The Dissertation Committee for David Marcelo Garza
certifies that this is the approved version of the following dissertation:

Application of Automatic Differentiation to Trajectory Optimization via Direct Multiple Shooting

Committee:

Wallace T. Fowler, Supervisor

David G. Hull

Robert H. Bishop

Leon S. Lasdon

Robert D. Braun

Application of Automatic Differentiation to Trajectory Optimization via Direct Multiple Shooting

by

David Marcelo Garza, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2003

This is dedicated to all the people who work to make the Austin music scene the treasure it is.

Acknowledgments

I would like to thank my advisor Dr. Wallace Fowler for his support and encouragement throughout this project, as well as the first pointer to the GSRP program. Thanks are also extended to the other members of my dissertation committee for their guidance, advice, and support: Dr. Leon Lasdon, Dr. David Hull, Dr. Robert Bishop, and Dr. Robert Braun.

Several people at Langley Research Center helped to make this work possible. Thanks are extended to the Vehicle Analysis Branch for taking me in and supporting my fellowship and providing great insight into the design and anaysis of launch vehicles. In particular I would like to thank David Way, Scott Striepe, Paul Tartabini, Prasun Desai, Walt Engelund, Bandu Pamadi, Garry Qualls, Ted Talay, Roger Lepsch, Mark McMillin, and Chuck Eldred. Thanks to Lloyd Evans and Rafaela Schwan in the Office of Education for helping me get in Langley and assisting with my fellowship. Special thanks go to Craig Hunter and Karen Deere of Langley's Configuration Aerodynamics Branch for providing the initial opportunity for me to work at Langley.

I would also like to thank my family for their support and encouragement. In particular my parents, Daniel and Mary Garza, brother and sister-in-law Peter and Megumi Garza, grandparents Anna Garza, James Boyce, and Marguerite Boyce, and my uncle and aunt, Romeo Escobedo and Edna Garza-

Escobedo. Special thanks go to my uncle and aunt James and Linda Boyce for the family contact during my summers in Virginia.

I have made many friends in the Aerospace Department and Center for Space Research at UT, and they have helped make the years of work more fun and stimulating. In particular I would like to thank Calina Seybold, Olivier DuBois-Matra, and Michelle Bailey, George Hindman and Lila Glaser for their support, friendship, and encouragement.

Thanks are also directed to the friends I've made at a couple of welcome diversions: SXSW and the P.F. Flyer. At SXSW I would especially like to thank Alissa McCain, Amin Sims, Rachel McGruder, Amanda Bowman, Yvonne Valdez, Ron Suman, Jamie Jameson, and Leslie Uppinghouse. On the Flyer I would like to thank the skipper, Brian Diebler, as well as the rest of the crew including Kim Diebler, Karen Davieds, Chauncy Wu, and Danny Wright, for taking me on as a novice sailor.

Most of this research was funded through NASA Graduate Student Researchers Program grant NGT-1-52147.

Application of Automatic Differentiation to Trajectory Optimization via Direct Multiple Shooting

Publication No. _____

David Marcelo Garza, Ph.D.
The University of Texas at Austin, 2003

Supervisor: Wallace T. Fowler

Automatic differentiation, also called computational differentiation and algorithmic differentiation, is the process of computing the derivatives or Taylor series of functions from the computer source code implementing the functions. To date, general-purpose trajectory optimization codes have relied on finite-differencing to compute the gradients needed by the nonlinear programming (NLP) algorithms within the codes. These codes typically support the selection of an arbitrary objective and constraint set from a library of a few hundred output variables. The use of automatic differentiation in these trajectory optimization programs can provide objective and constraint gradients to the same precision as the underlying functions without requiring the generation of hundreds of analytic derivative expressions by hand or via symbolic algebra packages. This work combines automatic differentiation with a direct multiple shooting method and uses the resulting method to solve a pair of example problems. The first is the well-known lunar launch problem, while the

second is a launch vehicle ascent problem similar in complexity to that which would be computed by a program such as the Program to Optimize Simulated Trajectories (POST) for use in vehicle design studies. Results include comparisons of convergence behavior of the NLP problem and solution accuracy. Tests comparing the use of Euler angles versus quaternion elements as control variables demonstrate the versatility of automatic differentiation. For loose convergence levels automatic differentiation provided faster convergence than finite differencing on the launcher ascent problem. For tight accuracy requirements, automatic differentiation resulted in fewer major iterations on the lunar launch problem.

Table of Contents

Chapter 1

Introduction

For both surface-to-orbit and in-space vehicles trajectory optimization is performed from the earliest conceptual design phase through the operation of the vehicle. A wide variety of computational approaches have been developed for the off-line computation of optimal control profiles for the vehicles, and many general-purpose trajectory design programs have been developed over the years for this job. These programs include Program to Optimize Simulated Trajectories (POST), General Trajectory Simulation (GTS), Optimal Trajectories by Implicit Simulation (OTIS), and Graphical Environment for Simulation and Optimization/Aerospace Trajectory Optimization Software (GESOP/ASTOS). These programs typically feature a "direct" solution method of some sort, where a basic optimal control problem is transformed into a parameter optimization problem via discretization of the states and controls and then the parameter optimization problem is solved using an existing parameter optimization code. This direct solver is coupled with a large library of various gravity, atmosphere, and vehicle models. The combination of direct solution method with the preexisting dynamical models allows an analyst to solve simple problems with no additional programming, and more complex

problems with a limited amount of extra programming focused on the special models required by the problem.

In 1982, Wolfe observed that gradient evaluation is typically about 1.5 times as computationally expensive as the underlying function, and rarely greater than 2 times as expensive[59]. Griewank later showed analytically the factor of 1.5 could be replaced with an upper bound of 5[30], and much of the research in automatic differentiation during the last decade has focused on reaching this bound. The concept of automatic differentiation has been around for a significant period of time, at least since 1964 when Wengert published his paper on what has come to be known as the "forward mode" of automatic differentiation [57].

In spite of over 30 years of research in automatic differentiation, little use has been made of it in solving major optimal control problems. While codes such as POST and OTIS have been used for many flight projects over the years, and their ease of use has been improved by the adoption of collocation and multiple-shooting methods in the newer programs, they still incorporate approximate derivative computations. On the other hand the primary focus on automatic differentiation in optimal control has been its potential to relieve the tedium of deriving the Euler-Lagrange equations, which require several different derivative expressions, for each problem. This work demonstrates the usefulness of automatic differentiation in direct trajectory optimization, in particular for general-purpose trajectory optimization codes such as those mentioned above. A direct multiple shooting solution method, similar to the

one used in TROPIC and presented in [19], is mated with a set of models to provide a variety of physical models and control variables. The models are implemented using a set of automatic differentiation routines to provide the machine precision derivatives required by nonlinear programming (NLP) codes.

Two problems are used to demonstrate the usefulness and versatility of automatic differentiation: a simple lunar launch problem described in Bryson and Ho [10], and a single-stage-to-orbit (SSTO) launch vehicle ascent problem. The lunar launch problem possesses an analytical solution which can be used to demonstrate the impact of derivative computation on the solution error, while the SSTO problem illustrates how the versatility of automatic differentiation matches that of finite differencing by solving the same problem using two different sets of control variables without requiring any special derivative expressions or procedures to select the derivatives computed. The SSTO problem is a typical use of a general-purpose trajectory optimization code– ascent trajectory optimization for a launch vehicle. While the example is a typical preliminary-level design problem, there would be little change for an ascent problem for an existing vehicle such as the Space Shuttle or expendable launch vehicle (ELV).

The dissertation is divided into six chapters and an appendix. Chapter 2 reviews the direct multiple shooting approach used, while Chapter 3 describes automatic differentiation with some examples. Chapter 4 presents the lunar launch problem and solution results, Chapter 5 the SSTO problem and results,

and Chapter 6 the conclusions and recommendations for further work.

Chapter 2

Direct Multiple Shooting for Trajectory Optimization Problems

This chapter presents the discretization used to solve the two optimal control problems presented later. The first section provides some basic background on optimal control problems and reviews some of the methods used for solving trajectory optimization problems. Next, the focus switches to the discretization used in this research, including the basic state and control discretization, the adjoint discretization, and the order of accuracy. This section also describes the mesh refinement procedure used. The final section briefly discusses the sparse NLP resulting from the solution method and SNOPT, the NLP solver used.

2.1 Numerical Methods for Optimal Control Problems

A free-final time optimal control problem, in Mayer form, seeks the control function $u(t) \in \Re^m$ which minimizes the performance index

$$J = \phi(t_f, x(t_f)) \tag{2.1}$$

subject to the system dynamics

$$\dot{x}(t) = f(t, x(t), u(t)) \tag{2.2}$$

the initial conditions

$$x(t_0) = x_0 \tag{2.3}$$

and the final constraints

$$\psi(t_f, x(t_f)) = 0 \tag{2.4}$$

where $x(t) \in \Re^n$ is the state, $\phi : \Re \times \Re^n \times \Re^m \mapsto \Re$, $f : \Re \times \Re^n \times \Re^m \mapsto \Re^n$, $\psi : \Re \times \Re^n \mapsto \Re^p$, $t \in [t_0, t_f]$, t_0 is the initial time, and t_f is the final time.

The analytical approach to solving the problem is to append the boundary conditions and differential equations to the performance index using Lagrange multipliers, $\nu \in \Re^p$ and $\lambda(t) \in \Re^n$, to form the augmented performance index

$$\tilde{J} = G + \int_{t_0}^{t_f} H \, dt \tag{2.5}$$

where H is the Hamiltonian, $H = \lambda^T f$, and G is the auxiliary function $G = \phi + \nu^T \psi$. Taking the first variation of the augmented performance index leads to the adjoint, or costate, differential equation

$$\dot{\lambda} = -\left(\frac{\partial H}{\partial x}\right)^T \tag{2.6}$$

adjoint terminal conditions

$$\lambda(t_f) = \left(\frac{\partial G}{\partial x_f}\right)^T \tag{2.7}$$

control optimality condition

$$\frac{\partial H}{\partial u} = 0 \tag{2.8}$$

and transversality condition

$$\frac{\partial G}{\partial t_f} = -H_f \tag{2.9}$$

in addition to the state equation, and initial and terminal conditions. Equations 2.2–2.9 are known as the Euler-Lagrange equations and they form a two-point boundary value problem (BVP) which determines the solution. In general the Euler-Lagrange equations have no analytical solution and the optimization problem must be solved numerically.

Many different schemes to numerically solve optimal control problems have been investigated over the years, but they can be classified into two broad categories: indirect methods and direct methods. Indirect methods solve the original problem by solving the Euler-Lagrange equations numerically for the state, adjoint, and control histories. Direct methods transform the original optimization problem into a parameter optimization problem which is then solved using a parameter optimization code. While the distinction has been made for over 30 years, a variety of terms have been used to describe the different direct methods. Bryson and Ho called direct methods "mathematical programming methods"[10]. Cannon, et. al. used the terms "direct tran-

scription" and "alternate transcription"[11], while Kraft used the terms "direct shooting" and "direct collocation"[43]. Enright [19] and Betts [7] reintroduced the term "direct transcription" to describe methods which create large NLP problems, such as from direct collocation.

Several types of indirect methods exist. Most of these are based on solving the Euler-Lagrange system using a general BVP solution technique, such as single shooting, multiple shooting, or collocation. Both single- and multiple-shooting are initial value methods–the differential equations are integrated using an algorithm for initial value problems and the boundary conditions are met by guessing and refining values for the unknown initial conditions. For details on shooting methods see Ascher, et. al. [4], or Stoer and Bulirsch[55]. The code BNDSCO, by Oberle and Grimm, is an indirect multiple shooting method[46].

In collocation, an approximating function represents the true state, control, and adjoint histories; and the derivative of the approximation is required to match the differential equations at specified points, called collocation points. The most common choice for an approximating function is a piecewise polynomial using a Hermite cubic basis (Dickmanns and Well [15]), a B-spline basis (Ascher, et. al. [2]), or a monomial basis (Bader and Ascher [5]. In pseudospectral collocation methods, the approximation is a high-order global Lagrange polynomial. However, these methods have not seen the same amount of use in optimal control applications as the piecewise polynomial-based methods. In fact, most applications of pseudospectral methods have been partial

differential equations (PDE's). For more on pseudospectral methods see Gottlieb [26].

Gradient methods are a different type of indirect method, designed to take advantage of the special structure of the Euler-Lagrange equations. However, they typically feature slow convergence near an optimum (see Gottlieb [27], Miele [45], and Bryson and Ho [10]). Jacobson investigated second order gradient methods to remedy the problem[40].

Indirect methods typically suffer from one or more of the following problems[19, 8]:

- They exhibit extreme sensitivity to initial guesses
- They require an *a priori* guess of constrained, unconstrained, and singular solution arcs when path constraints or linear controls are present
- They require an initial guess for the Lagrange multipliers, quantities which typically have little or no physical interpretation
- They require derivation of a Hamiltonian and several different derivative expressions

Direct methods have been explored, in part, as a means to avoid the disadvantages of indirect methods, and as with indirect methods many forms of direct methods have been developed. Most approaches can be viewed as adaptations of a BVP solution method, including direct single shooting, direct multiple

shooting, and direct collocation. In direct shooting, the discretized control history, initial time, final time, and initial conditions make up the independent variables. The terminal conditions are the constraints and with each iteration of the NLP solver the trajectory is integrated and the boundary conditions evaluated. The alternative transcription of Canon, et. al. [11], and the second transcription of Polak [48] are examples of early direct shooting methods. In these methods, the control and state were discretized at the same points in time. A variation of this approach introduces an approximation of the control which has many fewer variables than if the control was discretized at each integration step. The trajectory optimization code POST[49] operates on this principle allowing tables with linear and cubic spline interpolation, or polynomials up to degree three to represent the control. Table values and polynomial coefficients become part of the independent variable set. Direct multiple shooting splits the trajectory into segments with their own initial conditions and which are integrated separately. Additional constraints are introduced enforcing continuity of the state from one segment to another. The direct transcription method in Canon, et. al. [11] and the first transcription method in Polak [48] are examples of early direct multiple shooting methods. Later examples include Enright [19] and Betts and Huffman [9]. The "parallel shooting" method of Betts and Huffman [9] is a direct multiple shooting approach adapted to take advantage of parallel processing. As with single shooting, the control can be discretized at the same points as the state, or discretized separately.

Direct collocation is similar to collocation in the context of solving a BVP. The state and control histories are represented using some approximating function, and the collocation conditions are introduced as NLP constraints together with the initial and terminal conditions. Hargraves and Paris used the collocation scheme of Dickmanns and Well [36] as the basis for their direct collocation method. Herman and Conway investigated higher-order approximation and quadrature in a direct collocation scheme [38] as a way to improve accuracy. The programs SOCKS [8] and OTIS [37] both use direct collocation. A direct equivalent of pseudospectral collocation has been investigated recently by Elnagar, et. al. [17], and Elnagar and Razzaghi[18]. Fahroo and Ross applied a pseudospectral method to spacecraft trajectory optimization [20, 21].

Reviews of the names, classifications, and the lineage of different methods can be found in Enright [19] and Betts [7].

2.2 A Direct Multiple Shooting Method

2.2.1 Discretization

Consider the optimal control problem where the goal is to find the control $u(t) \in \Re^m$ which minimizes a performance index

$$J = \phi\left(t_f, x\left(t_f\right)\right) \tag{2.10}$$

subject to the differential equations

$$\dot{x}(t) = f(t, x(t), u(t)) \tag{2.11}$$

path constraints

$$C(t, x(t), u(t)) = 0 \tag{2.12}$$

and boundary conditions

$$\psi_0(t_0, x(t_0)) = 0 \tag{2.13}$$

$$\psi_f(t_f, x(t_f)) = 0 \tag{2.14}$$

where $x(t) \in \Re^n$ is the state, $\phi : \Re \times \Re^n \times \Re^m \mapsto \Re$, $f : \Re \times \Re^n \times \Re^m \mapsto \Re^n$, $\psi_0 : \Re \times \Re^n \mapsto \Re^{p_0}$, $\psi_f : \Re \times \Re^n \mapsto \Re^{p_f}$, $t \in [t_0, t_f]$, t_0 is the initial time, and t_f is the final time.

Partition and normalize time with the transformation

$$t_i = (1 - \tau_i)t_0 + \tau_i t_f \tag{2.15}$$

$$h_i = (\tau_{i+1} - \tau_i)(t_f - t_0) \tag{2.16}$$

where $\tau_i \in [0, 1]$, $\tau_i < \tau_{i+1}$, $i = 0, \ldots, N$.

A fourth-order Runge-Kutta discretization of the state differential equa-

tion on these nodes gives

$$t_{i,1} = t_i \qquad y_{i+1,1} = x_i \qquad K_{i,1} = h_i f(t_{i,1}, y_{i,1}, v_{i,1}) \tag{2.17}$$

$$t_{i,2} = t_i + \frac{1}{2}h_i \qquad y_{i,2} = x_i + \frac{1}{2}K_{i,1} \qquad K_{i,2} = h_i f(t_{i,2}, y_{i,2}, v_{i,2}) \tag{2.18}$$

$$t_{i,3} = t_i + \frac{1}{2}h_i \qquad y_{i,3} = x_i + \frac{1}{2}K_{i,2} \qquad K_{i,3} = h_i f(t_{i,3}, y_{i,3}, v_{i,3}) \tag{2.19}$$

$$t_{i,4} = t_i + h_i \qquad y_{i+1,4} = x_i + K_{i,3} \qquad K_{i,4} = h_i f(t_{i,4}, y_{i,4}, v_{i,4}) \tag{2.20}$$

$$x_{i+1} = x_i + \frac{1}{6}K_{i,1} + \frac{1}{3}K_{i,2} + \frac{1}{3}K_{i,2} + \frac{1}{6}K_{i,4} \tag{2.21}$$

where $x(t_i) = x_i$, and $u(t_i) = u_i$. Equations 2.17-2.20 are frequently called "stages," and the term will be used in this work. Eliminating the first stage and rearranging the remaining equations gives the "defect" functions

$$\Gamma_{i,2} = y_{i,2} - x_i - \frac{1}{2}K_{i,1} \tag{2.22}$$

$$\Gamma_{i,3} = y_{i,3} - x_i - \frac{1}{2}K_{i,2} \tag{2.23}$$

$$\Gamma_{i,4} = y_{i,4} - x_i - K_{i,3} \tag{2.24}$$

$$\Delta_i = x_{i+1} - x_i - \frac{1}{6}K_{i,1} - \frac{1}{3}K_{i,2} - \frac{1}{3}K_{i,2} - \frac{1}{6}K_{i,4} \tag{2.25}$$

The boundary conditions become

$$\psi_0(t_0, x_0,) = 0 \tag{2.26}$$

$$\psi_f(t_f, x_N,) = 0 \tag{2.27}$$

while the path constraints become

$$C_i = C\left(t_i, x_i, u_i\right) \qquad = 0 \tag{2.28}$$

$$C_{i,j} = C\left(t_{i,j}, y_{i,j}, v_{i,j}\right) = 0 \tag{2.29}$$

Combing the defect and path constraints gives the set of interval constraints w_i,

$$w_i = \begin{bmatrix} C_{i,1} \\ \Gamma_{i,2} \\ C_{i,2} \\ \Gamma_{i,3} \\ C_{i,3} \\ \Gamma_{i,4} \\ C_{i,4} \\ \Delta_i \end{bmatrix} = 0 \tag{2.30}$$

which are enforced at each time interval. The state and control variables for each integration interval form a set of variables, z_i, for each interval

$$z_i = \begin{bmatrix} x_i \\ u_i \\ y_{i,2} \\ v_{i,2} \\ y_{i,3} \\ v_{i,3} \\ y_{i,4} \\ v_{i,4} \end{bmatrix} \tag{2.31}$$

The nonlinear programming variables are the variables for the intervals, and the initial and final times. The nonlinear programming constraints are the

interval constraints and the boundary conditions.

$$X = \begin{bmatrix} z_0 \\ \vdots \\ z_{N-1} \\ x_N \\ u_N \\ t_0 \\ t_f \end{bmatrix} \quad F(X) = \begin{bmatrix} \psi_0\left(t_0, x_0\right) \\ w_0 \\ \vdots \\ w_{N-1} \\ C_N \\ \psi_f\left(t_f, x_N\right) \end{bmatrix} \tag{2.32}$$

2.2.2 Discrete Adjoint

A discrete equivalent of the Euler-Lagrange equations in optimal control theory can be derived from the resulting parameter optimization problem. Here they will be derived assuming no path constraints are present. First adjoin the constraints to the performance index. Let $\eta_{i,j}$ and λ_i be the multipliers on $\Gamma_{i,j}$ and Δ_i, respectively. Taking the derivative of the augmented performance index with respect to x_i and $y_{i,j}$, and setting the expressions equal to zero gives the equations

$$\left(\frac{\partial \Delta_{i-1}}{\partial x_i}\right)^T \lambda_{i-1} + \left(\frac{\partial \Gamma_{i,2}}{\partial x_i}\right)^T \eta_{i,2} + \left(\frac{\partial \Gamma_{i,3}}{\partial x_i}\right)^T \eta_{i,3} + \left(\frac{\partial \Gamma_{i,4}}{\partial x_i}\right)^T \eta_{i,4} + \left(\frac{\partial \Delta_i}{\partial x_i}\right)^T \lambda_i = 0 \tag{2.33}$$

$$\left(\frac{\partial \Gamma_{i,2}}{\partial y_{i,2}}\right)^T \eta_{i,2} + \left(\frac{\partial \Delta_i}{\partial y_{i,2}}\right)^T \lambda_i = 0 \tag{2.34}$$

$$\left(\frac{\partial \Gamma_{i,3}}{\partial y_{i,3}}\right)^T \eta_{i,3} + \left(\frac{\partial \Delta_i}{\partial y_{i,3}}\right)^T \lambda_i = 0 \tag{2.35}$$

$$\left(\frac{\partial \Gamma_{i,4}}{\partial y_{i,4}}\right)^T \eta_{i,4} + \left(\frac{\partial \Delta_i}{\partial y_{i,4}}\right)^T \lambda_i = 0. \tag{2.36}$$

After some algebra these equations give the system

$$\gamma_{i,4} = \lambda_i \qquad \kappa_{i,4} = h\left(\frac{\partial f}{\partial x}\right)^T_{i,4} \gamma_{i,4} \tag{2.37}$$

$$\gamma_{i,3} = \lambda_i + \frac{1}{2}\kappa_{i,4} \qquad \kappa_{i,3} = h\left(\frac{\partial f}{\partial x}\right)^T_{i,3} \gamma_{i,3} \tag{2.38}$$

$$\gamma_{i,2} = \lambda_i + \frac{1}{2}\kappa_{i,3} \qquad \kappa_{i,2} = h\left(\frac{\partial f}{\partial x}\right)^T_{i,2} \gamma_{i,2} \tag{2.39}$$

$$\gamma_{i,1} = \lambda_i + \kappa_{i,2} \qquad \kappa_{i,1} = h\left(\frac{\partial f}{\partial x}\right)^T_{i,1} \gamma_{i,1} \tag{2.40}$$

$$\lambda_{i-1} = \lambda_i + \frac{1}{6}\kappa_{i,4} + \frac{1}{3}\kappa_{i,3} + \frac{1}{3}\kappa_{i,2} + \frac{1}{6}\kappa_{i,1}, \tag{2.41}$$

where

$$\kappa_{i,4} = 6\eta_{i,4} \tag{2.42}$$

$$\kappa_{i,3} = 3\eta_{i,3} \tag{2.43}$$

$$\kappa_{i,2} = 3\eta_{i,2}. \tag{2.44}$$

The interval defect multipliers, λ_i, are approximations to the adjoints in the Euler-Lagrange equations, and the stage defect multipliers, $\eta_{i,j}$, are scaled stage derivatives for a Runge-Kutta-type backward integration of the adjoints.

