netPtr->N = 2;
            netPtr->K = 3;
            netPtr->nk[1] = netPtr->M;                      // by convention,
first layer matches number of inputs
            netPtr->nk[2] = 31;
            netPtr->nk[netPtr->K] = netPtr->N;        // by convention, last
layer matches number of outputs

//          OK.  Now make all the network connections
            create_net_connections(netPtr,xinputs);

            netPtr->layerPtr[1]->neuronPtr[1]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[1]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[1]->w[0] = (float)-2.073;
            netPtr->layerPtr[1]->neuronPtr[1]->w[1] = (float)0.01059;

            netPtr->layerPtr[1]->neuronPtr[2]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[2]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[2]->w[0] = (float)-0.8182;
            netPtr->layerPtr[1]->neuronPtr[2]->w[1] = (float)0.0001818;

            netPtr->layerPtr[1]->neuronPtr[3]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[3]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[3]->w[0] = (float)0.01518;
            netPtr->layerPtr[1]->neuronPtr[3]->w[1] = (float)0.08322;

            netPtr->layerPtr[1]->neuronPtr[4]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[4]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[4]->w[0] = (float)0.09417;
            netPtr->layerPtr[1]->neuronPtr[4]->w[1] = (float)14.98;

            netPtr->layerPtr[1]->neuronPtr[5]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[5]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[5]->w[0] = (float)-1.656;
            netPtr->layerPtr[1]->neuronPtr[5]->w[1] = (float)0.008691;
```

```
            netPtr->layerPtr[1]->neuronPtr[6]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[6]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[6]->w[0] = (float)-0.9572;
            netPtr->layerPtr[1]->neuronPtr[6]->w[1] = (float)5.556;

            netPtr->layerPtr[1]->neuronPtr[7]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[7]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[7]->w[0] = (float)-0.228;
            netPtr->layerPtr[1]->neuronPtr[7]->w[1] = (float)9.55;

            netPtr->layerPtr[1]->neuronPtr[8]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[8]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[8]->w[0] = (float)0.2639;
            netPtr->layerPtr[1]->neuronPtr[8]->w[1] = (float)2.465;

            netPtr->layerPtr[1]->neuronPtr[9]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[9]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[9]->w[0] = (float)-0.3378;
            netPtr->layerPtr[1]->neuronPtr[9]->w[1] = (float)0.0001277;

            netPtr->layerPtr[1]->neuronPtr[10]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[10]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[10]->w[0] = (float)0.8904;
            netPtr->layerPtr[1]->neuronPtr[10]->w[1] = (float)4.477;

            netPtr->layerPtr[1]->neuronPtr[11]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[11]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[11]->w[0] = (float)-0.2426;
            netPtr->layerPtr[1]->neuronPtr[11]->w[1] = (float)0.005534;

            netPtr->layerPtr[1]->neuronPtr[12]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[12]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[12]->w[0] = (float)-1.656;
            netPtr->layerPtr[1]->neuronPtr[12]->w[1] = (float)0.008692;

            netPtr->layerPtr[1]->neuronPtr[13]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[13]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[13]->w[0] = (float)-0.956;
            netPtr->layerPtr[1]->neuronPtr[13]->w[1] = (float)5.551;

            netPtr->layerPtr[1]->neuronPtr[14]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[14]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[14]->w[0] = (float)-0.228;
            netPtr->layerPtr[1]->neuronPtr[14]->w[1] = (float)9.551;

            netPtr->layerPtr[1]->neuronPtr[15]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[15]->threshold = (float)0;
```

```
            netPtr->layerPtr[1]->neuronPtr[15]->w[0] = (float)0.2648;
            netPtr->layerPtr[1]->neuronPtr[15]->w[1] = (float)2.466;

            netPtr->layerPtr[1]->neuronPtr[16]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[16]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[16]->w[0] = (float)-0.3378;
            netPtr->layerPtr[1]->neuronPtr[16]->w[1] = (float)0.0001277;

            netPtr->layerPtr[1]->neuronPtr[17]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[17]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[17]->w[0] = (float)0.8897;
            netPtr->layerPtr[1]->neuronPtr[17]->w[1] = (float)4.478;

            netPtr->layerPtr[1]->neuronPtr[18]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[18]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[18]->w[0] = (float)-0.2413;
            netPtr->layerPtr[1]->neuronPtr[18]->w[1] = (float)0.005541;

            netPtr->layerPtr[1]->neuronPtr[19]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[19]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[19]->w[0] = (float)-1.657;
            netPtr->layerPtr[1]->neuronPtr[19]->w[1] = (float)0.008694;

            netPtr->layerPtr[1]->neuronPtr[20]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[20]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[20]->w[0] = (float)-0.9546;
            netPtr->layerPtr[1]->neuronPtr[20]->w[1] = (float)5.548;

            netPtr->layerPtr[1]->neuronPtr[21]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[21]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[21]->w[0] = (float)-0.228;
            netPtr->layerPtr[1]->neuronPtr[21]->w[1] = (float)9.549;

            netPtr->layerPtr[1]->neuronPtr[22]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[22]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[22]->w[0] = (float)0.2656;
            netPtr->layerPtr[1]->neuronPtr[22]->w[1] = (float)2.467;

            netPtr->layerPtr[1]->neuronPtr[23]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[23]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[23]->w[0] = (float)-0.3378;
            netPtr->layerPtr[1]->neuronPtr[23]->w[1] = (float)0.0001277;

            netPtr->layerPtr[1]->neuronPtr[24]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[24]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[24]->w[0] = (float)0.8903;
            netPtr->layerPtr[1]->neuronPtr[24]->w[1] = (float)4.482;
```

```
            netPtr->layerPtr[1]->neuronPtr[25]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[25]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[25]->w[0] = (float)-0.2399;
            netPtr->layerPtr[1]->neuronPtr[25]->w[1] = (float)0.005548;

            netPtr->layerPtr[1]->neuronPtr[26]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[26]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[26]->w[0] = (float)-0.009425;
            netPtr->layerPtr[1]->neuronPtr[26]->w[1] = (float)0.07738;

            netPtr->layerPtr[1]->neuronPtr[27]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[27]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[27]->w[0] = (float)-0.06876;
            netPtr->layerPtr[1]->neuronPtr[27]->w[1] = (float)49.03;

            netPtr->layerPtr[1]->neuronPtr[28]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[28]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[28]->w[0] = (float)-0.008811;
            netPtr->layerPtr[1]->neuronPtr[28]->w[1] = (float)0.07729;

            netPtr->layerPtr[1]->neuronPtr[29]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[29]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[29]->w[0] = (float)-0.07689;
            netPtr->layerPtr[1]->neuronPtr[29]->w[1] = (float)48.2;

            netPtr->layerPtr[1]->neuronPtr[30]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[30]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[30]->w[0] = (float)-0.008432;
            netPtr->layerPtr[1]->neuronPtr[30]->w[1] = (float)0.07714;

            netPtr->layerPtr[1]->neuronPtr[31]->tf =
(transfer_function_type)0;
            netPtr->layerPtr[1]->neuronPtr[31]->threshold = (float)0;
            netPtr->layerPtr[1]->neuronPtr[31]->w[0] = (float)-0.04383;
            netPtr->layerPtr[1]->neuronPtr[31]->w[1] = (float)52.39;

            netPtr->layerPtr[2]->neuronPtr[1]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[1]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[1]->w[0] = (float)-0.3578;
            netPtr->layerPtr[2]->neuronPtr[1]->w[1] = (float)-0.1022;
            netPtr->layerPtr[2]->neuronPtr[1]->w[2] = (float)0.05405;
            netPtr->layerPtr[2]->neuronPtr[1]->w[3] = (float)-0.0333;
            netPtr->layerPtr[2]->neuronPtr[1]->w[4] = (float)0.2466;
            netPtr->layerPtr[2]->neuronPtr[1]->w[5] = (float)0.1565;
            netPtr->layerPtr[2]->neuronPtr[1]->w[6] = (float)-0.04836;
            netPtr->layerPtr[2]->neuronPtr[1]->w[7] = (float)0.2928;
            netPtr->layerPtr[2]->neuronPtr[1]->w[8] = (float)0.2499;
            netPtr->layerPtr[2]->neuronPtr[1]->w[9] = (float)-0.2909;
            netPtr->layerPtr[2]->neuronPtr[1]->w[10] = (float)-0.0422;
            netPtr->layerPtr[2]->neuronPtr[1]->w[11] = (float)0.207;
```

```
            netPtr->layerPtr[2]->neuronPtr[1]->w[12] = (float)-0.05883;
            netPtr->layerPtr[2]->neuronPtr[1]->w[13] = (float)-0.3037;
            netPtr->layerPtr[2]->neuronPtr[1]->w[14] = (float)0.3377;
            netPtr->layerPtr[2]->neuronPtr[1]->w[15] = (float)-0.1052;
            netPtr->layerPtr[2]->neuronPtr[1]->w[16] = (float)0.3641;
            netPtr->layerPtr[2]->neuronPtr[1]->w[17] = (float)0.2341;
            netPtr->layerPtr[2]->neuronPtr[1]->w[18] = (float)-0.3816;
            netPtr->layerPtr[2]->neuronPtr[1]->w[19] = (float)-0.2845;
            netPtr->layerPtr[2]->neuronPtr[1]->w[20] = (float)0.5151;
            netPtr->layerPtr[2]->neuronPtr[1]->w[21] = (float)-0.4038;
            netPtr->layerPtr[2]->neuronPtr[1]->w[22] = (float)-0.0504;
            netPtr->layerPtr[2]->neuronPtr[1]->w[23] = (float)0.1536;
            netPtr->layerPtr[2]->neuronPtr[1]->w[24] = (float)-0.05454;
            netPtr->layerPtr[2]->neuronPtr[1]->w[25] = (float)0.3925;
            netPtr->layerPtr[2]->neuronPtr[1]->w[26] = (float)-0.2775;
            netPtr->layerPtr[2]->neuronPtr[1]->w[27] = (float)0.1167;
            netPtr->layerPtr[2]->neuronPtr[1]->w[28] = (float)0.1982;
            netPtr->layerPtr[2]->neuronPtr[1]->w[29] = (float)-0.05844;
            netPtr->layerPtr[2]->neuronPtr[1]->w[30] = (float)-0.1614;
            netPtr->layerPtr[2]->neuronPtr[1]->w[31] = (float)0.1309;

            netPtr->layerPtr[2]->neuronPtr[2]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[2]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[2]->w[0] = (float)0.4986;
            netPtr->layerPtr[2]->neuronPtr[2]->w[1] = (float)0.1502;
            netPtr->layerPtr[2]->neuronPtr[2]->w[2] = (float)-0.05925;
            netPtr->layerPtr[2]->neuronPtr[2]->w[3] = (float)0.01046;
            netPtr->layerPtr[2]->neuronPtr[2]->w[4] = (float)-0.1907;
            netPtr->layerPtr[2]->neuronPtr[2]->w[5] = (float)-0.06433;
            netPtr->layerPtr[2]->neuronPtr[2]->w[6] = (float)-0.2796;
            netPtr->layerPtr[2]->neuronPtr[2]->w[7] = (float)-0.1688;
            netPtr->layerPtr[2]->neuronPtr[2]->w[8] = (float)0.1681;
            netPtr->layerPtr[2]->neuronPtr[2]->w[9] = (float)-0.3701;
            netPtr->layerPtr[2]->neuronPtr[2]->w[10] = (float)0.14;
            netPtr->layerPtr[2]->neuronPtr[2]->w[11] = (float)0.1149;
            netPtr->layerPtr[2]->neuronPtr[2]->w[12] = (float)-0.1936;
            netPtr->layerPtr[2]->neuronPtr[2]->w[13] = (float)-0.1355;
            netPtr->layerPtr[2]->neuronPtr[2]->w[14] = (float)0.08579;
            netPtr->layerPtr[2]->neuronPtr[2]->w[15] = (float)0.2088;
            netPtr->layerPtr[2]->neuronPtr[2]->w[16] = (float)-0.1613;
            netPtr->layerPtr[2]->neuronPtr[2]->w[17] = (float)-0.1355;
            netPtr->layerPtr[2]->neuronPtr[2]->w[18] = (float)0.188;
            netPtr->layerPtr[2]->neuronPtr[2]->w[19] = (float)-0.3392;
            netPtr->layerPtr[2]->neuronPtr[2]->w[20] = (float)-0.1466;
            netPtr->layerPtr[2]->neuronPtr[2]->w[21] = (float)0.1848;
            netPtr->layerPtr[2]->neuronPtr[2]->w[22] = (float)-0.1605;
            netPtr->layerPtr[2]->neuronPtr[2]->w[23] = (float)-0.1314;
            netPtr->layerPtr[2]->neuronPtr[2]->w[24] = (float)0.1031;
            netPtr->layerPtr[2]->neuronPtr[2]->w[25] = (float)0.1602;
            netPtr->layerPtr[2]->neuronPtr[2]->w[26] = (float)0.07414;
            netPtr->layerPtr[2]->neuronPtr[2]->w[27] = (float)0.5007;
            netPtr->layerPtr[2]->neuronPtr[2]->w[28] = (float)0.01055;
            netPtr->layerPtr[2]->neuronPtr[2]->w[29] = (float)-0.2587;
            netPtr->layerPtr[2]->neuronPtr[2]->w[30] = (float)0.3823;
            netPtr->layerPtr[2]->neuronPtr[2]->w[31] = (float)-0.2525;
```

```
            netPtr->layerPtr[2]->neuronPtr[3]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[3]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[3]->w[0] = (float)-0.1047;
            netPtr->layerPtr[2]->neuronPtr[3]->w[1] = (float)0.0706;
            netPtr->layerPtr[2]->neuronPtr[3]->w[2] = (float)-0.1389;
            netPtr->layerPtr[2]->neuronPtr[3]->w[3] = (float)0.241;
            netPtr->layerPtr[2]->neuronPtr[3]->w[4] = (float)-0.2054;
            netPtr->layerPtr[2]->neuronPtr[3]->w[5] = (float)-0.1981;
            netPtr->layerPtr[2]->neuronPtr[3]->w[6] = (float)-0.2809;
            netPtr->layerPtr[2]->neuronPtr[3]->w[7] = (float)-0.4099;
            netPtr->layerPtr[2]->neuronPtr[3]->w[8] = (float)-0.1175;
            netPtr->layerPtr[2]->neuronPtr[3]->w[9] = (float)0.154;
            netPtr->layerPtr[2]->neuronPtr[3]->w[10] = (float)0.3826;
            netPtr->layerPtr[2]->neuronPtr[3]->w[11] = (float)0.2235;
            netPtr->layerPtr[2]->neuronPtr[3]->w[12] = (float)-0.4076;
            netPtr->layerPtr[2]->neuronPtr[3]->w[13] = (float)-0.007999;
            netPtr->layerPtr[2]->neuronPtr[3]->w[14] = (float)0.5648;
            netPtr->layerPtr[2]->neuronPtr[3]->w[15] = (float)0.1721;
            netPtr->layerPtr[2]->neuronPtr[3]->w[16] = (float)-0.3454;
            netPtr->layerPtr[2]->neuronPtr[3]->w[17] = (float)-0.3328;
            netPtr->layerPtr[2]->neuronPtr[3]->w[18] = (float)-0.09247;
            netPtr->layerPtr[2]->neuronPtr[3]->w[19] = (float)0.3322;
            netPtr->layerPtr[2]->neuronPtr[3]->w[20] = (float)0.2635;
            netPtr->layerPtr[2]->neuronPtr[3]->w[21] = (float)-0.1652;
            netPtr->layerPtr[2]->neuronPtr[3]->w[22] = (float)0.2614;
            netPtr->layerPtr[2]->neuronPtr[3]->w[23] = (float)0.3895;
            netPtr->layerPtr[2]->neuronPtr[3]->w[24] = (float)0.0955;
            netPtr->layerPtr[2]->neuronPtr[3]->w[25] = (float)0.04142;
            netPtr->layerPtr[2]->neuronPtr[3]->w[26] = (float)0.2369;
            netPtr->layerPtr[2]->neuronPtr[3]->w[27] = (float)-0.1577;
            netPtr->layerPtr[2]->neuronPtr[3]->w[28] = (float)-0.3493;
            netPtr->layerPtr[2]->neuronPtr[3]->w[29] = (float)0.1614;
            netPtr->layerPtr[2]->neuronPtr[3]->w[30] = (float)0.2168;
            netPtr->layerPtr[2]->neuronPtr[3]->w[31] = (float)0.2779;

            netPtr->layerPtr[2]->neuronPtr[4]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[4]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[4]->w[0] = (float)0.1109;
            netPtr->layerPtr[2]->neuronPtr[4]->w[1] = (float)-0.03027;
            netPtr->layerPtr[2]->neuronPtr[4]->w[2] = (float)0.06014;
            netPtr->layerPtr[2]->neuronPtr[4]->w[3] = (float)0.2154;
            netPtr->layerPtr[2]->neuronPtr[4]->w[4] = (float)-0.1299;
            netPtr->layerPtr[2]->neuronPtr[4]->w[5] = (float)0.09878;
            netPtr->layerPtr[2]->neuronPtr[4]->w[6] = (float)-0.3156;
            netPtr->layerPtr[2]->neuronPtr[4]->w[7] = (float)-0.0301;
            netPtr->layerPtr[2]->neuronPtr[4]->w[8] = (float)-0.004091;
            netPtr->layerPtr[2]->neuronPtr[4]->w[9] = (float)0.05819;
            netPtr->layerPtr[2]->neuronPtr[4]->w[10] = (float)0.3018;
            netPtr->layerPtr[2]->neuronPtr[4]->w[11] = (float)0.1075;
            netPtr->layerPtr[2]->neuronPtr[4]->w[12] = (float)-0.1113;
            netPtr->layerPtr[2]->neuronPtr[4]->w[13] = (float)0.421;
            netPtr->layerPtr[2]->neuronPtr[4]->w[14] = (float)-0.2621;
            netPtr->layerPtr[2]->neuronPtr[4]->w[15] = (float)0.2002;
            netPtr->layerPtr[2]->neuronPtr[4]->w[16] = (float)0.0282;
            netPtr->layerPtr[2]->neuronPtr[4]->w[17] = (float)-0.2485;
```

```
            netPtr->layerPtr[2]->neuronPtr[4]->w[18] = (float)-0.09061;
            netPtr->layerPtr[2]->neuronPtr[4]->w[19] = (float)-0.3312;
            netPtr->layerPtr[2]->neuronPtr[4]->w[20] = (float)0.3784;
            netPtr->layerPtr[2]->neuronPtr[4]->w[21] = (float)0.4444;
            netPtr->layerPtr[2]->neuronPtr[4]->w[22] = (float)-0.04045;
            netPtr->layerPtr[2]->neuronPtr[4]->w[23] = (float)-0.03181;
            netPtr->layerPtr[2]->neuronPtr[4]->w[24] = (float)0.4365;
            netPtr->layerPtr[2]->neuronPtr[4]->w[25] = (float)-0.008869;
            netPtr->layerPtr[2]->neuronPtr[4]->w[26] = (float)0.3364;
            netPtr->layerPtr[2]->neuronPtr[4]->w[27] = (float)-0.5896;
            netPtr->layerPtr[2]->neuronPtr[4]->w[28] = (float)0.2983;
            netPtr->layerPtr[2]->neuronPtr[4]->w[29] = (float)0.1409;
            netPtr->layerPtr[2]->neuronPtr[4]->w[30] = (float)-0.2157;
            netPtr->layerPtr[2]->neuronPtr[4]->w[31] = (float)0.1466;

            netPtr->layerPtr[2]->neuronPtr[5]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[5]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[5]->w[0] = (float)-0.5761;
            netPtr->layerPtr[2]->neuronPtr[5]->w[1] = (float)-0.08807;
            netPtr->layerPtr[2]->neuronPtr[5]->w[2] = (float)0.1088;
            netPtr->layerPtr[2]->neuronPtr[5]->w[3] = (float)0.2048;
            netPtr->layerPtr[2]->neuronPtr[5]->w[4] = (float)0.1634;
            netPtr->layerPtr[2]->neuronPtr[5]->w[5] = (float)0.1356;
            netPtr->layerPtr[2]->neuronPtr[5]->w[6] = (float)0.3603;
            netPtr->layerPtr[2]->neuronPtr[5]->w[7] = (float)-0.4849;
            netPtr->layerPtr[2]->neuronPtr[5]->w[8] = (float)0.03373;
            netPtr->layerPtr[2]->neuronPtr[5]->w[9] = (float)0.266;
            netPtr->layerPtr[2]->neuronPtr[5]->w[10] = (float)-0.4309;
            netPtr->layerPtr[2]->neuronPtr[5]->w[11] = (float)0.2471;
            netPtr->layerPtr[2]->neuronPtr[5]->w[12] = (float)-0.1991;
            netPtr->layerPtr[2]->neuronPtr[5]->w[13] = (float)-0.03692;
            netPtr->layerPtr[2]->neuronPtr[5]->w[14] = (float)0.1466;
            netPtr->layerPtr[2]->neuronPtr[5]->w[15] = (float)0.4149;
            netPtr->layerPtr[2]->neuronPtr[5]->w[16] = (float)0.03624;
            netPtr->layerPtr[2]->neuronPtr[5]->w[17] = (float)0.07839;
            netPtr->layerPtr[2]->neuronPtr[5]->w[18] = (float)-0.385;
            netPtr->layerPtr[2]->neuronPtr[5]->w[19] = (float)0.306;
            netPtr->layerPtr[2]->neuronPtr[5]->w[20] = (float)-0.3694;
            netPtr->layerPtr[2]->neuronPtr[5]->w[21] = (float)0.334;
            netPtr->layerPtr[2]->neuronPtr[5]->w[22] = (float)-0.3897;
            netPtr->layerPtr[2]->neuronPtr[5]->w[23] = (float)0.04121;
            netPtr->layerPtr[2]->neuronPtr[5]->w[24] = (float)-0.007637;
            netPtr->layerPtr[2]->neuronPtr[5]->w[25] = (float)-0.2741;
            netPtr->layerPtr[2]->neuronPtr[5]->w[26] = (float)-0.4892;
            netPtr->layerPtr[2]->neuronPtr[5]->w[27] = (float)0.3902;
            netPtr->layerPtr[2]->neuronPtr[5]->w[28] = (float)-0.05148;
            netPtr->layerPtr[2]->neuronPtr[5]->w[29] = (float)0.4334;
            netPtr->layerPtr[2]->neuronPtr[5]->w[30] = (float)0.2627;
            netPtr->layerPtr[2]->neuronPtr[5]->w[31] = (float)-0.2919;

            netPtr->layerPtr[2]->neuronPtr[6]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[6]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[6]->w[0] = (float)-0.4503;
            netPtr->layerPtr[2]->neuronPtr[6]->w[1] = (float)0.2431;
            netPtr->layerPtr[2]->neuronPtr[6]->w[2] = (float)-0.4354;
```

```
            netPtr->layerPtr[2]->neuronPtr[6]->w[3] = (float)0.2328;
            netPtr->layerPtr[2]->neuronPtr[6]->w[4] = (float)0.1662;
            netPtr->layerPtr[2]->neuronPtr[6]->w[5] = (float)-0.1451;
            netPtr->layerPtr[2]->neuronPtr[6]->w[6] = (float)0.4627;
            netPtr->layerPtr[2]->neuronPtr[6]->w[7] = (float)-0.3663;
            netPtr->layerPtr[2]->neuronPtr[6]->w[8] = (float)-0.4208;
            netPtr->layerPtr[2]->neuronPtr[6]->w[9] = (float)0.194;
            netPtr->layerPtr[2]->neuronPtr[6]->w[10] = (float)-0.03267;
            netPtr->layerPtr[2]->neuronPtr[6]->w[11] = (float)-0.2498;
            netPtr->layerPtr[2]->neuronPtr[6]->w[12] = (float)0.3106;
            netPtr->layerPtr[2]->neuronPtr[6]->w[13] = (float)-0.4114;
            netPtr->layerPtr[2]->neuronPtr[6]->w[14] = (float)0.4027;
            netPtr->layerPtr[2]->neuronPtr[6]->w[15] = (float)-0.3256;
            netPtr->layerPtr[2]->neuronPtr[6]->w[16] = (float)0.184;
            netPtr->layerPtr[2]->neuronPtr[6]->w[17] = (float)0.4999;
            netPtr->layerPtr[2]->neuronPtr[6]->w[18] = (float)-0.09063;
            netPtr->layerPtr[2]->neuronPtr[6]->w[19] = (float)-0.1401;
            netPtr->layerPtr[2]->neuronPtr[6]->w[20] = (float)-0.1057;
            netPtr->layerPtr[2]->neuronPtr[6]->w[21] = (float)0.08788;
            netPtr->layerPtr[2]->neuronPtr[6]->w[22] = (float)-0.1655;
            netPtr->layerPtr[2]->neuronPtr[6]->w[23] = (float)0.4385;
            netPtr->layerPtr[2]->neuronPtr[6]->w[24] = (float)-0.3917;
            netPtr->layerPtr[2]->neuronPtr[6]->w[25] = (float)0.2579;
            netPtr->layerPtr[2]->neuronPtr[6]->w[26] = (float)-0.03475;
            netPtr->layerPtr[2]->neuronPtr[6]->w[27] = (float)0.358;
            netPtr->layerPtr[2]->neuronPtr[6]->w[28] = (float)0.1021;
            netPtr->layerPtr[2]->neuronPtr[6]->w[29] = (float)-0.2207;
            netPtr->layerPtr[2]->neuronPtr[6]->w[30] = (float)0.1079;
            netPtr->layerPtr[2]->neuronPtr[6]->w[31] = (float)-0.0738;

            netPtr->layerPtr[2]->neuronPtr[7]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[7]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[7]->w[0] = (float)-0.2875;
            netPtr->layerPtr[2]->neuronPtr[7]->w[1] = (float)-0.04928;
            netPtr->layerPtr[2]->neuronPtr[7]->w[2] = (float)-0.4607;
            netPtr->layerPtr[2]->neuronPtr[7]->w[3] = (float)-0.3858;
            netPtr->layerPtr[2]->neuronPtr[7]->w[4] = (float)0.05784;
            netPtr->layerPtr[2]->neuronPtr[7]->w[5] = (float)-0.414;
            netPtr->layerPtr[2]->neuronPtr[7]->w[6] = (float)-0.4052;
            netPtr->layerPtr[2]->neuronPtr[7]->w[7] = (float)0.07611;
            netPtr->layerPtr[2]->neuronPtr[7]->w[8] = (float)0.05932;
            netPtr->layerPtr[2]->neuronPtr[7]->w[9] = (float)-0.3966;
            netPtr->layerPtr[2]->neuronPtr[7]->w[10] = (float)0.1592;
            netPtr->layerPtr[2]->neuronPtr[7]->w[11] = (float)-0.2299;
            netPtr->layerPtr[2]->neuronPtr[7]->w[12] = (float)-0.499;
            netPtr->layerPtr[2]->neuronPtr[7]->w[13] = (float)-0.2249;
            netPtr->layerPtr[2]->neuronPtr[7]->w[14] = (float)0.09851;
            netPtr->layerPtr[2]->neuronPtr[7]->w[15] = (float)0.2965;
            netPtr->layerPtr[2]->neuronPtr[7]->w[16] = (float)0.5127;
            netPtr->layerPtr[2]->neuronPtr[7]->w[17] = (float)-0.01012;
            netPtr->layerPtr[2]->neuronPtr[7]->w[18] = (float)-0.3659;
            netPtr->layerPtr[2]->neuronPtr[7]->w[19] = (float)-0.4589;
            netPtr->layerPtr[2]->neuronPtr[7]->w[20] = (float)-0.2499;
            netPtr->layerPtr[2]->neuronPtr[7]->w[21] = (float)-0.4578;
            netPtr->layerPtr[2]->neuronPtr[7]->w[22] = (float)0.4614;
            netPtr->layerPtr[2]->neuronPtr[7]->w[23] = (float)0.0355;
```

