Optimization of Neural Network Feedback Control Systems Using Automatic Differentiation

by
Elizabeth Rollins
B.S. Aerospace Engineering
University of Notre Dame, 2007

Submitted to the Department of Aeronautics and Astronautics
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Aeronautics and Astronautics
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 2009

Signature of Author: .....
Department of Aeronautics and Astronautics
August 20, 2009

Certified by: .....
Steven R. Hall
Professor of Aeronautics and Astronautics
MacVicar Faculty Fellow
Thesis Supervisor

Certified by: .....
Christopher W. Dever
Senior Member, Technical Staff, Draper Laboratory
Thesis Supervisor

Accepted by: .....
Prof. David L. Darmofal
Associate Department Head
Chair, Committee on Graduate Students

Optimization of Neural Network Feedback Control Systems Using Automatic Differentiation

by

Elizabeth Rollins

Submitted to the Department of Aeronautics and Astronautics
on August 20, 2009, in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Aeronautics and Astronautics

Abstract

Optimal control problems can be challenging to solve, whether using analytic or numerical methods. This thesis examines the application of an adjoint method for optimal feedback control, which combines various algorithmic techniques into an original numerical method. In the method investigated here, a neural network defines the control input in both trajectory and feedback control optimization problems. The weights of the neural network that minimize a cost function are determined by an unconstrained optimization routine. By using automatic differentiation on the code that evaluates the cost function, the gradient of the cost with respect to the weights is obtained for the gradient search phase of the optimization process. Automatic differentiation is more efficient than hand-differentiating code for the user and provides exact gradients, allowing the optimization of the neural network weights to proceed more rapidly. Another benefit of this method comes from its use of neural networks, which are able to represent complex feedback control policies, because they are general nonlinear function approximators. Neural networks also have the potential to be generalizable, meaning that a control policy found using a sufficiently rich training set will often work well for other initial conditions outside of the training set. Finally, the software implementation is modular, which means that the user only needs to adjust a few codes in order to set up the method for a specific problem. The application of the adjoint method to three control problems with known solutions demonstrates the ability of the method to determine neural networks that produce near-optimal trajectories and control policies.

Thesis Supervisor: Steven R. Hall
Title: Professor of Aeronautics and Astronautics
MacVicar Faculty Fellow

Thesis Supervisor: Christopher W. Dever
Title: Senior Member, Technical Staff, Draper Laboratory

Acknowledgments

"Forgive the song that falls so low, beneath the gratitude I owe."

- Cowper, 168 - *The Sacred Harp* [1, p. 168]

The first people whom I must thank are my two advisors, Professor Hall and Chris Dever, especially for their patience throughout the entire research process as the thesis continually changed. Their advice on the research itself and on the writing of the thesis was invaluable. Also, I would like to thank the MIT graduate students who had worked on this topic before me and who were great resources for discussing research ideas and code issues. Additionally, I would like to thank Draper Laboratory for giving me the opportunity to do research here.

Finally, and most importantly, I would like to thank my family and friends, who always were willing to listen and to provide encouragement, who reminded me to persevere and to hope, and who always knew that this thesis would be completed.

This thesis is dedicated to everyone who has believed in me during these past two years, without whose support and encouragement this thesis might never have been written.

August 20, 2009

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under project activity numbers 22026-001 and 22951-001.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

Chapter 1

Introduction

> "'What happened first?' Imogene hollered.... 'Begin at the beginning!'
>
> That really scared me because the beginning would be The Book of Genesis, where it says 'In the beginning...' and if we were going to have to start with the Book of Genesis we'd never get through."
>
> - *The Best Christmas Pageant Ever* [2, p. 39-40].

1.1 Motivation

The desired solutions to optimal control problems are optimal trajectories and optimal control policies. Optimal control policies are more useful than optimal trajectories since trajectories define the control for one initial condition over time, while control policies give the feedback control necessary for any state. However, determining complete optimal control policies is often difficult, analytically or numerically. Analytic solutions are difficult to find and often apply to a small class of problems. For example, the analytic solution to the linear quadratic regulator problem is limited to a problem with linear or linearized dynamics and a quadratic cost function, and works well only in a region about the point that was used to create the linearized dynamics model. The restriction of the region in which the controller will work well means that several controllers would have to be designed to cover an array of initial conditions. This limitation becomes evident when considering the problem of aircraft control. Often, the dynamics equations are linearized about a steady-state, horizontal flight condition. If the aircraft encounters a condition that drastically upsets its flight, a linear quadratic regulator that has been designed for steady-state, level flight may not work well enough to recover the aircraft. It is possible for gain-scheduling to be used to define the gains for various flight conditions, but that type of solution may not always be practical.

Another type of solution option for optimal control problems is the use of numerical methods, such as dynamic programming. A dynamic programming solution provides the optimal control policy for a given problem. However, the usefulness of dynamic programming is limited by the *curse of dimensionality* [3, p. 63-64], which places practical limits on the dimensionality of the state space of problems that are

solved with this method. Some search-based numerical methods may incur truncation errors because the exact gradients are not available to be used in the optimization routine. Therefore, it would be beneficial if the exact gradients could be used for certain search methods.

The adjoint method for optimal feedback control is proposed as a new numerical method that combines techniques, such as neural networks and automatic differentiation, that currently are used to solve optimization and optimal control problems. Neural networks often define the fixed controller structure in the adjoint method. By using neural networks for the control structure in this method, and by training the neural network with multiple initial conditions, it should be possible to find generalizable neural networks that can represent the optimal control policies for a region of initial conditions. In order to solve optimal control problem using the adjoint method, an unconstrained optimization routine is used to determine the weights of the neural network that achieve the goal of the problem. Automatic differentiation is applied to compute the exact gradients for the optimization process, instead of using gradients found through numerical differentiation, which should increase the accuracy of the final solution without a significant increase in computational time. The adjoint method for optimal feedback control will be validated through comparisons to known optimal control solutions for the problems examined in this thesis. It is hoped that this new method will allow optimal control policies to be found, especially for problems in which the policies may be difficult to determine analytically. It should be noted that in this thesis the adjoint method for optimal feedback control will be interchangeably referred to as simply the adjoint method.

1.2 Problem Statement

This thesis studies the solution of the optimal control problem, which is concerned with finding the optimal control u that minimizes a cost function J given the equations of motion $\dot{x}(t) = f(x(t), u(t))$ for the state x and certain boundary conditions. The cost function J has the general form

$$J = h(x(t_f), t_f) + \int_{t_o}^{t_f} g(x(t), u(t), t)\, dt \tag{1.1}$$

where J is the cost, t_o is the initial time, t_f is the final time, $x(t_f)$ is the final state of the system, $h(x(t_f), t_f)$ is a function that penalizes the final state of the system, and $g(x(t), u(t), t)$ is another function that penalizes the system over the entire time period. For trajectory optimization problems, the control is expressed as a function of time, where $u = \phi(t)$. To define an optimal control policy that can be applied to a range of states in the state space, the control is expressed as a function of the states, where $u = \phi(x)$. Both types of control functions will be studied in this thesis.

When neural networks are used as the framework for the control policy, the issue of overtraining arises. Overtraining a neural network for an optimal control problem means that the neural network has learned the control policy for the given initial condition, but it does not control the system well for other initial conditions. By using

multiple initial conditions, it may be possible to prevent overtraining, but a tradeoff will occur between the performance of a neural network trained for one initial condition and the robustness of a neural network trained with multiple initial conditions, which must be examined.

The ability of the adjoint method for optimal feedback control to solve optimal control problems will be tested in this thesis. For the adjoint method to be considered a valid method for finding optimal control solutions, it must be able to determine solutions to example problems that match the optimal solutions found using proven methods. Additionally, the training of neural networks with single and multiple initial conditions using the adjoint method will be examined. For the case of a single initial condition, the neural network must be able to represent the optimal solutions to example problems. For the case of multiple initial conditions, the neural network should be able to recognize the general optimal control policy for conditions in the training data set and to be robust enough to be able to find a near-optimal solution for initial conditions outside of the training data set.

1.3 Approach

The adjoint method for optimal feedback control will be used to solve the optimal control problem for three different examples that will illustrate the capabilities of the method. Methods that use automatic differentiation to find gradients already are used to solve problems, but the combination of gradients found using automatic differentiation with neural networks in an unconstrained optimization routine represents a new method for solving optimal control problems. This method attempts to solve the optimal control problem using an unconstrained optimization routine that minimizes the cost function directly as opposed to using the explicit equations for the necessary conditions for optimality in the minimization process. The cost function is minimized with the guidance of the gradients of the cost function with respect to parameters that define the control policy. The gradients are calculated exactly using a version of the original system simulation code that has been adjointed, or reverse differentiated, to form the equations necessary to find the gradients. The use of the exact gradients in the code should increase the accuracy of the final optimal solution with only a small increase in computational time and eliminate issues arising from the use of numerical differentiation to compute gradients. The typical issues that arise from numerical differentiation are related to the step size involved in calculating derivatives. If the step size between the points being numerically differentiated is too small, noise and precision issues can skew the final result, while if the step size is too large, the estimate of the derivative can be highly inaccurate [4, p. 2]. Therefore, a benefit of calculating gradients using code that has been automatically differentiated is that the gradients will be accurate to within computational limits, and thus the optimization routine will be able to proceed more precisely toward the final solution at a faster rate because it knows more precisely the direction in which to search for location of the minimum cost.

For each iteration of the optimization routine, the parameters that define the

control policy are adjusted in order to minimize the cost function. Neural networks predominately are chosen to be the control structure. However, it also is possible to use the adjoint method with other fixed structure controllers. For the case in which neural networks represent the control structure, the neural network can be trained with multiple initial conditions to promote the development of a generalizable neural network that may be able to control the system for initial conditions that were not included with the training set of initial conditions. Neural networks also are used to solve the trajectory optimization problem, which will confirm that the adjoint method has the capability of finding optimal control solutions. One of the benefits of the adjoint method is that the neural network is able to learn the optimal control policy through the optimization process. Therefore, the structure of the optimal controller does not have to be known in advance to use this method. Another benefit of the adjoint method is that the code is designed to be highly modular so that a user only has to enter in the dynamics of the problem, the cost function, the desired structure of the neural network, and the set of training initial conditions in order to apply the method to a given problem. Therefore, the adjoint method for optimal control should be a beneficial addition to the existing set of methods used to solve optimal control problems.

1.4 Related Work

The optimal control problem has been studied for many years, and numerous references exist on the subject. The texts by Bryson and Ho [5] and by Kirk [6] provide good foundations for the study of optimal control and include examples of optimal control problems. For an explanation of optimal control theory with examples that also include their implementation codes, a good reference is the lecture notes from How [7]. Optimization methods also have a prolific number of references. A good basis for optimization theory and its application to optimal control can be found in the text by Betts [8].

Automatic differentiation provides a process to obtain gradients directly, which is an improvement over numerical differentiation methods that often are used to calculate gradients and are subject to truncation errors. By application of the chain rule, codes can be differentiated to provide exact gradients. The texts by Griewank [4] and by Rall [9] are two references that explain the theory of automatic differentiation, using examples to show how codes are differentiated using this process. Because differentiating code by hand can be difficult, highly error-prone, and time-consuming, especially when the code has to be re-differentiated after changes are made to the original code, computer programs have been written to automate the code differentiation process. One of these programs is TAPENADE [10], which has been integrated into methods to solve optimization problems for a computational fluid dynamics problem [11] and for the sonic boom reduction on a supersonic aircraft [12]. The adjoint method differs from these two optimization problems because it is applied to optimal control problems instead of optimum design problems. The differentiated code is used in the adjoint method to guide the optimization process by providing the optimization

routine with exact gradients of the cost function to reduce numerical errors during the search for the minimum.

Although the adjoint method may be used with different types of controllers, it is used in this thesis mainly with neural networks for the control structure. General references for background on neural networks are [13], [14], and [15]. Neural networks are useful because of their flexible structure, which can be shaped to fit specific problems through the number of neurons and of layers of neurons that compose the neural networks. A definitive rule for structuring the neural network does not exist [16, p. 21], but a general rule for designing neural networks is to use enough neurons and layers necessary for the learning task without making the computational time too large [15, p. 85]. Traditionally, neural networks have been used for the problem of function approximation, as noted in the article by Hagan and Demuth [13]. They have also been applied to other types of problems, including control problems involving system identification [13][15, p. 32-34], dynamic programming problems for missile defense and interceptor allocation [17], the definition of control laws for differential game problems [18][19], and the determination of adaptive controllers with the adaptive-critic method [20]. Numerous examples of the application of neural networks to control problems are contained in [21]. One well-known example application is the "truck backer-upper" problem by Nguyen and Widrow, which uses two neural networks to find the solution [16][21, p. 287-299]. In order for one of the neural networks to learn the control law for the system, it is necessary for a second neural network to be trained to emulate the plant dynamics for use in the backpropogation algorithm that trains the first neural network. By using automatic differentiation to determine the exact gradients, the need for a secondary system emulator neural network is eliminated in the adjoint method. Also, the adjoint method allows the optimization process design the control policy by adjusting the weights of the neural network for a control problem, which is a departure from common uses of neural networks for control.

The adjoint method for optimal feedback control previously has been studied by Johnson in his master's thesis [19] with a focus on the application of the method specifically to differential game problems. This thesis aims to study the adjoint method and its application to several types of problems, including fixed structure controllers, trajectory optimization, and training over sets of multiple initial conditions. The orbit-raising problem that is examined in this thesis was also examined in his thesis [19, p. 54-56] as a part of a series of examples that compared neural network control policies and known optimal control policies before applying the adjoint method to differential game problems. This thesis will study the orbit-raising problem in more depth, specifically examining the robustness of control policies for this problem that are found by training the neural network with multiple initial conditions using the adjoint method.

1.5 Thesis Outline

The following chapters of this thesis will explain the optimal control problem and the adjoint method for optimal feedback control in further detail and present example applications. Chapter 2 will provide a basic description of the optimal control problem and methods that are currently used to solve the problem. In Chapter 3, background for the elements of the adjoint method, including neural networks, nonlinear optimization, automatic differentiation, and training with initial conditions will be given. The software implementation of the adjoint method described in Chapter 3 will be discussed in Chapter 4. Chapter 5 will present example applications of the adjoint method to the problems of linearized aircraft dynamics, a double integrator system, and raising the orbit of a spacecraft. Conclusions and future work will be discussed in Chapter 6.

Chapter 2

Optimal Control Problems

Chapter 2 presents the main equations used in the formulation of optimal control problems, as well as basic analytic and numerical methods used to solve them. The adjoint method for optimal feedback control is presented as a particular subset of numerical methods.

2.1 Optimal Control Problem Formulation

The goal of an optimal control problem is to find a control $u(t)$ that minimizes an objective, or cost, function J considering a set of state equations and boundary conditions. For the problems involved using the adjoint method for optimal feedback control, the dynamics of the system will be expressed by the state equations

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

where $x(t)$ is the state of the system at time t, and $u(t)$ is the control at time t. It should be noted that the state equations are autonomous, meaning that $\dot{x}(t)$ is not written as an explicit function of time. The state equations are specifically written to be autonomous for the modular implementation of the adjoint method, which will be described in Chapter 4.

One of the challenging aspects of optimal control problem formulation is writing the cost function in a form that correctly expresses the goal of the problem. As stated in Section 1.2, for optimal control problems, the cost function J has the general form

$$J = h(x(t_f), t_f) + \int_{t_o}^{t_f} g(x(t), u(t), t)\, dt \tag{2.2}$$

where J is the cost, t_o is the initial time, t_f is the final time, $x(t_f)$ is the final state of the system, $h(x(t_f), t_f)$ is a function that penalizes the final state of the system, and $g(x(t), u(t), t)$ is a function that penalizes the system throughout the time period.

In addition to expressing the goal of the problem, the cost function can be used to enforce equality constraints indirectly using the penalty function method. This method is useful for the case where an unconstrained optimization routine is being

used to minimize the cost function J and equality constraints exist [5, p. 39-40]. Given the equality constraint $m(x) = 0$ and the cost function J, the equality constraints can be appended to the original cost function to form a new cost function

$$J_a = J + \alpha m(x) \tag{2.3}$$

where J_a is the augmented cost function, J is the original cost function, α is a scalar parameter to weight the equality constraint, and $m(x)$ is the equality constraint. Because this method does not guarantee that the constraints will be enforced, the user must verify that the constraints have been sufficiently satisfied after the optimization process. The penalty function method will be important in the orbit-raising example problem in Section 5.3, in which equality constraints on the terminal condition of the spacecraft must be enforced.

Through the application of variational calculus, necessary conditions for the solution of the optimal control problem can be derived [7, p. 6-1–6-3]. In order to express the necessary conditions efficiently, a quantity known as the Hamiltonian is defined as

$$H(x(t), u(t), p(t), t) = g(x(t), u(t), t) + p^T(t) f(x(t), u(t), t) \tag{2.4}$$

where $H(x(t), u(t), p(t), t)$ is the Hamiltonian, and $p(t)$ is the vector of the costates, or Lagrange multipliers. With the Hamiltonian and variational calculus applied to the cost function J, the necessary conditions that must be satisfied for the optimal control solution [7, p. 6-3] are

$$\dot{x} = f(x(t), u(t), t) \tag{2.5}$$

$$\dot{p}(t) = -\frac{\partial H^T}{\partial x} \tag{2.6}$$

$$\frac{\partial H}{\partial u} = 0 \tag{2.7}$$

subject to boundary conditions, including the condition

$$h_{t_f} + H(t_f) = 0 \tag{2.8}$$

where h_{t_f} is the partial derivative of the terminal state penalty with respect to the final time, and $H(t_f)$ is the value of the Hamiltonian evaluated at the final time. This boundary condition is applicable only for free final time problems. Additional boundary conditions specific to individual types of problems are described in Table 5-1 in the text by Kirk [6, p. 200-201]. As mentioned previously, the expressions of the Hamiltonian and necessary conditions will be autonomous for the problem formulations in this thesis. Also, the formulation of the necessary conditions in Equations (2.5), (2.6), and (2.7) does not consider limitations on the control. For bounded control problems, the third necessary condition expressed in Equation (2.7) must be adjusted to account for the limits on the control. Pontryagin's Minimum Principle is one formulation of the new necessary third condition that minimizes the Hamiltonian

within the limits of the control [7, p. 9-6] and has the form

$$u^*(t) = \arg\min_{u(t)\in U} H(x, u, p, t) \tag{2.9}$$

where $u^*(t)$ is the optimal control, and U is the range of permissible values of the control u. The bounded control problem is considered in the double integrator example problem in Section 5.2.

For further discussion and background on optimal control problem formulation, good references include [6], [5], and [7].

2.2 Analytic Methods

Analytic methods sometimes can be used to find solutions to the optimal control problem. The ability to write the solution to an optimal control problem analytically is desirable because the solutions provide a closed-form description of the control necessary for a given problem. For a given state and time, the optimal control can be determined exactly. However, analytic methods usually are difficult to derive and only apply to a specific class of problems. The absence of reliable procedures for determining analytic solutions means that many analytic solutions are found by trial-and-error, which is challenging.

One analytic solution that has been found for a certain class of problems is the linear quadratic regulator (LQR) [7, p. 4-6–4-9]. This solution is applicable only to problems that have linear dynamics equations for the system of the form

$$\dot{x}(t) = A(t)\,x(t) + B(t)\,u(t) \tag{2.10}$$

where $A(t)$ is the plant dynamics matrix, and $B(t)$ is the control matrix, and that have a quadratic cost function of the form

$$J = \frac{1}{2}x^T(t_f)\,\mathcal{H}(t_f)\,x(t_f) + \frac{1}{2}\int_{t_o}^{t_f}\left\{x^T(t)\,Q\,x(t) + u^T(t)\,R\,u(t)\right\}dt \tag{2.11}$$

where $\mathcal{H}(x(t_f), t_f)$ is the final penalty matrix on the states, where $\mathcal{H} \geq 0$, Q is the penalty weighting matrix on the states, where $Q \geq 0$, and R is the penalty weighting matrix on the controls, where $R > 0$. It is also assumed that the final time is fixed and that there are no bounds on the controls.

The optimal control solution to the LQR problem is

$$u(t) = -R^{-1}B^T(t)\,P(t)\,x(t) = -K(t)\,x(t) \tag{2.12}$$

where $P(t)$ is the matrix that is the solution to the Riccati equation, and $K(t)$ is the control gain matrix, where $K(t) = R^{-1}B^T(t)\,P(t)$. The first example in Section 5.1 involves a LQR controller and a system with constant plant dynamics, control, and penalty matrices. Therefore, the Riccati equation for this constant system, which is

known as the Control Algebraic Riccati Equation [7, p. 4-9], has the form

$$PA + A^T P + Q - PBR^{-1}B^T P = 0. \tag{2.13}$$

By solving the Riccati equation for the matrix P, and using P to find the LQR gains K, the control u can be defined for any state x.

2.3 Numerical Methods

Often, it is not feasible to determine an analytic solution to an optimal control problem. Numerical methods provide a way to obtain solutions to difficult optimal control problems that cannot be solved analytically. Two main types of numerical methods include dynamic programming, and a method that casts the given problem as a nonlinear programming problem to be solved. Within the latter type, there are two very broad categories of nonlinear programming methods that are used to solve these problems, which are direct methods and indirect methods. Direct methods involve minimizing the cost function, whereas indirect methods work with the necessary conditions for optimality in Equations (2.5), (2.6), and (2.7) to find the optimal control solution. For this thesis, the adjoint method for optimal feedback control, which is a direct method, is used to solve the optimal control problem. Of the many numerical methods that exist, dynamic programming and adjoint methods for optimal feedback control will be discussed in this section.

2.3.1 Dynamic Programming

The idea that for a given problem, the path from a certain point for an optimal solution that includes another point will be the same as the path for the optimal solution that originates from the second point [7, p. 3-1] is the principle of optimality. This principle is the basis for the method of dynamic programming, where optimal trajectories are found by working backward from the final state to find the optimal solution for a given initial condition. In order to solve optimal control problems using dynamic programming, first the state space must be discretized over time. Also, the control space must be discretized. After the final costs are found for the terminal states, the iterative process of finding optimal control solutions begins, working backward from the terminal states. For each iteration, at a given time increment, the cost-to-go is evaluated at each state for each possible control input, and by finding the minimum cost from the current state to the terminal state, the optimal control and the optimal trajectory can be found to reach a final state from the current state. The principle of optimality is applied in this method since for each iteration, the optimal control solution is already known from the states at the next time step. Therefore, only the incremental costs need to be calculated for current time step. Then the costs for the optimal control solutions can be used with the incremental costs to find the total cost from each state to the terminal states. When the iterations have reached the initial point in time, the optimal control solution is known from each initial state in the

discretized set of states. Detailed explanations of the mathematics behind dynamic programming can be found in [6] and [7].