The discrete equivalents of the boundary conditions on $\lambda(t)$ and the transversality conditions can be derived in a similar manner, as shown in Enright[19]. Hager, in [34], has shown equations 2.37-2.41 to be a fourth-order accurate discretization of the continuous adjoint equation

$$\dot{\lambda} = -H_x^T. \tag{2.45}$$

A similar procedure with respect to the discretized controls, u_i and $v_{i,j}$, yields the system

$$0 = h \left(\frac{\partial f}{\partial u} \right)_{i,4}^T \gamma_{i,4} \tag{2.46}$$

$$0 = h \left(\frac{\partial f}{\partial u} \right)_{i,3}^T \gamma_{i,3} \tag{2.47}$$

$$0 = h \left(\frac{\partial f}{\partial u} \right)_{i,2}^T \gamma_{i,2} \tag{2.48}$$

$$0 = h \left(\frac{\partial f}{\partial u} \right)_{i,1}^T \gamma_{i,1}. \tag{2.49}$$

Hager[33] and Schwartz[52] demonstrated this was a third-order accurate representation of the continuous control condition

$$H_u = 0, \tag{2.50}$$

however, in the first example problem in this work the control is fourth-order accurate except at the final node in the trajectory.

2.2.3 Mesh Refinement

Mesh refinement is an integral part of many schemes for solving boundary value problems and optimal control problems, for examples see Ascher, Christiansen, and Russell[3], Ascher, Mattheij, and Russell[4], Betts[6], and Betts [8]. The basic idea behind most mesh refinement schemes is equidistribution of the global error in the approximation. In practice most schemes are designed around equidistribution of the local truncation error. As Ascher[4] mentions, the particular scheme chosen is less important than simply having one. Mesh refinement strategies have employed redistribution of nodes, addition of nodes, and both. This work uses addition of nodes, thus its final mesh includes the points which defined the initial mesh.

Since the integration scheme in this work is a fourth-order Runge-Kutta scheme, a search was made for an embedded RK formula which could be used for error estimation. The formula chosen was developed by Zonneveld[60], and was taken from [35]. The coefficients are given in table 2.1.

The procedure is first to compute the local truncation error estimates for each integration interval according to the formula

$$\varepsilon_i = x_i - \hat{x}_i \tag{2.51}$$

where $\hat{x}_i$ is the state estimate at t_i using the RK scheme in Table 2.1, and ϵ_i is the local truncation error estimate. Next the sum of the truncation errors

Table 2.1: RK4(3) Coefficients due to Zonneveld

0					
$\frac{1}{2}$	$\frac{1}{2}$				
$\frac{1}{2}$	0	$\frac{1}{2}$			
1	0	0	1		
$\frac{3}{4}$	$\frac{5}{32}$	$\frac{7}{32}$	$\frac{13}{32}$	$-\frac{1}{32}$	
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$	
	$-\frac{1}{2}$	$\frac{7}{3}$	$\frac{7}{3}$	$\frac{13}{6}$	$-\frac{16}{3}$

for each interval is computed

$$\sigma = \sum_{i=0}^{N} |\varepsilon_i| \tag{2.52}$$

The contribution of each integration interval to the overall truncation error is computed by

$$\eta_i = \frac{|\varepsilon_i|}{\sigma} \tag{2.53}$$

Next, a node is added to the interval with the largest contribution, η_i, and an estimate for η_i in each of the new subintervals is computed using

$$\varepsilon_{i_{\text{new}}} = \varepsilon_{i_{\text{old}}} \left(\frac{h_{\text{new}}}{h_{\text{old}}}\right)^3 \tag{2.54}$$

$$\sigma_{\text{new}} = \sigma_{\text{old}} - \varepsilon_{i_{\text{old}}} \left[M_{i_{\text{old}}} - (M_{i_{\text{old}}} + 1)\left(\frac{h_{\text{new}}}{h_{\text{old}}}\right)^3\right] \tag{2.55}$$

$$\eta_{i_{\text{new}}} = \frac{\varepsilon_{i_{\text{new}}}}{\sigma_{\text{new}}} \tag{2.56}$$

where $M_{i_{\text{old}}}$ is the previous number of nodes in interval i. Next, the process of selecting an interval and adding a point is repeated using the new values of

the error contributions, η_i. The process of adding points continues until the number of points added to an arc is equal to one less than the previous number of nodes in the arc. This procedure is very similar to the one given by Betts in [6].

2.3 Sparse NLP

The multiple shooting approach described in the previous section results in a "sparse" nonlinear programming problem–most of the Hessian and constraint Jacobian elements are zero. Taking the derivative of the NLP constraint vector 2.30 results in the following Jacobian:

$$\begin{bmatrix} D_0 & & & & & E_0 \\ A_0 & B_0 & & & & C_0 \\ & \ddots & \ddots & & & \vdots \\ & & A_{N-2} & B_{N-2} & & C_{N-2} \\ & & & A_{N-1} & \tilde{B}_{N-1} & C_{N-1} \\ & & & & D_f & E_f \end{bmatrix} \tag{2.57}$$

where

$$A_i = \begin{bmatrix} \frac{\partial \Gamma_{i,2}}{\partial x_{i,1}} & \frac{\partial \Gamma_{i,2}}{\partial u_{i,1}} & \cdots & \frac{\partial \Gamma_{i,2}}{\partial x_{i,4}} & \frac{\partial \Gamma_{i,2}}{\partial u_{i,4}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{\partial \Gamma_{i,4}}{\partial x_{i,1}} & \frac{\partial \Gamma_{i,4}}{\partial u_{i,1}} & \cdots & \frac{\partial \Gamma_{i,4}}{\partial x_{i,4}} & \frac{\partial \Gamma_{i,4}}{\partial u_{i,4}} \\ \frac{\partial \Delta_i}{\partial x_{i,1}} & \frac{\partial \Delta_i}{\partial u_{i,1}} & \cdots & \frac{\partial \Delta_i}{\partial x_{i,4}} & \frac{\partial \Delta_i}{\partial u_{i,4}} \end{bmatrix} \tag{2.58}$$

$$B_i = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \frac{\partial \Delta_i}{\partial x_{i+1}} & \frac{\partial \Delta_i}{\partial u_{i+1}} & 0 & \dots & 0 \end{bmatrix} \tag{2.59}$$

$$\tilde{B}_{N-1} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \frac{\partial \Delta_{N-1}}{\partial x_N} & \frac{\partial \Delta_{N-1}}{\partial u_N} \end{bmatrix} \tag{2.60}$$

$$C_i = \begin{bmatrix} \frac{\partial \Gamma_{i,2}}{\partial t_0} & \frac{\partial \Gamma_{i,2}}{\partial t_1} \\ \frac{\partial \Gamma_{i,3}}{\partial t_0} & \frac{\partial \Gamma_{i,3}}{\partial t_1} \\ \frac{\partial \Gamma_{i,4}}{\partial t_0} & \frac{\partial \Gamma_{i,4}}{\partial t_1} \\ \frac{\partial \Delta_i}{\partial t_0} & \frac{\partial \Delta_i}{\partial t_1} \end{bmatrix} \tag{2.61}$$

$$D_0 = \begin{bmatrix} \frac{\partial \psi_0}{\partial x_0} & \frac{\partial \psi_0}{\partial u_0} & 0 & \dots & 0 \end{bmatrix} \tag{2.62}$$

$$E_0 = \begin{bmatrix} \frac{\partial \psi_0}{\partial t_0} & \frac{\partial \psi_0}{\partial t_1} \end{bmatrix} \tag{2.63}$$

$$D_f = \begin{bmatrix} \frac{\partial \psi_f}{\partial x_N} & \frac{\partial \psi_f}{\partial u_N} \end{bmatrix} \tag{2.64}$$

$$E_f = \begin{bmatrix} \frac{\partial \psi_f}{\partial t_0} & \frac{\partial \psi_f}{\partial t_1} \end{bmatrix} \tag{2.65}$$

Note path constraints have been removed for clarity.

To efficiently solve such a large, sparse NLP problem the NLP solver must take advantage of the sparsity in the problem. One such sparse solver is SNOPT, by Gill, Murray and Saunders, a sequential quadratic programming (SQP) code designed specifically for large, sparse problems[23, 24]. SNOPT uses a null-space SQP method intended primarily for NLP problems with many active constraints at the solution. Sparsity features include a limited-memory Broyden-Fletcher-Goldfarb-Shanno (BFGS) Hessian approximation

and a compressed sparse column constraint Jacobian. SNOPT is designed to solve the problem

$$\min_X \; J(X) \tag{2.66}$$

subject to

$$l \leq \begin{bmatrix} X \\ F(X) \\ GX \end{bmatrix} \leq u \tag{2.67}$$

where $J(X)$ is the objective function, X is the set of independent variables, $F(X)$ is the set of nonlinear constraint functions, GX is the set of linear constraints, and l and u are sets of upper and lower bounds. For equality constraints, $l = u$. Internally, SNOPT reformulates the constraints using slack variables as

$$\begin{bmatrix} F(X) \\ GX \end{bmatrix} - S = 0 \tag{2.68}$$

$$l \leq \begin{bmatrix} X \\ S \end{bmatrix} \leq u \tag{2.69}$$

where S is the vector of slack variables.

Convergence is based on two quantities: feasibility and optimality. The feasibility criteria is the maximum constraint violation, given by

$$\max_i r_i \leq \|(X, S)\| \, \epsilon_r \tag{2.70}$$

where

$$r_i = \begin{cases} F_i - u_{F_i} & u_{F_i} \leq F_i \\ 0 & l_{F_i} \leq F_i \leq u_{F_i} \\ l_{F_i} - F_i & F_i \leq l_{F_i} \end{cases} \tag{2.71}$$

F_i is the i-th nonlinear constraint, u_{F_i} its upper bound, and l_{F_i} is its lower bound. ϵ_r is the major feasibility tolerance, and controls how accurately constraints are satisfied at the solution. When the condition 2.70 is true SNOPT considers the nonlinear constraints satisfied. The optimality criteria is the maximum complementarity gap, given by

$$\max_j d_j \leq \|\pi\| \, \epsilon_d \tag{2.72}$$

where

$$d_j = \begin{cases} \left(g_j - a_j^T \pi\right) \min\left(x_j - l_{X_j}, 1\right) & g_j - a_j^T \pi \geq 0 \\ -\left(g_j - a_j^T \pi\right) \min\left(u_{X_j} - x_j, 1\right) & g_j - a_j^T \pi < 0 \end{cases} \tag{2.73}$$

d_j is the complementarity gap, g_j is the j-th element of the gradient, a_j is the j-th column of the Jacobian for the reformulated constraints 2.68-2.69, π is the vector of NLP Lagrange multipliers, and ϵ_d is the major optimality tolerance. The complementarity gap is simply a scaled form of the reduced gradient $\left(g_j - a_j^T \pi\right)$.

This chapter has presented the numerical method used to solve the trajectory optimization problems presented in Chapter 4 and Chapter 5. The method involves transforming the optimal control problem into a parameter optimization problem which is then solved using an existing NLP solver, in

this case SNOPT. The solver requires derivative values at each iteration to compute search directions and lengths. Typically these are computed using finite differencing, especially in general-purpose trajectory optimization codes. The next chapter reviews derivative computation in trajectory optimization, including finite differencing, and then introduces the approach used for this work.

Chapter 3

Derivative Computation for Trajectory Optimization

This chapter focuses on the derivative computation used for this research. Background on derivative computation methods used in trajectory optimization starts the chapter, followed by a brief section reviewing finite differencing and its implementation for this research, and then a section introducing automatic differentiation. Two examples are included to illustrate automatic differentiation and its advantages. Finally, system sensitivities and the role of automatic differentiation in the direct multiple shooting scheme are described.

3.1 Background

Most general-purpose trajectory optimization packages use finite differencing to compute gradients, and the application of other techniques has been somewhat limited. POST[49], GTS[22], and OTIS[37] all employ finite differencing. The SOCS optimal control code[8], which has been incorporated into OTIS, uses the sparse Jacobian computation methods described in Curtis, Powell, and Reid[14] and Coleman and Moré[13] for sparse finite differencing

to reduce the computational expense required for finite differencing and allows larger problems to be solved economically.

The use of automatic differentiation in trajectory optimization has been limited and seems to be confined to indirect methods. Kalaba describes the application of a basic forward mode approach to different indirect optimal control problems[41]. The tool developed by Melhorn and Sachs[44] uses automatic differentiation in combination with indirect multiple shooting to solve optimal control problems. Some timing results are given for three problems: a dynamic soaring problem, a planar ascent trajectory optimization problem, and a non-planar ascent problem. Schöpf and Deuflhard developed the package OCCAL, which uses the computer algebra package REDUCE to analytically derive needed derivative expressions for use by an indirect multiple shooting scheme[51]. The paper mentions the analytical expressions become complex and automatic differentiation is being investigated as an alternative. Application of OCCAL to a turbo generator control design, a reentry problem, and robot arm control are briefly described but no detailed results were given.

It is not surprising the focus has been on applying automatic differentiation to indirect methods. Especially for a general-purpose code with many different models, like those mentioned above, the number of expressions to be derived is intimidating. On the other hand, the situation with respect to derivative evaluation in software based on direct methods is not much better. The variational equations, which are expressions for the derivatives with respect to all the NLP variables, must be derived. As mentioned by Betts, "the

variational equations must be derived for each application and, consequently, are used *far* less in general-purpose software."[8]

3.2 Finite Differencing

Finite differencing has proven an extremely useful procedure to compute derivatives of complex functions for use in optimization and root finding. The first-order forward difference formula is

$$\frac{\partial f}{\partial x} = \frac{f(x + \Delta x) - f(x)}{\Delta x} + O(\Delta x) \tag{3.1}$$

while the second-order central difference formula is

$$\frac{\partial f}{\partial x} = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} + O\left(\Delta x^2\right) \tag{3.2}$$

The key to finite differencing is proper selection of the perturbation step size, Δx, and this problem has led to automatic step selection schemes by various authors, including Hull[39] and Gill, Murray, and Wright[25]. The numerical analysis folklore says that in the presence of finite precision arithmetic, the precision of a finite difference derivative is, at best, half the precision of the underlying function. Thus, if f is computed in double precision, with 16 digits of precision, then the best which can be expected is $\frac{\partial f}{\partial x}$ will be accurate to eight digits. If f is itself the product of some other numerical approximation process, such as numerical integration, then the accuracy of the numerical procedure will govern the precision of the finite difference computations. For example,

if f is the numerical solution to some differential equations determined to six digits of precision, then $\frac{\partial f}{\partial x}$ will only have three digits of precision. Some basis for this rule-of-thumb can be found in Gill, Murray, and Wright[25].

The greatest single advantage of finite differencing when used in a general-purpose trajectory optimization program is its treatment of the function as a "black box"–no expressions must be derived, coded, and checked. When the functions stretch across several subroutines this advantage becomes extremely attractive. If the objective and constraint functions are selected at run time from a set of a few hundred possibilities, finite-differencing can seem the only way to practically implement derivative computations.

The finite differencing approach used for this work uses a central difference estimate with a perturbation step Δx computed from

$$\Delta x = \begin{cases} \eta \Delta x & x \neq 0 \\ \eta & x = 0 \end{cases} \tag{3.3}$$

where η is a perturbation step factor with a value of 10^{-5}.

3.3 Automatic Differentiation

Automatic differentiation is a term used to describe a variety of schemes for computing derivatives and Taylor coefficients of arbitrary functions to machine precision. The terms computational differentiation and algorithmic differentiation have also been used to describe the technique.

Automatic differentiation is based on the fact complicated functions are

built up from a limited set of operations and functions whose derivative rules are known. With automatic differentiation, the source code is differentiated by the computer in a preprocessing step or derivatives are propagated at run time using special libraries. Automatic differentiation is frequently confused with differentiation via symbolic algebra software such as *Maple* and *Mathematica,* because in both cases the rules of differentiation are being applied to expressions for a function. However, automatic differentiation is typically considered a different, distinct approach because no algebraic expressions for the derivatives are formed, and the procedure is aimed fundamentally at generating numerical values for derivatives.

3.3.1 A History of Automatic Differentiation

In 1982, Wolfe observed gradient evaluation is typically about 1.5 times as computationally expensive as the underlying function, and rarely greater than 2 times as expensive[59]. Griewank showed analytically the factor of 1.5 could be replaced with an upper bound of 5[30]. Much of the work in automatic differentiation in the last decade has focused on achieving this bound with as much automation of the differentiation process as possible.

As mentioned in the introduction, the concept of automatic differentiation has existed for decades. Griewank mentions Beda, et. al. seem to be the earliest report of automatic differentiation[30]. Wengert was first in the United States to publish the concept[57]. Following Wengert's paper in the same issue of *Communication of the ACM,* Wilkins applies Wengert's ideas to

the calculation of derivatives which arise in geodesy and satellite orbit determination problems[58]. Wilkins compared the accuracy of the partials obtained using automatic differentiation with those from analytic differentiation of the equations and found care had to be used to avoid overflow in the automatic derivative computations.

Wengert and Wilkins used what is now called the forward mode to propagate the value of a single derivative during the course of a function evaluation. If multiple derivatives were desired, the function had to be called multiple times, similar to the case with finite differencing. Probably the first discussion of the reverse mode was by Ostrovskii[47], while Speelpenning provided the first practical tool using the reverse mode, JAKEF, as part of his dissertation research[54]. Griewank[30, 31] provides an introduction to the subject with emphasis on the reverse mode. In [31], Griewank also analyzes combinations of the two approaches, as well as extensions to higher-order derivatives and Taylor series. Older, comprehensive reviews of forward mode research work include Rall[50], Kedem[42], and Kagiwada[41].

Some applications of automatic differentiation have indicated care must still be taken to avoid incorrect results and excessive computational cost. Eberhard and Bischof discuss the application of automatic differentiation to numerical integration algorithms, and the care which must be taken with the results, especially when the integration scheme is a variable-step/variable-order scheme[16]. In particular, the work shows the "black-box" application to a variable-step integrator can produce unexpected results, and a correction

term must be included or the integrator code must be modified after the application of automatic differentiation. Carle, et. al.[12], Green, Newman, and Haigler[29], and Sherman, et. al.[53] discuss investigations applying automatic differentiation to computational fluid dynamics. The results of these studies indicated a straightforward, "black-box" application of automatic differentiation did not produce a derivative code which was competitive with finite differencing in terms of memory or run-time, although derivative accuracy was improved. Performance was significantly improved by using a combination of automatic differentiation and an incremental iterative form for the derivatives of the flow quantities and functions of the flow quantities. In general automatic differentiation has proven to be a valuable tool, but one which must still be used with some thought and care.