```
            netPtr->layerPtr[2]->neuronPtr[7]->w[24] = (float)0.4522;
            netPtr->layerPtr[2]->neuronPtr[7]->w[25] = (float)-0.05924;
            netPtr->layerPtr[2]->neuronPtr[7]->w[26] = (float)-0.06682;
            netPtr->layerPtr[2]->neuronPtr[7]->w[27] = (float)0.1799;
            netPtr->layerPtr[2]->neuronPtr[7]->w[28] = (float)0.08489;
            netPtr->layerPtr[2]->neuronPtr[7]->w[29] = (float)0.4091;
            netPtr->layerPtr[2]->neuronPtr[7]->w[30] = (float)0.04114;
            netPtr->layerPtr[2]->neuronPtr[7]->w[31] = (float)-0.008181;

            netPtr->layerPtr[2]->neuronPtr[8]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[8]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[8]->w[0] = (float)-0.2714;
            netPtr->layerPtr[2]->neuronPtr[8]->w[1] = (float)-0.4403;
            netPtr->layerPtr[2]->neuronPtr[8]->w[2] = (float)-0.1744;
            netPtr->layerPtr[2]->neuronPtr[8]->w[3] = (float)-0.3516;
            netPtr->layerPtr[2]->neuronPtr[8]->w[4] = (float)0.3489;
            netPtr->layerPtr[2]->neuronPtr[8]->w[5] = (float)0.4919;
            netPtr->layerPtr[2]->neuronPtr[8]->w[6] = (float)0.4659;
            netPtr->layerPtr[2]->neuronPtr[8]->w[7] = (float)-0.1446;
            netPtr->layerPtr[2]->neuronPtr[8]->w[8] = (float)0.437;
            netPtr->layerPtr[2]->neuronPtr[8]->w[9] = (float)0.5155;
            netPtr->layerPtr[2]->neuronPtr[8]->w[10] = (float)-0.4266;
            netPtr->layerPtr[2]->neuronPtr[8]->w[11] = (float)0.3016;
            netPtr->layerPtr[2]->neuronPtr[8]->w[12] = (float)-0.1283;
            netPtr->layerPtr[2]->neuronPtr[8]->w[13] = (float)-0.413;
            netPtr->layerPtr[2]->neuronPtr[8]->w[14] = (float)0.1258;
            netPtr->layerPtr[2]->neuronPtr[8]->w[15] = (float)-0.04682;
            netPtr->layerPtr[2]->neuronPtr[8]->w[16] = (float)-0.2841;
            netPtr->layerPtr[2]->neuronPtr[8]->w[17] = (float)0.4121;
            netPtr->layerPtr[2]->neuronPtr[8]->w[18] = (float)0.4636;
            netPtr->layerPtr[2]->neuronPtr[8]->w[19] = (float)-0.2033;
            netPtr->layerPtr[2]->neuronPtr[8]->w[20] = (float)0.2683;
            netPtr->layerPtr[2]->neuronPtr[8]->w[21] = (float)0.3738;
            netPtr->layerPtr[2]->neuronPtr[8]->w[22] = (float)-0.4327;
            netPtr->layerPtr[2]->neuronPtr[8]->w[23] = (float)-0.03309;
            netPtr->layerPtr[2]->neuronPtr[8]->w[24] = (float)0.3159;
            netPtr->layerPtr[2]->neuronPtr[8]->w[25] = (float)0.06743;
            netPtr->layerPtr[2]->neuronPtr[8]->w[26] = (float)-0.5684;
            netPtr->layerPtr[2]->neuronPtr[8]->w[27] = (float)-0.2941;
            netPtr->layerPtr[2]->neuronPtr[8]->w[28] = (float)-0.2206;
            netPtr->layerPtr[2]->neuronPtr[8]->w[29] = (float)-0.1643;
            netPtr->layerPtr[2]->neuronPtr[8]->w[30] = (float)0.05852;
            netPtr->layerPtr[2]->neuronPtr[8]->w[31] = (float)-0.2157;

            netPtr->layerPtr[2]->neuronPtr[9]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[9]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[9]->w[0] = (float)0.5199;
            netPtr->layerPtr[2]->neuronPtr[9]->w[1] = (float)-0.0753;
            netPtr->layerPtr[2]->neuronPtr[9]->w[2] = (float)-0.1136;
            netPtr->layerPtr[2]->neuronPtr[9]->w[3] = (float)-0.2598;
            netPtr->layerPtr[2]->neuronPtr[9]->w[4] = (float)-0.2207;
            netPtr->layerPtr[2]->neuronPtr[9]->w[5] = (float)0.1866;
            netPtr->layerPtr[2]->neuronPtr[9]->w[6] = (float)-0.1947;
            netPtr->layerPtr[2]->neuronPtr[9]->w[7] = (float)-0.3004;
            netPtr->layerPtr[2]->neuronPtr[9]->w[8] = (float)-0.03678;
```

```
            netPtr->layerPtr[2]->neuronPtr[9]->w[9] = (float)0.4718;
            netPtr->layerPtr[2]->neuronPtr[9]->w[10] = (float)0.004148;
            netPtr->layerPtr[2]->neuronPtr[9]->w[11] = (float)-0.1387;
            netPtr->layerPtr[2]->neuronPtr[9]->w[12] = (float)-0.4192;
            netPtr->layerPtr[2]->neuronPtr[9]->w[13] = (float)0.08172;
            netPtr->layerPtr[2]->neuronPtr[9]->w[14] = (float)0.3809;
            netPtr->layerPtr[2]->neuronPtr[9]->w[15] = (float)-0.06241;
            netPtr->layerPtr[2]->neuronPtr[9]->w[16] = (float)-0.05715;
            netPtr->layerPtr[2]->neuronPtr[9]->w[17] = (float)0.2731;
            netPtr->layerPtr[2]->neuronPtr[9]->w[18] = (float)-0.0157;
            netPtr->layerPtr[2]->neuronPtr[9]->w[19] = (float)0.03569;
            netPtr->layerPtr[2]->neuronPtr[9]->w[20] = (float)-0.4112;
            netPtr->layerPtr[2]->neuronPtr[9]->w[21] = (float)0.1174;
            netPtr->layerPtr[2]->neuronPtr[9]->w[22] = (float)-0.4284;
            netPtr->layerPtr[2]->neuronPtr[9]->w[23] = (float)-0.1721;
            netPtr->layerPtr[2]->neuronPtr[9]->w[24] = (float)-0.3395;
            netPtr->layerPtr[2]->neuronPtr[9]->w[25] = (float)0.05235;
            netPtr->layerPtr[2]->neuronPtr[9]->w[26] = (float)-0.4335;
            netPtr->layerPtr[2]->neuronPtr[9]->w[27] = (float)0.2436;
            netPtr->layerPtr[2]->neuronPtr[9]->w[28] = (float)-0.1492;
            netPtr->layerPtr[2]->neuronPtr[9]->w[29] = (float)-0.5338;
            netPtr->layerPtr[2]->neuronPtr[9]->w[30] = (float)-0.4091;
            netPtr->layerPtr[2]->neuronPtr[9]->w[31] = (float)0.3645;

            netPtr->layerPtr[2]->neuronPtr[10]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[10]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[10]->w[0] = (float)0.3804;
            netPtr->layerPtr[2]->neuronPtr[10]->w[1] = (float)-0.06391;
            netPtr->layerPtr[2]->neuronPtr[10]->w[2] = (float)0.05932;
            netPtr->layerPtr[2]->neuronPtr[10]->w[3] = (float)-0.4117;
            netPtr->layerPtr[2]->neuronPtr[10]->w[4] = (float)-0.4215;
            netPtr->layerPtr[2]->neuronPtr[10]->w[5] = (float)0.2763;
            netPtr->layerPtr[2]->neuronPtr[10]->w[6] = (float)-0.3147;
            netPtr->layerPtr[2]->neuronPtr[10]->w[7] = (float)0.1157;
            netPtr->layerPtr[2]->neuronPtr[10]->w[8] = (float)0.342;
            netPtr->layerPtr[2]->neuronPtr[10]->w[9] = (float)0.3344;
            netPtr->layerPtr[2]->neuronPtr[10]->w[10] = (float)0.08407;
            netPtr->layerPtr[2]->neuronPtr[10]->w[11] = (float)0.4147;
            netPtr->layerPtr[2]->neuronPtr[10]->w[12] = (float)0.3714;
            netPtr->layerPtr[2]->neuronPtr[10]->w[13] = (float)0.1701;
            netPtr->layerPtr[2]->neuronPtr[10]->w[14] = (float)0.2953;
            netPtr->layerPtr[2]->neuronPtr[10]->w[15] = (float)-0.04187;
            netPtr->layerPtr[2]->neuronPtr[10]->w[16] = (float)-0.3106;
            netPtr->layerPtr[2]->neuronPtr[10]->w[17] = (float)0.2651;
            netPtr->layerPtr[2]->neuronPtr[10]->w[18] = (float)-0.361;
            netPtr->layerPtr[2]->neuronPtr[10]->w[19] = (float)0.2365;
            netPtr->layerPtr[2]->neuronPtr[10]->w[20] = (float)-0.2698;
            netPtr->layerPtr[2]->neuronPtr[10]->w[21] = (float)0.01587;
            netPtr->layerPtr[2]->neuronPtr[10]->w[22] = (float)0.2237;
            netPtr->layerPtr[2]->neuronPtr[10]->w[23] = (float)-0.1359;
            netPtr->layerPtr[2]->neuronPtr[10]->w[24] = (float)0.08443;
            netPtr->layerPtr[2]->neuronPtr[10]->w[25] = (float)0.2358;
            netPtr->layerPtr[2]->neuronPtr[10]->w[26] = (float)0.2038;
            netPtr->layerPtr[2]->neuronPtr[10]->w[27] = (float)-0.4105;
            netPtr->layerPtr[2]->neuronPtr[10]->w[28] = (float)-0.09885;
            netPtr->layerPtr[2]->neuronPtr[10]->w[29] = (float)0.4507;
```

```
            netPtr->layerPtr[2]->neuronPtr[10]->w[30] = (float)-0.06084;
            netPtr->layerPtr[2]->neuronPtr[10]->w[31] = (float)0.0902;

            netPtr->layerPtr[2]->neuronPtr[11]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[11]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[11]->w[0] = (float)0.4169;
            netPtr->layerPtr[2]->neuronPtr[11]->w[1] = (float)-0.35;
            netPtr->layerPtr[2]->neuronPtr[11]->w[2] = (float)0.07748;
            netPtr->layerPtr[2]->neuronPtr[11]->w[3] = (float)0.1953;
            netPtr->layerPtr[2]->neuronPtr[11]->w[4] = (float)0.3161;
            netPtr->layerPtr[2]->neuronPtr[11]->w[5] = (float)0.5212;
            netPtr->layerPtr[2]->neuronPtr[11]->w[6] = (float)-0.07947;
            netPtr->layerPtr[2]->neuronPtr[11]->w[7] = (float)-0.4286;
            netPtr->layerPtr[2]->neuronPtr[11]->w[8] = (float)-0.08024;
            netPtr->layerPtr[2]->neuronPtr[11]->w[9] = (float)0.07185;
            netPtr->layerPtr[2]->neuronPtr[11]->w[10] = (float)-0.0378;
            netPtr->layerPtr[2]->neuronPtr[11]->w[11] = (float)-0.09041;
            netPtr->layerPtr[2]->neuronPtr[11]->w[12] = (float)-0.1779;
            netPtr->layerPtr[2]->neuronPtr[11]->w[13] = (float)0.1055;
            netPtr->layerPtr[2]->neuronPtr[11]->w[14] = (float)0.06818;
            netPtr->layerPtr[2]->neuronPtr[11]->w[15] = (float)-0.005258;
            netPtr->layerPtr[2]->neuronPtr[11]->w[16] = (float)0.1274;
            netPtr->layerPtr[2]->neuronPtr[11]->w[17] = (float)-0.09255;
            netPtr->layerPtr[2]->neuronPtr[11]->w[18] = (float)0.4399;
            netPtr->layerPtr[2]->neuronPtr[11]->w[19] = (float)-0.1382;
            netPtr->layerPtr[2]->neuronPtr[11]->w[20] = (float)0.000303;
            netPtr->layerPtr[2]->neuronPtr[11]->w[21] = (float)0.2856;
            netPtr->layerPtr[2]->neuronPtr[11]->w[22] = (float)0.414;
            netPtr->layerPtr[2]->neuronPtr[11]->w[23] = (float)-0.1876;
            netPtr->layerPtr[2]->neuronPtr[11]->w[24] = (float)0.04862;
            netPtr->layerPtr[2]->neuronPtr[11]->w[25] = (float)-0.4408;
            netPtr->layerPtr[2]->neuronPtr[11]->w[26] = (float)-0.2871;
            netPtr->layerPtr[2]->neuronPtr[11]->w[27] = (float)0.2919;
            netPtr->layerPtr[2]->neuronPtr[11]->w[28] = (float)0.1191;
            netPtr->layerPtr[2]->neuronPtr[11]->w[29] = (float)0.2717;
            netPtr->layerPtr[2]->neuronPtr[11]->w[30] = (float)0.1637;
            netPtr->layerPtr[2]->neuronPtr[11]->w[31] = (float)0.2723;

            netPtr->layerPtr[2]->neuronPtr[12]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[12]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[12]->w[0] = (float)-0.3139;
            netPtr->layerPtr[2]->neuronPtr[12]->w[1] = (float)0.4401;
            netPtr->layerPtr[2]->neuronPtr[12]->w[2] = (float)0.03129;
            netPtr->layerPtr[2]->neuronPtr[12]->w[3] = (float)0.2816;
            netPtr->layerPtr[2]->neuronPtr[12]->w[4] = (float)0.2791;
            netPtr->layerPtr[2]->neuronPtr[12]->w[5] = (float)-0.3283;
            netPtr->layerPtr[2]->neuronPtr[12]->w[6] = (float)0.08532;
            netPtr->layerPtr[2]->neuronPtr[12]->w[7] = (float)0.2954;
            netPtr->layerPtr[2]->neuronPtr[12]->w[8] = (float)0.2005;
            netPtr->layerPtr[2]->neuronPtr[12]->w[9] = (float)-0.1008;
            netPtr->layerPtr[2]->neuronPtr[12]->w[10] = (float)0.4255;
            netPtr->layerPtr[2]->neuronPtr[12]->w[11] = (float)-0.06443;
            netPtr->layerPtr[2]->neuronPtr[12]->w[12] = (float)-0.2183;
            netPtr->layerPtr[2]->neuronPtr[12]->w[13] = (float)-0.26;
            netPtr->layerPtr[2]->neuronPtr[12]->w[14] = (float)-0.07313;
```

```
            netPtr->layerPtr[2]->neuronPtr[12]->w[15] = (float)-0.2797;
            netPtr->layerPtr[2]->neuronPtr[12]->w[16] = (float)-0.08569;
            netPtr->layerPtr[2]->neuronPtr[12]->w[17] = (float)-0.4109;
            netPtr->layerPtr[2]->neuronPtr[12]->w[18] = (float)0.3703;
            netPtr->layerPtr[2]->neuronPtr[12]->w[19] = (float)0.1121;
            netPtr->layerPtr[2]->neuronPtr[12]->w[20] = (float)0.174;
            netPtr->layerPtr[2]->neuronPtr[12]->w[21] = (float)-0.3245;
            netPtr->layerPtr[2]->neuronPtr[12]->w[22] = (float)0.4561;
            netPtr->layerPtr[2]->neuronPtr[12]->w[23] = (float)0.214;
            netPtr->layerPtr[2]->neuronPtr[12]->w[24] = (float)-0.05293;
            netPtr->layerPtr[2]->neuronPtr[12]->w[25] = (float)-0.296;
            netPtr->layerPtr[2]->neuronPtr[12]->w[26] = (float)0.2533;
            netPtr->layerPtr[2]->neuronPtr[12]->w[27] = (float)-0.5482;
            netPtr->layerPtr[2]->neuronPtr[12]->w[28] = (float)0.3558;
            netPtr->layerPtr[2]->neuronPtr[12]->w[29] = (float)0.005701;
            netPtr->layerPtr[2]->neuronPtr[12]->w[30] = (float)0.2109;
            netPtr->layerPtr[2]->neuronPtr[12]->w[31] = (float)0.4208;

            netPtr->layerPtr[2]->neuronPtr[13]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[13]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[13]->w[0] = (float)-0.6726;
            netPtr->layerPtr[2]->neuronPtr[13]->w[1] = (float)-0.04436;
            netPtr->layerPtr[2]->neuronPtr[13]->w[2] = (float)-0.1477;
            netPtr->layerPtr[2]->neuronPtr[13]->w[3] = (float)-0.4087;
            netPtr->layerPtr[2]->neuronPtr[13]->w[4] = (float)-0.122;
            netPtr->layerPtr[2]->neuronPtr[13]->w[5] = (float)-0.2213;
            netPtr->layerPtr[2]->neuronPtr[13]->w[6] = (float)-0.02454;
            netPtr->layerPtr[2]->neuronPtr[13]->w[7] = (float)-0.1934;
            netPtr->layerPtr[2]->neuronPtr[13]->w[8] = (float)0.2021;
            netPtr->layerPtr[2]->neuronPtr[13]->w[9] = (float)0.1929;
            netPtr->layerPtr[2]->neuronPtr[13]->w[10] = (float)-0.1632;
            netPtr->layerPtr[2]->neuronPtr[13]->w[11] = (float)-0.07106;
            netPtr->layerPtr[2]->neuronPtr[13]->w[12] = (float)0.09413;
            netPtr->layerPtr[2]->neuronPtr[13]->w[13] = (float)-0.2048;
            netPtr->layerPtr[2]->neuronPtr[13]->w[14] = (float)-0.02645;
            netPtr->layerPtr[2]->neuronPtr[13]->w[15] = (float)0.2275;
            netPtr->layerPtr[2]->neuronPtr[13]->w[16] = (float)0.2623;
            netPtr->layerPtr[2]->neuronPtr[13]->w[17] = (float)-0.07837;
            netPtr->layerPtr[2]->neuronPtr[13]->w[18] = (float)0.3844;
            netPtr->layerPtr[2]->neuronPtr[13]->w[19] = (float)-0.0005479;
            netPtr->layerPtr[2]->neuronPtr[13]->w[20] = (float)0.2762;
            netPtr->layerPtr[2]->neuronPtr[13]->w[21] = (float)0.2444;
            netPtr->layerPtr[2]->neuronPtr[13]->w[22] = (float)-0.1466;
            netPtr->layerPtr[2]->neuronPtr[13]->w[23] = (float)-0.2021;
            netPtr->layerPtr[2]->neuronPtr[13]->w[24] = (float)-0.2243;
            netPtr->layerPtr[2]->neuronPtr[13]->w[25] = (float)-0.3034;
            netPtr->layerPtr[2]->neuronPtr[13]->w[26] = (float)-0.5876;
            netPtr->layerPtr[2]->neuronPtr[13]->w[27] = (float)-0.1503;
            netPtr->layerPtr[2]->neuronPtr[13]->w[28] = (float)0.0504;
            netPtr->layerPtr[2]->neuronPtr[13]->w[29] = (float)-0.3318;
            netPtr->layerPtr[2]->neuronPtr[13]->w[30] = (float)0.1338;
            netPtr->layerPtr[2]->neuronPtr[13]->w[31] = (float)0.08225;

            netPtr->layerPtr[2]->neuronPtr[14]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[14]->threshold = (float)0;
```

```
            netPtr->layerPtr[2]->neuronPtr[14]->w[0] = (float)-0.4636;
            netPtr->layerPtr[2]->neuronPtr[14]->w[1] = (float)0.3506;
            netPtr->layerPtr[2]->neuronPtr[14]->w[2] = (float)-0.07182;
            netPtr->layerPtr[2]->neuronPtr[14]->w[3] = (float)0.3556;
            netPtr->layerPtr[2]->neuronPtr[14]->w[4] = (float)-0.2751;
            netPtr->layerPtr[2]->neuronPtr[14]->w[5] = (float)-0.3147;
            netPtr->layerPtr[2]->neuronPtr[14]->w[6] = (float)0.007271;
            netPtr->layerPtr[2]->neuronPtr[14]->w[7] = (float)0.3289;
            netPtr->layerPtr[2]->neuronPtr[14]->w[8] = (float)0.171;
            netPtr->layerPtr[2]->neuronPtr[14]->w[9] = (float)-0.2646;
            netPtr->layerPtr[2]->neuronPtr[14]->w[10] = (float)-0.06386;
            netPtr->layerPtr[2]->neuronPtr[14]->w[11] = (float)0.2157;
            netPtr->layerPtr[2]->neuronPtr[14]->w[12] = (float)-0.3643;
            netPtr->layerPtr[2]->neuronPtr[14]->w[13] = (float)-0.09034;
            netPtr->layerPtr[2]->neuronPtr[14]->w[14] = (float)-0.3239;
            netPtr->layerPtr[2]->neuronPtr[14]->w[15] = (float)0.04698;
            netPtr->layerPtr[2]->neuronPtr[14]->w[16] = (float)0.5163;
            netPtr->layerPtr[2]->neuronPtr[14]->w[17] = (float)0.3015;
            netPtr->layerPtr[2]->neuronPtr[14]->w[18] = (float)-0.4194;
            netPtr->layerPtr[2]->neuronPtr[14]->w[19] = (float)0.3961;
            netPtr->layerPtr[2]->neuronPtr[14]->w[20] = (float)-0.173;
            netPtr->layerPtr[2]->neuronPtr[14]->w[21] = (float)0.2018;
            netPtr->layerPtr[2]->neuronPtr[14]->w[22] = (float)-0.277;
            netPtr->layerPtr[2]->neuronPtr[14]->w[23] = (float)-0.2546;
            netPtr->layerPtr[2]->neuronPtr[14]->w[24] = (float)0.01194;
            netPtr->layerPtr[2]->neuronPtr[14]->w[25] = (float)0.3239;
            netPtr->layerPtr[2]->neuronPtr[14]->w[26] = (float)-0.1211;
            netPtr->layerPtr[2]->neuronPtr[14]->w[27] = (float)-0.03918;
            netPtr->layerPtr[2]->neuronPtr[14]->w[28] = (float)-0.1744;
            netPtr->layerPtr[2]->neuronPtr[14]->w[29] = (float)0.3489;
            netPtr->layerPtr[2]->neuronPtr[14]->w[30] = (float)0.1051;
            netPtr->layerPtr[2]->neuronPtr[14]->w[31] = (float)0.02605;

            netPtr->layerPtr[2]->neuronPtr[15]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[15]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[15]->w[0] = (float)-0.02628;
            netPtr->layerPtr[2]->neuronPtr[15]->w[1] = (float)-0.3259;
            netPtr->layerPtr[2]->neuronPtr[15]->w[2] = (float)-0.2179;
            netPtr->layerPtr[2]->neuronPtr[15]->w[3] = (float)0.09619;
            netPtr->layerPtr[2]->neuronPtr[15]->w[4] = (float)0.1913;
            netPtr->layerPtr[2]->neuronPtr[15]->w[5] = (float)0.0936;
            netPtr->layerPtr[2]->neuronPtr[15]->w[6] = (float)-0.1033;
            netPtr->layerPtr[2]->neuronPtr[15]->w[7] = (float)0.05172;
            netPtr->layerPtr[2]->neuronPtr[15]->w[8] = (float)0.3825;
            netPtr->layerPtr[2]->neuronPtr[15]->w[9] = (float)-0.1468;
            netPtr->layerPtr[2]->neuronPtr[15]->w[10] = (float)-0.1256;
            netPtr->layerPtr[2]->neuronPtr[15]->w[11] = (float)0.5015;
            netPtr->layerPtr[2]->neuronPtr[15]->w[12] = (float)-0.1209;
            netPtr->layerPtr[2]->neuronPtr[15]->w[13] = (float)0.03037;
            netPtr->layerPtr[2]->neuronPtr[15]->w[14] = (float)-0.3632;
            netPtr->layerPtr[2]->neuronPtr[15]->w[15] = (float)-0.1526;
            netPtr->layerPtr[2]->neuronPtr[15]->w[16] = (float)0.1182;
            netPtr->layerPtr[2]->neuronPtr[15]->w[17] = (float)0.02129;
            netPtr->layerPtr[2]->neuronPtr[15]->w[18] = (float)-0.4528;
            netPtr->layerPtr[2]->neuronPtr[15]->w[19] = (float)0.05973;
            netPtr->layerPtr[2]->neuronPtr[15]->w[20] = (float)-0.08522;
```

```
            netPtr->layerPtr[2]->neuronPtr[15]->w[21] = (float)0.1542;
            netPtr->layerPtr[2]->neuronPtr[15]->w[22] = (float)-0.4226;
            netPtr->layerPtr[2]->neuronPtr[15]->w[23] = (float)0.3088;
            netPtr->layerPtr[2]->neuronPtr[15]->w[24] = (float)0.2321;
            netPtr->layerPtr[2]->neuronPtr[15]->w[25] = (float)0.1287;
            netPtr->layerPtr[2]->neuronPtr[15]->w[26] = (float)0.1713;
            netPtr->layerPtr[2]->neuronPtr[15]->w[27] = (float)-0.519;
            netPtr->layerPtr[2]->neuronPtr[15]->w[28] = (float)-0.4615;
            netPtr->layerPtr[2]->neuronPtr[15]->w[29] = (float)0.3302;
            netPtr->layerPtr[2]->neuronPtr[15]->w[30] = (float)0.163;
            netPtr->layerPtr[2]->neuronPtr[15]->w[31] = (float)0.03142;

            netPtr->layerPtr[2]->neuronPtr[16]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[16]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[16]->w[0] = (float)-0.1047;
            netPtr->layerPtr[2]->neuronPtr[16]->w[1] = (float)-0.04061;
            netPtr->layerPtr[2]->neuronPtr[16]->w[2] = (float)0.1789;
            netPtr->layerPtr[2]->neuronPtr[16]->w[3] = (float)-0.2132;
            netPtr->layerPtr[2]->neuronPtr[16]->w[4] = (float)0.3077;
            netPtr->layerPtr[2]->neuronPtr[16]->w[5] = (float)-0.07273;
            netPtr->layerPtr[2]->neuronPtr[16]->w[6] = (float)-0.4222;
            netPtr->layerPtr[2]->neuronPtr[16]->w[7] = (float)-0.305;
            netPtr->layerPtr[2]->neuronPtr[16]->w[8] = (float)-0.2826;
            netPtr->layerPtr[2]->neuronPtr[16]->w[9] = (float)0.03982;
            netPtr->layerPtr[2]->neuronPtr[16]->w[10] = (float)-0.2787;
            netPtr->layerPtr[2]->neuronPtr[16]->w[11] = (float)0.07771;
            netPtr->layerPtr[2]->neuronPtr[16]->w[12] = (float)0.0675;
            netPtr->layerPtr[2]->neuronPtr[16]->w[13] = (float)0.2624;
            netPtr->layerPtr[2]->neuronPtr[16]->w[14] = (float)0.1899;
            netPtr->layerPtr[2]->neuronPtr[16]->w[15] = (float)0.5501;
            netPtr->layerPtr[2]->neuronPtr[16]->w[16] = (float)0.08476;
            netPtr->layerPtr[2]->neuronPtr[16]->w[17] = (float)-0.1446;
            netPtr->layerPtr[2]->neuronPtr[16]->w[18] = (float)0.4572;
            netPtr->layerPtr[2]->neuronPtr[16]->w[19] = (float)0.1827;
            netPtr->layerPtr[2]->neuronPtr[16]->w[20] = (float)-0.2254;
            netPtr->layerPtr[2]->neuronPtr[16]->w[21] = (float)-0.1182;
            netPtr->layerPtr[2]->neuronPtr[16]->w[22] = (float)-0.4185;
            netPtr->layerPtr[2]->neuronPtr[16]->w[23] = (float)-0.02019;
            netPtr->layerPtr[2]->neuronPtr[16]->w[24] = (float)-0.3157;
            netPtr->layerPtr[2]->neuronPtr[16]->w[25] = (float)0.1163;
            netPtr->layerPtr[2]->neuronPtr[16]->w[26] = (float)-0.4054;
            netPtr->layerPtr[2]->neuronPtr[16]->w[27] = (float)0.3532;
            netPtr->layerPtr[2]->neuronPtr[16]->w[28] = (float)-0.3977;
            netPtr->layerPtr[2]->neuronPtr[16]->w[29] = (float)0.2436;
            netPtr->layerPtr[2]->neuronPtr[16]->w[30] = (float)0.3181;
            netPtr->layerPtr[2]->neuronPtr[16]->w[31] = (float)0.08939;

            netPtr->layerPtr[2]->neuronPtr[17]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[17]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[17]->w[0] = (float)0.516;
            netPtr->layerPtr[2]->neuronPtr[17]->w[1] = (float)0.09951;
            netPtr->layerPtr[2]->neuronPtr[17]->w[2] = (float)0.1889;
            netPtr->layerPtr[2]->neuronPtr[17]->w[3] = (float)0.01083;
            netPtr->layerPtr[2]->neuronPtr[17]->w[4] = (float)0.039;
            netPtr->layerPtr[2]->neuronPtr[17]->w[5] = (float)-0.3263;
```

