# CONSTRAINED LEARNING IN NEURAL CONTROL SYSTEMS

by

Mark A. Jensenius

Department of Mechanical Engineering and Materials Science
Duke University

Date: ______________________________

Approved:

______________________________
Dr. Silvia Ferrari, Advisor

______________________________
Dr. Henri P. Gavin

______________________________
Dr. Ronald Parr

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in the Department of Mechanical Engineering and Materials Science
in the Graduate School of
Duke University

2005

# Contents

# List of Tables

# List of Figures

# Nomenclature

| | |
|---|---|
| $n_h$ | The number of hidden layer nodes in the neural network |
| $n_o$ | The number of outputs of the neural network |
| $n_r$ | The number of regular inputs to the neural network |
| $n_m$ | The number of independent models used to describe the system |
| $n_p$ | Number of design points |
| $p$ | Aircraft body-axis roll rate |
| $q$ | Aircraft body-axis pitch rate |
| $r$ | Aircraft body-axis yaw rate |
| $u$ | Forward component of aircraft velocity |
| $V$ | Total Aircraft velocity |
| $v$ | Side component of aircraft velocity |
| $w$ | Downward component of aircraft velocity |
| $x_b$ | Aircraft x body-axis |
| $x_r$ | Inertial x axis |
| $y_b$ | Aircraft y body-axis |
| $y_r$ | Inertial y axis |
| $z_b$ | Aircraft z body-axis |
| $z_r$ | Inertial z axis |
| $\alpha$ | Aircraft angle of attack |
| $\beta$ | Aircraft sideslip angle |
| $\gamma$ | Aircraft path angle |
| $\mu$ | Aircraft bank angle |
| $\phi$ | Aircraft Euler roll angle |
| $\psi$ | Aircraft Euler yaw angle |

| | |
|---|---|
| $\theta$ | Aircraft Euler pitch angle |
| $\mathbf{a}$ | Auxiliary input vector |
| $\tilde{\mathbf{a}}$ | Scheduling vector |
| $\mathbf{b}$ | Neural network output bias vector |
| $\mathbf{e}$ | Neural network error vector |
| $\mathbf{p}$ | Neural network input vector |
| $\mathbf{u}$ | Control vector |
| $\mathbf{u}_c$ | Trim control values for the commanded state |
| $\tilde{\mathbf{u}}$ | Deviation from the trim control values for the commanded state |
| $\mathbf{x}$ | State vector |
| $\mathbf{x}_a$ | Augmented state |
| $\mathbf{x}_c$ | Commanded state |
| $\tilde{\mathbf{x}}$ | Deviation from commanded state |
| $\mathbf{y}$ | Neural network target output |
| $\mathbf{y}_c$ | Commanded plant output |
| $\mathbf{y}_s$ | System output vector |
| $\mathbf{z}$ | Neural network output vector |
| $\mathbf{z}_A$ | Neural network output vector for the action network |
| $\mathbf{z}_{adj}$ | Neural network output adjustment vector |
| $\mathbf{z}_C$ | Neural network output vector for the critic network |
| $H_\theta$ | The hidden layer node group corresponding to the $\theta^{th}$ output. |
| $I_m$ | The input group corresponding to the $m^{th}$ model. |
| $M_m$ | The hidden layer node group corresponding to the $m^{th}$ model. |
| $\Theta_m$ | The output group corresponding to the $m^{th}$ model. |
| $\mathbf{A}$ | Auxiliary input matrix |
| $\mathbf{B}$ | Output bias matrix |

$\mathbf{C}$ Linear control gain matrix

$\mathbf{D}_k$ The target gradients of the neural network outputs with respect to the state deviation vector

$\bar{\mathbf{E}}$ Matrix of target neural network output gradient

$\mathbf{E}$ Rank-3 tensor of target neural network output gradient

$\bar{\mathbf{E}}_m$ Matrix of neural network output gradient

$\mathbf{F}$ State Jacobian matrix of a linear dynamic system

$\mathbf{F}_m$ Ideal model state jacobian

$\mathbf{G}$ Control Jacobian matrix of a linear dynamic system

$\bar{\mathbf{K}}_m$ Matrix of neural network output gradient

$\bar{\mathbf{K}}_m^c$ Matrix of neural network output gradient

$\mathbf{M}$ Weighting of the cross-coupled states and controls in the cost function

$\mathbf{M}_a$ Weighting of the cross-coupled augmented states and control deviations in the cost function

$\mathbf{P}$ Riccati Matrix

$\mathbf{Q}$ Weighting of the state vector in the cost function

$\mathbf{Q}_a$ Weighting of the augmented state vector in the cost function

$\mathbf{R}$ Weighting of the control vector in the cost function

$\mathbf{R}_a$ Weighting of the control deviation vector in the cost function

$\mathbf{S}$ Output of the hidden layer nodes at all design points

$\mathbf{S}_\theta$ Output of the hidden layer nodes at all design points associated with $H_\theta$

$\mathbf{S}_\theta^c$ Output of the hidden layer nodes at all design points associated with $H_\theta^c$

$\mathbf{V}$ Output weight matrix

$\mathbf{V}_\theta$ Constrained output weights associated with the $\theta^{th}$ output

$\mathbf{V}_\theta^c$ Unconstrained output weights associated with the $\theta^{th}$ output

$\mathbf{W}$ Input weight matrix

| | |
|---|---|
| $\mathbf{W}_{\mathbf{A}}$ | Auxiliary input weight matrix |
| $\mathbf{W}_m$ | Constrained input weights associated with the $m^{th}$ model |
| $\mathbf{W}_m^c$ | Unconstrained input weights associated with the $m^{th}$ model |
| $\mathbf{W}_{\mathbf{R}}$ | Regular input weight matrix |
| $\mathbf{Z}$ | Neural network output matrix |
| $\bar{\mathbf{K}}$ | See , page 28 |
| $\dot{\mathbf{Z}}$ | Rank-3 tensor of neural network output gradient |
| $\boldsymbol{\Sigma}$ | Diagonal matrix of the hidden layer nodes' derivatives |
| $\boldsymbol{\Sigma}_{adj}$ | Diagonal matrix of the output adjustment's hidden layer nodes' derivatives |
| $H$ | Hamiltonian |
| $J$ | Cost function |
| $L$ | Legrangian |
| $\mathbf{NN}_A$ | Vector output mapping by the action neural network |
| $\mathbf{NN}_C$ | Vector output mapping by the critic neural network |
| $V$ | Value function |
| $\sigma$ | Neural network hidden layer sigmoidal function |
| $(\ )^c$ | Compliment of the indicated set |
| $(\ )_e$ | Equilibrium value |
| $(\ )_L$ | Related to the longitudinal model |
| $(\ )_{LD}$ | Related to the lateral-directional model |
| $(\dot{\ })$ | Derivative with respect to time |
| $(\ )_{(i,j,\ldots)}$ | Element of the indicated matrix at coordinates $(i, j, \ldots)$ |
| $(\ )^{-1}$ | Inverse |
| $(\ )^*$ | Optimal value |
| $(\ )_k$ | Variable evaluated at the $k^{th}$ design point or after the $k^{th}$ time interval |

| | |
|---|---|
| $(\ )^{(I,H,\Theta)}$ | Submatrix of the indicated matrix which corresponds to input group $I$, hidden layer node group $H$, and output group $\Theta$ |
| $(\ )^T$ | Transpose |
| $(\breve{\ })$ | Unconstrained error gradient with respect to the indicated matrix |
| $\Delta(\ )$ | Deviation from nominal value, $(\ ) - (\ )_e$ |
| $\hat{(}\ )$ | Constrained gradient matrix with respect to ( ) |
| $\breve{(}\ )$ | Unconstrained gradient matrix with respect to ( ) |
| $*$ | All available |

# Chapter 1

# Introduction

As science progress in today's technologically advanced world, there grows a need for improved performance and reliability in a wide range of devices. At the same time, these devices continue to become more complex and nonlinear in their behavior. There is a need to adjust the operation of these devices such that they perform optimally. A collection of components that act on a system to maintain the system's performance close to a desired set of performance specifications is called a *control system*. The system that is being controlled is often referred to as a *plant*. Control systems can be mechanical, electrical, computational, or even human in nature. They can be comprised of sensors, actuators, and other components that act together to cause the plant to produce some desired output. These control systems can be static in nature or they can adapt online to changes in the plant or the environment. A control system which does change its internal parameters online is called *adaptive* and can usually accommodate situations of greater uncertainty than a static controller could accommodate. A neural controller is a controller whose computations involve neural networks. The optimal linear control law for a particular plant can be determined at several equilibrium points by linearizing and applying linear control theory to the linearized dynamics equations. A controller is called a gain-scheduled controller when it uses the control law defined for each equilibrium point and it interpolates the control law when the system is between the equilibrium points.

The objective of this research is to show that an adaptive neural controller that is applicable to plants described by nonlinear ordinary differential equations can be designed and optimized online such that its performance improves while retaining the properties of a gain-scheduled design. Neural networks are particularly well-suited for approximating nonlinear, high-dimensional mappings through training. Neural control has been validated through many recent publications, such as [1], [2], and [3]. Recent advances in neural control have shown that it is possible to build a neural controller so that, prior to training, it operates as a gain-scheduled controller [4]. Taking this idea one step further, a neural network training algorithm will be presented which optimizes the neural controller's performance while satisfying the gain-scheduled requirements. The end result is a gain-scheduled controller that can optimize its interpolation and extrapolation abilities. This research also presents a streamlined approach to building such a controller, a new method for uniformly distributing neural network weights called *hyperspherical initialization*, and a matrix operation useful in obtaining derivatives in linear-algebra that is referred to as the *gradient transformation*.

## 1.1 Background and Motivation

Nonlinear multivariate systems with constraints, such as control specifications, are some of the most difficult systems to control optimally. Dynamic programming methods have been proposed for producing optimal control in these cases [5]. Dynamic programming is a direct-search method which guarantees optimal performance [6]. Approximate dynamic programming (ADP) tends to converge to an optimal or suboptimal solutions and do so with a much lower computational burden. ADP techniques require the use of parametric structures to approximate unknown nonlinear mappings over high dimensional compact spaces. Additionally, these parametric structures must have the ability to adapt and, ultimately, converge upon the unknown mappings. Neural networks, in this case, are ideal parametric structures for several reasons. Primarily, neural networks have the ability, given a sufficient number of parameters, to approximate any function [7]. Additionally, neural networks have the ability to learn in batch or incremental modes. Batch learning is beneficial in offline training when many training sets are available. Training sets include paired input and target output values. Sometimes target network gradients are also provided for each input value. Incremental training involved updating the neural network parameters based on a single training set. This frequently occurs when the neural network is operating online and training sets are generated sequentially.

### 1.1.1 Dynamic Programming and Approximate Dynamic Programming

The fundamental concept in dynamic programming is the *principle of optimality.* Consider the problem where a person needs to travel from point $a$ to point $c$ in figure 1.1.1. Every path the person could take involves a certain amount of effort. The dynamic programming, this mapping of path choice to effort is referred to as a cost function. For a path to be considered optimal, it must minimize the cost incurred. Now, suppose that the person has identified the path which requires the least effort, $S - T$. This path is known as the optimal path. The principle of optimality states that the every segment of the optimal path is also optimal. For example, let point $b$ lie on the optimal path from point $a$ to point $c$. The principle of optimality states that path $S$ is the optimal path from $a$ to $b$ and $T$ is the optimal path from $b$ to $c$. The proof is relatively simple. If segment $T$ is not the optimal path from $b$ to $c$, then there exists some optimal path $T'$ from $b$ to $c$. However, this implies that path $S - T'$ has a lower cost than path $S - T$ which contradicts the assumption that $S - T$ is optimal. Thus, $T'$ cannot be optimal. Since $T'$ is chosen arbitrarily, this implies that $T$ is optimal. By the same reasoning, $S$ is optimal. Since point $b$ is arbitrary, this holds for all points on the optimal path from $a$ to $c$.

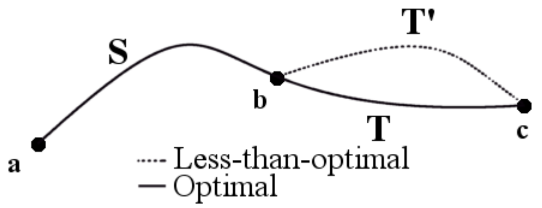

**Figure 1.1**: The principle of optimality: The optimal path from a to c is S-T. If T' is a better path than T, then S-T' becomes a better path than S-T which violates the assumption.

Dynamic programming requires the comparison of all possible paths to obtain the optimal plan. For continuous problems, this results in an infinitely large search space. Discrete dynamic programming breaks the state space into discrete points and then compares all possible strategies in order to select the globally optimal strategy [8]. By treating the problem as a multi-stage process and optimizing each stage sequentially, the space of admissible solutions is reduced. However, for higher dimensional spaces, such an approach is still too computationally expensive (due to the *curse of dimensionality*). Additionally, this approach assumes there is perfect knowledge of the system. In practice, this assumption is rarely true.

Approximate dynamic programming (ADP), on the other hand, is an incremental minimization approach that provides approximations for the optimal cost and the optimal strategy. Of particular interest in these methods is the *cost-to-go*. The cost-to-go is a measure of the cost over all future stages given the optimal strategy and the current state. Adaptive critic designs (ACD) implement ADP through the use of recurrence relations for the optimal policy, the optimal cost, and, in some instances, their derivatives. These designs attempt to overcome the curse of dimensionality while still converging to near optimal strategies and cost-to-go values.

Some of the better known ADP methods include heuristic dynamic programming

(HDP) and dual-heuristic programming (DHP). Both approaches involve the use of a parametric structure, referred to as the action network, to predict the optimal strategy. HDP uses another parametric structure, called the critic network, to predict the cost-to-go values. While relatively straightforward to implement, HDP tends to breakdown, through slow learning, as the size of the problem grows larger [6]. Alternatively, in DHP, the critic network approximates the derivatives of the cost-to-go function. These values are known as costates. While DHP does not share the same pitfalls as HDP, successful implementations are fairly rare. Because DHP requires derivative information to update the critic network, a more accurate system model is needed than for HDP. However, if a reliable system model is available, DHP is generally a better method for searching the space of strategies for the optimal solution.

### 1.1.2 Neural Networks

The use of parametric structures to approximate unknown mappings is a key component of the implementation of ADP designs. An artificial neural network is a parametric structure inspired by the biological neural networks that drive all intelligent animals. Often the term artificial is left out for sake of brevity. Neural networks have several key qualities that make them the parametric structure of choice for ACD. Neural networks have the ability to approximate any function given a sufficient number of parameters. Neural networks also have the ability to learn in batch mode or incrementally, thus are applicable to offline and online training, respectively. Finally, due to their parallel structure, neural networks can easily be constructed for higher dimensional mappings.

The basic components of neural networks are nodes and weights. Nodes accept one input and act as a nonlinear scalar function. Weights serve as linear transformations between layers of nodes. A neural network may consist of any number of layers and each layer may have any number of nodes. The first layer of a neural network, with nodes representing inputs, is called the input layer and the last layer of the network, with nodes representing outputs, is the called the output layer. All other layers are called hidden layers.

Sigmoidal neural networks contain nodes whose functions are identical and described by exponential or hyperbolic-tangent relationships. Sigmoidal networks are ideal for approximately smooth, nonlinear functions [7]. Another common neural network is the radial basis network, whose node functions are radial-basis in nature. These functions contain two parameters (radius and center) which vary from node to node. Radial-basis networks learn quickly and are good for classification problems where large sets of data are available. Due to the nature of the radial-basis functions, any incremental learning tends to be highly localized [7]. The neural networks used in this dissertation are sigmoidal networks with a single hidden layer.

## 1.2 Dissertation Overview

The main portion of this dissertation is organized into four chapters. Chapter 2 reviews the foundations of neural control. Criteria are established for optimality and linked to the classical linear quadratic regulator solution. The recurrence relations for dual heuristic programming are then established from these criteria. Additionally, the criteria are expressed in terms of the general neural network structure. Neural network training is discussed and some useful notation is established. Chapter 3 discusses, in detail, the neural network structures used in the DHP adaptive critic architecture. The first section of the chapter explicitly defines the structure of the neural network and relationships amongst the neural network parameters which draw directly from the optimality criteria established in chapter 2. The chapter presents a streamlined general approach for constructing such networks and also introduces a novel training method which takes into account the optimality criteria. Two methods are introduced during this chapter and, for sake of continuity, are explained in detail in the first two appendices. Chapter 4 describes the actual implementation of the theory outlined in chapter 3 to a business jet. This includes the selection of various control structure, network and training parameters. The chapter also briefly discusses the possible benefits of implementation on a parallel processing machine. Chapter 5 covers the results obtained through the numerical simulation of the business jet mentioned in the previous chapter. The controlled time response of the business jet is tested throughout the operational domain and for different step command inputs. The performance of the adapting neural controller is compared with an optimal linear controller and with a static version of the neural controller. The first appendix discusses a novel method for selecting randomized, uniformly distributed weights for a single neural network layer. The second appendix describes in detail an method which simplifies the processes of obtaining derivatives in a linear algebra setting. The last appendix gives specific details on the business jet model that was used for the software implementation in chapter 4.

# Chapter 2

# Neural Control Formulation

When a control system is created for a particular application, it is created with some performance metric in mind. The ideal control system maximizes total system performance. In this way, designing a control law for a dynamic system can be considered as an optimization problem. If a cost is associated with poor system performance, then the optimal control law will minimize that cost. The optimization problem is merely the minimization of the cost function subject to the system's dynamics.

There are many techniques available for generating an optimal control law for linear systems [9]. Real world systems, however, usually exhibit nonlinear behaviors. Thus, the application of a linear controller results in a tradeoff between robustness and optimality. Nonlinear controllers are desirable since they can perform robustly over a range of states. Where a linear controller may be stable only close to an equilibrium point, a nonlinear controller can be designed such that is it both stable and optimal for conditions not near the equilibrium point [10] [11].

The control scheme presented in this thesis provides optimal control for nonlinear systems which can be approximated as linear parameter varying (LPV). The purpose of this chapter is to define certain aspects of a system's optimal control law and to establish generalized equations in regards to neural networks and neural controllers.

## 2.1 The Nonlinear System

It is assumed that the dynamics of the nonlinear system being considered take the following form.

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \tag{2.1}$$

The system dynamics equation can be linearized for small state perturbations, $\Delta\mathbf{x}$, about the equilibrium point $(\mathbf{x}_e, \mathbf{u}_e)$. Since the equilibrium control values, $\mathbf{u_e}$, can be numerically computed from any given equilibrium state vector, the equilibrium point will be abbreviated as $\mathbf{x_e}$. Linearizing about this equilibrium point yields the differential equation

$$\dot{\mathbf{x}} = \dot{\mathbf{x}}_e + \Delta\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}_e) + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_e)\Delta\mathbf{x} + \frac{\partial \mathbf{f}}{\partial \mathbf{u}}(\mathbf{x}_e)\Delta\mathbf{u} \tag{2.2}$$

From the definition of equilibrium, $\dot{\mathbf{x}}_e = 0$. For convenience, the following Jacobian matrices are introduced: $\mathbf{F}(\mathbf{x_e}) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x_e})$ and $\mathbf{G}(\mathbf{x_e}) = \frac{\partial \mathbf{f}}{\partial \mathbf{u}}(\mathbf{x_e})$. The linearized system dynamics may change with deviations in certain system parameters. For example, the linearized dynamics of a missile may change significantly with deviations in airspeed but insignificantly with deviations in roll angle [12]. A vector of significant parameters, $\tilde{\mathbf{a}}$, is called a *scheduling vector* and may not necessarily include all elements of the state. However, equation 2.2 shows that a scheduling vector cannot include parameters which are not derivable from the state vector. Thus, recognizing that $\Delta\dot{\mathbf{x}} = \dot{\mathbf{x}}$, the system dynamics can be written as a linear parameter-varying differential equation.

$$\dot{\Delta\mathbf{x}} \simeq \mathbf{F}(\tilde{\mathbf{a}})\Delta\mathbf{x} + \mathbf{G}(\tilde{\mathbf{a}})\Delta\mathbf{u} \tag{2.3}$$

It is assumed that the output of the system is a function of the state of the system and the control values.

$$\mathbf{y}_s = \mathbf{h}(\mathbf{x}, \mathbf{u}) \tag{2.4}$$

Linearizing the system output at the equilibrium point $\mathbf{x}_e$ yields

$$\mathbf{y}_s = \mathbf{y}_{s_e} + \Delta\mathbf{y}_s = \mathbf{h}(\mathbf{x}_e) + \frac{\partial \mathbf{h}}{\partial \mathbf{x}}(\mathbf{x}_e)\Delta\mathbf{x} + \frac{\partial \mathbf{h}}{\partial \mathbf{u}}(\mathbf{x}_e)\Delta\mathbf{u} \tag{2.5}$$

For convenience, define $\mathbf{H}_x(\mathbf{x}_e)$ and $\mathbf{H}_u(\mathbf{x}_e)$ as the Jacobian matrices with respect to the state and control vectors, respectively. The system output then has the following perturbation model about $\mathbf{x}_e$.

$$\Delta\mathbf{y}_s = \mathbf{H}_x(\tilde{\mathbf{a}})\Delta\mathbf{x} + \mathbf{H}_u(\tilde{\mathbf{a}})\Delta\mathbf{u} \tag{2.6}$$

Section 4.2.1 shows that if the optimal control law cannot be expressed as a function solely of the state, then it can be expressed as a function of an augmented state. Therefore, the control law can be expressed as a function of the state without any loss of generality.

$$\mathbf{u} = \mathbf{c}(\mathbf{x}) \tag{2.7}$$

The control law is approximated as linear near the equilibrium point $\mathbf{x}_e$. The control law, linearized about $\mathbf{x}_e$, is

$$\mathbf{u} = \mathbf{u}_e + \Delta\mathbf{u} = \mathbf{c}(\mathbf{x}_e) + \frac{\partial \mathbf{c}}{\partial \mathbf{x}}(\mathbf{x}_e)\Delta\mathbf{x} \tag{2.8}$$

Again, for convenience, the Jacobian of the control law with respect to the state vector is defined as $\mathbf{C}(\mathbf{x_e}) = -\frac{\partial \mathbf{c}}{\partial \mathbf{x}}(\mathbf{x_e})$. The negative sign is chosen for convention. Recognizing that the first term in the above equation is merely $\mathbf{u_e}$ and that $\mathbf{C}(\mathbf{x_e})$ can be expressed as a function of $\tilde{\mathbf{a}}$ yields the parameter-varying control law

$$\Delta\mathbf{u} = -\mathbf{C}(\tilde{\mathbf{a}})\Delta\mathbf{x} \tag{2.9}$$

The control gradient, $\mathbf{C}(\tilde{\mathbf{a}})$, at the equilibrium point $\mathbf{x_e}$ will be referred to as the linear control gain matrix at the equilibrium point $\mathbf{x_e}$. The linear control gain matrix for an equilibrium point can be determined using any linear control technique, as long as it satisfies the above equation.

Sampling several equilibrium points within the region of plausible system operation yields a set of linear control gain matrices which describe the optimal control law. A gain-scheduled controller incorporates these matrices into a single control law. A gain-scheduled controller provides optimal control near each sampled point and interpolates between sampled points. Sample points used to design the linear controller gain matrices will be referred to as the *design points* of the controller.

## 2.2 Optimal Control Approach

An optimal control law is a control law which minimizes the costs associated with the system performance. These costs can be expressed in terms of a scalar terminal cost and an integral function of the state and control.

$$J = \phi(\mathbf{x}(t_f), t_f) + \int_{t_0}^{t_f} L(\mathbf{x}(t), \mathbf{u}(t), t)dt \tag{2.10}$$

For many infinite horizon problems, the terminal cost can be neglected without loss of generality. Some infinite horizon problems also seek to minimize a time-averaged cost rather than total cost [13].

Consider a system for which there exists an optimal controller. Then, for any state, there exists a value which equals the total cost accumulated over all future time. The function which maps the states to the remaining cost values is called the optimal value function and is defined as

$$V(\mathbf{x}(t)) = \min_{\mathbf{u}(t)} \left( \int_{t}^{t_f} L(\mathbf{x}(t), \mathbf{u}(t), t) dt \right) \tag{2.11}$$

At all times the action performed by the controller must be optimal (see section 1.1.1) and thus the derivative of the optimal value function with respect to the optimal control vector always is zero. Letting $\mathbf{x}^*(t)$ and $\mathbf{u}^*(t)$ be the optimal state and control values, respectively, an optimality condition can be written as

$$\frac{\partial V(\mathbf{x}^*(t))}{\partial \mathbf{u}^*(t)} = 0 \tag{2.12}$$

### 2.2.1 Linear Quadratic Regulator (LQR)

By definition, gain-scheduled control laws are optimal at their design points. Therefore, one of the goals of the initialization and learning algorithms for the neural controller is to incorporate a gain-scheduled design into the nonlinear neural network design. Additionally, gain-scheduled controllers have been used extensively and successfully [14] [15] [16] in industry. Gain-scheduling uses linear control theory to provide solutions to nonlinear control problems. The advantages of gain-scheduled controllers can be incorporated into the neural network based controllers through an approach entitled *Classical/Neural Control Synthesis of Nonlinear Control Systems* [17]. This section reviews some of the LQR theory which is relevant to both designs and to the training method outlined in section 3.3.

As mentioned in section 2.1, gain scheduling involves linearizing the dynamic system about a set of equilibrium points referred to as the design points. Due to the linear parameter varying nature of the system, each design point $(\mathbf{x}_e, \mathbf{u}_e)_k$ has a corresponding scheduling vector, $\tilde{\mathbf{a}}_k$ such that

$$\mathbf{0} = \mathbf{f}(\mathbf{x}(\tilde{\mathbf{a}}_k), \mathbf{u}(\tilde{\mathbf{a}}_k))\big|_{k=1,\ldots,n_p} \tag{2.13}$$

where $n_p$ is the total number of design points. At each of these design points, it is assumed that the cost function is quadratic

$$J = \frac{1}{2}\int_{t_0}^{t_f} (\Delta\mathbf{x}^T\mathbf{Q}\Delta\mathbf{x} + 2\Delta\mathbf{x}^T\mathbf{M}\Delta\mathbf{u} + \Delta\mathbf{u}^T\mathbf{R}\Delta\mathbf{u})dt \tag{2.14}$$

While the condition expressed in equation 2.12 is necessary for the optimization of the above cost function, it is not sufficient. There is a second condition which is derived by taking the time derivative of the value function.

$$\frac{\partial V(\mathbf{x}(t))}{\partial t} = -L(\mathbf{x}^*(t), \mathbf{u}^*(t)) - \frac{\partial V(\mathbf{x}^*(t))}{\partial \mathbf{x}^*(t)}\mathbf{f}(\mathbf{x}^*(t), \mathbf{u}^*(t)) - \frac{\partial V(\mathbf{x}^*(t))}{\partial \mathbf{u}^*(t)}\frac{\partial \mathbf{u}^*(t)}{\partial t} \tag{2.15}$$

Let the Hamiltonian be defined as

$$H(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t)) = L(\mathbf{x}(t), \mathbf{u}(t)) + \boldsymbol{\lambda}^T(t)\mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \tag{2.16}$$

where $\boldsymbol{\lambda}^T(t) = \frac{\partial V(\mathbf{x}(t))}{\partial \mathbf{x}(t)}$ is known as the costate vector. Applying the condition specified in equation 2.12 yields

$$\begin{aligned}\frac{\partial V}{\partial t}(\mathbf{x}^*(t)) &= -L(\mathbf{x}^*(t), \mathbf{u}^*(t)) - \boldsymbol{\lambda}^{*T}(t)\mathbf{f}(\mathbf{x}^*(t), \mathbf{u}^*(t)) &(2.17)\\ &= -\min_{\mathbf{u}(t)} H(\mathbf{x}^*(t), \mathbf{u}(t), \boldsymbol{\lambda}^*(t)) &(2.18)\end{aligned}$$

From equation 2.17, it is clear that the derivative of the Hamiltonian, with respect to the control values, must be zero for optimality to be satisfied. This condition is known as the Hamilton-Jacobi-Bellman (HJB) equation as is expressed as

$$\frac{\partial H(\mathbf{x}^*(t), \mathbf{u}^*(t), \boldsymbol{\lambda}^*(t))}{\partial \mathbf{u}^*(t)} = 0 \tag{2.19}$$

The linear quadratic (LQ) problem involves a linear system (equation 2.3), a quadratic cost function (equation 2.14) and the optimal value function

$$V(\Delta\mathbf{x}(t)) = \frac{1}{2}\Delta\mathbf{x}^{*T}(t)\mathbf{P}(t)\Delta\mathbf{x}^{*}(t) \tag{2.20}$$

where $\mathbf{P}(t) \rightarrow \mathbf{P}$ as $t \rightarrow \infty$ [18]. Substituting this value function into the HJB equation yields the condition

$$\Delta\mathbf{x}^{*T}(t)\mathbf{M} + \Delta\mathbf{u}^{*T}(t)\mathbf{R} + \Delta\mathbf{x}^{*T}(t)\mathbf{P}^{T}\mathbf{G} = \mathbf{0} \tag{2.21}$$

which implies that the optimal control law is

$$\Delta\mathbf{u}^{*}(t) = -\mathbf{R}^{-1}(\mathbf{G}^{T}\mathbf{P} + \mathbf{M}^{T})\Delta\mathbf{x}^{*}(t) = -\mathbf{C}\Delta\mathbf{x}^{*}(t) \tag{2.22}$$

Substituting this result back into equation 2.17 and simplifying yields the Riccati equation

$$\begin{aligned} \dot{\mathbf{P}}(t) &= -(\mathbf{F} - \mathbf{G}\mathbf{R}^{-1}\mathbf{M}^{T})\mathbf{P}(t) - \mathbf{P}(t)(\mathbf{F} - \mathbf{G}\mathbf{R}^{-1}\mathbf{M}^{T}) && (2.23) \\ &\quad + \mathbf{P}(t)\mathbf{G}\mathbf{R}^{-1}\mathbf{G}^{T}\mathbf{P}(t) - \mathbf{Q} + \mathbf{M}\mathbf{R}^{-1}\mathbf{M}^{T} && (2.24) \end{aligned}$$

Since $\dot{\mathbf{P}}(t) \rightarrow \mathbf{0}$, this equation can be solved for the steady state Riccati matrix, $\mathbf{P}$. Once $\mathbf{P}$ is obtained, it is used in equation 2.22 to compute the optimal control values. The control law obtained from the steady state Riccati matrix is time invariant.

Since each design point has its own linear system dynamics, the steady state Riccati matrix, and thus the control gains, will vary from design point to design point. The Riccati matrix and control gains associated with the $k^{th}$ design point, $\tilde{\mathbf{a}}_k$, will be denoted $\mathbf{P}_k$ and $\mathbf{C}_k$. Note that at each design point, the gradient of the control law with respect to the state is $\mathbf{C}$ and the gradient of the costate vector with respect to the state is $\mathbf{P}$.

## 2.3 Neural Control Architecture

The neural control system is composed of two neural networks, an *action network* and a *critic network*, which work together to approximate the global control law and costate vectors based on the nonlinear plant's performance. As mentioned above, the gradients of the control law and costate vector with respect to the system state are constrained at the design points to equal the control gain matrix and the Riccati matrix respectively. Thus, these matrices are used to determine the specific architecture and initial parameter values of the neural networks [17].

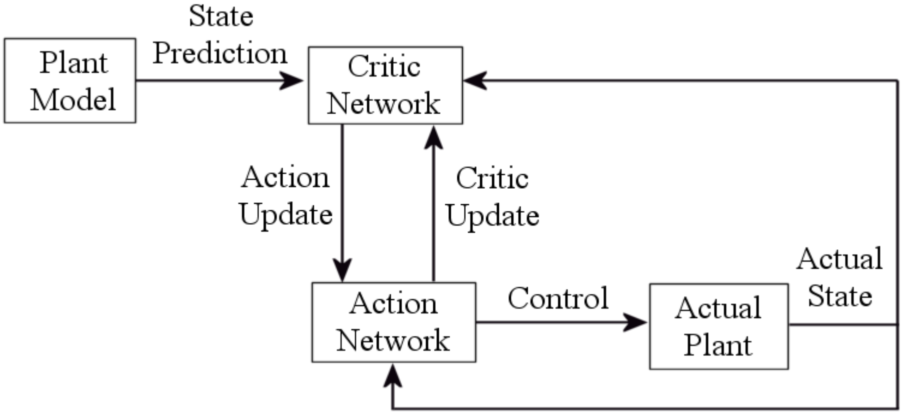

**Figure 2.1**: Block diagram of an dual heuristic programming adaptive critic neural controller

The purpose of the action network is to approximate the global control law while the critic network approximates the costate vector. The input to these networks, $\mathbf{p}$, is composed of states and scheduling variables allowing the network to react not only to changes in state but also, on a more global sense, to changes in scheduled parameters.

$$\mathbf{u}(t) = \mathbf{NN}_A(\mathbf{p}(t)) = \mathbf{z}_A(t) \tag{2.25}$$

$$\boldsymbol{\lambda}(t) = \mathbf{NN}_C(\mathbf{p}(t)) = \mathbf{z}_C(t) \tag{2.26}$$

As mentioned above, at the $k^{th}$ design point, the following equations must hold.

$$\frac{\partial \mathbf{z}_A(t)}{\partial \mathbf{x}(t)}\Big|_k = \mathbf{C}_k \tag{2.27}$$

$$\frac{\partial \mathbf{z}_C(t)}{\partial \mathbf{x}(t)}\Big|_k = \mathbf{P}_k \tag{2.28}$$

These equations will be referred to as the general gradient constraint equations. An additional constraint exists at the design points since, at these points, there is no state deviation and the system is in equilibrium. This implies that both the costate vector and the control law have values of zero.

$$\mathbf{z}_A(t)\Big|_k = \mathbf{0} \tag{2.29}$$

$$\mathbf{z}_C(t)\Big|_k = \mathbf{0} \tag{2.30}$$

This set of constraint equations is known as the general output constraint equations. Optimal controllers must satisfy all of the constraints in equations 2.27 through 2.30. However, the satisfaction of these constraints is not a sufficient condition for optimality. Therefore, the neural control system should be allowed to improve its performance based on some form of learning heuristic.

## 2.4 Dual Heuristic Programming Adaptive Critics

While the controller is operating, the action and critic networks continuously update themselves to more closely approximate the globally optimal control law subject to the current plant dynamics. This adaptation improves performance for conditions under which the linear gain-scheduled design does not perform adequately. This includes large command values and operation in regions outside of the design envelope. The adaptation rules are based on the recurrence relation of dynamic programming [8].

$$V(\mathbf{x}^*(t_k)) = \int_{t_k}^{t_{k+1}} L(\mathbf{x}^*(t_k), \mathbf{u}^*(t_k), t_k)dt + V(\mathbf{x}^*(t_{k+1})) \tag{2.31}$$

At sampled points in time, the networks undergo an updating phase, known as online training, in which the plant performance is analyzed and the controller is updated accordingly. The controller's built-in gain-scheduled knowledge is also perfectly preserved during the training due to a special training algorithm, *Constrained Resilient Backpropagation with scaling and backtracking* (CRPROP), which is outlined in section 3.3.