3.3.2 A Simple Introduction to Automatic Differentiation

The best way to introduce automatic differentiation is with a simple example. Assume the function

$$f(x) = \sqrt{x^2 + y^2} \tag{3.4}$$

is required by a computer program. The Fortran source code would be

```
double precision x, y, f
f = sqrt(x * x + y * y)
```

Now, assume the partial derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ are also required. At this point the traditional approach would be to derive the derivative formulas

$$\frac{\partial f}{\partial x} = \frac{x}{\sqrt{x^2 + y^2}} \tag{3.5}$$

$$\frac{\partial f}{\partial y} = \frac{y}{\sqrt{x^2 + y^2}} \tag{3.6}$$

and then write them as source code

```
double precision x, y, df(2)
df(1) = x / sqrt(x * x + y * y)
df(2) = y / sqrt(x * x + y * y)
```

or, combining with the function source code,

```
double precision x, y, f, df(2)
f = sqrt(x * x + y * y)
df(1) = x / f
df(2) = y / f
```

The normal alternative for computing the derivative values would be to use either a forward or central difference formula to compute approximate values for the derivative. For such a simple formula using finite differencing would not necessarily be justified, but for more complicated formulas which stretch across multiple functions or subroutines, finite differencing can be a useful alternative to deriving, coding, and debugging exact derivative formulas.

Now consider the function source code can be rewritten as a sequence of one- and two-argument functions and arithmetic operators:

```
t1 = x
t2 = y
t3 = t1 * t1
t4 = t2 * t2
t5 = t3 + t4
t6 = sqrt(t5)
f = t6
```

The functions and operators all have well-known rules for propagating derivatives, which can be included along with the lines for the function:

```
t1 = x
dt1 = dx
t2 = y
dt2 = dy
t3 = t1 * t1
dt3 = dt1 * t1 + t1 * dt1
t4 = t2 * t2
dt4 = dt2 * t2 + t2 * dt2
t5 = t3 + t4
dt5 = dt3 + dt4
```

```
t6 = sqrt(t5)
dt6 = .5 * dt5 / t5
f = t6
df = dt6
```

Note the expressions for dt_3 and dt_4 were not simplified after application of the product rule. Now, if the initializations $dx = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$ and $dy = \begin{bmatrix} 0 & 1 \end{bmatrix}^T$ are made when x and y are given their respective values, then df will have the values of the derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ at the end of the execution of the code above. We now have source code which will compute the values of $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ together with the function value f. Note we have source code to compute function and derivative values, not algebraic formulas.

Consider that any function written in a high-level computer language can have the above procedure applied to it:

1. Break the source lines containing formulas into a sequence of one- and two-argument "elementary functions"

2. Combine the rule for propagating derivative values with the corresponding elementary function expression

The result of the procedure will be source code which can compute derivative values together with the function values. The sequence of elementary functions is frequently called a "code list" in automatic differentiation literature, and the procedure above is known as the forward mode of automatic differentiation.

For any major, high-level programming language the set of possible elementary functions will be arithmetic operators and some special functions, so a limited number of derivative rules are required. These rules can be encapsulated in library of functions which are called at run time, or a set of rules in a source code preprocessor. The process of breaking the code down into these functions and using the proper rules can be automated, so the programmer simply writes

```
double precision x, y, f
f = sqrt(x * x + y * y)
```

and then runs the code through a preprocessor to produce the derivative calculation source code. If the source code language supports operator and function overloading, then the derivative rules can be implemented using the overloading facilities and a preprocessor can be avoided altogether. Operator and function overloading is a feature of some computer languages where a function or operator can be given more than one definition, and the correct definition is chosen at compile time based on the data types of the arguments. With either a preprocessor or operator overloading there are no expressions to derive and translate into source code, making the programming task simpler and less bug-prone.

3.3.3 A Second Example: Perturbing Acceleration Due to J_2

The previous example used an almost trivial example. Now consider the following, more complex example: the perturbing acceleration due to J_2. The perturbing acceleration is given by

$$a_1 = -\frac{3J_2\mu R_E^2 x_1 \left(x_1^2 + x_2^2 + x_3^2 - 5x_3^2\right)}{2r^7} \tag{3.7}$$

$$a_2 = -\frac{3J_2\mu R_E^2 x_2 \left(x_1^2 + x_2^2 + x_3^2 - 5x_3^2\right)}{2r^7} \tag{3.8}$$

$$a_3 = -\frac{3J_2\mu R_E^2 x_3 \left(3x_1^2 + 3x_2^2 + 3x_3^2 - 5x_3^2\right)}{2r^7} \tag{3.9}$$

where a_i, $i = 1, 2, 3$ are the x-, y-, and z-components of the disturbing acceleration, x_i are the x-, y-, and z-position components of the satellite's radius vector, μ is the Earth's gravitational parameter, R_E is the Earth's equatorial radius, r is magnitude of the radius vector, and J_2 is the second zonal harmonic coefficient. The source code for this might look like

```
r2 = x(1) * x(1) + x(2) * x(2) + x(3) * x(3)
z2r2 = x(3) * x(3) / r2
coef = -1.5 * j2 * mu * re * re / (r2 * r2 * sqrt(r2))
a(1) = coef * x(1) * (1 - 5 * z2r2)
a(2) = coef * x(2) * (1 - 5 * z2r2)
a(3) = coef * x(3) * (3 - 5 * z2r2)
```

Note the use of intermediate variables to hold values of common terms, making the resulting code slightly faster and somewhat less cluttered. Now imagine

the Jacobian $\frac{\partial a_i}{\partial x_j}$, $i, j = 1, 2, 3$, is required, say as part of a navigation program intended to determine satellite position based on a series of tracking measurements. The Jacobian elements are

$$\frac{\partial a_1}{\partial x_1} = \frac{3J_2\mu R_E^2\left(4x_1^4 - x_2^4 + 3x_2^2x_3^2 + 4x_3^4 + 3x_1^2\left(x_2^2 - 9x_3^2\right)\right)}{2\left(x_1^2 + x_2^2 + x_3^2\right)^{9/2}} \tag{3.10}$$

$$\frac{\partial a_1}{\partial x_2} = \frac{15J_2\mu R_E^2 x_1x_2\left(x_1^2 + x_2^2 - 6x_3^2\right)}{2\left(x_1^2 + x_2^2 + x_3^2\right)^{9/2}} \tag{3.11}$$

$$\frac{\partial a_1}{\partial x_3} = \frac{15J_2\mu R_E^2 x_1x_3\left(3x_1^2 + 3x_2^2 - 4x_3^2\right)}{2\left(x_1^2 + x_2^2 + x_3^2\right)^{9/2}} \tag{3.12}$$

$$\frac{\partial a_2}{\partial x_1} = \frac{15J_2\mu R_E^2 x_1x_2\left(x_1^2 + x_2^2 - 6x_3^2\right)}{2\left(x_1^2 + x_2^2 + x_3^2\right)^{9/2}} \tag{3.13}$$

$$\frac{\partial a_2}{\partial x_2} = -\frac{3J_2\mu R_E^2\left(x_1^4 - 4x_2^4 + 27x_2^2x_3^2 - 4x_3^4 - 3x_1^2\left(x_2^2 + x_3^2\right)\right)}{2\left(x_1^2 + x_2^2 + x_3^2\right)^{9/2}} \tag{3.14}$$

$$\frac{\partial a_2}{\partial x_3} = \frac{15J_2\mu R_E^2 x_2x_3\left(3x_1^2 + 3x_2^2 - 4x_3^2\right)}{2\left(x_1^2 + x_2^2 + x_3^2\right)^{9/2}} \tag{3.15}$$

$$\frac{\partial a_3}{\partial x_1} = \frac{15J_2\mu R_E^2 x_1x_3\left(3x_1^2 + 3x_2^2 - 4x_3^2\right)}{2\left(x_1^2 + x_2^2 + x_3^2\right)^{9/2}} \tag{3.16}$$

$$\frac{\partial a_3}{\partial x_2} = \frac{15J_2\mu R_E^2 x_2x_3\left(3x_1^2 + 3x_2^2 - 4x_3^2\right)}{2\left(x_1^2 + x_2^2 + x_3^2\right)^{9/2}} \tag{3.17}$$

$$\frac{\partial a_3}{\partial x_3} = \frac{3J_2\mu R_E^2\left(4x_1^4 - x_2^4 + 3x_2^2x_3^2 + 4x_3^4 + 3x_1^2\right)}{2\left(x_1^2 + x_2^2 + x_3^2\right)^{9/2}} \tag{3.18}$$

These expressions were derived using *Mathematica*, a symbolic algebra program, not by hand. Already the expressions are becoming complex and would require care to avoid introducing bugs in the expressions if they were typed by hand. Using *Mathematica*'s ability to print a formula as Fortran source code produces the following source for the Jacobian:

```
da(1,1) = (3*mu*j2*re**2*
    (4*x(1)**4 -
x(2)**4 + 3*x(2)**2*x(3)**2 + 4*x(3)**4 +
        3*x(1)**2*(x(2)**2 - 9*x(3)**2)))/
    (2.*(x(1)**2 + x(2)**2 + x(3)**2)**4.5)
da(1,2) = (15*mu*j2*re**2*x(1)*x(2)*
    (x(1)**2 + x(2)**2 - 6*x(3)**2))/
    (2.*(x(1)**2 + x(2)**2 + x(3)**2)**4.5)
da(1,3) = (15*mu*j2*re**2*x(1)*x(3)*
    (3*x(1)**2 + 3*x(2)**2 - 4*x(3)**2))/
    (2.*(x(1)**2 + x(2)**2 + x(3)**2)**4.5)
da(2,1) = (15*mu*j2*re**2*x(1)*x(2)*
    (x(1)**2 + x(2)**2 - 6*x(3)**2))/
    (2.*(x(1)**2 + x(2)**2 + x(3)**2)**4.5)
da(2,2) = (-3*mu*j2*re**2*
    (x(1)**4 - 4*x(2)**4 + 27*x(2)**2*x(3)**2 -
4*x(3)**4 -
        3*x(1)**2*(x(2)**2 + x(3)**2)))/
    (2.*(x(1)**2 + x(2)**2 + x(3)**2)**4.5)
da(2,3) = (15*mu*j2*re**2*x(2)*x(3)*
    (3*x(1)**2 + 3*x(2)**2 - 4*x(3)**2))/
    (2.*(x(1)**2 + x(2)**2 + x(3)**2)**4.5)
da(3,1) = (15*mu*j2*re**2*x(1)*x(3)*
```

```
     (3*x(1)**2 + 3*x(2)**2 - 4*x(3)**2))/
     (2.*(x(1)**2 + x(2)**2 + x(3)**2)**4.5)
da(3,2) = (15*mu*j2*re**2*x(2)*x(3)*
     (3*x(1)**2 + 3*x(2)**2 - 4*x(3)**2))/
     (2.*(x(1)**2 + x(2)**2 + x(3)**2)**4.5)
da(3,3) = (3*mu*j2*re**2*
     (-
35*x(3)**4 + 30*x(3)**2*(x(1)**2 + x(2)**2 + x(3)**2) -

         3*(x(1)**2 + x(2)**2 + x(3)**2)**2))/
     (2.*(x(1)**2 + x(2)**2 + x(3)**2)**4.5)
```

While Mathematica's ability to print formulas as source code prevents bugs from typing errors the expressions are beginning to become long and hard to read. Also, common subexpressions, such as $x_1^2 + x_2^2 + x_3^2$, must still be factored out by hand. Using automatic differentiation would involve passing the source code for computing a_i through a preprocessor to generate source code for computing the derivatives, or converting the source lines for computing a_i to a series of function calls to a library of elementary functions which would propagate the derivatives at every call. If the function routines were written using a language which provided operator and function overloading, such as C++ or Fortran 90, the conversion of a formula to a code list would be left to the compiler.

The process of providing source code for a function and its derivatives can be thought of as involving a sequence of two steps:

1. Differentiation

2. Translation of algebraic formulas into a programming language

The traditional approach to providing exact derivatives performs these two steps in this order. The algebraic formulas are differentiated, by hand or machine, to produce algebraic formulas for the derivatives. Next, the function and derivative formulas are both written in a high-level programming language. To use automatic differentiation is to reverse the order of the two operations. First, the function formula, or formulas, are written in source code. Next, the source code, which is an algorithmic representation of the function, is differentiated to produce source code for the derivatives.

Proceeding from the first to the second example increases the complexity of the expression involved and shows the problem with deriving derivative expressions from the algebraic formulas. The derivative formulas for the second example are still manageable, but they are beginning to become complex, with many terms. If they were derived by hand the potential for typing errors becomes significant, and if they are derived using a symbolic algebra program the expressions can become unreadable and would probably not be computationally efficient.

The first example points out the key to the practicality of automatic differentiation–making it truly automatic. Early applications of automatic

differentiation used handmade code lists and called a set of library functions. Later, preprocessors were introduced to avoid the tedium of creating a code list manually for every arithmetic expression in the source code. The rise in the last ten or fifteen years of programming languages with operator and function overloading has made automatic differentiation even more accessible.

3.4 System Sensitivities and the Role of Automatic Differentiation

When moving beyond complex explicit equations to systems of nonlinear algebraic equations or differential equations, the question arises of how automatic differentiation behaves when applied to an iterative procedure or numerical integration. In both cases, there are two basic options for propagating derivatives. The first approach is to apply automatic differentiation to source code for the numerical method as well as the function evaluation code. Unfortunately this approach requires some caution since the behavior of automatic differentiation is not necessarily what would be intended in these situations.

For nonlinear equation solvers, the problem can come from the recursive formulas used in iteration. Newton and secant updates can both exhibit a lag between solution and derivative convergence. For more details and numerical examples see Griewank, et. al. [32].

In a modern numerical integration code, step size, and possibly approx-

imation order, is adjusted to control truncation error and improve computational efficiency. The step size controller will make calculations based on the last step size and estimates of the truncation error for the last step. These formulas introduce functional dependencies which are not present in the differential equation itself and introduce an error in the derivative calculations if automatic differentiation is applied to the integrator source code. Eberhard and Bischof[16] investigated the problem in detail and derived the correction term for the derivatives computed via automatic differentiation.

An alternative approach, which is the one used in the present study, is to develop the equations for the sensitivities of the solutions and use automatic differentiation to compute the terms in the sensitivity equations. Consider a set of nonlinear equations of the form $f\left[x(a), a\right] = 0$, where $x \in \Re^n$, $a \in \Re^m$ and $f : \Re^{n \times m} \rightarrow \Re^n$. Differentiating with respect to the parameters a and solving for the sensitivities $\frac{\partial x}{\partial a}$ gives

$$\frac{\partial x}{\partial a} = \left[\frac{\partial f}{\partial x}\right]^{-1} \frac{\partial f}{\partial a}. \tag{3.19}$$

Automatic differentiation can be used to compute the terms on the right side of the equation, which can then be solved for the derivatives $\frac{\partial x}{\partial a}$. Equation 3.19 can be solved once, after the end of the iteration process, to compute the sensitivity of the solution with respect to parameters a.

For differential equations the process is similar. Consider the ODE

$$\dot{x}\left(t, a\right) = f\left[t, x\left(t, a\right), a\right] \tag{3.20}$$

Taking the derivative with respect to a of both sides and switching the order of differentiation on the left side gives

$$\frac{d}{dt}\left(\frac{\partial x}{\partial a}\right) = \frac{\partial f}{\partial x}\frac{\partial x}{\partial a} + \frac{\partial f}{\partial a} \tag{3.21}$$

This is a differential equation for $\frac{\partial x}{\partial a}$ which can be integrated forward in time together with the original ODE's. Automatic differentiation can be used here to compute the terms $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial a}$, or it can be used to compute the entire right-hand side via appropriate initialization of the gradient vectors.

The normalization of time in the multiple shooting scheme described above presents a twist which complicates the problem slightly. In this case the initial and final times, t_0 and t_f, are independent variables, and the time t_i and step size h_i are functions of t_0 and t_f. Also, the functions being differentiated are the defect functions, not the differential equations themselves. Differentiating the defect functions, eqns. 2.22-2.25, with respect to an arbitrary parameter a gives the system

$$\frac{\partial \Gamma_{i,2}}{\partial a} = \frac{\partial y_{i+1,2}}{\partial a} - \frac{\partial x_i}{\partial a} - \frac{1}{2}\frac{\partial K_{i,1}}{\partial a} \tag{3.22}$$

$$\frac{\partial \Gamma_{i,3}}{\partial a} = \frac{\partial y_{i+1,3}}{\partial a} - \frac{\partial x_i}{\partial a} - \frac{1}{2}\frac{\partial K_{i,2}}{\partial a} \tag{3.23}$$

$$\frac{\partial \Gamma_{i,4}}{\partial a} = \frac{\partial y_{i+1,4}}{\partial a} - \frac{\partial x_i}{\partial a} - \frac{\partial K_{i,3}}{\partial a} \tag{3.24}$$

$$\tag{3.25}$$

$$\frac{\partial \Delta_i}{\partial a} = \frac{\partial x_{i+1}}{\partial a} - \frac{\partial x_i}{\partial a} - \frac{1}{6}\frac{\partial K_{i,1}}{\partial a} - \frac{1}{3}\frac{\partial K_{i,2}}{\partial a} - \frac{1}{3}\frac{\partial K_{i,3}}{\partial a} - \frac{1}{6}\frac{\partial K_{i,4}}{\partial a} \tag{3.26}$$

where

$$\frac{\partial K_{i,j}}{\partial a} = \frac{\partial h_i}{\partial a} f_{i,j} + h_i \frac{\partial f_{i,j}}{\partial a}. \tag{3.27}$$

When a is the initial or final time the first term in the right side of the equation is nonzero, otherwise the term is zero. For this work, eqns. 3.22-3.27 were derived and coded by hand while the $\frac{\partial f_{i,j}}{\partial a}$ terms were computed using automatic differentiation.

While general-purpose trajectory optimization codes have traditionally used finite-differencing for their derivative computations, automatic differentiation provides a viable alternative for computing the derivative expressions required by the optimization routines. The application of automatic differentiation must be done with some care, however, to avoid introducing errors into the derivative computations. Next, the first of two trajectory optimization problems will be presented along with results comparing the results obtained using finite difference derivatives to results obtained using exact derivatives.

Chapter 4

Lunar Launch Problem

4.1 Description and Exact Solution

This well-known problem is Problem 13 in Section 2.7 of Bryson and Ho[10]. The objective is to minimize the final time, t_f, for a launch from the surface of a body to specified terminal conditions. The attracting body is modeled as flat and infinitely long, with no atmosphere. Gravity points down with a constant value. Figure 4.1 shows the geometry of the problem. The equations of motion are

$$\dot{x} = u \tag{4.1}$$

$$\dot{y} = v \tag{4.2}$$

$$\dot{u} = a \sin(\beta) \tag{4.3}$$

$$\dot{v} = a \sin(\beta) - g \tag{4.4}$$

where $x(t)$ is the horizontal position, $y(t)$ is the vertical position, $u(t)$ is the horizontal velocity, and $v(t)$ is the vertical velocity. The thrust acceleration, a, is assumed constant and the angle, β, between the thrust vector and horizontal

is the control. The boundary conditions are

$$x(0) = 0 \tag{4.5}$$

$$y(0) = 0 \qquad y(t_f) = h \tag{4.6}$$

$$u(0) = 0 \qquad u(t_f) = U \tag{4.7}$$

$$v(0) = 0 \qquad v(t_f) = 0 \tag{4.8}$$

where h is the specified insertion altitude, and U is the specified insertion velocity. For this work, g was 1.54 m/s^2, a is $3g$, h was 93,000 m, and U was 1617 m/s. These values correspond to ascent from the lunar surface to a circular orbit at an altitude of 93 km, in a vehicle with a thrust-to-weight ratio of 3.

The first-order necessary conditions lead to a pair of nonlinear algebraic equations

$$\frac{a}{g} = \frac{\tan\beta_0 - \tan\beta_f}{\sec\beta_0 - \sec\beta_f} \tag{4.9}$$

$$\frac{2ah}{U^2} = \left[\tan\beta_0 \sec\beta_f - \tan\beta_f \sec\beta_0 - \log\frac{\tan\beta_0 + \sec\beta_0}{\tan\beta_f + \sec\beta_f}\right] \left[\log\frac{\tan\beta_0 + \sec\beta_0}{\tan\beta_f + \sec\beta_f}\right]^{-2} \tag{4.10}$$

which specify the initial control, β_0, and the final control, β_f. The optimal control history follows a "linear tangent law"

$$\tan\beta = \tan\beta_0 - ct \tag{4.11}$$

where

$$c = \frac{\tan\beta_0 - \tan\beta_f}{T} \tag{4.12}$$

and the optimal final time, T, is given by the equation

$$\frac{aT}{U} = \frac{(\tan\beta_0 - \tan\beta_f)}{\log\left[\frac{\tan\beta_0 + \sec\beta_0}{\tan\beta_f + \sec\beta_f}\right]} \tag{4.13}$$

This analytic solution provides a useful tool to check the computational solutions.