```
            netPtr->layerPtr[2]->neuronPtr[17]->w[6] = (float)-0.1465;
            netPtr->layerPtr[2]->neuronPtr[17]->w[7] = (float)-0.1713;
            netPtr->layerPtr[2]->neuronPtr[17]->w[8] = (float)0.08295;
            netPtr->layerPtr[2]->neuronPtr[17]->w[9] = (float)0.01477;
            netPtr->layerPtr[2]->neuronPtr[17]->w[10] = (float)0.2464;
            netPtr->layerPtr[2]->neuronPtr[17]->w[11] = (float)0.3441;
            netPtr->layerPtr[2]->neuronPtr[17]->w[12] = (float)0.2438;
            netPtr->layerPtr[2]->neuronPtr[17]->w[13] = (float)-0.417;
            netPtr->layerPtr[2]->neuronPtr[17]->w[14] = (float)0.02903;
            netPtr->layerPtr[2]->neuronPtr[17]->w[15] = (float)-0.4182;
            netPtr->layerPtr[2]->neuronPtr[17]->w[16] = (float)-0.2605;
            netPtr->layerPtr[2]->neuronPtr[17]->w[17] = (float)-0.4196;
            netPtr->layerPtr[2]->neuronPtr[17]->w[18] = (float)-0.04617;
            netPtr->layerPtr[2]->neuronPtr[17]->w[19] = (float)0.2195;
            netPtr->layerPtr[2]->neuronPtr[17]->w[20] = (float)0.3394;
            netPtr->layerPtr[2]->neuronPtr[17]->w[21] = (float)0.2129;
            netPtr->layerPtr[2]->neuronPtr[17]->w[22] = (float)-0.03346;
            netPtr->layerPtr[2]->neuronPtr[17]->w[23] = (float)-0.01022;
            netPtr->layerPtr[2]->neuronPtr[17]->w[24] = (float)0.1143;
            netPtr->layerPtr[2]->neuronPtr[17]->w[25] = (float)-0.1405;
            netPtr->layerPtr[2]->neuronPtr[17]->w[26] = (float)-0.4122;
            netPtr->layerPtr[2]->neuronPtr[17]->w[27] = (float)-0.3986;
            netPtr->layerPtr[2]->neuronPtr[17]->w[28] = (float)-0.1111;
            netPtr->layerPtr[2]->neuronPtr[17]->w[29] = (float)-0.323;
            netPtr->layerPtr[2]->neuronPtr[17]->w[30] = (float)0.5513;
            netPtr->layerPtr[2]->neuronPtr[17]->w[31] = (float)0.3759;

            netPtr->layerPtr[2]->neuronPtr[18]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[18]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[18]->w[0] = (float)-0.4699;
            netPtr->layerPtr[2]->neuronPtr[18]->w[1] = (float)0.1372;
            netPtr->layerPtr[2]->neuronPtr[18]->w[2] = (float)0.1029;
            netPtr->layerPtr[2]->neuronPtr[18]->w[3] = (float)-0.09796;
            netPtr->layerPtr[2]->neuronPtr[18]->w[4] = (float)-0.2644;
            netPtr->layerPtr[2]->neuronPtr[18]->w[5] = (float)0.2714;
            netPtr->layerPtr[2]->neuronPtr[18]->w[6] = (float)-0.1301;
            netPtr->layerPtr[2]->neuronPtr[18]->w[7] = (float)0.2512;
            netPtr->layerPtr[2]->neuronPtr[18]->w[8] = (float)-0.2602;
            netPtr->layerPtr[2]->neuronPtr[18]->w[9] = (float)-0.1265;
            netPtr->layerPtr[2]->neuronPtr[18]->w[10] = (float)0.2749;
            netPtr->layerPtr[2]->neuronPtr[18]->w[11] = (float)-0.346;
            netPtr->layerPtr[2]->neuronPtr[18]->w[12] = (float)-0.3783;
            netPtr->layerPtr[2]->neuronPtr[18]->w[13] = (float)0.2943;
            netPtr->layerPtr[2]->neuronPtr[18]->w[14] = (float)0.1379;
            netPtr->layerPtr[2]->neuronPtr[18]->w[15] = (float)0.4095;
            netPtr->layerPtr[2]->neuronPtr[18]->w[16] = (float)0.1335;
            netPtr->layerPtr[2]->neuronPtr[18]->w[17] = (float)-0.1963;
            netPtr->layerPtr[2]->neuronPtr[18]->w[18] = (float)-0.07667;
            netPtr->layerPtr[2]->neuronPtr[18]->w[19] = (float)0.297;
            netPtr->layerPtr[2]->neuronPtr[18]->w[20] = (float)-0.01047;
            netPtr->layerPtr[2]->neuronPtr[18]->w[21] = (float)-0.3822;
            netPtr->layerPtr[2]->neuronPtr[18]->w[22] = (float)-0.2905;
            netPtr->layerPtr[2]->neuronPtr[18]->w[23] = (float)0.008579;
            netPtr->layerPtr[2]->neuronPtr[18]->w[24] = (float)0.04956;
            netPtr->layerPtr[2]->neuronPtr[18]->w[25] = (float)0.06445;
            netPtr->layerPtr[2]->neuronPtr[18]->w[26] = (float)0.4053;
```

```
            netPtr->layerPtr[2]->neuronPtr[18]->w[27] = (float)-0.3134;
            netPtr->layerPtr[2]->neuronPtr[18]->w[28] = (float)-0.3943;
            netPtr->layerPtr[2]->neuronPtr[18]->w[29] = (float)0.172;
            netPtr->layerPtr[2]->neuronPtr[18]->w[30] = (float)-0.3562;
            netPtr->layerPtr[2]->neuronPtr[18]->w[31] = (float)0.3262;

            netPtr->layerPtr[2]->neuronPtr[19]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[19]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[19]->w[0] = (float)0.3264;
            netPtr->layerPtr[2]->neuronPtr[19]->w[1] = (float)0.1218;
            netPtr->layerPtr[2]->neuronPtr[19]->w[2] = (float)-0.06935;
            netPtr->layerPtr[2]->neuronPtr[19]->w[3] = (float)-0.2469;
            netPtr->layerPtr[2]->neuronPtr[19]->w[4] = (float)-0.1845;
            netPtr->layerPtr[2]->neuronPtr[19]->w[5] = (float)0.3065;
            netPtr->layerPtr[2]->neuronPtr[19]->w[6] = (float)0.3925;
            netPtr->layerPtr[2]->neuronPtr[19]->w[7] = (float)0.1546;
            netPtr->layerPtr[2]->neuronPtr[19]->w[8] = (float)-0.1731;
            netPtr->layerPtr[2]->neuronPtr[19]->w[9] = (float)0.2501;
            netPtr->layerPtr[2]->neuronPtr[19]->w[10] = (float)-0.1681;
            netPtr->layerPtr[2]->neuronPtr[19]->w[11] = (float)0.3552;
            netPtr->layerPtr[2]->neuronPtr[19]->w[12] = (float)0.1563;
            netPtr->layerPtr[2]->neuronPtr[19]->w[13] = (float)0.2855;
            netPtr->layerPtr[2]->neuronPtr[19]->w[14] = (float)-0.4672;
            netPtr->layerPtr[2]->neuronPtr[19]->w[15] = (float)-0.028;
            netPtr->layerPtr[2]->neuronPtr[19]->w[16] = (float)-0.05494;
            netPtr->layerPtr[2]->neuronPtr[19]->w[17] = (float)-0.3091;
            netPtr->layerPtr[2]->neuronPtr[19]->w[18] = (float)-0.1449;
            netPtr->layerPtr[2]->neuronPtr[19]->w[19] = (float)-0.2635;
            netPtr->layerPtr[2]->neuronPtr[19]->w[20] = (float)-0.1565;
            netPtr->layerPtr[2]->neuronPtr[19]->w[21] = (float)0.1971;
            netPtr->layerPtr[2]->neuronPtr[19]->w[22] = (float)-0.3283;
            netPtr->layerPtr[2]->neuronPtr[19]->w[23] = (float)0.1001;
            netPtr->layerPtr[2]->neuronPtr[19]->w[24] = (float)0.2365;
            netPtr->layerPtr[2]->neuronPtr[19]->w[25] = (float)0.08314;
            netPtr->layerPtr[2]->neuronPtr[19]->w[26] = (float)0.1888;
            netPtr->layerPtr[2]->neuronPtr[19]->w[27] = (float)0.5071;
            netPtr->layerPtr[2]->neuronPtr[19]->w[28] = (float)-0.2137;
            netPtr->layerPtr[2]->neuronPtr[19]->w[29] = (float)-0.279;
            netPtr->layerPtr[2]->neuronPtr[19]->w[30] = (float)-0.3947;
            netPtr->layerPtr[2]->neuronPtr[19]->w[31] = (float)-0.04968;

            netPtr->layerPtr[2]->neuronPtr[20]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[20]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[20]->w[0] = (float)-0.3396;
            netPtr->layerPtr[2]->neuronPtr[20]->w[1] = (float)0.08571;
            netPtr->layerPtr[2]->neuronPtr[20]->w[2] = (float)-0.4872;
            netPtr->layerPtr[2]->neuronPtr[20]->w[3] = (float)0.3137;
            netPtr->layerPtr[2]->neuronPtr[20]->w[4] = (float)-0.2059;
            netPtr->layerPtr[2]->neuronPtr[20]->w[5] = (float)0.4377;
            netPtr->layerPtr[2]->neuronPtr[20]->w[6] = (float)-0.03809;
            netPtr->layerPtr[2]->neuronPtr[20]->w[7] = (float)-0.0009942;
            netPtr->layerPtr[2]->neuronPtr[20]->w[8] = (float)0.4874;
            netPtr->layerPtr[2]->neuronPtr[20]->w[9] = (float)0.09927;
            netPtr->layerPtr[2]->neuronPtr[20]->w[10] = (float)-0.37;
            netPtr->layerPtr[2]->neuronPtr[20]->w[11] = (float)0.1749;
```

```
            netPtr->layerPtr[2]->neuronPtr[20]->w[12] = (float)0.229;
            netPtr->layerPtr[2]->neuronPtr[20]->w[13] = (float)0.5333;
            netPtr->layerPtr[2]->neuronPtr[20]->w[14] = (float)0.361;
            netPtr->layerPtr[2]->neuronPtr[20]->w[15] = (float)0.04623;
            netPtr->layerPtr[2]->neuronPtr[20]->w[16] = (float)0.2387;
            netPtr->layerPtr[2]->neuronPtr[20]->w[17] = (float)0.09648;
            netPtr->layerPtr[2]->neuronPtr[20]->w[18] = (float)-0.08548;
            netPtr->layerPtr[2]->neuronPtr[20]->w[19] = (float)-0.05111;
            netPtr->layerPtr[2]->neuronPtr[20]->w[20] = (float)-0.3677;
            netPtr->layerPtr[2]->neuronPtr[20]->w[21] = (float)-0.3399;
            netPtr->layerPtr[2]->neuronPtr[20]->w[22] = (float)-0.2034;
            netPtr->layerPtr[2]->neuronPtr[20]->w[23] = (float)-0.2965;
            netPtr->layerPtr[2]->neuronPtr[20]->w[24] = (float)0.08242;
            netPtr->layerPtr[2]->neuronPtr[20]->w[25] = (float)-0.1761;
            netPtr->layerPtr[2]->neuronPtr[20]->w[26] = (float)-0.4103;
            netPtr->layerPtr[2]->neuronPtr[20]->w[27] = (float)-0.1333;
            netPtr->layerPtr[2]->neuronPtr[20]->w[28] = (float)-0.3362;
            netPtr->layerPtr[2]->neuronPtr[20]->w[29] = (float)-0.09917;
            netPtr->layerPtr[2]->neuronPtr[20]->w[30] = (float)0.3089;
            netPtr->layerPtr[2]->neuronPtr[20]->w[31] = (float)-0.2977;

            netPtr->layerPtr[2]->neuronPtr[21]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[21]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[21]->w[0] = (float)0.00253;
            netPtr->layerPtr[2]->neuronPtr[21]->w[1] = (float)-0.04939;
            netPtr->layerPtr[2]->neuronPtr[21]->w[2] = (float)-0.235;
            netPtr->layerPtr[2]->neuronPtr[21]->w[3] = (float)0.05813;
            netPtr->layerPtr[2]->neuronPtr[21]->w[4] = (float)-0.1403;
            netPtr->layerPtr[2]->neuronPtr[21]->w[5] = (float)-0.2252;
            netPtr->layerPtr[2]->neuronPtr[21]->w[6] = (float)-0.3458;
            netPtr->layerPtr[2]->neuronPtr[21]->w[7] = (float)-0.2997;
            netPtr->layerPtr[2]->neuronPtr[21]->w[8] = (float)-0.3703;
            netPtr->layerPtr[2]->neuronPtr[21]->w[9] = (float)0.2066;
            netPtr->layerPtr[2]->neuronPtr[21]->w[10] = (float)-0.05928;
            netPtr->layerPtr[2]->neuronPtr[21]->w[11] = (float)-0.06395;
            netPtr->layerPtr[2]->neuronPtr[21]->w[12] = (float)0.2596;
            netPtr->layerPtr[2]->neuronPtr[21]->w[13] = (float)-0.2802;
            netPtr->layerPtr[2]->neuronPtr[21]->w[14] = (float)-0.1615;
            netPtr->layerPtr[2]->neuronPtr[21]->w[15] = (float)-0.0159;
            netPtr->layerPtr[2]->neuronPtr[21]->w[16] = (float)-0.02831;
            netPtr->layerPtr[2]->neuronPtr[21]->w[17] = (float)0.029;
            netPtr->layerPtr[2]->neuronPtr[21]->w[18] = (float)-0.07539;
            netPtr->layerPtr[2]->neuronPtr[21]->w[19] = (float)0.3596;
            netPtr->layerPtr[2]->neuronPtr[21]->w[20] = (float)0.5138;
            netPtr->layerPtr[2]->neuronPtr[21]->w[21] = (float)0.3748;
            netPtr->layerPtr[2]->neuronPtr[21]->w[22] = (float)0.3082;
            netPtr->layerPtr[2]->neuronPtr[21]->w[23] = (float)0.1166;
            netPtr->layerPtr[2]->neuronPtr[21]->w[24] = (float)-0.1693;
            netPtr->layerPtr[2]->neuronPtr[21]->w[25] = (float)0.1183;
            netPtr->layerPtr[2]->neuronPtr[21]->w[26] = (float)-0.1939;
            netPtr->layerPtr[2]->neuronPtr[21]->w[27] = (float)-0.0217;
            netPtr->layerPtr[2]->neuronPtr[21]->w[28] = (float)0.2288;
            netPtr->layerPtr[2]->neuronPtr[21]->w[29] = (float)0.4037;
            netPtr->layerPtr[2]->neuronPtr[21]->w[30] = (float)0.488;
            netPtr->layerPtr[2]->neuronPtr[21]->w[31] = (float)-0.373;
```

```
            netPtr->layerPtr[2]->neuronPtr[22]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[22]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[22]->w[0] = (float)-0.1933;
            netPtr->layerPtr[2]->neuronPtr[22]->w[1] = (float)0.4039;
            netPtr->layerPtr[2]->neuronPtr[22]->w[2] = (float)0.04174;
            netPtr->layerPtr[2]->neuronPtr[22]->w[3] = (float)-0.2977;
            netPtr->layerPtr[2]->neuronPtr[22]->w[4] = (float)0.2226;
            netPtr->layerPtr[2]->neuronPtr[22]->w[5] = (float)-0.3972;
            netPtr->layerPtr[2]->neuronPtr[22]->w[6] = (float)-0.0442;
            netPtr->layerPtr[2]->neuronPtr[22]->w[7] = (float)0.3561;
            netPtr->layerPtr[2]->neuronPtr[22]->w[8] = (float)0.3264;
            netPtr->layerPtr[2]->neuronPtr[22]->w[9] = (float)0.2983;
            netPtr->layerPtr[2]->neuronPtr[22]->w[10] = (float)0.355;
            netPtr->layerPtr[2]->neuronPtr[22]->w[11] = (float)-0.2022;
            netPtr->layerPtr[2]->neuronPtr[22]->w[12] = (float)-0.1024;
            netPtr->layerPtr[2]->neuronPtr[22]->w[13] = (float)0.2011;
            netPtr->layerPtr[2]->neuronPtr[22]->w[14] = (float)0.0407;
            netPtr->layerPtr[2]->neuronPtr[22]->w[15] = (float)-0.2201;
            netPtr->layerPtr[2]->neuronPtr[22]->w[16] = (float)-0.4167;
            netPtr->layerPtr[2]->neuronPtr[22]->w[17] = (float)0.432;
            netPtr->layerPtr[2]->neuronPtr[22]->w[18] = (float)-0.3286;
            netPtr->layerPtr[2]->neuronPtr[22]->w[19] = (float)-0.0525;
            netPtr->layerPtr[2]->neuronPtr[22]->w[20] = (float)-0.4294;
            netPtr->layerPtr[2]->neuronPtr[22]->w[21] = (float)-0.1638;
            netPtr->layerPtr[2]->neuronPtr[22]->w[22] = (float)-0.3525;
            netPtr->layerPtr[2]->neuronPtr[22]->w[23] = (float)0.4783;
            netPtr->layerPtr[2]->neuronPtr[22]->w[24] = (float)-0.3454;
            netPtr->layerPtr[2]->neuronPtr[22]->w[25] = (float)-0.3255;
            netPtr->layerPtr[2]->neuronPtr[22]->w[26] = (float)0.482;
            netPtr->layerPtr[2]->neuronPtr[22]->w[27] = (float)0.4483;
            netPtr->layerPtr[2]->neuronPtr[22]->w[28] = (float)-0.134;
            netPtr->layerPtr[2]->neuronPtr[22]->w[29] = (float)-0.2467;
            netPtr->layerPtr[2]->neuronPtr[22]->w[30] = (float)-0.3447;
            netPtr->layerPtr[2]->neuronPtr[22]->w[31] = (float)0.03224;

            netPtr->layerPtr[2]->neuronPtr[23]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[23]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[23]->w[0] = (float)-0.209;
            netPtr->layerPtr[2]->neuronPtr[23]->w[1] = (float)0.5087;
            netPtr->layerPtr[2]->neuronPtr[23]->w[2] = (float)0.04873;
            netPtr->layerPtr[2]->neuronPtr[23]->w[3] = (float)0.3322;
            netPtr->layerPtr[2]->neuronPtr[23]->w[4] = (float)-0.08582;
            netPtr->layerPtr[2]->neuronPtr[23]->w[5] = (float)-0.2017;
            netPtr->layerPtr[2]->neuronPtr[23]->w[6] = (float)-0.4945;
            netPtr->layerPtr[2]->neuronPtr[23]->w[7] = (float)0.00568;
            netPtr->layerPtr[2]->neuronPtr[23]->w[8] = (float)-0.3671;
            netPtr->layerPtr[2]->neuronPtr[23]->w[9] = (float)-0.1641;
            netPtr->layerPtr[2]->neuronPtr[23]->w[10] = (float)0.04256;
            netPtr->layerPtr[2]->neuronPtr[23]->w[11] = (float)0.3561;
            netPtr->layerPtr[2]->neuronPtr[23]->w[12] = (float)-0.3865;
            netPtr->layerPtr[2]->neuronPtr[23]->w[13] = (float)0.34;
            netPtr->layerPtr[2]->neuronPtr[23]->w[14] = (float)-0.05907;
            netPtr->layerPtr[2]->neuronPtr[23]->w[15] = (float)-0.272;
            netPtr->layerPtr[2]->neuronPtr[23]->w[16] = (float)0.3398;
            netPtr->layerPtr[2]->neuronPtr[23]->w[17] = (float)0.239;
```

```
            netPtr->layerPtr[2]->neuronPtr[23]->w[18] = (float)-0.3633;
            netPtr->layerPtr[2]->neuronPtr[23]->w[19] = (float)0.1283;
            netPtr->layerPtr[2]->neuronPtr[23]->w[20] = (float)0.4396;
            netPtr->layerPtr[2]->neuronPtr[23]->w[21] = (float)0.02338;
            netPtr->layerPtr[2]->neuronPtr[23]->w[22] = (float)-0.3822;
            netPtr->layerPtr[2]->neuronPtr[23]->w[23] = (float)-0.3202;
            netPtr->layerPtr[2]->neuronPtr[23]->w[24] = (float)0.2648;
            netPtr->layerPtr[2]->neuronPtr[23]->w[25] = (float)-0.4325;
            netPtr->layerPtr[2]->neuronPtr[23]->w[26] = (float)0.3915;
            netPtr->layerPtr[2]->neuronPtr[23]->w[27] = (float)0.301;
            netPtr->layerPtr[2]->neuronPtr[23]->w[28] = (float)0.02508;
            netPtr->layerPtr[2]->neuronPtr[23]->w[29] = (float)-0.2994;
            netPtr->layerPtr[2]->neuronPtr[23]->w[30] = (float)-4.849e-005;
            netPtr->layerPtr[2]->neuronPtr[23]->w[31] = (float)-0.1783;

            netPtr->layerPtr[2]->neuronPtr[24]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[24]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[24]->w[0] = (float)0.4908;
            netPtr->layerPtr[2]->neuronPtr[24]->w[1] = (float)-0.2051;
            netPtr->layerPtr[2]->neuronPtr[24]->w[2] = (float)0.232;
            netPtr->layerPtr[2]->neuronPtr[24]->w[3] = (float)-0.1306;
            netPtr->layerPtr[2]->neuronPtr[24]->w[4] = (float)0.1785;
            netPtr->layerPtr[2]->neuronPtr[24]->w[5] = (float)-0.04346;
            netPtr->layerPtr[2]->neuronPtr[24]->w[6] = (float)0.1059;
            netPtr->layerPtr[2]->neuronPtr[24]->w[7] = (float)0.03068;
            netPtr->layerPtr[2]->neuronPtr[24]->w[8] = (float)0.3129;
            netPtr->layerPtr[2]->neuronPtr[24]->w[9] = (float)-0.06631;
            netPtr->layerPtr[2]->neuronPtr[24]->w[10] = (float)0.4921;
            netPtr->layerPtr[2]->neuronPtr[24]->w[11] = (float)-0.1096;
            netPtr->layerPtr[2]->neuronPtr[24]->w[12] = (float)0.177;
            netPtr->layerPtr[2]->neuronPtr[24]->w[13] = (float)0.3056;
            netPtr->layerPtr[2]->neuronPtr[24]->w[14] = (float)-0.09941;
            netPtr->layerPtr[2]->neuronPtr[24]->w[15] = (float)0.303;
            netPtr->layerPtr[2]->neuronPtr[24]->w[16] = (float)-0.1313;
            netPtr->layerPtr[2]->neuronPtr[24]->w[17] = (float)0.2192;
            netPtr->layerPtr[2]->neuronPtr[24]->w[18] = (float)-0.1876;
            netPtr->layerPtr[2]->neuronPtr[24]->w[19] = (float)0.1221;
            netPtr->layerPtr[2]->neuronPtr[24]->w[20] = (float)-0.02945;
            netPtr->layerPtr[2]->neuronPtr[24]->w[21] = (float)-0.2274;
            netPtr->layerPtr[2]->neuronPtr[24]->w[22] = (float)-0.4117;
            netPtr->layerPtr[2]->neuronPtr[24]->w[23] = (float)0.1586;
            netPtr->layerPtr[2]->neuronPtr[24]->w[24] = (float)0.1953;
            netPtr->layerPtr[2]->neuronPtr[24]->w[25] = (float)-0.0457;
            netPtr->layerPtr[2]->neuronPtr[24]->w[26] = (float)0.1008;
            netPtr->layerPtr[2]->neuronPtr[24]->w[27] = (float)-0.07812;
            netPtr->layerPtr[2]->neuronPtr[24]->w[28] = (float)0.2051;
            netPtr->layerPtr[2]->neuronPtr[24]->w[29] = (float)0.3461;
            netPtr->layerPtr[2]->neuronPtr[24]->w[30] = (float)-0.1298;
            netPtr->layerPtr[2]->neuronPtr[24]->w[31] = (float)-0.0004536;

            netPtr->layerPtr[2]->neuronPtr[25]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[25]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[25]->w[0] = (float)-0.2864;
            netPtr->layerPtr[2]->neuronPtr[25]->w[1] = (float)0.07102;
            netPtr->layerPtr[2]->neuronPtr[25]->w[2] = (float)-0.3933;
```