One of the major problems with dynamic programming is the *curse of dimensionality* noted by Bellman [3, p. 63-64]. For systems that have higher-dimensional state spaces and are discretized using a large number of grid points in each dimension, the space required to store the cost information can quickly become too large for a computer to handle [6, p. 78]. Therefore, dynamic programming cannot be used to solve large optimal control problems. Consequently, other numerical methods must be applied to solve these problems.

2.3.2 The Adjoint Method for Optimal Feedback Control

The adjoint method for optimal feedback control is a new direct numerical method that uses fixed structure controllers to define solutions to optimal control problems. The solutions expressed by the controllers are determined by an unconstrained optimization routine that is improved through the use of exact gradients, which are computed using automatic differentiation. The method finds the optimal control solution for a problem by using information from the gradients of the cost function with respect to certain adjustable parameters to find the minimum cost and optimal control policy. The parameters are adjusted by the optimization program iteratively, using the cost and the gradients of the cost function with respect to them to find the values of the parameters that will minimize the cost and satisfy the necessary conditions of the optimal control problem. The types of parameters that are used depend on the structure of the controller that is being implemented. For this thesis, the parameters represent either the gains for a linear quadratic regulator or the weights in a neural network. The gradients of the cost function are calculated using an adjointed, or differentiated, version of the code used to simulate the system and compute its cost. The adjointed code is determined using an automatic differentiation program. Because the adjointed code calculates the gradients directly, truncation errors are not incurred, in contrast to finite differencing. Through the use of an unconstrained optimization routine in conjunction with these gradients and a given control structure, optimal control policies can be determined for a given system within a region of its state space. The mechanics of the adjoint method used in this thesis are discussed in Chapter 3.

While the adjoint method for optimal feedback control is not affected by the traditional curse of dimensionality because the state space and control space do not have to be discretized, it is possible that the method may be affected by the size of the training data set necessary to find generalizable solutions or by the required complexity of the neural network for a given problem. In order to make an optimal control solution that is defined by a neural network generalizable, meaning that an optimal control solution can be found for initial conditions outside of those conditions used to train the neural network, a large number of initial conditions must be used in the training data set. However, a limit on the computational speed and the computational ability to handle large training data sets exists. As a result, it may not be possible to train a neural network with as many initial conditions as desired

to increase the potential for the creation of a highly generalizable optimal control policy that is defined by a neural network. Also, for higher-dimensional systems, it may be necessary to increase the complexity of the neural network so that the control policy can be accurately described throughout the state space. Increasing the complexity of the neural network involves increasing the number of weights, which consequently would increase the required computational time. Therefore, limitations on the computational speed may limit the allowable complexity of neural networks for higher-dimensional systems.

For this thesis, the adjoint method will be used to solve three example optimal control problems. One problem will examine the ability of the adjoint method to determine the LQR gains and the rate of convergence to them, while the other two problems will study the use of neural networks as the control structure for the adjoint method and their generalizability. The results of the adjoint method for each problem will be compared with known solutions that are found using either analytic methods or proven numerical methods, depending on the problem.

Chapter 3

An Adjoint Method for Optimal Feedback Control

In Chapter 2, the optimal control problem was introduced. The goal of the optimal control problem is to find a control policy that minimizes the cost function for a particular system. This chapter gives the details of the adjoint method, which combines various techniques used to solve control problems into a novel method for developing control policies. The control parametrization that is predominantly used for this method is a neural network. Nonlinear optimization is used in order to determine the weights of the neural network that will minimize the cost function. Actual gradients to be used in the optimization process are determined using automatic differentiation, which increases the accuracy and speed of the nonlinear optimization. In order to find weights that will produce a generalizable neural network that represents the optimal control policy for a particular problem, the neural network is trained with multiple initial conditions. This chapter will discuss the fundamentals of the four main components of the method, which are neural networks, nonlinear optimization, automatic differentiation, and training with ensembles of initial conditions.

3.1 Neural Networks

Neural networks are used as general nonlinear function approximators. The adjustable structure of neural networks allows them to be able to represent different functions, which is useful for optimization problems. For those problems, neural networks can describe the solutions to different types of optimization problems based on the inputs to the neural network. In this thesis, neural networks will be used to describe both the trajectory optimization problem $u = \phi(t; w)$ and the control policy development problem $u = \phi(x; w)$, where u represents the output of the neural network used to control the system being simulated, ϕ represents the neural network, t represents time, x represents the states of the system, and w represents the current weights of the neural network. The implementation of the adjoint method used for the majority of this thesis is based on the use of neural networks as controllers for the system, which is beneficial because of the flexible nature of the neural networks. Through the

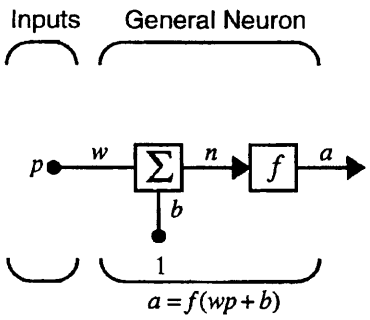

Figure 3-1: Diagram of a single neuron (from Figure 2 in [13, p. 1643]).

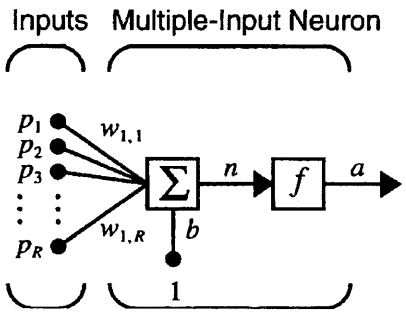

Figure 3-2: Diagram of a single neuron with multiple inputs (from Figure 4 in [13, p. 1643]).

use of neural networks in conjunction with the adjoint method, new control policies can be developed for a variety of systems, especially for systems in which it is difficult or impossible to find a closed-form solution for the optimal control policy. It should be noted that other types of controllers can be used with the adjoint method beside the neural network. The first example problem presented in Chapter 5 involves an LQR controller, which has a fixed structure. Nevertheless, because neural networks are the dominant controllers used for this thesis, the current section contains a basic overview of neural networks and how they are used in the adjoint method as the control policies that are developed. A basic reference for neural networks and the equations found in this section is the article by Hagan and Demuth [13].

The most basic component of a neural net is a neuron. As seen in Figure 3-1, an input signal p is received by the neuron. The signal is then multiplied by a weight w and added to a bias b. The resulting sum is passed through an activation function f, which scales and bounds the signal. The scaling limits are determined by the choice of activation function. Commonly used functions include piece-wise linear functions, hyperbolic tangent functions, and log-sigmoid functions. An example of a log-sigmoid function is depicted in Figure 3-4. The processed signal a from the activation function is the output signal. Therefore, the equation for the output signal from a neural network is

$$a = f(wp + b). \tag{3.1}$$

Neurons may have more than one input. Each input will have its own weight assigned to it as depicted in Figure 3-2. The set of weights for neurons W can be written in matrix form with the row number indicating the neuron to which an input is connecting and with the column number indicating the neuron from which the input is emanating. Therefore, a neuron with multiple inputs and a single output would be represented by the equation

$$a = f\left(w^T p + b\right) \tag{3.2}$$

where w is the column vector of weights for the neurons, $w_{i,j}$ is the weight for the ith node and the jth input, p is the column vector of inputs to the neuron, and b is the scalar bias parameter.

Further, neurons can be grouped into subsets of a given neural network, as can be seen in Figure 3-3, in which a circle represents an entire neuron. These subsets are known as layers. The input layer consists solely of the inputs to the neural network. The last layer of the neural network where the final output signals are generated is the output layer. If a layer does not process the input nor output signals of the neural net, the layer is known as a hidden layer. There is a limit to the computational and modeling power of a single-layer neuron, as noted by Minsky and Papert, cited in [14, p. 40]. Through the addition of hidden layers, a neural network is able to capture and to model more complex behavior in systems [14, p. 21-22].

There are many architectures for neural nets created through the various combinations of connections between individual neurons and between layers. One architecture is known as a feedforward network, in which every neuron in a given layer is connected to every neuron in the preceding and following layers; however, there is no intraconnection between neurons within a single layer, which means that the neural network is acyclic. A diagram of this type of neural network with two hidden layers can be seen in Figure 3-3. The input signals are always being processed forward through the network, hence the name "feedforward network" for this particular architecture. This type of neural network is also referred to as a multilayer feedforward network or a multilayer perceptron [14, p. 156].

There are several major choices included in the design of the structure of neural networks. One of the choices is the number of neurons and layers in a given neural network. A balance must be achieved in having a sufficient number of neurons to perform a task. If a neural network does not have enough neurons, it may not be able to perform the assigned task. However, the presence of too many neurons in a neural network will slow down the computational speed of the program [15, p. 85]. Difficulties also can arise in the generalization of a neural network when too many neurons make up the neural network since the neural network tends more toward memorization of a specific control policy for a given set of data rather than toward a general control policy that will apply to data sets that have not been used in training simulations [15, p. 84]. Another choice in the structure of a neural network is the activation function used in each neuron. One common choice for the activation function is the log-sigmoid transfer function shown in Figure 3-4, which is represented

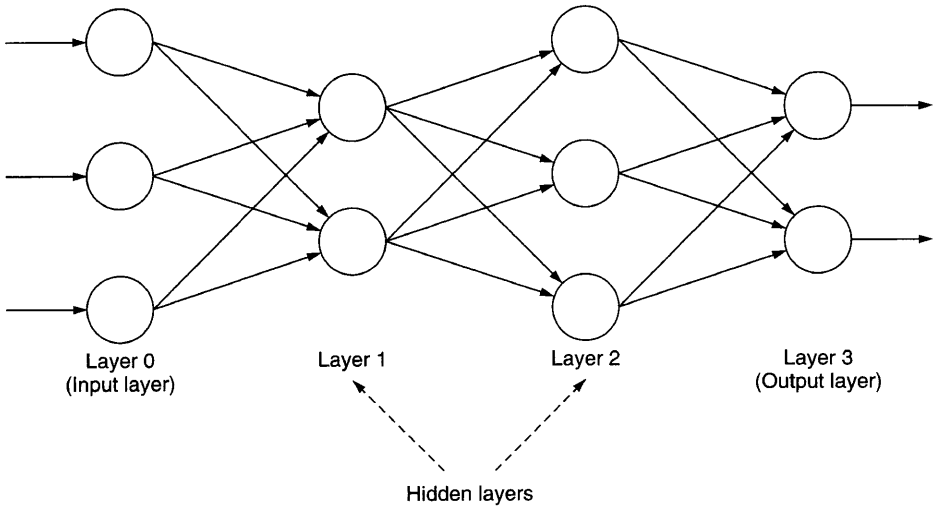

Figure 3-3: Diagram of a feedforward network (from Figure 1.15 in [15, p. 20]).

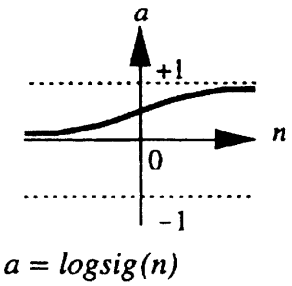

Figure 3-4: Plot of the log-sigmoid function, which is commonly used as an activation function in a neuron (from Figure 3 in [13, p. 1643]).

by the equation

$$a = \frac{1}{1 + e^{-n}} \tag{3.3}$$

where a is the output of the activation function, and n is the sum of the bias and the product of the inputs to the neuron and their corresponding weights as depicted in Figure 3-2 [13, p. 1643]. Other options for the activation function include the arctangent function, the hyperbolic tangent function, and the step function [15, p. 11-14]. The activation function does not have to be the same for all of the neurons in a given neural network, nor in a single layer.

3.2 Nonlinear Optimization

Nonlinear optimization involves the search for the minimum of a function. For the adjoint method, the function being minimized is the cost function with respect to the

weights in the neural network. Many methods for finding the minimum of a function through nonlinear optimization exist [8][22]. A Quasi-Newton method was chosen to be used for the nonlinear optimization of the cost function based on the availability of the exact derivatives of the cost function through automatic differentiation and their use in accurately computing the second derivative matrix, or Hessian, for updating the optimization routine. Additionally, the optimization method used for the problems in this thesis is an unconstrained optimization, meaning that no constraints are placed on the parameters of the function being minimized. In the case of the adjoint method, where the cost function is being minimized with respect to the weights of the neural network, the use of an unconstrained optimization method means that there are no explicit constraints placed on the values of the weights. For a further discussion of nonlinear optimization beyond what is covered in the following section, two good references are chapter 1 of the text by Betts [8] and the MATLAB Optimization Toolbox User's Guide [23, p. 3-1–3-17].

In a Quasi-Newton method, a second-order model of the function being minimized, $F(x)$, is created using the information from the gradient and the updated Hessian. The equation for the second-order model is derived from the first three terms of the Taylor series expansion about a point x and has the form

$$F(\bar{x}) = F(x) + g^T(x)(\bar{x} - x) + \frac{1}{2}(\bar{x} - x)^T H(x)(\bar{x} - x) \tag{3.4}$$

where x is the current point, $\bar{x}$ is the estimate of the new minimum from the line search, $g(x)$ is the gradient vector, and $H(x)$ is the Hessian matrix. It should be noted that the use of g and H in this section is not to be confused with their use as the integrand of the cost function and the Hamiltonian, respectively, in other sections of this thesis. The meaning of these variables should be clear for each equation. The gradient vector is defined as

$$g(x) \equiv \nabla_x F = \begin{bmatrix} \frac{\partial F}{\partial x_1} \\ \vdots \\ \frac{\partial F}{\partial x_n} \end{bmatrix} \tag{3.5}$$

and consequently, the Hessian matrix has the form

$$H(x) \equiv \nabla_x^2 F = \begin{bmatrix} \frac{\partial^2 F}{\partial x_1^2} & \frac{\partial^2 F}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 F}{\partial x_1 \partial x_n} \\ \frac{\partial^2 F}{\partial x_2 \partial x_1} & \frac{\partial^2 F}{\partial x_2^2} & \cdots & \frac{\partial^2 F}{\partial x_2 \partial x_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 F}{\partial x_n \partial x_1} & \frac{\partial^2 F}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 F}{\partial x_n^2} \end{bmatrix}. \tag{3.6}$$

For the Quasi-Newton method to have reached a strong local minimum, x^*, two necessary conditions must be satisfied. The first condition is that the gradient vector $g(x)$ at the minimum must equal 0, and the second condition is that the Hessian matrix must be positive definite. In terms of equations, the necessary conditions are

that

$$g(x^*) = 0 \tag{3.7}$$

$$p^T H^* p > 0 \tag{3.8}$$

where x^* is the local minimum, p is the search direction, where $p = \bar{x} - x$, and H^* is the Hessian matrix evaluated at the local minimum [8, p. 8-9].

3.2.1 Hessian Update Method

Quasi-Newton methods determine the Hessian matrix recursively by updating it with information from previous iterations after each optimization iteration. In the current implementation, the Hessian matrix is updated using the BFGS update because the update method is a well-accepted method for Quasi-Newton algorithms. The update method is named for Broyden, Fletcher, Goldfarb, and Shanno, who each developed this update method [8, p. 11]. The BFGS update is performed using the equation

$$H_{k+1} = H_k + \frac{\Delta g (\Delta g)^T}{(\Delta g)^T \Delta x} - \frac{H_k \Delta x (\Delta x)^T H_k}{(\Delta x)^T H_k \Delta x} \tag{3.9}$$

where H_k is the current Hessian matrix, H_{k+1} is the updated Hessian matrix, $\Delta g = g(x_{k+1}) - g(x_k)$, and $\Delta x = x_{k+1} - x_k$. If the condition that $(\Delta x)^T \Delta g > 0$ is enforced, the updated Hessian matrix will remain a positive definite matrix, which is a necessary property in the search for a minimum [8, p. 11].

3.2.2 Line Search Method

A line search to find the next potential minimum point is performed after the second-order model has been updated along with the gradient vector and the Hessian matrix. Because the derivatives are able to be calculated precisely, a cubic polynomial line search method is used by MATLAB. The next potential minimum point is calculated using the equation

$$x_{k+1} = x_k + \alpha_k d \tag{3.10}$$

where α_k is the current distance to the minimum, or step length, and d is the descent direction, where $d = -H_k^{-1} g(x_k)$.

The value of α_k for the initial evaluation of Equation 3.10 is set to unity. If the condition that $F(x_{k+1}) < F(x_k)$ is violated by the use of this α_k, the value of α_k is adjusted using the cubic polynomial line search procedure until the conditions that $F(x_{k+1}) < F(x_k)$ and that $(\Delta g)^T \alpha_k d > 0$ to ensure that the function being minimized decreases in magnitude are fulfilled. Then, the updates to the parameters in the second-order model being developed with the Quasi-Newton method can be performed for the new potential minimum point [23, p. 3-6–3-17].

3.3 Automatic Differentiation and Adjoints

In order to find the gradients necessary to perform the nonlinear optimization on the weights in the neural network, automatic differentiation using the reverse mode is used to differentiate the programs being used in the simulations. Automatic differentiation, which is also known as algorithmic or computational differentiation, does not incur truncation errors, unlike numerical differentiation [4, p. 2]. The code to calculate the gradients generated by automatic differentiation does not significantly increase the required computational time compared to the time required to run a finite differencing routine, even for a large number of gradients. The lack of truncation errors and the minimal increase in computational time makes this method particularly useful in the optimization process. One method of numerical differentiation is known as a forward difference uses the current value of $f(x)$, as well as the value of the function at the point $x + h$ through the equation

$$D_{+h}f(x) = \frac{f(x+h) - f(x)}{h}. \tag{3.11}$$

If the value of h is too large, truncation errors can affect the accuracy of the numerically calculated derivative [4, p. 2]. There are other methods of calculating derivatives numerically, but all involve truncation errors. However, through the process of automatic differentiation, a program is developed by applying the chain rule to the original code that calculates the desired derivatives. The developed program is not the symbolic derivative for the function; instead, it gives the numerical steps for calculating the derivatives [4, p. 3]. These numerical steps eliminate the truncation errors. As a result, the use of automatic differentiation to calculate derivatives for use in a nonlinear optimization method is desirable, particularly because a large error in the value of the derivative could lead to errors in the ability of the optimization method to find the location of the minima.

The two different methods for automatic differentiation are the forward mode and the reverse mode, both of which are variations of chain-rule differentiation [4, p. 4, 6-9]. In the forward mode, the output variable or variables are differentiated by one of the input variables at a time. The code is differentiated line by line sequentially though the code, which is the reason that this method is known as the forward mode [4, p. 6-8]. In the reverse mode, the output variable is differentiated by each of the input variables through the use of the chain rule. However, the derivative for each line of code is not calculated right after the calculation of each line of code as in the forward method. To calculate derivatives using the reverse mode, a forward sweep of the code occurs during which the original code is evaluated. Once the forward sweep is finished, the reverse sweep begins as the chain rule is applied starting from the output variable and working backwards through the code to the input variables [4, p. 8-9, 49]. The derivatives of the output variable with respect to the input variables are calculated in one iteration of the code generated by the automatic differentiator. The resulting gradients can then be used in the nonlinear optimization method to search for the weights of the neural network that give the minimum cost. The fact

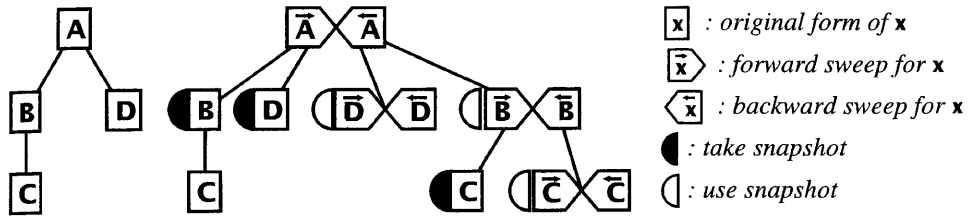

Figure 3-5: Procedural flow of TAPENADE that applies reverse differentiation to an example program (from Figure 1 in [24, p. 137]).

that all of the derivatives of the cost function with respect to the neural network weights can be computed in one run of the reverse differentiated code without a large increase in computational time compared with the forward mode makes the reverse mode more useful than the forward mode in which separate iterations of the forward differentiated code would be necessary to find the same gradients. It should be noted that the gradients are regularly used in the training of neural networks through the use of the backpropogation algorithm, which is the application of the adjoint method only to the neural network [13, p. 1646-1648].

3.3.1 TAPENADE

In order to implement the automatic differentiation, the program called TAPENADE was used to differentiate FORTRAN code that modeled the system. TAPENADE was created by the TROPICS team at INRIA Sophia-Antipolis [10]. The program has the capability for differentiating code using both the forward and the reverse modes. Figure 3-5 depicts the flow of the differentiation program through a program containing a main program A, two subprograms B and D, and a subprogram C to subprogram B. During the forward sweeps through the program, snapshots of the programs are taken, meaning that intermediate variables whose values may be rewritten when the reverse sweep is executed are saved. By saving the necessary intermediate variables, it is ensured that the correct values of the intermediate variables will be used for the desired derivatives. One of the advantages of using TAPENADE, especially with a system with complex dynamics, is that the differentiation is performed by the computer. Attempting to hand-differentiate complex models would be time-consuming and difficult; additionally, if any changes to the FORTRAN code representing the cost function, neural network, or system dynamics were made, the code would have to be redifferentiated manually. It is possible that errors could exist in the differentiated code generated by TAPENADE; however, the benefit of having codes differentiated rapidly by a computer outweighs the rare occurrence of an error in the differentiated code.