4.2 Solution Convergence Results

Figures 4.2-4.4 show the initial guess and converged solution for the case with 21 nodes and using automatic differentiation for computing derivatives. The true solution is indistinguishable from the solution in figures 4.2-4.4 at the resolution of the figure. Figure 4.2 shows the path of the vehicle, figure 4.3 the hodograph of the vehicle, and figure 4.4 the control history. The vehicle

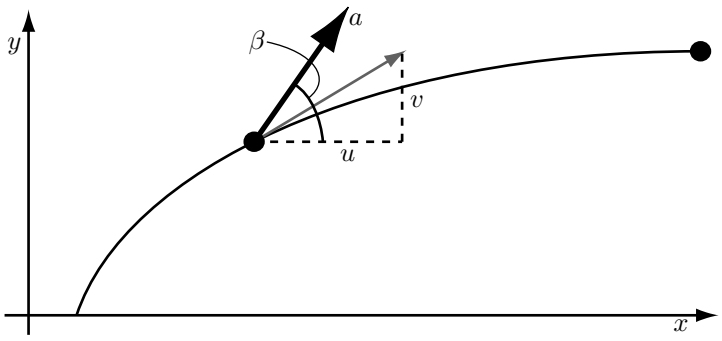

Figure 4.1: Lunar Launch Problem Geometry

starts with zero velocity and a thrust inclination angle of 56.5°. As the vehicle begins to rise the thrust direction decreases, passing through 0° at 310 seconds into the flight and reaching -27.8° at the end of the flight. The entire flight lasts 417.9 seconds. Maximum vertical velocity is 345 m/s, at 237 seconds.

Table 4.1 summarizes some results when step size was varied. For all the cases listed SNOPT was run with feasibility and optimality tolerances of 5×10^{-13} and a major iteration limit of 80 iterations. Because this is a relatively simple problem the optimal final time is determined to six digits with nine nodes and 10 digits with 101 nodes.

Figure 4.5 and 4.6 show the max norm of the error in the states, adjoints, and control as a function of step size. The data points were computed using automatic differentiation for derivatives. Each set of symbols at a particular step size corresponds to a solution on a particular mesh and a row in table 4.1. Note logarithmic scales are used on both axes. As expected all five quantities decrease as the step size is reduced, with a linear trend on the log-log scale, until roundoff error becomes a barrier to further improvement. As with the states, these discrete multipliers converge with fourth-order accuracy. The Lagrange multiplier corresponding to the horizontal position is not shown because the value was always below 10^{-12}.

Fitting an exponential model to the linear portion of figures 4.5 and 4.6 gives the exponents listed in table 4.2. The state and adjoint histories converge with fourth-order accuracy, as expected given the fourth-order integration rule used in the shooting scheme and the discrete adjoint results in Chapter 2. For

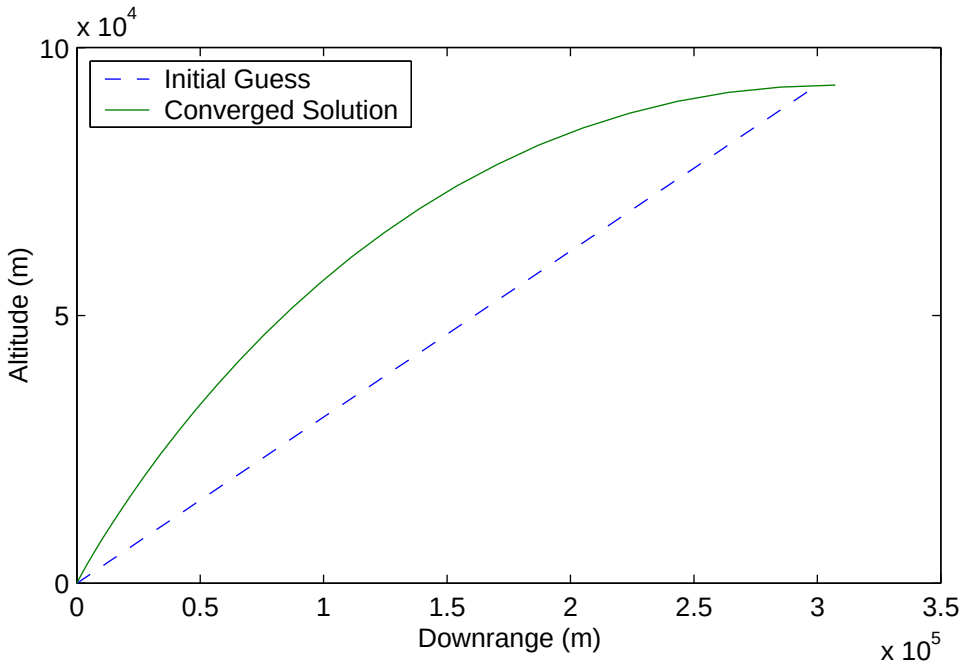

Figure 4.2: Initial Guess and Final Flight Paths

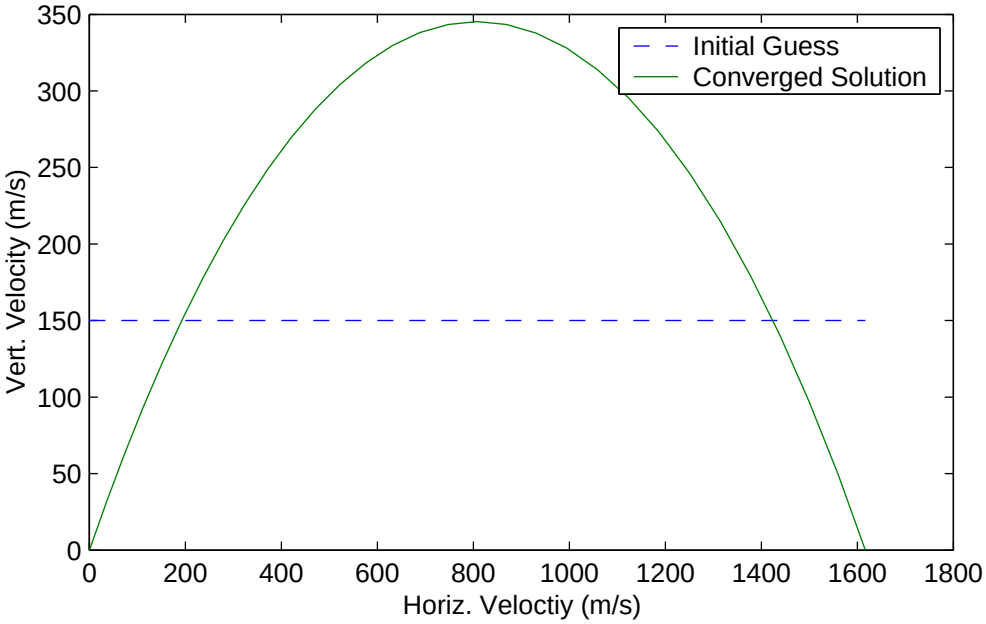

Figure 4.3: Initial Guess and Final Hodographs

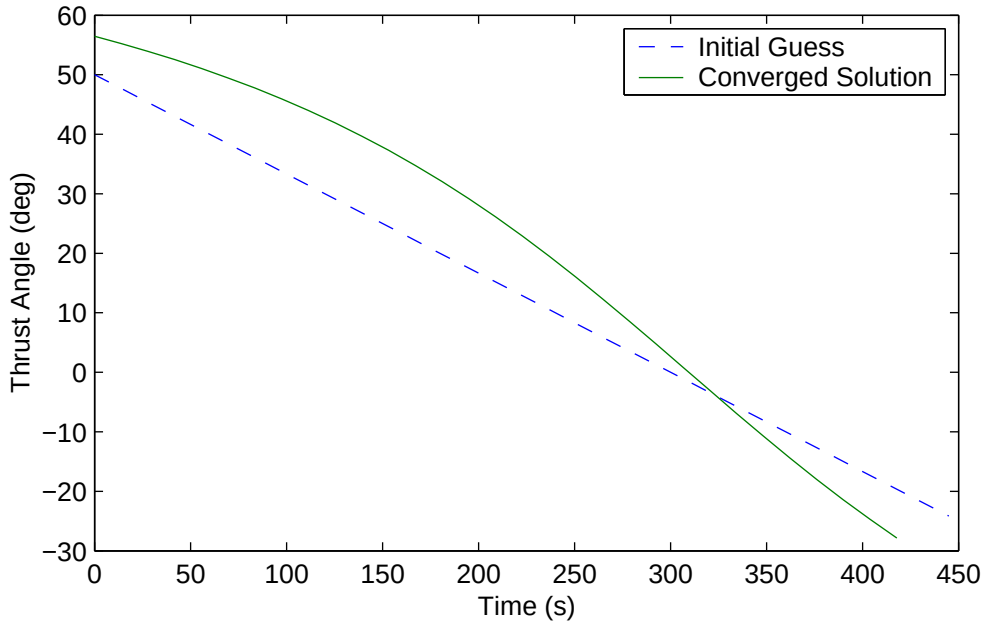

Figure 4.4: Initial Guess and Final Control Histories

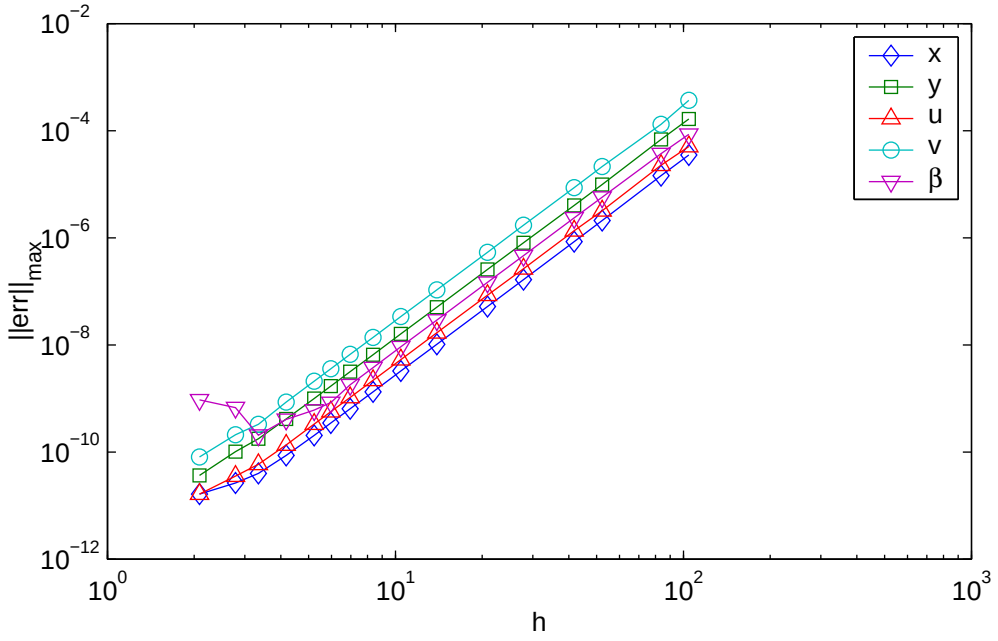

Figure 4.5: State and Control Error vs. Step Size Using Automatic Differentiation

Table 4.1: Lunar Launch Problem Solutions

# Nodes	Step Size (s)	Iterations	Final Time (s)
5	104.8	28	4.1790974742E+02
6	83.6	33	4.1790733211E+02
9	52.2	53	4.1790591670E+02
11	41.8	23	4.1790576799E+02
16	27.9	26	4.1790568552E+02
21	20.9	31	4.1790567168E+02
31	13.93	24	4.1790566655E+02
41	10.45	24	4.1790566568E+02
51	8.36	28	4.1790566545E+02
61	6.97	49	4.1790566536E+02
71	5.97	49	4.1790566533E+02
81	5.22	44	4.1790566531E+02
101	4.18	54	4.1790566529E+02
126	3.34	31	4.1790566529E+02
151	2.79	23	4.1790566529E+02
201	2.09	29	4.1790566528E+02

this problem, the control history exhibits fourth-order convergence, not the third-order convergence observed by Hager on some other problems in [33]. The exact solution λ_x is a constant value of zero, so the computed multiplier history was taken to be correct but a convergence order was not computed from it. While the fourth-order convergence of the discrete adjoint has been proven analytically for this scheme by others, control convergence order has not.

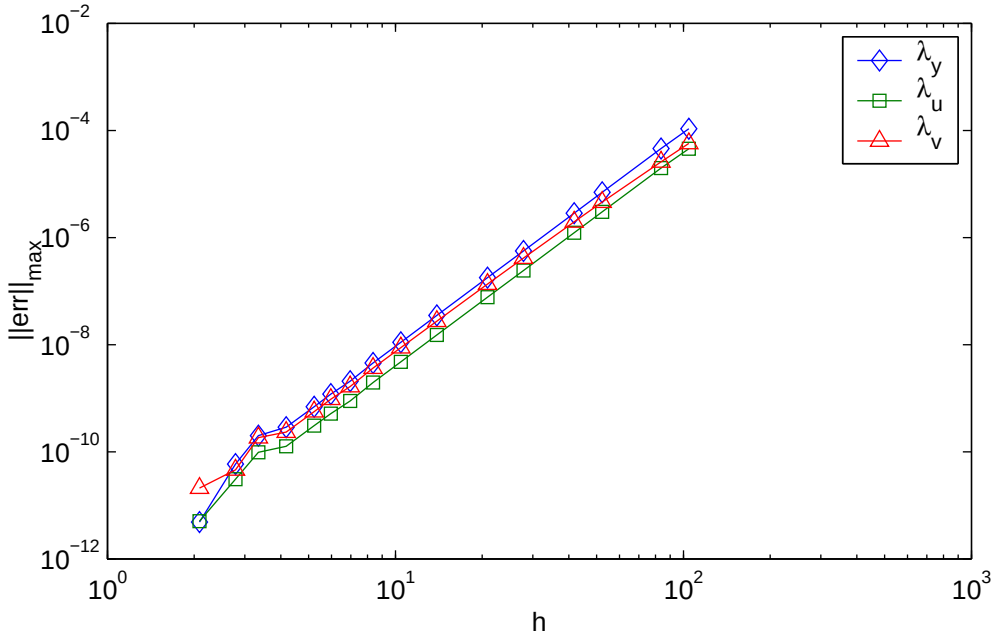

Figure 4.6: Adjoint Error vs. Step Size Using Automatic Differentiation

Table 4.2: Mesh Convergence Order Using Automatic Differentiation

Variable	Order
x	4.025
y	4.008
u	3.995
v	4.008
λ_y	4.001
λ_u	3.996
λ_v	3.874
β	3.999

4.3 Automatic Differentiation vs. Finite Differencing

Figures 4.7 and 4.8 show the same quantities as figures 4.5 and 4.6 except the data in figures 4.7 and 4.8 were generated using finite differencing for derivative computation. Comparing the trend for β in figure 4.5 and figure 4.7 there is little difference. The trend is linear until the final four data points, where roundoff begins to affect the results. Similar behavior can be seen in figures 4.6 and 4.8 where the adjoints follow a linear trend until the about the last four points, where roundoff error begins to affect the solutions. No major difference is seen between the results using finite differencing and the results using automatic differentiation.

Table 4.3 presents the results in table 4.2 along with exponents computed from the data in figures 4.7 and 4.8. Ideally, the exponents would all be four, and all the computed exponents are close. There is a pattern in the results with automatic differentiation producing the same or slightly more accurate results for all the variables except x. However, the differences are so small–a few or several thousandths–that for most problems it would not matter. There is greater variation between the exponents of the different variables than between the same variable using the different derivative computation methods.

Figure 4.9 illustrates one of the noticeable differences between optimization using approximate derivatives and optimization using exact derivatives. In the figure the number of major iterations required for convergence is given as a function of the number of nodes in the trajectory. The use of exact deriva-

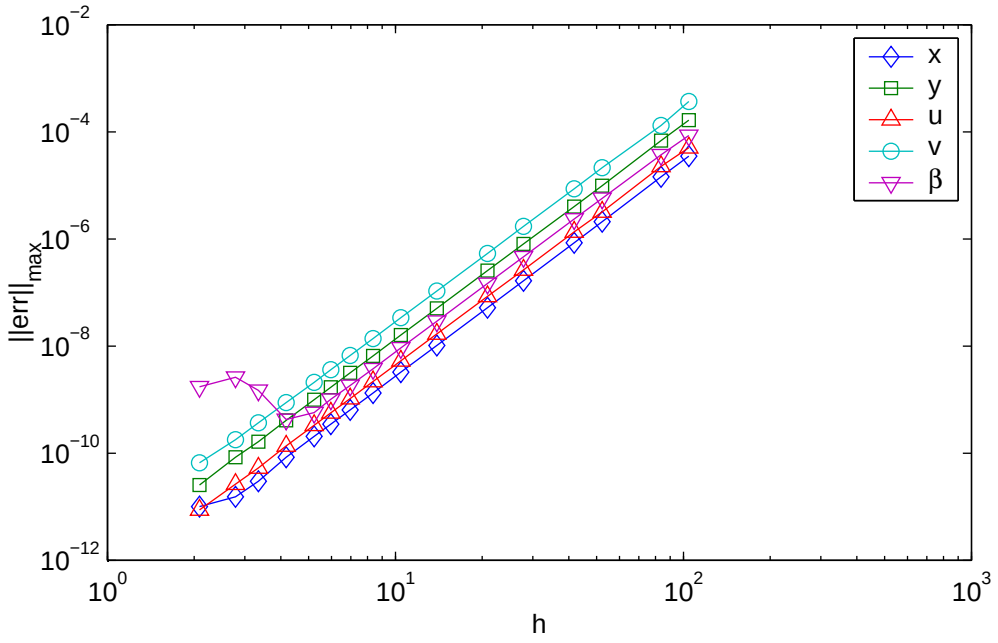

Figure 4.7: State and Control Error vs. Step Size Using Finite Differencing

Table 4.3: State Mesh Convergence Order

	Convergence Order	
Variable	FD	AD
x	4.024	4.025
y	4.009	4.008
u	3.995	3.995
v	4.008	4.008
λ_y	3.997	4.001
λ_u	3.992	3.996
λ_v	3.870	3.874
β	3.984	3.999

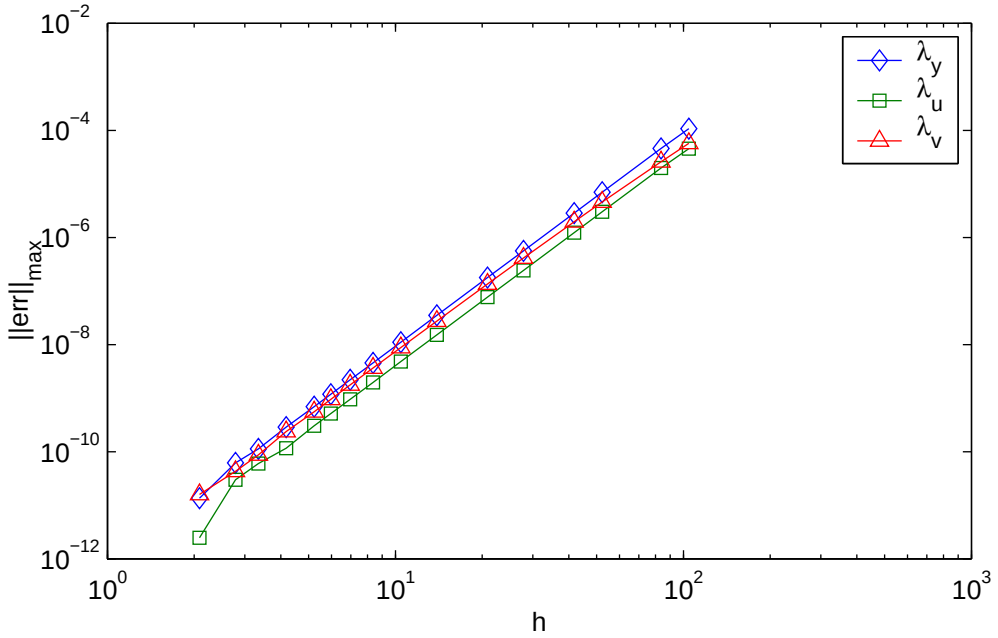

Figure 4.8: Adjoint Error vs. Step Size Using Finite Differencing

tives provides convergence in fewer iterations than when finite differences are used, with the cases using finite differencing often failing to reach the prescribed tolerance of 5×10^{-13} before reaching the major iteration limit of 80 iterations. Figures 4.10 and 4.11 show the feasibility and optimality convergence histories, respectively, for the case with 21 nodes. When using finite differencing the code levels off after about 24 iterations with optimality oscillating around 10^{-12}. An important point is the difference does not show up until the code reaches a very low value for both feasibility and optimality–less than 10^{-13} for feasibility and about 10^{-12} for optimality–much less than the default tolerances for SNOPT, which are 10^{-6} for both criteria. Using the default tolerances the code would stop after the same number of iterations,

regardless of the type of differencing computation used.

Finite differencing could be expected to give much worse performance than what is shown in this problem, with optimality and feasibility leveling off higher than 10^{-12}. The explanation lies in the simplicity of the problem–the differential equations are linear in the states, and the control only appears in a single trigonometric function in two of the four state differential equations. Computing the Jacobian of a linear system using finite differences will provide an exact Jacobian, even using a first-order forward-difference formula, so some of the required derivatives are being computed exactly.

4.4 Computational Considerations

The code used to generate the results in this chapter was written in Java, with two exceptions: SNOPT, which is written in Fortran 77; and the code to interface Java and Fortran 77, which was done using C++. All tests were run on a Power Macintosh G4 with dual 450MHz PowerPC G4 processors running Yellow Dog Linux 2.3. No use of parallel processing was made, in spite of the dual processors.