```
            netPtr->layerPtr[2]->neuronPtr[25]->w[3] = (float)0.362;
            netPtr->layerPtr[2]->neuronPtr[25]->w[4] = (float)-0.3629;
            netPtr->layerPtr[2]->neuronPtr[25]->w[5] = (float)0.2902;
            netPtr->layerPtr[2]->neuronPtr[25]->w[6] = (float)-0.3754;
            netPtr->layerPtr[2]->neuronPtr[25]->w[7] = (float)0.2182;
            netPtr->layerPtr[2]->neuronPtr[25]->w[8] = (float)0.3269;
            netPtr->layerPtr[2]->neuronPtr[25]->w[9] = (float)0.01763;
            netPtr->layerPtr[2]->neuronPtr[25]->w[10] = (float)0.251;
            netPtr->layerPtr[2]->neuronPtr[25]->w[11] = (float)-0.32;
            netPtr->layerPtr[2]->neuronPtr[25]->w[12] = (float)-0.4897;
            netPtr->layerPtr[2]->neuronPtr[25]->w[13] = (float)-0.1982;
            netPtr->layerPtr[2]->neuronPtr[25]->w[14] = (float)-0.2491;
            netPtr->layerPtr[2]->neuronPtr[25]->w[15] = (float)-0.003926;
            netPtr->layerPtr[2]->neuronPtr[25]->w[16] = (float)0.5217;
            netPtr->layerPtr[2]->neuronPtr[25]->w[17] = (float)0.224;
            netPtr->layerPtr[2]->neuronPtr[25]->w[18] = (float)-0.4636;
            netPtr->layerPtr[2]->neuronPtr[25]->w[19] = (float)-0.2339;
            netPtr->layerPtr[2]->neuronPtr[25]->w[20] = (float)-0.4118;
            netPtr->layerPtr[2]->neuronPtr[25]->w[21] = (float)0.2943;
            netPtr->layerPtr[2]->neuronPtr[25]->w[22] = (float)0.3618;
            netPtr->layerPtr[2]->neuronPtr[25]->w[23] = (float)0.01763;
            netPtr->layerPtr[2]->neuronPtr[25]->w[24] = (float)0.2285;
            netPtr->layerPtr[2]->neuronPtr[25]->w[25] = (float)-0.05333;
            netPtr->layerPtr[2]->neuronPtr[25]->w[26] = (float)0.1896;
            netPtr->layerPtr[2]->neuronPtr[25]->w[27] = (float)-0.236;
            netPtr->layerPtr[2]->neuronPtr[25]->w[28] = (float)-0.4025;
            netPtr->layerPtr[2]->neuronPtr[25]->w[29] = (float)-0.363;
            netPtr->layerPtr[2]->neuronPtr[25]->w[30] = (float)0.1885;
            netPtr->layerPtr[2]->neuronPtr[25]->w[31] = (float)-0.1364;

            netPtr->layerPtr[2]->neuronPtr[26]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[26]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[26]->w[0] = (float)0.03845;
            netPtr->layerPtr[2]->neuronPtr[26]->w[1] = (float)0.3757;
            netPtr->layerPtr[2]->neuronPtr[26]->w[2] = (float)-0.2651;
            netPtr->layerPtr[2]->neuronPtr[26]->w[3] = (float)-0.2886;
            netPtr->layerPtr[2]->neuronPtr[26]->w[4] = (float)0.3132;
            netPtr->layerPtr[2]->neuronPtr[26]->w[5] = (float)-0.3036;
            netPtr->layerPtr[2]->neuronPtr[26]->w[6] = (float)-0.2449;
            netPtr->layerPtr[2]->neuronPtr[26]->w[7] = (float)-0.2813;
            netPtr->layerPtr[2]->neuronPtr[26]->w[8] = (float)-0.4967;
            netPtr->layerPtr[2]->neuronPtr[26]->w[9] = (float)0.2449;
            netPtr->layerPtr[2]->neuronPtr[26]->w[10] = (float)0.06811;
            netPtr->layerPtr[2]->neuronPtr[26]->w[11] = (float)0.3302;
            netPtr->layerPtr[2]->neuronPtr[26]->w[12] = (float)-0.2085;
            netPtr->layerPtr[2]->neuronPtr[26]->w[13] = (float)0.3192;
            netPtr->layerPtr[2]->neuronPtr[26]->w[14] = (float)0.2961;
            netPtr->layerPtr[2]->neuronPtr[26]->w[15] = (float)0.1588;
            netPtr->layerPtr[2]->neuronPtr[26]->w[16] = (float)0.09482;
            netPtr->layerPtr[2]->neuronPtr[26]->w[17] = (float)0.1657;
            netPtr->layerPtr[2]->neuronPtr[26]->w[18] = (float)-0.4119;
            netPtr->layerPtr[2]->neuronPtr[26]->w[19] = (float)-0.3688;
            netPtr->layerPtr[2]->neuronPtr[26]->w[20] = (float)0.2636;
            netPtr->layerPtr[2]->neuronPtr[26]->w[21] = (float)-0.4052;
            netPtr->layerPtr[2]->neuronPtr[26]->w[22] = (float)0.1284;
            netPtr->layerPtr[2]->neuronPtr[26]->w[23] = (float)-0.0151;
```

```
            netPtr->layerPtr[2]->neuronPtr[26]->w[24] = (float)0.4186;
            netPtr->layerPtr[2]->neuronPtr[26]->w[25] = (float)0.1046;
            netPtr->layerPtr[2]->neuronPtr[26]->w[26] = (float)0.3426;
            netPtr->layerPtr[2]->neuronPtr[26]->w[27] = (float)0.2304;
            netPtr->layerPtr[2]->neuronPtr[26]->w[28] = (float)-0.1166;
            netPtr->layerPtr[2]->neuronPtr[26]->w[29] = (float)0.01401;
            netPtr->layerPtr[2]->neuronPtr[26]->w[30] = (float)0.2112;
            netPtr->layerPtr[2]->neuronPtr[26]->w[31] = (float)0.4428;

            netPtr->layerPtr[2]->neuronPtr[27]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[27]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[27]->w[0] = (float)0.1896;
            netPtr->layerPtr[2]->neuronPtr[27]->w[1] = (float)-0.05812;
            netPtr->layerPtr[2]->neuronPtr[27]->w[2] = (float)-0.1943;
            netPtr->layerPtr[2]->neuronPtr[27]->w[3] = (float)-0.3913;
            netPtr->layerPtr[2]->neuronPtr[27]->w[4] = (float)0.2709;
            netPtr->layerPtr[2]->neuronPtr[27]->w[5] = (float)-0.3175;
            netPtr->layerPtr[2]->neuronPtr[27]->w[6] = (float)-0.4383;
            netPtr->layerPtr[2]->neuronPtr[27]->w[7] = (float)0.09571;
            netPtr->layerPtr[2]->neuronPtr[27]->w[8] = (float)-0.3891;
            netPtr->layerPtr[2]->neuronPtr[27]->w[9] = (float)-0.2799;
            netPtr->layerPtr[2]->neuronPtr[27]->w[10] = (float)0.3661;
            netPtr->layerPtr[2]->neuronPtr[27]->w[11] = (float)0.02515;
            netPtr->layerPtr[2]->neuronPtr[27]->w[12] = (float)0.3624;
            netPtr->layerPtr[2]->neuronPtr[27]->w[13] = (float)-0.2772;
            netPtr->layerPtr[2]->neuronPtr[27]->w[14] = (float)0.2443;
            netPtr->layerPtr[2]->neuronPtr[27]->w[15] = (float)0.186;
            netPtr->layerPtr[2]->neuronPtr[27]->w[16] = (float)-0.2699;
            netPtr->layerPtr[2]->neuronPtr[27]->w[17] = (float)0.4213;
            netPtr->layerPtr[2]->neuronPtr[27]->w[18] = (float)-0.1603;
            netPtr->layerPtr[2]->neuronPtr[27]->w[19] = (float)-0.2034;
            netPtr->layerPtr[2]->neuronPtr[27]->w[20] = (float)0.2233;
            netPtr->layerPtr[2]->neuronPtr[27]->w[21] = (float)-0.2577;
            netPtr->layerPtr[2]->neuronPtr[27]->w[22] = (float)-0.1096;
            netPtr->layerPtr[2]->neuronPtr[27]->w[23] = (float)0.4351;
            netPtr->layerPtr[2]->neuronPtr[27]->w[24] = (float)-0.4596;
            netPtr->layerPtr[2]->neuronPtr[27]->w[25] = (float)0.2139;
            netPtr->layerPtr[2]->neuronPtr[27]->w[26] = (float)-0.3922;
            netPtr->layerPtr[2]->neuronPtr[27]->w[27] = (float)-0.325;
            netPtr->layerPtr[2]->neuronPtr[27]->w[28] = (float)-0.4127;
            netPtr->layerPtr[2]->neuronPtr[27]->w[29] = (float)-0.4487;
            netPtr->layerPtr[2]->neuronPtr[27]->w[30] = (float)-0.3498;
            netPtr->layerPtr[2]->neuronPtr[27]->w[31] = (float)0.4346;

            netPtr->layerPtr[2]->neuronPtr[28]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[28]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[28]->w[0] = (float)0.001154;
            netPtr->layerPtr[2]->neuronPtr[28]->w[1] = (float)-0.2838;
            netPtr->layerPtr[2]->neuronPtr[28]->w[2] = (float)-0.1978;
            netPtr->layerPtr[2]->neuronPtr[28]->w[3] = (float)-0.3473;
            netPtr->layerPtr[2]->neuronPtr[28]->w[4] = (float)-0.04779;
            netPtr->layerPtr[2]->neuronPtr[28]->w[5] = (float)0.2392;
            netPtr->layerPtr[2]->neuronPtr[28]->w[6] = (float)-0.4102;
            netPtr->layerPtr[2]->neuronPtr[28]->w[7] = (float)-0.2527;
            netPtr->layerPtr[2]->neuronPtr[28]->w[8] = (float)-0.4199;
```

```
            netPtr->layerPtr[2]->neuronPtr[28]->w[9] = (float)-0.1909;
            netPtr->layerPtr[2]->neuronPtr[28]->w[10] = (float)-0.3369;
            netPtr->layerPtr[2]->neuronPtr[28]->w[11] = (float)-0.2286;
            netPtr->layerPtr[2]->neuronPtr[28]->w[12] = (float)-0.01058;
            netPtr->layerPtr[2]->neuronPtr[28]->w[13] = (float)-0.3201;
            netPtr->layerPtr[2]->neuronPtr[28]->w[14] = (float)0.004729;
            netPtr->layerPtr[2]->neuronPtr[28]->w[15] = (float)-0.1782;
            netPtr->layerPtr[2]->neuronPtr[28]->w[16] = (float)0.2391;
            netPtr->layerPtr[2]->neuronPtr[28]->w[17] = (float)0.1256;
            netPtr->layerPtr[2]->neuronPtr[28]->w[18] = (float)0.08646;
            netPtr->layerPtr[2]->neuronPtr[28]->w[19] = (float)-0.463;
            netPtr->layerPtr[2]->neuronPtr[28]->w[20] = (float)0.05574;
            netPtr->layerPtr[2]->neuronPtr[28]->w[21] = (float)-0.3668;
            netPtr->layerPtr[2]->neuronPtr[28]->w[22] = (float)-0.4598;
            netPtr->layerPtr[2]->neuronPtr[28]->w[23] = (float)-0.1011;
            netPtr->layerPtr[2]->neuronPtr[28]->w[24] = (float)0.09243;
            netPtr->layerPtr[2]->neuronPtr[28]->w[25] = (float)-0.2087;
            netPtr->layerPtr[2]->neuronPtr[28]->w[26] = (float)-0.2298;
            netPtr->layerPtr[2]->neuronPtr[28]->w[27] = (float)0.2829;
            netPtr->layerPtr[2]->neuronPtr[28]->w[28] = (float)0.3317;
            netPtr->layerPtr[2]->neuronPtr[28]->w[29] = (float)-0.1806;
            netPtr->layerPtr[2]->neuronPtr[28]->w[30] = (float)0.02904;
            netPtr->layerPtr[2]->neuronPtr[28]->w[31] = (float)-0.1201;

            netPtr->layerPtr[2]->neuronPtr[29]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[29]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[29]->w[0] = (float)-0.4616;
            netPtr->layerPtr[2]->neuronPtr[29]->w[1] = (float)0.07346;
            netPtr->layerPtr[2]->neuronPtr[29]->w[2] = (float)0.03198;
            netPtr->layerPtr[2]->neuronPtr[29]->w[3] = (float)0.09652;
            netPtr->layerPtr[2]->neuronPtr[29]->w[4] = (float)0.06186;
            netPtr->layerPtr[2]->neuronPtr[29]->w[5] = (float)0.3755;
            netPtr->layerPtr[2]->neuronPtr[29]->w[6] = (float)-0.2614;
            netPtr->layerPtr[2]->neuronPtr[29]->w[7] = (float)-0.2561;
            netPtr->layerPtr[2]->neuronPtr[29]->w[8] = (float)-0.06507;
            netPtr->layerPtr[2]->neuronPtr[29]->w[9] = (float)-0.309;
            netPtr->layerPtr[2]->neuronPtr[29]->w[10] = (float)-0.2673;
            netPtr->layerPtr[2]->neuronPtr[29]->w[11] = (float)0.04937;
            netPtr->layerPtr[2]->neuronPtr[29]->w[12] = (float)-0.2094;
            netPtr->layerPtr[2]->neuronPtr[29]->w[13] = (float)0.1713;
            netPtr->layerPtr[2]->neuronPtr[29]->w[14] = (float)-0.1198;
            netPtr->layerPtr[2]->neuronPtr[29]->w[15] = (float)-0.1398;
            netPtr->layerPtr[2]->neuronPtr[29]->w[16] = (float)-0.369;
            netPtr->layerPtr[2]->neuronPtr[29]->w[17] = (float)-0.2978;
            netPtr->layerPtr[2]->neuronPtr[29]->w[18] = (float)0.3009;
            netPtr->layerPtr[2]->neuronPtr[29]->w[19] = (float)-0.3487;
            netPtr->layerPtr[2]->neuronPtr[29]->w[20] = (float)0.04838;
            netPtr->layerPtr[2]->neuronPtr[29]->w[21] = (float)0.3921;
            netPtr->layerPtr[2]->neuronPtr[29]->w[22] = (float)0.08523;
            netPtr->layerPtr[2]->neuronPtr[29]->w[23] = (float)0.5519;
            netPtr->layerPtr[2]->neuronPtr[29]->w[24] = (float)-0.02951;
            netPtr->layerPtr[2]->neuronPtr[29]->w[25] = (float)0.02243;
            netPtr->layerPtr[2]->neuronPtr[29]->w[26] = (float)-0.002219;
            netPtr->layerPtr[2]->neuronPtr[29]->w[27] = (float)0.495;
            netPtr->layerPtr[2]->neuronPtr[29]->w[28] = (float)0.1351;
            netPtr->layerPtr[2]->neuronPtr[29]->w[29] = (float)0.2521;
```

```
            netPtr->layerPtr[2]->neuronPtr[29]->w[30] = (float)-0.2718;
            netPtr->layerPtr[2]->neuronPtr[29]->w[31] = (float)-0.367;

            netPtr->layerPtr[2]->neuronPtr[30]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[30]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[30]->w[0] = (float)-0.2339;
            netPtr->layerPtr[2]->neuronPtr[30]->w[1] = (float)-0.1685;
            netPtr->layerPtr[2]->neuronPtr[30]->w[2] = (float)0.004084;
            netPtr->layerPtr[2]->neuronPtr[30]->w[3] = (float)0.1485;
            netPtr->layerPtr[2]->neuronPtr[30]->w[4] = (float)0.2207;
            netPtr->layerPtr[2]->neuronPtr[30]->w[5] = (float)0.01612;
            netPtr->layerPtr[2]->neuronPtr[30]->w[6] = (float)0.09334;
            netPtr->layerPtr[2]->neuronPtr[30]->w[7] = (float)-0.1793;
            netPtr->layerPtr[2]->neuronPtr[30]->w[8] = (float)0.2124;
            netPtr->layerPtr[2]->neuronPtr[30]->w[9] = (float)-0.2779;
            netPtr->layerPtr[2]->neuronPtr[30]->w[10] = (float)-0.2239;
            netPtr->layerPtr[2]->neuronPtr[30]->w[11] = (float)-0.02846;
            netPtr->layerPtr[2]->neuronPtr[30]->w[12] = (float)0.4076;
            netPtr->layerPtr[2]->neuronPtr[30]->w[13] = (float)-0.1629;
            netPtr->layerPtr[2]->neuronPtr[30]->w[14] = (float)0.06712;
            netPtr->layerPtr[2]->neuronPtr[30]->w[15] = (float)-0.3981;
            netPtr->layerPtr[2]->neuronPtr[30]->w[16] = (float)0.1527;
            netPtr->layerPtr[2]->neuronPtr[30]->w[17] = (float)-0.3182;
            netPtr->layerPtr[2]->neuronPtr[30]->w[18] = (float)0.1177;
            netPtr->layerPtr[2]->neuronPtr[30]->w[19] = (float)-0.2526;
            netPtr->layerPtr[2]->neuronPtr[30]->w[20] = (float)-0.4242;
            netPtr->layerPtr[2]->neuronPtr[30]->w[21] = (float)-0.05781;
            netPtr->layerPtr[2]->neuronPtr[30]->w[22] = (float)-0.06236;
            netPtr->layerPtr[2]->neuronPtr[30]->w[23] = (float)0.1477;
            netPtr->layerPtr[2]->neuronPtr[30]->w[24] = (float)-0.04314;
            netPtr->layerPtr[2]->neuronPtr[30]->w[25] = (float)0.01378;
            netPtr->layerPtr[2]->neuronPtr[30]->w[26] = (float)0.04439;
            netPtr->layerPtr[2]->neuronPtr[30]->w[27] = (float)0.5485;
            netPtr->layerPtr[2]->neuronPtr[30]->w[28] = (float)0.5092;
            netPtr->layerPtr[2]->neuronPtr[30]->w[29] = (float)-0.3588;
            netPtr->layerPtr[2]->neuronPtr[30]->w[30] = (float)0.3503;
            netPtr->layerPtr[2]->neuronPtr[30]->w[31] = (float)0.2516;

            netPtr->layerPtr[2]->neuronPtr[31]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[2]->neuronPtr[31]->threshold = (float)0;
            netPtr->layerPtr[2]->neuronPtr[31]->w[0] = (float)-0.03617;
            netPtr->layerPtr[2]->neuronPtr[31]->w[1] = (float)-0.06226;
            netPtr->layerPtr[2]->neuronPtr[31]->w[2] = (float)0.03386;
            netPtr->layerPtr[2]->neuronPtr[31]->w[3] = (float)0.0347;
            netPtr->layerPtr[2]->neuronPtr[31]->w[4] = (float)-0.1014;
            netPtr->layerPtr[2]->neuronPtr[31]->w[5] = (float)0.4695;
            netPtr->layerPtr[2]->neuronPtr[31]->w[6] = (float)0.3337;
            netPtr->layerPtr[2]->neuronPtr[31]->w[7] = (float)0.1582;
            netPtr->layerPtr[2]->neuronPtr[31]->w[8] = (float)0.04467;
            netPtr->layerPtr[2]->neuronPtr[31]->w[9] = (float)0.06391;
            netPtr->layerPtr[2]->neuronPtr[31]->w[10] = (float)0.08437;
            netPtr->layerPtr[2]->neuronPtr[31]->w[11] = (float)-0.4551;
            netPtr->layerPtr[2]->neuronPtr[31]->w[12] = (float)0.2342;
            netPtr->layerPtr[2]->neuronPtr[31]->w[13] = (float)0.3975;
            netPtr->layerPtr[2]->neuronPtr[31]->w[14] = (float)0.1098;
```

```
            netPtr->layerPtr[2]->neuronPtr[31]->w[15] = (float)0.0796;
            netPtr->layerPtr[2]->neuronPtr[31]->w[16] = (float)-0.241;
            netPtr->layerPtr[2]->neuronPtr[31]->w[17] = (float)0.4275;
            netPtr->layerPtr[2]->neuronPtr[31]->w[18] = (float)0.4034;
            netPtr->layerPtr[2]->neuronPtr[31]->w[19] = (float)-0.4711;
            netPtr->layerPtr[2]->neuronPtr[31]->w[20] = (float)-0.1985;
            netPtr->layerPtr[2]->neuronPtr[31]->w[21] = (float)-0.07986;
            netPtr->layerPtr[2]->neuronPtr[31]->w[22] = (float)0.1346;
            netPtr->layerPtr[2]->neuronPtr[31]->w[23] = (float)-0.02611;
            netPtr->layerPtr[2]->neuronPtr[31]->w[24] = (float)0.131;
            netPtr->layerPtr[2]->neuronPtr[31]->w[25] = (float)-0.1439;
            netPtr->layerPtr[2]->neuronPtr[31]->w[26] = (float)0.3633;
            netPtr->layerPtr[2]->neuronPtr[31]->w[27] = (float)-0.1897;
            netPtr->layerPtr[2]->neuronPtr[31]->w[28] = (float)0.443;
            netPtr->layerPtr[2]->neuronPtr[31]->w[29] = (float)-0.5533;
            netPtr->layerPtr[2]->neuronPtr[31]->w[30] = (float)-0.006624;
            netPtr->layerPtr[2]->neuronPtr[31]->w[31] = (float)-0.4474;

            netPtr->layerPtr[3]->neuronPtr[1]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[3]->neuronPtr[1]->threshold = (float)0;
            netPtr->layerPtr[3]->neuronPtr[1]->w[0] = (float)-0.135;
            netPtr->layerPtr[3]->neuronPtr[1]->w[1] = (float)0.0169;
            netPtr->layerPtr[3]->neuronPtr[1]->w[2] = (float)-0.05379;
            netPtr->layerPtr[3]->neuronPtr[1]->w[3] = (float)-0.1867;
            netPtr->layerPtr[3]->neuronPtr[1]->w[4] = (float)0.2937;
            netPtr->layerPtr[3]->neuronPtr[1]->w[5] = (float)-0.1014;
            netPtr->layerPtr[3]->neuronPtr[1]->w[6] = (float)-0.04214;
            netPtr->layerPtr[3]->neuronPtr[1]->w[7] = (float)-0.02094;
            netPtr->layerPtr[3]->neuronPtr[1]->w[8] = (float)0.06193;
            netPtr->layerPtr[3]->neuronPtr[1]->w[9] = (float)-0.1964;
            netPtr->layerPtr[3]->neuronPtr[1]->w[10] = (float)0.03226;
            netPtr->layerPtr[3]->neuronPtr[1]->w[11] = (float)-0.1023;
            netPtr->layerPtr[3]->neuronPtr[1]->w[12] = (float)-0.01802;
            netPtr->layerPtr[3]->neuronPtr[1]->w[13] = (float)-0.09041;
            netPtr->layerPtr[3]->neuronPtr[1]->w[14] = (float)-0.1015;
            netPtr->layerPtr[3]->neuronPtr[1]->w[15] = (float)-0.3776;
            netPtr->layerPtr[3]->neuronPtr[1]->w[16] = (float)-0.09301;
            netPtr->layerPtr[3]->neuronPtr[1]->w[17] = (float)0.02477;
            netPtr->layerPtr[3]->neuronPtr[1]->w[18] = (float)-0.201;
            netPtr->layerPtr[3]->neuronPtr[1]->w[19] = (float)-0.1509;
            netPtr->layerPtr[3]->neuronPtr[1]->w[20] = (float)-0.05158;
            netPtr->layerPtr[3]->neuronPtr[1]->w[21] = (float)0.477;
            netPtr->layerPtr[3]->neuronPtr[1]->w[22] = (float)-0.009418;
            netPtr->layerPtr[3]->neuronPtr[1]->w[23] = (float)0.0003337;
            netPtr->layerPtr[3]->neuronPtr[1]->w[24] = (float)-0.1629;
            netPtr->layerPtr[3]->neuronPtr[1]->w[25] = (float)-0.008675;
            netPtr->layerPtr[3]->neuronPtr[1]->w[26] = (float)0.09681;
            netPtr->layerPtr[3]->neuronPtr[1]->w[27] = (float)-0.05292;
            netPtr->layerPtr[3]->neuronPtr[1]->w[28] = (float)0.01493;
            netPtr->layerPtr[3]->neuronPtr[1]->w[29] = (float)-0.3651;
            netPtr->layerPtr[3]->neuronPtr[1]->w[30] = (float)0.2807;
            netPtr->layerPtr[3]->neuronPtr[1]->w[31] = (float)-0.2077;

            netPtr->layerPtr[3]->neuronPtr[2]->tf =
(transfer_function_type)1;
            netPtr->layerPtr[3]->neuronPtr[2]->threshold = (float)0;
```

```
        netPtr->layerPtr[3]->neuronPtr[2]->w[0] = (float)0.1233;
        netPtr->layerPtr[3]->neuronPtr[2]->w[1] = (float)0.1148;
        netPtr->layerPtr[3]->neuronPtr[2]->w[2] = (float)0.1015;
        netPtr->layerPtr[3]->neuronPtr[2]->w[3] = (float)0.3154;
        netPtr->layerPtr[3]->neuronPtr[2]->w[4] = (float)-0.3947;
        netPtr->layerPtr[3]->neuronPtr[2]->w[5] = (float)0.2446;
        netPtr->layerPtr[3]->neuronPtr[2]->w[6] = (float)-0.02308;
        netPtr->layerPtr[3]->neuronPtr[2]->w[7] = (float)0.0009731;
        netPtr->layerPtr[3]->neuronPtr[2]->w[8] = (float)-0.0155;
        netPtr->layerPtr[3]->neuronPtr[2]->w[9] = (float)0.02004;
        netPtr->layerPtr[3]->neuronPtr[2]->w[10] = (float)0.0001292;
        netPtr->layerPtr[3]->neuronPtr[2]->w[11] = (float)0.2967;
        netPtr->layerPtr[3]->neuronPtr[2]->w[12] = (float)-0.2933;
        netPtr->layerPtr[3]->neuronPtr[2]->w[13] = (float)-0.4221;
        netPtr->layerPtr[3]->neuronPtr[2]->w[14] = (float)0.1768;
        netPtr->layerPtr[3]->neuronPtr[2]->w[15] = (float)-0.3795;
        netPtr->layerPtr[3]->neuronPtr[2]->w[16] = (float)0.05439;
        netPtr->layerPtr[3]->neuronPtr[2]->w[17] = (float)-0.4557;
        netPtr->layerPtr[3]->neuronPtr[2]->w[18] = (float)-0.1118;
        netPtr->layerPtr[3]->neuronPtr[2]->w[19] = (float)0.1147;
        netPtr->layerPtr[3]->neuronPtr[2]->w[20] = (float)-0.04179;
        netPtr->layerPtr[3]->neuronPtr[2]->w[21] = (float)-0.05012;
        netPtr->layerPtr[3]->neuronPtr[2]->w[22] = (float)0.03362;
        netPtr->layerPtr[3]->neuronPtr[2]->w[23] = (float)0.05308;
        netPtr->layerPtr[3]->neuronPtr[2]->w[24] = (float)-0.08522;
        netPtr->layerPtr[3]->neuronPtr[2]->w[25] = (float)0.02071;
        netPtr->layerPtr[3]->neuronPtr[2]->w[26] = (float)0.1298;
        netPtr->layerPtr[3]->neuronPtr[2]->w[27] = (float)-0.007393;
        netPtr->layerPtr[3]->neuronPtr[2]->w[28] = (float)0.0007947;
        netPtr->layerPtr[3]->neuronPtr[2]->w[29] = (float)0.2094;
        netPtr->layerPtr[3]->neuronPtr[2]->w[30] = (float)0.227;
        netPtr->layerPtr[3]->neuronPtr[2]->w[31] = (float)0.2684;

        netPtr->w0[1] = (float)0.1299;
        netPtr->w1[1] = (float)16.13;
        netPtr->w0[2] = (float)0.00122;
        netPtr->w1[2] = (float)0.0251;

        return 0;

}
```

Function "initialize_network.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

//int initialize_network(Network *netPtr)
//          Copyright 2007 Hugh S. Bruner

int initialize_network(PointertoNetPtr *netPtr)
{
      short int k;
      short int m;
      short int n;
      Network *newnetPtr;


#ifdef VERBOSE
      printf("\nEntering initialize_network");
#endif

//    If the network does not exist (netPtr == 0)
      if (*netPtr == NULL)
      {
//          Allocate memory for the network
            newnetPtr = (Network *)malloc(sizeof(Network));
            if (newnetPtr == NULL)
            {
                  fprintf(stderr,"\nUnable to allocate memory for new network
in function initialize_network.");
                  return -1;
            }

            *netPtr = newnetPtr;
#ifdef VERBOSE
            printf("\nnetPtr = %p",netPtr);
            printf("\nnewnetPtr = %p",newnetPtr);
            printf("\nnetPtr = %p",netPtr);
#endif


//          Set M, N, and K all to zero
            newnetPtr->M = 0;
            newnetPtr->N = 0;
            newnetPtr->K = 0;

//          For each possible input, m
            for (m=0; m <=MMAX; m++)
            {
//                Set input pointer to NULL
                  newnetPtr->xPtr[m] = NULL;
            }
//          End-for

//          For each possible output, n
            for (n=0; n <=NMAX; n++)
            {
```

```
//                  Set pointers to NULL
                    newnetPtr->yPtr[n] = NULL;
//                  All weights to zero
                    newnetPtr->w0[n] = (float)0;
                    newnetPtr->w1[n] = (float)0;
//                  Output values to zero
                    newnetPtr->Y[n] = (float)0;
            }
//          End-for

//          For each possible layer, k
            for (k=0; k<=KMAX; k++)
            {
//                  Set neuron counts to zero
                    newnetPtr->nk[k]=0;
//                  Set pointers to NULL
                    newnetPtr->layerPtr[k] = NULL;
            }
//          End-for

//          Set training data set pointer to NULL
            newnetPtr->trainingdataPtr = NULL;

//          Default setting is to be silent on input errors
            newnetPtr->flag_input_errors = 0;         // which is FALSE

#ifdef VERBOSE
      printf("\n\tNew Network initialized");
#endif
      }
//    End-if on existence of network

      return 0;

}
/*          End of function initialize network
                    */
```