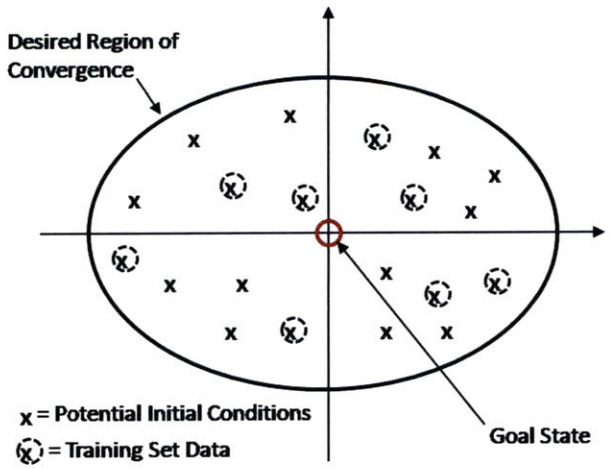

Figure 3-6: Training set data within a desired region of convergence in a notional state space. Optimal trajectories or control policies will be determined using the training data set as initial conditions. If the policy is generalizable, the trajectories or control policies defined by the trained neural network will be optimal or near-optimal for other initial conditions within the desired region of convergence.

3.4 Training with Ensembles of Initial Conditions

A desired characteristic of the found control policies is generalizability, which means that the neural network can correctly handle nearby initial conditions that were not included with the training data. Many systems have large state spaces, and it is not possible to train over every condition in the state space. However, a region of convergence that includes a subset of conditions from the entire state space can be defined for a system, and training data sets can be chosen from within this region of convergence as shown in Figure 3-6. For this implementation of the adjoint method, the training set data consists of initial trajectory conditions, and the neural network will be trained from those initial conditions to find trajectories that reach the goal state. If the neural network is generalizable, the system should be able to start from an initial trajectory condition that was not in the training data set and reach the final goal state. However, it should be noted that there is no guarantee of generalizability for a given neural network when it is being trained using the adjoint method.

One concern in neural networks is overtraining, which occurs when the neural network is able to succeed with the training data but then not to work well for values outside of the training set. When the neural network has been overtrained, the control polices are no longer robust, which is an undesirable trait. There is not a definitive rule for the number and sampling of initial conditions in the training data to ensure generalizability. One suggested rule of thumb is to train a neural network with at least five to ten times the number of elements as there are weights in the neural network [15, p. 86].

It may not always be practical to train the neural network with the number of suggested initial conditions in the training data because of the computational cost

of running a simulation with a large number of initial conditions. One possibility to achieve generalizability and robustness in the neural network without sacrificing the number of initial conditions to be used in the training data is to optimize the weights in the neural network for a portion of the training data for a set period of time. Before the neural network has had the opportunity to become overtrained on the set of data, the optimization routine is paused, and a new portion of the training data is entered as the new initial conditions. Using the final weights from the previous optimization run, the optimization routine is restarted with the final weights and the new initial condition set. This training method would allow a larger portion of the state space to be covered during training and possibly would aid in the prevention of overtraining. This method of neural network training is applied in the orbit-raising problem in Section 5.3.

In using multiple initial conditions to train the neural network, each initial condition is associated with a particular cost, and the question of expressing the final cost over the training data set to be optimized for each iteration arises. For this implementation, a simulation of the adjointed code was run for each initial condition, which produced a cost and cost gradient with respect to the weights for each initial condition. The final cost and the final cost gradient that would be used in the optimization were found by summing the individual costs and cost gradients, respectively. While more complex combination algorithms, such as weighted averages, could have been used, the straightforward summation method of the costs and the cost gradients for all of the initial conditions for a particular iteration appeared to allow the optimization function to perform its task.

Chapter 4

Software Implementation

This chapter discusses the implementation of the adjoint method described in Chapter 3. First, a description of the flow of the program written to implement the method will be discussed, followed by a discussion of the components of the code that make up the program described in the first section. Finally, the general considerations for the structure of the neural network used in this method will be mentioned.

4.1 Code Organization

The code for the adjoint method was designed to be very modular. Hence, changes only need to be made to a limited portion of the code in order to run a simulation for a different set of dynamics, cost function, or neural network. The adaptability of the code for the adjoint method makes the method ideal for attempting to determine control policies for systems. The code discussed in this section can be found in Appendices A and B.2 for the working example of double integrator problem, which will be presented in Chapter 5.

The unifying code for all of the modules written to execute the adjoint method is the top-level optimization code. The unconstrained nonlinear optimization MATLAB routine `fminunc` directs the optimization process, using the given initial conditions and optimization settings. A single initial condition vector or an array of initial condition vectors can be given as the input for the initial states in the optimization routine. The initial weights were randomly chosen using the `randn()` command in MATLAB and were scaled as needed for each example problem. Depending on the problem, smaller or larger weights than the values generated by the `randn()` command were necessary to get the optimizer to converge to a solution. The unifying code also contains a function that compiles all of the modules in FORTRAN necessary to run the adjoint method and the function that can be used to execute a forward run of the simulation after the optimizer has found the ideal weights for the particular problem.

The function that compiles all of the necessary modules in FORTRAN to run the adjoint method is known as `MakeMexFunctionsWin3`. One of the tasks that the code performs is to generate the code `nn` that represents the neural network for the problem given the desired structure of the neural network as defined in the file `parameters.m`.

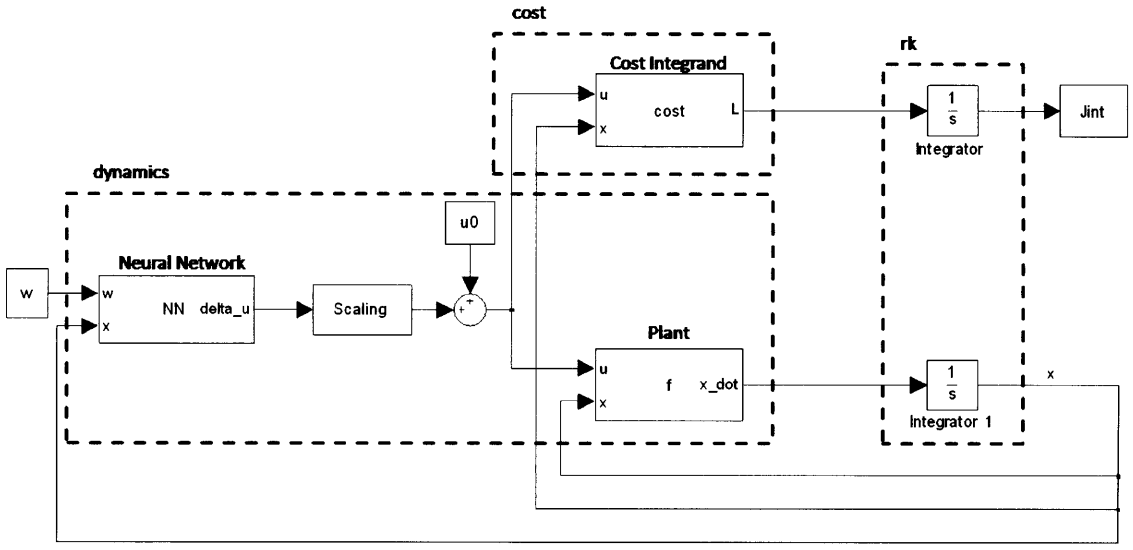

Figure 4-1: Block diagram of the forward integration process to simulate the system. Note that the variables in the diagram have the same values as previously defined, except for the undefined variables L, which is the integrand of the cost function, $Jint$, which is the integrated value of the cost, and $delta\_u$, which is the difference between the total control input u and the trim control $u0$. Also, w is constant for the forward integration calculations.

After the code defining the neural network for the specific problem has been created, all of the codes necessary to simulate the system are collated into one FORTRAN file, which is then differentiated by TAPENADE. Finally, the `mex` command is applied to translate the code to simulate the system and the adjointed code into a format that MATLAB can understand through the used of gateway functions. The purpose of gateway functions is to define the connection between the inputs and outputs of the FORTRAN functions and the variables in the MATLAB workspace.

There are four types of FORTRAN codes necessary to simulate the system. The four types include a main file, the Runge-Kutta integration code, the cost functions, and the dynamics code. The dynamics code named `dynamics` contains the equations of motion for the system, $\dot{x} = f(x, u)$, as well as a function call to the `nn` code to determine the control inputs u using the neural network. The dynamics code is called by the Runge-Kutta integration code `rk` to execute a fourth-order Runge-Kutta integration routine to find the location of the next state as well as the current cost. The cost function is constructed from three separate functions. The first function, `cost`, calculates the integrand of the cost function at the given state and control input. This function is the cost function that is called by the Runge-Kutta integration code. The two other cost function codes are `terminal` and `general` and are used to calculate the cost at the final time and a general cost that does not fit into the other two cost function definitions, respectively. The three codes combined express the cost function for the system as discussed in Section 2.1. Finally, the `main` file is used to keep track of the integration progress in the states, control inputs, and integrated

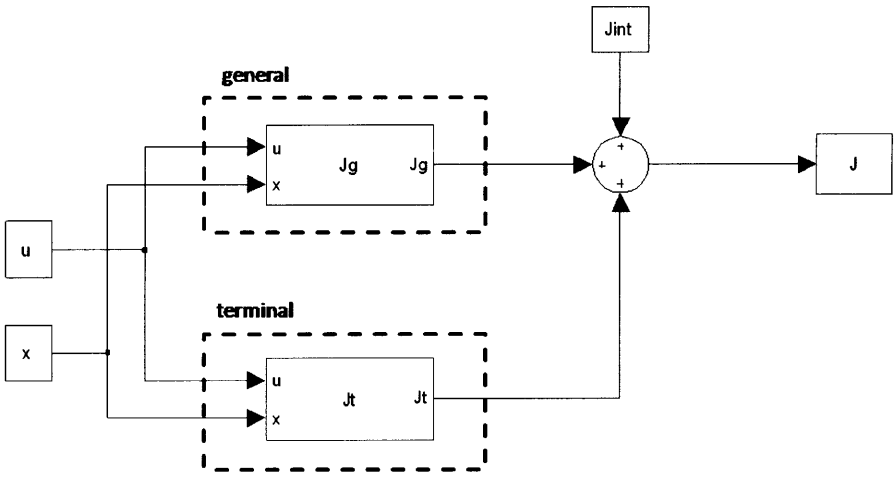

Figure 4-2: Block diagram of the post-integration cost calculation. Note that the values of x and u in the diagram are the time histories of the states and the control. Jt represents the terminal cost, and Jg represents the general cost, which is used for penalties on the system that do not fit into the integrated cost calculation or the computation of the cost at the final time. The two costs are summed along with the total integrated cost $Jint$ from Figure 4-1 to find the total cost J for a given optimization iteration.

cost as shown in Figure 4-1 and to calculate the total cost function by combining the integrated cost, the terminal cost, and the general cost as shown in Figure 4-2. The four codes described in this paragraph are combined together to simulate the system in the forward sense, meaning that integration is used to find the state and control input history as well as the final cost for the system. TAPENADE differentiates this entire forward code in order to construct the adjoint code, which runs in the reverse sense, meaning that starting with the final cost, differentiation is used to calculate the derivatives of the cost with respect to the weights of the neural network.

From the above description of the codes used in the optimization routine, it can be seen that each code has a specific function. The only codes that must be modified by the user in order to set up a problem for a given system configuration are the `dynamics` code to describe the equations of motion of the system, the `parameters.m` file to define the structure of the neural network, the three cost function codes to represent the cost function for the system, and the optimization file to define the initial conditions for the system. Therefore, the modularity of the system is advantageous to a user who may want to run several different system configurations without having to rewrite or add too much code. However, it should be noted that any changes to the dynamics or cost function codes as well as the `parameters.m` file will require the `MakeMexFunctionsWin3` function to be rerun in order to translate the modified FORTRAN code into a format that MATLAB can understand.

4.2 Design Choices for Neural Network Structure

In Section 3.1, several design considerations were mentioned for neural networks, including the choice of activation function for each neuron, the type of neural network to be used, and the structure of the layers of the neural network. This section describes the choices made for those considerations for the simulations run as a part of this thesis and possible limitations of the program used to generate the neural network.

One limitation of the neural network generation module of the program for the adjoint method was that the structure of the network had to be a feedforward network. However, this structure was the structure that had been decided upon to use in testing of the adjoint method for control policy development; therefore, for the purposes of this thesis research, the confinement of the neural network to one specific architecture was not a limitation. Another limitation of the neural network generation module was that all of the neurons in a single layer had to have the same transfer function. Nevertheless, each layer could have a different transfer function for its set of neurons. The ability to change transfer functions between layers should provide sufficient flexibility in the neural network for the purposes of determining control policies.

The arctangent function was chosen as the activation function for all of the neurons based on previous work using this method [19]. The limits of the function were $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$; therefore, the output signals from the neural net had to be scaled to the appropriate magnitudes for use in the controller. One of the benefits of using the arctangent function as the activation function for the neurons was that the limits of the arctangent function automatically bounded the control inputs. Consequently, for bounded control input problems, the limits on the control input could be easily set by scaling the output of the neural network appropriately.

Chapter 5

Example Applications

In this chapter, the validity of the adjoint method for developing closed-loop controllers is demonstrated through three example applications. The applications considered are the development of a linear quadratic regulator to maintain the altitude of an aircraft, and the definition of optimal trajectories and control policies for a double integrator system and for an orbit-raising problem. The latter two problems included the use of the neural network to calculate the control inputs for the system. Through these three problems, the ability to apply the adjoint method to multi-state systems and to different control structures is demonstrated. Each of the following sections contains a description of the problem, a discussion of the results from the adjoint method, and a comparison with optimal results obtained using other methods.

5.1 Linearized Aircraft Dynamics Problem

The first problem to be examined using the adjoint method was the problem of designing an altitude-hold controller for a Boeing 747 aircraft. The purpose of applying the adjoint method to the linearized aircraft dynamics problem was to check the ability of the adjoint method to correctly determine the weights for a fixed-gain LQR controller, which has a specific control structure. The LQR gains could be calculated in MATLAB using the `lqr` command, providing a baseline against which the performance of the adjoint controller in finding the correct LQR gains could be checked.

5.1.1 Description

The linearized aircraft dynamics problem is based on the example of control design for the longitudinal control of a Boeing 747 aircraft, which is found in [25, p. 743-761]. Figure 5-1 contains a diagram of the aircraft coordinates that are used in the problem. In order to find the equations of motion necessary to design an altitude-hold controller, the general aircraft equations of motion are linearized about the steady-state horizontal flight condition [25, p. 744] to find the linearized longitudinal equations of motion. These equations of motion are expressed in the form found in Equation (2.10), which is necessary for the design of LQR controllers, as explained in

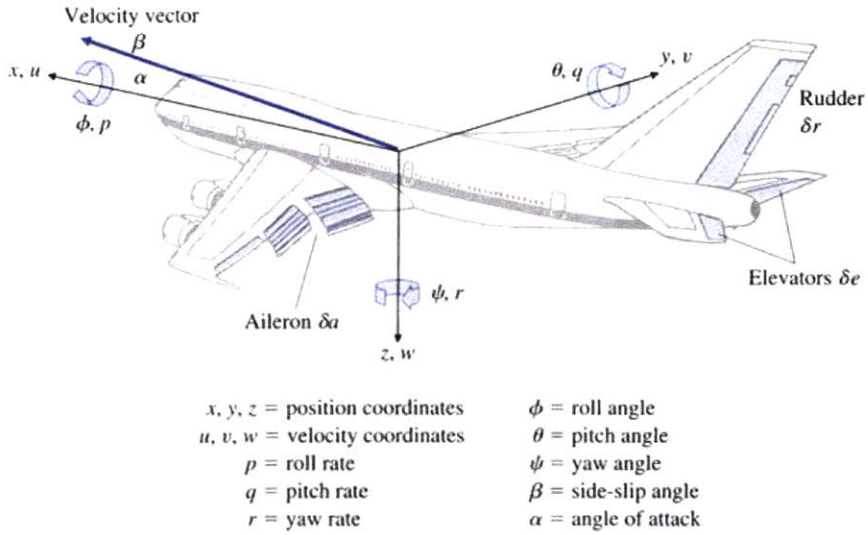

Figure 5-1: Diagram of aircraft coordinates for the linearized aircraft dynamics problem (from Figure 10.30 in [25, p. 743]).

Section 2.2. Also, a state defining the altitude is added to the list of states associated with the longitudinal equations of motion in order to regulate the altitude changes of the aircraft. Considering the level flight condition of with a horizontal speed of $U_o =$ [820] ft/sec at 20,000 ft, an aircraft weight of 637,000 lb, [25, p. 756] and an initial pitch angle of approximately 8.9 degrees for this problem, the resulting equations of motion after linearization and the addition of the altitude state are

$$\begin{bmatrix} \dot{u} \\ \dot{w} \\ \dot{q} \\ \dot{\theta} \\ \dot{h} \end{bmatrix} = \begin{bmatrix} -0.00643 & 0.0263 & 0 & -32.2 & 0 \\ -0.0941 & -0.624 & 820 & 0 & 0 \\ -0.000222 & -0.00153 & -0.668 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 830 & 0 \end{bmatrix} \begin{bmatrix} u \\ w \\ q \\ \theta \\ h \end{bmatrix} + \begin{bmatrix} 0 \\ -32.7 \\ -2.08 \\ 0 \\ 0 \end{bmatrix} \delta_e \quad (5.1)$$

where as defined in [25, p. 746], u is the forward velocity of the aircraft in the x direction, w is the velocity perturbation in the z direction, q is the angular rate about the positive y-axis, or pitch rate, θ is the pitch-angle perturbation from the reference value, h is the altitude perturbation of the aircraft, and δ_e is the perturbation of the elevator angle for pitch control.

For this problem, two LQR controllers are derived. One controller is found using the `lqr` command in MATLAB [26, p. 4-118], and the second controller is found using the adjoint method with a fixed structure controller of the LQR gains K from Equation (2.12). The rate of convergence to the LQR optimal control solution is studied for the adjoint method.

5.1.2 Results

Before the adjoint method could be applied to design an LQR controller for the Boeing 747 aircraft, a baseline LQR controller had to be designed using MATLAB. For the problem, it was decided to weight the altitude state h and the elevator angle δ_e. Therefore, the weighting matrices Q and R that define the LQR cost function in Equation (2.11) were chosen to be

$$Q = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

$$R = 1.0 \times 10^8. \tag{5.3}$$

A final penalty H was not imposed on the states for this problem. The value of R was chosen through an iterative process in which various values of R were used to find K, and then the response of the system to the gains was examined. The LQR gains K found using MATLAB for the given Q and R values to five significant digits are

$$K \simeq \begin{bmatrix} -3.0264 \times 10^{-5} & 4.7349 \times 10^{-4} & -0.17705 & -0.53762 & -1.0000 \times 10^{-4} \end{bmatrix}. \tag{5.4}$$

While the motion of the aircraft with the applied LQR gains K may not be suitable for an aircraft that is carrying passengers, the gains were determined to produce a stabilizing motion for the aircraft and to be satisfactory for use in the current problem.

Finally, the optimal cost was calculated for the initial condition that was to be used in the adjoint method simulations. The equation to calculate the optimal cost given in the notes by How [7, p. 4-7] has the form for this problem of

$$J^* = x^T P x \tag{5.5}$$

where J^* is the optimal cost for the given initial condition vector of the states x, and P is the solution to the Riccati equation. For the initial condition

$$x_o = \begin{bmatrix} 0 & 0 & 0 & 0 & 100 \end{bmatrix}^T \tag{5.6}$$

the solution to the Riccati equation found using the `lqr` command and the corresponding optimal cost to five significant digits are

$$P \simeq \begin{bmatrix} 0.43595 & -4.1011 & 1.5195 \times 10^3 & 4.6586 \times 10^3 & 0.91457 \\ -4.1011 & 64.498 & -2.3778 \times 10^4 & -7.3391 \times 10^4 & -15.184 \\ 1.5195 \times 10^3 & -2.3778 \times 10^4 & 8.8856 \times 10^6 & 2.7001 \times 10^7 & 5.0464 \times 10^3 \\ 4.6586 \times 10^3 & -7.3391 \times 10^4 & 2.7001 \times 10^7 & 8.3596 \times 10^7 & 1.7592 \times 10^4 \\ 0.91457 & -15.184 & 5.0464 \times 10^3 & 1.7592 \times 10^4 & 6.5128 \end{bmatrix} \tag{5.7}$$

$$J^* = 6.5128 \times 10^4. \tag{5.8}$$

The value of the optimal cost was used to ensure that the correct control policy was being determined using the adjoint method.

Once the baseline LQR gains K and the optimal cost J^* were defined, the adjoint method was applied given the system dynamics, cost function, and control structure. The code for the dynamics and cost expressions in this problem is listed in Appendix B.1. The optimization routine was run with the initial condition of only one altitude perturbation and with a set of randomly selected initial guesses for the five values in the vector K. It should be noted that the initial guesses for the five values in the vector K had to be scaled by 10^{-6} in order to get the optimization routine to go to completion. If the initial guesses were too large, the optimization routine would stop after a couple of iterations because of not a number (NaN) errors. It can be seen in the plots in Figure 5-2 that the optimal state trajectories as well as the control history found using the adjoint method match the state trajectories and control found using the LQR gains generated by MATLAB to the extent that the two lines present on each graph almost cannot be distinguished from each other. Therefore, the adjoint method is able to determine the LQR optimal control solution. The LQR gains K found using the adjoint method to five significant digits are

$$K \simeq \begin{bmatrix} -2.6630 \times 10^{-5} & 4.6805 \times 10^{-4} & -0.17341 & -0.53523 & -9.9770 \times 10^{-5} \end{bmatrix}. \tag{5.9}$$

Comparing the vectors K in Equations (5.4) and (5.9), it can be seen that the elements of the two vectors are very similar to each other. Also, Figure 5-3 confirms that the optimization routine used by the adjoint method converges approximately to the correct optimal cost. A closer examination of the actual final cost calculated by the adjoint method, which is equal to 6.5123×10^4, is almost the same as the optimal cost. Therefore, it can be concluded that the adjoint method is able to determine the LQR gains without knowing specifically that the problem is given in the form that suggests a LQR optimal control solution.