Because the training takes place at discrete time intervals, the plant model must also be considered in discrete time. Along the optimal path, the state at time $t_{k+1}$ can be expressed in terms of the state at time $t_k$.

$$\mathbf{x}^*(t_{k+1}) = \mathbf{x}^*(t_k) + \int_{t_k}^{t_{k+1}} \mathbf{f}(\mathbf{x}^*(t), \mathbf{u}^*(t))dt \tag{2.32}$$

At time $t_k$, target values for the critic network can be obtained by differentiating the optimal value function (as expressed in equation 2.31) with respect to the states.

$$\boldsymbol{\lambda}^T(t_k) = \frac{d}{d\mathbf{x}^*(t_k)} \left( \int_{t_k}^{t_{k+1}} L(\mathbf{x}^*(t), \mathbf{u}^*(t), t)dt \right) + \boldsymbol{\lambda}^T(t_{k+1}) \frac{\partial \mathbf{x}^*(t_{k+1})}{\partial \mathbf{x}^*(t_k)} \tag{2.33}$$

Substituting equation 2.32 into equation 2.33gives an update equation for the costate vector based on the predicted value of the costate vector at time $t_{k+1}$.

$$\boldsymbol{\lambda}^T(t_k) = \boldsymbol{\lambda}^T(t_{k+1}) + \frac{d}{d\mathbf{x}^*(t_k)} \left( \int_{t_k}^{t_{k+1}} L(\mathbf{x}^*(t), \mathbf{u}^*(t), t) + \boldsymbol{\lambda}^T(t_{k+1})\mathbf{f}(\mathbf{x}^*(t), \mathbf{u}^*(t))dt \right) \tag{2.34}$$

For the neural controlled system, the HJB equation becomes

$$\Delta \mathbf{x}^{*T}(t_k)\mathbf{M} + \Delta \mathbf{u}^{*T}(t_k)\mathbf{R} + \mathbf{z}_C^T(t_k) \frac{\partial \mathbf{f}(\mathbf{x}^*(t_k), \mathbf{u}^*(t_k))}{\partial \mathbf{u}^*(t_k)} = \mathbf{0} \tag{2.35}$$

This equation can be solved numerically for the control values using a root-finding technique such as Newton's method. The control values found using this technique and the costate values computed from equation 2.34 can then be used as targets for the training algorithm.

So far, the conditions placed upon the action and critic networks are similar. Gradient and output constraints are available for both networks and targets can be computed for both networks. Because the same type of information is available for both neural networks, their building and training methods are the same. Therefore, the sections discussing the internal neural network functions are discussed for the general case where, at the design points, the gradients are known and the output is zero.

## 2.5 Feedforward Neural Networks

Feedforward neural networks can be used as the action and critic networks in the adaptive critic architecture mentioned above. Single layer feedforward neural networks (figure 2.2) are useful because both their output and gain matrices are relatively straight forward to compute. As mentioned above, the methods for building and training the action network are the same methods used for the critic network. Therefore, the matrix **D** is created to represent the gradient of the network output to the state vector. It equals the matrix **C** for the action network and the matrix **P** for the critic network. The input and output vectors will be denoted as **p** and **z**, respectively.

The output of a feedforward network is computed as

$$\mathbf{z} = \mathbf{V}\sigma(\mathbf{W}\mathbf{p}) + \mathbf{b} \tag{2.36}$$

The input to the neural network, **p**, is comprised of the state deviation vector, $\Delta\mathbf{x}$, the scheduling vector, $\tilde{\mathbf{a}}$, and the bias input, 1. The state deviation vector inputs

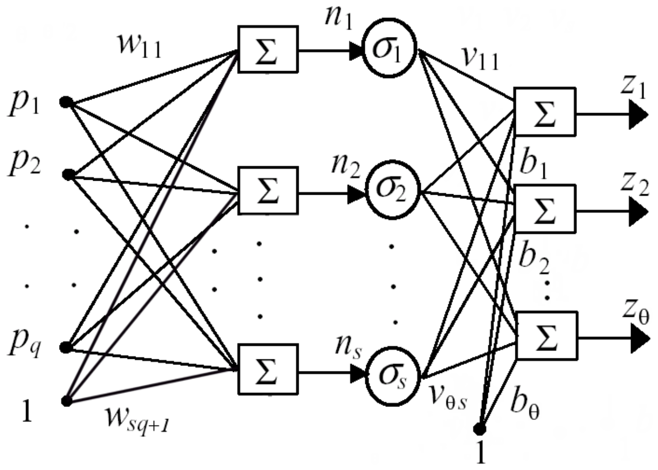

**Figure 2.2**: A neural network with a single hidden layer. There are $q$ inputs, $s$ hidden layer nodes, and $\theta$ outputs in this neural network.

are referred to as *regular inputs.* The scheduling vector and the bias input are referred to as auxiliary inputs and denoted by **a**. The output values which correspond to the $k^{th}$ design point are denoted by $\mathbf{z}_k$. Likewise, $\mathbf{a}_k$ refers to the auxiliary inputs at the $k^{th}$ design point.

At the design points, the state deviation values are equal to zero and the neural network output equation reduces to

$$\mathbf{z}_k = \mathbf{V}\sigma(\mathbf{W_A}\mathbf{a}_k) + \mathbf{b} \tag{2.37}$$

where $\mathbf{W_A}$ is the matrix of input weights associated with the auxiliary inputs. Thus, $\mathbf{W_A}$ is called the auxiliary input weight matrix. $\mathbf{W_R}$ is the matrix of input weights associated with the state deviations and is referred to as the regular input weight matrix.

As in section 2.2.1, $n_p$ is the number of design points. Let $\mathbf{Z}$ be the matrix of outputs for all the design points, such that the $k^{th}$ column of $\mathbf{Z}$ is the output at the $k^{th}$ design point. Let $\mathbf{A}$ be defined such that the $k^{th}$ column of $\mathbf{A}$ is the auxiliary input vector for the $k^{th}$ design point. Then the total design point output is

$$\mathbf{Z} = \mathbf{V}\sigma(\mathbf{W_A}\mathbf{A}) + \mathbf{B} \tag{2.38}$$

where $\mathbf{B}$ is a matrix with $n_p$ columns which are all given by $\mathbf{b}$.

Recall from equations 2.29 and 2.30, which are restated here

$$\mathbf{z}_A(t)\Big|_k = \mathbf{0}$$

$$\mathbf{z}_C(t)\Big|_k = \mathbf{0}$$

that the network should have an output of zero at the design points yields the equation

$$\mathbf{V}\sigma(\mathbf{W_A}\mathbf{A}) + \mathbf{B} = 0 \tag{2.39}$$

This is the output constraint equation in terms of the neural network structure and, as such, is a necessary condition for the gain-scheduled controller and the neural network controller to perform equally at the design points.

The gradient of the neural network with respect to the state deviations at the $k^{th}$ design point is

$$\frac{\partial \mathbf{z}_k}{\partial \Delta \mathbf{x}} = \mathbf{V}^T \mathbf{\Sigma}_k \mathbf{W_R}^T = \mathbf{D}_k \tag{2.40}$$

where $\mathbf{\Sigma}_k$ is a diagonal matrix whose elements, $\mathbf{\Sigma}(i,i)$ correspond to the derivative of the hidden layer's $i^t h$ node's output with respect to its input evaluated at the $k^{th}$ design point. Equation 2.40 also is a necessary condition for the gain-scheduled and neural controllers to perform equally at the design points.

Satisfying these two equations will not be enough to provide stable performance near the design points. One additional requirement is that the output of the networks must be zero whenever the regular inputs (the state deviations) are zero. This can be accomplished by adding an adjustment to the output of the neural network.

$$\mathbf{z} = \mathbf{V}\sigma(\mathbf{W}\mathbf{p}) + \mathbf{b} + \mathbf{z}_{adj} \tag{2.41}$$

$$\mathbf{z}_{adj} = -\mathbf{V}\sigma(\mathbf{W_A}\mathbf{a}) - \mathbf{b} \tag{2.42}$$

At the design points, this adjustment value is zero. Also, it has no effect on the output and also has no effect on the derivatives of the outputs with respect to the regular inputs. However, it does affect the error gradient used for training the neural network.

### 2.5.1 Neural Network Training

The neural controller should improve its performance at points other than the design points through the use of a training algorithm. The method for obtaining a target output vector for the action and critic neural networks is shown in section 2.4. In this section the target output vector will be denoted by $\mathbf{y}$. In order to train the networks, an error function is used to determine how far the neural network is from achieving the target output. Many training methods then adjust each weight's value according to the derivative of the error function with respect to that weight. The gradient of the error function with respect to all of the weights is referred to as the error gradient. In this research, the term *unconstrained error gradient* refers to the error gradient when it is assumed that all weights are independent of each other. In typical neural networks, the weights are independent. However, as shown in chapter 3, it is necessary to define dependencies in order to satisfy the constraint conditions.

Typically, the error function defined for a neural network is

$$e = \frac{1}{2}(\mathbf{z} - \mathbf{y})^T(\mathbf{z} - \mathbf{y}) \tag{2.43}$$

From this equation, the unconstrained error gradient with respect to the output weights is easily obtained via the chain rule. The output weight error gradient can be expressed as the matrix $\breve{\mathbf{V}}$ where

$$\breve{\mathbf{V}} = (\mathbf{z} - \mathbf{y}) \left[\sigma(\mathbf{W}\mathbf{p})^T - \sigma(\mathbf{W_A}\mathbf{a})^T\right] \tag{2.44}$$

The breve notation is used to represent the unconstrained error gradient with respect to the indicated matrix. The breve matrix is the same size as the indicated matrix and each element of the breve matrix is the partial derivative of the error function with respect to the corresponding element of the indicated matrix (all weights are assumed independent). A hat matrix, such as $\hat{\mathbf{V}}$, is used to represent the constrained error gradient with respect to the indicated matrix. The hat matrix is the same size as the indicated matrix and each element of the hat matrix is the derivative of the error function with respect to the corresponding element of the indicated matrix. The hat notation will be used more in chapter 3.

The unconstrained error gradient with respect to the regular input weights is

$$\breve{\mathbf{W}}_{\mathbf{R}} = \mathbf{\Sigma}\mathbf{V}^T(\mathbf{z} - \mathbf{y})\Delta\mathbf{x}^T \tag{2.45}$$

and the unconstrained error gradient with respect to the auxiliary input weights is

$$\breve{\mathbf{W}}_{\mathbf{A}} = \left[\mathbf{\Sigma} - \mathbf{\Sigma}_{adj}\right] \mathbf{V}^T(\mathbf{z} - \mathbf{y})\Delta\mathbf{a}^T \tag{2.46}$$

where $\mathbf{\Sigma}_{adj}$ is equal to $\mathbf{\Sigma}$ when all regular inputs are set to zero. The unconstrained error gradient with respect to the output bias is zero.

For the typical neural network which has independent weights, there are several different types of gradient-based learning techniques. Two popular techniques are Backpropagation and Resilient Backpropagation (RPROP). The backpropagation technique uses a scaling factor, called the learning rate, in combination with the gradient in a traditional gradient-descent method. One problem with such a technique is that, for large networks, the error gradient with respect to any particular weight's value is very small. Thus, successful training may take many iterations. RPROP was designed to overcome this problem and is presented, along with additional modifications, in chapter 3.

### 2.5.2 Some Useful Notation

A new and useful notation is introduced in order to explain how the constraint equations are incorporated into the neural networks' learning algorithm. Often, it is necessary to refer to the set of values associated with a group of inputs, a group of hidden layer nodes, or a group of outputs. The symbology presented here is meant to simplify the equations used to refer to these sets of values.

Let $I$ be a subset of all the inputs, let $H$ be a subset of all the hidden layer nodes, and let $\Theta$ be a subset of all the outputs. Then $I$, $H$, and $\Theta$ are referred to as groups of inputs, hidden layer nodes, and outputs. Let $I^c$, $H^c$, and $\Theta^c$ be the complements of $I$, $H$, and $\Theta$, respectively. If $\mathbf{A}$ is a matrix of values related to the neural network and if $\mathbf{B} = \mathbf{A}^{(I,H,\Theta)}$, then $\mathbf{B}$ is computed by removing from $\mathbf{A}$ all rows and columns corresponding to $(I^c, H^c, \Theta^c)$. $\mathbf{B}$ will be referred to as a submatrix of $\mathbf{A}$. A $*$ in place of a group reference indicates that the submatrix should not filter out any rows or columns based on the group type where the $*$ appears. In other words, the compliment of $*$ is the null set. For example, $\mathbf{A}^{(I,H,*)}$ refers to the submatrix of $\mathbf{A}$ pertaining to input group $I$, hidden layer group $H$, and all output groups. This notation can be used with $n^{th}$ dimensional data. The concepts of rows and columns in the two-dimensional definition is simply extended to all $n$ dimensions. For simplicity, the resulting data structure will still be referred to as a submatrix even though it is a rank-$n$ tensor. In figure 2.3, the input weights associated with input group 1 and hidden layer node group 2 are referred to as $\mathbf{W}^{(I_1,H_2,*)}$. Since input weights are not associated with output weights, the symbol $*$ is used to avoid confusion.

Subscripts are used throughout the thesis to refer to a specific element of a ma-

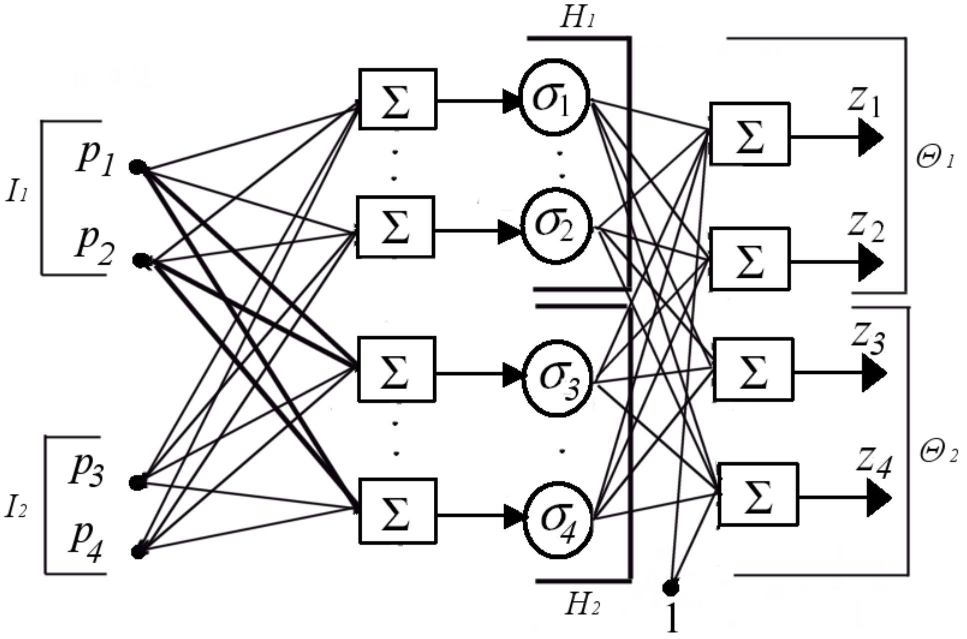

**Figure 2.3**: Neural network with input groups $I_1$ and $I_2$, hidden layer node groups $H_1$ and $H_2$, and output groups $\Theta_1$ and $\Theta_2$. The weights corresponding to $\mathbf{W}^{(I_1,H_2,*)}$ are shown in bold.

trix or tensor by specifying the element's coordinates. When used in conjunction with the superscript group identifiers, the subscripts refer to the coordinates within the submatrix specified by the superscripts. Similarly, the symbol $*$ in a subscript position indicates all elements of whatever position it takes. For example, $\mathbf{W}_{(*,1)}$ is the first column vector and $\mathbf{W}_{(*,*)}$ is the entire matrix $\mathbf{W}$.

## 2.6 Chapter Summary

Perturbation models have been established from the nonlinear dynamic system response, the system output and the optimal control law. Criteria for the optimal control law have been developed and related to the classical LQR result. The neural control architecture has been established and the recurrence relations have been developed for the dual heuristic programming adaptive critic. These relations have been expressed in terms of targets for the neural networks. Likewise, the optimality conditions have been related to the neural network parameters. The unconstrained error gradient for neural network training has been defined and some convenient notation has been introduced.

# Chapter 3

# Constrained Neural Controller

Neural controllers have been used successfully in several applications [2] [3]. These controllers work by allowing neural networks to approximate key parameters used in determining control values. Usually these networks are subject to large amounts of offline training prior to online use. This is done to ensure that the neural networks are able to perform well when first applied to the actual plant. The online training is used to optimize performance based on the real-time plant dynamics. Offline training usually consists of fitting the neural networks to large sets of input/target pairs. The input/target pairs can be generated from simulation or collected from actual plant experiments. A novel approach was recently suggested by Ferrari [17] to make use of the constraints mentioned in the prior chapter to determine the structure and the initial weight values of the neural network. This chapter presents modifications to this method and extends the concept of constraining the network beyond the initialization phase to the online training phase. A method for adapting the neural network weights is presented that allows the network to be optimized while satisfying the zero-output and gradient constraints.

## 3.1 Network Constraint Equations

As in the pervious chapter, let $n_p$ be the number of design points for the controller. Then, let $n_h$ be the number of hidden layer nodes. Finally, let $n_r$ be the number of regular inputs and let $n_o$ be the number of network outputs. For $n_h = n_o n_p$, there exists a method for choosing the weight values such that the constraints mentioned above are satisfied [19].

The hidden layer nodes can be partitioned such that each output has $n_p$ hidden layer nodes associated with it. The hidden layer nodes associated with the $\theta^{th}$ output are referred to as the $\theta^{th}$ hidden layer node group, $H_\theta$. Note that $H_\theta^c$ refers to the group of hidden layer nodes that are not associated with the $\theta^{th}$ output.

For convenience, define $\mathbf{S}$ as the hidden layer nodes' output at all design points where $\mathbf{S} = \sigma(\mathbf{W_A A})$. For the $\theta^{th}$ output, the output constraint equation reduces to

$$\mathbf{Z}_{(\theta,*)} = \mathbf{V}_{(\theta,*)}\mathbf{S} + \mathbf{B}_{(\theta,*)} = \mathbf{0} \tag{3.1}$$

Equation 3.1 can be rewritten as

$$\mathbf{Z}_{(\theta,*)} = \mathbf{V}_{(\theta,*)}^{(*,H_\theta,*)}\mathbf{S}^{(*,H_\theta,*)} + \mathbf{V}_{(\theta,*)}^{(*,H_\theta^c,*)}\mathbf{S}^{(*,H_\theta^c,*)} + \mathbf{B}_{(\theta,*)} = \mathbf{0} \tag{3.2}$$

Note that because each hidden layer group has $n_p$ nodes, $\mathbf{S}^{(*,H_\theta,*)}$is $n_p$ x $n_p$ and thus invertible. Rearranging equation 3.1 yields the function

$$\mathbf{V}_{(\theta,*)}^{(*,H_\theta,*)} = -\left[\mathbf{V}_{(\theta,*)}^{(*,H_\theta^c,*)}\mathbf{S}^{(*,H_\theta^c,*)} + \mathbf{B}_{(\theta,*)}\right]\left[\mathbf{S}^{(*,H_\theta,*)}\right]^{-1} \tag{3.3}$$

This function defines the value of $\mathbf{V}_{(\theta,*)}^{(*,H_\theta,*)}$ so that the output constraint with respect to the $\theta^{th}$ output is satisfied. For this reason, the output weights associated with $\mathbf{V}_{(\theta,*)}^{(*,H_\theta,*)}$ are referred to as *constrained output weights* and the weights associated with $\mathbf{V}_{(\theta,*)}^{(*,H_\theta^c,*)}$ are referred to as *unconstrained output weights.* The function is referred to as an *output weight construction function.*

For convenience, the following abbreviations are adopted from this point forward.

$$\begin{array}{rcl} \mathbf{V}_\theta & = & \mathbf{V}_{(\theta,*)}^{(*,H_\theta,*)} \\ \mathbf{V}_\theta^c & = & \mathbf{V}_{(\theta,*)}^{(*,H_\theta^c,*)} \\ \mathbf{S}_\theta & = & \mathbf{S}^{(*,H_\theta,*)} \\ \mathbf{S}_\theta^c & = & \mathbf{S}^{(*,H_\theta^c,*)} \end{array} \tag{3.4}$$

Then, the output weight construction function (equation 3.3) can be written as

$$\mathbf{V}_\theta = -\left[\mathbf{V}_\theta^c\mathbf{S}_\theta^c + \mathbf{B}_{(\theta,*)}\right]\left[\mathbf{S}_\theta\right]^{-1} \tag{3.5}$$

A similar function can be derived for the gradient constraints. For convenience, define $\dot{\mathbf{Z}}$ and $\mathbf{E}$ as rank-3 tensors where, for $\theta \in [1, n_o]$,

$$\dot{\mathbf{Z}}_{(*,*,\theta)} = \begin{bmatrix} \frac{\partial \mathbf{z}_{1(\theta,*)}}{\partial \Delta \mathbf{x}} \\ \vdots \\ \frac{\partial \mathbf{z}_{n_p(\theta,*)}}{\partial \Delta \mathbf{x}} \end{bmatrix}^T \tag{3.6}$$

$$\mathbf{E}_{(*,*,\theta)} = \begin{bmatrix} \mathbf{D}_{1(\theta,*)} \\ \vdots \\ \mathbf{D}_{n_p(\theta,*)} \end{bmatrix}^T \tag{3.7}$$

Using this notation, equation 2.40 can be rewritten as

$$\dot{\mathbf{Z}}_{(k,*,\theta)} = \mathbf{W}_{\mathbf{R}} \mathbf{\Sigma}_k \mathbf{V}_{(\theta,*)} = \mathbf{E}_{(k,*,\theta)} \tag{3.8}$$

for the $k^{th}$ design point and the $\theta^{th}$ output. If $\mathbf{K}$ is defined as

$$\mathbf{K}_{(*,*,\theta)} = \begin{bmatrix} \mathbf{\Sigma}_1 \mathbf{V}_{(\theta,*)} & \ldots & \mathbf{\Sigma}_{n_p} \mathbf{V}_{(\theta,*)} \end{bmatrix} \tag{3.9}$$

then equation 3.8 can be expressed as

$$\dot{\mathbf{Z}}_{(*,*,\theta)} = \mathbf{W}_{\mathbf{R}} \mathbf{K}_{(*,*,\theta)} = \mathbf{E}_{(*,*,\theta)} \tag{3.10}$$

This system of $n_o$ simultaneous equations can be written as a single equation.

$$\mathbf{W}_{\mathbf{R}} \begin{bmatrix} \mathbf{K}_{(*,*,1)} & \ldots & \mathbf{K}_{(*,*,n_o)} \end{bmatrix} = \begin{bmatrix} \mathbf{E}_{(*,*,1)} & \ldots & \mathbf{E}_{(*,*,n_o)} \end{bmatrix} \tag{3.11}$$

For convenience, define $\bar{\mathbf{K}}$ and $\bar{\mathbf{E}}$ as

$$\bar{\mathbf{K}} = \begin{bmatrix} \mathbf{K}_{(*,*,1)} & \ldots & \mathbf{K}_{(*,*,n_o)} \end{bmatrix} \tag{3.12}$$

$$\bar{\mathbf{E}} = \begin{bmatrix} \mathbf{E}_{(*,*,1)} & \ldots & \mathbf{E}_{(*,*,n_o)} \end{bmatrix} \tag{3.13}$$

Since $\bar{\mathbf{K}}$ is $n_o n_p$ x $n_o n_p$, equation 3.11 can be solved for $\mathbf{W_R}$.

$$\mathbf{W_R} = \bar{\mathbf{E}}\,\bar{\mathbf{K}}^{-1} \tag{3.14}$$

This approach assumes that the derivative of each output with respect to each input is known. However, in the case of an airplane, it may be easier to model the lateral dynamics and the longitudinal dynamics separately. If these two models are used to describe the system as a whole, the cross-model derivatives will not be known *a priori*. Instead, the neural controller must be allowed to learn these derivatives online. Therefore, it is necessary to have a method which constrains the network output derivatives for all input/output pairs belonging to the same model while leaving cross-model derivatives unconstrained.

Assume that the system being controlled is described by $n_m$ independent dynamic models. Then let the set of inputs related to the $m^{th}$ model be designated by the $m^{th}$ input group, $I_m$. Likewise, let the set of outputs related to this model be designated by the $m^{th}$ output group, $\Theta_m$. For convenience, let the group of hidden layer nodes associated with the outputs of model $m$ be designated by $M_m$. For now, assume that elements of $\mathbf{E}$ are undefined for coordinates which correspond to unknown derivatives. Then, equation 3.11 can be modified to express the value of the input weights corresponding to $I_m$ and $H_\theta$ where the $\theta^{th}$ output belongs to $\Theta_m$.

$$\mathbf{W_R}^{(I_m,*,*)}\bar{\mathbf{K}}^{(*,*,\Theta_m)} = \bar{\mathbf{E}}^{(I_m,*,\Theta_m)} \tag{3.15}$$

The left hand side of this equation can be broken into a sum of the gradient contribution of the $m^{th}$ model and the gradient contribution of all of the other models, as follows:

$$\mathbf{W_R}^{(I_m,M_m,*)}\bar{\mathbf{K}}^{(*,M_m,\Theta_m)} + \mathbf{W_R}^{(I_m,M_m^c,*)}\bar{\mathbf{K}}^{(*,M_m^c,\Theta_m)} = \bar{\mathbf{E}}^{(I_m,*,\Theta_m)} \tag{3.16}$$

It can be observed that $\bar{\mathbf{K}}^{(*,M_m,\Theta_m)}$ is $(n_{o_m} n_p)$ x $(n_{o_m} n_p)$, where $n_{o_m}$ is the number of outputs associated with the $m^{th}$ model. Thus equation 3.16 can be solved for the values of the weights associated with input group $I_m$ and hidden layer node group $M_m$.

$$\mathbf{W_R}^{(I_m,M_m,*)} = \left[\bar{\mathbf{E}}^{(I_m,*,\Theta_m)} - \mathbf{W_R}^{(I_m,M_m^c,*)}\bar{\mathbf{K}}^{(*,M_m^c,\Theta_m)}\right] \left[\bar{\mathbf{K}}^{(*,M_m,\Theta_m)}\right]^{-1} \tag{3.17}$$

For convenience, the following abbreviations are adapted from this point forward

$$\begin{array}{rcl} \bar{\mathbf{K}}_m & = & \bar{\mathbf{K}}^{(*,M_m,\Theta_m)} \\ \bar{\mathbf{K}}_m^c & = & \bar{\mathbf{K}}^{(*,M_m^c,\Theta_m)} \\ \bar{\mathbf{E}}_m & = & \bar{\mathbf{E}}^{(I_m,*,\Theta_m)} \\ \mathbf{W}_m & = & \mathbf{W_R}^{(I_m,M_m,*)} \\ \mathbf{W}_m^c & = & \mathbf{W_R}^{(I_m,M_m^c,*)} \end{array} \tag{3.18}$$

Then, equation 3.17 can be rewritten as

$$\mathbf{W}_m = \left[\bar{\mathbf{E}}_m - \mathbf{W}_m^c\bar{\mathbf{K}}_m^c\right] \left[\bar{\mathbf{K}}_m\right]^{-1} \tag{3.19}$$

This function defines the values of $\mathbf{W}_m$ so that the gradient constraint related to the $m^{th}$ model is satisfied. The weights represented by $\mathbf{W}_m$ are referred to as *constrained input weights* and the weights represented by $\mathbf{W}_m^c$ are referred to as *unconstrained input weights*. Equation 3.19 is referred to as the *input weight construction function.* This function and the output weight construction function determine the values of the constraint input and output weights so that all constraining equations are satisfied.

## 3.2 Initialization

A method is presented in [17] for the selection of initial neural network weight values. A modification of this method is presented here to produce satisfactory initial weight values using the construction functions and a new process for distributing network weights. The construction function defines the values of the constrained weights based on the values of the unconstrained weights, the output bias vector, and the auxiliary input weights. Therefore, the task of selecting initial values for all network parameters is reduced to selecting values for these three groups.

### 3.2.1 Unconstrained Weights

The unconstrained weights can be given initial values of zero. This makes the cross-model derivatives zero, which is equivalent to assuming decoupled plant dynamics. Knowing *a priori* that the initialized neural network will have derivatives of zero for all unconstrained derivatives implies that the initial network can be computed from the construction equations as if only one model were used.

### 3.2.2 Bias Vector

The values in the output bias vector can be chosen as random. While the precise value of the biases is unimportant, the order of magnitude of the bias values plays an important role. Through the output weight construction function, the order of magnitude of the bias dictates the order of magnitude of the output weights. The order of magnitude of the output weights effects the order of magnitude of the input weights through the input weight construction function. The order of magnitude of the input weights serves as a scale to the input of the sigmoidal functions of the hidden layer, essentially controlling how linear the sigmoidal functions are near the origin. Thus, the order of magnitude of the bias vector can be used to control the linearity of the sigmoidal functions with respect to the regular inputs near zero. Since the controller is based on a LPV model, linearity of the sigmoidal function is desirable near zero. Therefore, the biases should have high orders of magnitude. If $b^*$ has a desirable order of magnitude, a bias vector can be constructed using algorithm 3.1. Using this procedure results in an expected order of magnitude of $\log_{10}(b^*)$. The minimum and maximum possible orders of magnitude for elements of the bias vector are $\log_{10}(b^*) - 0.5$ and $\log_{10}(b^*) + 0.5$, respectively. Adjusting the value of $R$ will alter the minimum and maximum values accordingly.

**Algorithm 3.1** Initialize the bias vector, $\mathbf{b}$.

```
$b^* \leftarrow$ desired output bias value
$R \leftarrow 1$
for $i = 1$ to $n_\theta$ do
    $r \leftarrow R(rand - 0.5) + \log_{10}(b^*)$
    if $rand < 0.5$ then
        $\mathbf{b}_{(i)} \leftarrow -10^r$
    else
        $\mathbf{b}_{(i)} \leftarrow 10^r$
    end if
end for
```

### 3.2.3 Auxiliary Input Weights

The process for selecting the auxiliary input weights is more detailed. The output weight construction function specified earlier (equation 3.3) requires that the auxiliary input weights have values which allow certain matrices to be invertible. Specifically, the matrix of the $\theta^{th}$ hidden layer group's outputs, $\mathbf{S}^{(*,H_\theta,*)}$, must be invertible. Thus the matrix should be well conditioned, meaning it should have a condition number less than $\varepsilon^{-0.5}$ where $\varepsilon$ is the smallest positive number such that $(\varepsilon + 1) - 1 = 0$ on the chosen computing machine. A strategy for choosing auxiliary input weights such that $\mathbf{S}^{(*,H_\theta,*)}$ is well-conditioned with probability 1 is presented in [20].

The strategy consists of selecting a suitable matrix $\mathbf{T_n}^{(*,H_\theta,*)}$ as the target hidden layer inputs, solving for the values of $\mathbf{W_A}^{(*,H_\theta,*)}$ which produce the least squares best match between the target hidden layer inputs and the actual hidden layer inputs. The auxiliary input weight matrix, $\mathbf{W_A}^{(*,H_\theta,*)}$, is then scaled so that the maximum magnitude of the hidden layer inputs is 10. It can be observed that at input values of more than 10, the sigmoidal function is saturated.

A suitable target hidden layer input matrix is selected using the following process. All the diagonal elements of $\mathbf{T_n}^{(*,H_\theta,*)}$ are set to zero and all the non-diagonal elements are chosen independently from a normal distribution with mean zero and variance one. Generating the target matrix in this manner is equivalent to distributing the sigmoidal functions across the input space as in the Nguyen-Widrow initialization algorithm [21]. A modified version of this strategy can be used to pick auxiliary input weights which provide a best fit approach to achieving a condition number of one for $\mathbf{S}^{(*,H_\theta,*)}$. Using a method referred to as *Dual-point Hyperspherical Initialization*, a matrix with a condition number of approximately one, ${\mathbf{T_s}}^{(*,H_\theta,*)}$, is selected as the target hidden layer output matrix. This method is outlined in appendix A. The target hidden layer input matrix can then be determined by applying the inverse of the sigmoidal function. If such an inverse does not exist (for instance, more than one input can generate the same output), a pseudo-inverse function returns one of the values, selected at random, producing the desired sigmoidal function output:

$${\mathbf{T_n}}^{(*,H_\theta,*)} = \sigma^{-1}\left({\mathbf{T_s}}^{(*,H_\theta,*)}\right) \tag{3.20}$$

From this point onward, the procedure for selecting the input weights is the same as the strategy outlined in [20]. The auxiliary input weight matrix, ${\mathbf{W_A}}^{(*,H_\theta,*)}$ is sought for which

$${\mathbf{T_n}}^{(*,H_\theta,*)} = {\mathbf{W_A}}^{(*,H_\theta,*)}\mathbf{A} \tag{3.21}$$

Since $\mathbf{A}$ is not necessarily square, such a matrix does not always exist. However, the least squares best fit can be computed as

$$\mathbf{W}_{\mathbf{A}}^{\mathbf{LS}(*,H_\theta,*)} = {\mathbf{T_n}}^{(*,H_\theta,*)}\mathbf{A}^T\left[\mathbf{A}\mathbf{A}^T\right]^{-1} \tag{3.22}$$

Let $\mathbf{N}^{\mathbf{LS}(*,H_\theta,*)}$ be the matrix of the $\theta^{th}$ hidden layer node group's inputs when the least squares best fit auxiliary input weight matrix is used.

$$\mathbf{N}^{\mathbf{LS}(*,H_\theta,*)} = \mathbf{W}_{\mathbf{A}}^{\mathbf{LS}(*,H_\theta,*)}\mathbf{A} \tag{3.23}$$

Let $l_\theta$ be the largest magnitude of the elements of $\mathbf{N}^{\mathbf{LS}(*,H_\theta,*)}$. The auxiliary input weight matrix corresponding to the $\theta^{th}$ hidden layer node group can be set according to:

$$\mathbf{W_A}^{(*,H_\theta,*)} = \frac{10}{l_\theta}\mathbf{W_A^{LS}}^{(*,H_\theta,*)} \tag{3.24}$$

This modified method focuses on evenly distributing the outputs of the hidden layer. The Dual-point Hyperspherical Initialization method can also be used to generate the target hidden layer input matrix directly, in which case the distribution focus will be on the inputs to the hidden layer.

### 3.2.4 Constrained Weights

The initial values for the constrained weights are computed from the construction equations 3.5 and 3.19, using the values of the unconstrained weights, the output bias vector and the auxiliary input weights determined in the three previous sections. Once the values of the constrained weights are determined, the entire neural network is considered initialized and the off-line training is complete.

## 3.3 Constrained Gradient-Based Online Training

Once the neural network has been initialized, it is ready to be further adjusted online. Online, the neural network is fed an input vector in order to compute an output vector. The network is also supplied with a target output vector. The weights of the network are adjusted to minimize the different between the target output and the actual output corresponding to the input vector. Each cycle of input-output-target-training is referred to as an epoch. Successful training often requires that each input-target pair be used over several epochs.

As mentioned in section 2.5.1, many training algorithms use unconstrained error gradient based learning techniques. Application of such a technique to the neural network would allow the weights to be modified without any guarantee that the offline training requirements be satisfied. There are two approaches for allowing the neural networks to satisfy the constraint equations during training. One approach is to design a training algorithm which provides adjustments for the weights according to both the unconstrained error gradient and the constraint equations. A second and more feasible approach is to apply the construction equations after weight adjustments have been made. This requires that the weight adjustments be based on the *constrained error gradient* rather than the unconstrained error gradient. The second approach is more feasible because it reduces the training problem to two separate problems, each of which is solvable.