Function “initialize_training_pair.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int initialize_training_pair(TrainingPair *currentPtr)
/*

  This function initializes the training pair structure with the initial
values
  Written by H.S. Bruner
                   1/28/2007
      Copyright 2007 Hugh S. Bruner


  Training pair map:

      input[0] = Utrim
      input[1] = htrim
      input[2] = vpdot_req(t)
      input[3] = alphadot_req(t)

      input[4] = vtrue(t)
      input[5] = alpha(t)
      input[6] = qb(t)
      input[7] = theta(t)
      input[8] = hp(t)
      input[9] = de(t)
      input[10] = pla(t)

      input[11] = vtrue(t-1)
      input[12] = alpha(t-1)
      input[13] = qb(t-1)
      input[14] = theta(t-1)
      input[15] = hp(t-1)
      input[16] = de(t-1)
      input[17] = pla(t-1)

      input[18] = vtrue(t-2)
      input[19] = alpha(t-2)
      input[20] = qb(t-2)
      input[21] = theta(t-2)
      input[22] = hp(t-2)
      input[23] = de(t-2)
      input[24] = pla(t-2)

      input[25] = delta vp_req(t-1)
      input[26] = delta alphadot_req(t-1)

      input[27] = delta vp_req(t-2)
      input[28] = delta alphadot_req(t-2)

      input[29] = delta vp_req(t-3)
      input[30] = delta alphadot_req(t-3)
*/
{
```

```
	short int i;

	printf("\nInitializing training pair");

	currentPtr->input[0] = (float)Utrim;
	currentPtr->input[1] = (float)htrim;
	currentPtr->input[2] = 0.;
	currentPtr->input[3] = 0.;

	currentPtr->input[4] = (float)vtrue;
	currentPtr->input[5] = (float)alpha;
	currentPtr->input[6] = (float)qb;
	currentPtr->input[7] = (float)theta;
	currentPtr->input[8] = (float)h;
	currentPtr->input[9] = (float)de;
	currentPtr->input[10] = (float)pla;

	currentPtr->input[25] = 0.;
	currentPtr->input[26] = 0.;

//	Pass these values down to the "old values" ...

	for (i=4; i<11; i++)
	{
		currentPtr->input[i + 7] = currentPtr->input[i];		// t-
1 values
		currentPtr->input[i + 14] = currentPtr->input[i];		// t-
2 values
	}
	for (i=25; i<27; i++)
	{
		currentPtr->input[i + 2] = currentPtr->input[i];		// t-
1 values
		currentPtr->input[i + 4] = currentPtr->input[i];		// t-
2 values
	}

	return 0;
}
/*			End of Function initialize_training_pair			*/
```

input_excedance_alarm.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Function input_excedance_alarm

            Monitors input to the neural network.  Logs instances were input
values
            exceed the scaling limits for that input neuron.  Check to see if
the
            scaled input goes outside of plus/minus 1

            Copyright 2007 Hugh S. Bruner

*/

int input_excedance_alarm(Network *myNetPtr, FILE *errorLogPtr, const double
*mytimePtr)
{
      int m;                        // counter for marching through Neurons
      int occurences = 0;           // counter of excedances encountered
      float neuronValue;      // output value of neuron in question, scaled
      float neuronWorld;      // output value of neuron in "world" units
      Neuron *neuronPtr;      // pointer to neuron in question


//    If network is set to log errors
//    fprintf(errorLogPtr,"\nChecking inputs");
      if (myNetPtr->flag_input_errors != 0)
      {
//          For all the input neurons
            for (m=1; m<=myNetPtr->layerPtr[1]->M; m++)
            {
//                If any value is outside of plus/minus 1
                  neuronPtr = myNetPtr->layerPtr[1]->neuronPtr[m];
                  neuronValue = neuronPtr->y;
//                neuronWorld = (neuronValue - neuronPtr->w[0])/neuronPtr-
>w[1];
//                fprintf(errorLogPtr,"\nInput[%d] = %f in world coordinates,
%f scaled",
//                      m,neuronWorld,neuronValue);
                  if ((neuronValue < -1.) || (1. <  neuronValue ))
                  {
//                      Count up occurences
                        ++occurences;
//                      Print the appropriate error message
                        neuronWorld = (neuronValue - neuronPtr-
>w[0])/neuronPtr->w[1];
                        fprintf(errorLogPtr,"\nNetwork input excedance:
Input[%d] = %f in world coordinates, %9.3f scaled, at time = %f seconds",
                              m,neuronWorld, neuronValue,*mytimePtr );
                  }
//                End if
            }
```

```
//          End for
      }
//    Endif
      return occurences;
}
//    End of function
```

Function "inverse_xfunction.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int inverse_xfunction(int reset, const float udot_req, const float
alphadot_req, double *ddePtr, double *dplaPtr)
/*    Function to implement the inverse transfer function

              Implements a discretized state space representation of the
inverse transfer function
              for the DEF JR controller.

                  Copyright 2007 Hugh S. Bruner

              There is another set of Ainv, Binv, Cinv, and Dinv that
represent the inverse transfer function.

              Arguement List:
              --------------

                  udot_req = the udot for which a control output is desired
                  alphadot_req = the alphadot for which a control output is
desired
                  *ddePtr = a pointer back to the perturbation elevator
position, measured in radians.  To be added to equilibrium de
                  *dplaPtr = a pointer back to the perturbation PLA, measured
in radians.  To be added to equilibrium PLA.

              Hugh S. Bruner
              December 21, 2006


*/
{
/*
//          Original code...  Don't erase!  Only comment out
======================================

      static double Ainvd[4][4] = {{           1, 3.4255e-017, -6.231e-019,-
1.4212e-016,},
                  { 4.9112e-019,           1,-1.2331e-017,           0,},
                  {   0.0024937,      1.7907,    -0.12307,    0.017038,},
                  { 2.4937e-005,    0.017907,   0.0087693,      1.0002,},
                  };

      static double Binvd[4][2] = {{          -1, 2.2606e-017,},
                  {-2.4556e-019,          -1,},
                  {  -0.0012468,     -56.815,},
                  {-1.2468e-005,    -0.56815,},
                  };

      static double Cinvd[2][4] = {{  0.00017557,     0.14343,   -0.074342,
0.0011995,},
                  { -0.00012534,    0.022453, -0.00040218,   -0.091732,},
                  };
```

```
      static double Dinvd[2][2] = {{-8.7784e-005,     -4.1091,},
                  {     0.14295,    0.014419,},
                  };

//          End of original code
==================================================================
*/

//    U1 = 170.0
//    Created 12/30/2006 at 8:40


      static double Ainvd[4][4] = {{     0.99999,  -0.0038111,   0.0024701,-
3.6259e-005,},
                  {-9.7019e-007,      0.9993,-1.0397e-008,-6.6331e-006,},
                  {   0.0024385,      1.7511,    -0.12307,    0.016661,},
                  { 2.4385e-005,    0.017511,   0.0087693,      1.0002,}};

      static double Binvd[4][2] = {{          -1,    -0.15561,},
                  { 4.8509e-007,    -0.99965,},
                  {  -0.0012193,     -56.795,},
                  {-1.2193e-005,    -0.56795,}};

      static double Cinvd[2][4] = {{   0.0001801,     0.14668,   -0.074342,
0.0012305,},
                  { -0.00012533,    0.022463, -0.00040233,   -0.091732,}};

      static double Dinvd[2][2] = {{-9.0049e-005,     -4.1108,},
                  {     0.14295,    0.014424,}};





//    double Ainv[4][4], Binv[4][2], Cinv[2][4], Dinv[2][2];

      static double Xinv[4] = {0.};                   // Old values of X,
retained as static variables
      double Xnewinv[4];                                          // New
values of X computed in this execution of the function
      double Yinv[2];                                             // Values
for Y computed in this execution of the function
      short nrows=4,ncols=4;                                // Dimensions of
Ainv matrix
      short mAmd=4, nAmd=4;
      short i,j;                                                  // Counters
used in matrix multiplication


//    If this is the first time through...

      if (reset)                              // reset command is sent be
calling subprogram
      {
```

```
            printf("\nReinitializing Inverse Transfer Function");
      // announce reinitializatin

      update_inv_state_space(Ainvd,Binvd,Cinvd,Dinvd,(float)U1,.02,stdout);
      // reinitialize inverse x-fer state space
            for (i=0; i<nrows; i++)
            {
                  Xnewinv[i] = 0.;
                        // reinitialize to zero
                  Xinv[i] = 0.;
                              // reinitialize to zero
            }

      }

//    Compute inverse transfer function from state space model
//
//    Matrix multiplication, Xnew = A*X + B*u

      for (i=0; i<nrows; i++)
      {
            Xnewinv[i] = 0.;
            for (j=0; j<ncols; j++)
            {
                  Xnewinv[i] += Ainvd[i][j]*Xinv[j];
            }
            Xnewinv[i] += Binvd[i][0]*udot_req + Binvd[i][1]*alphadot_req;

      }

//    Matrix multiplication, Y = C*X + D*u
      for (i=0; i<2; i++)
      {
            Yinv[i] = 0.;
            for (j=0; j<ncols; j++)
            {
                  Yinv[i] += Cinvd[i][j]*Xinv[j];
            }
            Yinv[i] += Dinvd[i][0]*udot_req + Dinvd[i][1]*alphadot_req;

      }

//    Results for Yinv are assigned to the variables pointed to by ddePtr and
dplaPtr, for return to the calling program

      *ddePtr = Yinv[0];
      *dplaPtr = 100.*Yinv[1];


//    X = Xnew...  Update the values contained in static array Xinv for use
in the next cycle

      for (i=0; i<nrows; i++)
      {
            Xinv[i] = Xnewinv[i];
      }
```

```
      return 0;
}
/*          End off DEF Control function                          */
```

Function "linear_prefilter_l2A.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int linear_prefilter(int reset, const double dt, double ucom, double
gammacom,float ux, float gammax, float *udot_reqPtr, float *alphadot_reqPtr,
FILE *errFilePtr)
/*    Function to the linear prefilter portion of the controller

              Implements a discretized state space representation of the
A,B,C, and D matrices for the DEF JR controller.
              Includes feedback on speed and flight path angle.

                  Copyright 2007 Hugh S. Bruner

              Matricies Amd, Bmd, Cmd, and Dmd represent the model.

              Arguement List:
              --------------

                  ucom = commanded speed perturbation, fps
                  gammacom = commended flight path angle perturbation,
radians
                  ux = current value of speed perturbation, fps
                  gammax = current value of flight path angle perturbation,
radians
                  *udot_reqPtr = pointer back to the variable recieving the
required linear u-dot
                  *alphadot_reqPtr = pointer back to the variable recieving
the required linear alpha-dot

              Hugh S. Bruner
              December 21, 2006

                  Updated 3/4/2007 to implement a better discretized
implementation of the
                  linear prefilter.  Use "Tustin's" method from MATLAB to
generate A, B,
                  C, and D matrices.


*/
{
      static double Amd[4][4] = {{      0.9685,           0,           0,
0,},
                  {    0.019685,           1,           0,           0,},
                  {           0,           0,     0.32231,     -8.2645,},
                  {           0,           0,    0.013223,     0.91736,},
                  };

      static double Bmd[4][4] = {{     0.98425,           0,    -0.98425,
0,},
                  {   0.0098425,           0,  -0.0098425,           0,},
                  {           0,           0,           0,      413.22,},
                  {           0,           0,           0,      4.1322,},
```

```
                };

      static double Cmd[4][4] = {{    0.019685,           0,           0,
0,},
                {  0.00019685,        0.02,           0,           0,},
                {           0,           0,    0.013223,   -0.082645,},
                {           0,           0,  0.00013223,    0.019174,},
                };

      static double Dmd[4][4] = {{   0.0098425,           0,  -0.0098425,
0,},
                { 9.8425e-005,           0,-9.8425e-005,           0,},
                {           0,           0,           0,      4.1322,},
                {           0,           0,           0,    0.041322,},
                };



      static double Xm[4] = {0.};                               // Old
values of X, retained as static variables
      static double ucomold = 0.,gcomold = 0.;        // Old values of
command variables, retained as static variables
      double Xnewm[4];                                          // New
values of X computed in this execution of the function
      double Ym[2];                                                   //
Values for Y computed in this execution of the function
      short mAmd=4, nAmd=4;
      short i,j;                                                      //
Counters used in matrix multiplication
//    static int matrix_done=0;                                 // logical
value if the matricies have been computed
//    double alphadot, gamma, gammadot;                   // Flight path
parameters
      double lambda1, lambda2;
      double zetag = .707, omegang = 2.50;
      double Ymid[4];


      if (reset)                          // if a reset command was sent from
the calling program
      {
            printf("\nNew Linear Prefilter Model calculated");
      // announce a reset taking place
            for (i=0; i<4; i++)
            {
                  Xm[i] = 0.;
                        // reinitialize Xm to zero
            }
            ucomold = 0.;
                        // reinitialize ucomold
            gcomold = 0.;
                        // reinitialize gcomold
      }

//    Compute model response
//
```

```
//    Matrix multiplication, Xnew = A*X + B*u
      for (i=0; i<mAmd; i++)
      {
            Xnewm[i] = 0.;
            for (j=0; j<nAmd; j++)
            {
                  Xnewm[i] += Amd[i][j]*Xm[j];
            }

            Xnewm[i] += Bmd[i][0]*ucom + Bmd[i][1]*gammacom + Bmd[i][2]*ux+
Bmd[i][3]*gammax;

      }
//    Intermediate y values:  Ymid = C*X + D*u                    Added
3/4/2007
      for (i=0; i<mAmd; i++)
      {
            Ymid[i] = 0.;
            for (j=0; j<nAmd; j++)
            {
                  Ymid[i] += Cmd[i][j]*Xm[j];
            }

            Ymid[i] += Dmd[i][0]*ucom + Dmd[i][1]*gammacom + Dmd[i][2]*ux+
Dmd[i][3]*gammax;

      }

            lambda1 = mass*vtrue/(q1sw*cla);
            lambda2 = (-2.*weight)/(q1sw*cla*vtrue);

            Ym[0] = Ymid[0];
            Ym[1] = lambda2*Ymid[0] -2.*zetag*omegang*lambda1*Ymid[2] -
omegang*omegang*lambda1*(Ymid[3] - gammacom);

//    Results for Y are assigned to the variables *udot_reqPtr and
*alphadot_reqPtr

      *udot_reqPtr = (float)Ym[0];
      *alphadot_reqPtr = (float)Ym[1];

//    X = Xnew... Update the values contained in static array Xm for use in
the next cycle

      for (i=0; i<mAmd; i++)
      {
            Xm[i] = Xnewm[i];
      }

      return 0;
}
/*          End off DEF Control function                    */
```

Function “load_existing_network.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int load_existing_network(Network *netPtr, FILE *filePtr, float
xinputs[],logical readweights)
{
	/*	Copyright 2007 Hugh S. Bruner	*/
	char data_record[READRECLEN] = "";
	int tokenized = FALSE;
	int xinteger;
	int ierror;
	double xdouble;
	short int k;
	short int m;
	short int n;

//	Read in number of network inputs, M. Print error if M > MMAX
	if ((ierror =
read_item(NULL,&xinteger,NULL,filePtr,data_record,&tokenized)) != 0) return
ierror;
	if (xinteger > MMAX)
	{
		fprintf(stderr,"Value of %d exceeds allowable number of network
inputs",xinteger);
		return -3;
	}
	netPtr->M = xinteger;
	netPtr->nk[1] = netPtr->M;		// by convention, first layer
matches number of inputs
//	Read in number of network outputs, N.  Print error if N > NMAX.
	if ((ierror =
read_item(NULL,&xinteger,NULL,filePtr,data_record,&tokenized)) != 0) return
ierror;
	if (xinteger > NMAX)
	{
		fprintf(stderr,"Value of %d exceeds allowable number of network
outputs",xinteger);
		return -4;
	}
	netPtr->N = xinteger;
//	Read in number of network layers, K
	if ((ierror =
read_item(NULL,&xinteger,NULL,filePtr,data_record,&tokenized)) != 0) return
ierror;
	if (xinteger > KMAX)
	{
		fprintf(stderr,"Value of %d exceeds allowable number of network
layers",xinteger);
		return -5;
	}
	netPtr->K = xinteger;
	netPtr->nk[netPtr->K] = netPtr->N;		// by convention, last layer
matches number of outputs
//	For each hidden layer
```

```
      for (k=2; k <= (netPtr->K - 1); k++)
      {
//          Read in the number of neurons
            if ((ierror =
read_item(NULL,&xinteger,NULL,filePtr,data_record,&tokenized)) != 0) return
ierror;
            netPtr->nk[k] = xinteger;
      }
//    End-for on hidden layers

//    OK.  Now make all the network connections
      create_net_connections(netPtr,xinputs);

//    For each layer, k
      for (k=1; k <= netPtr->K; k++)
      {
//          For each neuron on the layer, n
            for (n=1; n <= netPtr->layerPtr[k]->N ; n++)
            {
//                Read the transfer function type
                  if ((ierror =
read_item(NULL,&xinteger,NULL,filePtr,data_record,&tokenized)) != 0) return
ierror;
                  netPtr->layerPtr[k]->neuronPtr[n]->tf =
(transfer_function_type)xinteger;
//                Read the threshold value
                  if ((ierror =
read_item(&xdouble,NULL,NULL,filePtr,data_record,&tokenized)) != 0) return
ierror;
                  netPtr->layerPtr[k]->neuronPtr[n]->threshold =
(float)xdouble;

                  if (readweights)   //   If readweights is TRUE, read all
the individual neuron weights
                  {
//                      Read the bias weight
                        if ((ierror =
read_item(&xdouble,NULL,NULL,filePtr,data_record,&tokenized)) != 0) return
ierror;
                        netPtr->layerPtr[k]->neuronPtr[n]->w[0] =
(float)xdouble;

//                      For each neuron input, m
                        for (m=1; m <= netPtr->layerPtr[k]->neuronPtr[n]->M;
m++)
                        {
//                            Read the weight
                              if ((ierror =
read_item(&xdouble,NULL,NULL,filePtr,data_record,&tokenized)) != 0) return
ierror;
                              netPtr->layerPtr[k]->neuronPtr[n]->w[m] =
(float)xdouble;
                        }
//                      End-for on inputs
                  }
                  else if (k==1)   //     Else if readweights is FALSE, you
must still read the individual neuron weights for the first layer only
```

```
                  {
//                      Read the bias weight

                        if ((ierror =
read_item(&xdouble,NULL,NULL,filePtr,data_record,&tokenized)) != 0) return
ierror;

                        netPtr->layerPtr[k]->neuronPtr[n]->w[0] =
(float)xdouble;

//                      For each neuron input, m
                        for (m=1; m <= netPtr->layerPtr[k]->neuronPtr[n]->M;
m++)
                        {
//                            Read the weight
                              if ((ierror =
read_item(&xdouble,NULL,NULL,filePtr,data_record,&tokenized)) != 0) return
ierror;
                              netPtr->layerPtr[k]->neuronPtr[n]->w[m] =
(float)xdouble;
                        }
//                      End-for on inputs
                  }
            }
//          End-for on neurons
      }
//    End-for on layers


//    For each network output, n
      for (n=1; n <= netPtr->N; n++)
      {
//          Read its scale factor

            if ((ierror =
read_item(&xdouble,NULL,NULL,filePtr,data_record,&tokenized)) != 0) return
ierror;
            netPtr->w0[n] = (float)xdouble;
            if ((ierror =
read_item(&xdouble,NULL,NULL,filePtr,data_record,&tokenized)) != 0) return
ierror;
            netPtr->w1[n] = (float)xdouble;
      }
//    End-for on network outputs

      if (!readweights)             // if readweights is FALSE, seed the
network with random weights
      {
            seed_net(netPtr);
      }

      return 0;
}
```

Function “MyDeclarations.h”

```
//    Global declarations

//          Copyright 2007 Hugh S. Bruner

extern FILE *testoutPtr;

extern short int real_time ;                               // Controls
whether running realtime or not
extern short int log_training_data ;                       // Controls
whether or not to log training data
extern int control_printfreq;                              // Controls
frequency of printing output during sim
extern int control_ptrainfreq;                                   // Controls
frequency of printing training data during sim
extern int control_simfailure;                                   // Controls
whether or not to execute failures during sim
extern double nnrms;                                             // neural
network rms error.


//  Global variables


//          Simulation:

extern DerivPolynomial clov, clav, cldev;
extern DerivPolynomial cldetabv, cladotv, clqv;
extern DerivPolynomial cmov, cmav, cmdev;
extern DerivPolynomial cmdetabv, cmadotv, cmqv;
extern DerivPolynomial cyov, cybv, cydrv;
extern DerivPolynomial cydav, cypv, cyrv;
extern DerivPolynomial cnov, cnbv, cndrv;
extern DerivPolynomial cndav, cnpv, cnrv;
extern DerivPolynomial crov, crbv, crdrv;
extern DerivPolynomial crdav, crpv, crrv;
extern DerivPolynomial epsv;
extern double cheo, chea, chede, chedetab;                             //
elevator hinge moment characteristics
extern double max_alpha, min_alpha;

extern double weight, xcg, dx, mass, ixxb,
      iyyb, izzb, ixzb, idivisor;                                // Mass
properties
extern double sref, cref, bref, cgref;                                 //
Reference geometry
extern double se, ce;
      // elevator area and elevator chord
extern double fds, fbw, ds0, ddsdde, bw0, dbwdde;                //
bobweight and downspring values
extern double ge, gev[4];
      // Elevator gearing curve

extern double clo, cla, clde, cldetab, cladot, clq;              // lift
coefficients and derivatives
```

```
extern double cmo, cma, cmde, cmdetab, cmadot, cmq;              // pitching
moment coefficients and derivatives
extern double cdo, cdk, cli;                                           //
drag polar
extern double cyo, cyb, cydr, cyda, cyp, cyr;                    // side
force coefficients and derivatives
extern double cno, cnb, cndr, cnda, cnp, cnr;                    // yawing
moment coefficients and derivatives
extern double cro, crb, crdr, crda, crp, crr;                    // rolling
moment coefficients and derivatives

extern double ub, vb, wb;                    // Body axis linear velocities
extern double us, vs, ws;                    // Stability axis linear velocities
extern double pb, qb, rb;                    // Body exis rotational velocities
extern double ps, qs, rs;                    // Stability axis rotational
velocities
extern double psi, theta, phi;               // Euler angles
extern double ub0, ubd[5];                   // U body-axis time-derivative and
history vector
extern double vb0, vbd[5];                   // V body-axis time-derivative and
history vector
extern double wb0, wbd[5];                   // W body-axis time-derivative and
history vector
extern double pb0, pbd[5];                   // P body-axis time-derivative and
history vector
extern double qb0, qbd[5];                   // Q body-axis time-derivative and
history vector
extern double rb0, rbd[5];                   // R body-axis time-derivative and
history vector
extern double phi0, phid[5];           // Phi time-derivative and history vector
extern double theta0, thetad[5];       // Theta time-derivative and history
vector
extern double psi0, psid[5];           // Psi time-derivative and history vector
extern double h0, hd[5];                     // Altitude time-derivative and
history vector
extern double xearth,yearth;           // Earth coordinates
extern double mach, vkeas, h, vtrue, rho, q1, U1;
extern double htrim, Utrim;                  // trim values for altitude and
speed
extern double q1sw, q1swbw, q1swcw;

extern double alpha, beta;
extern double sina, cosa, sinth, costh, tanth, sinphi, cosphi;    // trig
quatities of alpha and theta
extern double pla, tcp;                // power lever angle, thrust coefficent
extern double kthrust;                 // thrust multiplier
extern double de,detab,da,dr;          // Control deflectins: elevator, elevator
tab, aileron, and rudder
extern unsigned int start_time, stop_time, elapsed_time;