One aspect of the adjoint LQR solution that was examined was the rate of convergence to the LQR solution for various sets of random initial values for the LQR gains K_o. Figure 5-3 depicts the rate of convergence of the cost J to the final LQR cost J. The initial jump that the optimization routine makes during the solution process can be seen in the upper plot, as well as a secondary jump that occurs as the optimizer is searching for the gains that will minimize the cost. The lower plot shows the subsequent iterations to reach the final cost value. While the various values of initial weights appear to affect the rate of convergence after the secondary jump in the cost, all of the sets of initial weights converge to approximately the same final cost. Therefore, the optimization routine is robust enough to determine that it should continue iterating until the optimal LQR cost is reached. The number of iterations needed to converge to the final solution is nearly identical for the five sets of initial values. The first set converged in 154 iterations, while the remaining sets converged in 161 iterations.

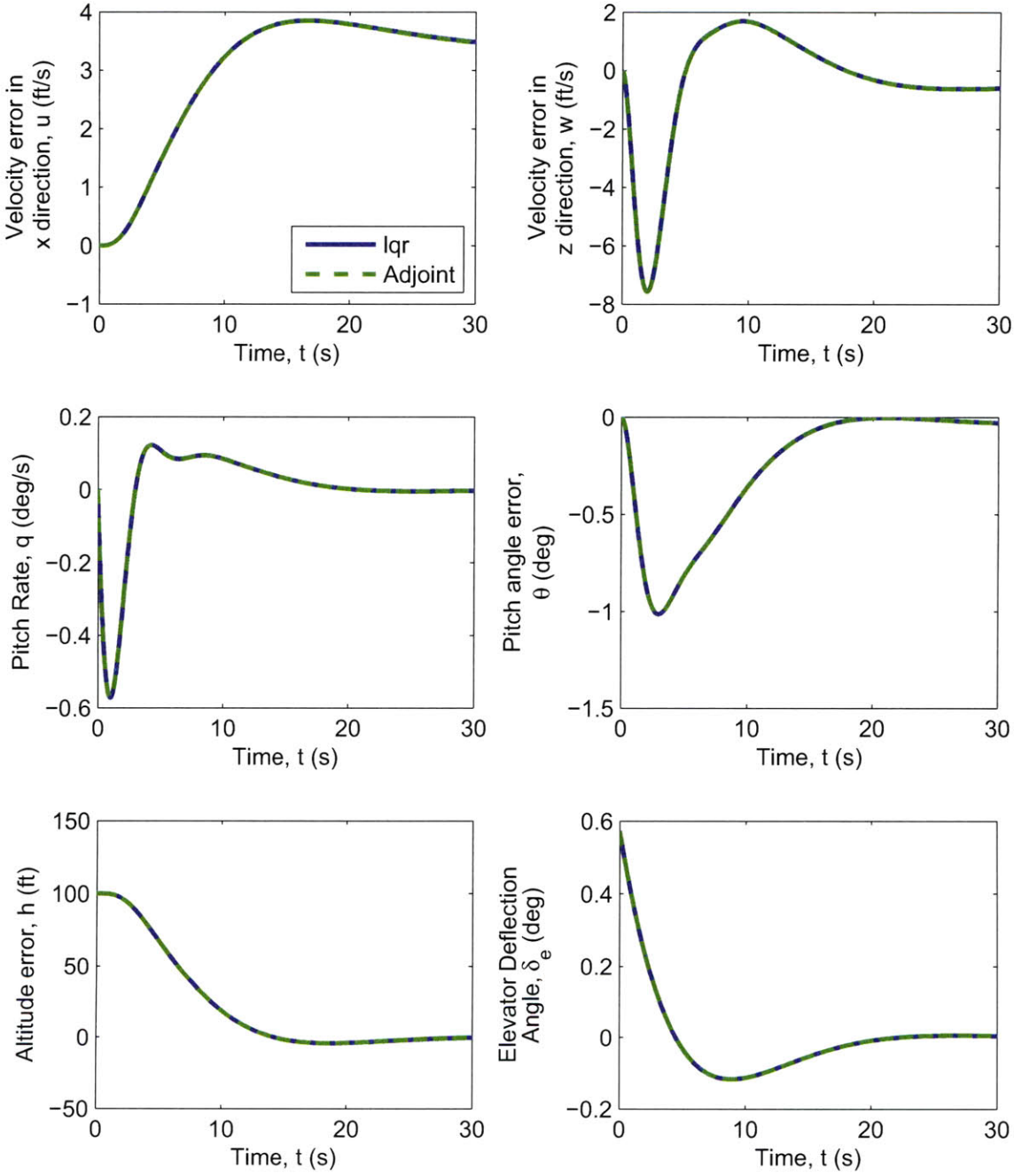

Figure 5-2: Comparison of LQR results found using the `lqr` command in MATLAB and the adjoint method. It should be noted that only one initial perturbation in altitude was used for the adjoint method simulation. The two solutions agree to the extent that the two lines representing the solution cannot be distinguished from each other.

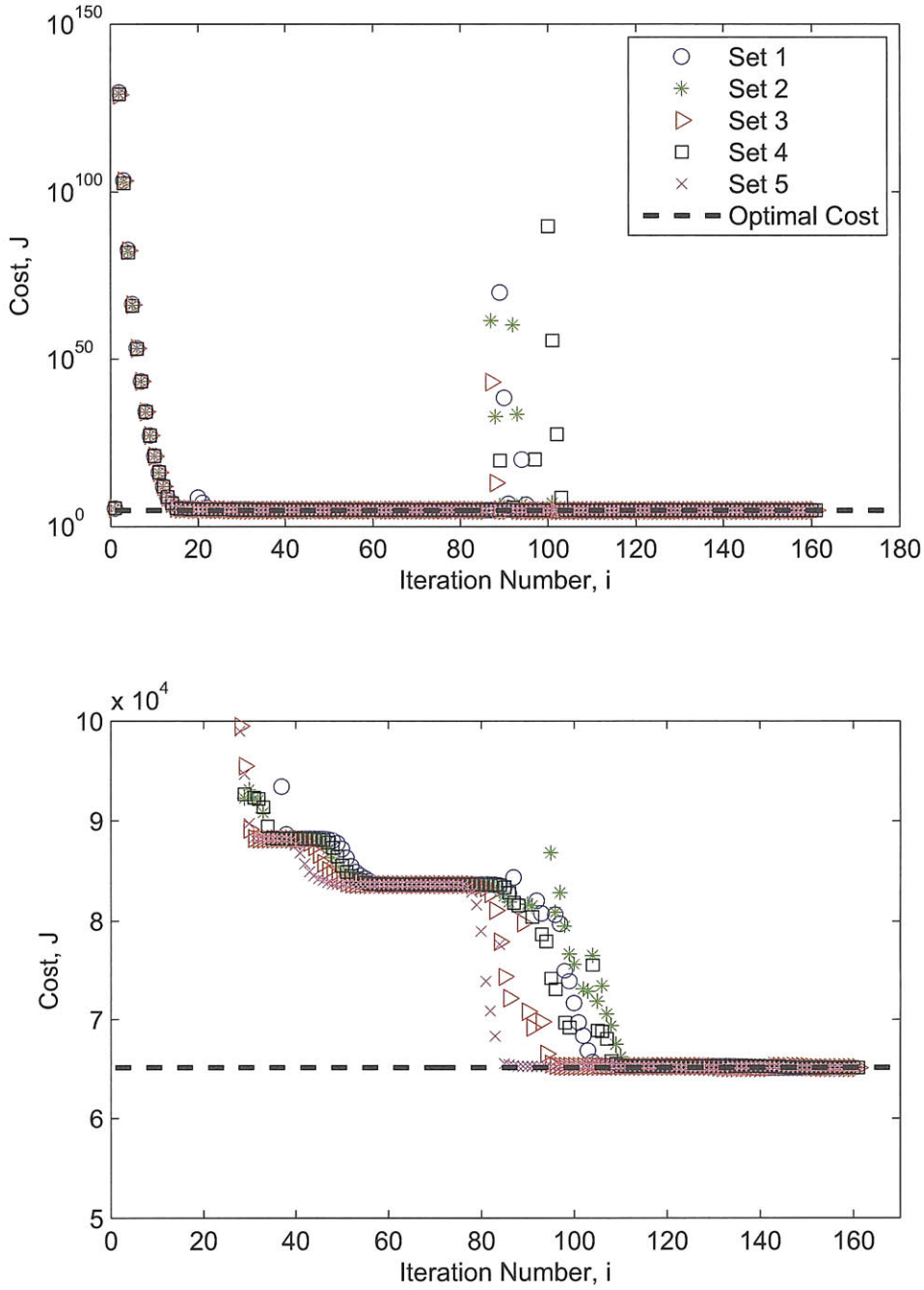

Figure 5-3: Plots of the rates of convergence to the LQR solution for a group of random initial vectors K_o using the adjoint method. The upper plot shows the entire time history of the cost, while the lower plot shows the iterations after the optimization routine has recovered from the initial jump in the cost. It can be seen that the adjoint method converges approximately to the optimal cost that was calculated.

5.2 Double Integrator Problem

The second application of the adjoint method was the double integrator problem. The problem was a good first candidate for testing the use of the neural network for the control structure with the adjoint method because of its simple dynamics and cost function. Also, a comparison with the known optimal solution was possible, which allowed the ability of adjoint method to determine optimal trajectories and control policies to be assessed.

5.2.1 Description

The double integrator problem, which may also be viewed as a simple one-dimensional thruster problem, is the second problem to be examined under the adjoint method. This problem appears in the texts by Bryson and Ho [5, p. 112-115] and by Kirk [6, p. 300-307]. The double integrator system is described by two states, which are position x and velocity v, and one control input u. The control input is bounded by -1 and 1. The equations of motion are

$$\dot{x} = v \tag{5.10}$$

$$\dot{v} = u \tag{5.11}$$

where x is the position, v is the velocity, and u is the control input, or thrust [5, p. 112].

The goal of the double integrator problem is to take the system from an arbitrary initial condition (x_o, v_o) to the origin. The problem is an infinite time horizon problem because the cost function does not include a penalty on time. Consequently, from these two conditions, the cost function that is used for this problem is a quadratic cost function that only involves the states of the system and has the form

$$J = \int_{t_o}^{t_f} \left(x^2 + v^2\right) dt \tag{5.12}$$

where J is the cost, t_o is the initial time, and t_f is the final time. It should be noted that a final time is defined for the optimization runs using the adjoint method because it is not possible to run the simulations for infinite time. However, the final time is chosen to be significantly greater than the time necessary for the position and velocity states to reach the origin, and therefore, the problem can be viewed as an infinite time horizon problem.

5.2.2 Results

For the double integrator problem, the analytic optimal control solution was derived to serve as a baseline against which the results from the adjoint method could be compared. The results section for the double integrator problem will contain an abridged version of the derivation of the analytical solution, which will be followed by

a comparison with the numerical solution from the adjoint method for the trajectory optimization and optimal control problems.

5.2.2.1 Derivation of the Analytical Optimal Control Solution

The complete derivation of the analytical optimal control solution for the double integrator problem can be found in [6, p. 300-307]. This section will provide the beginning steps of the derivation to the point that it was taken by the author of this thesis before referring to Kirk for the final steps of the derivation and will give the final analytical optimal control solution. In this derivation, the necessary conditions for optimality are determined, and the equation for the singular arc is calculated. The remaining derivation of the switching curve is presented in Kirk. It should be noted that for the derivation, a factor of $\frac{1}{2}$ was added to the cost function in Equation (5.12) in order to simplify the coefficients. The addition of this factor does not impact the final solution. Also, throughout this section, variables in equations have the same meaning as previously defined in this thesis, unless otherwise indicated.

For the double integrator problem, the first step in finding the optimal control solution is to form the Hamiltonian, which is given in Equation (2.4). Given the state dynamics and the cost function in Section 5.2.1, the equation for the Hamiltonian H is

$$H = \frac{1}{2}x^2 + \frac{1}{2}v^2 + p_1 v + p_2 u \tag{5.13}$$

where p is the costate vector defined as $p = \begin{bmatrix} p_1 & p_2 \end{bmatrix}^T$.

After forming the Hamiltonian, the necessary conditions for optimality presented in Equations (2.5), (2.6), and (2.7) are applied. The first condition in Equation (2.5) is already defined by the state equations. The second condition in Equation (2.6), which involves the derivative of the Hamiltonian with respect to the states to find the derivatives of the costates, can be written as

$$\dot{p}_1 = -x \tag{5.14}$$

$$\dot{p}_2 = -v - p_1. \tag{5.15}$$

Because the control u for the double integrator problem is bounded, the third necessary condition expressed in Equation (2.7) must be modified to become the more general condition known as Pontryagin's Minimum Principle [7, p. 9-6] in which

$$u^*(t) = \arg\min_{u(t)\in U} H(x, u, p, t) \tag{5.16}$$

where $u^*(t)$ is the optimal control, and U is the set of permissible control values.

In order to apply Pontryagin's Minimum Principle to find the optimal control within a bounded range of control values, the portion of the Hamiltonian that only depends on u is examined. For this problem, the portion of the Hamiltonian in Equation (5.13) that only depends on u, which is denoted by $\tilde{H}$, is

$$\tilde{H} = p_2 u. \tag{5.17}$$

Therefore, the optimal control that minimizes the modified Hamiltonian is

$$u^*(t) = \begin{cases} -1, & p_2 > 0 \\ 1, & p_2 < 0 \end{cases} . \tag{5.18}$$

Although it may seem that Equation (5.18) gives the complete optimal control solution for this problem, the case where $p_2 = 0$ is not defined by this control policy. Pontryagin's Minimum Principle does not give information about the optimal control for this condition, which is known as a singular arc. For the condition where $p_2 = 0$, the singular arc is defined by the fact that the derivative of the Hamiltonian with respect to the control $\frac{\partial H}{\partial u}$ will equal 0 as well as all of the time derivatives of $\frac{\partial H}{\partial u}$. By taking the derivative of the Hamiltonian in Equation (5.13) and setting it equal to 0 since the system is on a singular arc, the derivative is determined to be

$$\frac{\partial H}{\partial u} = p_2 = 0 \tag{5.19}$$

which is consistent with the missing case in Equation (5.18). In order to find the control used on the singular arc, the time derivative of $\frac{\partial H}{\partial u}$ is calculated and set equal to 0. From the analysis of this time derivative, an expression can be found for the first costate p_1 on the singular arc as follows

$$\frac{d}{dt}\frac{\partial H}{\partial u} = \dot{p}_2 = 0 \tag{5.20}$$

and substituting Equation (5.15) into Equation (5.20),

$$-v - p_1 = 0 \tag{5.21}$$

which implies that

$$p_1 = -v. \tag{5.22}$$

Because the Hamiltonian is not an explicit function of time, and because the final time is free in the problem, the Hamiltonian must be equal to 0 on the singular arc [6, p. 301]. Given that $p_2 = 0$ and $p_1 = -v$ on the singular arc, substituting these values for the costates into Equation (5.13), where $H = 0$, gives the relation between x and v on the singular arc that

$$x^2 - v^2 = 0. \tag{5.23}$$

To this point, the author's derivation agrees with the derivation of the optimal control solution given by Kirk [6, p. 300-307]. The analysis by Kirk continues further to find the possible control policies on the singular arc and to determine the complete optimal control solution for the problem. The optimal control law as defined using

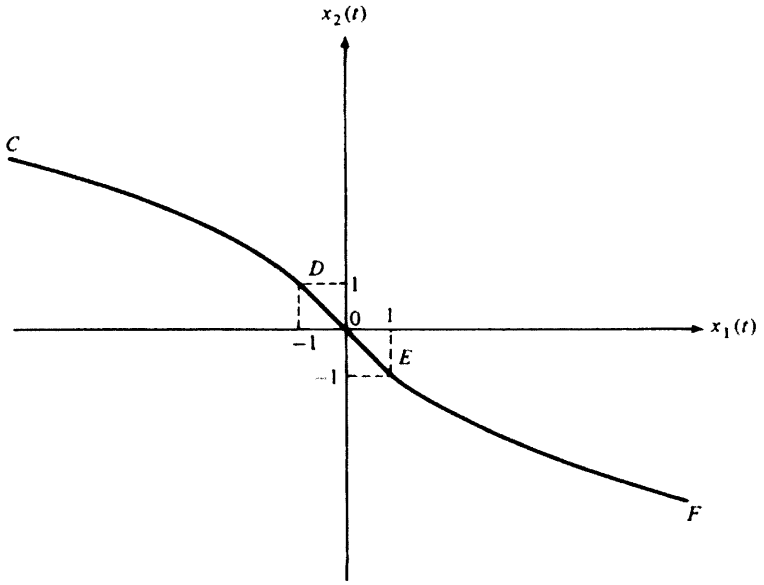

Figure 5-4: Switching curve defining the optimal control law for the double integrator problem (from Figure 5-43 in [6, p. 305]). It should be noted that the variables x_1 and x_2 in this plot are represented as x and v, respectively, in this thesis.

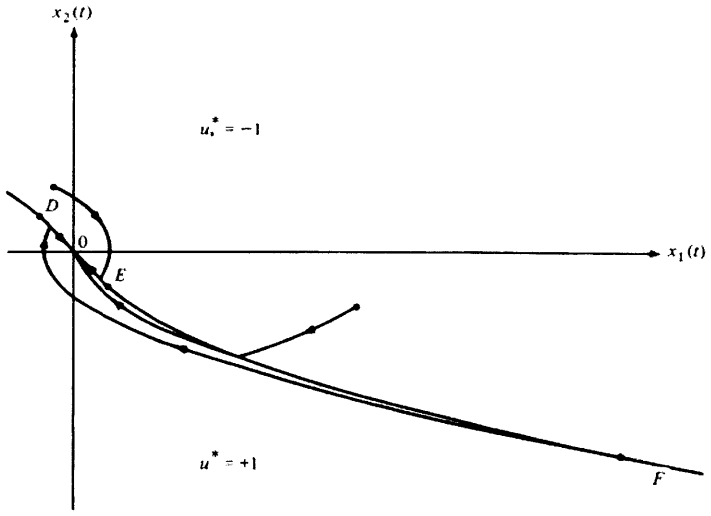

Figure 5-5: Example optimal trajectories for the double integrator problem (from Figure 5-44 in [6, p. 306]). It should be noted that the variables x_1 and x_2 in this plot are represented as x and v, respectively, in this thesis.

the switching curve labels in Figure 5-4 [6, p. 306] is

$$u^*(t) = \begin{cases} -1, & \text{for } \tilde{x}(t) \text{ to the right of C-0-F} \\ +1, & \text{for } \tilde{x}(t) \text{ to the right of C-0-F} \\ -1, & \text{for } \tilde{x}(t) \text{ on segment C-D} \\ +1, & \text{for } \tilde{x}(t) \text{ on segment E-F} \\ -v, & \text{for } \tilde{x}(t) \text{ on segment D-0-E} \end{cases} \tag{5.24}$$

where $\tilde{x}(t)$ is the entire state vector defined as $\tilde{x}(t) = \begin{bmatrix} x & v \end{bmatrix}^T$. Figure 5-5 shows example optimal trajectories. As noted in Kirk, "*the switching curve is not a trajectory* except on the singular line D-0-E" [6, p. 306]. This property of the switching curve will be important in the following results section for the double integrator problem. It should be noted that in Figures 5-4 and 5-5 from Kirk, x_1 is the same as x, and x_2 is the same as v in this thesis.

5.2.2.2 Comparison of the Analytical and Adjoint Optimal Control Solutions

In order to compare the analytical and adjoint optimal control solutions, it first was necessary to find the switching curves for the analytical solution, which would be used to graphically confirm that the adjoint control policy was determining the optimal state trajectories. A routine that integrated the state and costate equations in reverse with respect to time had to be used in order to solve the equations defining the analytical optimal control policy because the two costate equations given in Equations (5.14) and (5.15) were written with respect to the states. Given the matrix set of equations for the states and costates

$$\begin{bmatrix} \dot{x} \\ \dot{v} \\ \dot{p}_1 \\ \dot{p}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ v \\ p_1 \\ p_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} u \tag{5.25}$$

where the variables have the same definition as previously indicated in this section, and the terminal conditions that $v(t_f)$ is free within the range between -1 and 1, $x(t_f) = -v(t_f)$, $p_1(t_f) = -v(t_f)$, and $p_2(t_f) = 0$, a backwards Euler integration routine can be used to determine the switching curve by monitoring the sign of $p_2(t)$ during the integration for changes in sign throughout the integration. The terminal velocity is constrained to be between -1 and 1 because the singular arc marks the end of the use of the costate equations found from the Hamiltonian to define the switching curve. A change in sign of $p_2(t)$ indicates that the control has switched signs as described in the text by Kirk [6, p. 303], and by saving the states and control at the time when the change in sign occurs, the switching curve can be constructed for the phase plane plot. For the backward integration routine, the control u is set equal to 1 in order to generate the portion of the switching curve in the fourth quadrant and to -1 in order to generate the portion of switching curve in the second quadrant.

To define the complete switching curve, the singular arc must also be added, which is represented by the equation

$$v = -x \tag{5.26}$$

where x and v have a range of -1 to 1.

After the optimal control switching curve was established, it was possible to begin tests of the adjoint method on the double integrator problem. The code for the dynamics and cost expressions in this problem is listed in Appendix B.2. The first case to be examined was trajectory optimization, in which the time remaining was the only input into the neural network. The controller represented by the neural network had the form $u = \phi(t_r; w)$, where t_r is the time remaining. Figure 5-6 shows the results from the trajectory optimization using the adjoint method for a grid of initial conditions, where $-2.5 \leq x_o, v_o \leq 2.5$, and the same set of initial weights for all of the initial conditions. The optimization process was performed for each initial condition individually. For this case, a 3-7-3-1 feedforward neural network was used to capture the control policy. Most of the trajectories determined by the adjoint method switch control at the appropriate times as indicated by the switching curves. Several of the trajectories do not resemble the shape of the optimal trajectories. The reason for the lack of resemblance is that the optimizer is not able to complete the optimization process with the given initial weights. When different random initial weights are selected, the optimizer is able to determine trajectories that follow the optimal solution indicated by the switching curve for the initial conditions that have unoptimized trajectories with the first set of initial weights.