### 3.3.1 Constrained Error Gradient

The constrained error gradient is similar to the unconstrained error gradient except that the dependencies between the weights are taken into account. Computing the constrained error gradient makes use of the extension of the chain rule to linear algebra referred to as the *gradient transformation.* The gradient transformation is used to determine the value of one gradient matrix based on another, related gradient matrix. For instance, let the matrix $\mathbf{A}$ be defined by some function applied to the matrix $\mathbf{B}$.

$$\mathbf{A} = \mathbf{f}(\mathbf{B}) \tag{3.25}$$

Also let the scalar value $e$ represent some error value which is dependent on both the matrix $\mathbf{A}$ and the matrix $\mathbf{B}$. Using the notation from section 2.5.1, the unconstrained error gradient with respect to $\mathbf{A}$ is denoted by $\breve{\mathbf{A}}$, and unconstrained error gradient with respect to $\mathbf{B}$ is denoted by $\breve{\mathbf{B}}$. However, $\breve{\mathbf{B}}$ does not reflect the actual error gradient because of the dependency in equation 3.25. Since the value of $\mathbf{A}$ is constrained by this equation, the constrained error gradient is the value that will provide the true error gradient. The constrained error gradient with respect to $\mathbf{B}$, denoted by $\hat{\mathbf{B}}$, is expressed as

$$\hat{\mathbf{B}} = \breve{\mathbf{B}} + \mathcal{G}[\mathbf{A}, \mathbf{B}, \breve{\mathbf{A}}] \tag{3.26}$$

where the final term is called the gradient transformation of $\mathbf{A}$ with respect to $\mathbf{B}$ given the unconstrained gradient $\breve{\mathbf{A}}$. The gradient transformation can be applied recursively. For instance, if

$$\mathbf{B} = \mathbf{g}(\mathbf{C}) \tag{3.27}$$

then

$$\hat{\mathbf{C}} = \mathcal{G}[\mathbf{B}, \mathbf{C}, \hat{\mathbf{B}}] = \mathcal{G}[\mathbf{B}, \mathbf{C}, \breve{\mathbf{B}} + \mathcal{G}[\mathbf{A}, \mathbf{B}, \breve{\mathbf{A}}]] \tag{3.28}$$

More information about the gradient transformation is contained in appendix B.

In neural network training, the gradient transformation can be used to compute the constrained error gradient from the construction equations and the unconstrained error gradient. It is important to note that since the constrained weights are fixed by the construction functions, it is only necessary to compute the constrained error gradient with respect to the unconstrained weights. By definition of the gradient transformation, the constrained error gradient with respect to the unconstrained input weights is

$$\hat{\mathbf{W}}^c_m = \breve{\mathbf{W}}^c_m + \mathcal{G}[\mathbf{W}_m, \mathbf{W}^c_m, \breve{\mathbf{W}}_m] \tag{3.29}$$

Applying the above gradient transformation (as shown in Appendix B), using equation 3.19 yields

$$\hat{\mathbf{W}}^c_m = \breve{\mathbf{W}}^c_m + \breve{\mathbf{W}}_m \left[\bar{\mathbf{K}}^c_m \left[\bar{\mathbf{K}}_m\right]^{-1}\right]^T \tag{3.30}$$

Computing the constrained error gradient with respect to the unconstrained output weights is more involved. Note that the above gradient transformation involves only one construction function. As shown in equation 3.5, the unconstrained output weights effect the constrained output weights. However, the definition of $\bar{\mathbf{K}}$ and equation 3.19 show that all of the values of the output weights affect the constrained input weight values. Thus, by application of the gradient transformation, the constrained error gradient with respect to the output weights is:

$$\hat{\mathbf{V}}^c_\theta = \check{\mathbf{V}}^c_\theta + \mathcal{G}[\mathbf{V}_\theta, \mathbf{V}^c_\theta, \check{\mathbf{V}}_\theta] + \sum_{m=1}^{n_m} \mathcal{G}[\mathbf{W}_m, \mathbf{V}^c_\theta, \check{\mathbf{W}}_m] \tag{3.31}$$

The summation term of the above equation can be expanded as

$$\begin{aligned}
\mathcal{G}\Big[\mathbf{W}_m, \mathbf{V}^c_\theta, \check{\mathbf{W}}_m\Big] &= \mathcal{G}\Big[\bar{\mathbf{K}}^c_m, \mathbf{V}^c_\theta, \mathcal{G}\Big[\mathbf{W}_m, \bar{\mathbf{K}}^c_m, \check{\mathbf{W}}_m\Big]\Big] \\
&+ \mathcal{G}\Big[\bar{\mathbf{K}}_m, \mathbf{V}^c_\theta, \mathcal{G}\Big[\bar{\mathbf{K}}^{-1}_m, \bar{\mathbf{K}}_m, \mathcal{G}\Big[\mathbf{W}_m, \bar{\mathbf{K}}^{-1}_m, \check{\mathbf{W}}_m\Big]\Big]\Big] \\
&+ \mathcal{G}\Big[\mathbf{V}_\theta, \mathbf{V}^c_\theta, \mathcal{G}\Big[\bar{\mathbf{K}}^c_m, \mathbf{V}_\theta, \mathcal{G}\Big[\mathbf{W}_m, \bar{\mathbf{K}}^c_m, \check{\mathbf{W}}_m\Big]\Big]\Big] \\
&+ \mathcal{G}\Big[\mathbf{V}_\theta, \mathbf{V}^c_\theta, \mathcal{G}\Big[\bar{\mathbf{K}}_m, \mathbf{V}_\theta, \mathcal{G}\Big[\bar{\mathbf{K}}^{-1}_m, \bar{\mathbf{K}}_m, \mathcal{G}\Big[\mathbf{W}_m, \bar{\mathbf{K}}^{-1}_m, \check{\mathbf{W}}_m\Big]\Big]\Big]\Big]
\end{aligned} \tag{3.32}$$

From the definition of $\bar{\mathbf{K}}_m$, the gradient transformation of $\mathbf{W}_m$ with respect to $\mathbf{V}^c_\theta$ given $\check{\mathbf{W}}_m$ is zero if the $\theta^{th}$ output does not belong to the output group $\Theta_m$ (does not belong to the $m^{th}$ model). The following intermediate values are defined to simplify the expression for the constrained error gradient.

$$\begin{aligned}
\check{\mathbf{V}}^c_\theta &= \check{\mathbf{V}}^c_\theta + \mathcal{G}\Big[\bar{\mathbf{K}}^c_m, \mathbf{V}^c_\theta, \mathcal{G}\Big[\mathbf{W}_m, \bar{\mathbf{K}}^c_m, \check{\mathbf{W}}_m\Big]\Big] \\
&+ \mathcal{G}\Big[\bar{\mathbf{K}}_m, \mathbf{V}^c_\theta, \mathcal{G}\Big[\bar{\mathbf{K}}^{-1}_m, \bar{\mathbf{K}}_m, \mathcal{G}\Big[\mathbf{W}_m, \bar{\mathbf{K}}^{-1}_m, \check{\mathbf{W}}_m\Big]\Big]\Big]
\end{aligned} \tag{3.33}$$

$$\begin{aligned}
\check{\mathbf{V}}_\theta &= \check{\mathbf{V}}_\theta + \mathcal{G}\Big[\bar{\mathbf{K}}^c_m, \mathbf{V}_\theta, \mathcal{G}\Big[\mathbf{W}_m, \bar{\mathbf{K}}^c_m, \check{\mathbf{W}}_m\Big]\Big] \\
&+ \mathcal{G}\Big[\bar{\mathbf{K}}_m, \mathbf{V}_\theta, \mathcal{G}\Big[\bar{\mathbf{K}}^{-1}_m, \bar{\mathbf{K}}_m, \mathcal{G}\Big[\mathbf{W}_m, \bar{\mathbf{K}}^{-1}_m, \check{\mathbf{W}}_m\Big]\Big]\Big]
\end{aligned} \tag{3.34}$$

In these definitions, $m$ is the number of the model to which the $\theta^{th}$ output belongs. Substituting equations 3.32, 3.33, and 3.34 into equation 3.31 and applying the gradient transformation yields a simplified expression for the constrained error gradient with respect to the output weights which depends on the intermediate values defined above.

$$\hat{\mathbf{V}}^c_\theta = \check{\mathbf{V}}^c_\theta + \check{\mathbf{V}}_\theta \left[\mathbf{S}^c_\theta \left[\mathbf{S}_\theta\right]^{-1}\right]^T \tag{3.35}$$

The gradient transformations in equations 3.33 and 3.34 can be evaluated using the techniques shown in appendix B. The constrained error gradient with respect to the auxiliary input weights can be written as

$$\begin{aligned}\hat{\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)} &= \breve{\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)} + \sum_{\theta_2} \mathcal{G}[\mathbf{V}_{\theta_2}, {\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)}, \check{\mathbf{V}}_{\theta_2}] \\ &+ \sum_{m_2} \mathcal{G}[\mathbf{W}_{m_2}, {\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)}, \check{\mathbf{V}}_{\theta_2}]\end{aligned} \tag{3.36}$$

This can be expanded as

$$\begin{aligned}\hat{\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)} &= \breve{\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)} + \mathcal{G}\left[\mathbf{S}_\theta, {\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)}, \mathcal{G}\left[\mathbf{V}_\theta, \mathbf{S}_\theta, \check{\mathbf{V}}_\theta\right]\right] \\ &+ \sum_{\theta_2} \mathcal{G}\left[\mathbf{S}^c_{\theta_2}, {\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)}, \mathcal{G}\left[\mathbf{V}_{\theta_2}, \mathbf{S}^c_{\theta_2}, \check{\mathbf{V}}_{\theta_2}\right]\right] \\ &+ \sum_{m_2} \mathcal{G}\left[\bar{\mathbf{K}}_{m_2}, {\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)}, \mathcal{G}\left[\mathbf{W}_{m_2}, \bar{\mathbf{K}}_{m_2}, \check{\mathbf{W}}_{m_2}\right]\right] \\ &+ \sum_{m_2} \mathcal{G}\left[\bar{\mathbf{K}}^c_{m_2}, {\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)}, \mathcal{G}\left[\mathbf{W}_{m_2}, \bar{\mathbf{K}}^c_{m_2}, \check{\mathbf{W}}_{m_2}\right]\right] \\ &= \breve{\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)} + \mathcal{G}\left[\mathbf{S}_\theta, {\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)}, \mathcal{G}\left[[\mathbf{S}_{\theta_2}]^{-1}, \mathbf{S}_{\theta_2}, -\left[\mathbf{V}^c_{\theta_2}\mathbf{S}^c_{\theta_2}\right]^T \check{\mathbf{V}}_\theta\right]\right] \\ &+ \sum_{\theta_2} \mathcal{G}\left[\mathbf{S}^c_{\theta_2}, {\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)}, -\left[\mathbf{V}^c_\theta\right]^T \check{\mathbf{V}}_{\theta_2}\left[[\mathbf{S}_\theta]^{-1}\right]^T\right] \\ &+ \sum_{m_2} \mathcal{G}\left[\bar{\mathbf{K}}_{m_2}, {\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)}, \mathcal{G}\left[[\bar{\mathbf{K}}_{m_2}]^{-1}, \bar{\mathbf{K}}_{m_2}, \left[\bar{\mathbf{E}}_{m_2} - \mathbf{W}^c_{m_2}\bar{\mathbf{K}}^c_{m_2}\right]^T \check{\mathbf{W}}_{m_2}\right]\right] \\ &+ \sum_{m_2} \mathcal{G}\left[\bar{\mathbf{K}}^c_{m_2}, {\mathbf{W}_{\mathbf{A}}}^{(*,H_\theta,*)}, -\left[\mathbf{W}^c_{m_2}\right]^T \check{\mathbf{W}}_{m_2}\left[[\bar{\mathbf{K}}^c_{m_2}]^{-1}\right]^T\right]\end{aligned} \tag{3.37}$$

The remaining gradient transformations in equation 3.37 are carried out in as specified in appendix B.

Although the output bias does not directly effect the output of the neural network, it does effect the constrained weights. Thus, the output bias has an indirect effect on the neural network error and the effect must be incorporated into the training. The constrained error gradient with respect to the output bias is solved as

$$\begin{aligned}\hat{\mathbf{b}} &= \breve{\mathbf{b}} + \sum_{\theta} \mathcal{G}[\mathbf{V}_{\theta}, \mathbf{b}, \check{\mathbf{V}}_{\theta}] \\ &= -\sum_{\theta} \check{\mathbf{V}}_{\theta} \left[ [\mathbf{S}_{\theta}]^{-1} \right]^{T}\end{aligned} \tag{3.38}$$

The gradient transformation is an incredibly powerful tool in tracing the effects of the values in an entire matrix throughout a series of construction functions. Once the constrained error gradient has been obtained, gradient-based training, such as modified resilient backpropagation, can be used to update the values of the unconstrained weights.

### 3.3.2 Modified Resilient Backpropagation

As mentioned in section 2.5.1, backpropagation uses the error gradient to adjust the values of the neural network weights. The application of gradient-based training to a set of weights can be discussed for the general case, where the constrained error gradient is known. Let **U** represent an arbitrary weight matrix associated with the neural network. The backpropagation algorithm updates the values of the weights during the $k^{th}$ epoch according to the following rule

$$\mathbf{U}^{k} = \mathbf{U}^{k-1} - \eta \hat{\mathbf{U}}^{k-1} \tag{3.39}$$

where $\mathbf{U}^0$ is the initialized value of **U** and $\eta$ is a parameter called the *learning parameter* which has a fixed value between 0 and 1. This rule allows training to stop when the error gradient becomes small. However, there are some shortcomings to this approach. While a small gradient can be indicative of good weight values, it can also indicate that the sigmoidal functions are saturated. Input parameters with small magnitudes either in the input layer or in the preceding sigmoidal layer also can cause small derivatives. These small derivatives cause the associated weights to remain relatively unchanged.

The *resilient backpropagation* (RPROP) technique overcomes these deficiencies by adapting the weights based on the sign of the gradient and the increment of the previous weight adaptation [22]. This eliminates the effect of the gradient magnitude on the update. The RPROP algorithm adapts weights according to the following rule

$$\mathbf{U}^k = \mathbf{U}^{k-1} + \Delta\mathbf{U}^k \tag{3.40}$$

where $\Delta\mathbf{U}^k$ is defined by the following procedure.

$$\Delta^k_{(i,j)} = \begin{cases} \eta^+ \quad \Delta^{k-1}_{(i,j)}, & \text{if } \hat{\mathbf{U}}^{k-1}_{(i,j)}\hat{\mathbf{U}}^k_{(i,j)} > 0; \\ \eta^- \quad \Delta^{k-1}_{(i,j)}, & \text{if } \hat{\mathbf{U}}^{k-1}_{(i,j)}\hat{\mathbf{U}}^k_{(i,j)} < 0; \\ \qquad \Delta^{k-1}_{(i,j)}, & \text{if } \hat{\mathbf{U}}^{k-1}_{(i,j)}\hat{\mathbf{U}}^k_{(i,j)} = 0. \end{cases} \tag{3.41}$$

$$\Delta\mathbf{U}^k_{(i,j)} = \begin{cases} \text{sign}\Big[\hat{\mathbf{U}}^k_{(i,j)}\Big]\Delta^k_{(i,j)}, & \text{if } \hat{\mathbf{U}}^{k-1}_{(i,j)}\hat{\mathbf{U}}^k_{(i,j)} \geq 0; \\ -\Delta\mathbf{U}^{k-1}_{(i,j)}, & \text{if } \hat{\mathbf{U}}^{k-1}_{(i,j)}\hat{\mathbf{U}}^k_{(i,j)} < 0; \end{cases} \tag{3.42}$$

Note that the sign[ ] operator returns 1 if argument is positive, $-1$ if the argument is negative, and 0 otherwise. To aid with convergence, an additional rule is added. If the error at stage $k$ is greater than 110% of than the error at stage $k-1$, then the network reverts to stage $k-1$ and a new $\Delta\mathbf{U}^k$ is selected as

$$\Delta\mathbf{U}^k = s_d(\Delta\mathbf{U}^k)_{old} \tag{3.43}$$

where $s_d$ is a positive number less than 1 and is referred to as the scale-down factor. The scale-down factor scales the weight adaptation so that the minimization search remains local. Likewise, a second rule states that if the error at stage $k$ is less than 0.05% lower than the error at stage $k-1$, the network reverts to stage $k-1$ and a new $\Delta\mathbf{U}^k$ is selected as

$$\Delta\mathbf{U}^k = s_u(\Delta\mathbf{U}^k)_{old} \tag{3.44}$$

where $s_u$ is a positive number greater than 1 and is referred to as the scale-up factor. This factor serves to accelerate training when the step size is too small. Because these scaling factors can be applied directly to the old $\Delta\mathbf{U}^k$, the constrained error gradient does not need to be recalculated for a scale-up or scale-down step, which results in significant computational savings. Additionally, because incremental training focuses on only the current training set, too many training epochs can result in a network that forgets previous training sets. This is often referred to as overtraining. To prevent overtraining, the network training algorithm stops after the error is reduced by ten percent.

For all elements of $\Delta^0$, Riedmiller and Braun suggest picking a fixed value, citing the robustness of the RPROP algorithm [22]. Ferrari [17] notes that this is equivalent to disregarding initial information stored in the network weights and offers a modified RPROP algorithm in which values for $\Delta^0$ are based on the initial values of the weights.

$$\Delta^0 = f_u\,|\mathbf{U}| + f_0 \tag{3.45}$$

However, the weights to be modified are the unconstrained weights and, by definition, they have initial values of zero. Therefore, the modification suggested by Ferrari is reduced, in this case, to picking an arbitrary value. While RPROP is robust in regards to initial parameter choice, well-chosen parameters can significantly reduce the number of epochs required for satisfactory training. A revised method of picking meaningful values for $\Delta^0$ draws upon the original backpropagation algorithm. The original backpropagation algorithm remains local by choosing the magnitudes of the first adjustment (and all subsequent adjustments) based upon gradient information. Thus, choosing $\Delta^0$ based on the error gradient will preserve the local information contained in the weights. The magnitude of $\Delta^0$ should also depend on the magnitude of the constrained weights since the two sets of weights are related. The following rule is proposed for determining values for $\Delta^0$.

$$\Delta^0 = \eta\mu \left|\hat{\mathbf{U}} + \mathbf{R}_u\right| \tag{3.46}$$

where $\mu$ is the ratio of the sum of all elements of $|\mathbf{U}|$ to the sum of all elements of $\left|\hat{\mathbf{U}}\right|$, $\mathbf{R}_u$ is the zero-replacement matrix, and $\eta$ is the learning rate borrowed from backpropagation. Each element in the zero-replacement matrix, $\mathbf{R}_u$, whose corresponding element of $\hat{\mathbf{U}}$ is zero has a positive value, $R_u$, which ensures that the value of each element of $\Delta^0$ is positive. The value added has an order of magnitude equal to the average order of magnitude of the non-zero elements of $\hat{\mathbf{U}}$.

$$R_u = e^{\frac{1}{n_u}\sum_i \ln\left|\hat{\mathbf{U}}_{c(i)}\right|} \tag{3.47}$$

where $\mathbf{U}_c$ is a vector composed of all the constrained weights found in $\mathbf{U}$ and $n_u$ is the number of constrained weights found in $\mathbf{U}$.

Once $\Delta^0$ is determined, the algorithm runs according to the rules outlined in equations 3.41 and 3.42. Riedmiller and Braun point out that computation devices cannot accurately display every number and that if $\Delta$ becomes too large, numerical errors induced by the computing device may compromise the training algorithm. They suggest that to overcome this problem, a maximum be imposed on the elements of $\Delta$. Riedmiller and Braun suggest that this maximum value be 50. However, due to the potentially large magnitudes of weight values in the neural networks, this limit can effectively halt training. For example, if the values of the constrained weights are on the order of $10^8$ and the elements of $\Delta$ have a value of 50, then the new weights will not change significantly. However, if chosen well, a maximum for $\Delta$ is beneficial in that they prevent the RPROP algorithm from taking too large of a step and skipping local minima. A $\Delta$ maximum of 50 is hardly useful in this regard when the average magnitude of the constrained weights is $10^{-7}$. Therefore, the *Delta* maximum should be a function of the constrained weights. The following function has been found to work well.

$$\Delta \mathbf{U}_{limit} = 0.1\frac{\bar{u}}{n_u} \tag{3.48}$$

where $\bar{u}$ is the average of the elements of $|\mathbf{U}_c|$. These modifications to the RPROP algorithm work for both the input and output unconstrained weights and ensure quicker, localized training.

### 3.3.3 Satisfying Network Constraints

The final stage of every epoch is the network constraining stage. In this stage, the construction functions are used to determine the values of the constrained weights which cause the network weight values to satisfy the constraints imposed on the network performance, based on the linear gain-scheduled controller. From section 3.1, the construction functions are

$$\mathbf{V}_\theta = -\left[\mathbf{V}_\theta^c \mathbf{S}_\theta^c + \mathbf{B}_{(\theta,*)}\right]\left[\mathbf{S}_\theta\right]^{-1}$$

$$\mathbf{W}_m = \left[\bar{\mathbf{E}}_m - \mathbf{W}_m^c \bar{\mathbf{K}}_m^c\right]\left[\bar{\mathbf{K}}_m\right]^{-1}$$

At this point the error function is evaluated with the new weights, the current input and the current target. If the error is acceptably small and all other stopping conditions necessary are satisfied, then the training algorithm ends. If the error level is not acceptable or other conditions are not satisfied, then a new training epoch begins based on these new network weights. Throughout the entire process, the network satisfies the constraint equations 2.39 and 2.40.

## 3.4 Chapter Summary

In this chapter, construction functions are developed to define dependencies among the network weights. The application of these construction functions yields a neural network whose parameters satisfy the network constraint equations. A method which makes use of dual-point hyperspherical initialization provides initial values for the auxiliary input weights. Initial bias values are picked based on their influence on the linearity of the sigmoidal functions. Unconstrained weights are initialized as zero and initial constrained weights are determined from the construction functions. A constrained error gradient is determined using a new operator called the gradient transformation. The constrained error gradient is used in conjunction with a new modified version of resilient backpropagation to optimize the network weights with respect to an input/target pair. The use of the construction functions is built into this process to ensure that the network adapts subject to the zero-output and gradient constraints.

# Chapter 4

# Software Implementation

The new procedures introduced in the previous chapter are tested here, using a six-degree-of-freedom Matlab simulation of a business jet. The aircraft neural control system is modeled after a gain-scheduled proportional-integral (PI) controller. The neural controller is obtained by using the gains specified in the PI controller to initialize and constrain the neural networks. The software models coupled longitudinal and lateral-directional dynamic effects. For steady, level flight these coupled effects are small and linearization yields decoupled models. Therefore, independent linear controllers are designed for the longitudinal and lateral-directional models and the neural controller is constructed from the two independent models. Because the coupled dynamic effects are present in the simulation, the neural controller has the opportunity to improve its performance.

The software models the dynamics of the business jet as

$$\dot{\mathbf{x}} = f(\mathbf{x}, \tilde{\mathbf{a}}, \mathbf{u}) \tag{4.1}$$

The longitudinal, $(\ )_L$, aircraft model can be written as

$$\dot{\mathbf{x}}_L = f_L(\mathbf{x}_L, \tilde{\mathbf{a}}, \mathbf{u}_L) \tag{4.2}$$

and the lateral-directional, $(\ )_{LD}$, aircraft model can be written as

$$\dot{\mathbf{x}}_{LD} = f_{LD}(\mathbf{x}_{LD}, \tilde{\mathbf{a}}, \mathbf{u}_{LD}) \tag{4.3}$$

Both systems of dynamic equations are based on mathematical models, full-scale wind tunnel data, and the physical and performance characteristics of an early twin-jet configuration [23]. The longitudinal state vector, $\mathbf{x}_L = [\ V \quad \gamma \quad q \quad \theta\ ]^T$ is comprised of the velocity, $V$ (m/s), the path angle, $\gamma$ (rad), the pitch rate, $q$ (rad/s), and the pitch angle, $\theta$ (rad). The lateral-directional state vector, $\mathbf{x}_{LD} = [\ r \quad \beta \quad p \quad \mu\ ]^T$ is comprised of the yaw rate, $r$ (rad/s), the sideslip angle, $\beta$ (rad), the roll rate, $p$ (rad/s), and the bank angle, $\mu$ (rad). The scheduling vector, $\tilde{\mathbf{a}} = [\ V \quad H\ ]^T$, is comprised of the velocity and the altitude, $H$ (km). The longitudinal control vector, $\mathbf{u}_L = [\ \delta T \quad \delta S\ ]^T$, is comprised of the control values for the throttle, $\delta T$ (%), and the stabilator, $\delta S$ (rad). The lateral-directional control vector, $\mathbf{u}_{LD} = [\ \delta A \quad \delta R\ ]^T$, is comprised of the control values for the aileron ,$\delta A$ (rad), and the rudder, $\delta R$ (rad). Details concerning the equations of motion are provided in appendix C.

The aircraft is allowed to fly anywhere in its operational domain which is defined as the set of points $(\mathbf{x}_e, \tilde{\mathbf{a}})$ for which there exists a control vector $\mathbf{u}_e$ such that the aircraft can be brought to equilibrium. Typically, the flight envelope is determined for steady, level flight conditions by considering the stall speed, the thrust/power required and available, compressibility effects, and the maximum allowable dynamic pressure to prevent structural damage [23]. In this approach, the design envelope used to initialize and constrain the neural controller is the entire flight envelope or a subset of its operating conditions.

## 4.1 Design Envelope

One of the major factors considered when selecting the design envelope and the design points therein is the expected operating region of the airplane. Once the design envelope is chosen, it is used to initialize and constrain the neural controller. Thus, another important consideration is the computational power available. As pointed out in appendix B, the processes of training the neural controller is $O(n^5)$, where $n$ is related to the number of design points. Thus, the number of design points directly influences the time required to complete a training epoch. If too many design points are used, the training will require infeasible computational time.

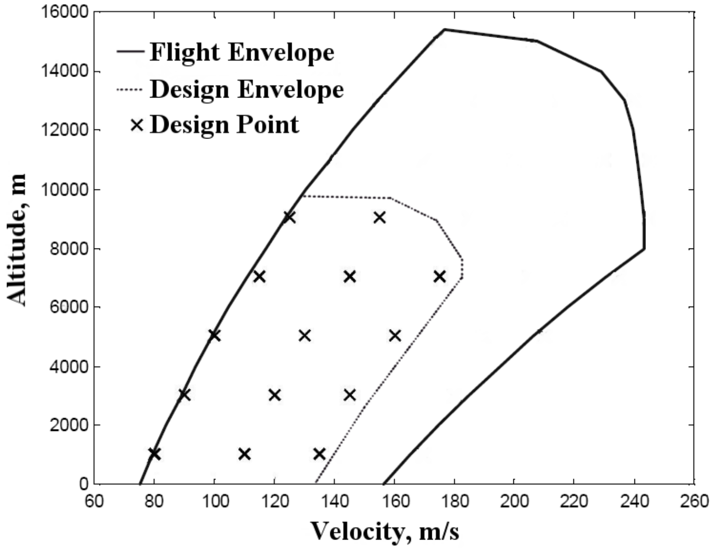

**Figure 4.1**: Flight envelope, design envelope and design points

Figure 4.1 shows the flight envelope and a smaller design envelope picked for the initialization and for the constraints used for the neural controller. By using the smaller envelope to initialize the controller, the remainder of the flight envelope can be used to test the extrapolating capability of the adaptive neural controller. The number of design points is effectively limited by the computational device used to train the neural networks. This device is a desktop computer with an Intel Pentium-4 3.06GHz central processing unit, 1.25GB of system memory, and a 533MHz front-side bus running Matlab 6.5 R13. A selection of 14 design points was found to have an average training epoch time of around one second for the action network. This time interval is unreasonable for real-time control of an aircraft. However, it is reasonable for the timely completion of the aircraft simulation. Section 4.5 proposes methods for reducing the epoch time through parallelization for real-time application. The 14 design points are then selected to cover the design envelope as shown in figure 4.1

| $k$ | $(\#)$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $V$ | $(m/s)$ | 80 | 110 | 135 | 90 | 120 | 145 | 100 |
| $H$ | $(m)$ | 1000 | 1000 | 1000 | 3000 | 3000 | 3000 | 5000 |

| $k$ | $(\#)$ | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|
| $V$ | $(m/s)$ | 130 | 160 | 115 | 145 | 175 | 125 | 155 |
| $H$ | $(m)$ | 5000 | 5000 | 7000 | 7000 | 7000 | 9000 | 9000 |

**Table 4.1**: The 14 design points used to initialized and constrain the neural controller

and table 4.1.

## 4.2 Linear Control

Equations 4.2 and 4.3 are linearized for each of the 14 design points as outlined in section 2.1. This is done using a built-in numerical Jacobian function in Matlab called **numjac**. Linearization yields the following approximate perturbation models

$$\Delta\dot{\mathbf{x}}_L = \mathbf{F}_L(\tilde{\mathbf{a}}_k)\Delta\mathbf{x}_L + \mathbf{G}_L(\tilde{\mathbf{a}}_k)\Delta\mathbf{u}_L \tag{4.4}$$

$$\Delta\dot{\mathbf{x}}_{LD} = \mathbf{F}_{LD}(\tilde{\mathbf{a}}_k)\Delta\mathbf{x}_{LD} + \mathbf{G}_{LD}(\tilde{\mathbf{a}}_k)\Delta\mathbf{u}_{LD} \tag{4.5}$$

The Jacobian matrices $\mathbf{F}$ and $\mathbf{G}$ can be use to obtain the linear control gains $(\mathbf{C}_k)_L$ and $(\mathbf{C}_k)_{LD}$ along with the Riccati matrices $(\mathbf{P}_k)_L$ and $(\mathbf{P}_k)_{LD}$ (section 2.2.1). These matrices are then used to generate the target output gradients for the neural networks (section 2.3).

### 4.2.1 Proportional-Integral Control

As mentioned in chapter 2, the neural controller is designed to improve upon a gain-scheduled controller. The gain-scheduled design selected to motivate the neural controller is the proportional-integral (PI) controller. Since there are two independent models, a PI controller can be constructed for each model. Because the process is the same for the longitudinal and lateral-directional models, the subscripts $(\;)_L$ and $(\;)_{LD}$ are generally omitted.

The objective of a proportional-integral controller is to minimize the quadratic cost function

$$\begin{aligned} J &= \int_0^{t_f} L(\tilde{\mathbf{x}}_a(t), \tilde{\mathbf{u}}(t), t) dt \\ &= \frac{1}{2} \int_{t_0}^{t_f} \left( \tilde{\mathbf{x}}_a^T \mathbf{Q}_a \tilde{\mathbf{x}}_a + 2\tilde{\mathbf{x}}_a^T \mathbf{M}_a \tilde{\mathbf{u}} + \tilde{\mathbf{u}}^T \mathbf{R}_a \tilde{\mathbf{u}} \right) dt \end{aligned} \tag{4.6}$$

with respect to $\tilde{\mathbf{u}}$. The augmented state deviation, $\tilde{\mathbf{x}}_a$, includes the deviation of the state from the commanded state,

$$\tilde{\mathbf{x}} = \mathbf{x} - \mathbf{x}_c \tag{4.7}$$

and the time integral of the system output error, $\xi$ such that

$$\tilde{\mathbf{x}}_a = \left[ \tilde{\mathbf{x}}^T \; \xi^T \right]^T \tag{4.8}$$

The system output error is defined as $\tilde{\mathbf{y}_s} = \mathbf{y}_s - \mathbf{y}_c$ where $\mathbf{y}_c$ is the commanded system output. The control deviation is defined as the difference between the control values and the control values which trim the aircraft at the commanded state.

$$\tilde{\mathbf{u}} = \mathbf{u} - \mathbf{u}_c \tag{4.9}$$

The commanded state and the corresponding control values can be obtained from the commanded system output. The linearized systems are at equilibrium when

$$\begin{bmatrix} \mathbf{0} \\ \Delta \mathbf{y}_c \end{bmatrix} = \begin{bmatrix} \mathbf{F} & \mathbf{G} \\ \mathbf{H}_x & \mathbf{H}_u \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_c \\ \Delta \mathbf{u}_c \end{bmatrix} \tag{4.10}$$

Because there are as many control elements as system outputs, the system can be solved for the commanded state and the control values which trim the aircraft about the commanded state.

$$\begin{aligned}\left[\begin{array}{c}\Delta\mathbf{x}_c\\ \Delta\mathbf{u}_c\end{array}\right] &= \left[\begin{array}{cc}\mathbf{F} & \mathbf{G}\\ \mathbf{H}_x & \mathbf{H}_u\end{array}\right]^{-1}\left[\begin{array}{c}\mathbf{0}\\ \Delta\mathbf{y}_c\end{array}\right] = \left[\begin{array}{cc}\mathbf{B}_{11} & \mathbf{B}_{12}\\ \mathbf{B}_{21} & \mathbf{B}_{22}\end{array}\right]\left[\begin{array}{c}\mathbf{0}\\ \Delta\mathbf{y}_c\end{array}\right]\\ &= \left[\begin{array}{c}\mathbf{B}_{12}\Delta\mathbf{y}_c\\ \mathbf{B}_{22}\Delta\mathbf{y}_c\end{array}\right]\end{aligned} \tag{4.11}$$

The weighting matrices $\mathbf{Q}_a$, $\mathbf{M}_a$, and $\mathbf{R}_a$ are designed using *implicit model following* (IMF). IMF is a technique used to prompt the system to behave like a known, ideal model. In the case of the aircraft, the ideal system should satisfy established design criteria. The state Jacobian matrices for the ideal longitudinal and lateral-directional models are obtained from [17].

$$\mathbf{F}_{m_L} = \left[\begin{array}{cccc}-0.016 & -1.8066 & 0 & -8\\ 2\cdot 10^{-4} & -0.5 & 0 & 0.5\\ 0 & 5 & -1.7 & -5\\ 0 & 0 & 1 & 0\end{array}\right] \tag{4.12}$$

$$\mathbf{F}_{m_{LD}} = \left[\begin{array}{cccc}-2.4 & 8 & 0 & 0\\ -1 & -1.8 & 0 & 0\\ 0 & -2 & -2 & 0\\ 0 & 0 & 1 & 0\end{array}\right] \tag{4.13}$$

The cost function associated with following the dynamic response of the ideal system model is

$$J = \frac{1}{2}\int_{t_0}^{t_f}\left(\left[\Delta\dot{\mathbf{x}}_a - \Delta\dot{\mathbf{x}}_{a_m}\right]^T\mathbf{Q}_m\left[\Delta\dot{\mathbf{x}}_a - \Delta\dot{\mathbf{x}}_{a_m}\right] + \tilde{\mathbf{u}}^T\mathbf{R}_m\tilde{\mathbf{u}}\right)dt \tag{4.14}$$

where $\mathbf{Q}_m$ is the weighting matrix for the difference in the time derivatives of the states and $\mathbf{R}_m$ is the cost associated with the control usage. This cost function can be rewritten in the form of equation 4.6 if the cost-weighting matrices $\mathbf{Q}$, $\mathbf{R}$, and $\mathbf{M}$

are defined as

$$\mathbf{Q} = \left[\mathbf{F} - \mathbf{F}_m\right]^T \mathbf{Q}_m \left[\mathbf{F} - \mathbf{F}_m\right] \quad (4.15)$$

$$\mathbf{M} = \left[\mathbf{F} - \mathbf{F}_m\right]^T \mathbf{Q}_m \mathbf{G} \quad (4.16)$$

$$\mathbf{R} = \mathbf{G}^T \mathbf{Q}_m \mathbf{G} + \mathbf{R}_m \quad (4.17)$$

Then, the augmented weighting matrices can be defined in terms of the implicit model following weighting matrices [24] as

$$\mathbf{Q}_a = \begin{bmatrix} \mathbf{Q} & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_\xi \end{bmatrix} \quad (4.18)$$

$$\mathbf{M}_a = \begin{bmatrix} \mathbf{M} \\ \mathbf{0} \end{bmatrix} \quad (4.19)$$

$$\mathbf{R}_a = \mathbf{R} \quad (4.20)$$

The values for $\mathbf{Q}_m$, $\mathbf{R}_m$, and $\mathbf{Q}_\xi$ are obtained from [17] to provide a good compromise between the ideal model behavior and the intrinsic aircraft dynamic behavior.

$$\mathbf{Q}_{m_L} = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.21)$$

$$\mathbf{R}_{m_L} = \begin{bmatrix} 0.5 & 0 \\ 0 & 5 \end{bmatrix} \quad (4.22)$$

$$\mathbf{Q}_{\xi_L} = \begin{bmatrix} 1 & 0 \\ 0 & 10^3 \end{bmatrix} \quad (4.23)$$

$$\mathbf{Q}_{m_{LD}} = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.24)$$

$$\mathbf{R}_{m_{LD}} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad (4.25)$$

$$\mathbf{Q}_{\xi_{LD}} = \begin{bmatrix} 10 & 0 \\ 0 & 0.1 \end{bmatrix} \quad (4.26)$$

With $\mathbf{Q}_a$, $\mathbf{M}_a$, and $\mathbf{R}_a$ fully defined at each design point, the linear quadratic result from section 2.2.1 is used to obtain the linear control gain matrices and the Riccati matrices.