extern double g;                                   // Gravitational constant
extern double rtod;                                // Radians to degrees
conversion
```

Function "MyDefinitions.h"

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <sys/timeb.h>

/*          Globally used enumerations
            Copyright 2007 Hugh S. Bruner
                                                                */
enum transfer_function_type {IDENTITY,TANH,SIGMOID,BINARY};
enum connectivity_type {LINEAR,FULLY};
enum logical {FALSE,TRUE};

//    Definitions
#define FILEOUT "output.txt"
#define SCRATCH "scratch.tmp"
#define SCRATCH2 "scratch2.tmp"
#define PLOTFILE "output.csv"
#define TRAINFILE "traindata.csv"
#define NMAX 50
#define MMAX 50
#define KMAX 4
#define READRECLEN 256

#define SOURCE "NNinput.txt"
#define OUTPUTFILE "neuralnetout.txt"
#define NETWORKCSV "netconfig.csv"
#define NETWORKCPP "netconfig.cpp"
#define PROGNAM "6DOF NN Program"

//#define DEBUG
//#define VERBOSE
//#define SHOWNEURONS

typedef struct {                                    // Structure to define
a polynomial form
      float constant_term;                          // of expressing the
stability derivatives
      float alpha_term;
      float tcp_term;
      float alpha_tcp_term;
} DerivPolynomial;
/*
typedef struct {
            float time;
            float ucom;
            float gammacom;
            float vpdot_req;
            float alphadot_req;
            float vp;
            float alpha;
            float qrate;
            float theta;
            float vpdot;
```

```
            float alphadot;
            float dde;
            float dpla;
} TrainingData;               */

typedef struct {
      double time;
      double vkeas;
      double alpha;
      double beta;
      double phi;
      double theta;
      double gamma;
      double psi;
      double de;
      double detab;
      double dr;
      double da;
      double pla;
      double dde;
      double ddr;
      double dda;
      double dpla;
      double h;
      double xearth;
      double yearth;
      double rmserror;
      double ucom;
      double gammacom;
      double vtrue;
} BinaryRecord;



/*          Structure definition for user defined data-type Neuron
      */
typedef struct {
      float y;
      float yprime;
      float delta;
      transfer_function_type tf;
      float threshold;
      short int M;
      float *xPtr[MMAX + 1];              //*xPtr[0] is never used, data on
[1] through [MMAX+1]
//    Layer *parent_layerPtr;
      float w[MMAX + 1];
} Neuron;

/*          Structure definition for user defined data-type Layer
      */
typedef struct {
      short int N;
      Neuron *neuronPtr[NMAX + 1];  //*neuronPtr[0] is never used, data on
[1] through [MMAX+1]
      short int M;
```

```
      float *xPtr[MMAX + 1];              //*xPtr[0] is never used, data on 
[1] through [MMAX+1]
      connectivity_type connect;
      float *yPtr[NMAX + 1];              //*yPtr[0] is never used, data on 
[1] through [NMAX+1]
//    Network *networkPtr;
} Layer;

/*          Structure definition of user defined data-type TrainingDataSet
      */
typedef struct{
      struct trainstruct *topPtr;
      struct trainstruct *bottomPtr;
      int npairs;
      float beta;
      float learning_rate;
      int max_epochs;
      float tolerance;
} TrainingDataSet;

/*          Structure definition of user defined data-type Network
      */
typedef struct {
      short int M;
      short int N;
      float *xPtr[MMAX + 1];        //*xPtr[0] is never used, data on [1] 
through [MMAX+1]
      float *yPtr[NMAX + 1];        // pointers to the output neurons
      float Y[NMAX + 1];                  // neuron values scaled up to world 
units
      float w0[NMAX + 1];                 //w0[0] is never used, data on [1] 
through [NMAX+1]
      float w1[NMAX + 1];                 //w1[0] is never used, data on [1] 
through [NMAX+1]
      short int K;
      short int nk[KMAX + 1];       //nk[0] is never used, data on [1] 
through [KMAX+1]
      Layer *layerPtr[KMAX + 1];    //*layerPtr[0] is never used, data on [1] 
through [KMAX+1]
      TrainingDataSet *trainingdataPtr;
      short int flag_input_errors;
} Network;
typedef Network *PointertoNetPtr;   // "pointer to a pointer"  used to call 
initialize_network



/*          Structure definition of user defined data-type TrainingPair
      */
struct trainstruct{
      float input[MMAX];
      float output[NMAX];
      struct trainstruct *prevPtr;
      struct trainstruct *nextPtr;
} ;
typedef trainstruct TrainingPair;
```

```
double eval(DerivPolynomial *, const double *, const double *);
double predict(const double *, const double *, const double *);
double correct(const double *, const double *, const double *);
double std_temp(double hp);
double std_pressure(double hp);
double std_density(double hp);
int initialize_sim(void);
int fast(double *, double dt);
int push_down(void);
int CJ144_data(void);
int force_roskam(void);
int case_data_write(FILE *ffilePtr, char *descriptorPtr);
int fetrim(double *alpha1Ptr, double *de1Ptr, double *detab1Ptr, double
*pla1Ptr);
int solve3by3(double a[3][3], double x[3], double y[3]);
int autopilot(Network *myNetPtr, const double *mytimePtr, const double dt,
const double vkeastrim, double utrimfps,
                    const int manuever, double de1, double da1, double dr1,
double pla1,
                    double *dePtr, double *daPtr, double *drPtr, double
*plaPtr, double *ucommandPtr, double *gammacommandPtr, FILE *filePtr,
                    FILE *errFilePtr, FILE *modeloutPtr, double
time_to_learn, double time_to_control);
int format_files(FILE *ffilePtr, int isims, char *descriptor);
int simulation_timer(int wait_millisec);
double thrust_model(double PLA, double vtrue_fps, double air_density);
int DEF_JRcontrol(Network *myNetPtr, const double *mytimePtr, const double
dt, double time_to_learn, double time_to_control,
                          float ucom,float gammacom,float ux, float gammax,
double *ddePtr,double *dplaPtr, FILE *filePtr,
                          FILE *errFilePtr, FILE *modeloutPtr);
int write_out_training(TrainingPair myTrainingData, FILE *filePtr);
int pop_training_pair(TrainingPair *currentPtr, float vpdot_req, float
alphadot_req);
int format_training_data(FILE *outfilePtr,int pfreq);
int forward_pass(Network *currentNet, TrainingPair *currentpairPtr);
float backprop(Network *netPtr, TrainingPair *dataPtr, float learning_rate);
int simNN(FILE *inputPtr, Network *myNetPtr, double const timemax, double
time_to_learn, double time_to_control);
int NNexec(char *sourcefilePtr);
char * extract_line(char * sourcePtr, char * var_namePtr, char * op_namePtr,
char * value_stringPtr);
int read_actions(FILE *inputPtr, FILE *outputPtr, PointertoNetPtr *netPtrPtr,
char direct_object[],float xinputs[]);
int print_the_network_csv(Network *netPtr, FILE *logfilePtr);
int print_the_network_Ccode(Network *netPtr, FILE *logfilePtr);
int print_the_network(Network *netPtr, FILE *outputPtr);
int no_white_space(const char *sourcePtr, char *targetPtr);
int clear_the_network(Network *myNetworkPtr);
int erase_network(Network *networkPtr);
int initialize_network(PointertoNetPtr *netPtr);
int error_handler(int error_code, char *descr);
int create_new__network(Network *netPtr, FILE *filePtr, float xinputs[]);
int load_existing_network(Network *netPtr, FILE *filePtr, float
xinputs[],logical readweights);
int delete_net_connections(Network *myNetwork);
```

```
int read_item(double *dvaluePtr, int *ivaluePtr, char *svaluePtr, FILE
*sourcePtr, char *last_recordPtr, int *tokenizedPtr);
int create_net_connections(Network *networkPtr, float xinputs[]);
int print_training_performance(Network *networkPtr);
int delete_training_data(Network *netPtr);
int populate_layer(Layer *current_layer);
float evaluate_neuron(Neuron *current);
int randomCom(float *ucommand, float *alphacommand);
int seed_net(Network *netPtr);
int get_train_options(char *sourcePtr, double *start_learn, double
*start_control);
int initialize_by_code(Network *netPtr, float xinputs[]);
int input_excedance_alarm(Network *myNetPtr, FILE *errorLogPtr, const double
*mytimePtr);
int print_matrix(char *string, double matrix[][4], int m, int n, FILE
*outputPtr);
int print_matrix5(char *string, double matrix[][5], int m, int n, FILE
*outputPtr);
int command_input(const double *mytimePtr, const double dt, float
*ucommandPtr, float *gammacommandPtr,
                          const int manuever, FILE *errFilePtr);
int inverse_xfunction(int reset, const float udot_req, const float
alphadot_req, double *ddePtr, double *dplaPtr);
int update_inv_state_space(double Ainv[4][4], double Binv[4][2],
                                       double Cinv[2][4], double Dinv[2][2],
                                       float Uref, double dt, FILE
*logfilePtr);
int linear_prefilter(int reset, const double dt, double ucom, double
gammacom,float ux, float gammax, float *udot_reqPtr, float *alphadot_reqPtr,
FILE *errFilePtr);
int model_follower( double A[4][4], int ma, int na, double B[4][4], int mb,
int nb, double C[2][4], int mc, int nc,
                           double D[2][4], int md, int nd, const double dt,
                           const double lambda1, const double lambda2);
double alpha_limiter(double alphadot_command,double alpha,double alphamin,
double alphamax,
                          double alphadot_current,double dt, double mytime);
double alpha_limiter3(double alphadot_command,double alpha,double alphamin,
double alphamax,
                          double alphadot_current,double dt, double mytime,
FILE *logfilePtr);
int read_options(FILE *inputPtr, FILE *outputPtr);
int execute_failure(double mytime, int manuever);
int initialize_training_pair(TrainingPair *currentPtr);
int netfailure(double mytime, int ucode,int alphacode,float *udotPtr, float
*alphadotPtr, float ugain, float alphagain);
//
```

Function "NN6DOF_2.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"


int main(int argc, char* argv[])
{
	/*		Copyright 2007 Hugh S. Bruner		*/

	int ierror;
	char sourcefile[80];

	printf("\nargc = %d",argc);
	printf("\nargv[1] = %s",argv[1]);

	if (argc == 1)
	{
		strcpy(sourcefile,SOURCE);
	}
	else
	{
		strcpy(sourcefile,argv[1]);
	}

	printf("\nSource file is %s",sourcefile);

	ierror = NNexec(sourcefile);
	if (ierror != 0)
	{
		fprintf(stderr,"\nSimulation failed with error code = 
%d.\n",ierror);
	}


	return 0;
}
```

Function "netfailure.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int netfailure(double mytime, int ucode,int alphacode,float *udotPtr, float
*alphadotPtr, float ugain, float alphagain)
{
/*
            Function to simulate failures in the neural network
            Copyright 2007 Hugh S. Bruner
*/
      float omegau = (float).6;                  // frequency for the udot
oscillatory failure
      float omegaa = (float)1.785;         // frequency for the alpha dot
oscillatory failure
      float tstart = 6;                          // time when the failure
occurs
      static int latched = 0;
/*
============================U dot failure section ============================

                                    */
      switch (ucode)    /*          Switch based on value supplied for ucode
      */
      {
            case 0:           /*          Do nothing case         */

                  break;

            case 1:           /*          Hardover                */

                  if (mytime > tstart)
                  {
                        *udotPtr = ugain;
                        if (!latched)
                        {
                              printf("\nUdot hardover network failure at %.2f
seconds.", mytime);
                              printf("\n\tugain = %f",ugain);
                              printf("\n\talphagain = %f",alphagain);
                              latched = 1;
                        }
                  }
                  break;

            case 2:           /*          Oscillatory failure           */

                  if (mytime > tstart)
                  {
                        *udotPtr = ugain*sin(omegau*(mytime - tstart));
                        if (!latched)
                        {
                              printf("\nUdot oscillatory network failure at
%.2f seconds.", mytime);
                              printf("\n\tugain = %f",ugain);
```

```
                              latched = 1;
                        }
                  }
                  break;
      }
/*
================  Alpha dot failure section ================================

                                    */
      switch (alphacode)      /*          Switch based on value supplied for
alphacode         */
      {
            case 0:           /*          Do nothing case         */

                  break;

            case 1:           /*          Hardover                */

                  if (mytime > tstart)
                  {
                        *alphadotPtr = alphagain;
                        if (!latched)
                        {
                              printf("\nAlphadot hardover network failure at
%.2f seconds.", mytime);
                              printf("\n\talphagain = %f",alphagain);
                              latched = 1;
                        }
                  }
                  break;

            case 2:           /*          Oscillatory failure           */

                  if (mytime > tstart)
                  {
                        *alphadotPtr = alphagain*sin(omegaa*(mytime -
tstart));
                        if (!latched)
                        {
                              printf("\nAlpha dot oscillatory network failure
at %.2f seconds.", mytime);
                              printf("\n\talphagain = %f",alphagain);
                              latched = 1;
                        }
                  }
                  break;
      }

      return 0;

}
```

Function “NNexec.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"


int NNexec(char *sourcefilePtr)
{
	/*          Copyright 2007 Hugh S. Bruner             */
	char verb[128];
	char operatorName[2];
	char direct_object[128];
	char data_record[READRECLEN];
	char *remaining;
	float xinputs[MMAX+1] = {0.};
	double timemax;
	double time_to_learn;
	double time_to_control;
	char *stringPtr;
	FILE *outputPtr;
	FILE *inputFilePtr;
	Network *myNetPtr;
	time_t long_time;            // special data type defined in time.h
for use with date/time
	struct tm *newtime;                // special data type defined in
time.h for use with date/time
	int ierror;

//	Initialize variables:

//	Set myNetPtr to NULL
	myNetPtr = NULL;
//	Set outputPtr to stdout
	outputPtr = stdout;
//	Try to open output file for write-only access
	outputPtr = fopen(OUTPUTFILE,"w");
//	If attempt fails
	if (outputPtr == NULL)
	{
//		Print error message to stderr
		fprintf(stderr,"\nUnable to open file %s for write-only
access.",OUTPUTFILE);
		fprintf(stderr,"\nIERROR = -1");
//		Set error code to -1
		ierror = -1;
	}
//	Else if file opens OK
//	Initialize options via a user-defined function
//	Try to open input file for read-only access
//	inputFilePtr = fopen(SOURCE,"r");
	inputFilePtr = fopen(sourcefilePtr,"r");
//	If attempt fails
	if (inputFilePtr == NULL)
	{
//		Print error message to stderr
```

```
            fprintf(stderr,"\nUnable to open file %s for read-only 
access.",SOURCE);
            fprintf(stderr,"\nIERROR = -1");
//          Set error code to -1
            ierror = -1;
      }

//    Else if file opens OK
      else
      {
//          Verify this data file belongs with this program by looking at the 
first data record
            fprintf(stderr,"\nFile %s opened correctly for read-only 
access.",SOURCE);
            fgets(data_record,READRECLEN,inputFilePtr);
//          If the record does not contain the appropriate signature string, 
it does not belong
            if (strstr(data_record,"6DOF NN Ver") == NULL)
            {
//                Print error message to stderr
                  fprintf(stderr,"\nCannot recognize this file as a 6DOF NN 
file.");
//                Set error code to -2
                  ierror = -2;
                  fprintf(stderr,"\nIERROR = %d",ierror);
            }

//          Else, all is well.  Proceed.  The following code block is the 
core of the program.
//          This is where the fun really begins!!
            else
            {
                  time( &long_time );                /* Get time as long 
integer. */
                  newtime = localtime( &long_time ); /* Convert to local 
time. */
//          Display time
                  fprintf(outputPtr,"\n%02d:%02d:%02d",newtime-
>tm_hour,newtime->tm_min,newtime->tm_sec);
                  fprintf(outputPtr,"%48s%02d/%02d/%4d"," ",(1+newtime-
>tm_mon),newtime->tm_mday,(1900+newtime->tm_year));

                  fprintf(outputPtr,"\nExecuting program %s\n", PROGNAM);
//                Read a data record from the input file
                  fgets(data_record,READRECLEN,inputFilePtr);
//                While not at the end of the input file
                  while (!feof(inputFilePtr))
                  {
                        printf("\nData Record: %s",data_record);
#ifdef VERBOSE
                        printf("\nData Record: %s",data_record);
#endif
//                      Extract a verb, an operator, and a direct object from 
the record
                        remaining = 
extract_line(data_record,verb,operatorName,direct_object);
```

```
//                        If the verb is READ
                          if (strcmp(verb,"READ") == 0)
                          {
//                              Execute function read_actions
//                              read_actions(inputFilePtr, outputPtr,
myNetPtr,direct_object, xinputs);
                                read_actions(inputFilePtr, outputPtr,
&myNetPtr,direct_object, xinputs);

                          }
//                        Else if the verb is TRAIN  (TRAIN runs the simulation
with the NN active).

                          else if (strcmp(verb,"TRAIN") == 0)
                          {

                                if (strcmp(direct_object,"\0") == 0)
      // if the direct object is an empty string
                                {
                                      timemax = 50.;
                                      time_to_learn = 0.;
                                      time_to_control = 0;
                                }
                                else
                                {
                                      timemax =
strtod(direct_object,&stringPtr);
                                      get_train_options(remaining,
&time_to_learn, &time_to_control);

                                }
//                              Execute the function simNN to train the net

                                /*    Log your start time before beginning the
simulation  */

                                time( &long_time );                /* Get time
as long integer. */
                                newtime = localtime( &long_time ); /* Convert
to local time. */
                                fprintf(outputPtr,"\n%02d:%02d:%02d",newtime-
>tm_hour,newtime->tm_min,newtime->tm_sec);
                                fprintf(outputPtr,"\tWill train for %.2f
simulated seconds",timemax);
                                fprintf(outputPtr,"\n\tCommence learing at %f
seconds.",time_to_learn);
                                fprintf(outputPtr,"\n\tNeural network engages
at %f seconds.",time_to_control);

                                if (myNetPtr != NULL)              // Here is
where you finally run the simulation <<<<<<
                                {

                                      ierror = simNN(inputFilePtr,
myNetPtr,timemax,time_to_learn,time_to_control);
                                }
                                else
```

```
                              {

                                    ierror = -22;
                                    fprintf(outputPtr,"\n\tNo Network created
prior to starting simulation");
                                    fprintf(stderr,"\n\tNo Network created
prior to starting simulation");
                              }

                              /*    Log your end time after running the
simulation  */

                              time( &long_time );                /* Get time
as long integer. */
                              newtime = localtime( &long_time ); /* Convert
to local time. */
                              fprintf(outputPtr,"\n%02d:%02d:%02d",newtime-
>tm_hour,newtime->tm_min,newtime->tm_sec); // log end time
                              if (ierror == 0)
                              {
                                    fprintf(outputPtr,"\tSimulation
terminated normally.");
                              }
                              else
                              {
                                    fprintf(outputPtr,"\tSimulation ended
abnormally with error code %d",ierror);
                                    fprintf(stderr,"\tSimulation ended
abnormally with error code %d",ierror);
                              }

                              printf("\nCase completed.\n");
                        }


//                      Else if the verb is PRINT
                        else if (strcmp(verb,"PRINT") == 0)
                        {
//                            Execute function print_the_network
                              time( &long_time );                /* Get time
as long integer. */
                              newtime = localtime( &long_time ); /* Convert
to local time. */
                              fprintf(outputPtr,"\n%02d:%02d:%02d",newtime-
>tm_hour,newtime->tm_min,newtime->tm_sec);
                              fprintf(outputPtr,"\tPrinted network
configuration follows:");
                              if (strcmp(direct_object,"CSV") == 0)
                              {

      print_the_network_csv(myNetPtr,outputPtr);
                              }
                              else if (strcmp(direct_object,"CPP") == 0)
                              {

      print_the_network_Ccode(myNetPtr,outputPtr);
                              }
```

```
                              else
                              {
                                    print_the_network(myNetPtr,outputPtr);
                              }

                        }
//                      Else if the verb is CLEAR
                        else if (strcmp(verb,"CLEAR") == 0)
                        {
//                            Execute function delete_the_net
                              clear_the_network(myNetPtr);
                              fprintf(outputPtr,"\nNormally you would CLEAR
here.");
                        }
//                      Else if you cannot recognize the verb
                        else
                        {
//                            Print error message to stderr
                              fprintf(stderr,"\nCannot recognize %s as a
valid verb in main.",verb);
//                            Set error code to -3
                              ierror = -3;
                              fprintf(stderr,"\nIERROR = %d",ierror);
                        }
//                      end-if on verb
//                      Read a data record from the input file
                        fgets(data_record,READRECLEN,inputFilePtr);
                  }
//                End while on end of file

//                Free memory associated with  all items created by memory
allocation:
//                      Training data, Network connections, and the Network
itself
                  erase_network(myNetPtr);
//                Set error code to 0 for correct execution
                  ierror = 0;
            }
//          End-if on proper data file
      }
//    End-if on file opening correctly

//    Return error code to calling program

      return ierror;

}

/*                End of main program
                        */
```

Function "no_white_space.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int no_white_space(const char *sourcePtr, char *targetPtr)
/*
      A function to strip away any white space from the source string, and
copy the
      remaining characters to the target string.  White space is defined as
non-printing
      character, including blanks, with an ASCII code less than 33.

      Copyright 2007 Hugh S. Bruner

                                                */
{
      short int i;                                          // counter
      short int len_source;                           // will receive
the length of the source string
      short int started = 0;                          // logical if
started collecting text to target
      short int wt_space = 32;                        // threshhold
ASCII value for white space

      len_source = strlen(sourcePtr);                 // get the length
of the source string

      for (i = 0; i < len_source; i++)          // for each character
in the string
      {
            if (*sourcePtr > wt_space)                // if the ASCII
value is greater than threshhold
            {
                  if (*sourcePtr == 34)               // If the
character is a quotation mark...
                  {
                        if (!started)                       // ... and
you haven't accumulated anything, take this
                        {                                         // as
an open quote.  Close down the white space
                              wt_space = 31;                // 
threshhold to accept blanks.
                        }
                        else                                // ...
else if you are already accumulating,
                        {                                         // 
assume this is a close quote.  Stop accepting blanks.
                              wt_space = 32;                // Reset
the white space theshhold.
                        }
                  }

                  *targetPtr = *sourcePtr;            // copy the
character to the target string
```

```
                  started = 1;                                    // started
is now true
                  ++targetPtr;                                    //
increment the target pointer
            }
            ++sourcePtr;                                          //
increment the source pointer
      }
      *targetPtr = '\0';                                          // null
terminate the target
      return 0;
}
```

Function "pop_training_pair.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int pop_training_pair(TrainingPair *currentPtr, float vpdot_req, float
alphadot_req)
/*

      Copyright 2007 Hugh S. Bruner

  This function populates the training pair structure with the current values

  Training pair map:

      input[0] = Utrim
      input[1] = htrim
      input[2] = vpdot_req(t)
      input[3] = alphadot_req(t)

      input[4] = vtrue(t)
      input[5] = alpha(t)
      input[6] = qb(t)
      input[7] = theta(t)
      input[8] = hp(t)
      input[9] = de(t)
      input[10] = pla(t)

      input[11] = vtrue(t-1)
      input[12] = alpha(t-1)
      input[13] = qb(t-1)
      input[14] = theta(t-1)
      input[15] = hp(t-1)
      input[16] = de(t-1)
      input[17] = pla(t-1)

      input[18] = vtrue(t-2)
      input[19] = alpha(t-2)
      input[20] = qb(t-2)
      input[21] = theta(t-2)
      input[22] = hp(t-2)
      input[23] = de(t-2)
      input[24] = pla(t-2)

      input[25] = delta vp_req(t-1)
      input[26] = delta alphadot_req(t-1)

      input[27] = delta vp_req(t-2)
      input[28] = delta alphadot_req(t-2)

      input[29] = delta vp_req(t-3)
      input[30] = delta alphadot_req(t-3)
*/
{
      short int i;
```

```
//    Shuffle the "old values" down...

      for (i=24; i>=11; i--)
      {
            currentPtr->input[i] = currentPtr->input[i - 7];
      }
      for (i = 30; i>=27; i--)
      {
            currentPtr->input[i] = currentPtr->input[i-2];
      }

      currentPtr->input[0] = (float)Utrim;
      currentPtr->input[1] = (float)htrim;
      currentPtr->input[2] = vpdot_req;
      currentPtr->input[3] = alphadot_req;

      currentPtr->input[4] = (float)vtrue;
      currentPtr->input[5] = (float)alpha;
      currentPtr->input[6] = (float)qb;
      currentPtr->input[7] = (float)theta;
      currentPtr->input[8] = (float)h;
      currentPtr->input[9] = (float)de;
      currentPtr->input[10] = (float)pla;

      currentPtr->input[25] = currentPtr->output[0];
      currentPtr->input[26] = currentPtr->output[1];

/*
      currentPtr->input[25] = (float)((ub*ubd[0] + wb*wbd[0])/vtrue) -
currentPtr->input[1];
      currentPtr->input[26] = (float)((ub*wbd[1] - wb*ubd[0])/(vtrue*vtrue))
- currentPtr->input[1];
      */

      return 0;
}
/*                End of Function pop_training_pair               */
```

Function "populate_layer.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int populate_layer(Layer *current_layer)
{
	/*		Copyright 2007 Hugh S. Bruner		*/
	short int m;
	short int n;
	Neuron *newNeuron;

//	For each neuron desired, n
	for (n=1;n<=current_layer->N;n++)
	{
//		Allocate memory and get pointer to new instance
		newNeuron = (Neuron *)malloc(sizeof(Neuron));
//		Assign pointer address to neuronPtr[n], and initialize other
values
		current_layer->neuronPtr[n] = newNeuron;
		newNeuron->y = 0.;
		newNeuron->yprime = 1;
		newNeuron->tf = TANH;
		newNeuron->threshold = 0.;
		newNeuron->w[0] = 0.;				// set bias input to
zero
//		Assign the layer address to the neuron's parent_layer_ptr
attribute
//		If connectivity is LINEAR
		if (current_layer->connect==LINEAR)
		{
//			There is only one input to the neuron
			newNeuron->M = 1;
//			Assign the address at layer input pointer [n] to the neuron
input pointer
			newNeuron->xPtr[1] = current_layer->xPtr[n];
//			initialize weight to 1.
			newNeuron->w[1] = 1;
		}
//		Otherwise, connectivity must be FULL
		else
		{
			newNeuron->M = current_layer->M;
//			For each layer input, m
			for (m=1; m<=current_layer->M; m++)
			{
//				Assign the address at the layer input pointer [m] to
the neuron input ponter [m]
				newNeuron->xPtr[m] = current_layer->xPtr[m];
//				Initialize weight to 1.
				newNeuron->w[m]=1.;
			}
//			End-for
		}
//		End-if on connectivity
```

```
//          Assign the memory address of the neuron output to layer output
pointer [n]
            current_layer->yPtr[n] = &(newNeuron->y);
#ifdef VERBOSE
            printf("\n\nNeuron %d:\n",n);
            dump_neuron(newNeuron);
#endif
      }
//    end-for on neurons
      return 0;
}
/*          End function populate_layer
                                    */
```

Function "predict.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Function to predict a state variable
            Copyright 2007 Hugh S. Bruner
                                            */
double predict(const double *ydotPtr, const double *yoldPtr, const double
*dtPtr)
{
//    Declarations
      double ynew;
//    Apply fourth-order Adams predictor formula
//    predict = yold + dt * (ydot(2) * a(1) - ydot(3) * a(2) + ydot(4) * a(3)
- ydot(5) * a(4))
      ynew = *yoldPtr + (*dtPtr) * (ydotPtr[1] * 2.2917 - ydotPtr[2] * 2.4583
            + ydotPtr[3] * 1.5417 - ydotPtr[4] * 0.375);
      return ynew;
}
/*          End of function
                         */
```

Function "print_matrix.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int print_matrix(char *string, double matrix[][4], int m, int n, FILE
*outputPtr)

/*          Function to print the m by n matrix to stream outputPtr
            Copyright 2007 Hugh S. Bruner
                  */
{
      int i, j;

      fprintf(outputPtr,"\nMatrix %s = ",string);
      for (i=0; i<m; i++)
      {
            fprintf(outputPtr,"\n");
            for (j=0; j<n; j++)
            {
                  fprintf(outputPtr,"%f  ",matrix[i][j]);
            }
      }

      return 0;

}
```

Function "print_the_network.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int print_the_network(Network *netPtr, FILE *outputPtr)
{
	/*		Copyright 2007 Hugh S. Bruner		*/

	short int k;
	short int m;
	short int n;

#ifdef VERBOSE
	printf("\nEntering print_the_network");
#endif

//	Print the number of network inputs, M.
	fprintf(outputPtr,"\n%d  ",netPtr->M);

//	Print the  number of network outputs, N.
	fprintf(outputPtr,"%d  ",netPtr->N);

//	Print the number of network layers, K
	fprintf(outputPtr,"%d  ",netPtr->K);

//	For each hidden layer
	for (k=2; k <= (netPtr->K - 1); k++)
	{
//		Print the number of neurons
		fprintf(outputPtr,"%d  ",netPtr->nk[k]);
	}
//	End-for on hidden layers

	fprintf(outputPtr,"\n");

//	For each layer, k
	for (k=1; k <= netPtr->K; k++)
	{
//		For each neuron on the layer, n
		for (n=1; n <= netPtr->layerPtr[k]->N ; n++)
		{
//			Print the transfer function type
			fprintf(outputPtr,"%d  ",netPtr->layerPtr[k]->neuronPtr[n]-
>tf);

//			Print the threshold value
			fprintf(outputPtr,"%12.4g  ",netPtr->layerPtr[k]-
>neuronPtr[n]->threshold);

//			Print the bias weight
			fprintf(outputPtr,"%12.4g  ",netPtr->layerPtr[k]-
>neuronPtr[n]->w[0]);

//			For each neuron input, m
			for (m=1; m <= netPtr->layerPtr[k]->neuronPtr[n]->M; m++)
```

```
                  {
//                      Print the weight
                        fprintf(outputPtr,"%12.4g  ",netPtr->layerPtr[k]-
>neuronPtr[n]->w[m]);
                  }
//                End-for on inputs

                  fprintf(outputPtr,"\n");
            }
//          End-for on neurons
      }
//    End-for on layers