The second case involved the study of the feedback control law $u = \phi(x)$ problem for the cases of both one initial condition and multiple initial conditions. Equation (5.24) shows that it is possible to define an optimal control law $u = \phi(x)$ for the double integrator problem. For this set of simulations, a 3-7-5-1 feedforward neural network was used because it represented the optimal control policy well. A shallower 3-1 feedforward neural network initially was used for this set of simulations. However, as can be seen in Figure 5-7, the neural network is not able to learn the optimal control law well, instead finding many suboptimal trajectories that have a long linear final section to the origin. Consequently, additional neurons and hidden layers were added to the neural network to create the 3-7-5-1 feedforward neural network, which was able to represent the optimal control policy better than the shallower neural network could. Figure 5-8 depicts the trajectories in the phase plane given by the control policy derived using the adjoint method and a neural network that was trained for each initial condition on the plot individually. The control policy from the adjoint method trains to the general shape of the switching curve, which means that the neural network is almost able to learn the optimal control policy.

One of the goals of this thesis is to examine the training of neural networks with multiple initial conditions, which is studied for the double integrator problem. The two plots in Figure 5-9 show the trajectories determined using a neural network that was trained using 10 initial conditions in the training data set. The plots were created from two simulations that had different initial conditions and initial weights. The lower plot matches the optimal control solution more closely than the upper

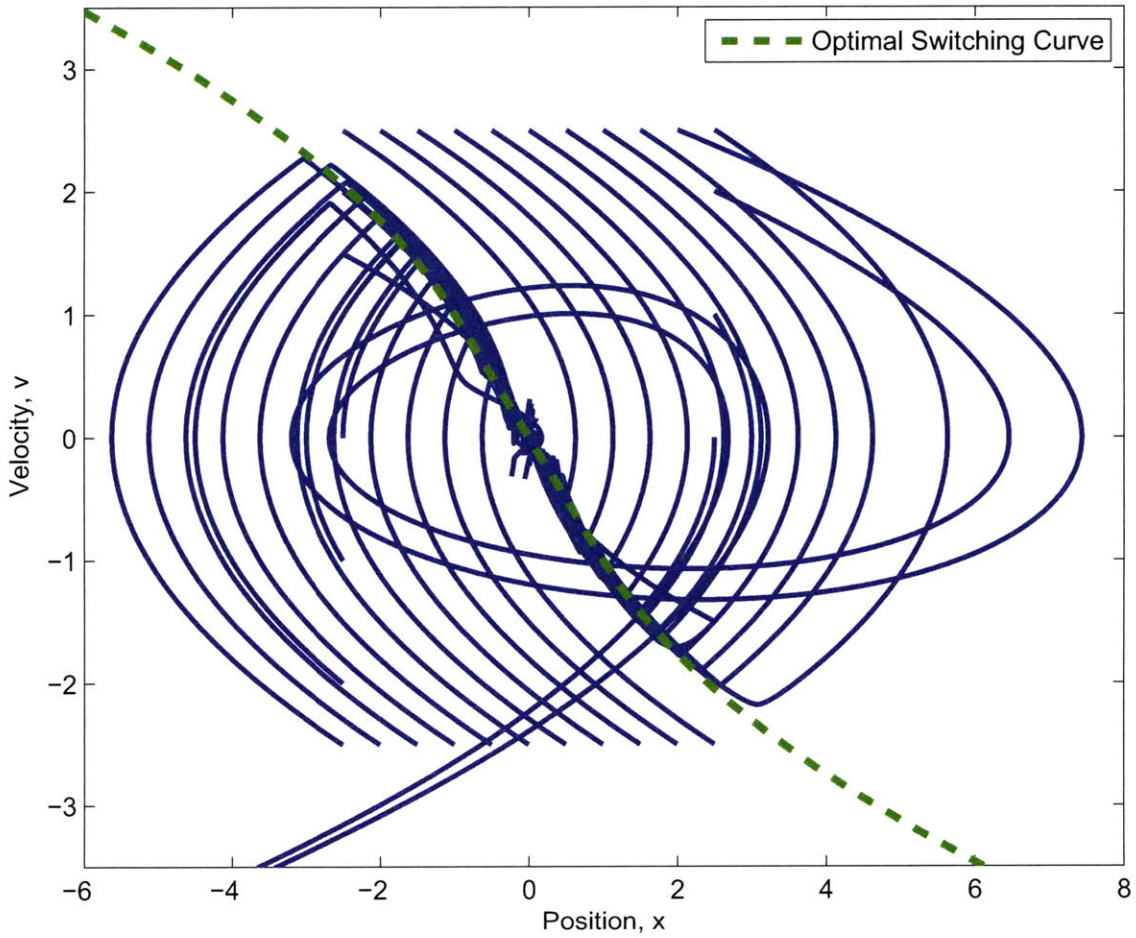

Figure 5-6: Results from the trajectory optimization of the double integrator problem using the adjoint method compared with the optimal switching curve. The curves that do not follow the switching curve or general trajectory shape represent suboptimal solutions for which the adjoint method could not complete the optimization process. Changing the initial weights for the errant initial conditions allowed the optimal solutions to be found.

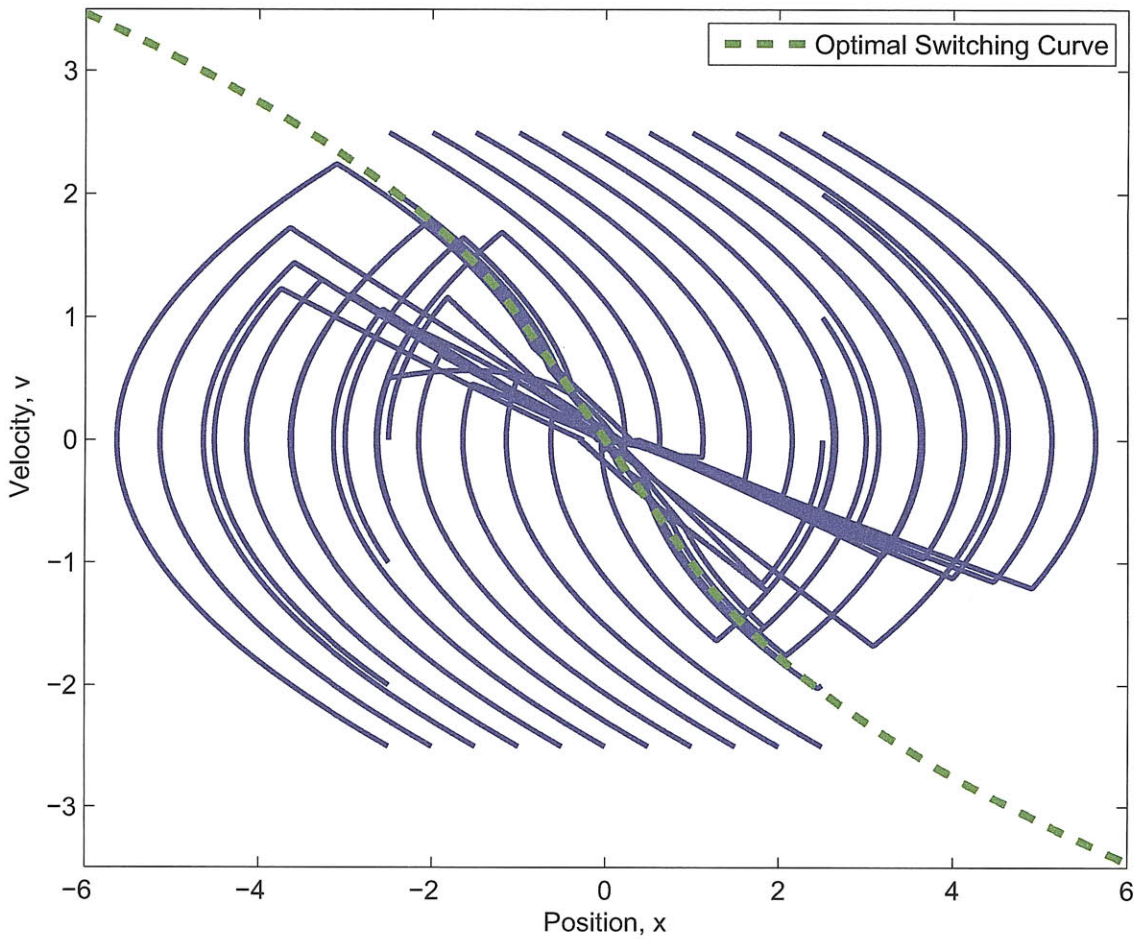

Figure 5-7: Results for the optimal control law problem using the adjoint method and a shallow 3-1 neural network trained with 1 initial condition at a time compared against the optimal switching curve. Comparing this figure with Figure 5-8, which has a 3-7-5-1 neural network, it can be seen that the shallower neural network is not able to learn the optimal control policy as well as the more complex neural network.

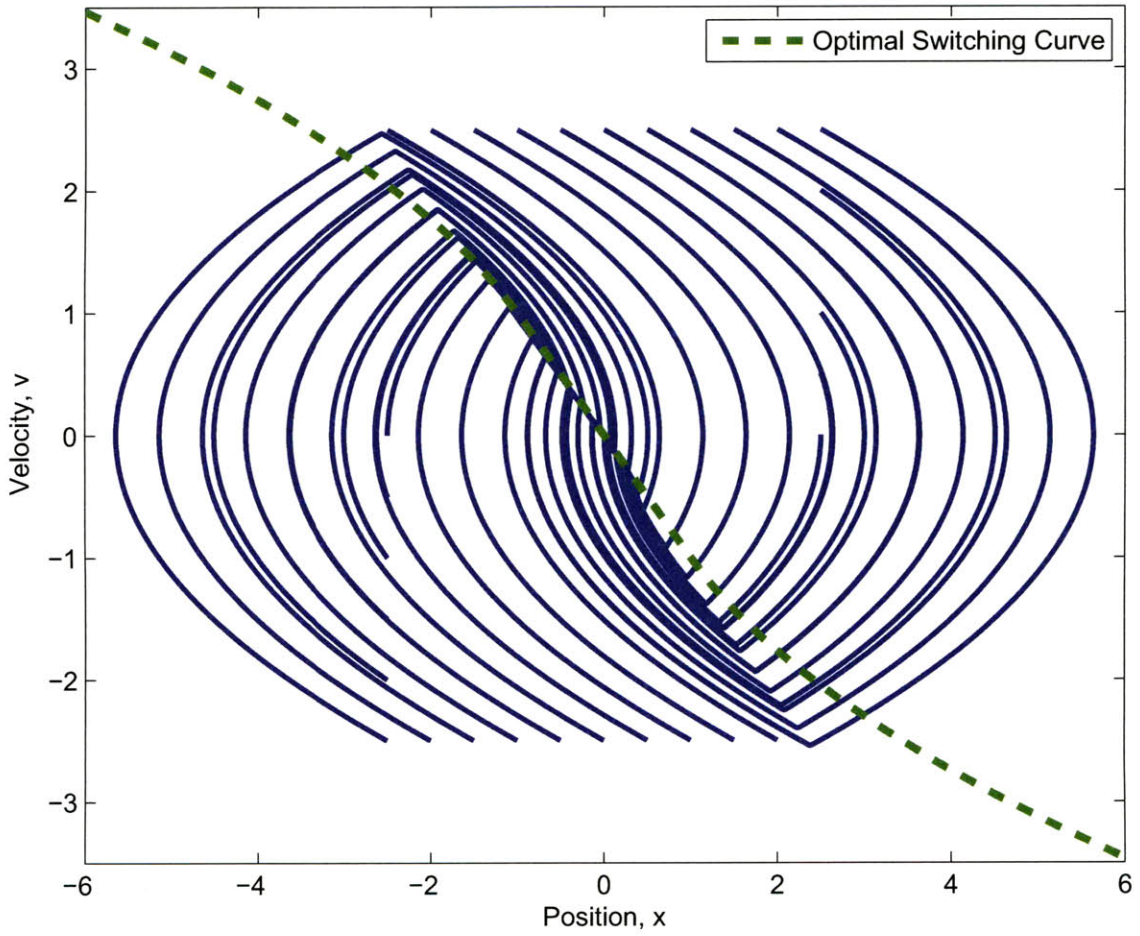

Figure 5-8: Results for the optimal control law problem using the adjoint method and a neural network trained with 1 initial condition at a time compared against the optimal switching curve.

plot, which undershoots the switching curve. From the comparison of the two plots, it can be concluded that the choice of initial conditions and initial weight sets affects the final results from the adjoint method when a neural network is being trained. Figure 5-10 shows the trajectories generated by a neural network that was trained with 100 initial conditions using the adjoint method. The neural network is able to successfully learn the general control policy for the double integrator problem, switching the control at approximately the correct time indicated by the switching curve.

5.3 Orbit-Raising Problem

After the double integrator method proved the validity of the adjoint method using neural networks for the control structure, the orbit-raising problem became the final problem to be examined using this method. Because of the use of four states and more complex dynamics, the orbit-raising problem tested the ability of the adjoint method to be applied to develop control policies for higher-dimensional systems. Also, the ability of the adjoint method to enforce equality constraints using the penalty method to form the cost function was studied. The orbit-raising problem additionally tested the use of the adjoint method for trajectory optimization, which involved using the time remaining as the only parameter entering the neural network.

5.3.1 Description

The orbit-raising problem appears in [5, p. 66-69] and is also discussed in [7, p. 7-18–7-23]. The three main states of the orbit-raising problem are the radial position r, the radial component of the velocity u, and the tangential component of the velocity v, which are shown in Figure 5-11. The dynamics code is designed to be autonomous, meaning that time is not an explicit input in the dynamics equations. Therefore, an additional state representing the remaining time t_r has to be added to the three state variables to permit the calculation of the current mass of the spacecraft without having to make time an explicit input. The control input for the problem is the thrust direction angle ϕ. As a result, the equations of motion for the spacecraft in the orbit-raising problem are

$$\dot{r} = u \tag{5.27}$$

$$\dot{u} = \frac{v^2}{r} - \frac{\mu}{r^2} + \frac{T \sin\phi}{m_o + |\dot{m}|\, t_r} \tag{5.28}$$

$$\dot{v} = -\frac{uv}{r} + \frac{T \cos\phi}{m_o + |\dot{m}|\, t_r} \tag{5.29}$$

$$\dot{t_r} = -1 \tag{5.30}$$

where r is the radial position, u is the radial component of the velocity, v is the tangential component of the velocity, t_r is the remaining time, μ is the gravitational constant, T is the thrust, m_o is the empty weight of the spacecraft, and $\dot{m}$ is the fuel

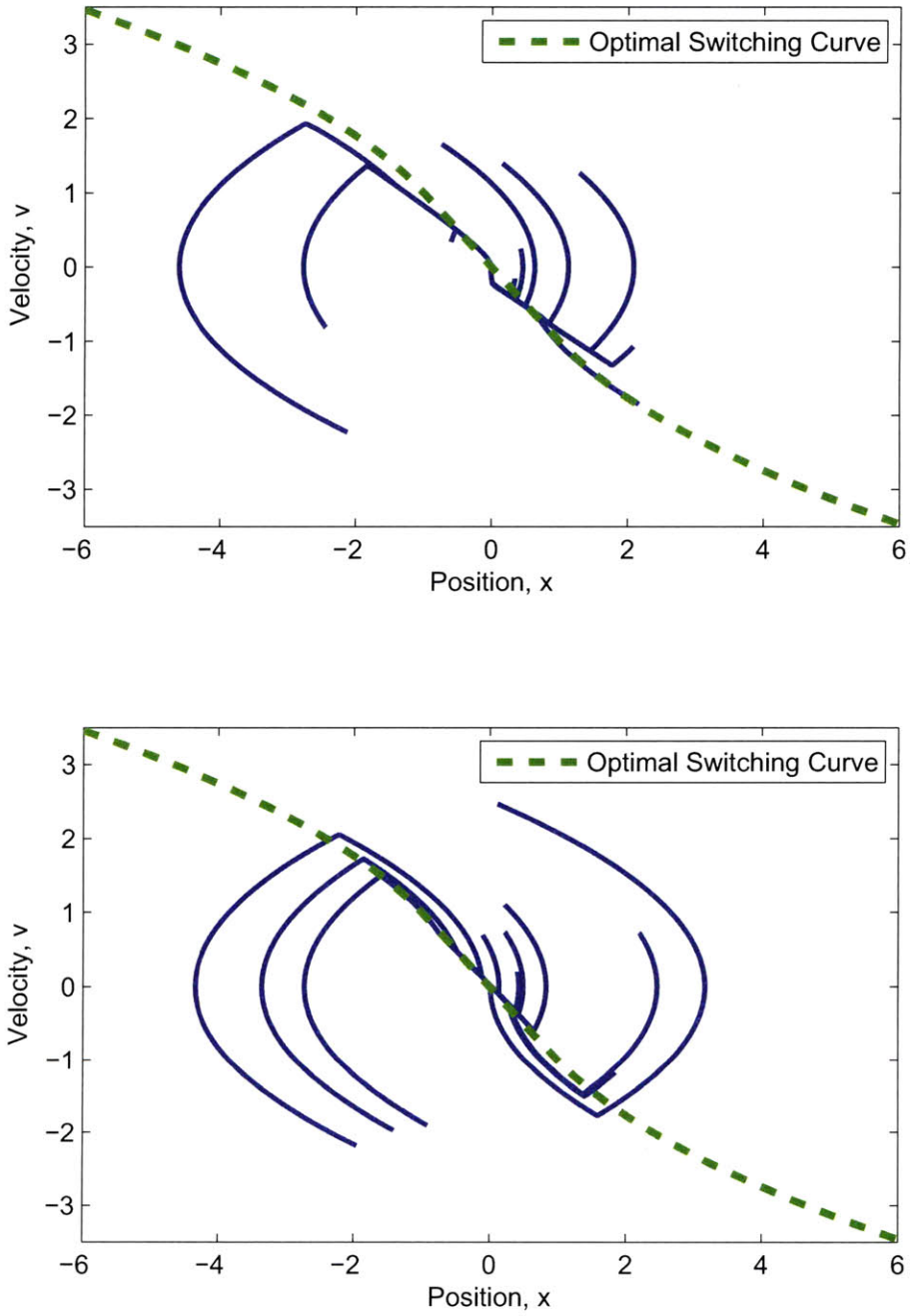

Figure 5-9: Results for the optimal control law problem using the adjoint method and a neural network trained with 10 initial conditions, compared against the optimal switching curve.

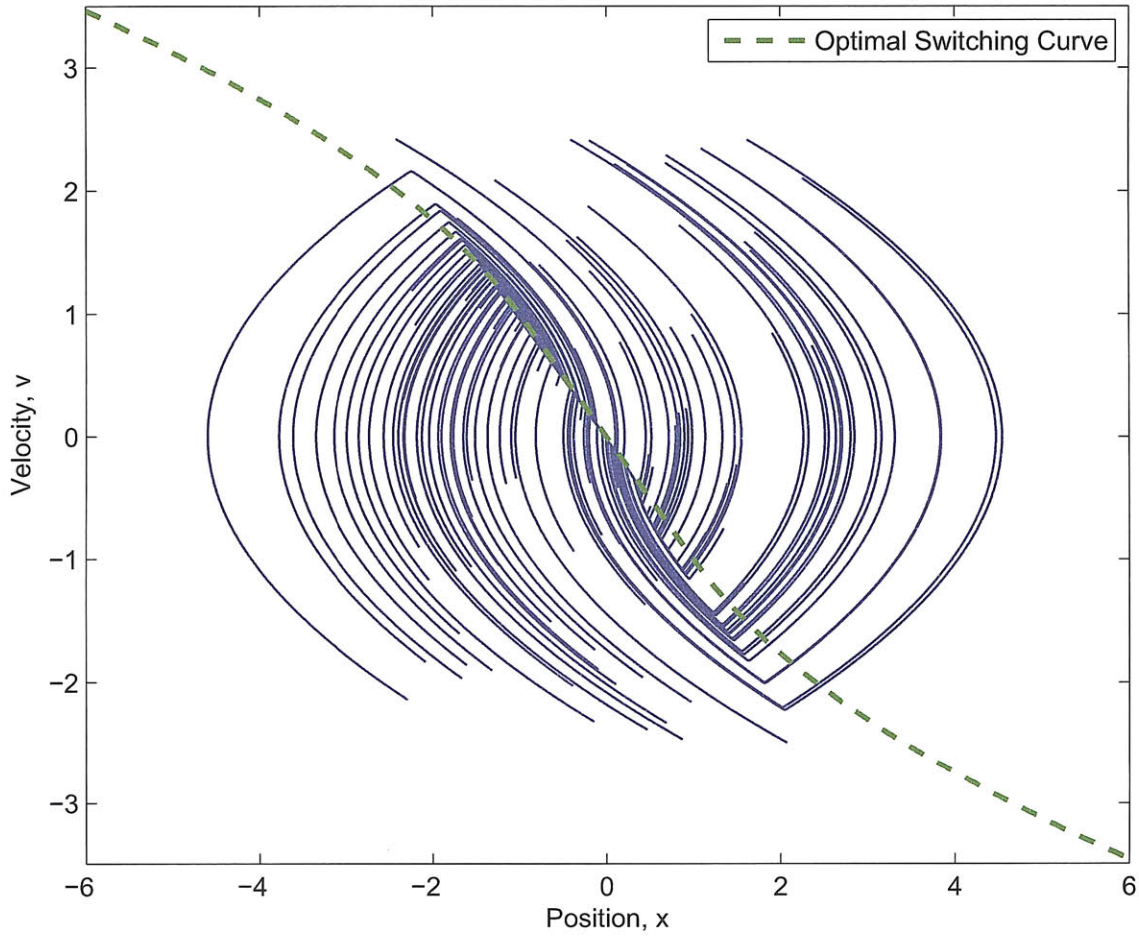

Figure 5-10: Results for the optimal control law problem using the adjoint method and a neural network trained with 100 initial conditions, compared against the optimal switching curve.