## 4.3 Neural Controller Initialization

The neural controller is initialized using the methodology outlined in section 3.2. The longitudinal model is the first model and the lateral-directional model is the second model. The desired value of the output bias is $10^9$, which provides ample linearity for the sigmoidal functions. The auxiliary input weights are chosen using the dual-point hyperspherical initialization method to determine the targets for the hidden layer output values. The dual-point hyperspherical initialization is initialized with the target values provided by Ferrari's procedure (see section 3.2.3). Upon completion of the neural network initialization phase, the outputs and output gradients of the network are compared with their targets. The mean square difference between the target outputs and the actual outputs at the design points is on the order of $10^{-7}$. Likewise, the mean square difference between the target output gradients and the actual output gradients at the design points is on the order of $10^{-28}$.

## 4.4 Target Selection and Training

Selection of the target follows from the theory presented in section 2.4. For the action network, the target is found by solving the Hamilton-Jacobi-Bellman equation. The HJB equation for the action neural network is

$$\tilde{\mathbf{x}}_a^{*T}\mathbf{M} + \tilde{\mathbf{u}}^{*T}\mathbf{R} + [\mathbf{NN}_C(\tilde{\mathbf{x}}_a)]^T \frac{\partial \mathbf{f}(\mathbf{x},\mathbf{u})}{\partial \tilde{\mathbf{u}}} = \mathbf{0} \tag{4.27}$$

The target for the action neural network is the value of $\tilde{\mathbf{u}}$ that satisfies the above equation. The Jacobian of the open loop dynamic system with respect to the states is calculated using the **numjac** Matlab function. Using another built-in Matlab function, called **fminsearch**, the mean square error between the left-hand-side and the zero vector can be minimized to provide the target value for the action network. However, the aircraft dynamics impose physical limitations on the control values and target control values which exceed these limits are not valid. Thus, any excessive target values are reduced appropriately. If the corresponding target control values are equal to the reduced control values, then corresponding error is set to zero during the neural network training. This step recognizes that the excessive values still results the target control value. Consider, for example, the thrust, which is limited to 100% of available thrust. There is no difference in cost when the neural network output is 125% as opposed to 150%. Both control values are reduced to 100%. If the target thrust value is 100%, the neural network does not need to train. However, if the target value for the thrust is 99%, then training to meet the desired target requires that the full errors of 26% and 51%, respectively, be considered.

The target for the critic network is found from equation 2.34 which can be rewritten as

$$\mathbf{NN}_C(\tilde{\mathbf{x}}_a(t)) = \left[\mathbf{NN}_C(\tilde{\mathbf{x}}_a(t+\Delta t))\right] \frac{\partial \tilde{\mathbf{x}}_a(t+\Delta t)}{\partial \tilde{\mathbf{x}}_a(t)} + \frac{d\Delta J}{d\tilde{\mathbf{x}}_a(t)} \tag{4.28}$$

where

$$\begin{aligned}\Delta J = \frac{1}{2}\int_t^{t+\Delta t} & \tilde{\mathbf{x}}_a^T \mathbf{Q}_a \tilde{\mathbf{x}}_a + 2\tilde{\mathbf{x}}_a^T \mathbf{M}_a \left[\mathbf{NN}_A(\tilde{\mathbf{x}}_a(t))\right] \\ & + \left[\mathbf{NN}_A(\tilde{\mathbf{x}}_a(t))\right]^T \mathbf{R}_a \left[\mathbf{NN}_A(\tilde{\mathbf{x}}_a(t))\right] dt\end{aligned} \tag{4.29}$$

In section 4.1 it is mentioned that the average epoch for the action network takes approximately one second. The average epoch for the critic network takes approximately 70 seconds. Rather than reduce the number of design points further (which would reduce the quality of the controller), the critic network is left unconstrained during simulation. This reduces epoch time to a fraction of a second because the

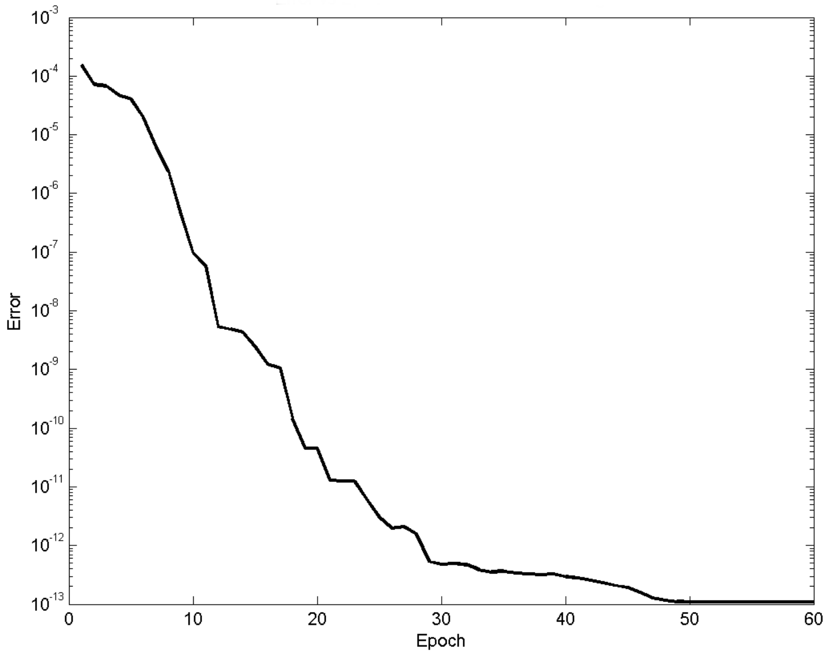

**Figure 4.2**: Error vs epoch in a typical action network training session

gradient transformation of an inverse matrix is removed from the training process.

Once the target has been chosen, training occurs via the modified RPROP algorithm outlined in section 3.3.2. The scale-up and scale-down factors found to work the best are 3.1 and 0.5 respectively. Using these factors results in 10 or fewer training epochs during the first training session (when $\Delta$ has not yet adapted to the network). The graph in figure 4.2 shows a training session for the action network. In this particular session, the network is allowed to train beyond the error tolerance level so that the effectiveness of the training algorithm itself could be observed. Normally an error tolerance of $10^{-4}$ is used for the action network and an error tolerance of 10 is used for the critic network.

## 4.5 Parallel Hardware Architecture

For significant time savings, the neural control system outlined in this thesis can be implemented using a parallel hardware architecture. Neural networks are, by definition, parallel structures. The computation of both their outputs and their unconstrained error gradients can be performed using parallel procedures. Many of the gradient transformation operations involve parallel operations. The most costly set of computations is the gradient transformation of the inverse of a matrix. From the definition of the gradient transformation of the inverse of an $n$ x $n$ matrix (see appendix B), this set of computations can be broken into parallel sets of computations. Every term being summed can be computed independently and there are $n^2$ terms that are summed. When computing the constrained error gradient, $n = n_p n_{o_m}$. For the neural controller used in this paper, $n = 28$ for all models in the action network and $n = 84$ for all models in the critic network. Therefore, the time to compute the gradient transformation of the inverse could theoretically be reduced by 99.87% on the action network by using 784 processors and by 99.98 on the critic network by using 7056 processors. These are the limits of parallel savings. More realistically, by using 8 processors, this step could theoretically be completed in 87.5% less time. Since many other steps are also highly parallel, they also would benefit from multiple processors. Thus, for a real-world application of this controller, it is necessary (with today's computing equipment) to have a multiprocessor machine to perform the controller update process in real time.

## 4.6 Chapter Summary

A proportional-integral controller is used to motivate the neural controller design. The controller is developed to mimic the performance of a previously published ideal model. The PI control gains and Riccati matrices at the design points are then used to initialize both the action network and the critic network. Numerical tests of the training algorithm reveal that it does extremely well at reducing the network error, while keeping the network constrained at the design points. Obtaining the constrained error gradient is computationally expensive. Fortunately, the neural networks and related methods are fairly parallel in nature and the controller adaptation can be significantly accelerated by a multiple processor implementation.

# Chapter 5

# Numerical Simulation Results

In this chapter, the performance of the neural control system obtained in chapter 4 is examined. To evaluate the performance of the training algorithm, the initialized neural controller is tested with and without the adaptation routine engaged. Without adaptation, the neural controller is simply a gain-scheduled controller where the neural network itself computes the interpolated control values. The performance of the control system is also compared to a linear controller optimized for the test point. The three types of controllers are tested at a design point (1), an interpolation point (2) and an extrapolation point (3) which are shown in figure 5.1.

At the design point, all three controllers should perform identically for small commands. At the interpolation point, the linear controller should outperform the neural controllers for small commands. The adaptive neural controller should perform on par to slightly better than the non-adapting neural controller for small commands. For larger commands, the adaptive neural controller should show significant improvement over the non-adapting neural controller. At the extrapolation point, the adaptive neural controller should perform much better than the non-adapting neural controller.

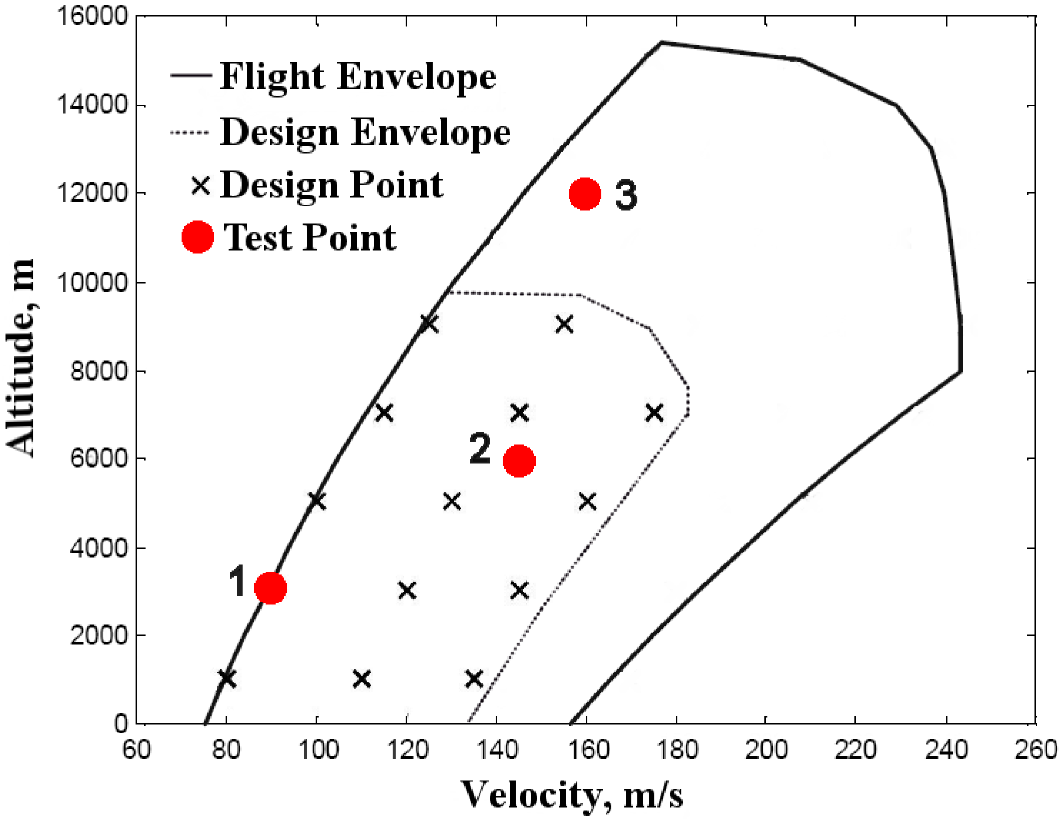

**Figure 5.1**: The flight envelope with design points and test points. Test points: (1) a design point, (2) an interpolation point, (3) an extrapolation point

## 5.1 Design Point

The system is tested at a design point to verify that the adapting neural controller, the non-adapting neural controller, and the linear controller all perform identically. Identical performance is expected because the initialization algorithm for the neural controllers specifically builds them to match the performance of the optimal linear controller at the design points. Because performance is optimal, the adapting neural controller is expected to refrain from adapting its weights. At the design point selected, the aircraft is initially flying 90$^m/_s$ at an altitude of 3000$m$.

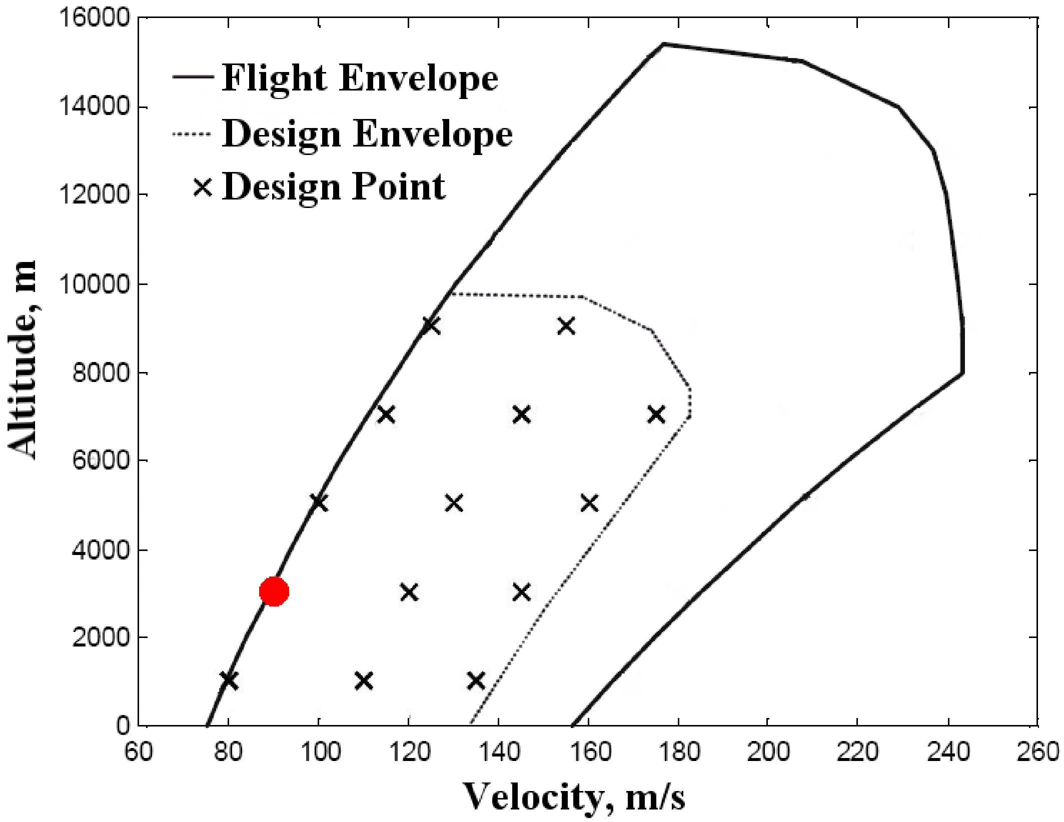

**Figure 5.2**: The design point [90$^m/_s$, 3000$m$] at which the current set of simulations take place.

### 5.1.1 Longitudinal Maneuver

The aircraft response is tested for a small longitudinal command. At time $t = 0$, the aircraft is given a command to increase speed to $92 {}^{m}\!/\!{}_{s}$ and climb at $2^{\circ}$. Because the neural controllers are built to match the linear controller at the design point, it is expected that the dynamic response, the control usage, and the cost accrued for the neural controllers and the linear controller will be identical. Figure 5.3 shows that the dynamic response is indeed identical. Figure 5.4 shows the control usage during the 10 seconds after the step command is issued. As expected, the controllers all have identical control usage. Figure 5.5 shows the incremental and cumulative costs associated with the flight. The incremental cost value is the amount of cost, $\Delta J$, accrued during the prior 0.1 seconds. As can be seen in the figure, all three controllers accrue the same amount of cost. This test case is a validation that the three controllers perform identically upon initialization in the longitudinal plane.

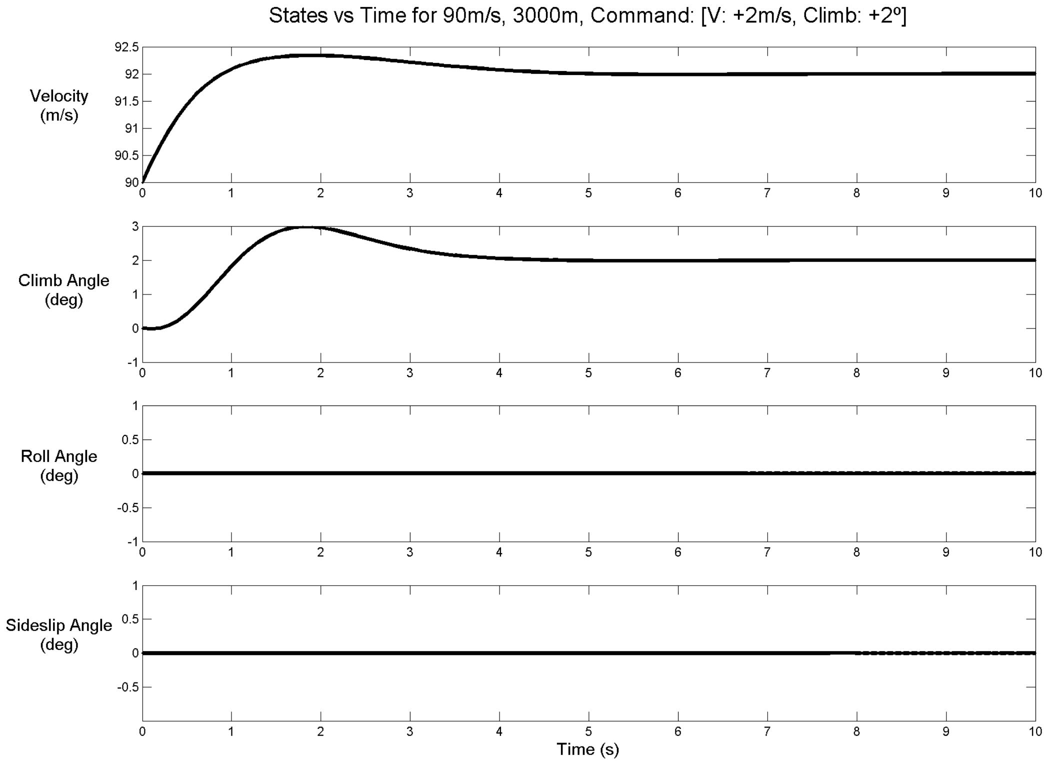

**Figure 5.3**: System output values at the design point with a longitudinal command (at [$90^m/_s$, $3000m$] with a command of [$+2^m/_s$, $+2°$, $+0°$, $+0°$] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line. All lines overlap in this graph.

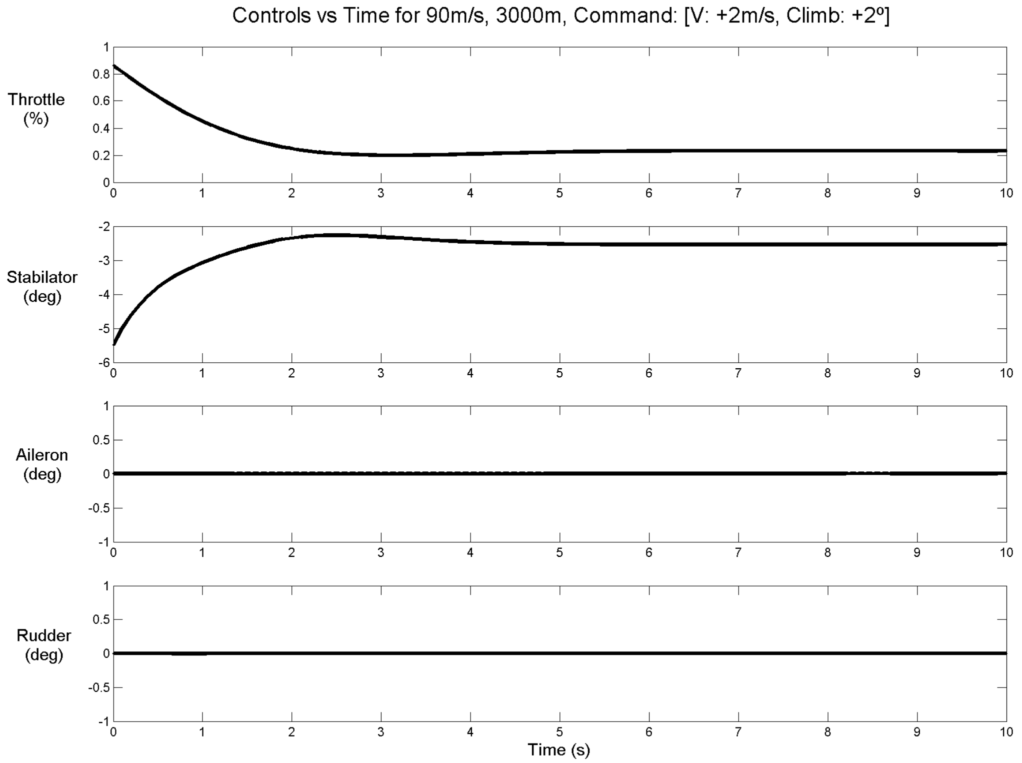

**Figure 5.4**: System control values at the design point with a longitudinal command (at [$90^m/_s$, $3000m$] with a command of [$+2^m/_s$, $+2°$, $+0°$, $+0°$] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line. All lines overlap in this graph.

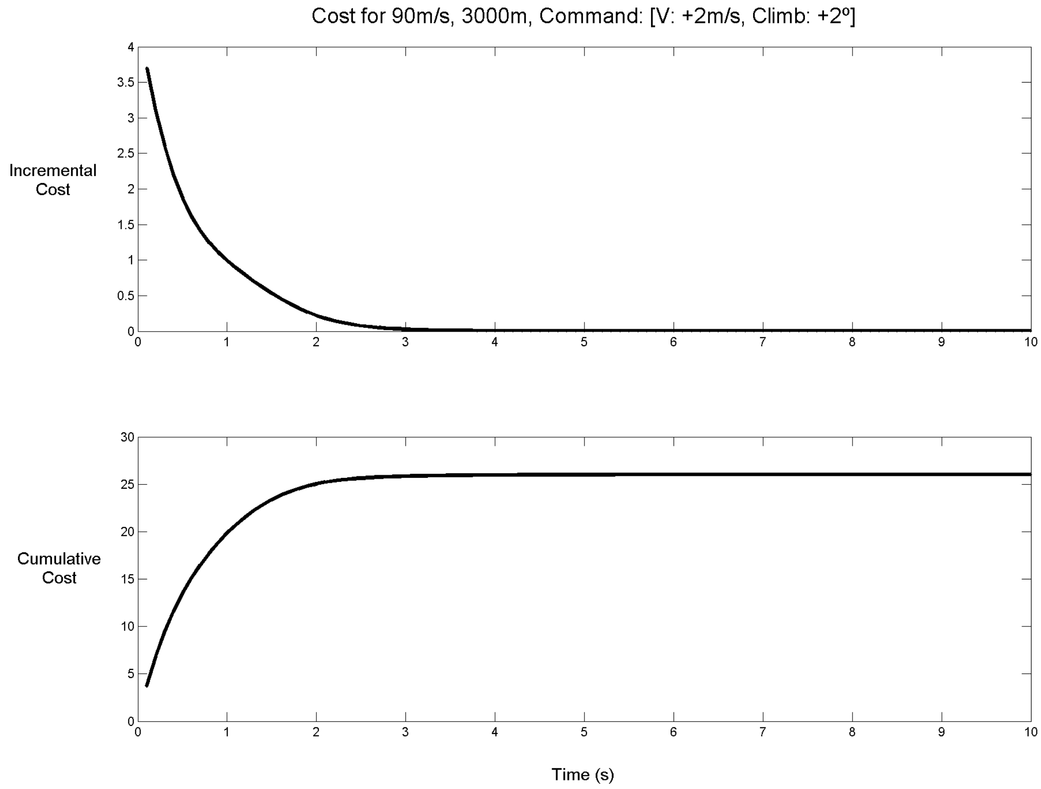

**Figure 5.5**: Incremental and cumulative cost at the design point with a longitudinal command (at [$90^m/_s$, $3000m$] with a command of [$+2^m/_s$, $+2°$, $+0°$, $+0°$] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line. All lines overlap in this graph.

### 5.1.2 Lateral-Directional Maneuver

The aircraft response is tested using a small lateral-directional command at a design point. As in the previous case, the three controllers are expected to perform identically. At time $t = 0$, the aircraft is given a command to bank at $5^\circ$ with a sideslip angle of $5^\circ$. Figure 5.6 shows the system output, which is identical for all controllers. Figure 5.7 shows the control usage during the 10 seconds after the step command is issued. Figure 5.8 shows the incremental and cumulative costs associated with the flight. As can be seen in the figures, the neural controllers and the linear controller perform identically, validating the initialization method in the lateral and directional planes.

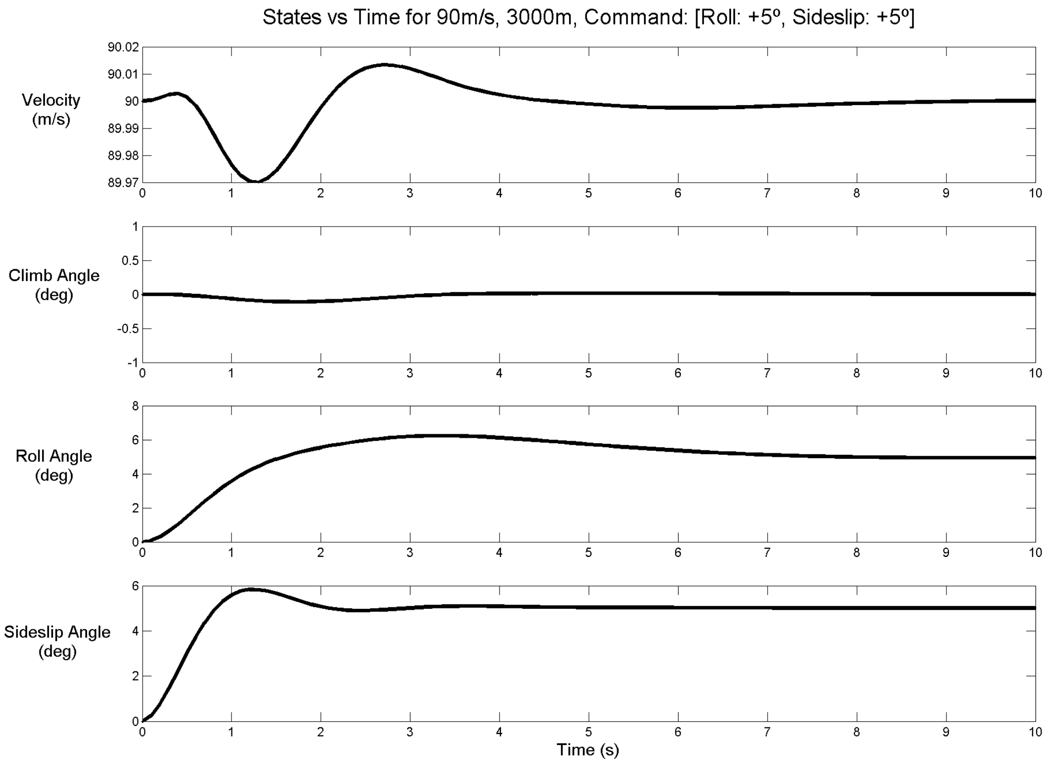

**Figure 5.6**: System output values at the design point with a lateral-directional command (at $[90{}^{m}\!/_{s},\ 3000m]$ with a command of $[+0{}^{m}\!/_{s},\ +0°,\ +5°,\ +5°]$ ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line. All lines overlap in this graph.

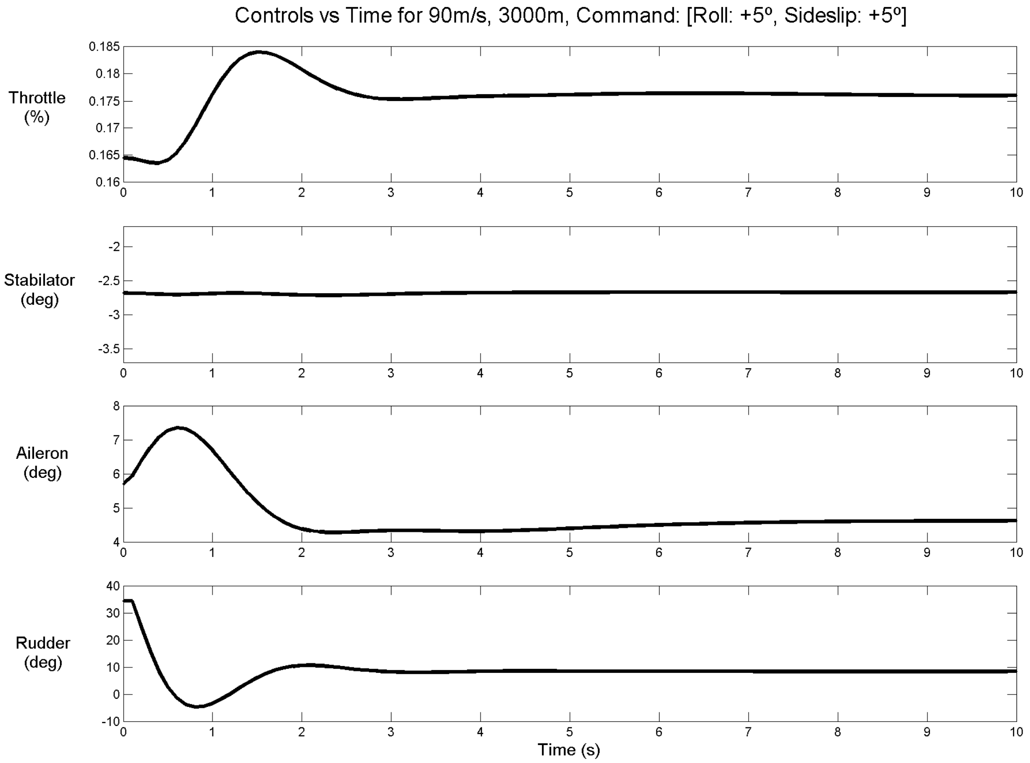

**Figure 5.7**: System control values at the design point with a lateral-directional command (at [$90 m/s$, $3000 m$] with a command of [$+0 m/s$, $+0°$, $+5°$, $+5°$] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line. All lines overlap in this graph.

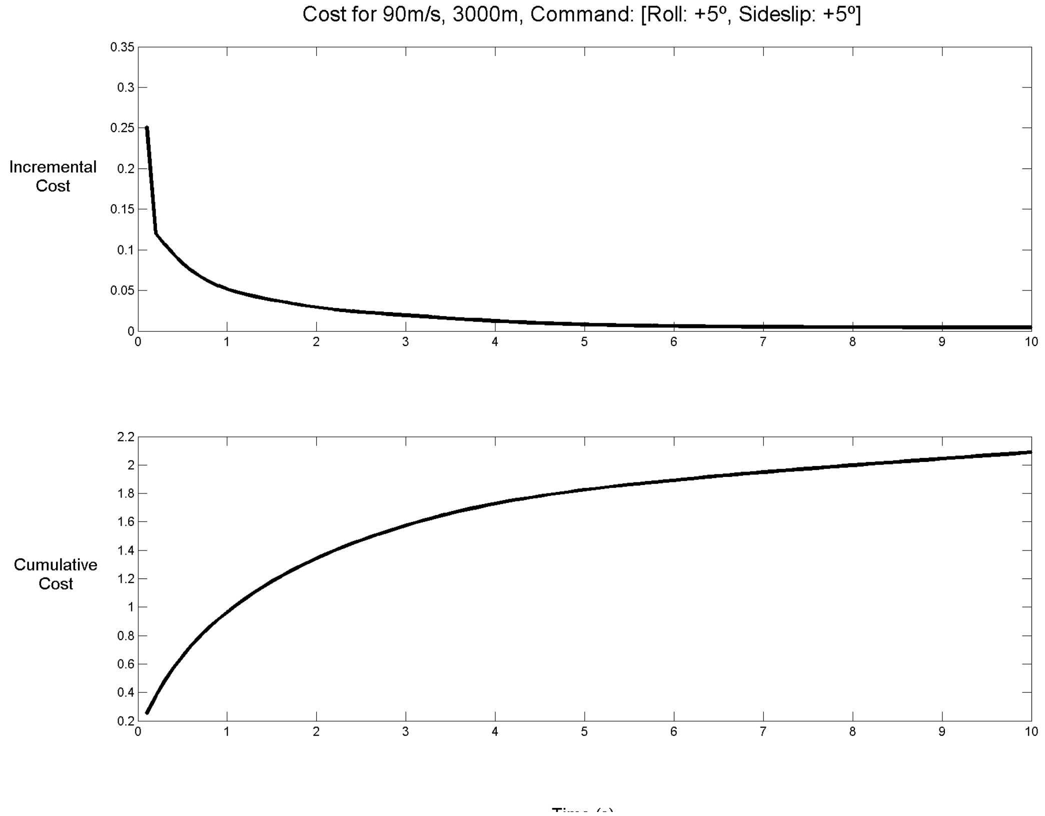

**Figure 5.8**: Incremental and cumulative cost at the design point with a lateral-directional command (at [90$^m/_s$, 3000$m$] with a command of [+0$^m/_s$, +0°, +5°, +5°] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line. All lines overlap in this graph.

### 5.1.3 Coupled Longitudinal-Lateral-Directional Maneuver

The aircraft response is tested using when given a small coupled command at a design point. As with the previous two cases, it is expected that the three controllers perform identically. At time $t = 0$, the aircraft is given a command to increase speed to $93^{m}/_{s}$, nose down at $-2°$, and bank at $5°$. Figure 5.9 shows the system output. Figure 5.10 shows the control usage during the 10 seconds after the step command is issued. Figure 5.11 shows the incremental and cumulative costs associated with the flight. As can be seen in the figures, the neural controllers and the linear controller perform identically, validating the initialization method for small coupled maneuvers.