//    For each network output, n
      for (n=1; n <= netPtr->N; n++)
      {
//          Print its scale factor
            fprintf(outputPtr,"%12.4g  ",netPtr->w0[n]);
            fprintf(outputPtr,"%12.4g  ",netPtr->w1[n]);
      }
//    End-for on network outputs

      fprintf(outputPtr,"\n");

      return 0;
}
```

Function "print_the_network_Ccode.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int print_the_network_Ccode(Network *netPtr, FILE *logfilePtr)
{
	/*		Copyright 2007 Hugh S. Bruner		*/

	short int k;
	short int m;
	short int n;
	time_t long_time;		// special data type defined in time.h
for use with date/time
	struct tm *newtime;			// special data type defined in
time.h for use with date/time
	FILE *outputPtr;

#ifdef VERBOSE
	printf("\nEntering print_the_network_Ccode");
#endif
//	Open the network C code file
	outputPtr = fopen(NETWORKCPP,"w");
	if (outputPtr == NULL)
	{
		fprintf(stderr,"Unable to open code file for saving network
configuration.");
		return -1;
	}
	fprintf(logfilePtr,"\n\t\tWriting network configuration data to file
%s",NETWORKCPP);


	time( &long_time );			/* Get time as long integer. */
	newtime = localtime( &long_time ); /* Convert to local time. */

//	Generate pre-processor directives
	fprintf(outputPtr,"\n#include \42stdafx.h\42");
	fprintf(outputPtr,"\n#include \42MyDefinitions.h\42");
	fprintf(outputPtr,"\n#include \42MyDeclarations.h\42\n");

//	Generate function definition with annotation, including date and time
generated
	fprintf(outputPtr,"\nint initialize_by_code(Network *netPtr, float
xinputs[])");
	fprintf(outputPtr,"\n/*\t\tFunction to initialize network
programmatically");
	fprintf(outputPtr,"\n\t\tCreated %02d:%02d:%02d",newtime-
>tm_hour,newtime->tm_min,newtime->tm_sec);
	fprintf(outputPtr,"%24s%02d/%02d/%4d"," ",(1+newtime->tm_mon),newtime-
>tm_mday,(1900+newtime->tm_year));
	fprintf(outputPtr,"\t\t\t\t\t*/");
	fprintf(outputPtr,"\n{\n");

//	Print the number of network inputs, M.
	fprintf(outputPtr,"\n\t\tnetPtr->M = %d;  ",netPtr->M);
```

```
//    Print the  number of network outputs, N.
      fprintf(outputPtr,"\n\t\tnetPtr->N = %d;  ",netPtr->N);

//    Print the number of network layers, K
      fprintf(outputPtr,"\n\t\tnetPtr->K = %d;  ",netPtr->K);

      fprintf(outputPtr,"\n\t\tnetPtr->nk[1] = netPtr->M;\t\t\t\t// by
convention, first layer matches number of inputs");

//    For each hidden layer
      for (k=2; k <= (netPtr->K - 1); k++)
      {
//          Print the number of neurons
            fprintf(outputPtr,"\n\t\tnetPtr->nk[%d] = %d;  ",k, netPtr-
>nk[k]);
      }
//    End-for on hidden layers

//    Insert necessary code verbatim
      fprintf(outputPtr,"\n\t\tnetPtr->nk[netPtr->K] = netPtr->N;\t\t// by
convention, last layer matches number of outputs");
      fprintf(outputPtr,"\n\n//\t\tOK.  Now make all the network
connections");
      fprintf(outputPtr,"\n\t\tcreate_net_connections(netPtr,xinputs);");

      fprintf(outputPtr,"\n");

//    For each layer, k
      for (k=1; k <= netPtr->K; k++)
      {
//          For each neuron on the layer, n
            for (n=1; n <= netPtr->layerPtr[k]->N ; n++)
            {
//                Print the transfer function type
                  fprintf(outputPtr,"\n\t\tnetPtr->layerPtr[%d]-
>neuronPtr[%d]->tf = (transfer_function_type)%d;  ",k,n,netPtr->layerPtr[k]-
>neuronPtr[n]->tf);

//                Print the threshold value
                  fprintf(outputPtr,"\n\t\tnetPtr->layerPtr[%d]-
>neuronPtr[%d]->threshold = (float)%.4g;  ",k,n,netPtr->layerPtr[k]-
>neuronPtr[n]->threshold);

//                Print the bias weight
                  fprintf(outputPtr,"\n\t\tnetPtr->layerPtr[%d]-
>neuronPtr[%d]->w[0] = (float)%.4g;  ",k,n,netPtr->layerPtr[k]->neuronPtr[n]-
>w[0]);

//                For each neuron input, m
                  for (m=1; m <= netPtr->layerPtr[k]->neuronPtr[n]->M; m++)
                  {
//                      Print the weight
                        fprintf(outputPtr,"\n\t\tnetPtr->layerPtr[%d]-
>neuronPtr[%d]->w[%d] = (float)%.4g;  ",k,n,m,netPtr->layerPtr[k]-
>neuronPtr[n]->w[m]);
                  }
```

```
//                  End-for on inputs

                    fprintf(outputPtr,"\n");
              }
//            End-for on neurons
       }
//     End-for on layers

//     For each network output, n
       for (n=1; n <= netPtr->N; n++)
       {
//            Print its scale factor
              fprintf(outputPtr,"\n\t\tnetPtr->w0[%d] = (float)%.4g;
",n,netPtr->w0[n]);
              fprintf(outputPtr,"\n\t\tnetPtr->w1[%d] = (float)%.4g;
",n,netPtr->w1[n]);
       }
//     End-for on network outputs

       fprintf(outputPtr,"\n");

       fprintf(outputPtr,"\n\t\treturn 0;\n");
       fprintf(outputPtr,"\n}\n");

       fclose(outputPtr);

       return 0;
}
```

Function "print_the_network_csv.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int print_the_network_csv(Network *netPtr, FILE *logfilePtr)
{
	/*		Copyright 2007 Hugh S. Bruner			*/
	short int k;
	short int m;
	short int n;
	FILE *outputPtr;

#ifdef VERBOSE
	printf("\nEntering print_the_network");
#endif
//	Open the network CSV file
	outputPtr = fopen(NETWORKCSV,"w");
	if (outputPtr == NULL)
	{
		fprintf(stderr,"Unable to open CSV for saving network 
configuration.");
		return -1;
	}
	fprintf(logfilePtr,"\nWriting network configuration data to file 
%s",NETWORKCSV);

//	Print the number of network inputs, M.
	fprintf(outputPtr,"\n%d,  ",netPtr->M);

//	Print the  number of network outputs, N.
	fprintf(outputPtr,"%d,  ",netPtr->N);

//	Print the number of network layers, K
	fprintf(outputPtr,"%d,  ",netPtr->K);

//	For each hidden layer
	for (k=2; k <= (netPtr->K - 1); k++)
	{
//		Print the number of neurons
		fprintf(outputPtr,"%d,  ",netPtr->nk[k]);
	}
//	End-for on hidden layers

	fprintf(outputPtr,"\n");

//	For each layer, k
	for (k=1; k <= netPtr->K; k++)
	{
//		For each neuron on the layer, n
		for (n=1; n <= netPtr->layerPtr[k]->N ; n++)
		{
//			Print the transfer function type
			fprintf(outputPtr,"%d,  ",netPtr->layerPtr[k]-
>neuronPtr[n]->tf);
```

```
//                  Print the threshold value
                    fprintf(outputPtr,"%12.4g,  ",netPtr->layerPtr[k]-
>neuronPtr[n]->threshold);

//                  Print the bias weight
                    fprintf(outputPtr,"%12.4g,  ",netPtr->layerPtr[k]-
>neuronPtr[n]->w[0]);

//                  For each neuron input, m
                    for (m=1; m <= netPtr->layerPtr[k]->neuronPtr[n]->M; m++)
                    {
//                        Print the weight
                          fprintf(outputPtr,"%12.4g,  ",netPtr->layerPtr[k]-
>neuronPtr[n]->w[m]);
                    }
//                  End-for on inputs

                    fprintf(outputPtr,"\n");
             }
//           End-for on neurons
      }
//    End-for on layers

//    For each network output, n
      for (n=1; n <= netPtr->N; n++)
      {
//           Print its scale factor
             fprintf(outputPtr,"%12.4g,  ",netPtr->w0[n]);
             fprintf(outputPtr,"%12.4g,  ",netPtr->w1[n]);
      }
//    End-for on network outputs

      fprintf(outputPtr,"\n");

      fclose(outputPtr);

      return 0;
}
```

Function “push_down.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Function to push down history arrays
            Copyright 2007 Hugh S. Bruner
      */
int push_down(void)
{
//    Declarations
      int i,j;                      // Counter

//    In each history array, push each element one place deeper.
//    The last one is overwritten, no longer used.
//    for (i=4; i >=0; i--)
      for (i=4; i >0; i--)
      {
            j = i -1;
            ubd[i] = ubd[j];
            vbd[i] = vbd[j];
            wbd[i] = wbd[j];
            pbd[i] = pbd[j];
            qbd[i] = qbd[j];
            rbd[i] = rbd[j];
            phid[i] = phid[j];
            thetad[i] = thetad[j];
            psid[i] = psid[j];
            hd[i] = hd[j];
      }
      return 0;
}
/*          End of push down function
                        */
```

Function “RandomCom.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int randomCom(float *ucommand, float *gammacommand)
{
/*	Function to produce random commands for u and gamma (speed and flight
path angle perturbations)
	Copyright 2007 Hugh S. Bruner

*****************************************************
*
	*
*	ucommand = rand()*(.5/(s+.5))				*
*	gammacommand = rand()*(1/(s+1))				*
*
	*
*****************************************************
			*/

	float Agenu = (float).9900;
	float Agena = (float).9802;
	float Bgenu = (float).1250;
	float Bgena = (float).1250;
	float Cgenu = (float).0792;
	float Cgena = (float).1568;
	float Dgenu = (float).0050;
	float Dgena = (float).0099;
	double xnewu, inputu = 0., inputudot;
	double xnewa, inputa = 0., inputadot;
	static double xgenu = 0.;
	static double xgena = 0.;

	inputudot = (float)20.*((float)fmod(rand(),1000.) -
(float)500.)/(float)500.;  // Random commands between -20 and 20 fps
	inputadot = (float).175*((float)fmod(rand(),1000.) -
(float)500.)/(float)500.;	// Random commands between -10 and 10 degrees

	inputu += inputudot; //*(float).02;
	inputa += inputadot; //*(float).02;

	xnewu = Agenu*xgenu + Bgenu*inputu;
	*ucommand = (float)10.*(Cgenu*(float)xgenu + Dgenu*(float)inputu);

	xnewa = Agena*xgena + Bgena*inputa;
	*gammacommand = (float)10.*(Cgena*(float)xgena + Dgena*(float)inputa);

	xgenu = xnewu;
	xgena = xnewa;

	return 0;

}
```

Function “read_actions.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"



int read_actions(FILE *inputPtr, FILE *outputPtr, PointertoNetPtr *netPtrPtr,
char direct_object[],float xinputs[])
{
	/*		Copyright 2007 Hugh S. Bruner		*/
	int ierror;					// error code for return

#ifdef VERBOSE
	printf("\nIn read_actions:  Direct object is %s",direct_object);
#endif

//	If the direct object is "NEW"
	if (strcmp(direct_object,"NEW")==0)
	{
//		If the network pointer points to NULL, a new network must be
created
		if (*netPtrPtr == NULL)
		{
//			Initialize a new network
			ierror = initialize_network(netPtrPtr);
			if (error_handler(ierror,"return from initialize_network")
!= 0) return -99;
		}
//		End-if on NULL pointer
//		Create a new network using user-provided configuration data
		create_new__network(*netPtrPtr,inputPtr,xinputs);
#ifdef VERBOSE
	printf("\nIn read_actions, after create_new_network:  **netPtrPtr =
%p",*netPtrPtr);
#endif
//		Set error code to zero for success
		ierror = 0;
	}
//	Else, if the direct object is "EXISTING", read the net config and all
of its weights
	else if (strcmp(direct_object,"EXISTING")==0)
	{
//		If the network pointer points to a network already in use
(pointer not equal NULL)
		if (*netPtrPtr != NULL)
		{
//			Print an error message to stderr
			fprintf(stderr,"\nCannot load a user-specified
configuration into an existing network");
//			Set  error code to -2
			ierror = -2;
			fprintf(stderr,"\nIERROR = %d",ierror);
		}
//		Else,
		else
```

```
            {
//                Initialize a new network
                  ierror = initialize_network(netPtrPtr);
                  if (error_handler(ierror,"return from initialize_network")
!= 0) return -99;
//                Load data from the user describing an existing network
                  load_existing_network(*netPtrPtr,inputPtr,xinputs,TRUE);
//                Set error code to zero for success
                  ierror = 0;
            }
//          End-if on network in use
      }
//    Else, if the direct object is "EXISTING_NO_WEIGHTS", read the net
configuration only
      else if (strcmp(direct_object,"EXISTING_NO_WEIGHTS")==0)
      {
//          If the network pointer points to a network already in use
(pointer not equal NULL)
            if (*netPtrPtr != NULL)
            {
//                Print an error message to stderr
                  fprintf(stderr,"\nCannot load a user-specified
configuration into an existing network");
//                Set  error code to -2
                  ierror = -2;
                  fprintf(stderr,"\nIERROR = %d",ierror);
            }
//          Else,
            else
            {
//                Initialize a new network
                  ierror = initialize_network(netPtrPtr);
                  if (error_handler(ierror,"return from initialize_network")
!= 0) return -99;
//                Load data from the user describing an existing network
                  fprintf(outputPtr,"\n\t\tCall to load_existing_network");
                  load_existing_network(*netPtrPtr,inputPtr,xinputs,FALSE);
//                Set error code to zero for success
                  ierror = 0;
            }
//          End-if on network in use
      }
//    Else, if the direct object is "CODE", read the net config and all of
its weights from code
      else if (strcmp(direct_object,"CODE")==0)
      {
//          If the network pointer points to a network already in use
(pointer not equal NULL)
            if (*netPtrPtr != NULL)
            {
//                Print an error message to stderr
                  fprintf(stderr,"\nCannot load a user-specified
configuration into an existing network");
//                Set  error code to -2
                  ierror = -2;
                  fprintf(stderr,"\nIERROR = %d",ierror);
            }
```

```
//          Else,
            else
            {
//                Initialize a new network
                  ierror = initialize_network(netPtrPtr);
                  if (error_handler(ierror,"return from initialize_network")
!= 0) return -99;
//                Load data from code describing an existing network, found
in function initialize_by_code
                  fprintf(outputPtr,"\n\t\tLoading network configuration from
code.");
                  fprintf(stderr,"\nLoading network configuration from
code.");
                  initialize_by_code(*netPtrPtr,xinputs);
//                Set error code to zero for success
                  ierror = 0;
            }
//          End-if on network in use
      }
//    Else, if the direct object is "TRAININGDATA"
      else if (strcmp(direct_object,"TRAININGDATA") == 0)
      {
#ifdef VERBOSE
      printf("\nIn read_actions, inside TRAININGDATA clause:  **netPtrPtr =
%p",*netPtrPtr);
#endif
//          If the network pointer does not point to a network in use
(pointer not equal NULL)
            if (*netPtrPtr == NULL)
            {
//                Print an error message to stderr
                  fprintf(stderr,"\nCannot load training data for a non-
existant network");
//                Set  error code to -3
                  ierror = -3;
                  fprintf(stderr,"\nIERROR = %d",ierror);
            }
//          Else,
            else
            {
//                Read training data
#ifdef VERBOSE
      printf("\nIn read_actions, prior to read_training data:  **netPtrPtr =
%p",*netPtrPtr);
#endif
/*                read_training_data(inputPtr,outputPtr, *netPtrPtr);
                  print_training_data(outputPtr, *netPtrPtr);
                        */
//                Set error code to zero for success
                  ierror = 0;
            }
//          End-if on network in use
      }
//    Else, if the direct object is "OPTIONS"
      else if (strcmp(direct_object,"OPTIONS") == 0)
      {
//          Read options
```

```
            read_options(inputPtr, stdout);
            fprintf(stderr,"\n");
//          Set error code to zero for success
            ierror = 0;
      }
//    Else you don't recognize the direct object
      else
      {
//          Print an error message to stderr
            fprintf(stderr,"\nCannot recognize direct object in function
read_actions");
//          Set  error code to -4
            ierror = -4;
            fprintf(stderr,"\nIERROR = %d",ierror);
      }
//    End-if on direct object

//    Return error code to calling program

      return ierror;

}

/*          End of function read_actions
      */
```

Function “read_item.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

char delims[]=" ,;|\t\n";

int read_item(double *dvaluePtr, int *ivaluePtr, char *svaluePtr, FILE
*sourcePtr, char *last_recordPtr, int *tokenizedPtr)
/*
      Function to return a single value from the input file.  Depending
      on the data type, the value will be return to the address at
      dvaluePtr (for a double), ivaluePtr (for an integer), or
      svaluePtr (for a character/string).  The user must provide the
      correct pointer according to the data type expected.  The other
      data pointers must be set to NULL.

      Copyright 2007 Hugh S. Bruner

      Other arguments:

      *sourcPtr points to the data file
      *last_recordPtr points to the last data record previously read
      *tokenizedPtr points to an integer value indicating whether or
            not this string has been tokenized yet: 0 is FALSE, 1 is TRUE.

*/
{
      short int value_needed=TRUE;  //    Assume value_needed is true
      char *tokenPtr;
      char scratch[20];
      char valtype = '\0';
      short int errorcode=-10;

      if ((dvaluePtr == NULL) && (ivaluePtr == NULL) && (svaluePtr != NULL))
valtype = 's';
      else if ((dvaluePtr == NULL) && (svaluePtr == NULL) && (ivaluePtr !=
NULL)) valtype = 'i';
      else if ((ivaluePtr == NULL) && (svaluePtr == NULL) && (dvaluePtr !=
NULL)) valtype = 'd';
      else
      {
            printf("\nAmbiguous variable type request in function
read_item");
            printf("\nError code = -1");
            return -1;
      }

//    While value_needed
      while (value_needed)
      {
//          Get a string token pointer from the last record read
            if (!*tokenizedPtr)
            {
                  tokenPtr = strtok(last_recordPtr,delims);
                  *tokenizedPtr = TRUE;
```

```
            }
            else
            {
                  tokenPtr = strtok(NULL,delims);
            }
//          If token pointer is not null
            if (tokenPtr != NULL)
            {
//                Assign string at token pointer to a scratch string
                  strcpy(scratch,tokenPtr);
//                printf("\n*** Token returned is:%s:",scratch);
//                Convert the scratch string to an appropriate variable type,
//                and ssign the value to the destination variable
                  if (valtype == 'i') *ivaluePtr = atoi(scratch);
                  else if (valtype == 'd') *dvaluePtr = atof(scratch);
                  else if (valtype == 's') strcpy(svaluePtr,scratch);
                  else
                  {
                        printf("\nCannot determine variable type in
read_item");
                        printf("\nError code = -2");
                        return -2;
                  }

//                Value_needed is false
                  value_needed=FALSE;
//                Set return code for success
                  errorcode = 0;
            }
//          Else, token pointer is null
            else
            {
//                If not at the end of file
                  if (!feof(sourcePtr))
                  {
//                      Read a new record
                        fgets(last_recordPtr,READRECLEN,sourcePtr);

//                      If the first character of the line is a % sign,
ignore the record and read another
                        while(last_recordPtr[0]=='%')
                        {
                              fgets(last_recordPtr,READRECLEN,sourcePtr);
                        }

                        *tokenizedPtr = FALSE;
                  }
//                Else at end of file
                  else
                  {
//                      Print an error message
                        printf("\nUnexpected EOF encountered in read_item.");
                        printf("\nError code = -3");
//                      Set an error code and Return
                        return -3;
                  }
//                End-if on end-of-file
```

```
            }
//          End-if on token pointer is null
      }
//    End-while on value needed
//    Return value of return code to calling program
      return errorcode;
}
/*                End of function read_item
                  */
```

Function “read_options.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"


int read_options(FILE *inputPtr, FILE *outputPtr)
/*
      Function to read in program options to exercise during execution

      Written by Hugh S. Bruner, 01/14/2007
      Copyright 2007 Hugh S. Bruner

*/
{
      char variable[128];
      char operatorName[2];
      char newvalue[128];
      char *remaining;
      char data_record[READRECLEN];

//    Read a data record from the input file
      fgets(data_record,READRECLEN,inputPtr);
      remaining = data_record;

      while (!feof(inputPtr))
      {
//          Extract a verb, an operator, and a direct object from the record
            remaining =
extract_line(remaining,variable,operatorName,newvalue);


//          Parse for valid variable names
            if (strcmp(variable,"END_OPTIONS")==0)
            {
                  return 0;
            }
            else if (strcmp(variable,"PRINTFREQ")==0)
            {
                  control_printfreq = atoi(newvalue);
                  fprintf(outputPtr,"\n\tprintfreq = %d",control_printfreq);

            }
            else if (strcmp(variable,"PTRAINFREQ")==0)
            {
                  control_ptrainfreq = atoi(newvalue);
                  fprintf(outputPtr,"\n\tptrainfreq =
%d",control_ptrainfreq);

            }
            else if (strcmp(variable,"SIMFAILURE")==0)
            {
                  control_simfailure = atoi(newvalue);
            // use specific variable name here
                  fprintf(outputPtr,"\n\tsimfailure =
%d",control_simfailure);      // use specific variable name here
```

```
            }
            else        // you didn't find a valid variable name
            {
                  fprintf(outputPtr,"\n\tCannot recognize variable
%s",variable);
//                return -1;
            }


//    Read another record, if necessary

            if (remaining == NULL)
            {
                  fgets(data_record,READRECLEN,inputPtr);
                  remaining = data_record;
            }

      }



      return 0;

}
```

Function "seed_net.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"


int seed_net(Network *netPtr)
{
	/*		Copyright 2007 Hugh S. Bruner		*/
	short k;
	short m;
	short n;

	if (netPtr == NULL)
	{
		fprintf(stderr,"\nNetwork not yet defined in seed_net.");
		return -1;
	}


//	For each layer, k, above layer 1
	for (k=2; k <= netPtr->K; k++)
	{
//		For each neuron in the layer, n
		for (n=1; n <= netPtr->nk[k]; n++)
		{
//			For each weight, m
			for (m=0; m <= netPtr->layerPtr[k]->neuronPtr[n]->M; m++)
			{
//				Apply a random number between -.5 and .5
				netPtr->layerPtr[k]->neuronPtr[n]->w[m] = (float)(-.5
+ (float)(rand()%200)/200.);

			}
//			End-for on weights
#ifdef VERBOSE
			printf("\nSeed Net: ");
			dump_neuron(netPtr->layerPtr[k]->neuronPtr[n]);
#endif
		}
//		End-for on neurons
	}
//	End-for on layers