Tables 4.5 and 4.4 show some parameters related to computer time and memory requirements for the cases using automatic differentiation. First, table 4.4 shows basic NLP problem dimensions as well as Jacobian sparsity. Note with only nine nodes about 10% of the Jacobian are nonzero elements, and as the number of nodes grows larger the Jacobian becomes more sparse.

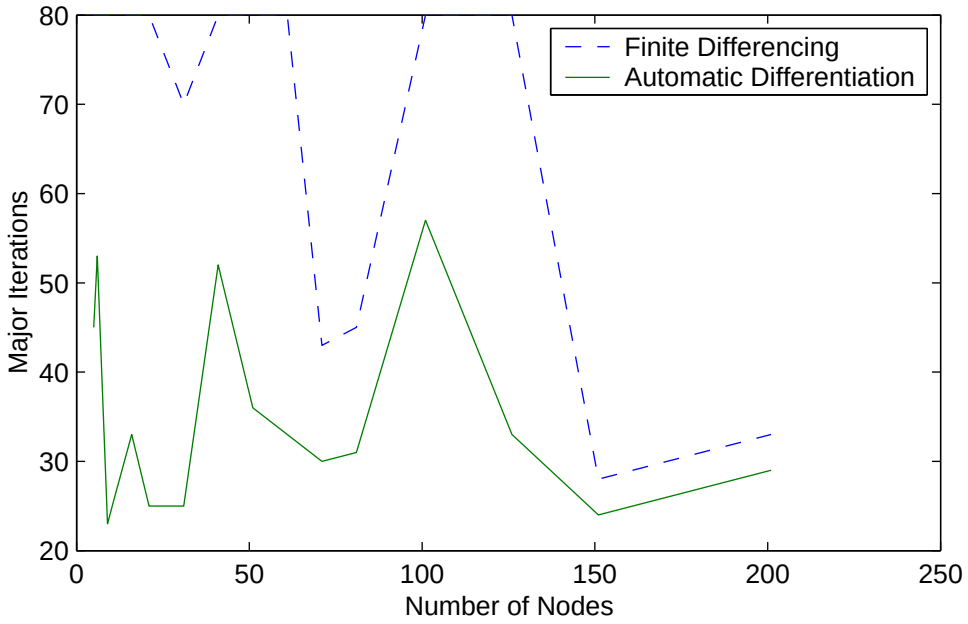

Figure 4.9: Number of Major Iterations for Solution vs. Number of Nodes

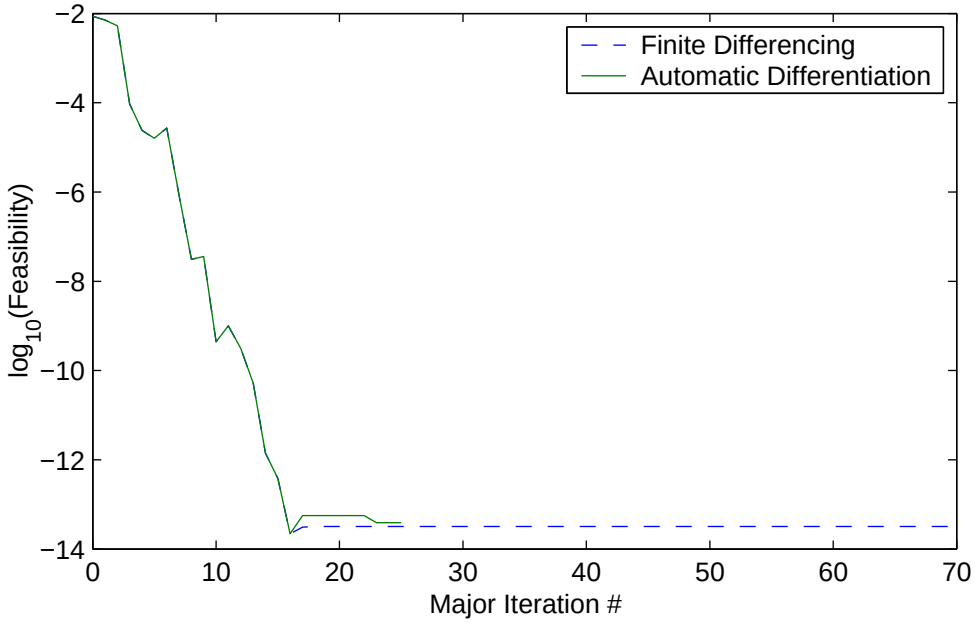

Figure 4.10: Feasibility Convergence History for 21 Nodes

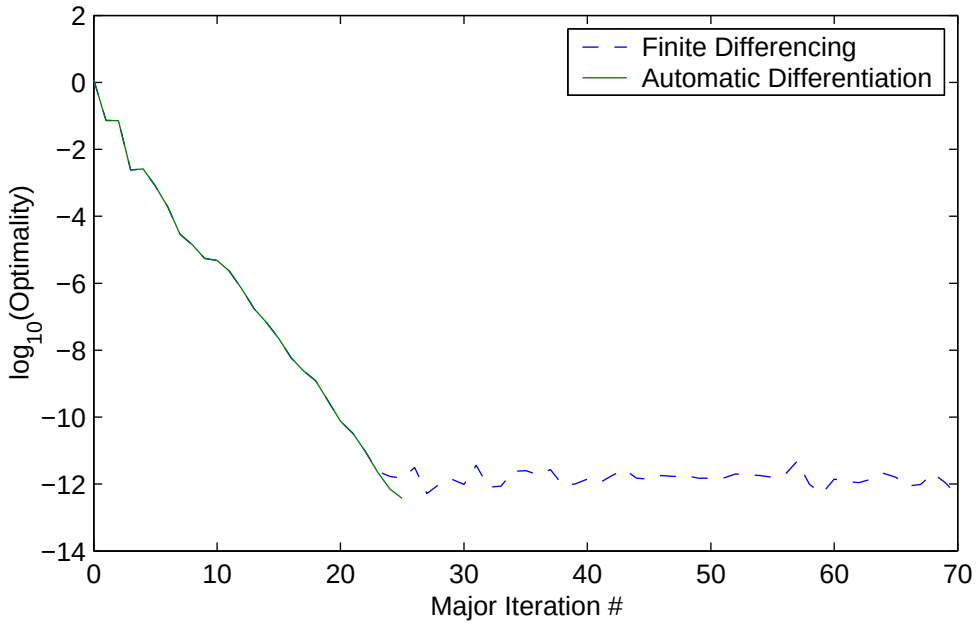

Figure 4.11: Optimality Convergence History for 21 Nodes

Between 81 and 101 nodes, the Jacobian drops below 1% nonzero elements. This, along with the problem dimensions listed, underscores the importance of using an NLP solver specifically designed for sparse problems. If a dense NLP solver was used most of the linear algebra work would involve arithmetic operations on zeros, and the solution time would grow dramatically as problem size increased.

Table 4.4: NLP Problem Sizes

#	NLP Dimensions		Jacobian Nonzeros	
Nodes	Variables	Constraints	Number	%
5	87	74	1228	19.1
6	107	90	1520	15.8
9	167	138	2396	10.4
11	207	170	2980	8.5
16	307	250	4440	5.8
21	407	330	5900	4.4
31	607	490	8820	3.0
41	807	650	11740	2.2
51	1007	810	14660	1.8
61	1207	970	17580	1.5
71	1407	1130	20500	1.3
81	1607	1290	23420	1.1
101	2007	1610	29260	.9
126	2507	2010	36560	.7
151	3007	2410	43860	.6
201	4007	3210	58460	.5

Table 4.5 shows processor timing information for the different problems. SNOPT time is the time spent in SNOPT routines doing computations, and does not include time spent for output, constraint functions, or objective functions. Constraint time is the time spent in the constraint function, including both function and gradient computation. For all cases the gradient evaluation time overwhelmed the function evaluation time, so the resulting values are primarily a reflection of the constraint gradient evaluation time. The total time columns show the total run time, including time for output and objective function evaluation. Note time spent in SNOPT and in constraint evaluation dominated total run time, with the SNOPT fraction becoming greater as the problems became larger. Another interesting trend is the growth in the SNOPT fraction. As the problem becomes larger, the constraint function evaluation becomes a smaller fraction of the overall run time–the time required by SNOPT is growing at a much faster rate than the time required for constraint evaluation. This means for many problems improving the speed of the constraint function and gradient routines, say through parallel processing, will only have a limited impact on run time. The primary place for improving speed will come from refinements of SNOPT itself.

Table 4.5: Solution Times

#	CPU Time (s)			% of Total Time		Constraint
Nodes	SNOPT	Constraints	Total	SNOPT	Constraints	Time (s)
5	2.3	3.8	6.3	36	61	0.085
6	2.6	3.9	6.7	39	58	0.073
9	2.1	3.8	6.1	34	62	0.166
11	1.4	3.6	5.2	27	79	0.140
16	2.2	3.8	6.2	36	61	0.115
21	2.5	0.5	3.0	83	16	0.019
31	4.7	4.1	9.0	52	45	0.162
41	15.4	5.5	21.1	73	26	0.106
51	16.7	4.9	21.9	76	22	0.136
61	26.1	5.4	31.9	82	17	0.164
71	29.2	5.6	35.1	83	16	0.185
81	40.8	6.1	47.2	86	13	0.198
101	165.0	13.3	178.7	92	7	0.233
126	134.4	10.1	144.9	93	7	0.306
151	167.3	9.7	177.4	94	5	0.402
201	427.5	16.6	444.6	96	4	0.571

Looking at the constraint function evaluation time per iteration, constraint function evaluation time hovers around 0.1 s/iteration until the problem size reaches about 61 nodes. At this point the time required begins to grow approximately linearly as the number of nodes increases. This would indicate certain overhead was dominating the computation until the problem reached a size large enough to make the additional node function evaluations computationally significant. The trend provides one illustration of the efficiencies possible when exploiting sparsity. When the number of constraints and variables is doubled, the total number of Jacobian elements grows by a factor of four. However, the number of nonzero elements in the Jacobian only doubles. Thus doubling the number of variables and constraints would ideally only double the time required to evaluate the Jacobian. The constraint call times roughly follow this trend with the larger problem sizes.

Chapter 5

Single-Stage-to-Orbit Ascent Problem

5.1 Description

In this example the problem is to compute the maximum-payload ascent for a single-stage launch vehicle from the surface of the Earth to some target orbit. Figure 5.1 depicts the basic geometry of trajectory and forces. $\vec{T}_I(t)$ is the thrust vector, $\vec{W}_I(\vec{r}, m)$ is the weight vector, and $\vec{F}_{A_I}(t, \vec{r}, \vec{v})$ is the aerodynamic force vector. The states are inertial position, $\vec{r_I}(t)$, inertial velocity, $\vec{v_I}(t)$, and mass, $m(t)$. Control is via the vehicle's attitude, T_{IB}, and throttle coefficient, $\eta(t)$. The transformation T_{IB} can be specified two different ways: with Euler angles or with quaternions. To demonstrate the versatility of automatic differentiation both representations are used, and comparisons are made of the results using each set of control variables.

The trajectory is modeled using three trajectory phases: (see Fig. 5.2)

1. Seven-second vertical rise
2. Full throttle phase
3. Throttled phase maintaining 3 g sensed acceleration

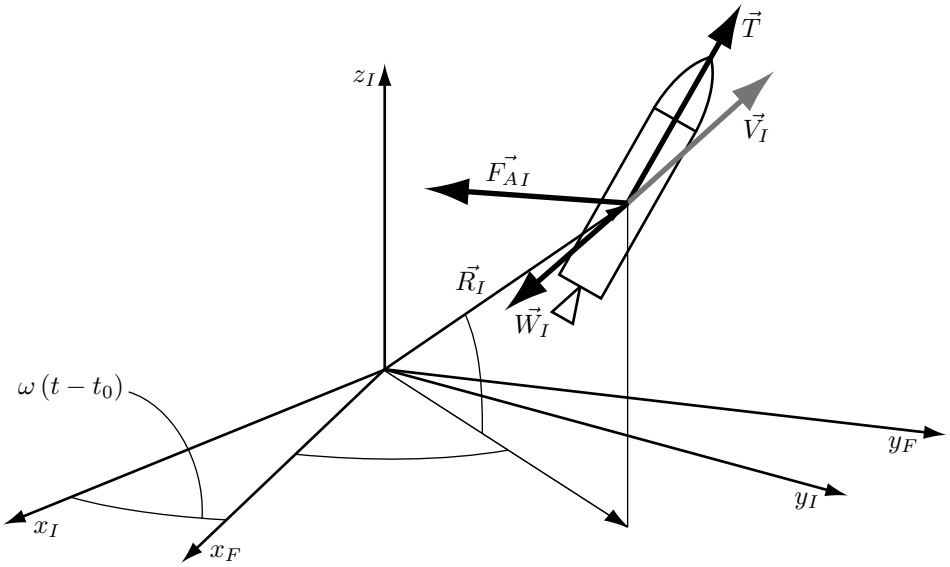

Figure 5.1: SSTO Launch Problem

The phases are divided by four events:

1. Liftoff
2. $t = 7$ seconds
3. Sensed acceleration reaches 3 g
4. Terminal conditions reached

The terminal constraints are given in table 5.1. where e is orbit eccentricity, h_{Z_I} is the inertial z-component of the angular momentum vector, and h_{eq} is the projection of the angular momentum in the equatorial plane. Specifying the angular momentum components is equivalent to specifying the semimajor axis and inclination. Note this set of terminal constraints does not

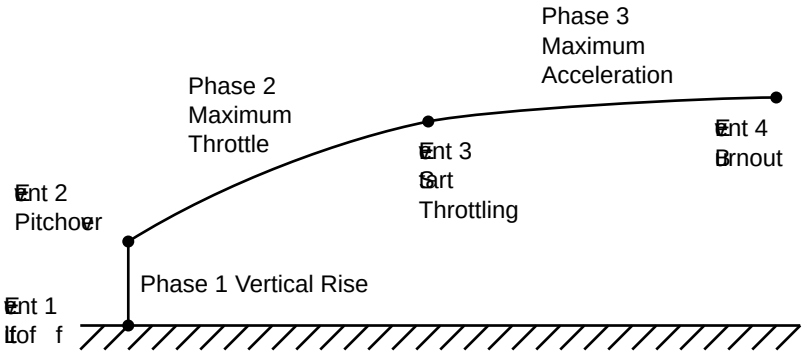

Figure 5.2: SSTO Ascent Trajectory

fix a target longitude of the ascending node, so the terminal orbit plane does not have to include the launch site.

Path constraints are summarized in table 5.2, where α, β, and ϕ are angle-of-attack, sideslip angle, and bank angle, respectively. The sensed acceleration, $\vec{a}_{\mathrm{s}}$, is given by

$$\frac{1}{m}\left\|\vec{T}\left(t\right)+\vec{F}_{A}\left(t,\vec{r},\vec{v}\right)\right\|=a_{\mathrm{s}} \tag{5.1}$$

where a_{s} is the maximum permissible "sensed" acceleration. This quantity is the reaction force between a seated crew member and the seat, and is used with the Space Shuttle to ease crew stress during ascent.

Table 5.1: Terminal Constraints

Variable	Value
e	6.9093×10^{-3}
h_{Z_I}	$2.4274 \times 10^{10}\ \mathrm{m}^2/\mathrm{s}$
h_{eq}	$4.4706 \times 10^{10}\ \mathrm{m}^2/\mathrm{s}$

Table 5.2: Path Constraints

Variable	Phase 1	Phase 2	Phase 3
α	free	$\geq -10°, \leq 30°$	$\geq -45°, \leq 35°$
β	free	$\geq -90°, \leq 90°$	$\geq -90°, \leq 90°$
ϕ	free	$0°$	$0°$
$\|\vec{a}_s\|$	$\leq 3g$	$\leq 3g$	$3g$

5.1.1 Coordinate Systems

As with any other general-purpose trajectory optimization package, there are several coordinate systems used. The coordinate systems mirror the most common systems used in POST. These are the Earth-Centered-Inertial (ECI) frame, indicated with a subscript I; the Earth-Centered-Earth-Fixed (ECEF) frame, indicated with a subscript F; the Launch frame, indicated with a subscript L; the Geographic frame, indicated with a subscript G; the Air-Path frame, indicated with a subscript W; and the Body frame, indicated with a subscript B. While some of the coordinate systems are commonly known, such as the ECI system, others are not. Appendix A gives descriptions of the different coordinate systems and the transformations relating them. The primary systems for specifying vehicle attitude in this example are the Launch and Body frames, with the controls coming from the Body-Launch frame transformation. As mentioned before, these transformations were specified using both the corresponding Euler angles and quaternion elements.

5.1.2 Equations of Motion

The equations of motion describing the SSTO flight are

$$\dot{\vec{r}}_I(t) = \vec{v}_I(t) \tag{5.2}$$

$$\dot{\vec{v}}_I(t) = \frac{1}{m(t)}\vec{F}_I\left(t, \vec{r}_I(t), \vec{v}_I(t), m(t)\right) \tag{5.3}$$

$$\dot{m}(t) = -\frac{\left\|\vec{T}(t)\right\|}{I_{sp}\, g_c} \tag{5.4}$$

where $\vec{r_I}(t)$ is the radius vector in inertial coordinates, $\vec{v_I}(t)$ is the vehicle velocity vector in inertial coordinates, $m(t)$ is the vehicle mass, $\vec{F}_I\left(t, \vec{r_I}(t), \vec{v_I}(t)\right)$ is the total force acting on the vehicle, I_{sp} is engine specific impulse, and g_c is the acceleration due to gravity at the surface of the Earth. For brevity and clarity, the states' explicit dependence on time will be omitted from this point through the rest of the problem description.

The total force vector $\vec{F}_I$ is the sum of three forces acting on the vehicle

$$\vec{F}_I(t) = \vec{T}_I(t) + \vec{W}_I(\vec{r}, m) + \vec{F}_{A_I}(t, \vec{r}, \vec{v}) \tag{5.5}$$

where $\vec{T}_I(t)$ is the thrust vector, $\vec{W}_I(\vec{r}, m)$ is the weight vector, and $\vec{F}_{A_I}(t, \vec{r}, \vec{v})$ is the aerodynamic force vector. The subscript I indicates the vectors are in the ECI frame.

5.1.3 Aerodynamic Model

Aerodynamic forces are calculated in a system rotated about the Air Path x-axis and then transformed into the inertial coordinate system. The Air Path components are lift, drag, and side force. Drag and lift are computed as

$$\vec{F}_{A_W} = \vec{D} + \vec{L} \tag{5.6}$$

$$\vec{D} = -C_D\left(M,\alpha\right) qS_{ref}\hat{e}_{x_W} \tag{5.7}$$

$$\vec{L} = C_L\left(M,\alpha\right) qS_{\text{ref}}\hat{e}_{z_W} \tag{5.8}$$

$$q = \frac{1}{2}\rho\left(\vec{r}_I\right)\left\|\vec{v}_R^R\right\|^2 \tag{5.9}$$

where ρ is the atmospheric density, C_L and C_D are the lift and drag coefficients, S_{ref} is the vehicle aerodynamic reference area, q is the dynamic pressure, M is the vehicle Mach number, α is the vehicle angle of attack, and $\vec{v}_R^R$ is the atmosphere-relative velocity. Since sideslip is constrained to be zero, it is not shown.

Data for drag coefficient and lift coefficient for a vehicle configuration studied at Langley Research Center were provided as tables of points at a series of Mach number and angle-of-attack values. An initial model used a least-squares curve fit with tensor-product cubic B-splines as the basis functions. Attempts to use these models met with little success except on very coarse meshes. The NLP solver would fail to converge on a refined mesh.

As an alternative to tensor-product splines a model was constructed using an approach similar to that in Gottlieb[28]. First, model equations were

fit to the data at each Mach number. Lift coefficient was approximated using a cubic

$$C_L(\alpha) = a\alpha^3 + b\alpha + c \tag{5.10}$$

The lack of a quadratic term forces an inflection point at $\alpha = 0$, a feature noticed in the hypersonic data. Drag coefficient was modeled using a quadratic

$$C_D(\alpha) = a\alpha^2 + b\alpha + c \tag{5.11}$$

Once the curve fits were made at each M, a least-squares cubic spline was fit to the coefficients as a function of Mach number. The resulting models are

$$C_L(M, \alpha) = a_{C_L}(M)\,\alpha^3 + b_{C_L}(M)\,\alpha + c_{C_L}(M) \tag{5.12}$$

$$C_D(M, \alpha) = a_{C_D}(M)\,\alpha^2 + b_{C_D}(M)\,\alpha + c_{C_D}(M) \tag{5.13}$$

Temperature, density, and pressure are based on a modified 1962 U.S. Standard Atmosphere model, similar to one used on the same model in POST[1, 49]. The POST model departs from the 1962 atmosphere model above a geometric altitude of about 90 km by assuming the molecular mass of air remains constant with altitude. In the 1962 standard the molecular mass of air changes with altitude above about 90 km. The model used for this example uses least-squares B-spline curve fits to the temperature, pressure, and density variations

from the POST model. Speed of sound is computed from temperature by

$$a = \frac{\gamma R^* T}{M_0} \tag{5.14}$$

where T is the temperature, γ is 1.40, R^* is 8314.32 N·m/kmol·K, and M_0 is 28.9644 kg/kmol.