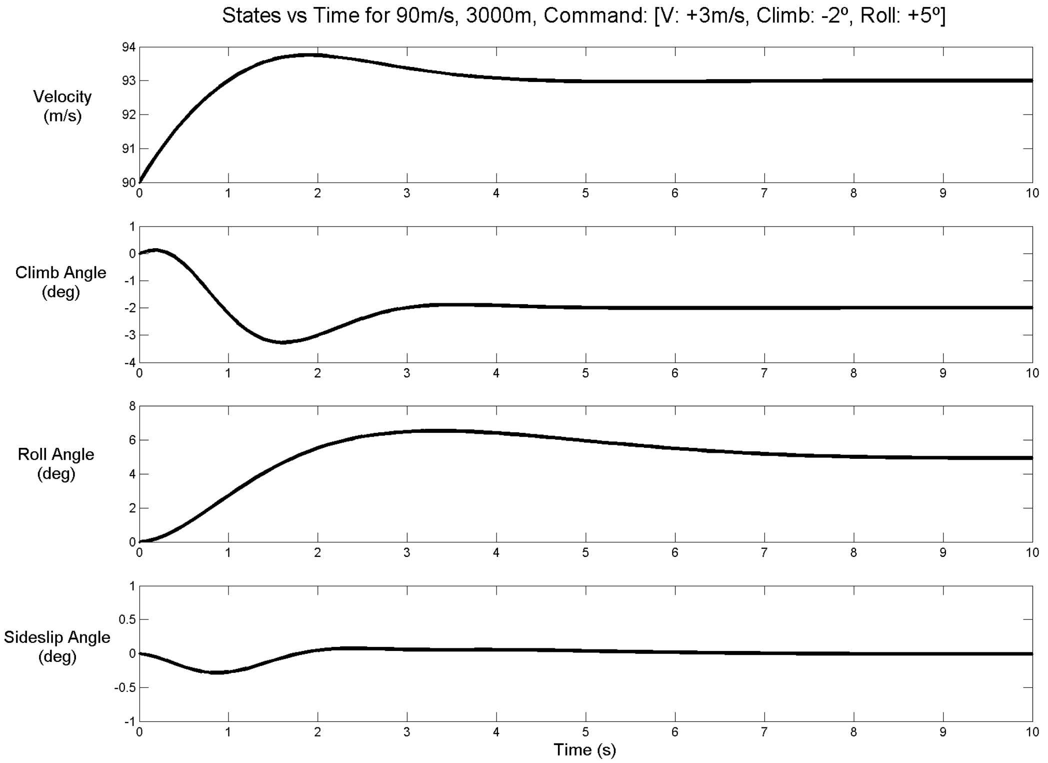

**Figure 5.9**: System output values at the design point with a coupled longitudinal-lateral-directional command (at $[90^{m}/_{s}, 3000m]$ with a command of $[+3^{m}/_{s}, -2°, +5°, +0°]$ ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line. All lines overlap in this graph.

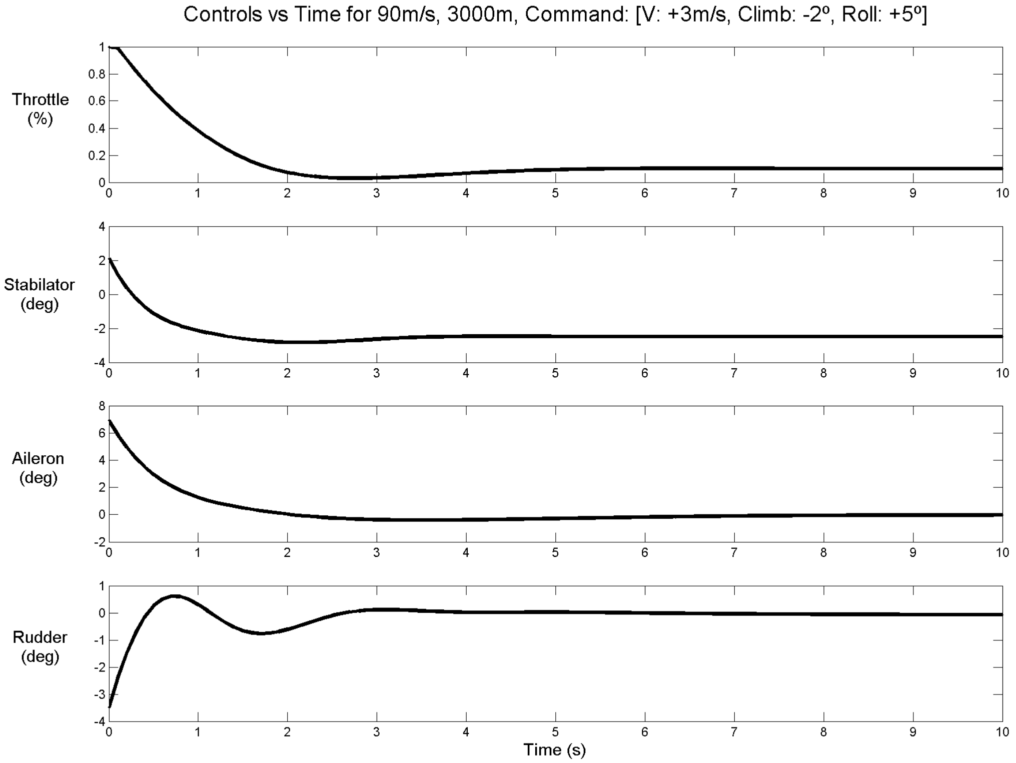

**Figure 5.10**: System control values at the design point with a coupled longitudinal-lateral-directional command (at $[90 ^{m}\!/_{s}, 3000m]$ with a command of $[+3 ^{m}\!/_{s}, -2°, +5°, +0°]$ ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line. All lines overlap in this graph.

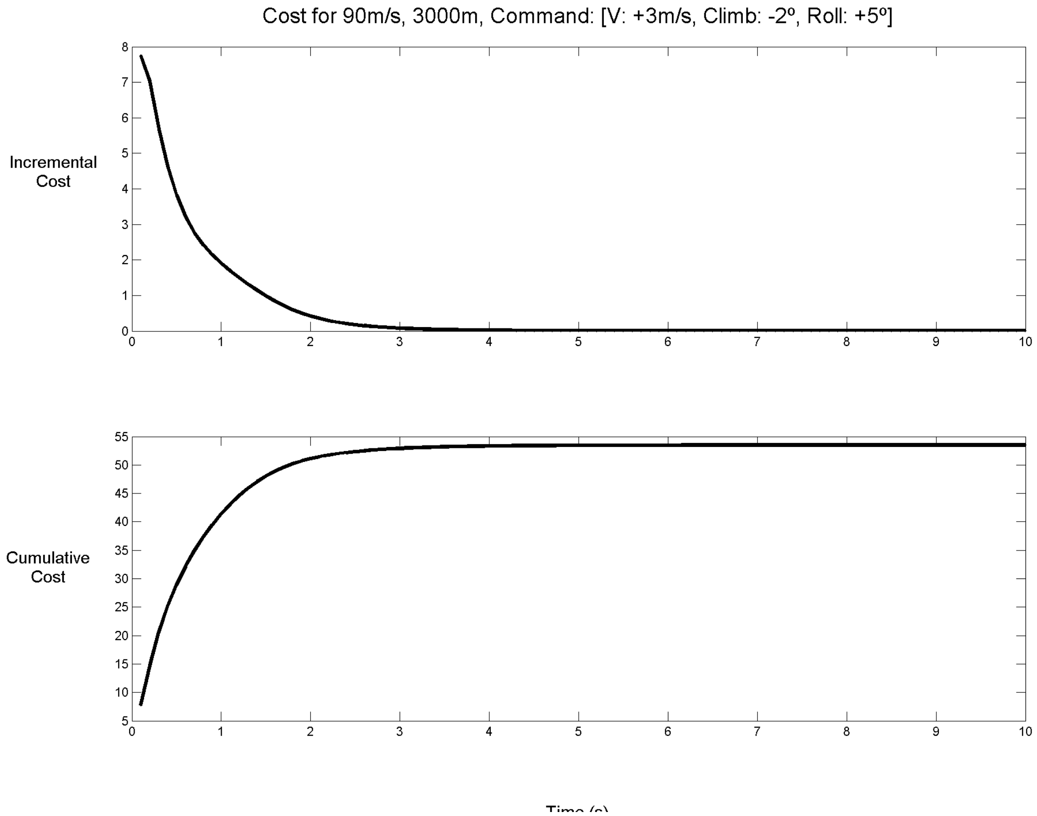

**Figure 5.11**: Incremental and cumulative cost at the design point with a coupled longitudinal-lateral-directional command (at $[90\,{}^{m}\!/_{s},\ 3000m]$ with a command of $[+3\,{}^{m}\!/_{s},\ -2^\circ,\ +5^\circ,\ +0^\circ]$ ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line. All lines overlap in this graph.

## 5.2 Interpolation Point

The system is tested at an interpolation point to observe the performance of the adapting neural controller when compared to the non-adapting neural controller and the linear controller. It is expected that the adapting neural controller will improve performance in cases where the performance of the non-adapting neural controller is suboptimal. For large command inputs, it is expected that the adapting neural controller will perform better than the linear controller due to the nonlinear nature of the plant. At the interpolation point selected, the aircraft is initially flying 145$^m/_s$ at an altitude of 6000$m$.

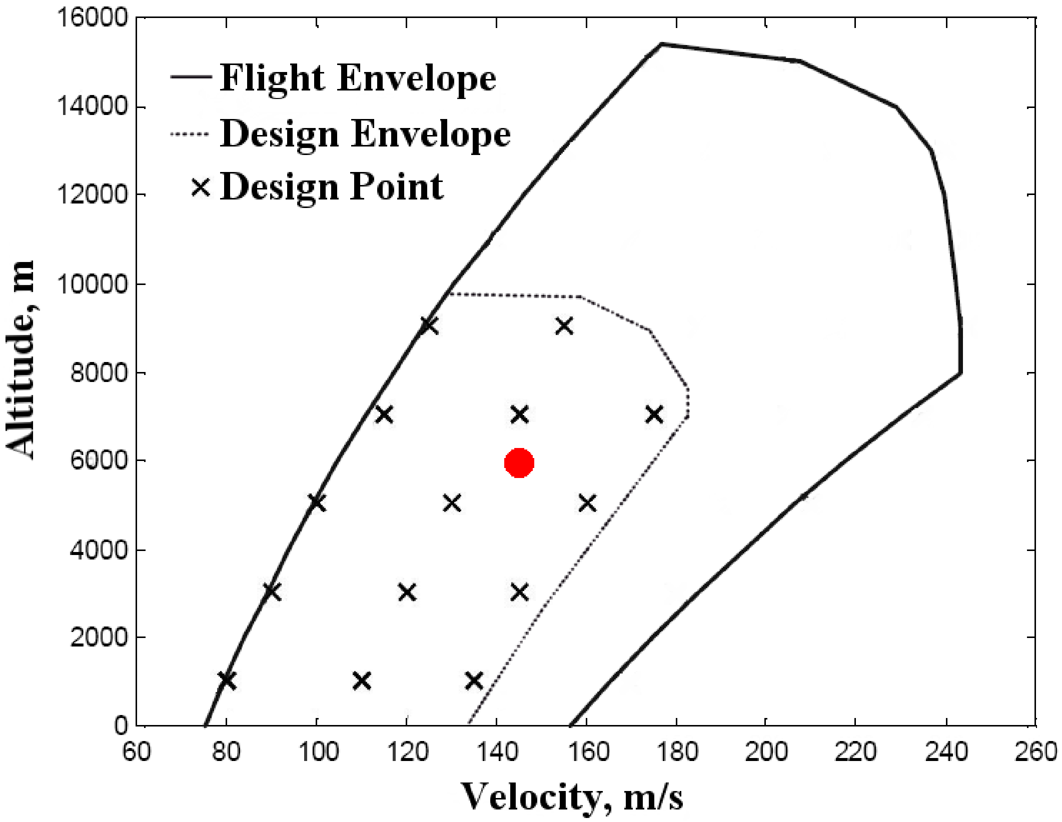

**Figure 5.12**: The design point [145$^m/_s$, 6000$m$] at which the current set of simulations take place.

### 5.2.1 Longitudinal Maneuver

The aircraft response is tested for a small longitudinal command at an interpolation point. At an interpolation point, it is expected that the adapting neural network will attempt to improve its performance through training. At time $t = 0$, the aircraft is given a command to increase speed to $148 {}^{m}\!/\!{}_{s}$ and dive at 5°. Figure 5.13 shows the system output. At the interpolation point, the adaptive neural controller adapts as evidenced by the system response. Since the non-adapting neural controller produces an optimal response, the action network must be optimal at the test point. For the adaptive neural controller to train, the critic network must be suboptimal. The suboptimal critic provides a poor target to the action network. The training of the action network causes the system to have suboptimal performance. As the critic network adapts, the action network is provided with better targets and the system response becomes optimal again. Figure 5.14 shows the control usage during the 10 seconds after the step command is issued. Due to incorrect costate values provided by the critic network, the adaptive control system operates suboptimally during the first second of the simulation. After the first second, the system is quickly able to recover and converge on the desired states. In figure 5.15, a spike in incremental cost can be seen corresponding to the beginning of the simulation. After three seconds, the incremental cost falls to the same level as the non-adapting neural controller.

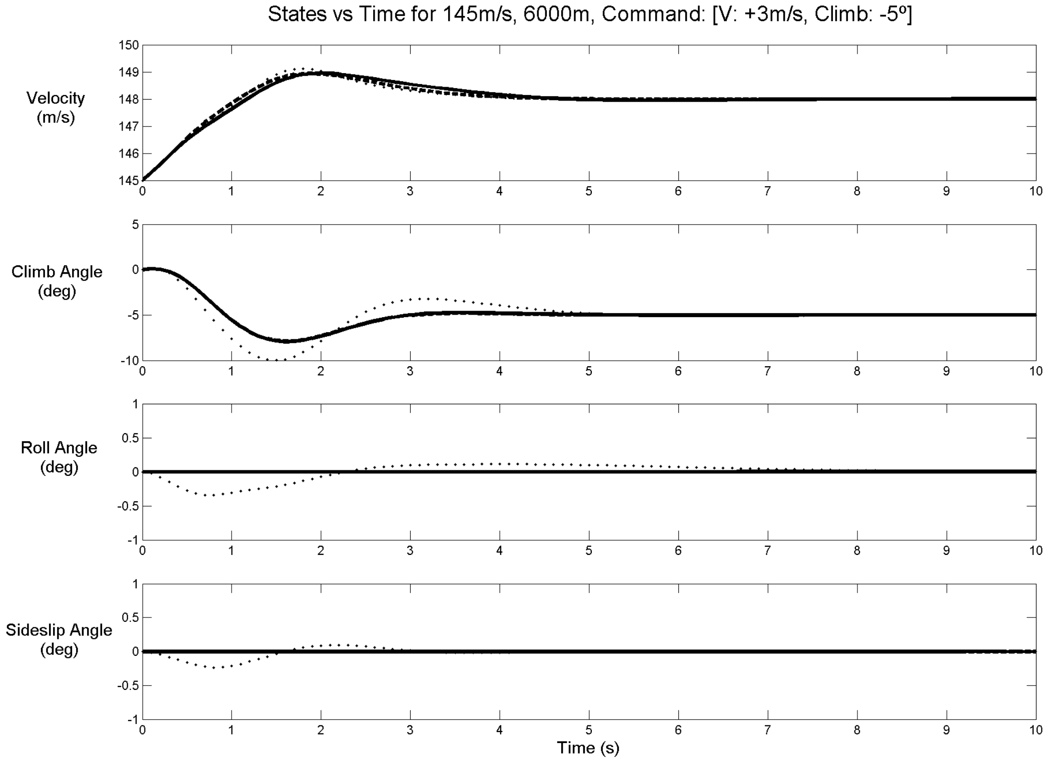

**Figure 5.13**: System output values at the interpolation point with a longitudinal command (at [$145^m/_s$, $6000m$] with a command of [$+3^m/_s$, $-5°$, $+0°$, $+0°$] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

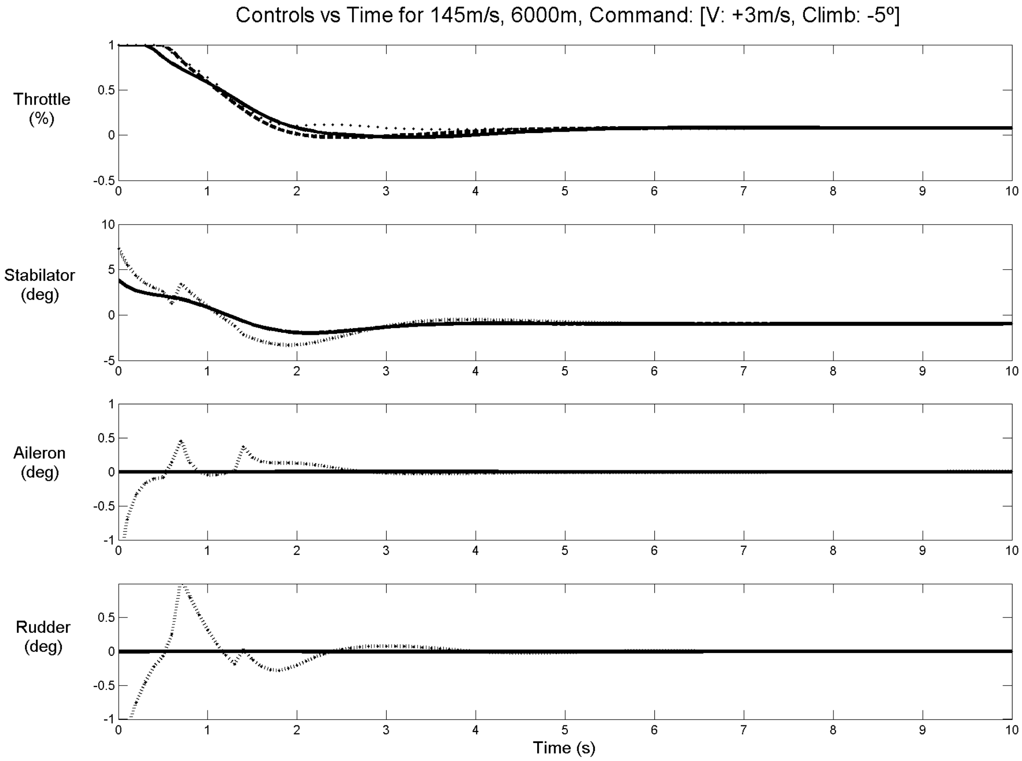

**Figure 5.14**: System control values at the interpolation point with a longitudinal command (at [145m/s, 6000$m$] with a command of [+3m/s, $-5°$, $+0°$, $+0°$] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

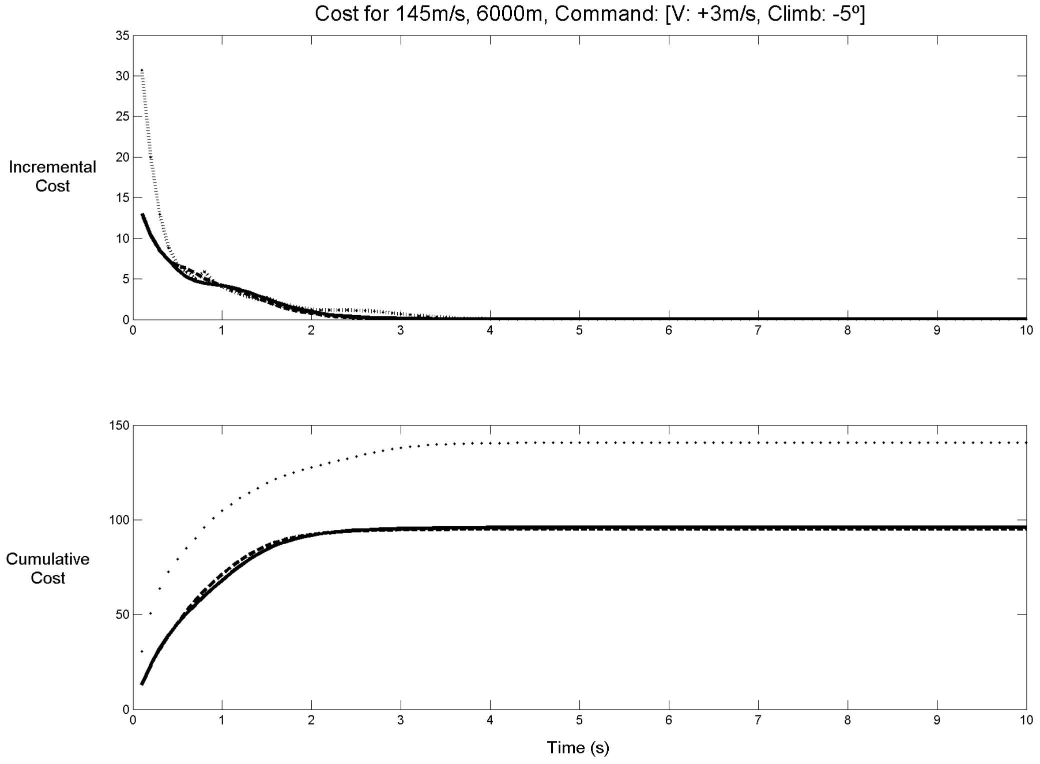

**Figure 5.15**: Incremental and cumulative cost at the interpolation point with a longitudinal command (at [$145^{m}/_{s}$, $6000m$] with a command of [$+3^{m}/_{s}$, $-5°$, $+0°$, $+0°$] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

### 5.2.2 Lateral-Directional Maneuver

The aircraft response is tested when given a lateral-directional command corresponding at an interpolation point. At time $t = 0$, the aircraft is given a command to bank at 60°. At such a large command, the nonlinear and coupled effects are significant. Figure 5.16 shows the system response. The system with the adaptive neural controller has a faster settling time than the non-adapting neural controller and the linear controller. Figure 5.17 shows the control usage during the 15 seconds after the step command is issued. The system with the adaptive neural controller has a much lower control usage than the systems with the non-adapting neural controller and the linear controller. The cost graphs in figure 5.18 are the most revealing about the flight simulation. The adapting neural controller performs far better than both the non-adapting neural controller and the linear controller.

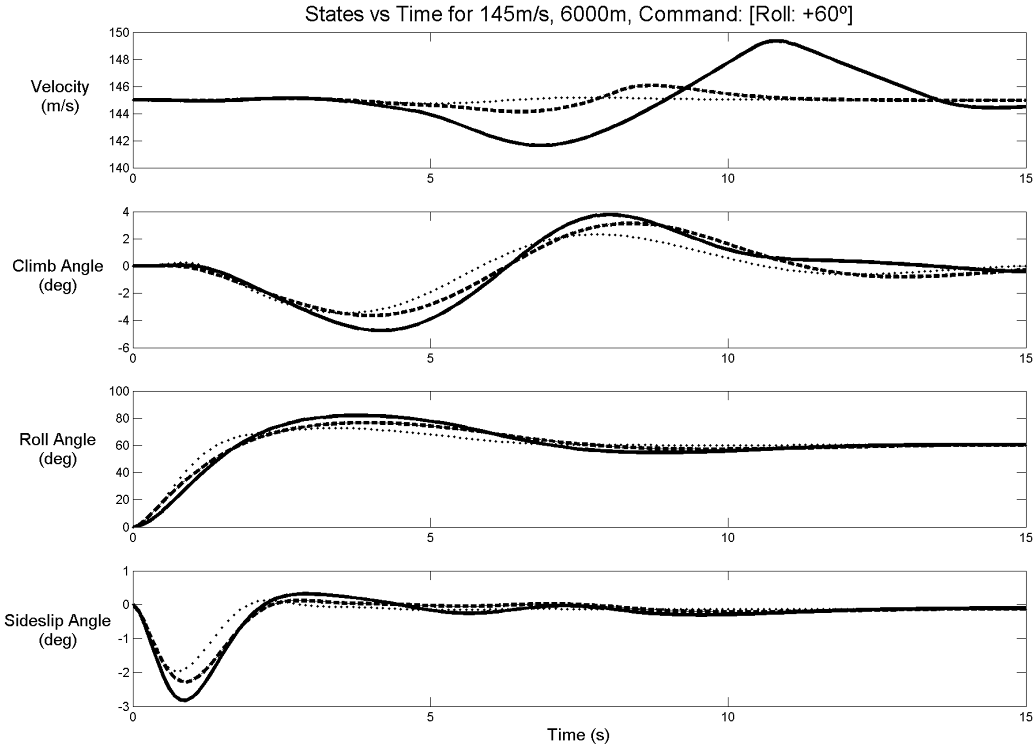

**Figure 5.16**: System output values at the interpolation point with a lateral-directional command (at [145$^m/_s$, 6000$m$] with a command of [+0$^m/_s$, +0°, +60°, +0°] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

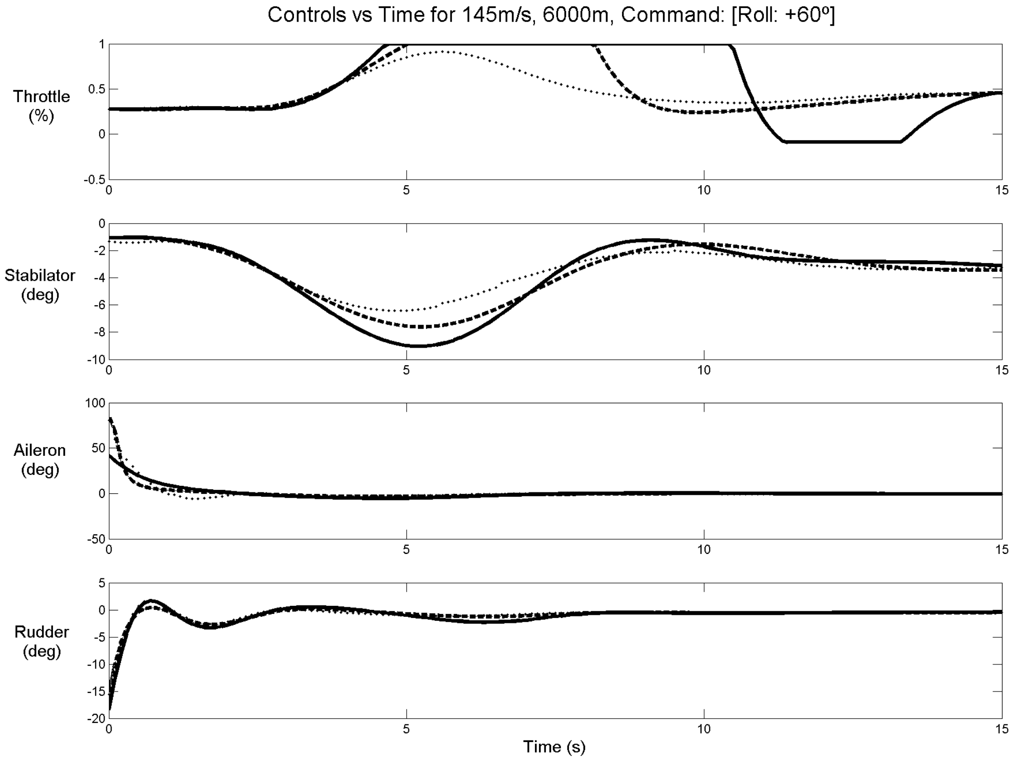

**Figure 5.17**: System control values at the interpolation point with a lateral-directional command (at [145$^m/_s$, 6000$m$] with a command of [+0$^m/_s$, +0°, +60°, +0°] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

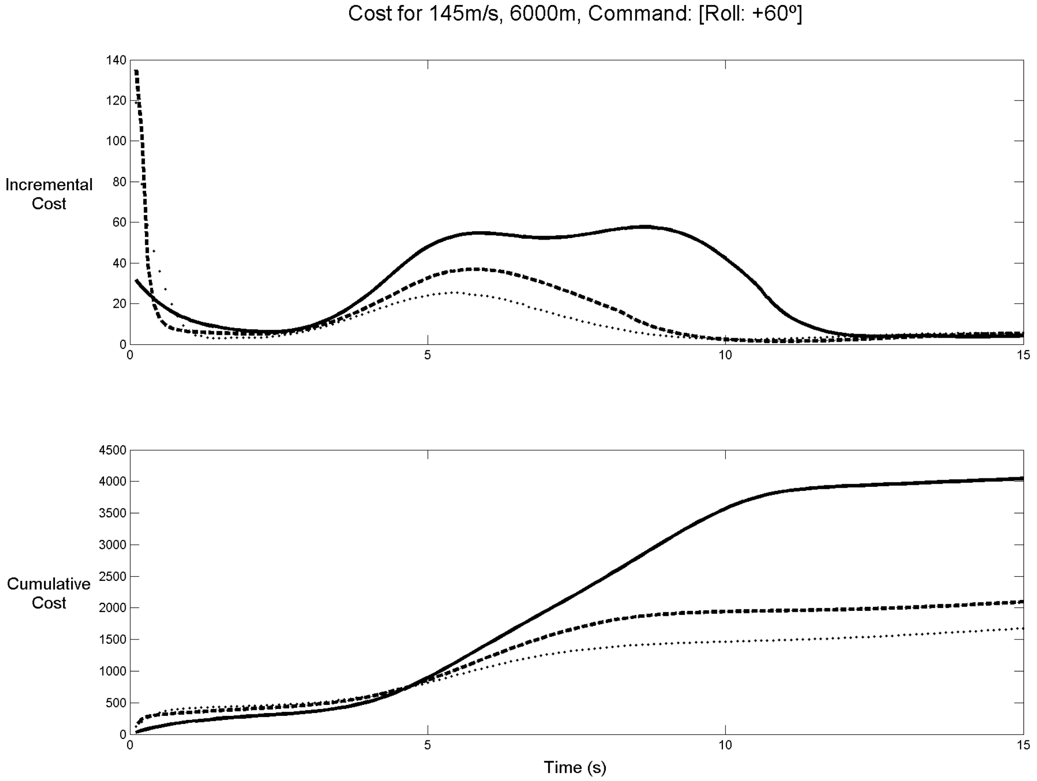

**Figure 5.18**: Incremental and cumulative cost at the interpolation point with a lateral-directional command (at [145$^m/_s$, 6000$m$] with a command of [+0$^m/_s$, +0°, +60°, +0°] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

### 5.2.3 Coupled Longitudinal-Lateral-Directional Maneuver

The aircraft response is test when given a coupled command at an interpolation point. At time $t = 0$, the aircraft is given a command to increase speed to $150 {}^{m}\!/\!{}_{s}$ and bank at 45°. Figure 5.19 shows the system response. At the interpolation point, the adaptive neural controller undergoes training as evidenced by the system output. The system with the non-adapting neural controller has a near optimal response. Figure 5.20 shows the control usage during the 15 seconds after the step command is issued. Figure 5.21 shows the incremental and cumulative costs associated with the flight. As with the longitudinal case, a suboptimal critic network initially worsens system performance. Based on incorrect costate values, the system chooses to overuse the rudder and to use the stabilator in a non-optimal manner. As the critic network improves its approximation of the costate function, the action network returns to near optimal performance. The cost increase over the comes mostly from correcting the stabilator and rudder mistakes. The adaptive neural controller ends the simulation with almost the same cost as the non-adapting neural controller.