	return 0;
}

/*		End of function seed_net					*/
```

Function "simNN.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Aircraft simulation manager for a 6 degree of freedom flight
simulation        */
/*          Written by H.S. Bruner
            Copyright 2007 Hugh S. Bruner


                              */
int simNN(FILE *inputPtr, Network *myNetPtr, double const timemax, double
time_to_learn, double time_to_control)
{
//    Declarations

      BinaryRecord time_slice;                  // To catch data at each time
slice desired
      int printfreq = 1;                              // How many times
through the hot loop between data captures
      int printcount = 1;                             // Counter on cycles
between capturing print data
      static isims=0;                                 // Counter on how many
times the simulation has been run.

      const double dt=.02;                      // Time step per computation
cycle

      double sim_time;                          // simulation clock
      double cospsi,sinpsi;                     // sine and cosine of euler
angle psi
      double alpha1, de1, detab1;               // trim values for alpha, de,
and detab
      double vkeastrim;                         // trim values for equivalent
airspeed
      double pla1;                                    // trim value for power
lever angle
      double ucomm=0., gammacom=0.;       // commanded values of u and gamma
received from autopilot
      int manuever;                                   // code for manuever to
be performed by autopilot
      FILE *ffilePtr;                                 // File pointer for the
formatted output
      FILE *bfilePtr;                                 // File pointer for the
unformatted output
      FILE *btrainPtr;                          // File pointer for the
binary training data output
      FILE *ftrainPtr;                          // File pointer for the
formatted formatted training data output
      FILE *modeloutPtr;                              // File pointer for the
formatted linear pre-filter output
      char trainingfile[20];                    // String that contains the
file name for formatted training data
      char outputfilename[20];                  // String that contains the
file name for the formatted output file
```

```
	char case_descriptor[120];		// String that contains a
description of the case to be read

	char data_record[READRECLEN] = "";  // Used for tokenized read of data
	int tokenized = FALSE, ierror;		// Used for tokenized read of
data
	double xdouble;				// Used for tokenized
read of data
	int xinteger;				// Used for tokenized
read of data

	++isims;				// increment the simulation
counter

	printf("\n-----------------------------------------------");

//		Open files.  Stop if you cannot open the files -------------------
----------------------------------------

	sprintf(outputfilename,"output%02d.txt",isims);		//
Sythesize file name for formatted output.

		// Each successive case gets a new output file name

	if(((ffilePtr = fopen(outputfilename,"w"))==NULL) || ((bfilePtr =
fopen(SCRATCH,"wb")) == NULL))
	{
		printf("\nCannot open files");
		return -1;
	}
	else					// Carry on
		sprintf(trainingfile,"traindata%02d.csv",isims);
	{
		if (log_training_data ==1)		// open files to
capture training data if log_training_data == 1
		{
			if(((ftrainPtr = fopen(trainingfile,"w"))==NULL) ||
((btrainPtr = fopen(SCRATCH2,"wb")) == NULL))
			{
				printf("\nCannot open training files");
				return -1;
			}
		}
		printf("\nTraining data file opened as %s",trainingfile);

//		testoutPtr = fopen("testout.csv","w");

		if ((modeloutPtr=fopen("modelout.csv","w")) == NULL)
		{
			printf("\nCannot open model output file.");
			return -1;
		}


//		End of opening files ---------------------------------------------
----------------------------------------
```

```
//          Initialize the Aircraft data

            CJ144_data();

//          Read in the case descriptor string

            fgets(case_descriptor,sizeof(case_descriptor),inputPtr);
            printf("\n%s",case_descriptor);

//          Read in altitude, VKEAS, and manueuver code from input file

            if ((ierror =
read_item(&xdouble,NULL,NULL,inputPtr,data_record,&tokenized)) != 0) return
ierror;
            h = (float)xdouble;                         /*          Altitude
      */

            if ((ierror =
read_item(&xdouble,NULL,NULL,inputPtr,data_record,&tokenized)) != 0) return
ierror;
            vkeas = (float)xdouble;             /*          VKEAS
      */

            if ((ierror =
read_item(NULL,&xinteger,NULL,inputPtr,data_record,&tokenized)) != 0) return
ierror;
            manuever = xinteger;                /*          Manuever code
      */

            printf("\nVKEAS = %.1f, h = %.1f",vkeas,h);

            dx = (xcg - cgref)*cref;                                // shift in
c.g. from reference
            mass = weight/g;                                        // compute
mass
            idivisor = ixxb * izzb - ixzb * ixzb;           // this parameter
is used in function "fast",

      // but is constant throughout the simulation run

            clo = eval(&clov,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            cla = eval(&clav,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            clde = eval(&cldev,&alpha,&tcp);          // evaluate the value
of this parameter, considering alpha and tcp
            cldetab = eval(&cldetabv,&alpha,&tcp);    // evaluate the value
of this parameter, considering alpha and tcp
            clq = eval(&clqv,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            cladot = eval(&cladotv,&alpha,&tcp);      // evaluate the value
of this parameter, considering alpha and tcp

            cmo = eval(&cmov,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
```

```
            cma = eval(&cmav,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            cmde = eval(&cmdev,&alpha,&tcp);          // evaluate the value
of this parameter, considering alpha and tcp
            cmdetab = eval(&cmdetabv,&alpha,&tcp);    // evaluate the value
of this parameter, considering alpha and tcp
            cmq = eval(&cmqv,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            cmadot = eval(&cmadotv,&alpha,&tcp);      // evaluate the value
of this parameter, considering alpha and tcp

            cyb = eval(&cybv,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            cyp = eval(&cypv,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            cyr = eval(&cyrv,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            cydr = eval(&cydrv,&alpha,&tcp);          // evaluate the value
of this parameter, considering alpha and tcp
            cyda = eval(&cydav,&alpha,&tcp);          // evaluate the value
of this parameter, considering alpha and tcp

            cnb = eval(&cnbv,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            cnp = eval(&cnpv,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            cnr = eval(&cnrv,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            cndr = eval(&cndrv,&alpha,&tcp);          // evaluate the value
of this parameter, considering alpha and tcp
            cnda = eval(&cndav,&alpha,&tcp);          // evaluate the value
of this parameter, considering alpha and tcp

            crb = eval(&crbv,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            crp = eval(&crpv,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            crr = eval(&crrv,&alpha,&tcp);                  // evaluate the
value of this parameter, considering alpha and tcp
            crdr = eval(&crdrv,&alpha,&tcp);          // evaluate the value
of this parameter, considering alpha and tcp
            crda = eval(&crdav,&alpha,&tcp);          // evaluate the value
of this parameter, considering alpha and tcp

            rho = std_density(h);                           // compute the
density of air for this altitude
            vtrue = 1.689*vkeas/sqrt(rho/.0023769);   // compute true
airspeed, givn the KEAS for trim
            U1 = vtrue;                                           // U1
equals true airspeed
            q1 = .5*rho*vtrue*vtrue;                        // compute
dynamic pressure
            q1sw = q1*sref;                                       //
frequently repeated calculations.  Do now to save time later.
            q1swbw = q1sw*bref;                                   //
frequently repeated calculations.  Do now to save time later.
```

```
            q1swcw = q1sw*cref;                                    //
frequently repeated calculations.  Do now to save time later.


//          Trim the airplane for zero stick force;

            fetrim(&alpha1, &de1, &detab1, &pla1);    // Calculate the zero
stick-force trim condition
            htrim = h;                                             // capture
the trim conditions
            Utrim = U1;                                            // capture
the trim conditions
            vkeastrim = vkeas;                                     // capture
the trim conditions

//          Simulation variables

            sim_time = 0;
            initialize_sim();                                // This function
initializes variables associated with the sim

//
//          Write case data to formatted output file

            case_data_write(ffilePtr,case_descriptor);

//          Writing time = 0 output in binary form to a scratch file

            time_slice.alpha = alpha;
            time_slice.beta = beta;
            time_slice.da = da;
            time_slice.de = de;
            time_slice.detab = detab;
            time_slice.dr = dr;
            time_slice.pla = pla;
            time_slice.h = h;
            time_slice.phi = phi;
            time_slice.psi = psi;
            time_slice.theta = theta;
            time_slice.gamma = theta - alpha;
            time_slice.time = sim_time;
            time_slice.dde =  (double)0.;
            time_slice.dda =  (double)0.;
            time_slice.ddr =  (double)0.;
            time_slice.dpla = (double)0.;
            time_slice.vkeas = vkeas;
            time_slice.xearth = xearth;
            time_slice.yearth = yearth;
            time_slice.rmserror = (double)1.;
            time_slice.ucom = ucomm;
            time_slice.gammacom = gammacom;
            time_slice.vtrue = vtrue;

            fwrite(&time_slice,sizeof(BinaryRecord),1,bfilePtr);


//
```

```
// ===================================================
//          Start the hot loop here.
//          Function fast will be called five times during
//          each cycle through the hot loop.  Low-frequency
//          computations are interspersed between calls to
//          fast.
// ===================================================
//
//          While time is less than the maximum specified time

            start_time = time(NULL);
            myNetPtr->flag_input_errors = 1;

            while (sim_time <= timemax)
            {
//
//                Call the fast loop five times
//                Intersperse with updates on

                  fast(&sim_time,dt);                 // call 1

//                Autopilot calculations

                  autopilot(myNetPtr,&sim_time, dt, vkeastrim, U1, manuever,
de1, (double)0, (double)0, pla1,
                        &de, &da, &dr, &pla, &ucomm, &gammacom, btrainPtr,
ffilePtr, modeloutPtr, time_to_learn,time_to_control);

//                Update stability derivatives, considering current value for
alpha and tcp

                  clo = eval(&clov,&alpha,&tcp);
                  cla = eval(&clav,&alpha,&tcp);
                  clde = eval(&cldev,&alpha,&tcp);
                  cldetab = eval(&cldetabv,&alpha,&tcp);
                  clq = eval(&clqv,&alpha,&tcp);
                  cladot = eval(&cladotv,&alpha,&tcp);

                  cmo = eval(&cmov,&alpha,&tcp);
                  cma = eval(&cmav,&alpha,&tcp);
                  cmde = eval(&cmdev,&alpha,&tcp);
                  cmdetab = eval(&cmdetabv,&alpha,&tcp);
                  cmq = eval(&cmqv,&alpha,&tcp);
                  cmadot = eval(&cmadotv,&alpha,&tcp);

                  cyb = eval(&cybv,&alpha,&tcp);
                  cyp = eval(&cypv,&alpha,&tcp);
                  cyr = eval(&cyrv,&alpha,&tcp);
                  cydr = eval(&cydrv,&alpha,&tcp);
                  cyda = eval(&cydav,&alpha,&tcp);

                  cnb = eval(&cnbv,&alpha,&tcp);
                  cnp = eval(&cnpv,&alpha,&tcp);
                  cnr = eval(&cnrv,&alpha,&tcp);
                  cndr = eval(&cndrv,&alpha,&tcp);
                  cnda = eval(&cndav,&alpha,&tcp);
```

```
                    crb = eval(&crbv,&alpha,&tcp);
                    crp = eval(&crpv,&alpha,&tcp);
                    crr = eval(&crrv,&alpha,&tcp);
                    crdr = eval(&crdrv,&alpha,&tcp);
                    crda = eval(&crdav,&alpha,&tcp);

//                  if there is a failure, it is introduced here...

                    if(control_simfailure) execute_failure(sim_time, manuever);

                    fast(&sim_time,dt);                 // call 2

//                  Autopilot calculations

                    autopilot(myNetPtr,&sim_time, dt, vkeastrim, U1, manuever,
de1, (double)0, (double)0, pla1,
                          &de, &da, &dr, &pla, &ucomm, &gammacom, btrainPtr,
ffilePtr, modeloutPtr, time_to_learn,time_to_control);

//                  Atmospheric properties

                    rho = std_density(h);
                    vtrue = sqrt(ub*ub + vb*vb + wb*wb);
                    q1 = .5*rho*vtrue*vtrue;
                    q1sw = q1*sref;
                    q1swbw = q1sw*bref;
                    q1swcw = q1sw*cref;
                    vkeas = sqrt(q1*295.);

                    fast(&sim_time,dt);                 // call 3

//                  Autopilot calculations

                    autopilot(myNetPtr,&sim_time, dt, vkeastrim, U1, manuever,
de1, (double)0, (double)0, pla1,
                          &de, &da, &dr, &pla, &ucomm, &gammacom, btrainPtr,
ffilePtr, modeloutPtr, time_to_learn,time_to_control);

//                  Earth coordinates

                    cospsi = cos(psi);
                    sinpsi = sin(psi);
                    xearth = xearth + (cospsi * costh * ub + (-sinpsi * cosphi
+ sinphi * cospsi * sinth) * vb
                          + (sinpsi * sinphi + cospsi * cosphi * sinth) * wb) *
dt;
                    yearth = yearth + (sinpsi * costh * ub + (cospsi * cosphi +
sinphi * sinpsi * sinth) * vb
                          + (-cospsi * sinphi + sinpsi * sinth * cosphi) * wb)
* dt;

                    fast(&sim_time,dt);                 // call 4

//                  Autopilot calculations

                    autopilot(myNetPtr,&sim_time, dt, vkeastrim, U1, manuever,
de1, (double)0, (double)0, pla1,
```

```
                        &de, &da, &dr, &pla, &ucomm, &gammacom, btrainPtr,
ffilePtr, modeloutPtr, time_to_learn,time_to_control);

                  fast(&sim_time,dt);                 // call 5

//                Autopilot calculations

                  autopilot(myNetPtr,&sim_time, dt, vkeastrim, U1, manuever,
de1, (double)0, (double)0, pla1,
                        &de, &da, &dr, &pla, &ucomm, &gammacom, btrainPtr,
ffilePtr, modeloutPtr, time_to_learn,time_to_control);

//                Writing summary output in binary form to a scratch file,
but only at proper frequency

                  if (printcount++ >= control_printfreq)
                  {
                        printcount = 1;

                        time_slice.alpha = alpha;
                        time_slice.beta = beta;
                        time_slice.da = da;
                        time_slice.de = de;
                        time_slice.detab = detab;
                        time_slice.dr = dr;
                        time_slice.pla = pla;
                        time_slice.dde = de - de1;
                        time_slice.dda = da;
                        time_slice.ddr = dr;
                        time_slice.dpla = pla - pla1;
                        time_slice.h = h;
                        time_slice.phi = phi;
                        time_slice.psi = psi;
                        time_slice.theta = theta;
                        time_slice.gamma = theta - alpha;
                        time_slice.time = sim_time;
                        time_slice.vkeas = vkeas;
                        time_slice.xearth = xearth;
                        time_slice.yearth = yearth;
                        time_slice.rmserror = sqrt(nnrms);
                        time_slice.ucom = ucomm;
                        time_slice.gammacom = gammacom;
                        time_slice.vtrue = vtrue;

                        fwrite(&time_slice,sizeof(BinaryRecord),1,bfilePtr);
                  }

//          End while loop
            }
//
// ====================================================
//          Hot loop ends here
// ====================================================

            stop_time = time(NULL);                                   //
capture the stop time for the simulation
```

```
            elapsed_time = stop_time - start_time;                //
calculate the elapsed time for the run
            fprintf(ffilePtr,"\n      %d seconds run-time\n",elapsed_time);
      // write elapsed time to output file

//          Close the binary file for writing

            fclose(bfilePtr);

//          Use data from the scratch file to create a neatly formatted
output file

            format_files(ffilePtr, isims, case_descriptor);

//          Close the formatted file now

            fclose(ffilePtr);
            fclose(modeloutPtr);

//          If training data was requested...

            if (log_training_data == 1)
            {
                  fclose(btrainPtr);
                  // close the binary training file
                  fprintf(ftrainPtr,"%ch = %.1f Vkeas =
%.1f\n",'\45',htrim,vkeastrim);
                  format_training_data(ftrainPtr, control_ptrainfreq);  //
translate the binary data to a formatted file
                  fclose(ftrainPtr);
                  // close the formatted file
            }
            else
            {
                  fprintf(stdout,"\nNo training data requested");
            }

//          I do believe we are done!

            return 0;

      }                                  // End of else statement on file
open logic
//    End program

}

/*          End of main program
                        */
```

Function “simulation_timer.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Timer function
            Copyright 2007 Hugh S. Bruner
                                  */
int simulation_timer(int wait_millisec)
{
//    declarations
      int sec1000;
      int msec, mytime;
      int timepassed;
      static int lasttime=0;

//    declare a variable of type struct _timeb
      static struct _timeb timebuffer;
//    If first pass through, calculate lasttime for first time.
      if (lasttime == 0)
      {
//          Get system time
            _ftime(&timebuffer);
//          Multiply milliseconds by 1000
            msec = timebuffer.millitm;

//          Get units value from seconds and mulitply by 1000
            sec1000 = (timebuffer.time%10)*1000;

//          Add these two together
            lasttime = sec1000+msec;
      }

      timepassed = 0;
      while (timepassed < wait_millisec)
      {

//          Get system time
            _ftime(&timebuffer);
//          Multiply milliseconds by 1000
            msec = timebuffer.millitm;

//          Get units value from seconds and mulitply by 1000
            sec1000 = (timebuffer.time%10)*1000;

//          Add these two together
            mytime = sec1000+msec;

            timepassed = mytime - lasttime;
            if (timepassed < 0) timepassed += 10000;

//    End-while
      }
//    Reset lasttime
      lasttime = mytime;
//    Quit.
```

```
    return 0;
}
```

Function "solve3x3.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*          Function to solve [A]x = y for the unknown vector x
            using Cramer's rule
            Copyright 2007 Hugh S. Bruner
                                                */
int solve3by3(double a[3][3], double x[3], double y[3])
{
//    Declarations
      double detA, det0, det1, det2;      // computed determinants

//    Solve the determinant of A
      detA = a[0][0]*a[1][1]*a[2][2] + a[0][1]*a[1][2]*a[2][0] +
a[0][2]*a[1][0]*a[2][1];
      detA = detA -(a[2][0]*a[1][1]*a[0][2] + a[2][1]*a[1][2]*a[0][0] +
a[2][2]*a[1][0]*a[0][1]);

      if (detA !=0 )
      {

//          Solve for the determinant of augmented matrix 0
            det0 = y[0]*a[1][1]*a[2][2] + a[0][1]*a[1][2]*y[2] +
a[0][2]*y[1]*a[2][1];
            det0 = det0 -(y[2]*a[1][1]*a[0][2] + a[2][1]*a[1][2]*y[0] +
a[2][2]*y[1]*a[0][1]);

//          Solve for the determinant of augmented matrix 1
            det1 = a[0][0]*y[1]*a[2][2] + y[0]*a[1][2]*a[2][0] +
a[0][2]*a[1][0]*y[2];
            det1 = det1 -(a[2][0]*y[1]*a[0][2] + y[2]*a[1][2]*a[0][0] +
a[2][2]*a[1][0]*y[0]);

//          Solve for the determinant of augmented matrix 2
            det2 = a[0][0]*a[1][1]*y[2] + a[0][1]*y[1]*a[2][0] +
y[0]*a[1][0]*a[2][1];
            det2 = det2 -(a[2][0]*a[1][1]*y[0] + a[2][1]*y[1]*a[0][0] +
y[2]*a[1][0]*a[0][1]);

//          x0 = det 0/ det A

            x[0] = det0/detA;

//          x1 = det 1/ det A
            x[1] = det1/detA;

//          x2 = det 2/ det A
            x[2] = det2/detA;

            return 0;
      }
      else                                // You get here if the determinant
of A was zero.
      {
```

```
            printf("\nDeterminant of A is zero in function solve3by3.");
            return -1;
      }
//    That's all!
}
/*                End of function solve3by3
                                */
```

Function "std_atmos.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*    Function to compute standard temperature for a specified altitude
      */
/*               Copyright 2007 Hugh S. Bruner                         */
/*
      Reference:  Standard atmosphere - tables and data for altitudes to
65,800 feet,
                        NACA Report 1235, 1955


                                    */
double std_temp(double h)
{
      double k1=6.87535e-6;
      double temp;
//    If h is below the tropopause (36,089 feet) then
      if (h < 36089.) temp = 518.6*(1-k1*h);
//    else use the high altitude equation
      else temp = 389.6;
//    return the calculated value
      return temp;
}


/*    Function to compute standard pressure for a specified altitude
      */
/*
      Reference:  Standard atmosphere - tables and data for altitudes to
65,800 feet,
                        NACA Report 1235, 1955


                                    */
double std_pressure(double h)
{
      double k1=6.87535e-6;
      double k2=4.80634e-5;
      double pressure;
//    If h is below the tropopause (36,089 feet) then
      if (h < 36089.) pressure = 2116.22*pow((1-k1*h),5.2561);
//    else use the high altitude equation
      else pressure=472.6702*exp(-k2*(h-36089));
//    return the calculated value
      return pressure;
}


/*    Function to compute standard density for a specified altitude
      */
/*
      Reference:  Standard atmosphere - tables and data for altitudes to
65,800 feet,
```

```
                                    */
double std_density(double h)
{
      double k1=6.87535e-6;
      double k2=4.80634e-5;
      double density;
//    If h is below the tropopause (36,089 feet) then
      if (h < 36089.) density = .0023769*pow((1-k1*h),4.2561);
//    else use the high altitude equation
      else density=.00070611*exp(-k2*(h-36089));
//    return the calculated value
      return density;
}
```

Function “thrust_model.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

/*                  Thrust model

                    Copyright 2007 Hugh S. Bruner


                                */
double thrust_model(double PLA, double vtrue_fps, double air_density)
{
//      Declarations
        double eta_prop, max_power, thrust;

//      Compute prop efficiency
//              (This can be a more complex calculation later)
        eta_prop = .85;                                         // assumption

//      Compute max_power available
//              (This can be a more complex calculation later)
        max_power = 285.*(air_density/.0023769);        // SL
horsepower*(density ratio)

//      Compute thrust
        thrust = eta_prop*(double)550.*max_power*(PLA/(double)100.)/vtrue_fps;
        // pounds

//      return thrust to calling program
        return thrust;
}
/*                  End of thrust model
                          */
```

Function "update_inv_state_space.cpp"

```
#include "stdafx.h"
#include "MyDefinitions.h"
#include "MyDeclarations.h"

int update_inv_state_space(double Ainv[4][4], double Binv[4][2],
                                        double Cinv[2][4], double Dinv[2][2],
                                        float Uref, double dt, FILE
*logfilePtr)
/*
            Function to update the discrete state space representaton of the
inverse
            transfer function based on Uref.  THIS MODEL IS SPECIFIC TO
SAMPLE
            AIRCRAFT. Curve fits are utilized to minimize computation time.
            This routine should only be applied to general aircraft with
caution.

            Constants used in series expansions can be found in
InvDiscCoefficients.xls.
            These constants were derived by curve fitting the coefficients in
the
            state space matrices calculated in MATLAB.

            Created by Hugh S. Bruner
                              01/13/2007
            Copyright 2007 Hugh S. Bruner


*/
{
/*
      char descriptor[] = "Approximate Inverse Controller";
      double coeffs[180] = {  -4.09086E-10,     3.01894E-07,
      0.999916044,      0.011726051,      -0.87467867,
      -1.93131E-09,     1.84261E-06,      -0.001053653,     -0.769244884,
      43.63712413,
      -3.58591E-08,     3.05941E-05,      -0.010237651,     1.739864736,
      -48.89250525,
      -5.10593E-09,     4.97362E-06,      -0.001864518,     0.324071964,
      -22.42413944,

      -9.39501E-10,     7.54595E-07,      -0.000226764,     0.030506258,
      -1.583702151,
      8.06525E-09,      -4.98814E-06,     1.000363276,      0.111782992,
      -31.86099054,
      -4.18489E-10,     3.24841E-07,      -9.31975E-05,     0.011741644,
      -0.549675113,
      -2.33897E-08,     1.84238E-05,      -0.005392299,     0.698933474,
      -34.16715201,

      -3.52681E-08,     3.00937E-05,      -0.010072865,     1.712943897,
      -48.02704988,
      3.40111E-06,      0.015482351,      -1.274737302,     59.70334751,
      -1606.159753,
```

```
1.94342E-06,      -0.001927679,     -0.185771206,     66.55999093,
-1655.130196,
-9.11762E-07,     0.000708776,      -0.200159962,     20.28805436,
95.92485517,

-3.52681E-10,     3.00937E-07,      -0.000100729,     0.017129439,
-0.480270499,
3.40111E-08,      0.000154824,      -0.012747373,     0.597033475,
-16.06159753,
1.94245E-08,      -1.92703E-05,     0.008140766,      0.665747387,
-16.55611886,
-1.75584E-08,     1.29111E-05,      0.996546499,      0.358191636,
-5.035655039,


2.68231E-10,      -1.94295E-07,     -0.999944325,     -0.007873919,
0.538191281,
2.90171E-08,      1.26541E-05,      -0.033613217,     -29.83748628,
1460.289464,

4.69749E-10,      -3.77296E-07,     0.000113382,      -0.015253098,
0.791849949,
8.09309E-09,      -7.04237E-06,     -0.997401277,     -0.413720457,
33.15757478,

1.76351E-08,      -1.50476E-05,     0.005036652,      -0.856499023,
24.01471298,
9.64408E-05,      -0.104777537,     -58.09263113,     3248.455214,
-80491.81989,

1.76351E-10,      -1.50476E-07,     5.03665E-05,      -0.00856499,
0.24014713,
9.64408E-07,      -0.001047775,     -0.580926311,     32.48455214,
-804.9181989,


2.39358E-08,      -1.98711E-05,     0.006314563,      -0.941228671,
60.37331964,
5.45087E-07,      -0.000505754,     0.217432246,      -3.876508453,
643.8932847,
2.22017E-07,      -0.00021794,      0.090931658,      -21.92177077,
-164.3586712,
8.7951E-07, -0.000715776,     0.219657264,      -30.58673301,
1669.650217,

-1.36658E-09,     -3.64726E-07,     8.28019E-05,      -0.008259727,
-1.680342435,
-5.11367E-08,     0.000143908,      -0.000824977,     0.062930077,
-1.98814302,
1.64029E-09,      -1.40394E-06,     -0.000225732,     -0.001767297,
0.723967273,
2.55338E-08,      -0.000563057,     0.00752846, -1.322424088,
101.1694261,


-1.24021E-08,     1.02096E-05,      -0.00322186,      0.477345631,
-30.44418306,
```

```
	1.0147E-05, -0.010092857,	4.267611697,	-1126.22127,	-
9565.023116,

	1.26374E-09,	0.000840301,	6.27144E-05,	-0.006716752,
	1.236874941,
	5.91515E-08,	0.000163737,	-0.021027418,	1.240883017,
	-40.24734851};
*/

	char descriptor[] = "Exact Inverse Controller";
	double coeffs[180] = {  1.62509E-14,	-4.17089E-12,
	0.999999999,	1.30152E-07,	-4.05312E-06,
	0,	0,	0,	0,	0,
	0,	0,	0,	0,	0,
	0,	0,	0,	0,	0,
	0,	0,	0,	0,	0,
	1.62509E-14,	-4.17089E-12,	0.999999999,	1.30152E-07,
	-4.05312E-06,
	0,	0,	0,	0,	0,
	0,	0,	0,	0,	0,
	-6.50366E-09,	6.61267E-06,	-0.002858218,	0.711901979,
	6.590019921,
	2.46016E-06,	0.016227634,	-1.450968276,	78.73588788,
	-1480.333298,
	1.95961E-06,	-0.001940283,	-0.182138109,	66.09932735,
	-1633.48192,
	-1.10656E-07,	7.07967E-05,	-0.010730948,	-4.751746248,
	1354.799095,
	-2.8675E-11,	5.90379E-08,	-2.97772E-05,	0.007481462,
	0.043049896,
	2.35709E-08,	0.000162987,	-0.014687905,	0.806449798,
	-15.54083884,
	1.9417E-08, -1.92821E-05,	0.008149133,	0.664042722,	-
16.44616649,
	-1.87851E-09,	1.29513E-06,	0.999733972,	-0.029467766,
	12.8196868,
	-1.62509E-14,	4.17089E-12,	-0.999999999,	-1.30152E-07,
	4.05312E-06,
	0,	0,	0,	0,	0,
	0,	0,	0,	0,	0,
	-1.62509E-14,	4.17089E-12,	-0.999999999,	-1.30152E-07,
	4.05312E-06,
	3.52613E-09,	-3.48242E-06,	0.001471937,	-0.360590366,
	-3.110346617,
	9.63602E-05,	-0.10472806,	-58.1248125,	3254.154347,
	-81274.9572,
	-8.94547E-11,	6.67899E-08,	-1.42535E-05,	-7.90274E-05,
	-0.187076916,
	9.63899E-07,	-0.001047449,	-0.581214892,	32.53885814,
	-812.6784882,
	1.63227E-08,	-1.39208E-05,	0.004577177,	-0.714667496,
	49.05236514,
	5.3555E-07, -0.000483242,	0.203505211,	-0.775477582,
	322.6146011,
	2.19889E-07,	-0.000216167,	0.090389311,	-21.8495734,
	-167.8942146,
```

```
	7.17592E-07,	-0.000588718,	0.182681283,	-25.83623398,
	1440.679326,
	-1.36658E-09,	-3.64726E-07,	8.28019E-05,	-0.008259727,
	-1.680342435,
	-4.98586E-08,	0.000142961,	-0.000594247,	0.037486329,
	-1.03058525,
	3.1821E-10, -4.87786E-07,	-0.000455367,	0.022842765,	-
0.214835366,
	2.62107E-08,	-0.000563567,	0.007667518,	-1.338409978,
	101.8109401,
	-8.80425E-09,	7.39699E-06,	-0.002394439,	0.368138214,
	-24.91587315,
	1.02342E-05,	-0.010168832,	4.293409217,	-1130.192209,
	-9287.599511,
	1.26374E-09,	0.000840301,	6.27144E-05,	-0.006716752,
	1.236874941,
	6.03474E-08,	0.000162955,	-0.020849729,	1.224161246,
	-39.84012906};



	short i,j;		// counters
	double u2;		// Contains the value of Uref squared
	double u1;		// Contains the value of Uref
	double u_1;		// Contains the value of one over Uref
	double u_2;		// Contains the value of one over Uref squared

	double *arrayPtr, *coeffPtr;		// Pointers for manuevering
through this mess
									// Ainv has
16 entries
									// Binv has
8 entries
									// Cinv has
8 entries
									// Dinv has
4 entries

	printf("\n%s",descriptor);

	if (Uref > 170.)
	{
	u1 = (double)Uref;
	}
	else
	{
		u1 = (double)Uref;  //170.;
	}
	u2 = u1*u1;
	u_1 = 1./u1;
	u_2 = u_1*u_1;



//	Do Ainv first

	arrayPtr = &Ainv[0][0];
```

```
	coeffPtr = &coeffs[0];

	for (i=0; i<16; i++)
	{
		*arrayPtr++ = *coeffPtr*u2 + *(coeffPtr+1) *u1 + *(coeffPtr+2) +
*(coeffPtr+3) *u_1 + *(coeffPtr+4) *u_2;
		coeffPtr = coeffPtr + 5;
	}

//	Do Binv next

	arrayPtr = &Binv[0][0];		// note that coeffPtr is already
positioned correctly

	for (i=0; i<8; i++)
	{
		*arrayPtr++ = *coeffPtr*u2 + *(coeffPtr+1) *u1 + *(coeffPtr+2) +
*(coeffPtr+3) *u_1 + *(coeffPtr+4) *u_2;
		coeffPtr = coeffPtr + 5;
	}

//	Do Cinv next

	arrayPtr = &Cinv[0][0];		// note that coeffPtr is already
positioned correctly

	for (i=0; i<8; i++)
	{
		*arrayPtr++ = *coeffPtr*u2 + *(coeffPtr+1) *u1 + *(coeffPtr+2) +
*(coeffPtr+3) *u_1 + *(coeffPtr+4) *u_2;
		coeffPtr = coeffPtr + 5;
	}

//	Do Dinv next

	arrayPtr = &Dinv[0][0];		// note that coeffPtr is already
positioned correctly

	for (i=0; i<4; i++)
	{
		*arrayPtr++ = *coeffPtr*u2 + *(coeffPtr+1) *u1 + *(coeffPtr+2) +
*(coeffPtr+3) *u_1 + *(coeffPtr+4) *u_2;
		coeffPtr = coeffPtr + 5;
	}



	return 0;

}
```

Function “write_out_training.cpp”

```
#include "stdafx.h"
#include "MyDefinitions.h"


int write_out_training(TrainingPair myTrainingData, FILE *filePtr)
{
	/*                Copyright 2007 Hugh S. Bruner             */
	short int i;
#ifdef DEBUG
	fprintf(stderr,"\nEntering write_out_training");
#endif

	for (i=0; i<=28; i++)
	{
		fprintf(filePtr,"%.6g,",myTrainingData.input[i]);
	}
	for (i=0; i<=1;
	i++)
	{
		fprintf(filePtr,"%.6g,",myTrainingData.output[i]);
	}
	fprintf(filePtr,"\n");

	return 0;

}
```