5.1.4 Gravity and Thrust Model

Weight is computed in the Geographic coordinate system using Newton's inverse-square law

$$\vec{W}\left(\vec{r_G}, m\right) = -\frac{\mu m}{\left\|\vec{r}_G\right\|^3}\vec{r}_G \tag{5.15}$$

where μ is Earth's gravitational parameter. No oblateness, third-body, or other gravitational perturbations were used. The vectors are expressed in the Geographic coordinate system because those perturbations are very small compared to the magnitude of the thrust force.

Thrust is modeled assuming a constant vacuum specific impulse rocket engine firing along the Body frame x-axis

$$\vec{T}_B\left(t\right) = \left[\eta\left(t\right) T_{vac} - P\left(\vec{r}\right) A_e\right] \hat{e}_{x_B}\left(t\right) \tag{5.16}$$

where A_e is the total engine nozzle exit area, $P\left(\vec{r}\right)$ is the atmospheric pressure, and $\hat{e}_{x_B}\left(t\right)$ is the unit vector in the x-direction in the Body coordinate system. This model does not allow for gimballing of the engine.

5.2 Results

The results for this problem are presented in three subsections: the first describes the trajectory itself, the second describes the differences which arose based on the type of derivative computation used, and the third describes differences arising from the choice of control variables.

All trajectories were computed with optimality and feasibility tolerances of 10^{-9} and major iteration limits of 150 iterations. All cases failed to converge on the first mesh, and a refinement was performed after the major iteration limit was reached on the first mesh. In figures which plot quantities as a function of iteration, the first iteration on the new mesh is iteration 151. None of the cases finished by satisfying the specified optimality and feasibility tolerances. Instead, optimality would plateau around 10^{-6} and feasibility around 10^{-7} or 10^{-8}. A second mesh refinement did not improve the results, while a third refinement could not be attempted because of insufficient memory. The workspace required by SNOPT for solving the problem from a third mesh refinement pass exceeded the memory available in the computer. A possible workaround is mentioned in Chapter 6, under suggestions for further work.

5.2.1 Trajectory Description

The vehicle's flight is broken into three major trajectory phases, with different control behavior in each phase. The first phase is a fixed-attitude, full-throttle segment where the vehicle rises at a fixed 89^{o} pitch attitude for

seven seconds. Pitch was held at 89° rather than 90° because yaw and roll are no longer independent with a pitch of 90°, and this introduces a singularity in one of the coordinate transformations. Next is a maximum-throttle phase where attitude is controlled to optimize the trajectory. In the third phase the engine throttle parameter is allowed to vary along with the attitude variables, and the sensed acceleration path constraint is imposed. If attitude is controlled using a quaternion rather than Euler angles, an additional path constraint is imposed on all phases to force the quaternion to be a unit quaternion.

The trajectory is illustrated in figures 5.3-5.5. Figure 5.3 shows the vehicle's ground track and altitude history. Flight starts at an altitude of 0 m and continues until the terminal conditions are reached at an altitude of 90,000 m. Thus, the vehicle reaches the target orbit at the orbit's perigee even though this is not an explicit requirement. Figure 5.4 shows an enlargement of the first portion of the ground track where a noticeable kink occurs in the flight path. After finishing the vertical rise the vehicle makes a very small, gradual turn left from a heading of about 93° through about 92°, at which point it turns more quickly to a heading of 90.2°. As the rocket continues to accelerate it slowly turns slightly south, as would be typical of a vehicle staying within a certain plane in inertial space. Final position over the ground is 28.1° N, 70.7° W. Figure 5.5 shows the vehicle's longitude of the ascending node and inclination through the flight. While inclination changes over the course of the trajectory, it stays within .002° of 28.6° throughout the flight. Similarly, longitude of the ascending node has relatively small change over the

course of the flight–less than 1°.

Figure 5.6 shows speed, heading and flight-path-angle for the entire flight. The heading change mentioned earlier is obvious, occurring between about 60 seconds and 70 seconds into the flight. Flight path angle has a small oscillation in the early portion of the flight which is a consequence of the pitch history, while speed increases monotonically until the end of the trajectory. Final velocity is 7,880 m/s, at a flight path angle of .02° and a heading of 95.6°.

The vehicle's mass history is shown in figure 5.7, along with the throttle setting history. As engine thrust is reduced to maintain the acceleration limit fuel flow drops off. Burnout mass is 140,300 kg, making the ratio of final mass to initial mass .12, somewhat higher than would be expected.

Figure 5.8 shows dynamic pressure, which peaks at 24,600 Pa. No dynamic pressure limit was explicitly included because of the complication of having a state path constraint, but the maximum here is within the flight experience of the space shuttle and below maximum values used in other studies, such as Tartabini, et. al. [56].

Pitch, roll, and yaw for the entire flight are shown in figure 5.9. A discontinuity is allowed in the control history at the end of the vertical rise and there is a discontinuity in all three angles at this point in the optimal trajectory. The vehicle begins the second phase with a pitch of 77°, decreasing pitch slightly before entering a pitch-up maneuver at 19 seconds. Maximum