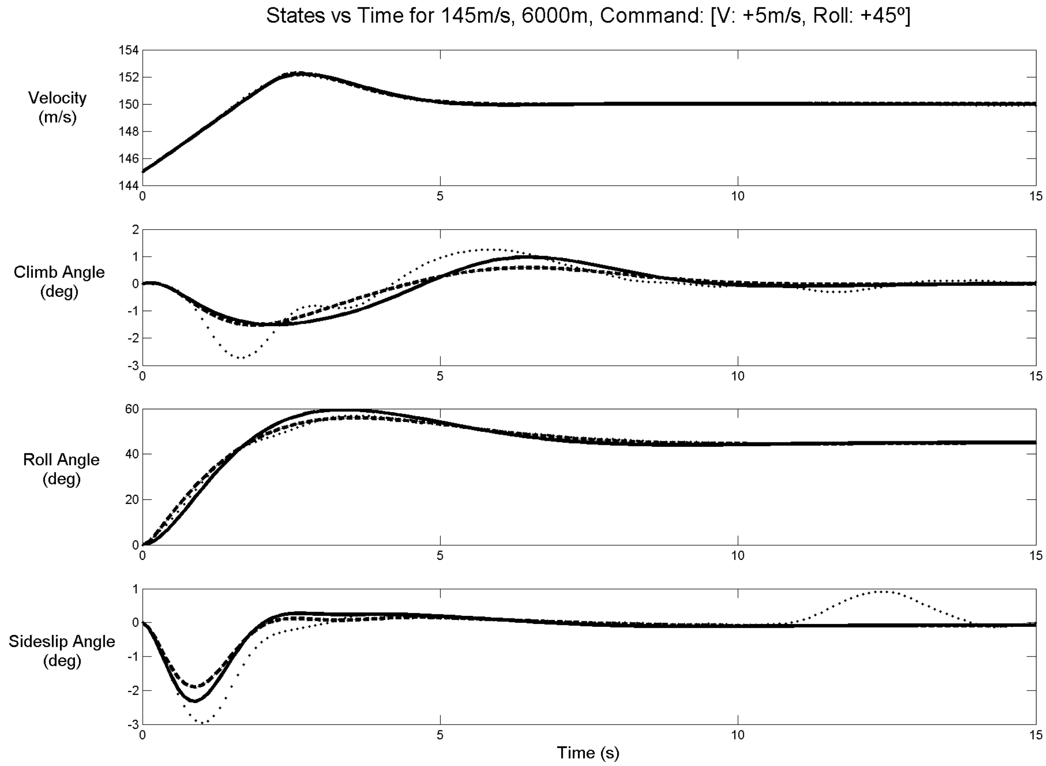

**Figure 5.19**: System output values at the interpolation point with a coupled longitudinal-lateral-directional command (at [$145^m/_s$, $6000m$] with a command of [$+5^m/_s$, +0°, +45°, +0°] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

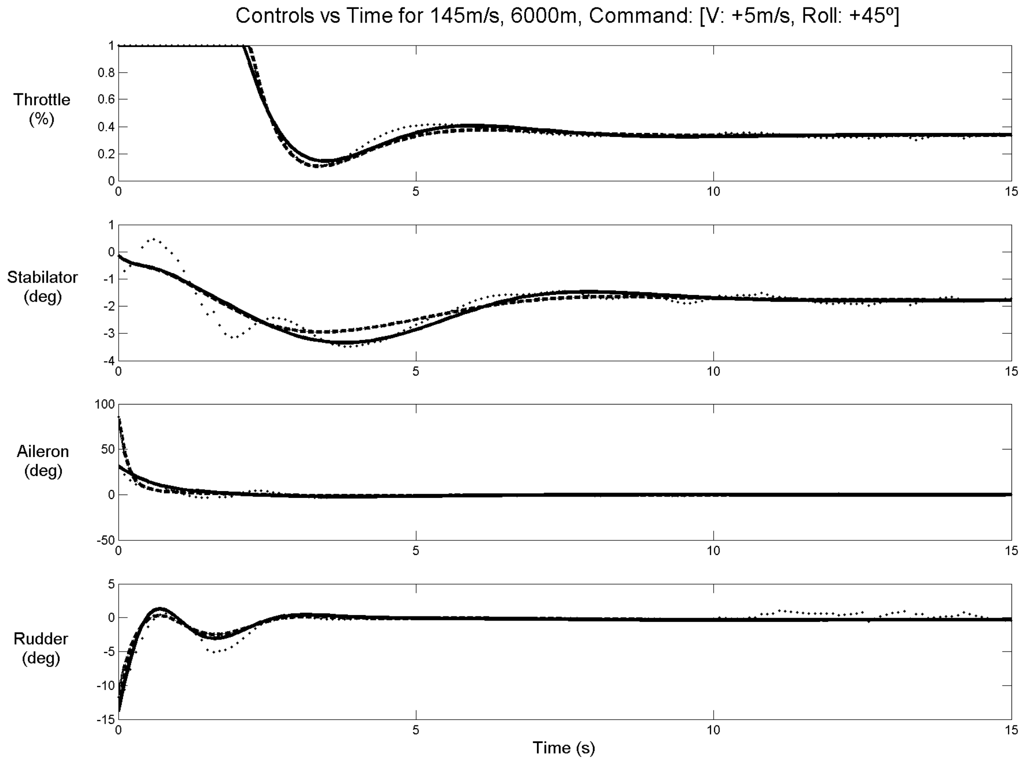

**Figure 5.20**: System control values at the interpolation point with a coupled longitudinal-lateral-directional command (at [$145^m/_s$, $6000m$] with a command of [$+5^m/_s$, $+0°$, $+45°$, $+0°$] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

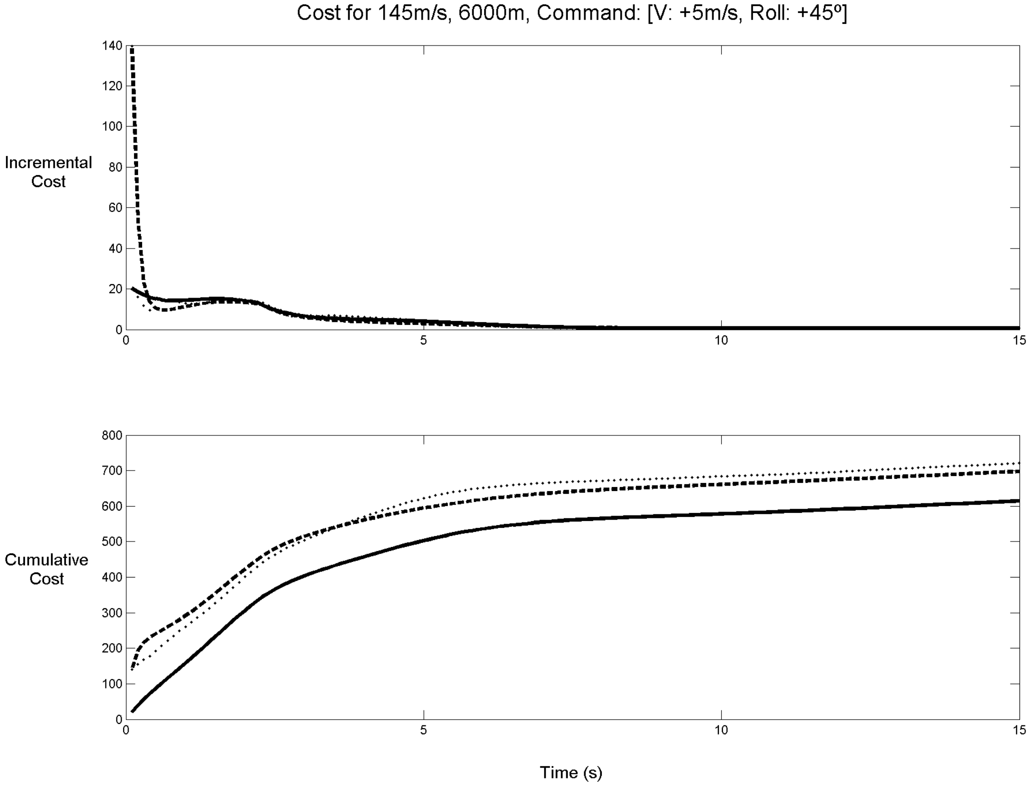

**Figure 5.21**: Incremental and cumulative cost at the interpolation point with a coupled longitudinal-lateral-directional command (at [145$^m/_s$, 6000$m$] with a command of [+5$^m/_s$, +0°, +45°, +0°] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

## 5.3 Extrapolation Point

The system is tested at an extrapolation point to observe the performance of the adapting neural controller when the system encounters unmodeled dynamics. The performance of the adapting neural controller is compared to the non-adapting neural controller and to the linear controller. Because the test point selected is outside of the design envelope, the non-adapting neural controller is expected to perform sub-optimally. It is expected that the adapting neural controller will improve performance in all cases. For large command inputs, it is expected that the adapting neural controller will perform better than the linear controller due to the nonlinear nature of the plant. At the extrapolation point selected, the aircraft is initially flying $160^{m}/_{s}$ at an altitude of $12000m$.

### 5.3.1 Longitudinal Maneuver

The aircraft response is tested when given a longitudinal command at an extrapolation point. At time $t = 0$, the aircraft is given a command to decrease speed to $159^{m}/_{s}$ and to climb at $6°$. Figure 5.23 shows the system response. The adaptive neural controller pursues a slightly different path than the non-adapting neural controller and the linear controller. Figure 5.24 shows the control usage during the 10 seconds after the step command is issued. The controls chosen by the adaptive neural controller are almost identical to those chosen by the non-adapting neural controller until the very end of the simulation. At the end of the simulation, the adapting neural controller reduces thrust sooner than the non-adapting neural controller. The cost accrued by each controller is shown in figure 5.25. The adapting neural controller performs better for the simulation than both the non-adapting neural controller and the linear controller.

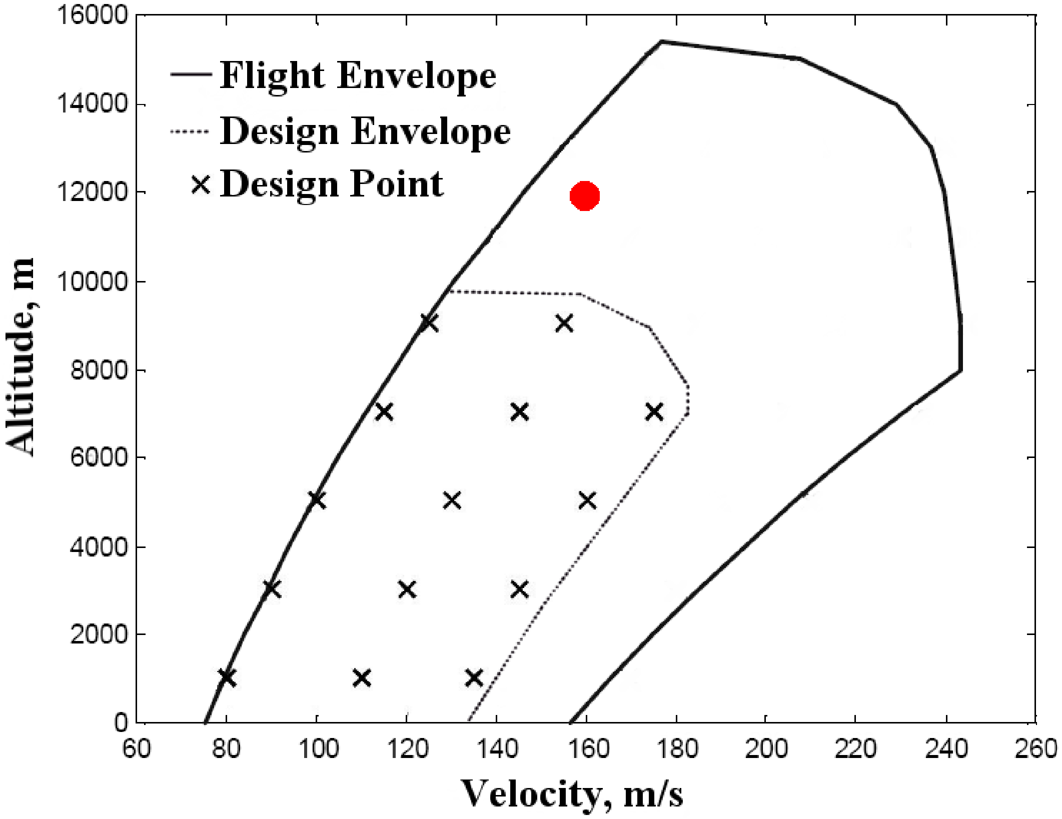

**Figure 5.22**: The design point [$160 ^{m}/_{s}$, $12000m$] at which the current set of simulations take place.

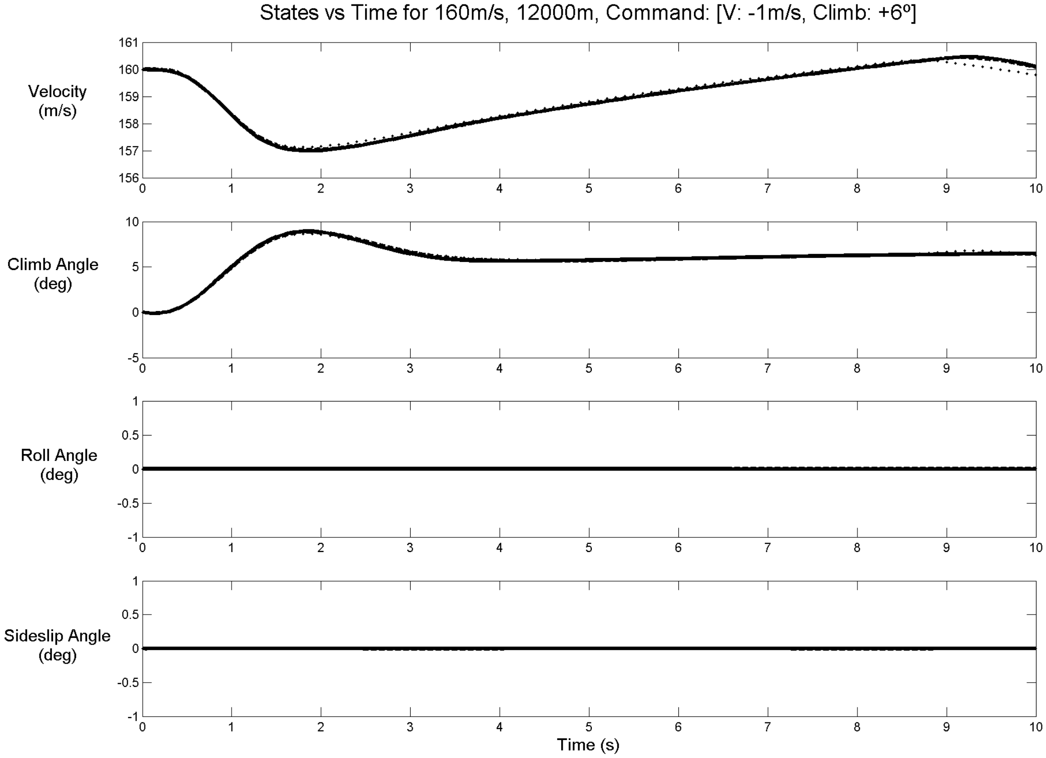

**Figure 5.23**: System output values at the extrapolation point with a longitundinal command (at [160$^m/_s$, 12000$m$] with a command of [$-1^m/_s$, +6°, +0°, +0°] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

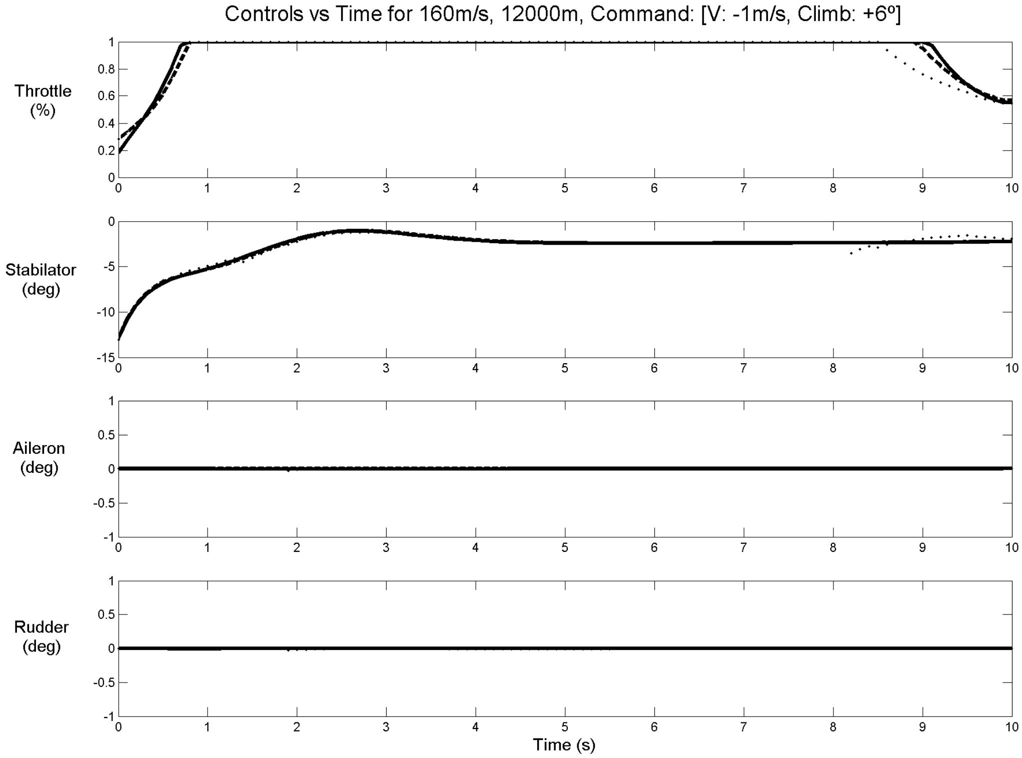

**Figure 5.24**: System control values at the extrapolation point with a longitundinal command (at [$160 m/s$, $12000m$] with a command of [$-1 m/s$, $+6°$, $+0°$, $+0°$] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

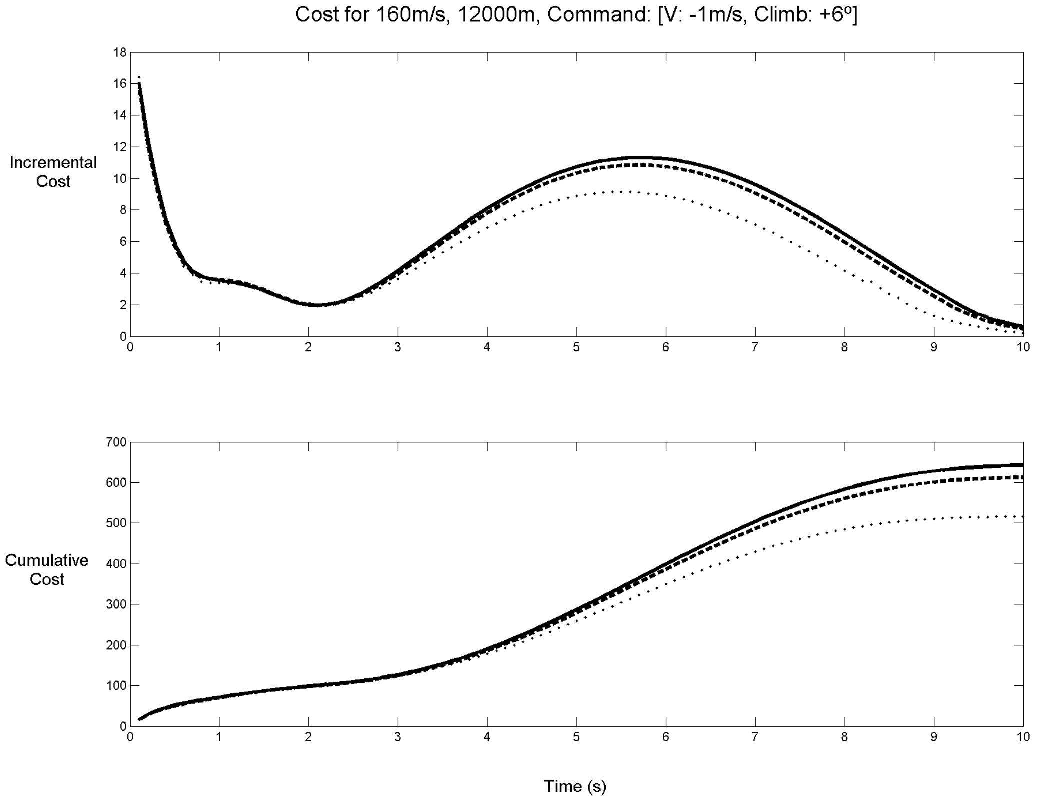

**Figure 5.25**: Incremental and cumulative cost at the extrapolation point with a longitudinal command (at [160$^m/_s$, 12000$m$] with a command of [$-1^m/_s$, $+6°$, $+0°$, $+0°$] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

### 5.3.2 Lateral-Directional Maneuver

The aircraft response is tested when given a lateral-directional command at an extrapolation point. At time $t = 0$, the aircraft is given a command to bank at 60°. At such a large command, the nonlinear and coupled effects are significant. Figure 5.26 shows the system response. The adaptive neural controller responds much faster than the non-adapting neural controller. Figure 5.27 shows the control usage during the 20 seconds after the step command is issued. The controls chosen by the adaptive neural controller differ significantly from those chosen by the non-adapting neural controller. The adaptive neural controller uses the stabilator and the throttle to maintain speed during the banking maneuver. Additionally, the ailerons are used more aggressively by the adaptive neural controller than by the non-adapting neural controller. The cost graphs in figure 5.28 reveal that the adapting neural controller performs much better than the non-adapting neural controller and the linear controller.

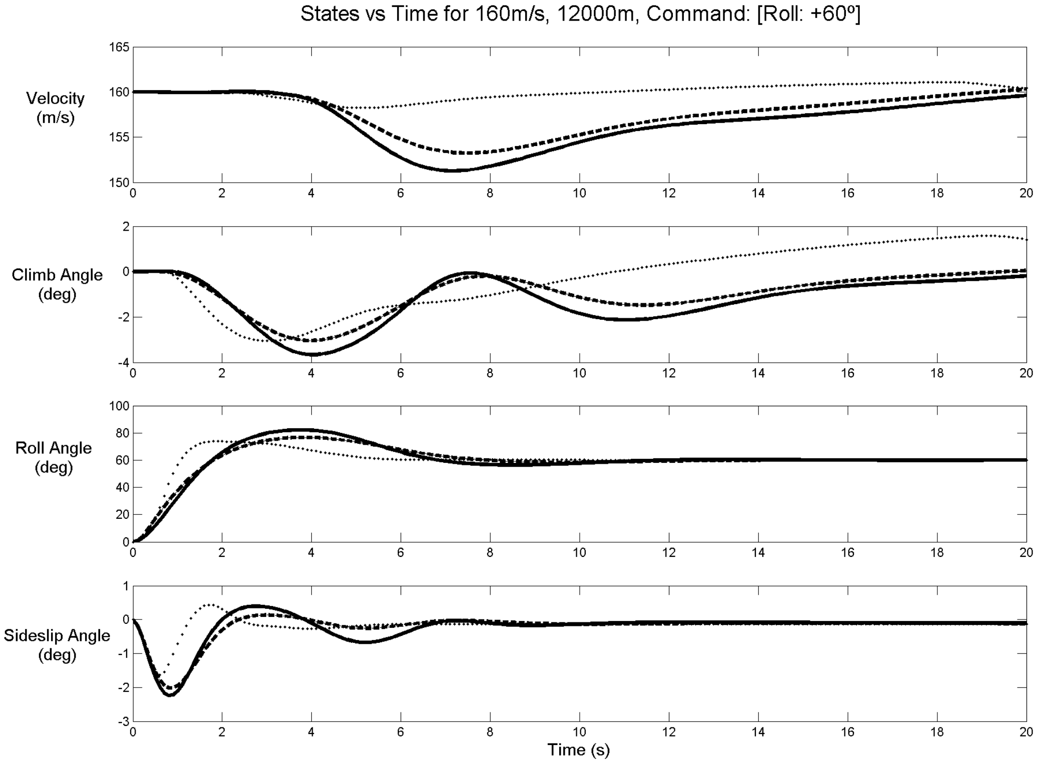

**Figure 5.26**: System output values at the extrapolation point with a lateral-directional command (at [160$^m/_s$, 12000$m$] with a command of [+0$^m/_s$, +0°, +60°, +0°] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

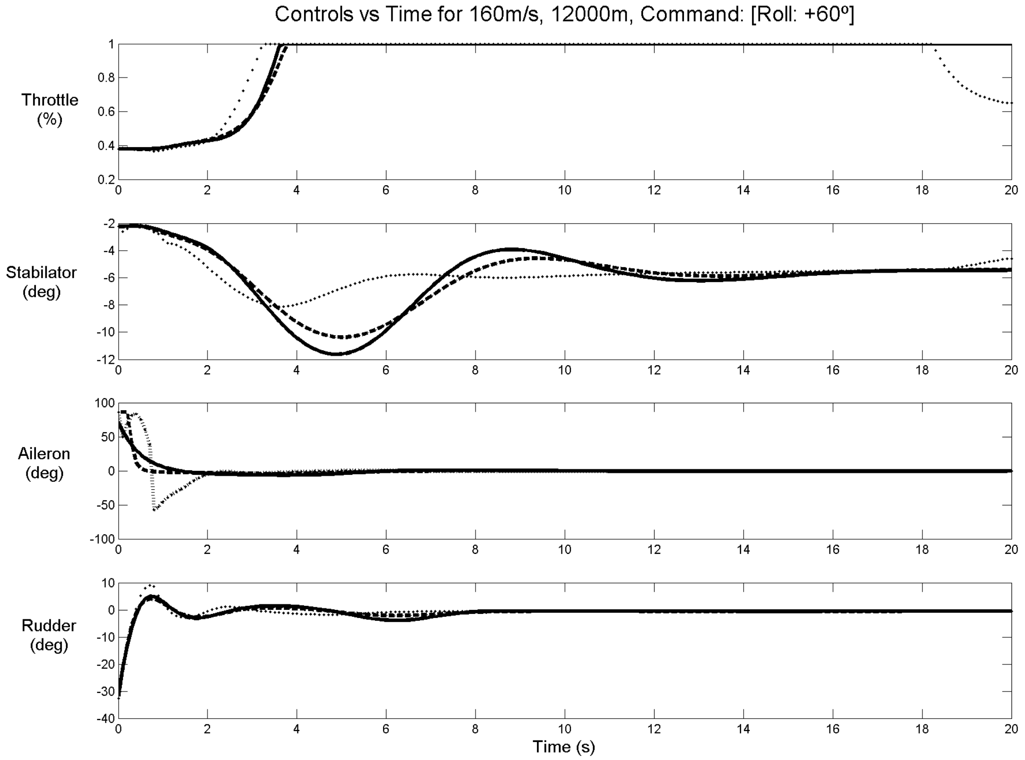

**Figure 5.27**: System control values at the extrapolation point with a lateral-directional command (at [160$^{m}/_{s}$, 12000$m$] with a command of [+0$^{m}/_{s}$, +0°, +60°, +0°] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

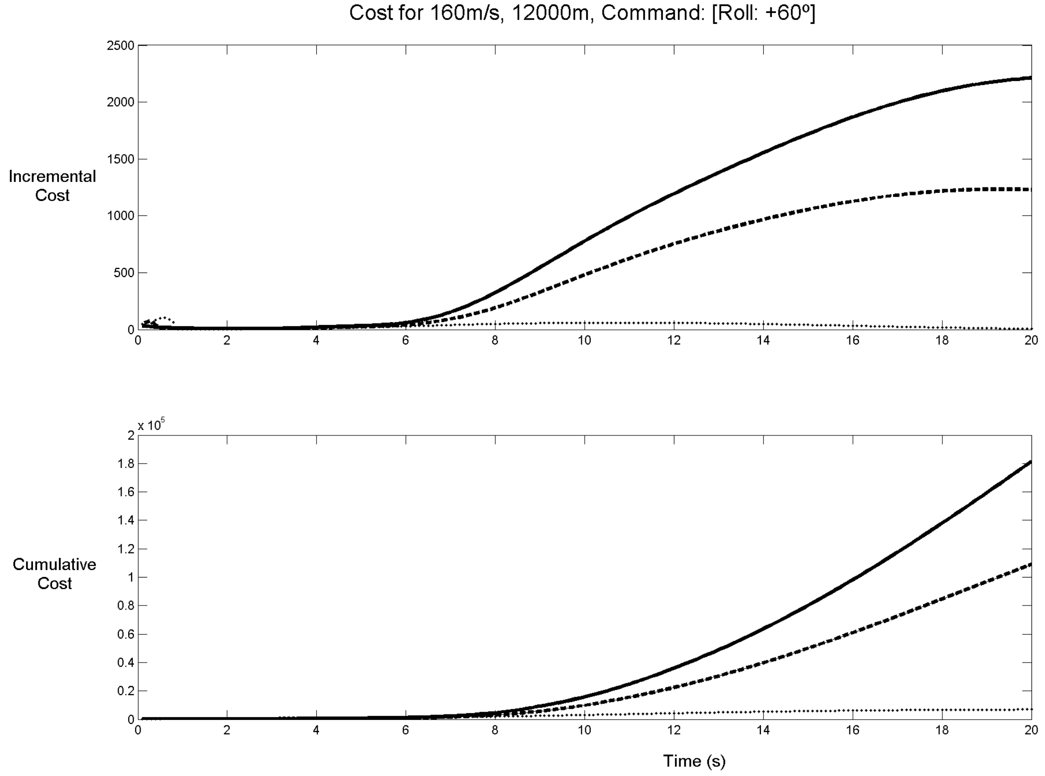

**Figure 5.28**: Incremental and cumulative cost at the extrapolation point with a lateral-directional command (at [160$^m/_s$, 12000$m$] with a command of [+0$^m/_s$, +0°, +60°, +0°] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

### 5.3.3 Coupled Longitudinal-Lateral-Directional Maneuver

The aircraft response is test when given a coupled command at an extrapolation point. At time $t = 0$, the aircraft is given a command to decrease speed to $155 ^{m}\!/_{s}$ and to bank at 45°. At such a large command, the nonlinear and coupled effects are significant. Figure 5.29 shows the system response. The system with the adaptive neural controller reaches its target values faster than the other controllers. Figure 5.30 shows the control usage during the 10 seconds after the step command is issued. The adaptive neural controller uses the ailerons excessively during the first two seconds of the simulation. The non-optimal usage of controls in the presence of a near optimal non-adapting controller indicates a suboptimal critic network. As the critic network improves its approximation of the costate function, the adaptive neural controller returns to optimality. The incremental and total cost graphs in figure 5.31 reveal that after the first two seconds of simulation, the adaptive neural controller performs better than the non-adaptive controller and the linear controller.

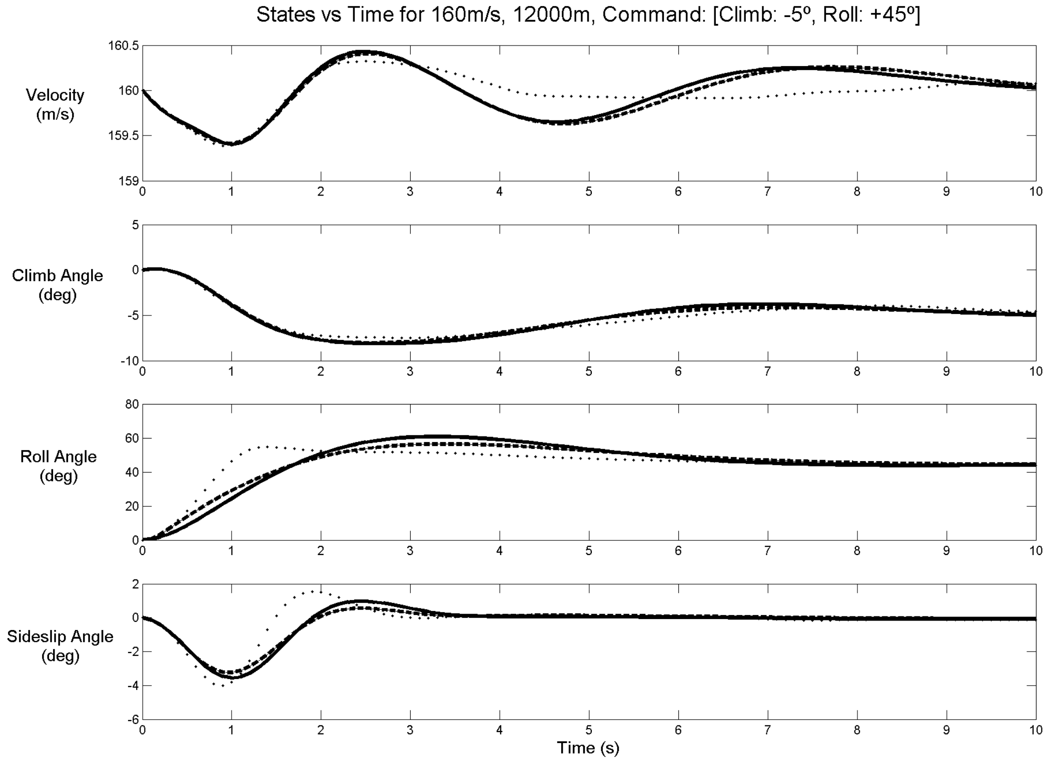

**Figure 5.29**: System output values at the extrapolation point with a coupled longitudinal-lateral-directional command (at [160$^{m}/_{s}$, 12000$m$] with a command of [$-5^{m}/_{s}$, $+0°$, $+45°$, $+0°$] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

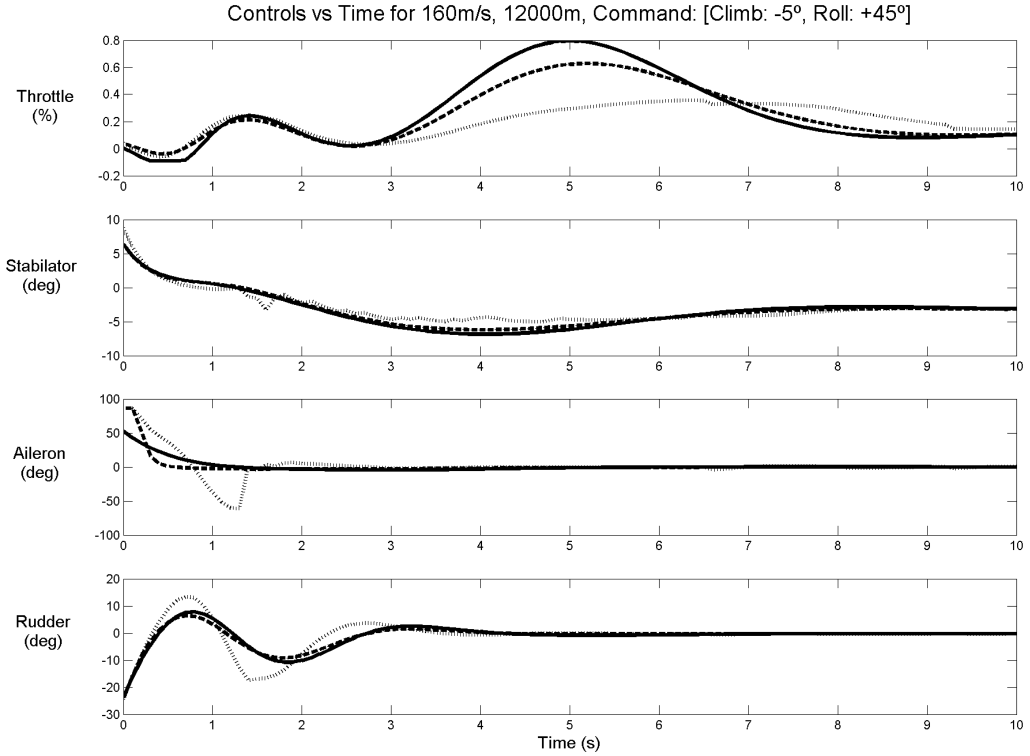

**Figure 5.30**: System control values at the extrapolation point with a coupled longitudinal-lateral-directional command (at $[160^m/_s,\ 12000m]$ with a command of $[-5^m/_s,\ +0°,\ +45°,\ +0°]$ ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

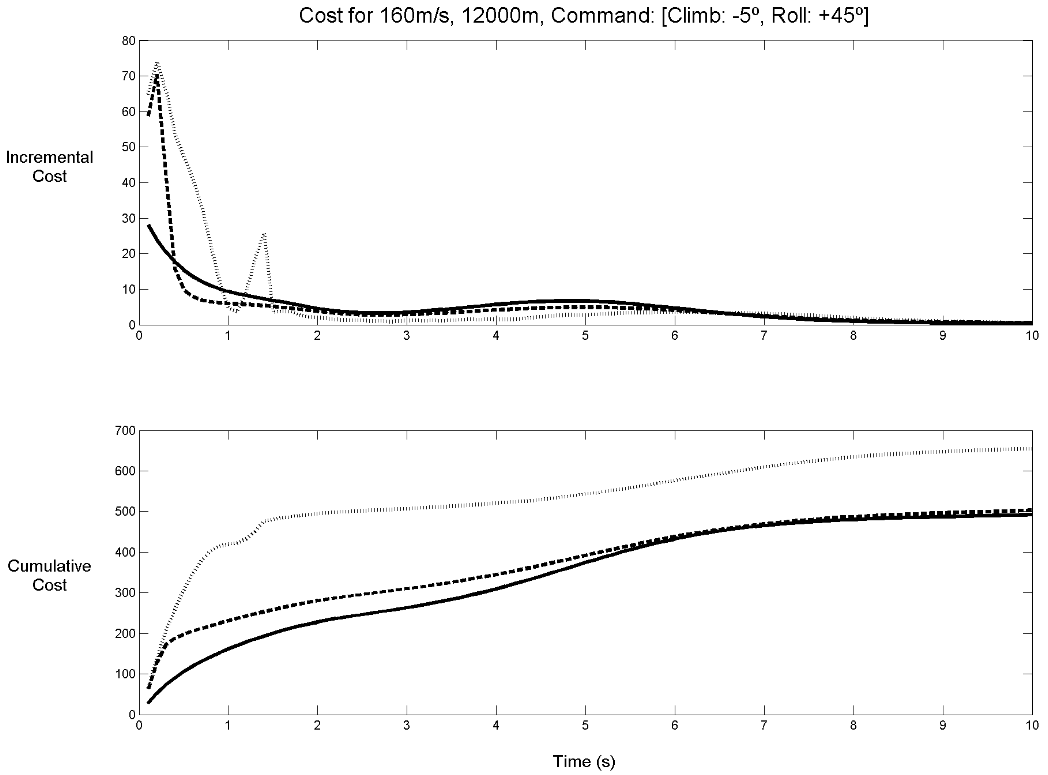

**Figure 5.31**: Incremental and cumulative cost at the extrapolation point with a coupled longitudinal-lateral-directional command (at [$160^m/_s$, $12000m$] with a command of [$-5^m/_s$, $+0°$, $+45°$, $+0°$] ). The ideal controller is represented by the solid line. The non-adapting neural controller is represented by the dashed line. The adaptive neural controller is represented by the dotted line.

## 5.4 Constrained and Unconstrained Training

The performance of a constrained adapting neural controller is compared to that of an unconstrained neural controller at an interpolation point with a large lateral-directional command. The neural controllers are then held fixed while the aircraft is flown at a design point and given a small coupled command to demonstrate the constrained neural controller's ability to preserve design point optimality.