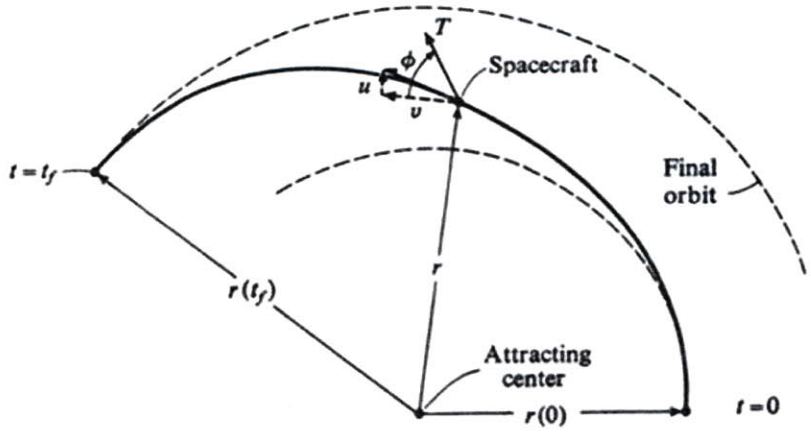

Figure 5-11: Diagram of spacecraft coordinates for the orbit-raising problem (from Figure 2.5.1 in [5, p. 66]). It should be noted that the variables $r(0)$ and $r(t_f)$ in this diagram are the initial and terminal radii of the orbit, respectively.

consumption rate [5, p. 66-67]. For this problem, the final four parameters in the list of parameters for the equations of motion are considered to be constants.

The goal of the problem is to have the spacecraft attain the highest circular orbit possible in a fixed amount of time, subject to the two terminal constraints

$$u(t_f) = 0 \tag{5.31}$$

$$v(t_f) - \sqrt{\frac{\mu}{r(t_f)}} = 0 \tag{5.32}$$

where t_f is the final time. The cost function used to express this goal is

$$J = -\frac{r(t_f)}{r(t_i)} + \alpha\left(\left(\frac{u}{\sqrt{\mu r(t_f)}}\right)^2 + \left(\frac{v}{\sqrt{\mu r(t_f)}} - 1\right)^2\right) \tag{5.33}$$

where J is the cost, $r(t_i)$ is the radial position of the spacecraft at the point when the thruster begins to fire, $r(t_f)$ is the final radius, and α is a penalty on the conditions ensuring a final circular orbit. The parameters penalized by the cost function have been normalized, which allows the constraints on the two terminal conditions to ensure that a circular orbit is reached to be weighted equally. Consequently, it is easier to determine the appropriate magnitude for the penalty α, which must be sufficiently large to emphasize the fact that the circular shape of the final orbit reached by the spacecraft should be maintained during the search for the control policy to maximize the final radius. It should be noted that the initial orbit is not assumed to be circular as the final orbit is constrained to be. By allowing the final constraints to be enforced by the cost function using the penalty α, the program for the adjoint method remains an unconstrained optimization problem, and extensive changes do not have to be

made to the code for this new problem with terminal constraints on the states.

5.3.2 Results

In order to obtain the optimal control solution to serve as a baseline for the orbit-raising problem, the MATLAB command `bvp4c`, which is a boundary value problem solver for ordinary differential equations [27, p. 8-64], was used. The code that calculated the optimal control solution can be found in [7, p. 7-22–7-23].

Once the baseline optimal control solution had been determined, the adjoint method was applied to the orbit-raising problem. The code for the dynamics and cost expressions in this problem is listed in Appendix B.3. For all of the results in this section, a 3-5-2 feedforward neural network was used. The output layer had two neurons, which were used to represent the sine and cosine of the thrust angle ϕ, which was the control variable. The expression of the thrust angle ϕ in this manner attempted to allow the neural network to handle the angle more accurately as it increased in magnitude throughout the integration process. Based on the results of the double integrator problem with a small neural network, shallower networks with fewer hidden layers would not be able to capture the characteristics of this system. The 3-5-2 neural network structure represented the optimal control policies well as can be seen in the following results. The first case to be studied was trajectory optimization, in which the time remaining was the only input to the neural network. Figure 5-12 shows a comparison between the optimal control solution found using `bvp4c` and the adjoint method solution for the initial conditions $r_o = 1$, $u_o = 0$, $v_o = 1$, and $t_{r,o} = 4$, which means that the spacecraft is initially in a circular orbit with a radius of 1. The solutions match to the point that they cannot be distinguished since the lines nearly coincide with each other. However, it should be noted that not all of the simulation runs with the adjoint methods produced results that matched the `bvp4c` solution this well. The accuracy of the adjoint method results is highly dependent on the initial weights.

The next case to be studied was the determination of an optimal control policy using the neural network for one initial condition. Because optimal control policies were being developed, the inputs into the neural network were all of the states of the system. Figure 5-13 shows a comparison between the optimal control solution found using `bvp4c` and the adjoint method solution for the same initial condition used in the trajectory optimization case. Again, the solutions match to the extent that they cannot be distinguished from each other at this level. Therefore, the adjoint method is able to find the optimal control policy for the system for one initial condition.

After finding an optimal control policy for the case of one initial condition, the adjoint method was run with multiple initial conditions to increase the generalizability and robustness of the neural network. First, the neural network was trained with 10 initial conditions, and the results of the optimization routine were examined to determine if the neural network was deriving a policy that matched the goal of the problem. As Figure 5-14 shows, the neural network that was trained with 10 initial conditions does define a control policy that increases the radius of the orbits for all of the initial conditions. Also, the control policy defined by the neural network is able

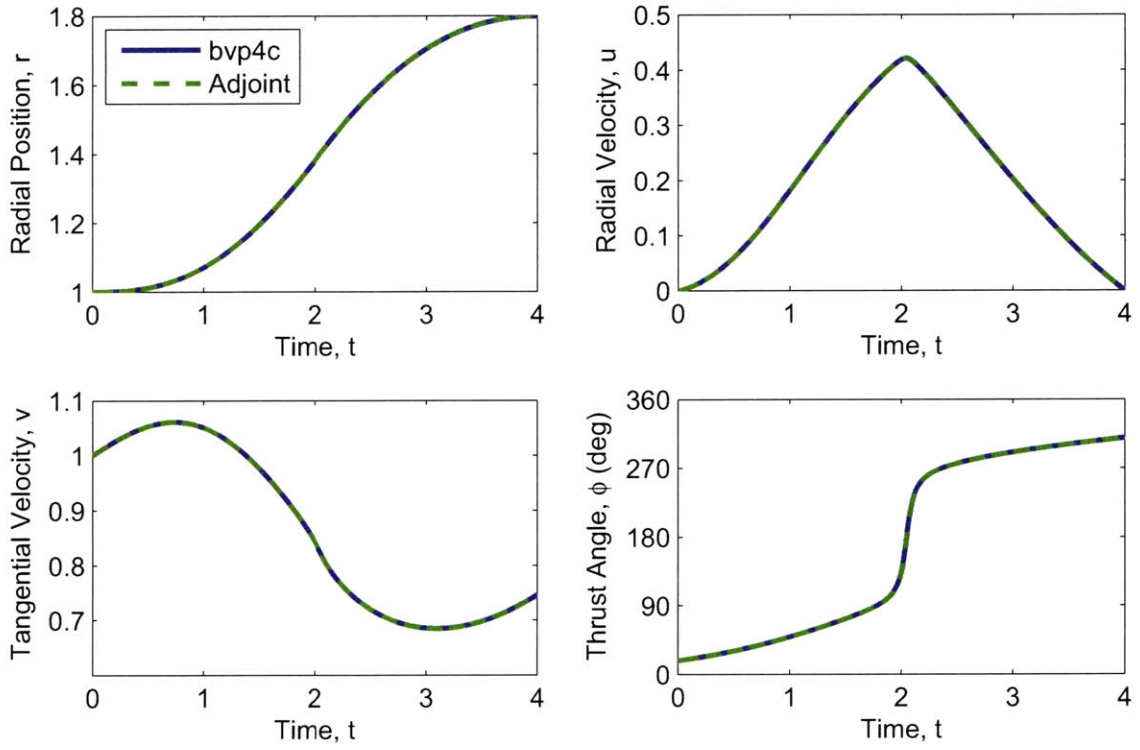

Figure 5-12: Comparison of trajectory optimization results found using MATLAB `bvp4c` and the adjoint method. The two solutions agree to the extent that the two lines representing the solution cannot be distinguished from each other.

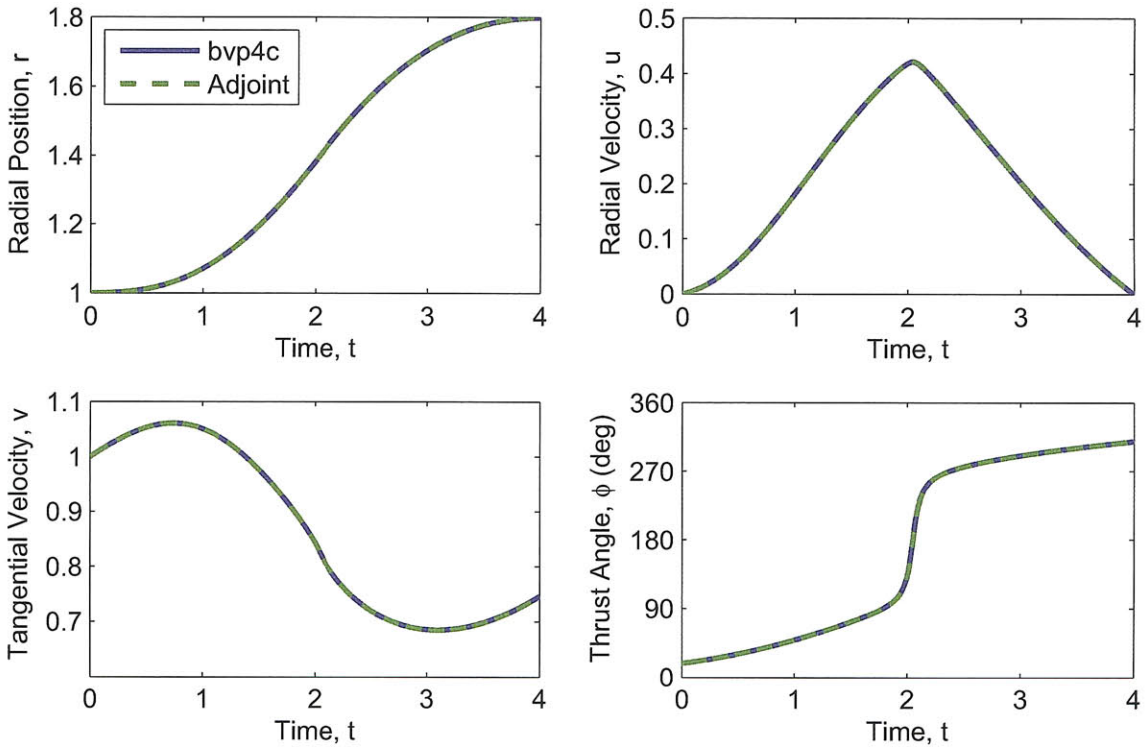

Figure 5-13: Comparison of optimal control policy results found using MATLAB `bvp4c` and the adjoint method for 1 initial condition. The two solutions agree to the extent that the two lines representing the solution cannot be distinguished from each other.

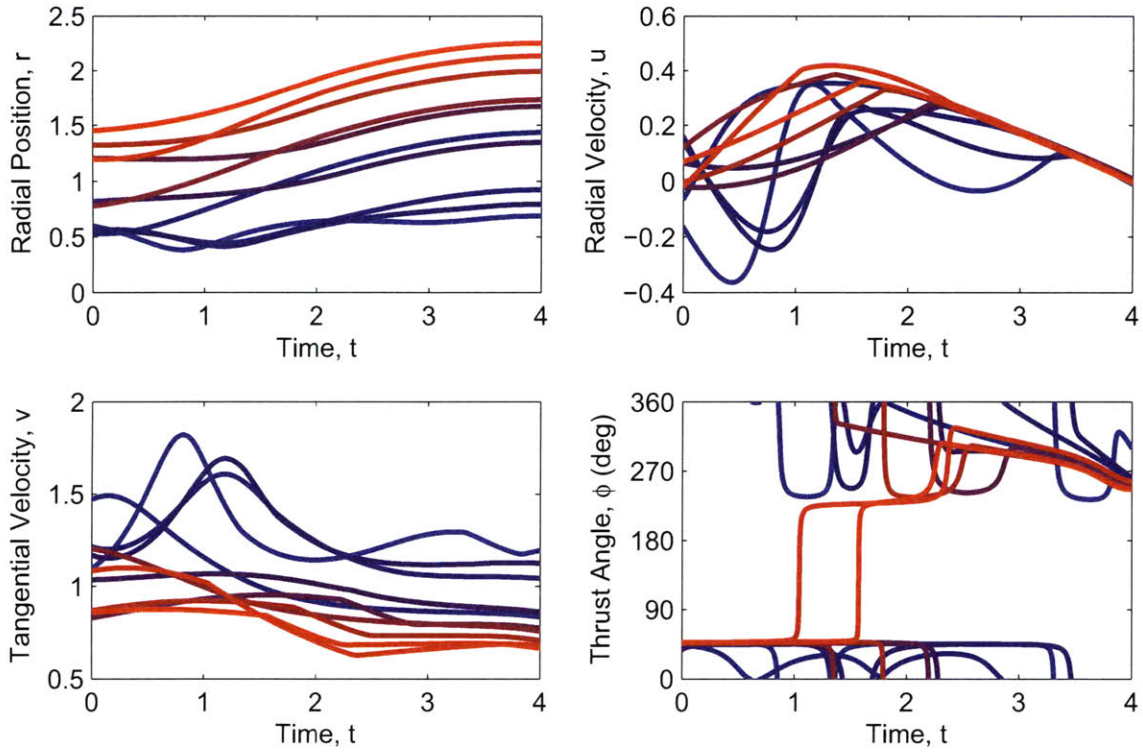

Figure 5-14: Co-plotted results of the optimal control policy determined using the adjoint method for a set of 10 initial conditions.

to enforce the terminal conditions. This result can be seen particularly in the plot of the radial velocity u. In this plot, the terminal velocity is approximately 0 for the 10 initial conditions, which is necessary in order to have the spacecraft end in a circular orbit. Therefore, the adjoint method is able to derive control policies using a neural network that is trained with 10 initial conditions. Comparing Figure 5-14 with Figure 5-15, which depicts the optimal control policies determined using the adjoint method and the same 10 initial conditions optimized individually, and also examining the final radii of the orbits achieved by training with single and multiple initial conditions in Table 5.1, it can be seen that training the neural network with 10 initial conditions does not detrimentally affect the definition of the control policy using the adjoint method and multiple initial conditions. It should be noted that the optimizer was not able to find the optimal control policy in case 9 in Table 5.1 with the given initial weights applied in the other cases. Using different initial weights allowed solutions to be found that had larger final orbit heights than the final height of case 9 for the multiple initial condition case, which is a suboptimal solution. Additionally, one interesting observation about the thrust angle plot in Figure 5-14 is that the neural network tends toward three main angles of approximately 50 degrees, 250 degrees, and 300 degrees, which differs from the optimal control policy seen in the thrust angle plot of Figure 5-15, yet the system is still able to perform near optimally.

After the adjoint method was shown to derive reasonable control policies for 10 initial conditions, the generalizability of training the neural network with multiple

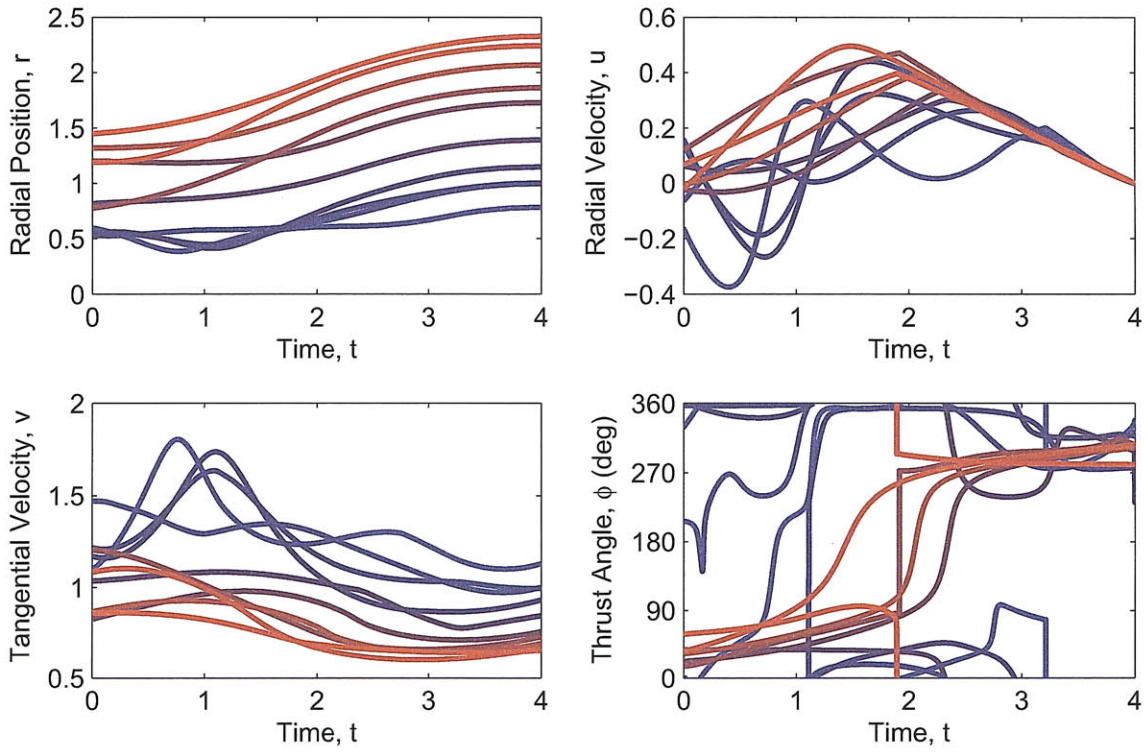

Figure 5-15: Co-plotted results of 10 individual optimal control policies determined using the adjoint method for 10 separate initial conditions.

Table 5.1: Comparison of the final orbit height reached for 10 different initial conditions between the neural networks trained with 1 initial condition and with 10 initial conditions using the adjoint method. The solutions trained with a single initial condition and the same weights from the simulation in Figure 5-13 were used as the baseline optimal control policy based on the previous success of the method. An exception occurred in case 9, where the optimizer was not able to determine the optimal solution from that set of initial weights. When the case was run with different initial weights, final orbit heights greater than the height for the multiple initial condition case were sometimes achieved.

	Final Orbit Height	
Case	1 Initial Condition	10 Initial Conditions
1	2.3270	2.2508
2	2.2423	2.1325
3	2.0680	1.9919
4	1.8630	1.7295
5	1.7289	1.6682
6	1.3938	1.3456
7	1.1496	0.7929
8	1.0041	0.9230
9	0.9971	1.4337
10	0.7818	0.6866

initial conditions was studied. This case was described in Section 3.4. For this case, 100 sets of initial conditions were randomly selected; all of the sets of initial conditions had the same value for the time remaining state t_r of 4. The radius r was allowed to vary between 0.5 and 1.5. The tangential component of the velocity v had a range of $\pm 20\,\%$ around the value of v necessary for a circular orbit at the radius r. The radial component of the velocity u was allowed to vary around 0 with a range of $\pm 20\,\%$ of the value of v. In order to attempt to prevent overtraining of the neural network, the optimization routine was run for sets of 10 initial conditions for a small, restricted number of iterations, and the resulting weights for the neural network at the end of each run were used as the initial weights for the following run with a new initial condition. Figure 5-16 depicts the comparison between the optimal control solution found using `bvp4c` and the adjoint method solution for the initial conditions $r_o = 1$, $u_o = 0$, $v_o = 1$, and $t_{r,o} = 4$, which was in the final set of 10 initial conditions from the 100 initial conditions used to train the neural network. Comparing Figure 5-16 with Figure 5-13, where the initial condition of the plot was the only initial condition used for training, it can be seen that there is a loss of performance in the control policy of the neural network trained with multiple initial conditions. However, the loss of performance is offset by a gain in the robustness of the control policy developed using multiple initial conditions, which will be able to handle initial conditions outside of the training set better than the control policy developed using one initial condition. Figure 5-17 shows the comparison between the optimal control solution found using `bvp4c` and the adjoint method solution for the initial conditions $r_o = 1$, $u_o = 0$, $v_o = 1$, and $t_{r,o} = 4$, which was not in the 10 sets of 10 initial conditions used to train the neural network. The adjoint method is able to recognize the general control solution for the new initial condition. However, the response is suboptimal for this initial condition that is not in the training data set.

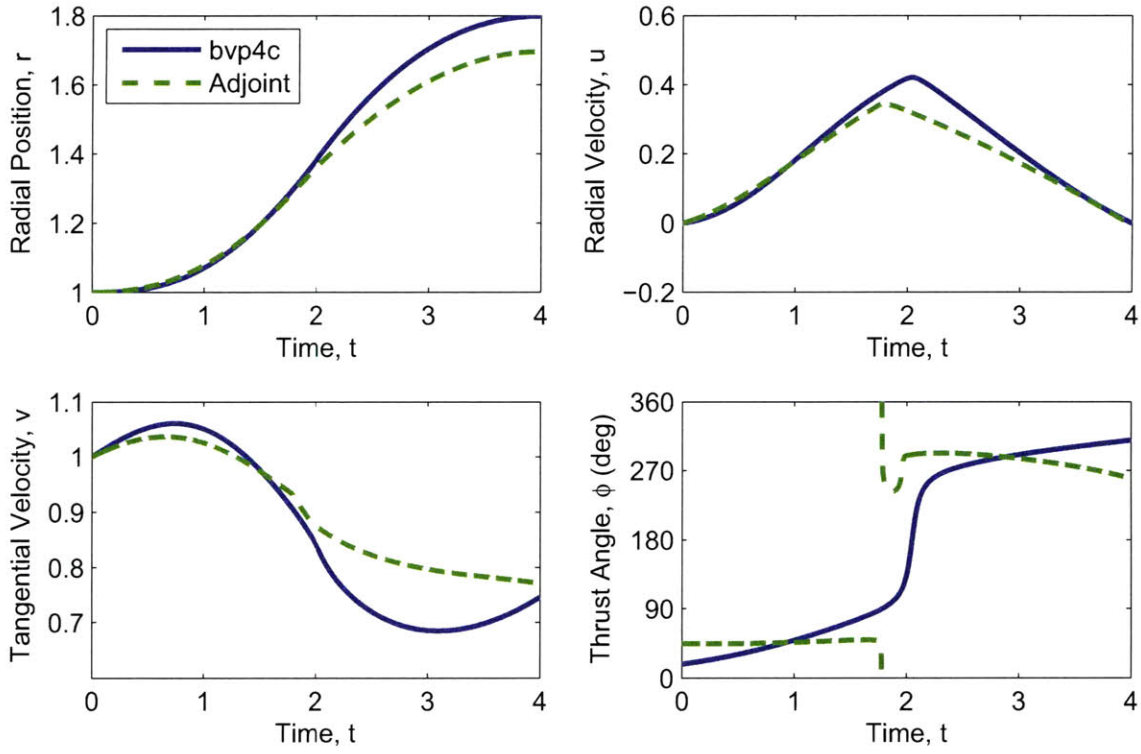

Figure 5-16: Comparison of optimal control policy results found using MATLAB `bvp4c` and the adjoint method trained with 10 sets of 10 initial conditions and run for a selected initial condition in the training data set.