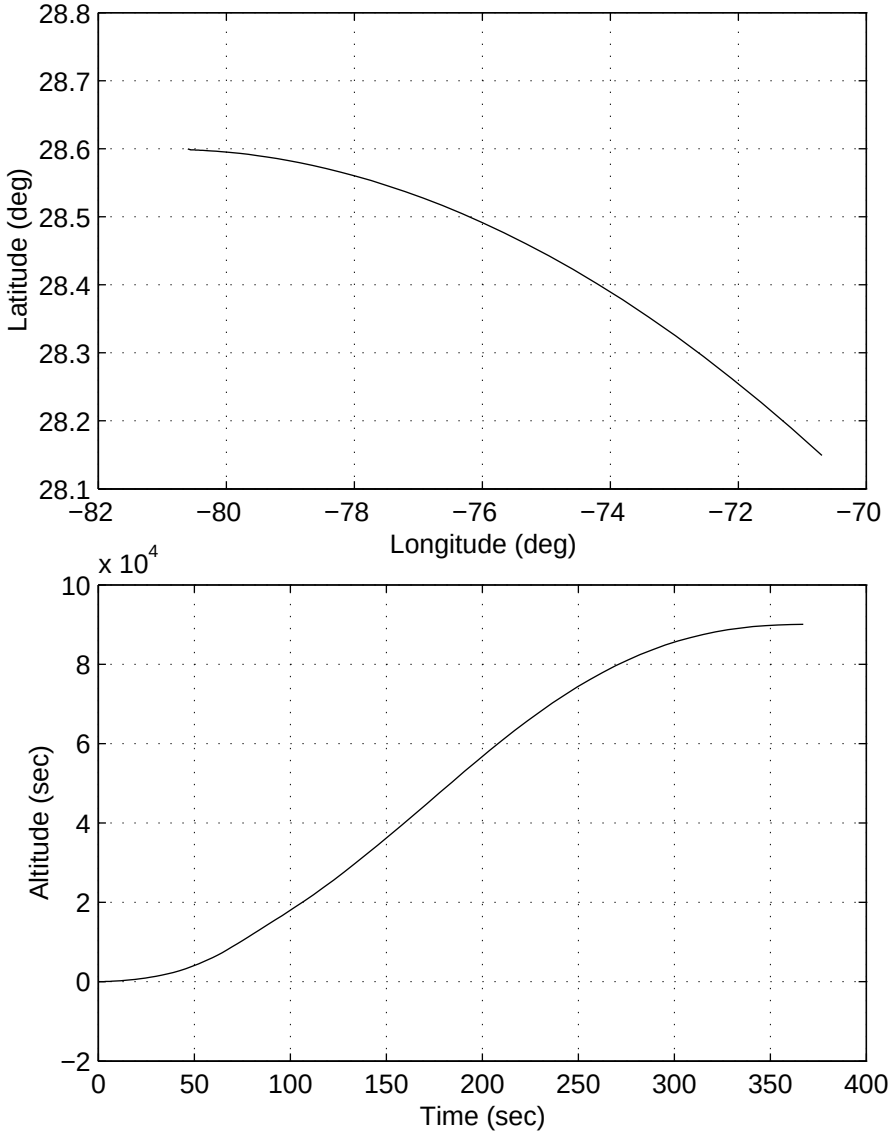

Figure 5.3: Ground Track and Altitude History

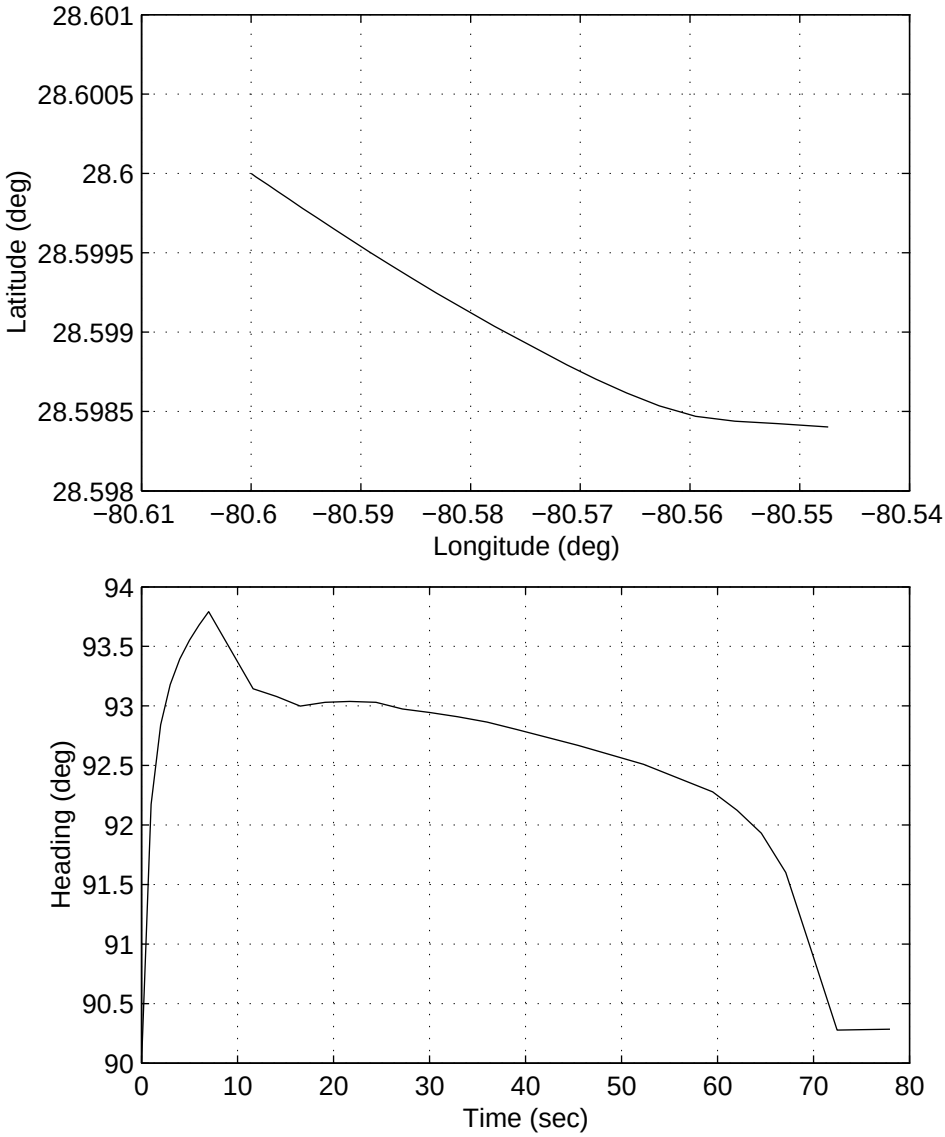

Figure 5.4: Early Ground Track

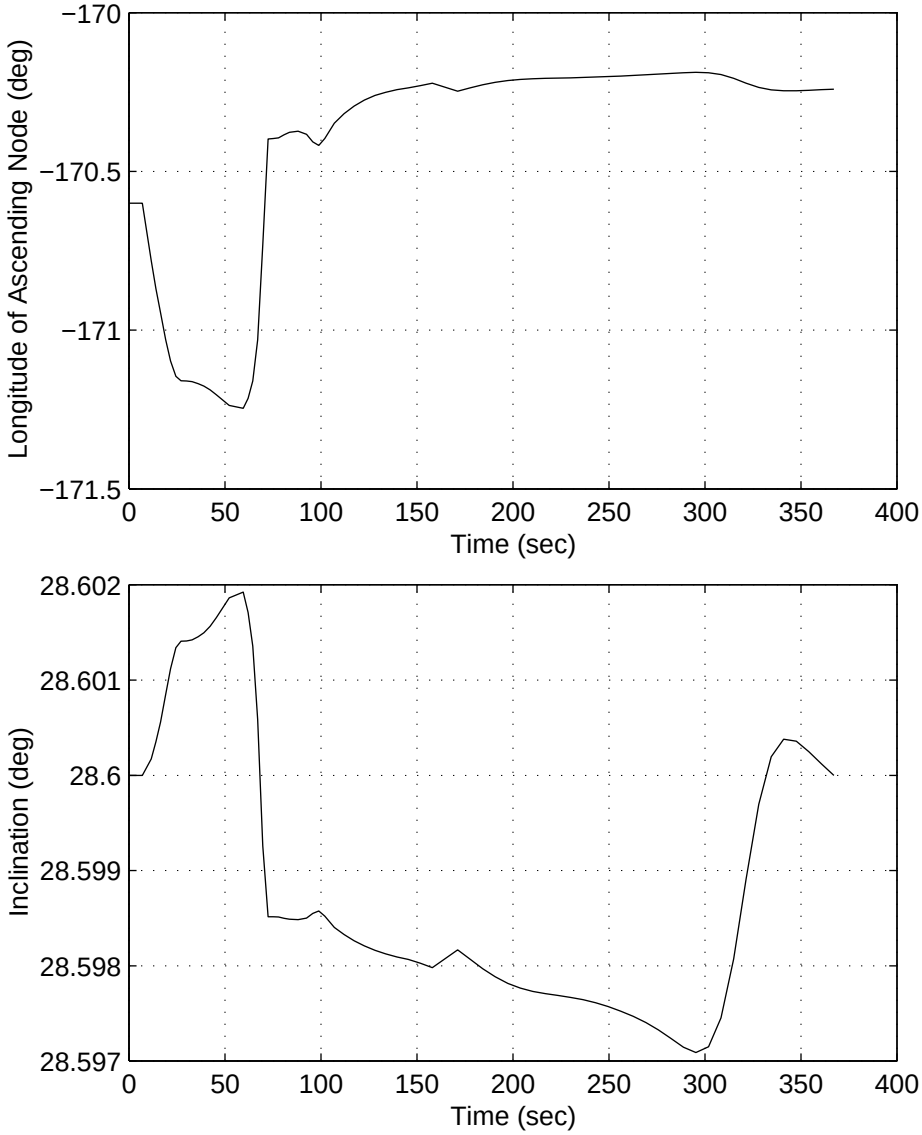

Figure 5.5: Inclination and Longitude of the Ascending Node

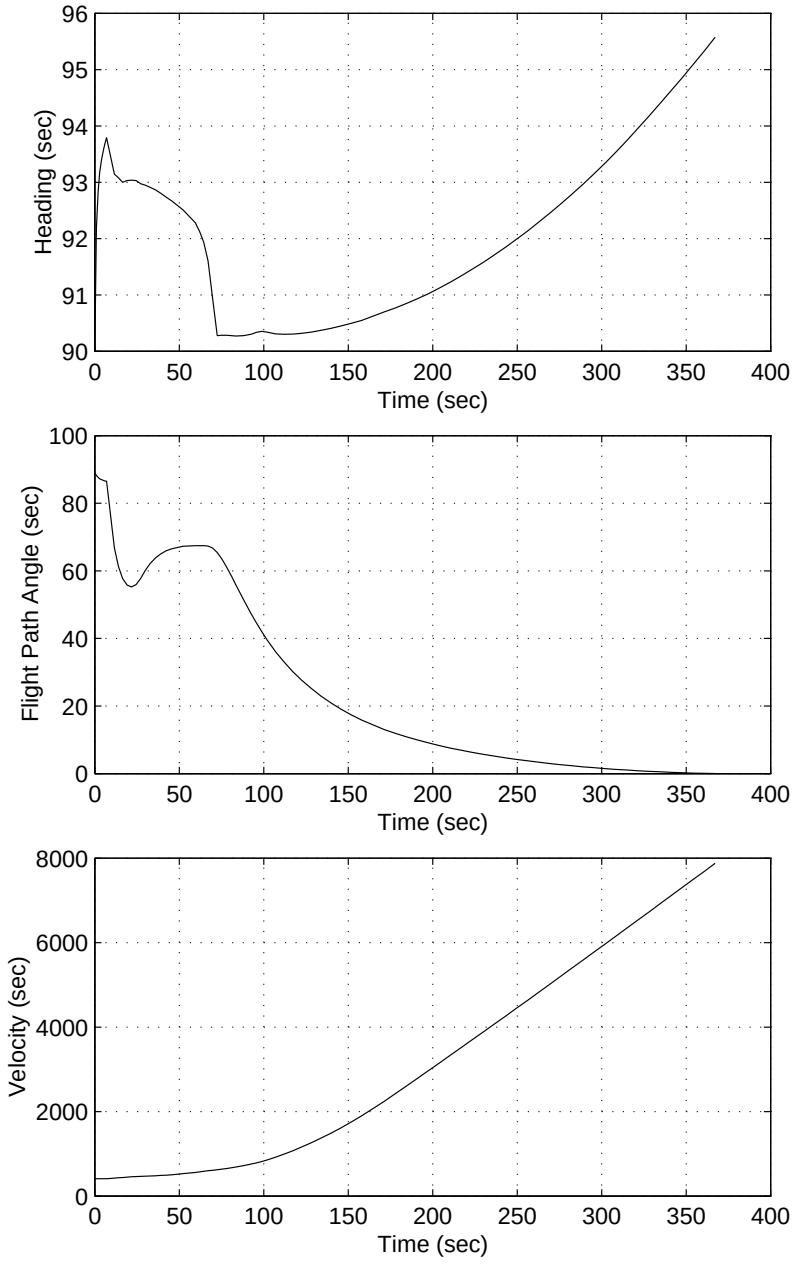

Figure 5.6: Heading, Flight Path Angle, and Velocity Histories

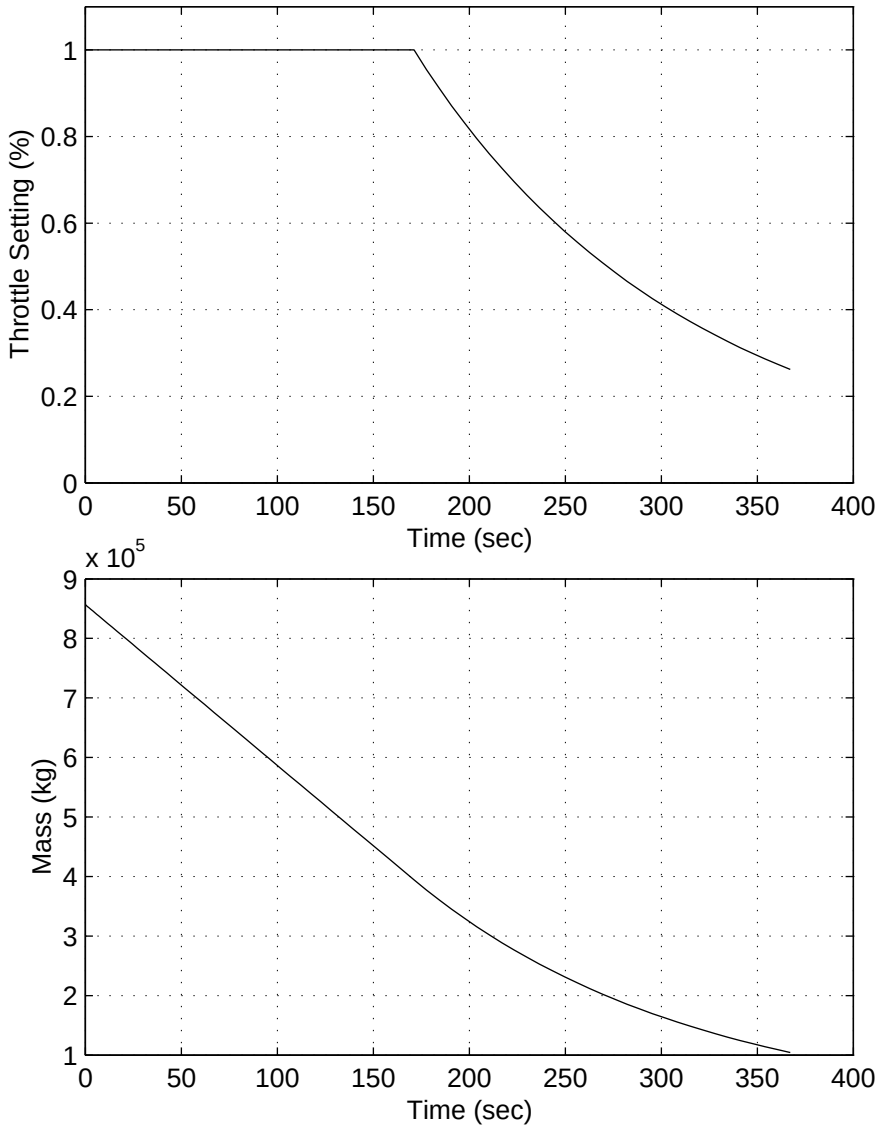

Figure 5.7: Mass and Throttle Histories

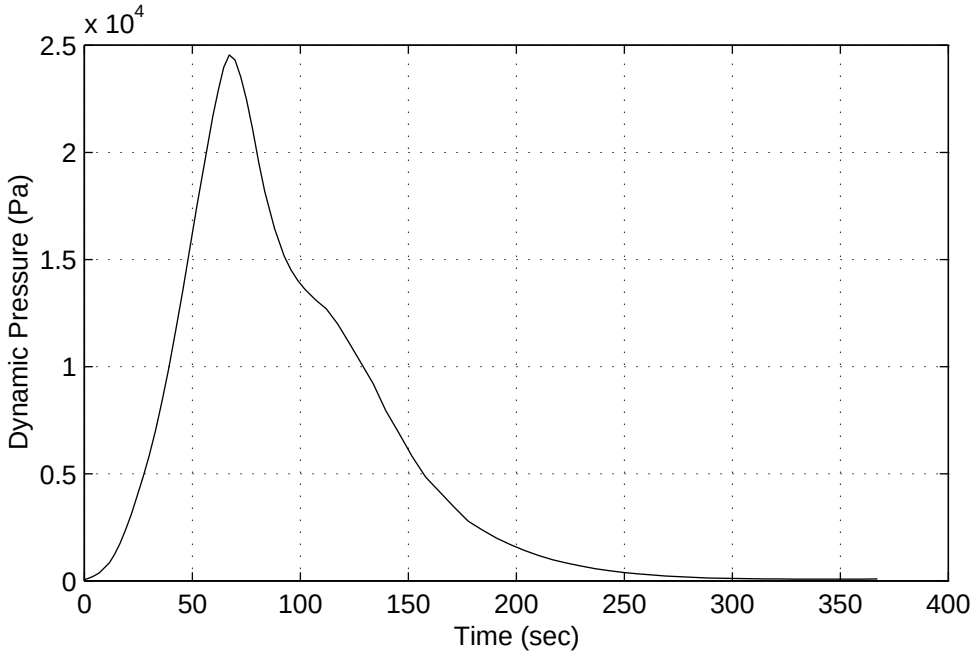

Figure 5.8: Dynamic Pressure History

angle-of-attack, 18.5°, is reached at 24.4 seconds. The pitch-up maneuver ends at 30 seconds as the vehicle normal force boundary is reached. Once the vehicle reaches the normal force limit it maneuvers to stay on the limit for 37.1 seconds. This can be seen better in figure 5.10, which shows pitch and normal force during the first two phases of the flight. As it leaves the normal force boundary, the launcher goes through a series of roll and yaw maneuvers to bank the vehicle about its velocity vector. Figure 5.11 shows bank angle, lift coefficient, and the cross-track component of the total force on the vehicle during this portion of the flight, from 42 seconds to 117 seconds. The vehicle banks to the left until reaching -16.8°, at 69.8 seconds, when it begins to reverse the bank. It passes through 0° of bank at 74.9 seconds, just before the vehicle's lift coefficient changes sign at 76.4 seconds, and remains slightly positive during most of the time lift coefficient is negative. As the vehicle begins to develop positive lift again it banks back to the left, with another peak of -5° at 102 seconds, before reducing bank to near-zero. The combination of banking and lift coefficient has the effect of keeping the cross-track component of the vehicle's force vector almost entirely positive while lift coefficient becomes slightly negative. The reason for the negative lift coefficient is obvious in figure 5.12, which shows angle-of-attack, bank, and mach number during the first 150 seconds of flight: the vehicle reduces angle-of-attack to maintain near-zero zero lift, and thus minimum drag, during transonic flight. As it leaves the transonic regime, the launcher begins to develop some lift again. During the second half of the flight pitch, roll, and yaw, are slowly and smoothly varied

to optimize the thrust direction and reach the terminal constraints. By this time in the flight the vehicle is out of the sensible atmosphere, and there is no more maneuvering to take advantage of aerodynamic forces.

Looking again at figure 5.9 there is a gradual decrease in roll in the last phase of the flight, followed by a reversal. Figure 5.13 shows angle-of-attack is decreasing and bank is reversing during this portion of the flight. As the vehicle decreases angle-of-attack through zero, it tries to keep the thrust vector pointed slightly left of the velocity vector while maintaining the zero sideslip condition. The bank reversal insures the thrust vector continues to point left of the velocity vector after angle-of-attack becomes negative, which helps bring inclination to its specified terminal value of 28.6°.

Figure 5.14 presents the throttle history and sensed acceleration magnitude for the flight. Throttling to maintain the sensed acceleration limit begins at 171 seconds and continues until the end of the flight. The throttle setting at the end of the trajectory is 26.2%. Note the peak in sensed acceleration due to the pull-up maneuver during the early flight, followed by the valley during transonic flight.

5.2.2 Automatic Differentiation vs. Finite Differencing

Figure 5.15 shows the lack of influence the type of derivative computation has on SNOPT's feasibility criteria. While there is some difference during the middle iterations on the first mesh, during the majority of the problem there is little difference in feasibility. When using quaternions with finite dif-