### 5.4.1 Interpolation Point

The aircraft response is tested at a interpolation point when given a lateral-directional command. Initially, the aircraft is flying $145^{m}/_{s}$ at an altitude of $6000m$. At time $t = 0$, the aircraft is given a command to bank at $60°$. At such a large command, the nonlinear and coupled effects are significant. Figure 5.32 shows the system response. Figure 5.33 shows the control usage during the 10 seconds after the step command is issued. Figure 5.34 shows the accrued cost of the controllers. The constrained adaptive neural controller clearly performs better than the unconstrained controller and the linear controller. The outputs and gradients of each neural controller's action neural network are compared at the design points with the outputs and gradients of the linear controller. Table 5.4.1 shows the mean squared error of each controller's output and gradient values during the simulation. The constrained training algorithm succeeds in preserving these values while the unconstrained algorithm does not preserve the values.

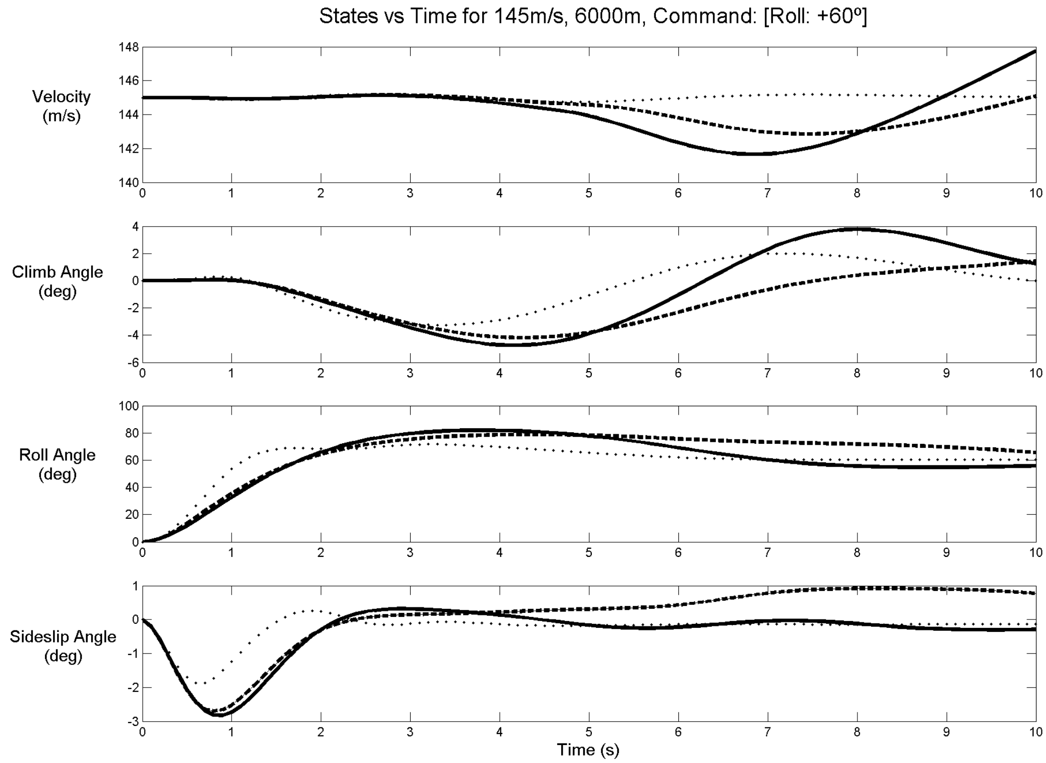

**Figure 5.32**: System output values at the interpolation point with a lateral-directional command (at [145$^m\!/_s$, 6000$m$] with a command of [+0$^m\!/_s$, +0°, +60°, +0°] ). The ideal controller is represented by the solid line. The unconstrained adapting neural controller is represented by the dashed line. The constrained adaptive neural controller is represented by the dotted line.

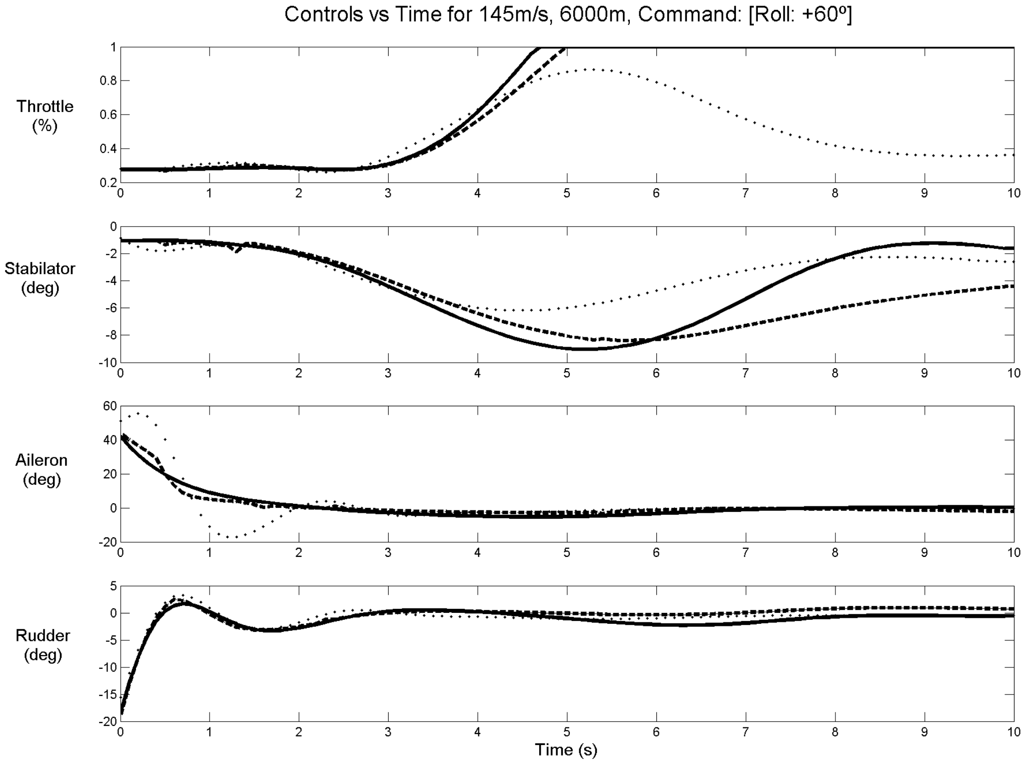

**Figure 5.33**: System control values at the interpolation point with a lateral-directional command (at [145$^{m}/_{s}$, 6000$m$] with a command of [+0$^{m}/_{s}$, +0°, +60°, +0°] ). The ideal controller is represented by the solid line. The unconstrained adapting neural controller is represented by the dashed line. The constrained adaptive neural controller is represented by the dotted line.

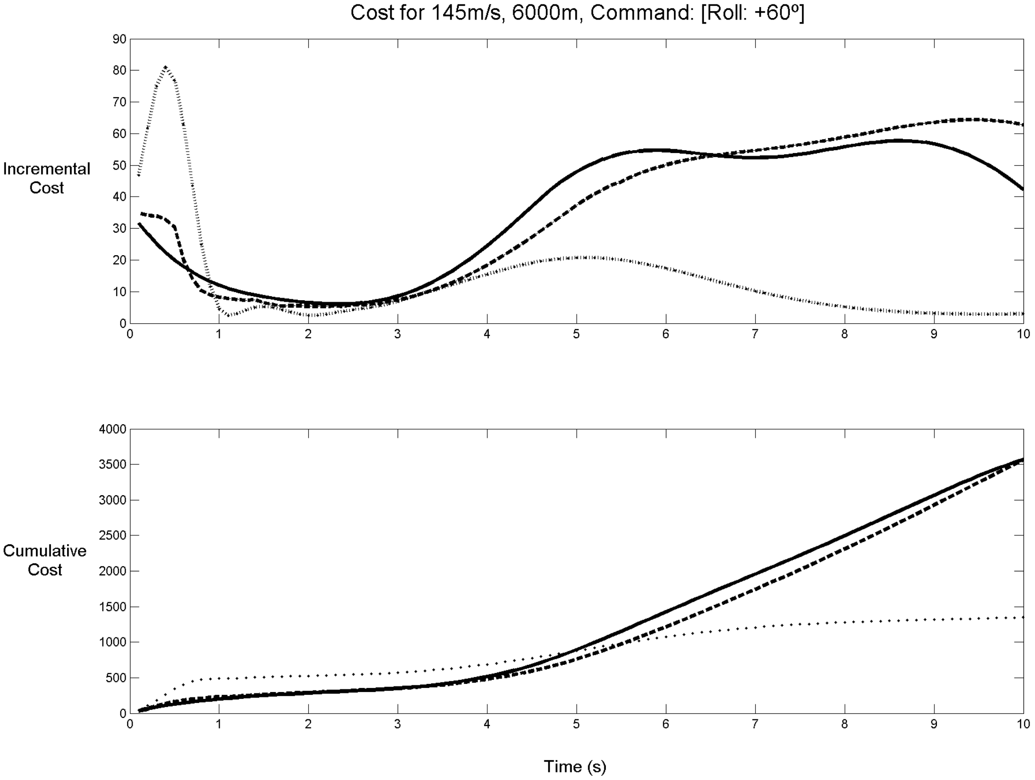

**Figure 5.34**: Incremental and cumulative cost at the interpolation point with a lateral-directional command (at $[145^m/_s, 6000m]$ with a command of $[+0^m/_s,\ +0°,\ +60°,\ +0°]$ ). The ideal controller is represented by the solid line. The unconstrained adapting neural controller is represented by the dashed line. The constrained adaptive neural controller is represented by the dotted line.

| Action Neural Network | $T = 0$ | $T = 5$ | $T = 10$ |
|---|---|---|---|
| Constrained Output MSE | $2.729x10^{-7}$ | $2.404x10^{-7}$ | $5.555x10^{-7}$ |
| Unconstrained Output MSE | $2.729x10^{-7}$ | $1.770x10^{12}$ | $3.168x10^{11}$ |
| Constrained Gradient MSE | $8.470x10^{-28}$ | $7.545x10^{-26}$ | $4.057x10^{-27}$ |
| Unconstrained Gradient MSE | $8.470x10^{-28}$ | $7.868x10^{-4}$ | $1.373x10^{-4}$ |

**Table 5.1**: The mean squared error between the constrained action neural network values and the equivalent linear gain matrix values during a simulation.

### 5.4.2 Design Point

After adapting at the interpolation point mentioned above, the constrained and unconstrained adapting neural controllers are held fixed and tested at a design point. The response of the aircraft with a linear optimal controller is also included for comparison. The performances of the systems with the linear controller and the constrained adaptive neural controller are identical while the the system with the unconstrained adaptive neural controller has a suboptimal response. Training at the interpolation point involved a large roll command and, as seen in figure 5.35, the unconstrained adaptive neural controller overcompensates when faced with a smaller command.

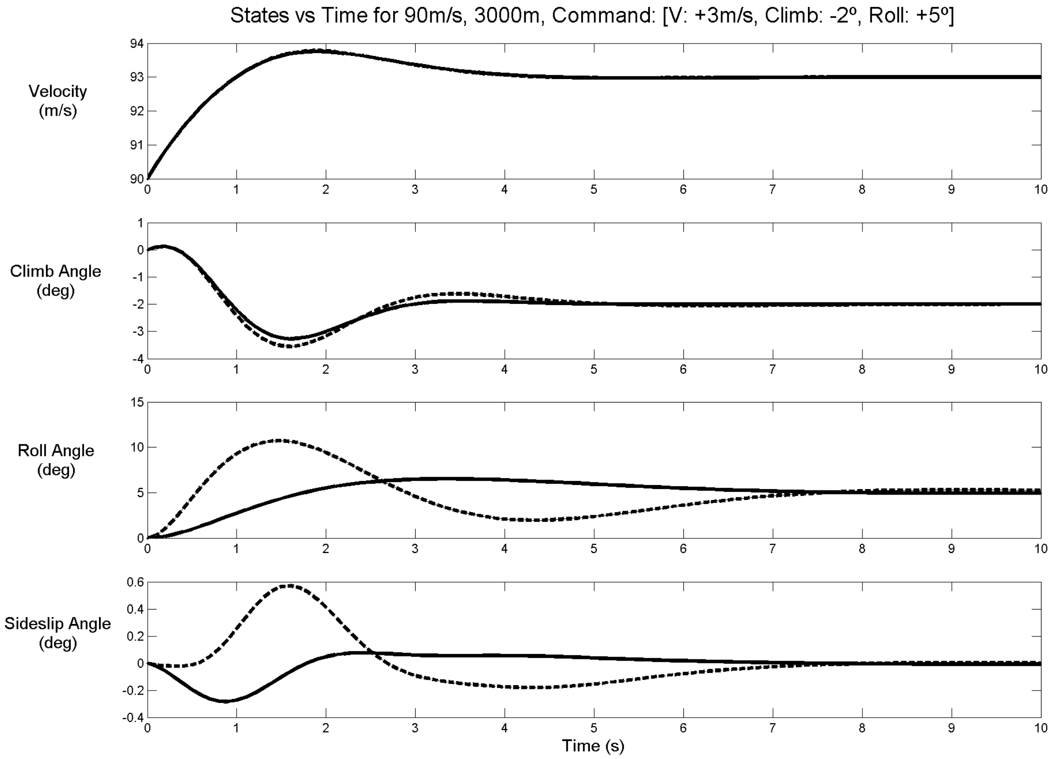

**Figure 5.35**: System output values at the design point with a coupled longitudinal-lateral-directional command (at [90$^m/_s$, 3000$m$] with a command of [+3$^m/_s$, −2°, +5°, +0°] ). The ideal controller is represented by the solid line. The unconstrained adapting neural controller is represented by the dashed line. The constrained adaptive neural controller is represented by the dotted line. During this simulation, the neural controllers adaptation functions are turned off. The linear and constraining adapting neural controller have identical performance.

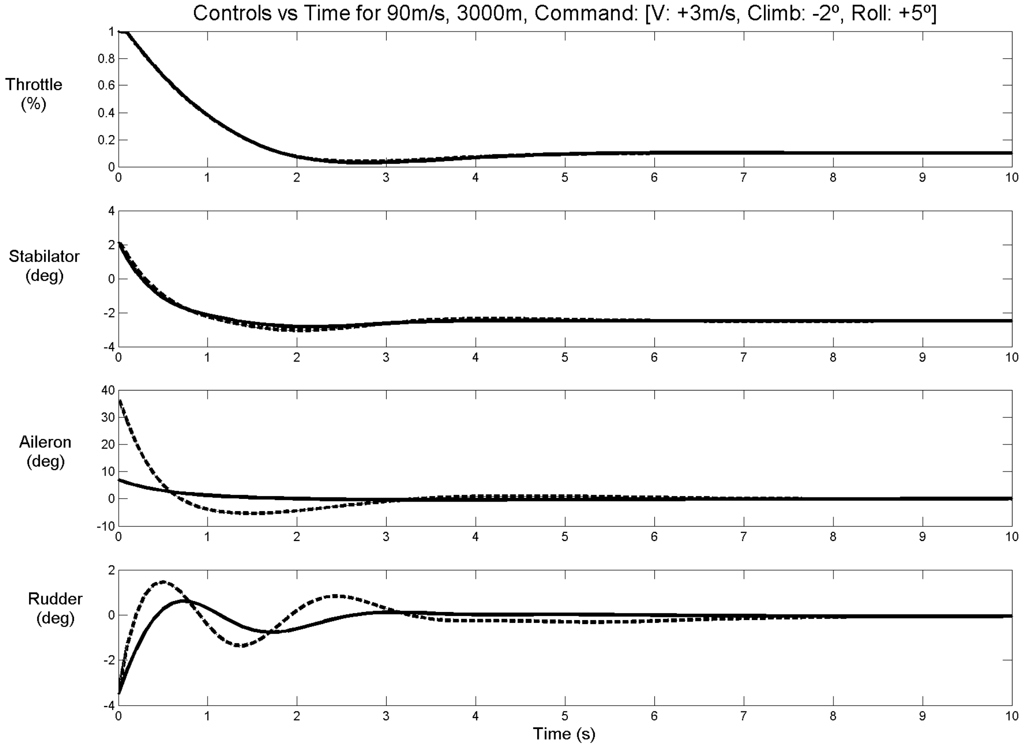

**Figure 5.36**: System control values at the design point with a coupled longitudinal-lateral-directional command (at [$90^m/_s$, $3000m$] with a command of [$+3^m/_s$, $-2°$, $+5°$, $+0°$] ). The ideal controller is represented by the solid line. The unconstrained adapting neural controller is represented by the dashed line. The constrained adaptive neural controller is represented by the dotted line. During this simulation, the neural controllers adaptation functions are turned off. The linear and constraining adapting neural controller have identical performance.

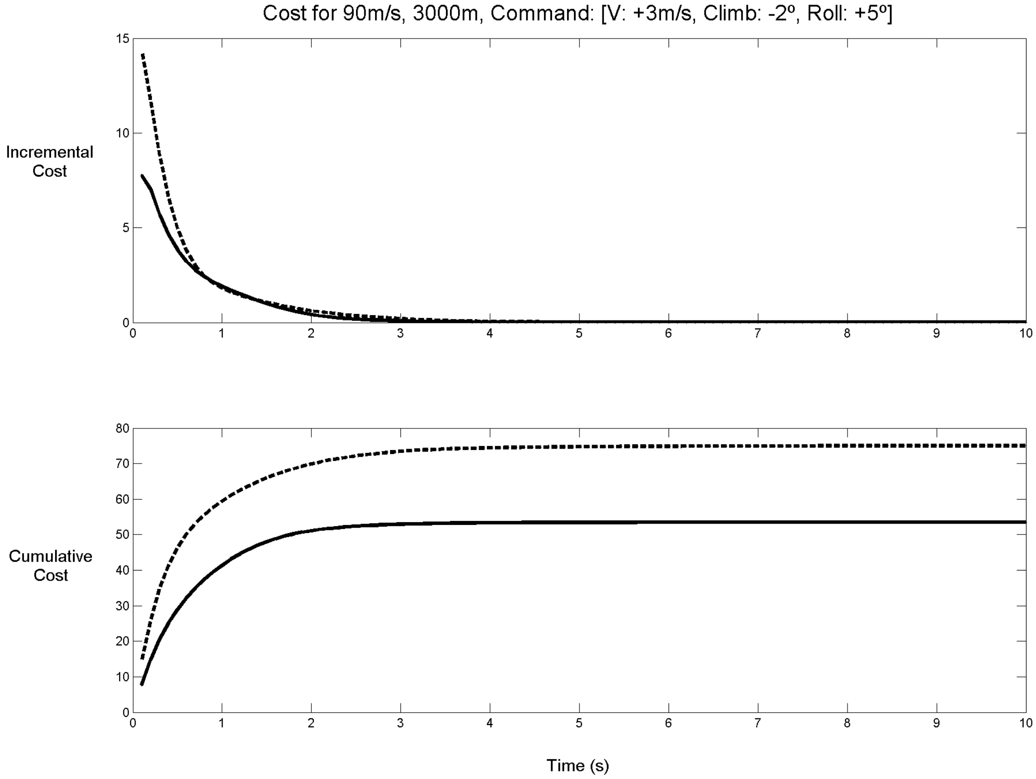

**Figure 5.37**: Incremental and cumulative cost at the design point with a coupled longitudinal-lateral-directional command (at [$90^m/_s$, $3000m$] with a command of [$+3^m/_s$, $-2°$, $+5°$, $+0°$] ). The ideal controller is represented by the solid line. The unconstrained adapting neural controller is represented by the dashed line. The constrained adaptive neural controller is represented by the dotted line. During this simulation, the neural controllers adaptation functions are turned off. The linear and constraining adapting neural controller have identical performance.

## 5.5 Chapter Summary

In this chapter, the performance of the neural control system obtained in chapter 4 is examined at a design point, an interpolation point and an extrapolation point (figure 5.38). The performance of the adapting neural controller is compared with the performances of a linear controller and a non-adapting version of the neural controller. At the design point, all three controllers perform identically, which is to be expected given the algorithm used to build the networks. At interpolation points the adapting neural controller experiences a period of suboptimal performance at the beginning of two of the simulations due to a suboptimal critic network. The critic network quickly adapts and the adaptive neural controller returns to optimality. At extrapolation points, the adaptive neural controller significantly improves performance.

The constrained adapting neural controller is also compared against an unconstrained version of the neural controller. After flying large angle maneuvers at an interpolation point, the performance of the unconstrained controller does not match the performance of the linear controller at a design point. The constrained controller performs better at the interpolation point, retains the qualities of the gain-scheduled design at the design points and matches the linear controller in performance at the design point.

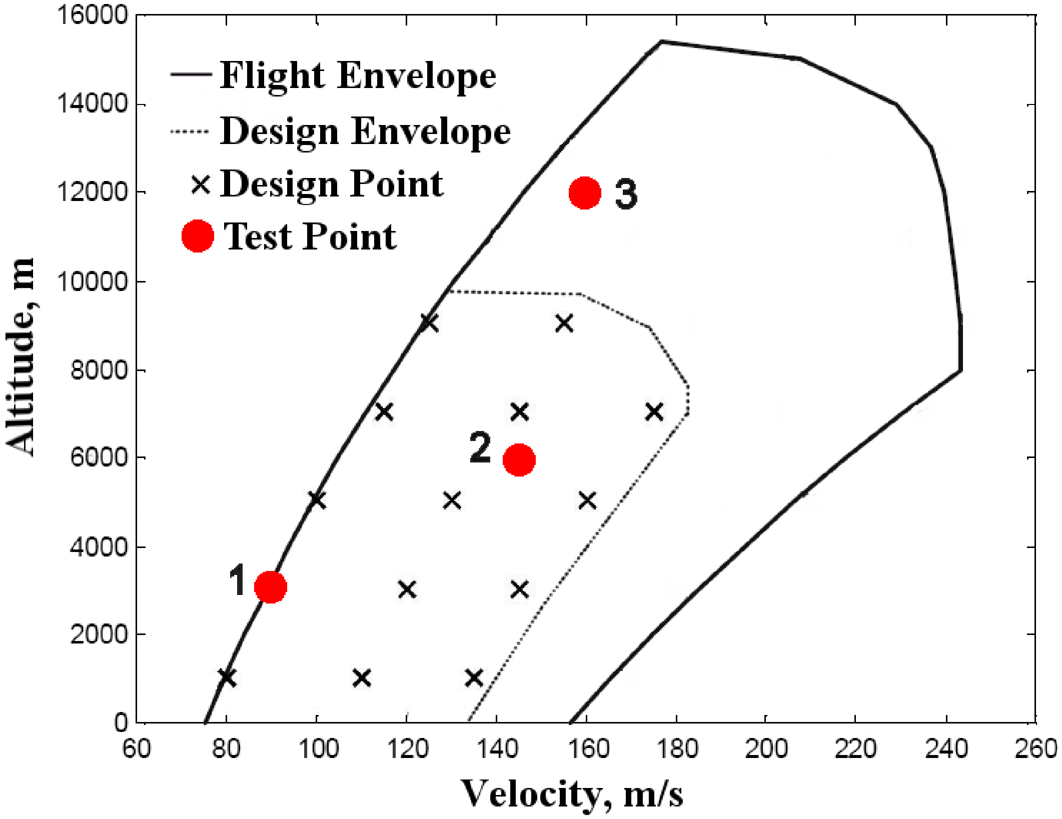

**Figure 5.38**: The flight envelope with design points and test points. Test points: (1) a design point, (2) an interpolation point, (3) an extrapolation point

# Chapter 6

# Conclusions

This research investigates the design of an adaptive neural controller applicable to plants described by nonlinear ordinary differential equations that can be designed and optimized online in order to improve the performance of the system while retaining the qualities of a gain-scheduled design at the design points. This dissertation provides the theoretical framework for the online training of such a controller. A dual-heuristic programming adaptive critic is used for adapting the control system subject to the online nonlinear plant dynamics. Sigmoidal neural networks are used to approximate the optimal control law and the costate values for the DHP controller. The neural networks are gain-scheduled based on the proportional-integral control laws obtained at fourteen design points. Construction functions are used to initialize and constrain the network by defining the constrained weights as a function of the unconstrained weights. The auxiliary input weights are determined using dual-point hyperspherical initialization. In training the neural networks, the gradient transformation is used to determine the error gradient with respect to the weights given that the weights are constrained by the construction functions. The resulting constrained error gradient is fed to a modified version of the resilient backpropagation algorithm that updates the weights to improve the approximation of the control law (in the action network's case) or the costate values (in the critic network's case).

As shown in chapter 5, the resulting controller performs very well. For a near-optimal controller, the performance is slightly degraded upon commencement of training. This is to be expected when the DHP adaptive critic architecture tries to training the near-optimal action network by means of a suboptimal critic network.

Once the critic network is appropriately trained, the control systems rapidly adjusts its performance accordingly. In scenarios where unmodeled, nonlinear dynamics are encountered, the controller provides greatly improved performance compared to the non-adapting neural controller and, when given large commands, performs better than the linear controller optimized for the test point. Throughout the entire simulation, the controller retains the qualities of the gain-scheduled design at the design points. This, along with the improvements made by the controller during simulation, indicates that the controller is optimizing the interpolation and extrapolation capabilities of the controller. Thus, an adaptive neural controller can be designed and optimized online such that the controller retains the qualities of a gain-scheduled design.

## 6.1 Recommendations for Future Research

The gain-scheduled design used to constrain the adaptive neural controller is based on an offline model of the aircraft. During flight, control failures or other problems can cause changes to the aircraft dynamics at the design points. Therefore, one recommendation for future research is the investigation of the use of a neural network to model the aircraft dynamics. Online changes to this model could then be used to update neural controller constraints. Another recommendation is to investigate the cost savings realized by using a parallel hardware implementation. This is a necessary step towards a real-world implementation of the control design. Lastly, the performance of the controller is suboptimal during the start of a few simulations because of poor interpolation by the critic network. An investigation of various node functions should be done to determine the best function for interpolating the critic network. Better interpolation of the costate function will result in improved performance from the constrained adaptive neural controller.

# Appendix A

# Hyperspherical Initialization

The spherical initialization algorithm is based on two observations relating a neural network layer with normalize inputs and an $n$-dimenional unit sphere. The first observation is that that a uniform distribution of $p$ points on an $n$-dimensional unit sphere corresponds to a uniform distribution of a layer's weights when there are $p$ nodes in the layer and $n$ nodes in the previous layer. The second observation is that the uniform distribution of $p$ dual-points on an $n$-dimensional unit sphere corresponds to a $p$ x $n$ matrix with condition number 1. A dual-point is defined as two points that are separated by a distance of twice the hypersphere's radius. A matrix with condition number of 1 can be used, as in section 3.2.3, to evenly distribute the layer's sigmoids across the input or output space. Therefore, an algorithm used to determine the coordinates for such point and dual-point distributions can be used to generate layer weights.

The following sections detail how to uniformly distribute points and dual-points over a hypersphere. The algorithms work by setting up a dynamic system which has stable equilibriums with points uniformly distributed or nearly uniformly distributed.

## A.1 Distributing points on a unit hypersphere

The process outlined in this section is called *Hyperspherical Initialization* and abbreviated HSI. Let there be $p$ points to be distributed on an $n$-dimensional hypersphere. The first step is to generate $p$ $n$-dimensional points. This can be accomplished by letting individual coordinate values to be picked randomly from a uniform distribution over $[-1, 1]$. Other methods can also be used to generate the coordinates, such as the Nguyen-Widrow (NW) initialization algorithm [21] or the method presented by Ferrari in [20].

A vector, $V_{P_i}$ from the origin can be computed for each point $P_i$. Scaling $V_{P_i}$ to be a unit vector places $P_i$ on the surface of a unit $n$-dimensional sphere. Let there be a force caused by the presence of $P_j$ which acts on $P_i$, where $P_i \neq P_j$, such that

$$F_{P_i,P_j} = \frac{V_{P_i} - V_{P_j}}{\left|\left|V_{P_i} - V_{P_j}\right|\right|^3} \tag{A.1}$$

Let no point induce a force upon itself. This system may have more than one stable equilibrium, but all equilibriums will involve an almost uniform distribution of points. Non-uniform equilibriums become possible when the number of points greatly exceeds the number of dimensions. In these cases, hyperrings can form on the hypersurface which keep points from entering the interior of the hyperring. Even so, outside of any hyperrings, the points will be uniformly distributed.

To compute the equilibrium positions of all the points, it is possible to assign each point a mass and numerically integrate, cancelling out all forces which are normal to the hypersphere's surface. Alternatively, the system can be thought of as discrete-time in which the forces change only at discrete intervals in time. Additionally, assuming that after each interval, all velocities are zero implies that the path each point travels in each time interval is line in the direction of the net force on the particle. Computing the new location of the point no longer requires the numerical integration of a complex system. After each interval, the position of each point is scaled so that the point is back on the surface of the hypersphere. These assumptions about the system dynamics do not change the stable equilibrium where all points are evenly distributed on the surface, however they make such an equilibrium computationally simpler to obtain.

## A.2 Distributing dual-points on a unit hypersphere

The process outlined in this section is called *Dual-point Hyperspherical Initialization* and abbreviated DPHSI. Uniformly distributing dual-points is equivalent to uniformly distributing vectors in $n$-dimensions. A matrix of uniformly distributed vectors has a condition number of one. Therefore, this process is useful when seeking a randomized matrix of condition number one. The process for distributing $p$ dual-points on an $n$-dimensional sphere is almost identical to the above process. The only difference between the two is the definition of the forces acting between points. The definition used in DPHSI allows a point $P_i$ to be effected by both $P_j$ and the reflection of $P_j$ where the reflection of $P_j$ is a point with coordinates $-V_{P_j}$. Reflections are not affected by other points or by other reflections. Thus the force equation becomes

$$F_{P_i,P_j} = \frac{V_{P_i} - V_{P_j}}{\left|\left|V_{P_i} - V_{P_j}\right|\right|^3} + \frac{V_{P_i} + V_{P_j}}{\left|\left|V_{P_i} - V_{P_j}\right|\right|^3} \tag{A.2}$$

Using the discrete-time assumptions as made in the previous section allow the system to progress towards an equilibrium in which a matrix composed of the coordinates of the points has condition number one.

## A.3 Effectiveness

Feedforward neural networks with single hidden layers of various size are used to compare the effectiveness of HSI with random initialization, NW initialization and a combination of HSI and NW initialization where NW is used to provide the initial point coordinates for HSI. The neural networks are then trained to imitate the function $f(x, y) = x^2 + y^2 + x * y + 3$. Thirty random inputs are selected from $-4 < x < 4$ and $-4 < y < 4$ and paired with their the corresponding outputs to form the training set. Thirty additional random inputs and corresponding outputs are selected in the same manner to form the test set. The input weights are generated using one of the above algorithms and the output weights are all set to initially be zero. This gives each network the same starting error amount. Using the Neural Networks package for Matlab, the networks are each trained for 1000 epochs using the Levenberg-Marquardt algorithm. The mean squared error for the training set and the means squared error for the test set are measured for each neural network and compared. For each hidden layer size, this routine is performed 30 times. Figures A.1-A.6 show histograms of the orders of magnitude of the mean squared error for training and test sets. For small numbers of hidden layer nodes, all algorithms perform equally well. However, as the number of hidden layer nodes grow, the HSI-based neural networks perform much better on the test sets and the non-HSI neural networks much better on the training sets. This implies that the HSI-based networks are less suspectable to overfitting their data.

## A.4 Conclusion

The hyperspherical initialization algorithm does much better job at protecting from overfitting than the Nguyen-Widrow initialization algorithm does. Less overfitting translates into better global optimization during training. The benefits of hyperspherical initialization clearly outweigh the extra computational cost.