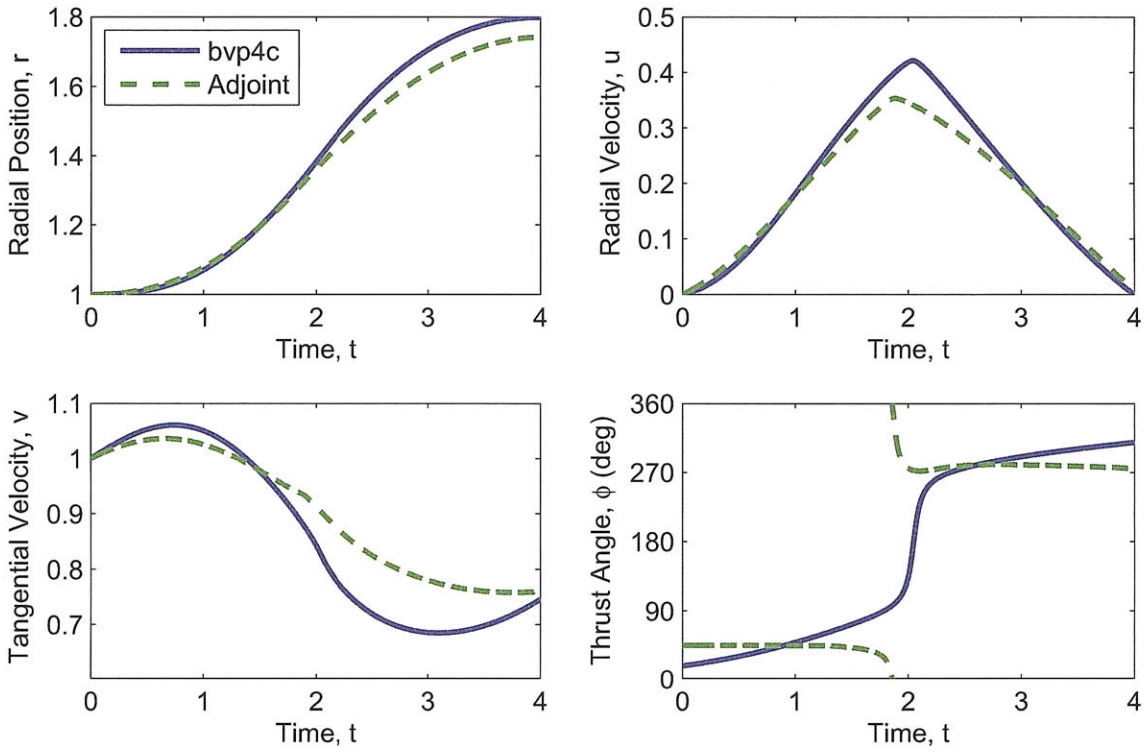

Figure 5-17: Comparison of optimal control policy results found using MATLAB `bvp4c` and the adjoint method trained with 10 sets of 10 initial conditions and run for a selected initial condition outside of the training data set.

Chapter 6

Conclusions

"Let me explain. No, there is too much. Let me sum up."

- *The Princess Bride* [28].

6.1 Conclusions

This thesis examined the ability of the adjoint method to solve control problems using a fixed structure controller, optimized using automatic differentiation. The adjoint method was applied to three example applications with known solutions to evaluate if optimal trajectories and control policies could be determined for a fixed-gain controller and for a neural network feedback control structure. Additionally, the generalizability of control policies defined by neural networks trained with multiple initial conditions was studied to evaluate the tradeoff between performance and robustness for these types of controllers.

One of the main goals of this thesis was to validate the adjoint method using three problems for which optimal solutions were known. For both the case of the fixed-gain controller and the case of the neural network controller trained with one initial condition, the adjoint method was able to determine the optimal control solution. The solutions from several cases matched the optimal solutions defined by proven methods, to the point where the two solutions could not be distinguished from each other. The cases in which this result occurred were the adjoint method for the linearized aircraft dynamics, the trajectory optimization tests for the double integrator problem and for the orbit-raising problem, and the optimal control policy test for a neural network trained with one initial condition in the orbit-raising problem. For the single initial condition optimal control policy case of the double integrator problem, the solutions derived using the adjoint method generally followed the optimal control policy indicated by the switching curve, although the control often switched slightly after the optimal switching curve. From all of these problems, it was discovered that the ability of the method to determine the optimal solution was highly dependent on the depth of the neural network and on the initial weights. Increasing the depth and complexity of the neural network for the double integrator problem allowed the optimal control policy to be determined more consistently. However, for the double

integrator and orbit-raising problems that used neural networks as the control structure, a solution could not be found for every set of initial weights. Therefore, it was concluded that the choice of the initial weights affected the final control policies. The optimization of some initial weights resulted in optimal control policies, while other initial weights caused the optimizer to fail or give suboptimal solutions. Nevertheless, the adjoint method for optimal feedback control was able to be applied successfully to three different types of problems.

Another goal of this thesis was to determine if neural networks that were trained with multiple initial conditions could recognize general optimal control policies and be robust enough to result in near-optimal solutions for points not included in the training data set. For the double integrator problem, the neural network was able to successfully determine the general shape of the optimal control policy for most sets of initial weights. However, suboptimal control policies and state trajectories were found for some sets of initial weights. For the orbit-raising problem, the solution for a neural network that was trained using 10 initial conditions matched the general shape of the state trajectories for 8 of the 10 initial conditions. Interestingly, in some cases, the controls defined by the neural network trained with 10 initial conditions did not resemble the optimal control policy. For those cases, the thrust angle would rotate in the opposite direction from the optimal angle to a final value that was typically different than the corresponding angle for the optimal control solution. This tendency of the neural network to find an dissimilar control policy that still produced solutions with near-optimal performance was also noted in the cases in which the neural network was trained with 10 sets of 10 initial conditions. For these cases of the orbit-raising problem, it appears that the optimization process found a local minimum, and after the neural network begins to learn the policy associated with this minimum, it cannot learn the optimal control policy indicated by the known solution.

Additionally, in the orbit-raising problem, the adjoint method was found to produce neural networks that were generalizable for initial conditions not included in the training data set, in the sense that the shape of the optimal state trajectories could be determined. However, while the state responses for tests of initial conditions outside of the training data set were qualitatively correct, the final radii for the multiple initial condition case were less than the optimal final radii, and a different control than that of the known optimal solution was found. Based on these results, while the neural networks were able to find the general optimal control policies for most cases, the use of neural networks as the control structure in the adjoint method should continue to be evaluated.

6.2 Future Work

This thesis found that the performance of the adjoint method using neural networks was highly influenced by the initial weights. The effect of the initial weights on the resulting optimized control policy should be examined further. Also, because of the initial weight sensitivity, it appears that neural networks do not determine optimal solutions in every case. Other control structures, such as radial basis functions,

should be studied in conjunction with the adjoint method to see if the reliability of the method could be improved. At the same time, techniques for improving the ability of the adjoint method to determine optimal control solutions using neural networks should be studied, such as the introduction of noise into the optimization process to help avoid local minima.

Another aspect of the adjoint method that was noticed in the simulations for the example problems was the fact that the MATLAB unconstrained optimization routine had a tendency to take large initial steps as noted in the linearized aircraft dynamics problem. Occasionally, the optimization process could not recover from the first jumps and find the minimum cost. The settings of the unconstrained optimization routine should be analyzed further to determine if settings could be applied to limit the step size that the optimizer could take so that the convergence to the minimum cost could proceed more rapidly and efficiently.

Finally, the applicability of the adjoint method to more complex and higher-dimensional systems should be studied. The addition of states to the system often causes a corresponding increase in the complexity of the control structure, which will mean that the optimizer will have more parameters to adjust during the optimization process, potentially making it harder to find optimal solutions. A validation process similar to the procedure carried out in this thesis should be used to build confidence in the method as applied to higher-dimensional systems.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A

General Infrastructure Codes

This appendix contains the general infrastructure codes that are used to run the adjoint method. The codes from the double integrator problem are presented as a working example. The codes included in this appendix are the two optimization files `optimizeWin2.m` and `myfun.m`, the code used in the automatic differentiation process `MakeMexFunctionsWin3.m`, an example gateway code for the forward simulation `gateway1win1.F90`, the generated neural network code `nn.F90`, the main code used to simulate the system `main.F90`, and the file that defines the system and neural network structure `parameters.m`. Combined with the `dynamics.F90` code in Appendix B.2, these constitute the set of codes necessary to simulate the system and to run the adjoint method. It should be noted that the codes `optimizeWin2.m`, `myfun.m`, `MakeMexFunctionsWin3.m`, and `parameters.m` codes are written in MATLAB, while the codes `gateway1win1.F90`, `nn.F90`, and `main.F90` are written in FORTRAN. For an explanation of the integration and operation of the codes, see Chapter 4.

A.1 Optimization Files

```
% optimizeWin2.m
global w1 w2 w3 w4 w5 %#ok<*NUSED> %puts suboptimal weights for
                      %neural net on a stack so they can be used
global nx nu ny nInputs nNeurons neuronType
y = MakeMexFunctionsWin3('simpleThruster')   %#ok<*NOPTS>

dt = 0.01;
t = 0:dt:30;

options = optimset('GradObj','on','Display','on',...
    'LevenbergMarquardt','on','DerivativeCheck','off',...
    'LargeScale','off','Diagnostics','on','HessUpdate','bfgs',...
    'TolFun',1e-9,'TolX',1e-9,'MaxIter',5000);

enum = 100;
```

```
X(1,:) = -3 + 6*rand(enum,1);
X(2,:) = -2 + 4*rand(enum,1);

W = 0.1*randn(y,1);
[w,fval] = fminunc(@(w) myfun2(w,X,t,'simpleThruster_b'),W,options);

[j,x,u] = simpleThruster(X(:,1),w,t);

figure(1)
plot(x(1,:),x(2,:))
title('v vs. x')
xlabel('x (m)')
ylabel('v (m/s)')

figure(2)
plot(t,x(1,:))
title('x')
xlabel('t (s)')
ylabel('x (m)')

figure(3)
plot(t,x(2,:))
title('v')
xlabel('t (s)')
ylabel('v (m/s)')

figure(4)
plot(t,u)
title('u')
xlabel('t (s)')
ylabel('u (m/s^2)')

% myfun2.m
function [F,G] = myfun2(w,X,t,func)

global w1 w2 w3 w4 w5
w5 = w4;
w4 = w3;
w3 = w2;
w2 = w1;
w1 = w;
persistent N

if length(N) == 0, N = 0; end
N = N+1;
```

```
W = w;
F = 0;
G = 0;

if nargout ==1
    for i=1:size(X,2);
        x = X(:,i);
        [f] = eval([func '(x,w,t)']);
        F = F + f;
    end
else
    for i=1:size(X,2);
        x = X(:,i);
        [f,g] = eval([func '(x,w,t)']);
        F = F + f;
        G = G + g;
    end
end

if mod(N,1) == 0
     fprintf('%i,  %f\n',nargout,F)
end
return
```

A.2 MakeMexFunctionsWin3 Code

```
% MakeMexFunctionsWin3.m
function y = MakeMexFunctionsWin3(name,varargin)
%Y = MakeMexFunctionsWin3(NAME);
%Y = MakeMexFunctionsWin3(NAME,APP);
%Creates MEX functions based on files stored in the directory referred
%to by NAME.  In general, this funcion will look for a file named
%'dynamics.F90' (and a few other files as well), if a second input is
%given, the file ['dynamics' APP '.F90'] will be used.
%Examples:
%name = 'pendulum';  pendulum.F
%name = 'glider1'; app = '_con'; glider1_con.F90
%
if nargin == 1;
    app = '';
elseif nargin ==2;
    app = varargin{1};
end;
```

```
%function y = MakeMexFunctionsWin(name)
%
%   Based on Prof Hall's MakeMexFunctions.m
%
%   Function that makes the appropriate mex functions in the directory
%   'dir' to allow optimization of a neural net control strategy
%
%   The following files must be in the directory 'name':
%
%       parameters.m      parameters for neural net controller,
%                         dynamic system
%       dynamics.F90
%

cr = [' \' char(10)];
sp = [' '];

%...Set up the directory names used throughout
basedir     = [fileparts(which('MakeMexFunctionsWin3')) ];
dynamicsdir = [basedir     '\..\' name ];
builddir    = [dynamicsdir '\build'];
tapdir      = ['C:\tapenade3.1'];

fortrandir  = [basedir      '\..\FortranTemplates'];
stackdir    = [fortrandir   '\ADFirstAidKit'];

%Clean up directories
curdir = pwd;

cd(tapdir);
fclose all;

cd(builddir);
fclose all;

cd(curdir);

%Read data from the parameter file
%Change from MakeMexFunctions.m:
%Instead of copying parameter.m to basedir, just run it from the
%dynamicsdir
disp('Reading from parameter file ...')

cd(dynamicsdir);
parameters;  %Runs parameter.m to load needed parameters into workspace
```

```
cd(curdir);

%Using data from the parameter file, set up the neural net controller
disp('Generating neural net program ...');

[s,nw] = makenn1(nInputs,nNeurons,neuronType);
file1 = [builddir '\nn.F90'];
fclose all;
fid = fopen(file1,'w');
if fid == 0
    disp(['Couldn''t open file ' file1])
end
n = fwrite(fid,s);
if n ~= length(s)
    disp(['Couldn''t write to file ' file1])
end;

%-----------------------------------------------------------------------
% We need to find the adjoint code, using TAPENADE. To do this, we
% first preprocess the fortran code, since TAPENADE doesn't recognize
% compiler directives. So first compile all the files with the -E
% compiler directive, to replace all the symbolic values with numeric
% values.

disp('Stripping compiler directives from code ...')

file2 = [fortrandir     '\main.F90'];
file3 = [dynamicsdir    '\dynamics' app '.F90'];
file4 = [builddir       '\nn.F90'];
file5 = [builddir       '\' name app '.f90'];

%Combine RK integration routine (main.F90), equation of motion for the
%problem (dynamics.F90), and the neural net (nn.F90) into one file.
command = ['gfortran -E -DNX=' num2str(nx) ' -DNU=' num2str(nu) ...
    ' -DNW=' num2str(nw) ' -DNY=' num2str(ny) sp '"' file2 ...
    '"' sp '"' file3 '"' sp '"' file4 '"' sp ' > ' '"' file5 '"'];
disp(' ');
disp(command);

[status,result] = system(command);
if status
    disp(result);
    return
end
```

```
% Now we must comment out remaining compiler directives
fclose all;
fid = fopen(file5,'r');
s = fread(fid,'*char')';

fclose all;
s = strrep(s,'#','!#');
fid = fopen(file5,'w');
n = fwrite(fid,s);

if n ~= length(s)
    disp(['Failed to write to file ' file5])
end
%------------------------------------------------------------
% Now we are ready to call TAPENADE
command = ['"' tapdir '\bin\tapenade' '"'...
    ' -inputlanguage fortran95 ' ...
    '-outvars "j" -vars "alpha w" -head main4 -reverse -o ' ...
    name app ' -outputlanguage fortran90 -O ' '"' builddir ...
    '"' ' -diffvarname ' '_b -i4 -dr8 -r4 ' '"' file5 '"'];
% Took out -parserfileseparator "\"  from command

disp(' ');
disp(command);

[status,result]=system(command);
disp(' ')
disp(result)
if status, return, end

file3 = [builddir    '\' name app '.F90'];
file4 = [builddir    '\' name app '_b.F90'];
file6 = [stackdir    '\' 'adBuffer.f'];
file7 = [stackdir    '\' 'adStackNoUnderscore.c'];

% Do the MEX command for the forward case.
file1 = [fortrandir  '\' 'gateway1win1.F90'];

%Also need mexinterface.mod compiled from mexinterface_c.f90 ...
%Make sure mexinterface.mod is accessible by MEX, and is up-to-date

command = ['mex '...
    '-DNX=' num2str(nx) ' -DNU=' num2str(nu) ' -DNW=' num2str(nw) ...
    ' -DNY=' num2str(ny) ' -output ' name app ...
    ' ' '"' file1 '"' ' ' '"' file3 '"' ];
```

```
disp(command);
eval(command);

% Do the MEX command for the reverse case.
file1 = [fortrandir  '\' 'gateway2win1.F90'];
command = ['mex '...
    '-DNX=' num2str(nx) ' -DNU=' num2str(nu) ' -DNW=' num2str(nw) ...
    ' -DNY=' num2str(ny) ' -output ' name app '_b' ...
    ' ' '"' file1 '"' ' ' '"' file3 '"' ' ' '"' file4 '"' ' ' '"' ...
    file6 '"' ' ' '"' file7 '"'];
disp(command);
eval(command);

y=nw;

fclose('all');

return
```

A.3 Example Gateway Code

```
! gateway1win1.F90
!#include <fintrf.h>
! use #include "fintrf.h" for mwIndex, mwPointer mwSize
!
! This is the generic gateway function
!
! It assumes a MATLAB call of the form
!
!    [J,x,u] = main(x0,w,t)
!
! where
!    x0 is the initial state vector (must be NXx1)
!    w is the vector of controller parameters (must be a vector)
!    t is the time vector over which the cost J is computed (nt x 1)
!    Jw is the sensitivity of the cost to the controller parameters
!    x is the resulting state history (nt x 4)
!    u is the resulting control history (nt x 1)
!
!234567890123456789012345678901234567890123456789012345678901234567890123456789
```

```
subroutine mexfunction(nlhs, plhs, nrhs, prhs)

    use mexinterface
    implicit none

    !Insert interface here if needed
    interface
      subroutine main1(x0,w,t,nt,J)
        integer :: nt
        real*8  :: x0(*), w(*), t(*)
        real*8 :: J !, x(*)
      end subroutine main1
      subroutine main2(x0,w,t,nt,J,x)
        integer :: nt
        real*8  :: x0(*), w(*), t(*)
        real*8 :: J, x(*)
      end subroutine main2
      subroutine main3(x0,w,t,nt,J,x,u)
        integer :: nt
        real*8  :: x0(*), w(*), t(*)
        real*8 :: J, x(*), u(*)
      end subroutine main3
    end interface

    integer(4) :: ii, error, i, i2
    integer(4) :: nlhs, nrhs
    integer(4) :: plhs(nlhs), prhs(nrhs)
    real(8), pointer :: Jp, xp(:), up(:), x0p(:), wp(:), tp(:)
    integer(4) :: m, n, nel, nt, nw, nx
    real(8)    :: J, x0(NX), w(NW)

    nw = NW
    nx = NX

! check for proper number of arguments

if (nrhs .ne. 3) then
call mexerrmsgtxt('pendulum requires three input arguments'//char(0))
elseif (nlhs .gt. 3) then
call mexerrmsgtxt( &
       & 'pendulum requires three or fewer output arguments'//char(0))
endif

! check the dimensions of x0.  It can be NX x 1 or 1 x NX.
```

```
m = mxgetm(prhs(1))
n = mxgetn(prhs(1))

if ((max(m,n) .ne. NX) .or. (min(m,n) .ne. 1)) then
call mexerrmsgtxt('pendulum requires that x0 be a nx x 1 vector' &
          &//char(0))
endif

! check the dimensions of w.  it can be nw x 1 or 1 x nw.

m = mxgetm(prhs(2))
n = mxgetn(prhs(2))

if ((max(m,n) .ne. NW) .or. (min(m,n) .ne. 1)) then
call mexerrmsgtxt('pendulum requires that w be a nw x 1 vector' &
          &//char(0))
endif

! check the dimensions of t.  it can be nt x 1 or 1 x nt.

m = mxgetm(prhs(3))
n = mxgetn(prhs(3))

if ((max(m,n) .lt. 2) .or. (min(m,n) .ne. 1)) then
call mexerrmsgtxt( &
   &'pendulum requires that t be a vector with at least 2 elements' &
          &//char(0))
endif
nt = max(m,n)

! create a matrix for J argument on LHS

plhs(1) = mxcreatedoublematrix(1,1,0)
call c_f_pointer(mxgetpr(plhs(1)),Jp)

! create a matrix for x argument on LHS

if (nlhs .ge. 2) then
        plhs(2) = mxcreatedoublematrix(NX,nt,0)
        nel = NX*nt
    call c_f_pointer(mxgetpr(plhs(2)),xp, [nel])
endif

! create a matrix for u argument on LHS
```

```
if (nlhs .ge. 3) then
        plhs(3) = mxcreatedoublematrix(NU,nt,0)
        nel = NU*nt
        call c_f_pointer(mxgetpr(plhs(3)),up,[nel])
endif

! copy right hand arguments to local arrays or variables

    nel = NX
    call c_f_pointer(mxgetpr(prhs(1)),x0p,[nel])
    call c_f_pointer(mxgetpr(prhs(2)),wp,[nw])
    call c_f_pointer(mxgetpr(prhs(3)),tp,[nt])

! do the actual computations in a subroutine and then copy result
! to LHS

if (nlhs .le. 1) then
    call main1(x0p,wp,tp,nt,Jp)
        return
end if

if (nlhs .eq. 2) then
call main2(x0p,wp,tp,nt,Jp,xp)
        return
end if

if (nlhs .eq. 3) then
call main3(x0p,wp,tp,nt,Jp,xp,up)
        return
end if

return
end
```

A.4 nn Code

```
subroutine nn(p,q,w)
!
! Automatically generated neural net code
!
  implicit none

  real*8 :: p(2), q(1), w(83)
```

```
   intrinsic  tanh, exp, atan

   real*8 :: p_1_1, q_1_1
   real*8 :: p_1_2, q_1_2
   real*8 :: p_1_3, q_1_3
   real*8 :: p_2_1, q_2_1
   real*8 :: p_2_2, q_2_2
   real*8 :: p_2_3, q_2_3
   real*8 :: p_2_4, q_2_4
   real*8 :: p_2_5, q_2_5
   real*8 :: p_2_6, q_2_6
   real*8 :: p_2_7, q_2_7
   real*8 :: p_3_1, q_3_1
   real*8 :: p_3_2, q_3_2
   real*8 :: p_3_3, q_3_3
   real*8 :: p_3_4, q_3_4
   real*8 :: p_3_5, q_3_5
   real*8 :: p_4_1, q_4_1

   p_1_1 = w(1)*p(1) + w(2)*p(2) + w(3)
   q_1_1 = atan(p_1_1)
   p_1_2 = w(4)*p(1) + w(5)*p(2) + w(6)
   q_1_2 = atan(p_1_2)
   p_1_3 = w(7)*p(1) + w(8)*p(2) + w(9)
   q_1_3 = atan(p_1_3)
   p_2_1 = w(10)*q_1_1 + w(11)*q_1_2 + w(12)*q_1_3 + w(13)
   q_2_1 = atan(p_2_1)
   p_2_2 = w(14)*q_1_1 + w(15)*q_1_2 + w(16)*q_1_3 + w(17)
   q_2_2 = atan(p_2_2)
   p_2_3 = w(18)*q_1_1 + w(19)*q_1_2 + w(20)*q_1_3 + w(21)
   q_2_3 = atan(p_2_3)
   p_2_4 = w(22)*q_1_1 + w(23)*q_1_2 + w(24)*q_1_3 + w(25)
   q_2_4 = atan(p_2_4)
   p_2_5 = w(26)*q_1_1 + w(27)*q_1_2 + w(28)*q_1_3 + w(29)
   q_2_5 = atan(p_2_5)
   p_2_6 = w(30)*q_1_1 + w(31)*q_1_2 + w(32)*q_1_3 + w(33)
   q_2_6 = atan(p_2_6)
   p_2_7 = w(34)*q_1_1 + w(35)*q_1_2 + w(36)*q_1_3 + w(37)
   q_2_7 = atan(p_2_7)
   p_3_1 = w(38)*q_2_1 + w(39)*q_2_2 + w(40)*q_2_3 + w(41)*q_2_4 + &
&    w(42)*q_2_5 + w(43)*q_2_6 + w(44)*q_2_7 + w(45)
   q_3_1 = atan(p_3_1)
   p_3_2 = w(46)*q_2_1 + w(47)*q_2_2 + w(48)*q_2_3 + w(49)*q_2_4 + &
&    w(50)*q_2_5 + w(51)*q_2_6 + w(52)*q_2_7 + w(53)
   q_3_2 = atan(p_3_2)
```

```
  p_3_3 = w(54)*q_2_1 + w(55)*q_2_2 + w(56)*q_2_3 + w(57)*q_2_4 + &
&   w(58)*q_2_5 + w(59)*q_2_6 + w(60)*q_2_7 + w(61)
  q_3_3 = atan(p_3_3)
  p_3_4 = w(62)*q_2_1 + w(63)*q_2_2 + w(64)*q_2_3 + w(65)*q_2_4 + &
&   w(66)*q_2_5 + w(67)*q_2_6 + w(68)*q_2_7 + w(69)
  q_3_4 = atan(p_3_4)
  p_3_5 = w(70)*q_2_1 + w(71)*q_2_2 + w(72)*q_2_3 + w(73)*q_2_4 + &
&   w(74)*q_2_5 + w(75)*q_2_6 + w(76)*q_2_7 + w(77)
  q_3_5 = atan(p_3_5)
  p_4_1 = w(78)*q_3_1 + w(79)*q_3_2 + w(80)*q_3_3 + w(81)*q_3_4 + &
&   w(82)*q_3_5 + w(83)
  q(1) = atan(p_4_1)

  return

end subroutine
```

A.5 main Code

```
! main.F90
subroutine main1(x0,w,t,nt,J)
     implicit none

! !Input variables
integer :: nt
real*8  :: x0(NX), w(NW), t(nt)

! !Output variables
real*8 :: J

! !Working variables
integer  :: i
real*8   :: dt, x1(NX), xf(NX), u1(NU), dJ, x(NX,nt), u(NU,nt), C

! !Initialize cost, state vector
J = 0.
! x1 = x0
    x(:,1) = x0

! !Integrate the dynamics and cost forward in time
do i=1,nt-1
! !Find the time increment to pass to Runge-Kutta routine
dt = t(i+1)-t(i)
! !Do the Runge-Kutta step
```

```
x1 = x(:,i)
call rk(x1,w,dt,xf,dJ,u1)
x(:,i+1) = xf
        u(:,i) = u1
! x1 = xf
J = J + dJ
end do
    u(:,nt) = u(:,nt-1)

    call terminal(x(:,nt),u(:,nt),t(nt),dJ)
    J = J + dJ

    call general(x,u,nt,t,C)
    J = J + C

return
end subroutine
!=============================================================
subroutine main2(x0,w,t,nt,J,x)
implicit none

! !Input variables
integer :: nt
real*8  :: x0(NX), w(NW), t(nt)

! !Output variables
real*8 :: J, x(NX,nt)

! !Working variables
integer  :: i
real*8   :: dt, x1(NX), xf(NX), u1(NU), dJ, u(NU,nt), C

! !Initialize cost, state vector
J = 0.
x(:,1) = x0

! !Integrate the dynamics and cost forward in time
do i=1,nt-1
! !Find the time increment to pass to Runge-Kutta routine
dt = t(i+1)-t(i)
!Do the Runge-Kutta step
x1 = x(:,i)
call rk(x1,w,dt,xf,dJ,u1)
x(:,i+1) = xf
u(:,i) = u1
```

```
        J = J + dJ
end do
    u(:,nt) = u(:,nt-1)

    call terminal(x(:,nt),u(:,nt),t(nt),dJ)
    J = J + dJ

    call general(x,u,nt,t,C)
    J = J + C

return
end subroutine
!=============================================================
subroutine main3(x0,w,t,nt,J,x,u)
implicit none

! !Input variables
integer :: nt
real*8  :: x0(NX), w(NW), t(nt)

! !Output variables
real*8 :: J, x(NX,nt), u(NU,nt)

! !Working variables
integer  :: i
real*8   :: dt, x1(NX), xf(NX), u1(NU), dJ, C

! !Initialize cost, state vector
J = 0.
x(:,1) = x0

! !Integrate the dynamics and cost forward in time
do i=1,nt-1
! !Find the time increment to pass to Runge-Kutta routine
dt = t(i+1)-t(i)
! !Do the Runge-Kutta step
x1 = x(:,i)
call rk(x1,w,dt,xf,dJ,u1)
x(:,i+1) = xf
u(:,i) = u1
J = J + dJ
end do
u(:,nt) = u(:,nt-1)

    call terminal(x(:,nt),u(:,nt),t(nt),dJ)
```

```
    J = J + dJ

    call general(x,u,nt,t,C)
    J = J + C

return
end subroutine
!=============================================================
subroutine main4(x0,w,t,nt,J,alpha)
     implicit none

! !Input variables
integer :: nt
real*8  :: x0(NX), w(NW), t(nt), alpha

! !Output variables
real*8 :: J

! !Working variables
integer  :: i
real*8   :: dt, x1(NX), xf(NX), u1(NU), dJ, x(NX,nt), u(NU,nt), C

! !Initialize cost, state vector
J = 0.
x(:,1) = x0

! !Integrate the dynamics and cost forward in time
do i=1,nt-1
! !Find the time increment to pass to Runge-Kutta routine
dt = t(i+1)-t(i)
! !Do the Runge-Kutta step
  x1 = x(:,i)
call rk(x1,w,dt,xf,dJ,u1)
  x(:,i+1) = xf
        u(:,i) = u1
J = J + dJ
end do
    u(:,nt) = u(:,nt-1)

    call terminal(x(:,nt),u(:,nt),t(nt),dJ)
    J = J + dJ

call general(x,u,nt,t,C)
    J = J + C
```

```
    J = J*alpha

    return
end subroutine
!**********************************************************************
subroutine rk(x1,w,dt,xf,dJ,u1)
    use mexinterface
implicit none

! !Input variables
real*8  :: x1(NX), w(NW), dt

! !Output variables
real*8  :: xf(NX), dJ, u1(NU)

! !Working variables
real*8 :: x2(NX), x3(NX), x4(NX)
real*8 :: xdot1(NX), xdot2(NX), xdot3(NX), xdot4(NX)
real*8 :: J1, J2, J3, J4, u2(NU), u3(NU), u4(NU), L1, L2, L3, L4

! !Find xdot and L (Jdot) at each sample point
call f(x1,w,xdot1,u1)
call cost(x1,u1,L1)

x2 = x1 + xdot1 * (dt/2.)
    call f(x2,w,xdot2,u2)
call cost(x2,u2,L2)

x3 = x1 + xdot2 * (dt/2.)
call f(x3,w,xdot3,u3)
call cost(x3,u3,L3)

x4 = x1 + xdot3 * dt
call f(x4,w,xdot4,u4)
call cost(x4,u4,L4)

! !Find the final point, and increment in cost
xf = x1 + (xdot1 + 2.*xdot2 + 2.*xdot3 + xdot4) * (dt/6.)
dJ = (L1 + 2.*L2 + 2.*L3 + L4) * (dt/6.)

return
end subroutine
```

A.6 parameters Code

```
% parameters.m
global nx nu ny nInputs nNeurons neuronType

% Parameters of dynamic system and controller
nx = 2;
nu = 1;
ny = 2;

% Parameters of neural net controller
nInputs = ny;
nNeurons = [3 7 5 nu];
neuronType = {'atan';'atan';'atan';'atan'};
```

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B

Dynamics and Cost Function Codes for the Example Applications

This appendix contains the `dynamics.F90` codes with the `cost`, `general`, `terminal`, and `dynamics` functions for the linearized aircraft dynamics problem, the double integrator problem, and the orbit-raising problem. It should be noted that all of the codes in this appendix are written in FORTRAN. Also, it should be noted that the code implementations for the optimal solution comparisons are not given in this thesis; however, they are easily reproduced. For a description of the example applications associated with the three codes, see Chapter 5.

B.1 Linearized Aircraft Dynamics Problem

```
# if NY-4
error: Should have NY = 4
# endif
# if NX-5
error: Should have NX = 5
# endif
# if NU-1
error: Should have NU = 1
# endif

!This file contains the following:
! cost
! general
! terminal
! f dynamics of the linearized aircraft dynamics problem
!
```

```
subroutine cost(x,u,L)

    implicit none
    real(8) :: x(NX), u(NU), L
    real(8), parameter :: Q = 1.0D0, R =1.0D8
!   intrinsic sin, cos, sqrt

    L = Q*(x(5)**2) + R*(u(1)**2)

    return
end subroutine

subroutine general(x,u,nt,t,C)

    implicit none

!   !Input variables
    integer :: nt
    real(8) :: x(NX,nt), u(NU,nt), t(nt)

!   !Output variables
    real(8) :: C

    C = 0

    return
end subroutine

subroutine terminal(x,u,t,dJ)

    implicit none
    real*8 :: x(NX), u(NU), t, dJ

    dJ = 0

return
end subroutine

!***********************************************************************

subroutine f(x,w,x_dot,u)

    use mexinterface

    implicit none
```

```
    real(8) :: x(NX), w(NW), x_dot(NX), y(NY), t
    real(8) :: u(NU)
    real(8), parameter :: PI = 3.14159265D0

    intrinsic sin, cos, atan2

! Control input to neural network:
! This problem does not use a neural network. The states are
! defined as:
! x(1) is the velocity in the x direction in the body frame
! x(2) is the velocity in the z direction in the body frame
! x(3) is the pitch rate
! x(4) is the pitch angle
! x(5) is the altitude error for the aircraft

! Control inputs:
u(1) = - w(1)*x(1) - w(2)*x(2) - w(3)*x(3) - w(4)*x(4) - w(5)*x(5)
! u(1) represents the elevator angle deflection. The w vector
! contains the LQR gains.

! Dynamics
! Outputs:
x_dot(1) = -0.00643D0*x(1)+0.0263D0*x(2)-32.2D0*x(4)
x_dot(2) = -0.0941D0*x(1)-0.624D0*x(2)+820.0D0*x(3)-32.7D0*u(1)
x_dot(3) = -0.000222D0*x(1)-0.00153D0*x(2)-0.668D0*x(3)-2.08D0*u(1)
x_dot(4) = x(3)
x_dot(5) = -x(2)+830.0D0*x(4)

    return
end subroutine
```

B.2 Double Integrator Problem

```
# if NY-2
error: Should have NY = 2
# endif
# if NX-2
error: Should have NX = 2
# endif
# if NU-1
error: Should have NU = 1
# endif

!This file contains the following:
```

```
! cost
! general
! terminal
! f dynamics of the double integrator problem
!

subroutine cost(x,u,L)

    implicit none
    real(8) :: x(NX), u(NU), L
!   intrinsic sin, cos, sqrt

L = x(1)**2 + x(2)**2

    return
end subroutine

subroutine general(x,u,nt,t,C)

    implicit none

!   !Input variables
    integer :: nt
    real(8) :: x(NX,nt), u(NU,nt), t(nt)

!   !Output variables
    real(8) :: C

    C = 0

    return
end subroutine

subroutine terminal(x,u,t,dJ)

    implicit none
    real*8 :: x(NX), u(NU), t, dJ

    dJ = 0

return
end subroutine

!*****************************************************************
```

```
subroutine f(x,w,x_dot,u)

    use mexinterface

    implicit none
    real(8) :: x(NX), w(NW), x_dot(NX), y(NY)
    real(8) :: u(NU)
    real(8), parameter :: PI = 3.14159265D0

! Control input to neural network:
y(1) = x(1)  !x(1) is the position
y(2) = x(2)  !x(2) is the velocity
! For the trajectory optimization problem, the previous two lines
! were replaced with y(1) = x(3) where x(3) was an additional state
! that represented the time remaining so that the time remaining
! was the only input to the neural network.

! Neural network:
call nn(y,u,w)  !u is the acceleration

! Scaling control inputs:
u(1) = u(1)*2/PI !u(1) is the thrust

! Dynamics
! Outputs:
x_dot(1) = x(2)
x_dot(2) = u(1)
! For the trajectory optimization problem, the addition of the third
! state required the addition of a third state equation, in which
! x_dot(3) = -1.

    return
end subroutine
```

B.3 Orbit-Raising Problem

```
# if NY-4
error: Should have NY = 4
# endif
# if NX-4
error: Should have NX = 4
# endif
# if NU-2
error: Should have NU = 2
```

```
# endif

!This file contains the following:
! cost
! general
! terminal
! f dynamics of the orbit-raising problem
!

subroutine cost(x,u,L)

    implicit none
    real(8) :: x(NX), u(NU), L
!   intrinsic sin, cos, sqrt

    L = 0

    return
end subroutine

subroutine general(x,u,nt,t,C)

    implicit none

!   !Input variables
    integer :: nt
    real(8) :: x(NX,nt), u(NU,nt), t(nt)

!   !Output variables
    real(8) :: C

    C = 0

    return
end subroutine

subroutine terminal(x,u,t,dJ)

    implicit none
    real*8 :: x(NX), u(NU), t, dJ
    real*8 :: mur
    real*8, parameter :: alpha = 1000.0D0, mu = 1.0D0, ri = 1.0D0

    mur = sqrt(mu/x(1))
    dJ = -x(1)/ri + alpha*((x(2)/mur)**2 + ((x(3)/mur) - 1)**2)
```

```
! For the multiple initial condition case, in which the initial
! radius differed for every initial condition, the general cost
! function was used instead of the terminal function to be able
! to use the initial radius for each initial condition in the
! cost calculation.

return
end subroutine

!*****************************************************************

subroutine f(x,w,x_dot,u)

    use mexinterface

    implicit none
    real(8) :: x(NX), w(NW), x_dot(NX), y(NY), t
    real(8) :: u(NU)
    real(8) :: thrustfract, phi
    real(8), parameter :: PI = 3.14159265D0
    real(8), parameter :: Thrust = 0.1405D0, mu = 1.0D0
    real(8), parameter :: m0 =  0.7006D0, m_dot = 0.07485D0

    intrinsic sin, cos, atan2

! Control input to neural network:
y(1) = x(1)  !x(1) is the radial distance from the attracting center
y(2) = x(2)  !x(2) is the radial component of the velocity
y(3) = x(3)  !x(3) is the tangential component of the velocity
y(4) = x(4)  !x(4) is the time remaining
! For the trajectory optimization problem, the previous four lines
! were replaced with y(1) = x(4) so that the time remaining was the
! only input to the neural network.

! Neural network:
call nn(y,u,w)  !u is the thrust direction angle

! Scaling control inputs:
u(1) = u(1)*2/PI  !u(1) is cos(phi)
u(2) = u(2)*2/PI  !u(2) is sin(phi)
phi = atan2(u(2),u(1))  !phi is the thrust angle

! Dynamics
! Outputs:
```

```
thrustfract = Thrust/(m_dot*x(4) + m0)
x_dot(1) = x(2)
x_dot(2) = (x(3)**2)/x(1) - mu/(x(1)**2) + thrustfract*sin(phi)
x_dot(3) = -(x(2)*x(3))/x(1) + thrustfract*cos(phi)
x_dot(4) = -1

return
end subroutine
```

Bibliography

[1] W. Cowper and O. Holden, "Cowper, 168," in *The Sacred Harp*, Sacred Harp Publishing Company, Inc., 1991 revision ed., 1991.

[2] B. Robinson, *The Best Christmas Pageant Ever*. New York: HarperTrophy, 1972.

[3] R. Bellman and R. Kalaba, *Dynamic Programming and Modern Control Theory*. New York: Academic Press, 1965.

[4] A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Philadelphia: Society for Industrial and Applied Mathematics, 2000.

[5] A. E. Bryson, Jr. and Y.-C. Ho, *Applied Optimal Control: Optimization, Estimation, and Control*. Washington: Hemisphere Publishing Corporation, revised printing ed., 1975.

[6] D. E. Kirk, *Optimal Control Theory: An Introduction*. Mineola: Dover Publications, Inc., 1998.

[7] J. How, *16.323 Principles of Optimal Control*, Spring 2008. `http://ocw.mit.edu/OcwWeb/Aeronautics-and-Astronautics/16-323Spring-2008/CourseHome/index.htm` .

[8] J. T. Betts, *Practical Methods for Optimal Control Using Nonlinear Programming*. Advances in Design and Control, Philadelphia: Society for Industrial and Applied Mathematics, 2001.

[9] L. B. Rall, *Automatic Differentiation: Techniques and Applications*. No. 120 in Lecture Notes in Computer Science, Berlin: Springer-Verlag, 1981.

[10] *The Tapenade Tutorial*, March 2009. `http://www-sop.inria.fr/tropics/tapenade.html` .

[11] F. Courty, A. Dervieux, B. Koobus, and L. Hascoët, "Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation," *Optimization Methods and Software*, vol. 18, no. 5, pp. 615–627, 2003.

[12] L. Hascoët, M. Vázquez, and A. Dervieux, "Automatic differentiation for optimum design applied to sonic boom reduction," in *Proceedings of the International Conference on Computational Science and its Applications, ICCSA '03, Montreal, Canada* (V. K. et al., ed.), pp. 85–94, LNCS 2668, Springer, 2003.

[13] M. T. Hagan and H. B. Demuth, "Neural networks for control," in *Proceedings of the 1999 American Control Conference*, vol. 3, (San Diego, California), pp. 1642–1656, American Automatic Control Council, American Automatic Control Council, 1999.

[14] S. Haykin, *Neural Networks: A Comprehensive Foundation*. Upper Saddle River: Prentice Hall, 2nd ed., 1999.

[15] K. Mehrotra, C. K. Mohan, and S. Ranka, *Elements of Artificial Neural Networks*. Cambridge, Massachusetts: MIT Press, 1997.

[16] D. H. Nguyen and B. Widrow, "Neural networks for self-learning control systems," *IEEE Control Systems Magazine*, vol. 10, pp. 18–23, April 1990.

[17] D. P. Bertsekas, M. L. Homer, D. A. Logan, S. D. Patek, and N. R. Sandell, "Missile defense and interceptor allocation by neuro-dynamic programming," *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*, vol. 30, pp. 42–51, January 2000.

[18] H.-L. Choi, Y. Park, H.-G. Lee, and M.-J. Tahk, "A three-dimensional differential game missile guidance law using neural networks," in *AIAA Guidance, Navigation, and Control Conference and Exhibit*, (Montreal, Canada), Aug. 6-9, 2001.

[19] P. A. Johnson, "Numerical solution methods for differential game problems," Master's thesis, Massachusetts Institute of Technology, 2009.

[20] P. He and S. Jagannathan, "Reinforcement learning neural-network-based controller for nonlinear discrete-time systems with input constraints," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 37, pp. 425–436, April 2007.

[21] W. Thomas Miller, III, R. S. Sutton, and P. J. Werbos, eds., *Neural Networks for Control*. Neural Network Modeling and Connectionism, Cambridge, Massachusetts: The MIT Press, 1990.

[22] D. P. Bertsekas, *Nonlinear Programming*. Belmont: Athena Scientific, second ed., 1999.

[23] *Optimization Toolbox User's Guide*. The MathWorks, Inc., version 2 ed., 2008.

[24] L. Hascoët and M. Araya-Polo, "The adjoint data-flow analyses: Formalization, properties, and applications," in *Automatic Differentiation: Applications, Theory, and Implementations* (M. Bücker, G. Corliss, P. Hovland, U. Naumann, and

B. Norris, eds.), no. 50 in Lecture Notes in Computational Science and Engineering, pp. 135–146, Berlin: Springer, 2006.

[25] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*. Upper Saddle River: Pearson Prentice Hall, fifth ed., 2006.

[26] *Control System Toolbox 8 Getting Started*. The MathWorks, Inc., 2009.

[27] *Matlab 7 Mathematics*. The MathWorks, Inc., 2009.

[28] *The Princess Bride*. DVD. MGM/UA Home Entertainment, 2000.