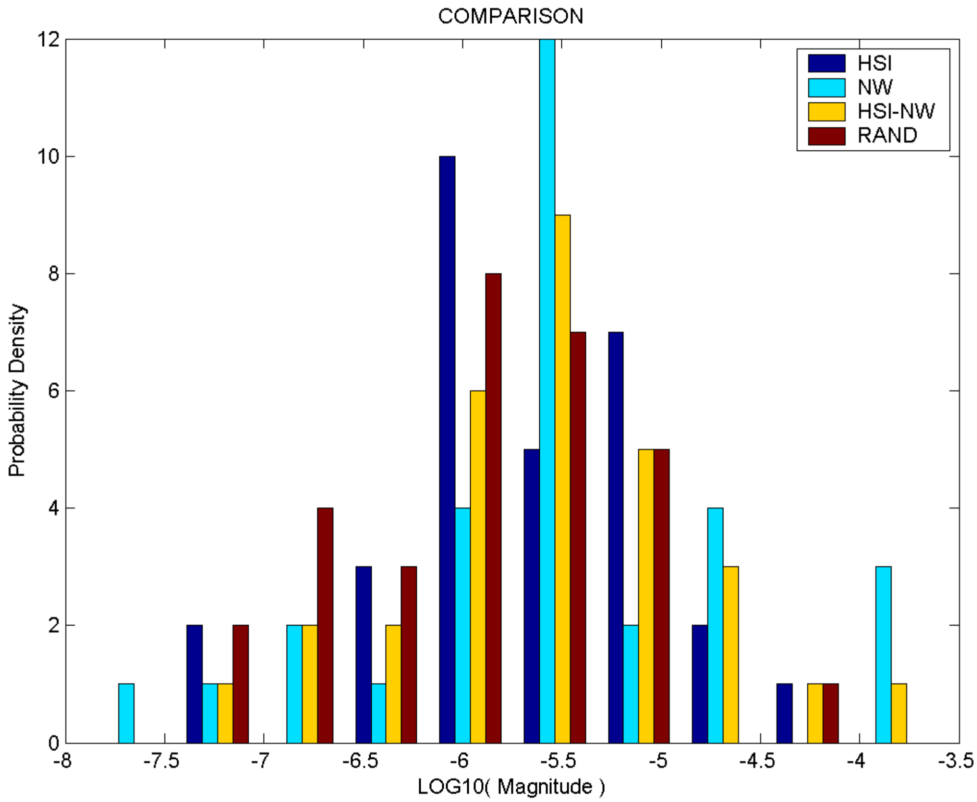

**Figure A.1**: A comparison of the training set MSE after 1000 epochs for different weight initialization algorithms (10 hidden layer nodes)

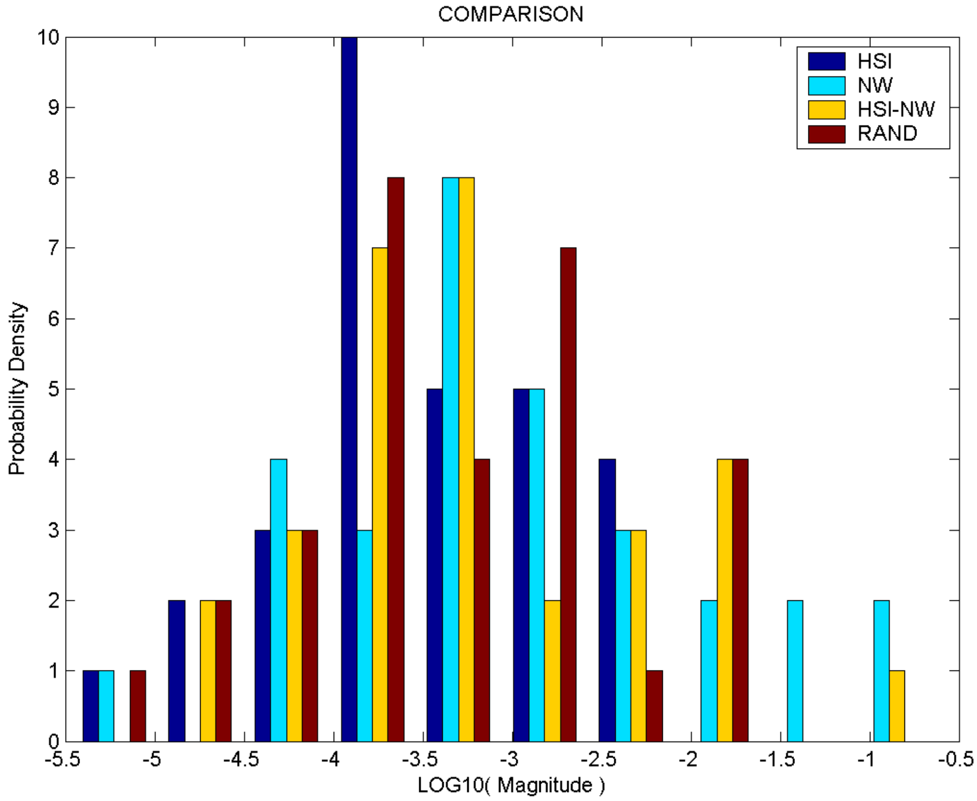

**Figure A.2**: A comparison of the test set MSE after 1000 epochs for different weight initialization algorithms (10 hidden layer nodes)

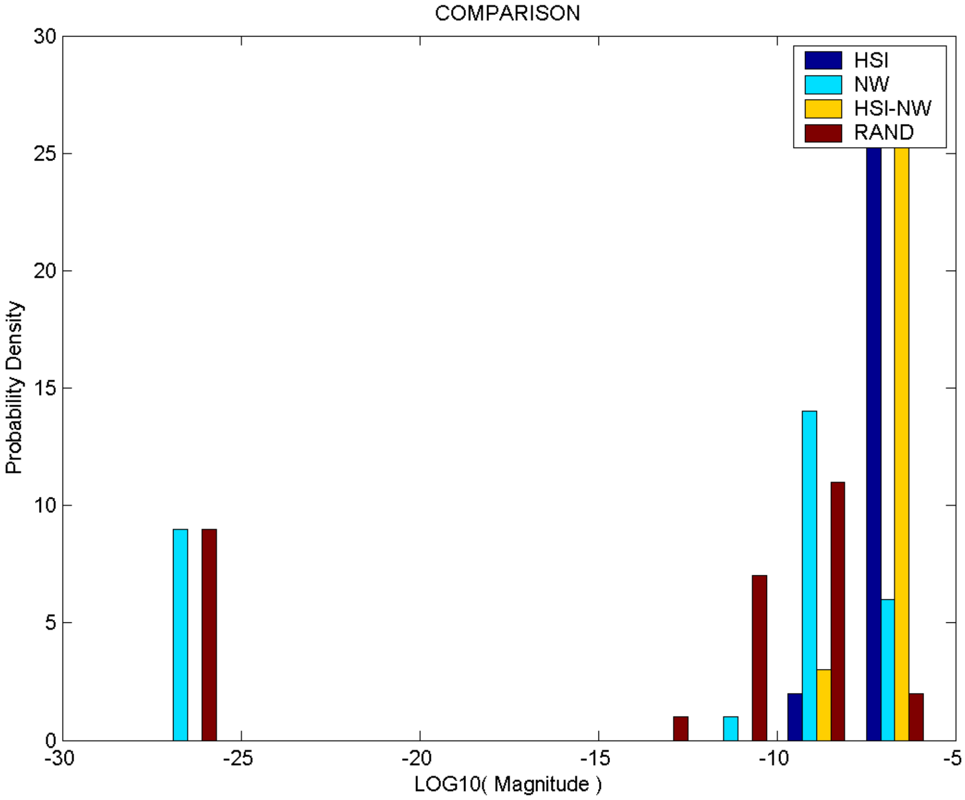

**Figure A.3**: A comparison of the training set MSE after 1000 epochs for different weight initialization algorithms (50 hidden layer nodes)

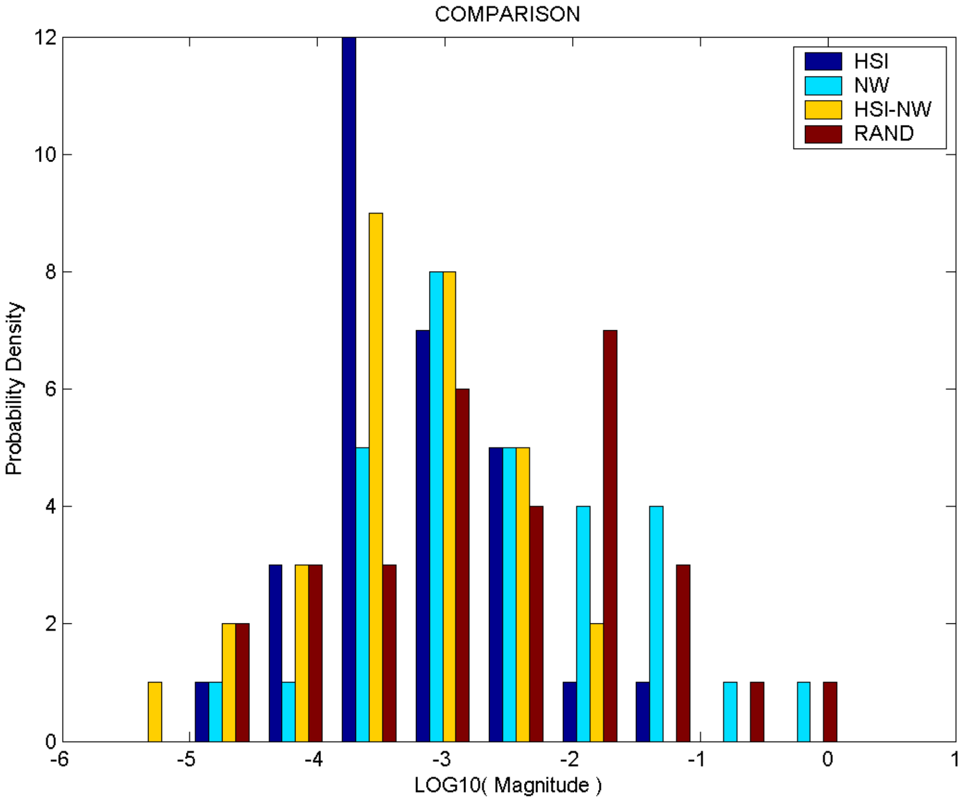

**Figure A.4**: A comparison of the test set MSE after 1000 epochs for different weight initialization algorithms (50 hidden layer nodes)

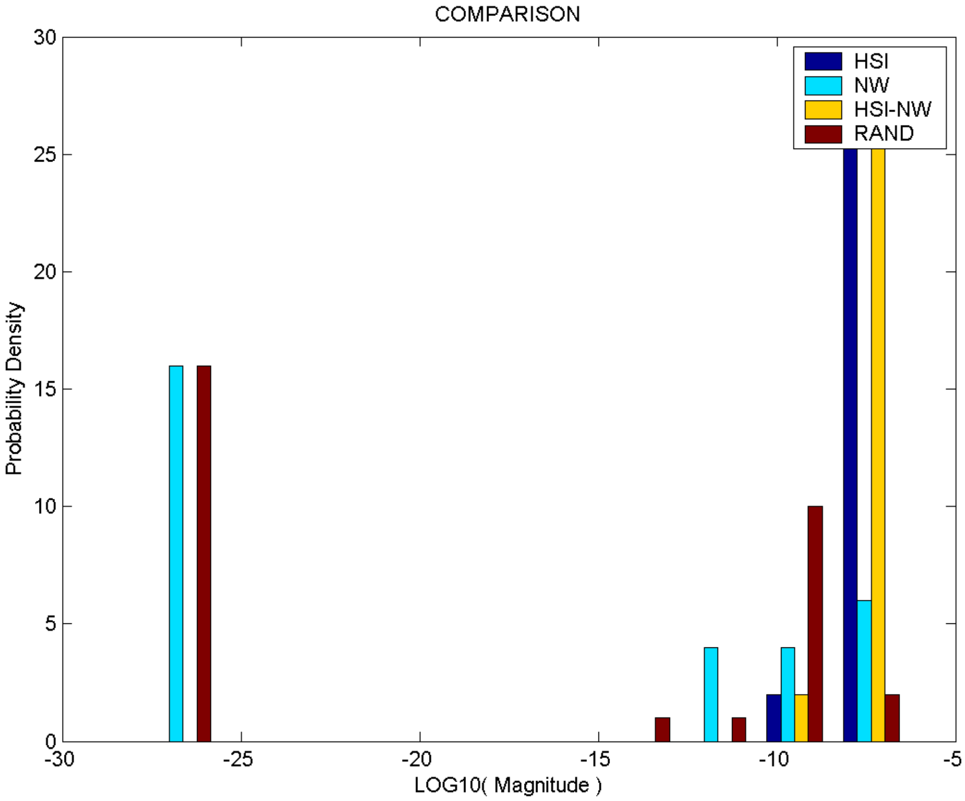

**Figure A.5**: A comparison of the training set MSE after 1000 epochs for different weight initialization algorithms (100 hidden layer nodes)

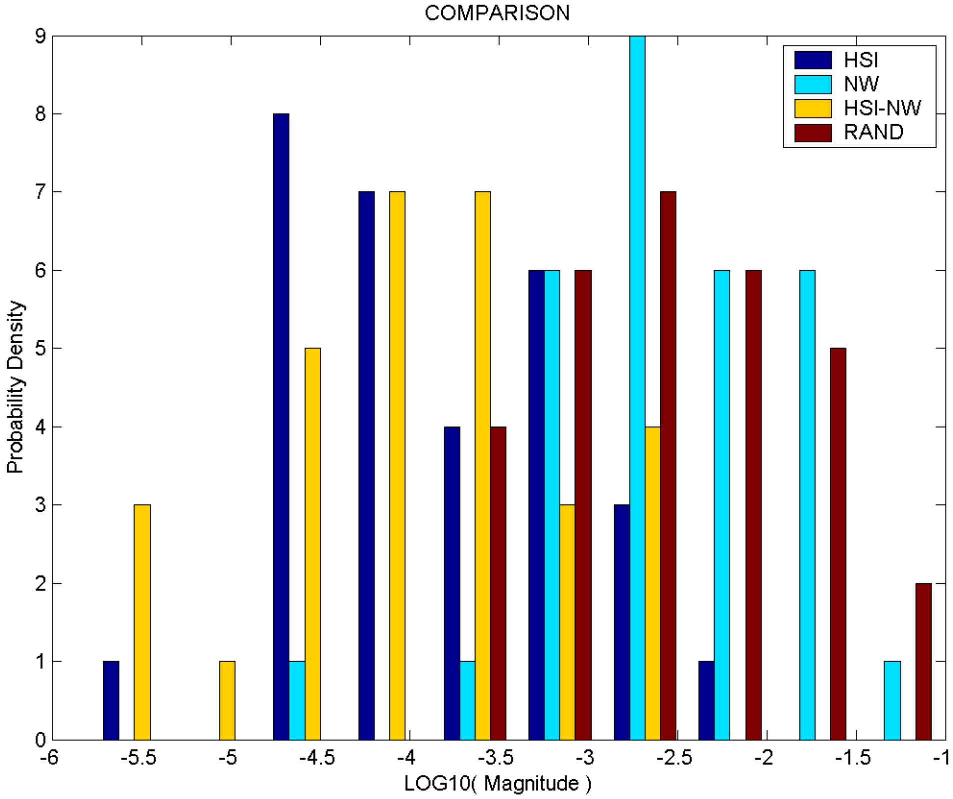

**Figure A.6**: A comparison of the test set MSE after 1000 epochs for different weight initialization algorithms (100 hidden layer nodes)

# Appendix B

# Gradient Transformation

The gradient transformation is the extension of the chain rule to matrix algebra. The gradient transformation is used to determine the value of one gradient matrix based on another, related gradient matrix. This is useful in computing gradients in a situation where many dependencies exist between variables. For instances let $e$ be some scalar values determined from the matrices $\mathbf{A}$ and $\mathbf{B}$. Then let $\mathbf{A}$ be defined as a function of $\mathbf{B}$.

$$\mathbf{A} = \mathbf{f}(\mathbf{B}) \tag{B.1}$$

The derivatives of $e$ with respect to each element of $\mathbf{B}$ can be determined using scalar algebra and the chain rule. This is a tedious process and is unnecessary. The gradient transformation defines how to compute all the derivatives at once. Borrowing notation from section 2.5.1, the unconstrained gradient of $e$ with respect to $\mathbf{A}$ is $\breve{\mathbf{A}}$ and with respect to $\mathbf{B}$ is $\breve{\mathbf{B}}$. However, $\breve{\mathbf{B}}$ does not reflect the actual gradient of $e$ because of the dependency in equation B.1. Since the value of $\mathbf{A}$ is constrained by this equation, the constrained error gradient is the value that will give the true error gradient. The constrained error gradient with respect to $\mathbf{B}$ is expressed as

$$\hat{\mathbf{B}} = \breve{\mathbf{B}} + \mathcal{G}[\mathbf{A}, \mathbf{B}, \breve{\mathbf{A}}] \tag{B.2}$$

where the final term is called the gradient transformation of $\mathbf{A}$ with respect to $\mathbf{B}$ given the unconstrained gradient $\breve{\mathbf{A}}$. The gradient transformation is applied recursively. For instance, if

$$\mathbf{B} = \mathbf{g}(\mathbf{C}) \tag{B.3}$$

then

$$\hat{\mathbf{C}} = \mathcal{G}[\mathbf{B}, \mathbf{C}, \hat{\mathbf{B}}] = \mathcal{G}[\mathbf{B}, \mathbf{C}, \breve{\mathbf{B}} + \mathcal{G}[\mathbf{A}, \mathbf{B}, \breve{\mathbf{A}}]] \tag{B.4}$$

Note that because $e$ is not directly a function of $\mathbf{C}$, there is no $\breve{\mathbf{C}}$ in the equation for $\hat{\mathbf{C}}$. For all of the following cases it is assumed that the matrix $\mathbf{A}$ does not directly affect the scalar value.

## B.1 Matrix Transpose

The transpose is the simplest function of which to compute a gradient transformation. Let the gradient $\hat{\mathbf{B}}$ be known and let $\mathbf{B}$ be defined as

$$\mathbf{B} = \mathbf{A}^T \tag{B.5}$$

Since the element $\mathbf{B}_{(i,j)} = \mathbf{A}_{(j,i)}$, the chain rule implies that $\hat{\mathbf{A}}_{(j,i)} = 1\hat{\mathbf{B}}_{(j,i)}$. Thus, the gradient with respect to $\mathbf{A}$ is

$$\hat{\mathbf{A}} = \hat{\mathbf{B}}^T \tag{B.6}$$

## B.2 Linear Functions

The gradient transformation of a linear function is also fairly simple to compute. Let the gradient $\hat{\mathbf{B}}$ be known and let $\mathbf{B}$ be defined as

$$\mathbf{B} = \mathbf{C}\mathbf{A} \tag{B.7}$$

The element $\mathbf{B}_{(i,j)} = \sum_k \mathbf{C}_{(i,k)}\mathbf{A}_{(k,j)}$. Thus, the chain rule implies that $\hat{\mathbf{A}}_{(i,j)} = \sum_i \mathbf{C}_{(i,k)}\hat{\mathbf{B}}_{(i,j)}$. Therefore, the gradient with respect to $\mathbf{A}$ is

$$\hat{\mathbf{A}} = \mathbf{C}^T\hat{\mathbf{B}} \tag{B.8}$$

Similarly, for the function

$$\mathbf{B} = \mathbf{A}\mathbf{C} \tag{B.9}$$

the gradient with respect to $\mathbf{A}$ is

$$\hat{\mathbf{A}} = \hat{\mathbf{B}}\mathbf{C}^T \tag{B.10}$$

This can easily be extended to functions like

$$\mathbf{B} = \mathbf{A}\mathbf{A} \tag{B.11}$$

where the gradient with respect to $\mathbf{A}$ is

$$\hat{\mathbf{A}} = \hat{\mathbf{B}}\mathbf{A}^T + \mathbf{A}^T\hat{\mathbf{B}} \tag{B.12}$$

## B.3 Matrix Inverse

The gradient transformation of the inverse function is much more complex than the transformation over the a linear function. Consider the adjoint matrix method of computing the inverse of a matrix

$$\mathbf{B} = \mathbf{A}^{-1} = \frac{adj(\mathbf{A})}{|\mathbf{A}|} = \frac{cof(\mathbf{A})^T}{|\mathbf{A}|} \tag{B.13}$$

where $adj(\mathbf{A})$ is the adjoint matrix of $\mathbf{A}$, $cof(\mathbf{A})$ is the cofactor matrix of $\mathbf{A}$ and $|\ |$ is the determinant of the enclosed matrix. The cofactor matrix is defined as

$$cof(\mathbf{A})_{(i,j)} = -1^{(i+j)}\,|\mathbf{C}_{ij}| \tag{B.14}$$

where $\mathbf{C}_{ij}$ equals the matrix $\mathbf{A}$ with row $i$ and column $j$ removed. From this definition,

$$\mathbf{B}_{(j,i)} = -1^{(i+j)}\frac{|\mathbf{C}_{ij}|}{|\mathbf{A}|} \tag{B.15}$$

Now, the derivative of $\mathbf{B}_{(j,i)}$ with respect to $\mathbf{A}_{(k,l)}$ can be expressed as

$$\frac{d\mathbf{B}_{(j,i)}}{d\mathbf{A}_{(k,l)}} = -1^{(i+j)}\frac{\frac{d|\mathbf{C}_{ij}|}{d\mathbf{A}_{(k,l)}}\,|\mathbf{A}| - |\mathbf{C}_{ij}|\,\frac{d|\mathbf{A}|}{d\mathbf{A}_{(k,l)}}}{|\mathbf{A}|^2} \tag{B.16}$$

From the definition of the determinant and the definition of $\mathbf{C}_{ij}$,

$$\frac{d\,|\mathbf{C}_{ij}|}{d\mathbf{A}_{(k,l)}} = \begin{cases} cof(\mathbf{C}_{ij})_{(k,l)} & k < i\, and\, l < j \\ cof(\mathbf{C}_{ij})_{(k-1,l)} & k > i\, and\, l < j \\ cof(\mathbf{C}_{ij})_{(k,l-1)} & k < i\, and\, l > j \\ cof(\mathbf{C}_{ij})_{(k-1,l-1)} & k > i\, and\, l > j \\ 0 & otherwise \end{cases} \tag{B.17}$$

and

$$\frac{d\left|\mathbf{A}\right|}{d\mathbf{A}_{(k,l)}} = cof(\mathbf{A})_{(k,l)} \tag{B.18}$$

Let $\mathbf{D}_{ij}$ be equal to $cof(\mathbf{C}_{ij})$ with an extra row of zeros is inserted as the $i^{th}$ row and an extra column of zeros is inserted as the $j^{th}$ column. Now assuming that $\hat{\mathbf{B}}$ is known, $\hat{\mathbf{A}}$ can be expressed as

$$\hat{\mathbf{A}} = \sum_i \sum_j -1^{(i+j)} \hat{\mathbf{B}}_{(j,i)} \left[ \frac{\mathbf{D}_{ij}\left|\mathbf{A}\right| - \left|\mathbf{C}_{ij}\right| cof(\mathbf{A})}{\left|\mathbf{A}\right|^2} \right] \tag{B.19}$$

Let $\mathbf{E}_{ij}$ be equal to $\mathbf{C}_{ij}^{-1}$ with an extra row of zeros is inserted as the $i^{th}$ row and an extra column of zeros is inserted as the $j^{th}$ column. Equation B.19 simplifies to

$$\hat{\mathbf{A}} = \sum_i \sum_j \mathbf{B}_{(j,i)} \hat{\mathbf{B}}_{(j,i)} \left[ \mathbf{E}_{ij} - \mathbf{A}^{-1} \right]^T \tag{B.20}$$

This is the gradient with respect to $\mathbf{A}$ given $\hat{\mathbf{B}}$. In the event that the matrix $\mathbf{C}_{ij}$ has a derivative of zero, equation B.19 may be used. Note that computing the gradient transformation over an inverse is a computationally expensive task. Computing the gradient transformation of the inverse of an $n$ x $n$ matrix requires $n^2$ inversions of an $(n-1)$ x $(n-1)$ matrix.

## B.4 Positional Function

Positional functions are functions that concatenate, split, or otherwise rearrange the position of elements in a matrix. For instance, consider the function

$$\mathbf{B} = \left[ \begin{array}{ccc} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_1 \end{array} \right] \tag{B.21}$$

Let $\mathbf{B}_1$ be the left portion of $\mathbf{B}$ which corresponds to $\mathbf{A}_1$. Let $\mathbf{B}_2$ be the center portion of $\mathbf{B}$ which corresponds to $\mathbf{A}_2$. Let $\mathbf{B}_3$ be the right portion of $\mathbf{B}$ which corresponds to $\mathbf{A}_1$. If $\hat{\mathbf{B}}$ is known, then

$$\hat{\mathbf{A}}_1 = \hat{\mathbf{B}}_1 + \hat{\mathbf{B}}_3 \tag{B.22}$$

and

$$\hat{\mathbf{A}}_2 = \hat{\mathbf{B}}_2 \tag{B.23}$$

Now consider the function

$$\mathbf{A} = \left[ \begin{array}{cc} \mathbf{A}_1 & \mathbf{A}_2 \end{array} \right] \tag{B.24}$$

If $\hat{\mathbf{A}}_1$ and $\hat{\mathbf{A}}_2$ are known, then

$$\hat{\mathbf{A}} = \left[ \begin{array}{cc} \hat{\mathbf{A}}_1 & \hat{\mathbf{A}}_2 \end{array} \right] \tag{B.25}$$

## B.5 Element-wise Functions

This section deals with functions that act on the individual elements of a matrix. Consider the following equation, where $\otimes$ signifies element-wise multiplication.

$$\mathbf{B} = \mathbf{A} \otimes \mathbf{C} \tag{B.26}$$

Assume that $\hat{\mathbf{B}}$ is known. Applying the chain rule on the individual elements yields

$$\hat{\mathbf{A}} = \hat{\mathbf{B}} \otimes \mathbf{C} \tag{B.27}$$

Another type of element-wise function is

$$\mathbf{B} = f(\mathbf{A}) \tag{B.28}$$

where $f$ operates on each element individually. In this case, assuming $f$ is differentiable at every element and that $\hat{\mathbf{B}}$ is known, the chain rule yields

$$\hat{\mathbf{A}} = \hat{\mathbf{B}} \otimes f'(\mathbf{A}) \tag{B.29}$$

where $f'$ is the derivative of the scalar function $f$ with respect to its argument.

## B.6 Concluding Remarks

Each of the individual gradient transformations mentioned above is relatively straightforward. The power of the gradient transformation comes from its ability to handle any function which is a combination of these functions. The gradient transformation is also not limited to the types of functions listed above. The above list is given to cover the types of functions used in this thesis.

# Appendix C

# Aircraft Model

This appendix reviews the equations of motion and the state elements for the simulated aircraft. The aircraft is a business jet with two turbojet engines providing a total of 26, 423N of thrust at sea level and 11, 735N of thrust at 10, 000m. The aircraft has a gross cruising weight of 4, 536kg and a nominal cruising Mach number of 0.79. The service and performance ceilings (from [23]) are 15, 315m and 15, 275m, respectively. A full explanation of the physical and performance characteristics modeled by the business jet simulation appears in [23]. The simulation models estimate low-angle-of-attach Mach effects, power effects and the moments and products of inertia by using data from full-scale wind tunnel tests according to the methods outlined in [23].

The following nonlinear equations govern the motion of the aircraft [23]. Note that the velocities and angular rates are with respect to the aircraft body-axes and the position is relative to an inertial frame of reference.

$$\dot{u} = X_b + g_x + rv - qw \tag{C.1}$$

$$\dot{v} = Y_b + g_{b_y} + pw - ru \tag{C.2}$$

$$\dot{w} = Z_b + g_{b_z} + qu - pv \tag{C.3}$$

$$\begin{aligned} \dot{x}_r &= u\cos\theta\cos\psi + v(\sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi) \\ &+ w(\cos\phi\sin\theta\cos\psi - \sin\phi\sin\psi) \end{aligned} \tag{C.4}$$

$$\begin{aligned} \dot{y}_r &= u\cos\theta\sin\psi + v(\sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi) \\ &+ w(\cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi) \end{aligned} \tag{C.5}$$

$$\dot{z}_r = -u\sin\theta + v\sin\phi\cos\theta + w\cos\phi\cos\theta \tag{C.6}$$

$$\dot{p} = \frac{q\left(I_{zz}L_b + I_{xz}N_b - p\left(I_{xz}\left(I_{yy} - I_{xx} - I_{zz}\right)\right) + r\left(I_{xz}^2 + I_{zz}\left(I_{zz} - I_{yy}\right)\right)\right)}{I_{xx}I_{zz} - I_{xz}^2} \tag{C.7}$$

$$\dot{q} = \frac{\left(M_b - pr\left(I_{xx} - I_{zz}\right) - I_{xz}\left(p^2 - r^2\right)\right)}{I_{yy}} \tag{C.8}$$

$$\dot{r} = \frac{q\left(I_{xz}L_b + I_{zz}N_b + r\left(I_{xz}\left(I_{yy} - I_{xx} - I_{zz}\right)\right) + p\left(I_{xz}^2 + I_{xx}\left(I_{xx} - I_{yy}\right)\right)\right)}{I_{xx}I_{zz} - I_{xz}^2} \tag{C.9}$$

$$\dot{\phi} = p + \left(q\sin\phi + r\cos\phi\right)\tan\theta \tag{C.10}$$

$$\dot{\theta} = q\cos\phi - r\sin\phi \tag{C.11}$$

$$\dot{\psi} = \frac{q\sin\phi + r\cos\phi}{\cos\theta} \tag{C.12}$$

The body-axis gravity components are a function of the state. The state accelerations, $X_b$, $Y_b$, $Z_b$, $L_b$, $M_b$, and $N_b$ are functions of both the state and the control values. The moments and products of inertia are estimated using simplified mass distributions and have fixed values during the simulation. References [23], [25] and [26] contain detailed descriptions of the aircraft angles and the coordinate transformations.

The state vector used during the simulation is $\mathbf{x} = [\begin{array}{cccccccc} V & \gamma & q & \theta & r & \beta & p & \mu \end{array}]^T$. This state values not specified in the above set of equations can be obtained through the following set of equations.

$$V = \sqrt{u^2 + v^2 + w^2} \tag{C.13}$$

$$\beta = \sin^{-1}\left(v/V\right) \tag{C.14}$$

$$\gamma = \sin^{-1}\left(-w/V\right) \tag{C.15}$$

$$\mu = \sin^{-1}\left(\frac{\cos\theta\sin\phi\cos\beta + \left(\cos\alpha\sin\theta - \sin\alpha\cos\theta\cos\phi\right)\sin\beta}{\cos\gamma}\right) \tag{C.16}$$

where $\alpha$ is the angle of attack and is defined as

$$\alpha = \cos^{-1}\left(\frac{\cos\gamma\cos\theta + \sin\gamma\sin\theta}{\cos\beta}\right) \tag{C.17}$$

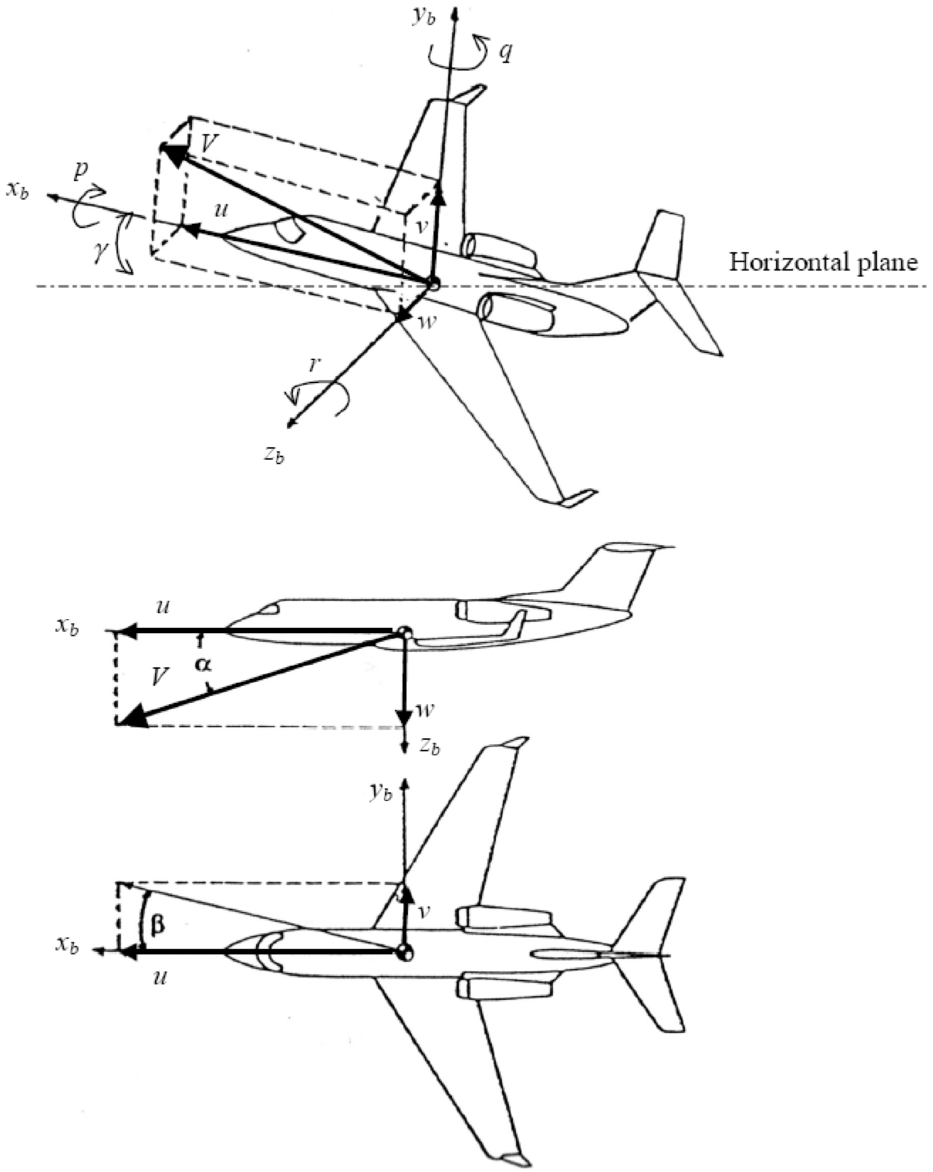

**Figure C.1**: Definition of path angle, angle of attack and sideslip from [17] (adapted from [26]).

# References

[1] Aude Billard and Auke Jan Ijspeert. Biologically inspired neural controllers for motor control in a quadruped robot. In *Proceedings of the International Joint Conference on Neural Networks*, volume 6, pages 637–641, Como, Italy, 2000. IEEE Press.

[2] Siri Weerasooriya and Mohamed A. El-Sharkawi. Laboratory implementation of a neural network trajectory controller for a DC motor. *IEEE Transactions on Energy Conversion*, 8(1):107–113, 1993.

[3] Won Seok Oh, Bimal K. Bose, Kyu Min Cho, and Hee Jun Kim. Self tuning neural network controller for induction motor drives. In *IECON Proceedings (Industrial Electronics Conference*, volume 1, pages 152–156, Sevilla, Spain, 2002. IEEE Computer Society.

[4] S. Ferrari and R.F. Stengel. An adaptive critic global controller. In *Proceedings of the American Control Conference*, volume 4, pages 2665–2670, New York, NY, 2002. IEEE Press.

[5] Silvia Ferrari and Robert F. Stengel. *Handbook of Learning and Approximate Dynamic Programming*, chapter Model-based Adaptive Critic Designs. IEEE Press and John Wiley & Sons, Piscataway, NJ, 2004.

[6] David A. White and Donald A. Sofge, editors. *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches.* Van Nostrand Reinhold, New York, 1992.

[7] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, Great Britain, 2003.

[8] R. Howard. *Dynamic Programming and Markov Processes.* MIT Press, 1960.

[9] Torkel Glad and Lennart Ljung. *Control Theory: Multivariable and Nonlinear Methods.* Taylor & Francis, 2000.

[10] Ralph W Rietz and Daniel J. Inman. Comparison of linear and nonlinear control on a distributed parameters system. *ASME Design Engineering: Vibration and Control of Mechanical Systems*, 61:43–58, 1993.

[11] Hassan Bevrani, M. Abrishamchian, and N. Safari-shad. Nonlinear and linear robust control of switching power converters. In *Proceedings of the IEEE Conference on Control Applications*, volume 1, pages 808–813, Kohala Coast-Island of Hawaii, HI, 1999. IEEE Press.

[12] Jeff S. Shamma and James R. Cloutier. Linear parameter varying approach to gain scheduled missile autopilot design. In *Proceedings of the American Control Conference*, volume 2, pages 1317–1321, Chicago, IL, 1992. American Automatic Control Council.

[13] Valery A. Ugrinovskii and Ian R. Petersen. Time-averaged robust control of stochastic partially observed uncertain systems. In *Proceedings of the IEEE Conference on Decision and Control*, volume 1, pages 784–789, Tampa, FL, 1998. American Automatic Control Council.

[14] R.F. Harrison. Non-linear stabilization and regulation via an optimal gain schedule. *IEE Colloquium (Digest)*, (521):9/1–9/3, 1998.

[15] E. Prempain. New two-degree-of-freedom gain scheduling method applied to the Lynx MK7. *Journal of Systems and Control Engineering*, 214(4):299–311, 2000.

[16] Dynamic modeling and computer control of a retort for thermal processing. *Journal of Food Engineering*, 11(4):273–289, 1990.

[17] S. Ferrari. *Algebraic and Adaptive Learning in Neural Control Systems.* PhD thesis, Princeton University, Princeton, NJ, 2002.

[18] Robert F. Stengel. *Optimal Control and Estimation.* Dover Publications, 1994.

[19] S. Ferrari and R.F. Stengel. Classical/neural synthesis of nonlinear control systems. *Journal of Guidance, Control, and Dynamics*, 25(3):442–448, 2002.

[20] S. Ferrari and R.F. Stengel. Smooth function approximation using neural networks. *IEEE Transactions on Neural Networks*, 16(1):24–38, 2005.

[21] D. Nguyen and B. Widrow. Improving the learning speed of 2-layer neural network bychoosing initial values of the adaptive weights. In *Proceedings of the IEEE First International Joint Conference on Neural Networks*, volume 3, pages 21–26, San Diego, CA, 1990. IEEE Press.

[22] Martin Riedmiller and Heinrich Braun. Direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *IEEE International Conference on Neural Networks*, pages 586–591, New York, NY, 1993. IEEE Press.

[23] Robert F. Stengel. *Flight Dynamics.* Princeton University Press, Princeton, NJ, 2004.

[24] Chien Y. Huang and Robert F. Stengel. Restructurable control using proportional-integral implicit model following. *Journal of Guidance, Control, and Dynamics*, 13(2):303–309, 1990.

[25] Marcello R.M. Crespo da Silva. *Intermediate Dynamics: Complimented with Simulations and Animations.* McGraw-Hill Science/Engineering/Math, New York, NY, first edition, 2003.

[26] Robert C. Nelson. *Flight Stability and Automatic Control.* McGraw-Hill Science/Engineering/Math, New York, NY, second edition, 1997.