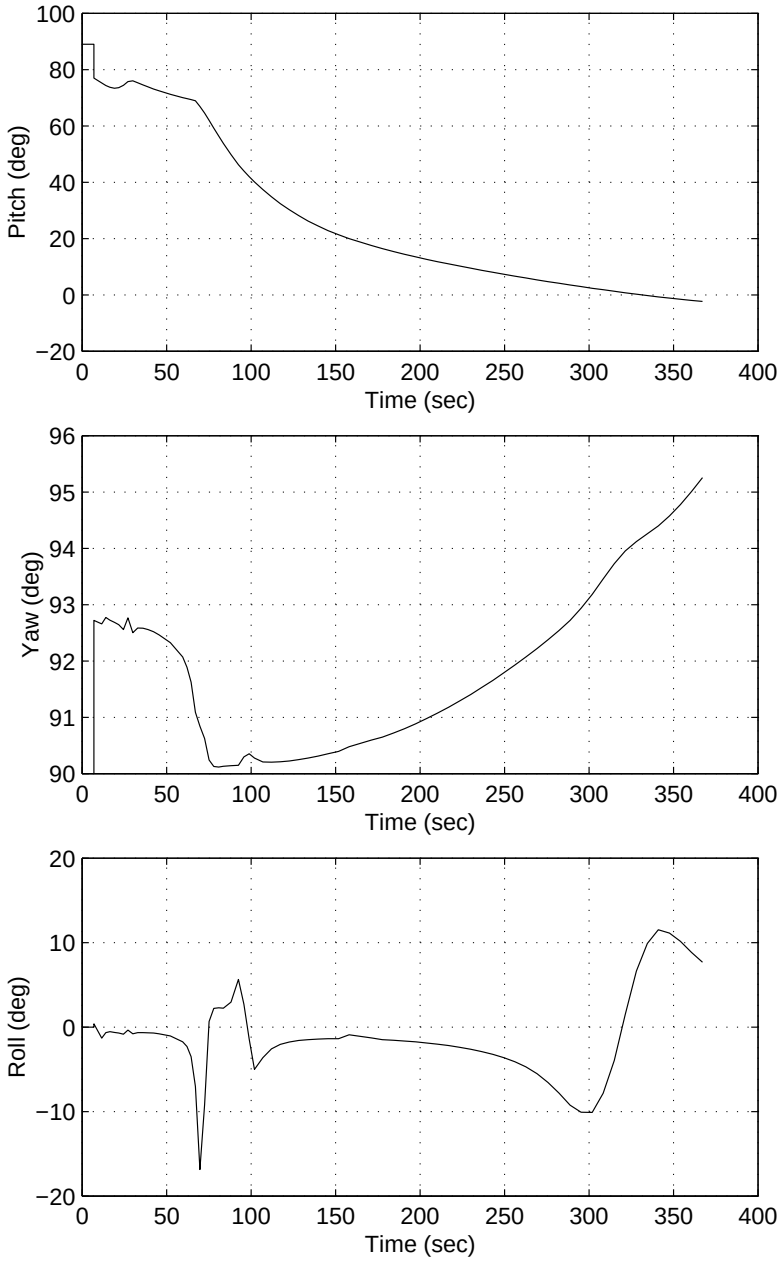

Figure 5.9: Pitch, Roll, and Yaw Histories

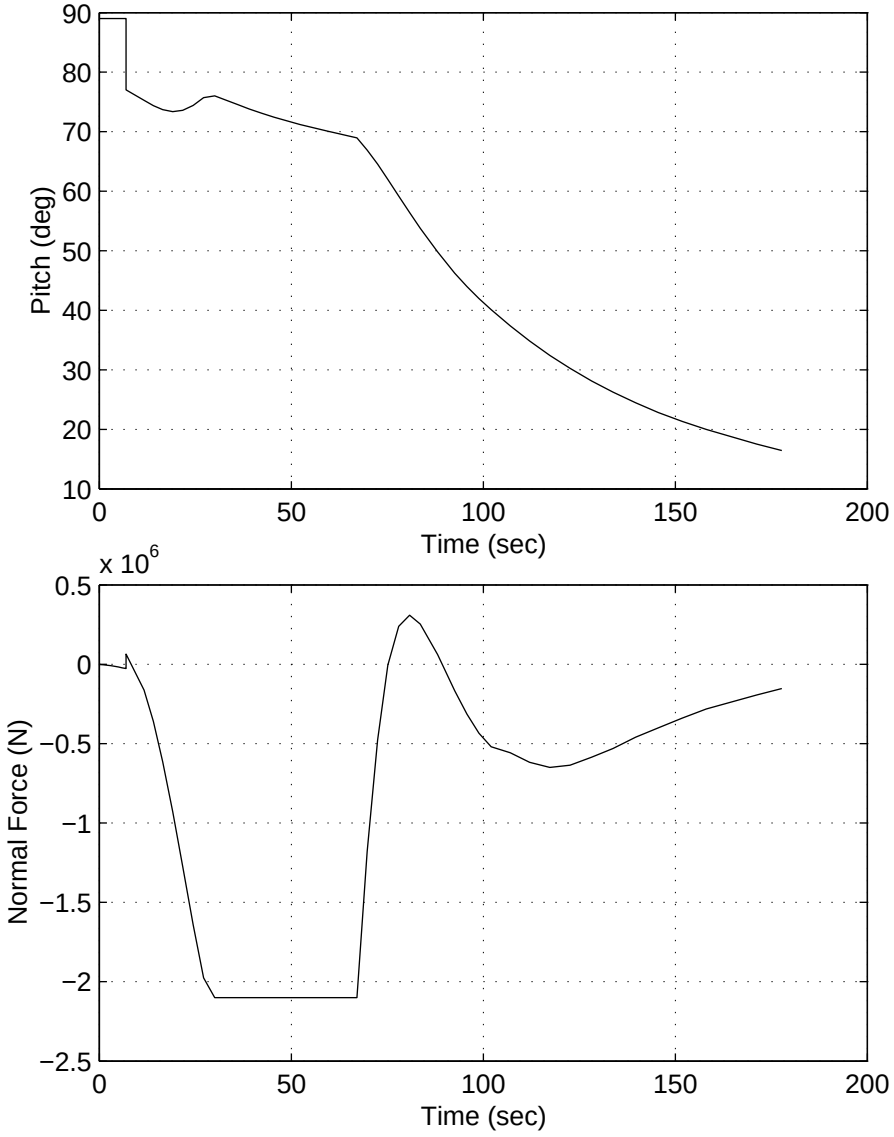

Figure 5.10: Pitch and Normal Force Histories

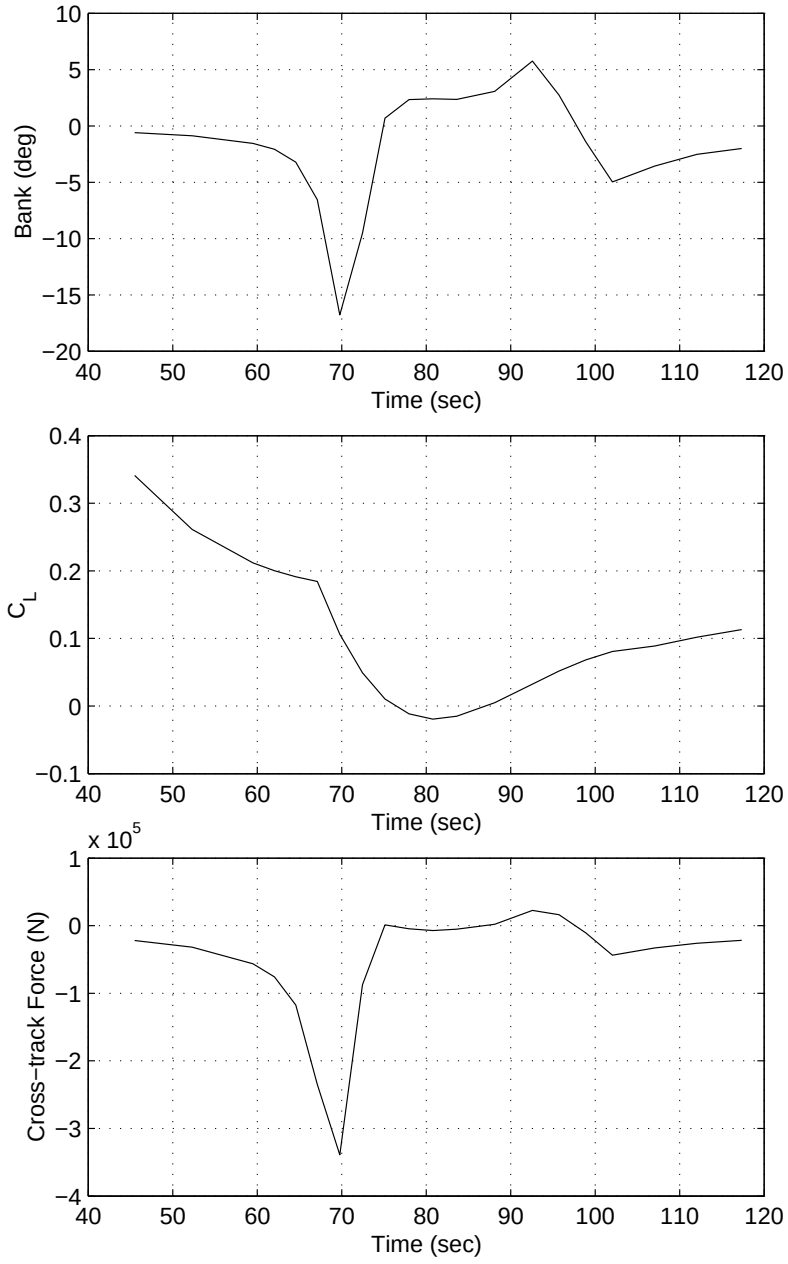

Figure 5.11: Vehicle Bank, Lift Coefficient, and Mach Histories

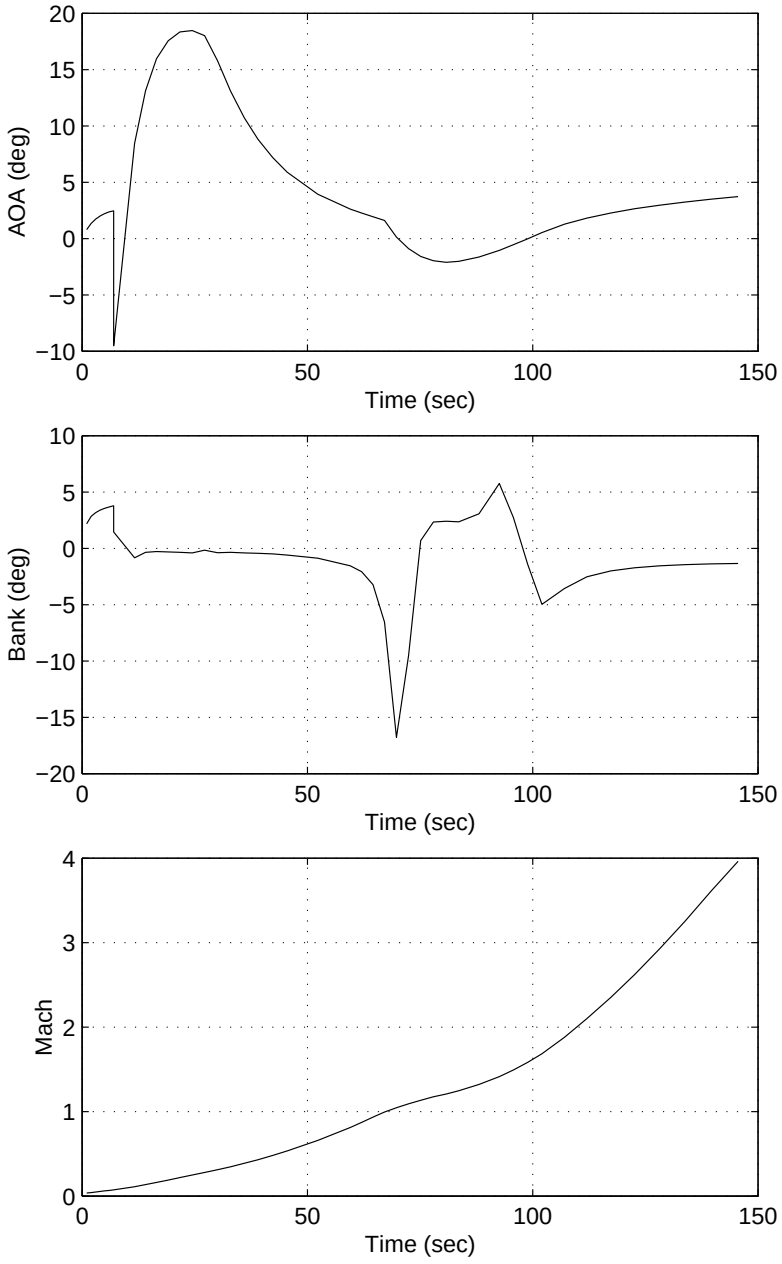

Figure 5.12: Angle-of-Attack, Bank , and Mach Histories

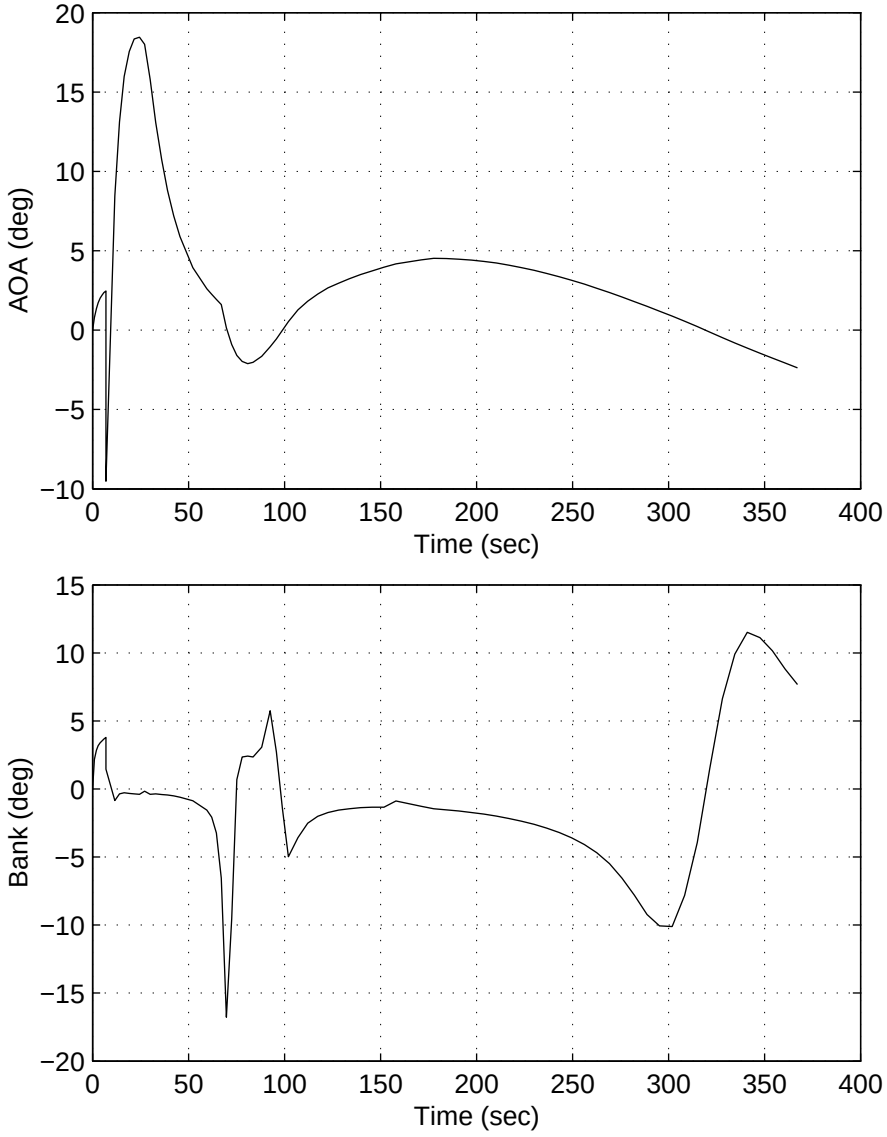

Figure 5.13: Angle-of-Attack and Bank Histories for the Entire Trajectory

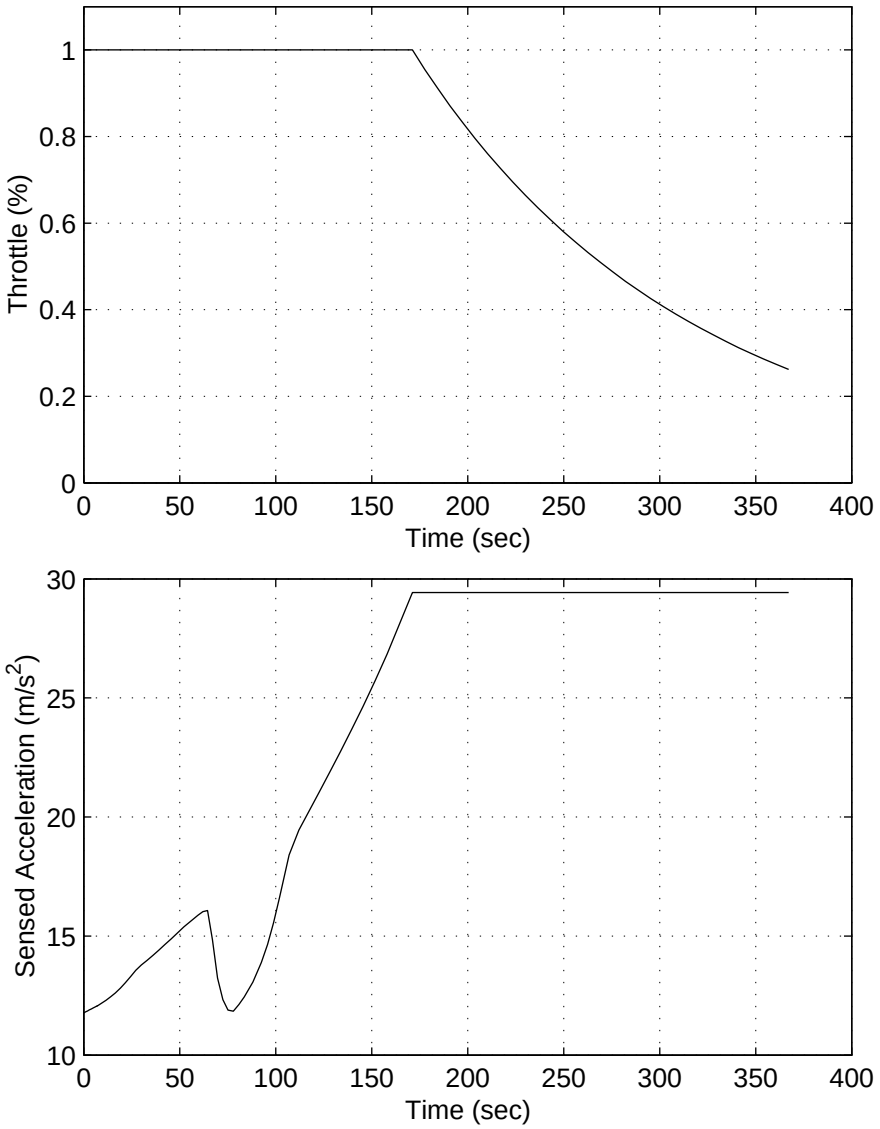

Figure 5.14: Throttle History

ferencing there is a plateau around 10^{-4} between about 30 iterations and 75 iterations, while there is a smaller and higher peak in feasibility when using automatic differentiation. For the trajectory using Euler angles as the control variables and finite differencing for derivatives there is a similar plateau in feasibility, but higher than the case using quaternions. When Euler angles are combined with automatic differentiation there is a lower, narrower plateau. These plateaus are discussed more in the following subsection. All the cases briefly pass below 10^{-8} for feasibility but this level is not sustained, and a steady-state feasibility of about 10^{-7} is reached within 50 iterations on the second mesh. The expected behavior for this problem is using finite differencing would result in somewhat slower or poorer feasibility than using automatic differentiation, and if feasibility could be driven lower a difference would likely show up as it did in the lunar launch problem.

Figure 5.16 presents SNOPT's optimality criteria in the same way figure 5.15 presents feasibility. While there is little difference in feasibility convergence between trajectories using different derivative computation methods, there is somewhat more difference in optimality convergence. Using finite differencing results in significantly worse optimality than using automatic differentiation until after 50 iterations, regardless of the type of control variable. With quaternion elements as control variables, optimality drops below 10^{-4} in 29 iterations when automatic differentiation is used but takes 72 iterations when finite differencing is used. For the cases with Euler angles as control variables, optimality reached the 10^{-4} level in 39 iterations using automatic

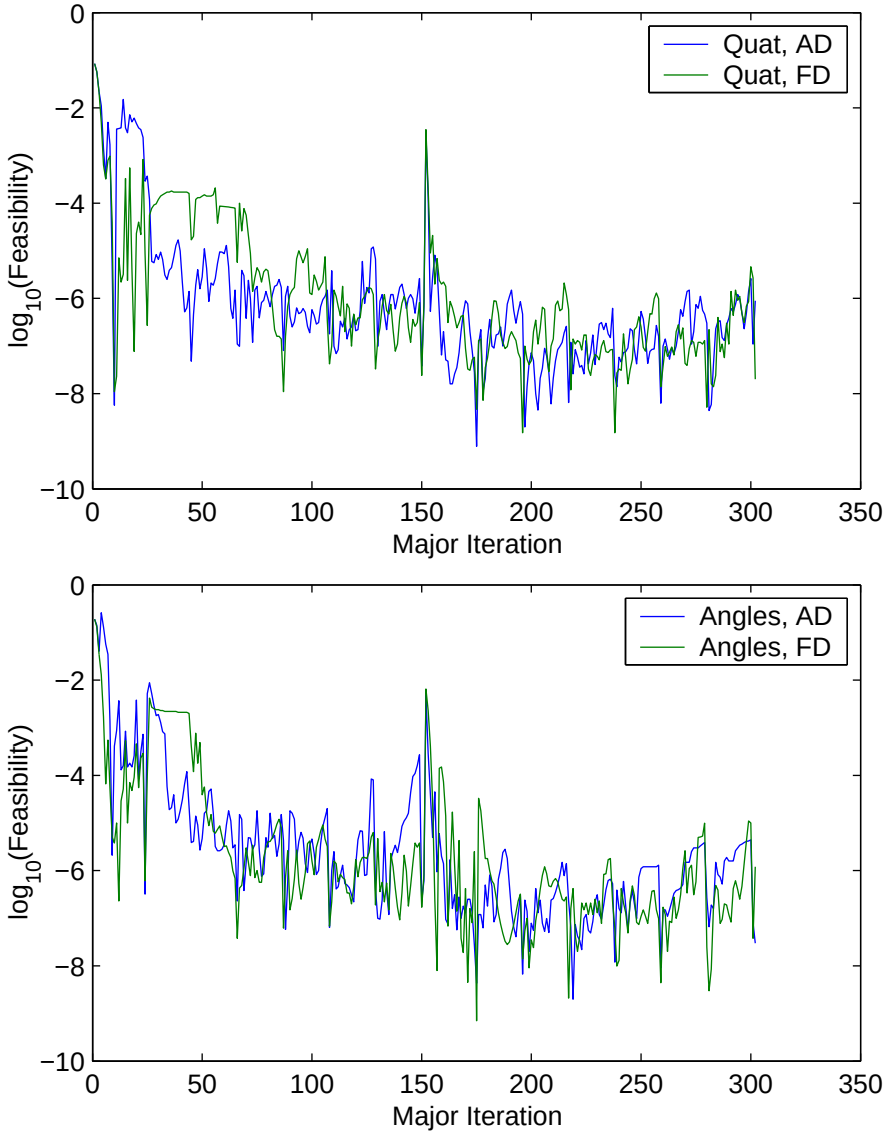

Figure 5.15: Derivative Impact on Feasibility Convergence

differentiation and 57 iterations using finite differencing. When quaternions are used, optimality remains worse for the finite difference case until 105 iterations. As with feasibility, there is floor of sorts below which optimality does not descend, in this case the floor is at about 10^{-6}.

The likely reason for the slower initial convergence in optimality when finite differencing is used is that SNOPT takes longer to establish a good basis when finite differencing is used–until the basis is established an NLP code does not have a good idea of which variables should change and which should remain at their bounds. The basis in an NLP code is the partitioning of the variables into basic, superbasic, and those at bounds, and once an SQP method has acquired a reasonable basis it ideally requires one minor iteration per major iteration. Figure 5.17 shows the difference in the number of minor iterations caused by the type of derivative calculation. The first graph compares finite differencing against automatic differentiation for the case of quaternion elements as control variables, while the second graph is the same comparison but with Euler Angles as the control variables. Regardless of the type of control variables, using finite difference derivatives drew out the number of iterations required to establish a good basis with a more pronounced effect when quaternions were used than when Euler angles were used. For the quaternion cases, the minor iteration count dropped to one in 26 and 69 major iterations when automatic differentiation and finite differencing were used, respectively. The combination of Euler angles and automatic differentiation required 37 major iterations to establish a basis, while the combination of

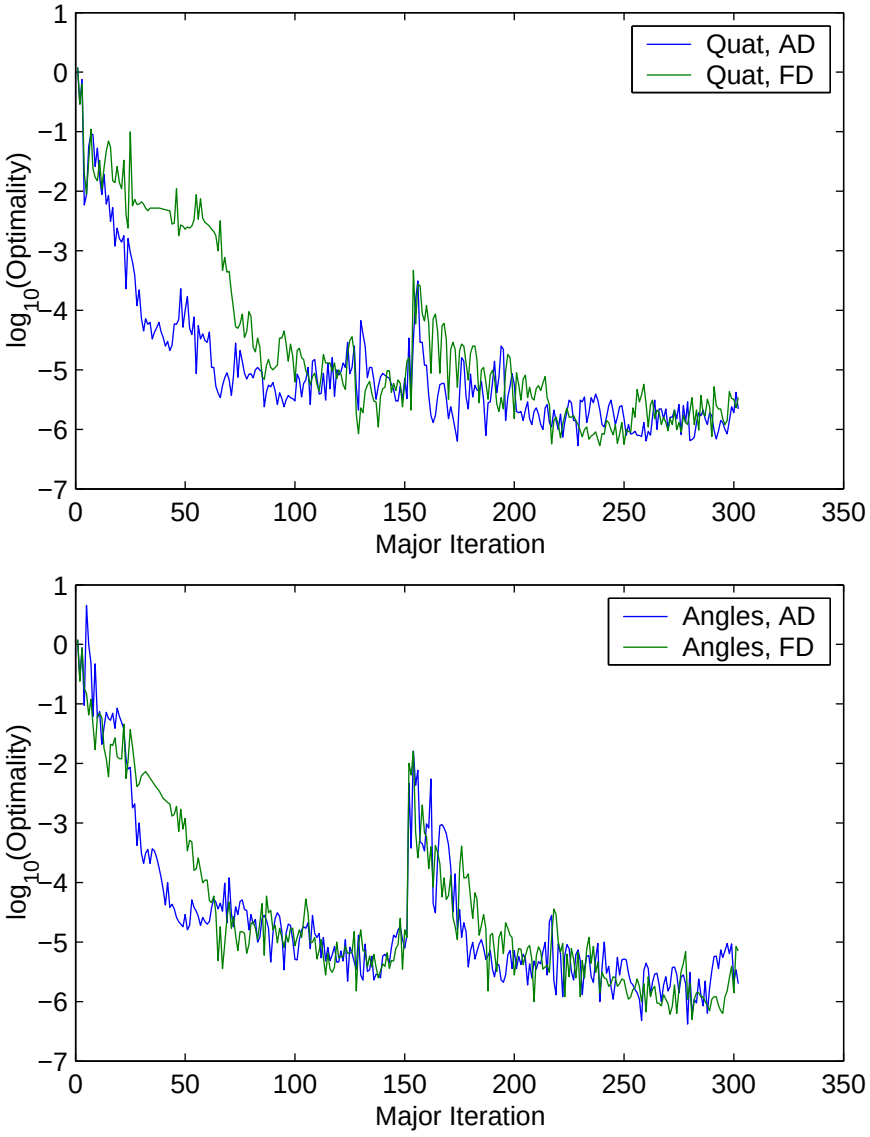

Figure 5.16: Derivative Impact on Optimality Convergence

Euler angles and finite differencing required 53 major iterations. After a mesh refinement pass all cases exhibited a brief spike while SNOPT computed an initial basis for the new NLP problem, but there was no significant difference in the number of major iterations required for this transient due to the type of derivatives used.

5.2.3 Quaternions vs. Euler Angles

Figure 5.18 presents the same data as figure 5.15 but with grouping to highlight the similarities and differences between using quaternion elements and Euler angles as control variables. When automatic differentiation is used to compute derivatives, using quaternions results in slightly better convergence on the first mesh, but once the mesh is refined there is no significant difference. When finite differences are used there is little difference between them. As mentioned earlier, both cases have a large initial improvement in feasibility, followed by a worsening which plateaus, and then fairly steady and rapid improvement. When quaternion elements are the control variables, the plateau occurs at just above the 10^{-4} level and lasts for approximately 40 iterations. On the other hand, when Euler angles are used the plateau is above 10^{-3} but only lasts about 20 iterations. These plateaus are more like wide peaks than plateaus when automatic differentiation is used, and they have about the same magnitude. The end of these feasibility plateaus coincides with the establishment of a good basis for the problem, which is not surprising.

Figure 5.19 compares optimality convergence for cases using the same

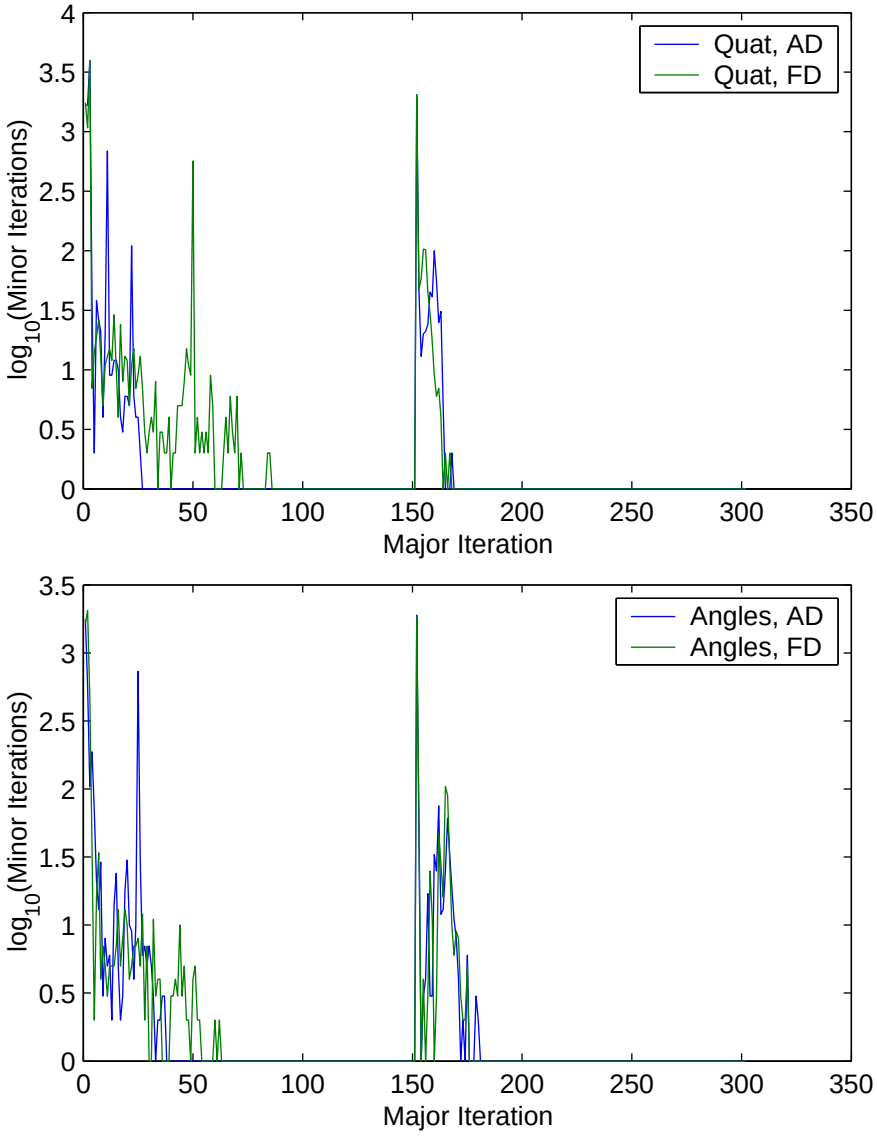

Figure 5.17: Derivative Impact on Minor Iterations

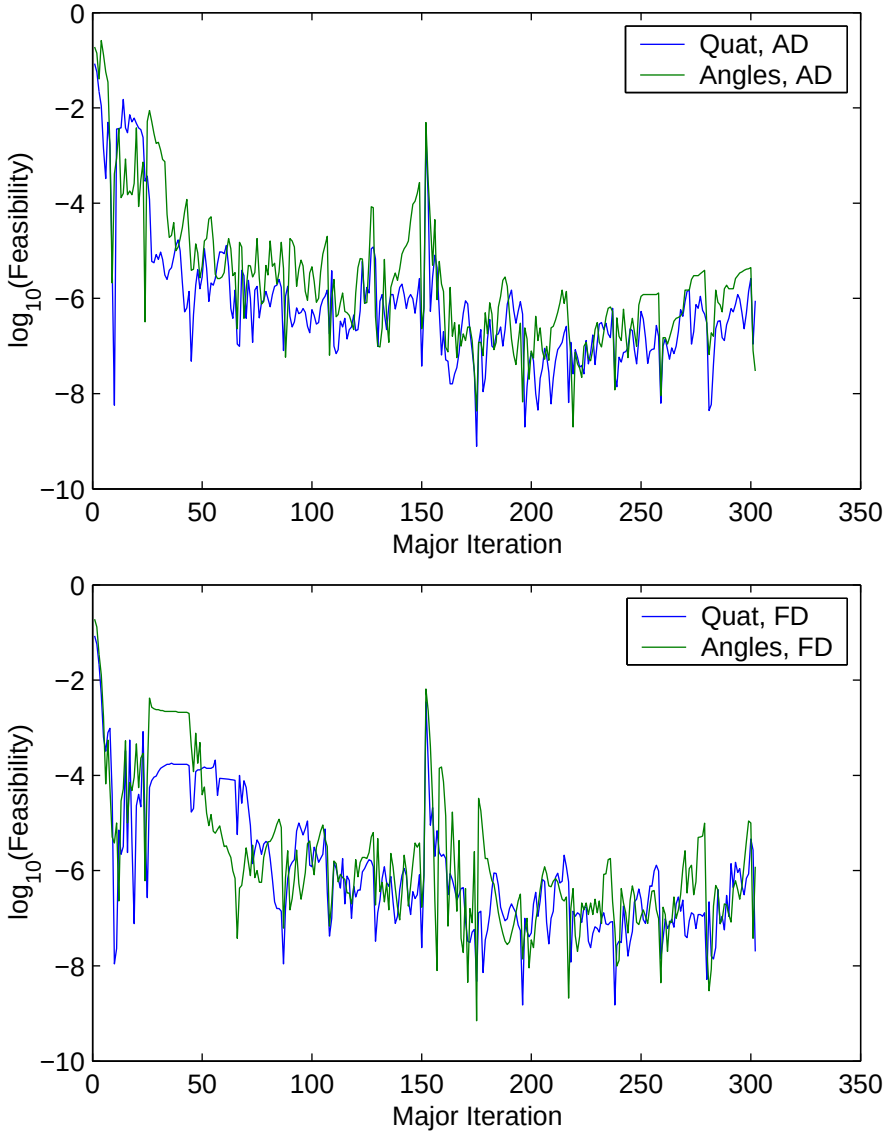

Figure 5.18: Control Variable Impact on Feasibility Convergence

type of derivative computation. When using automatic differentiation quaternion elements provide the same or slightly better optimality convergence than Euler angles, with most of the difference occurring in the initial iterations on each mesh when SNOPT is trying to compute a good basis. When finite differencing is used for derivatives there is again little difference in convergence due to the type of control variables used. What difference occurs happens in the startup period on each mesh.

Figure 5.20 presents the minor iteration histories from figure 5.17, but grouped by type of derivative computation. The primary feature is the basis is established more quickly on the refined meshed when quaternions were used. For the initial mesh, using quaternions provided faster startup only when automatic differentiation is used.

5.3 Computational Considerations

The code used to generate the results in this chapter is identical to that used for the lunar launch example, with the exception of the function and gradient computation routines. Function and gradient computations were written in Java, like the previous example. The same computer was also used: a Power Macintosh G4 with dual 450MHz PowerPC G4 processors running Yellow Dog Linux 2.3.

Table 5.3 shows the NLP dimensions for the cases using automatic differentiation. Results are shown for both meshes in each solution. Because

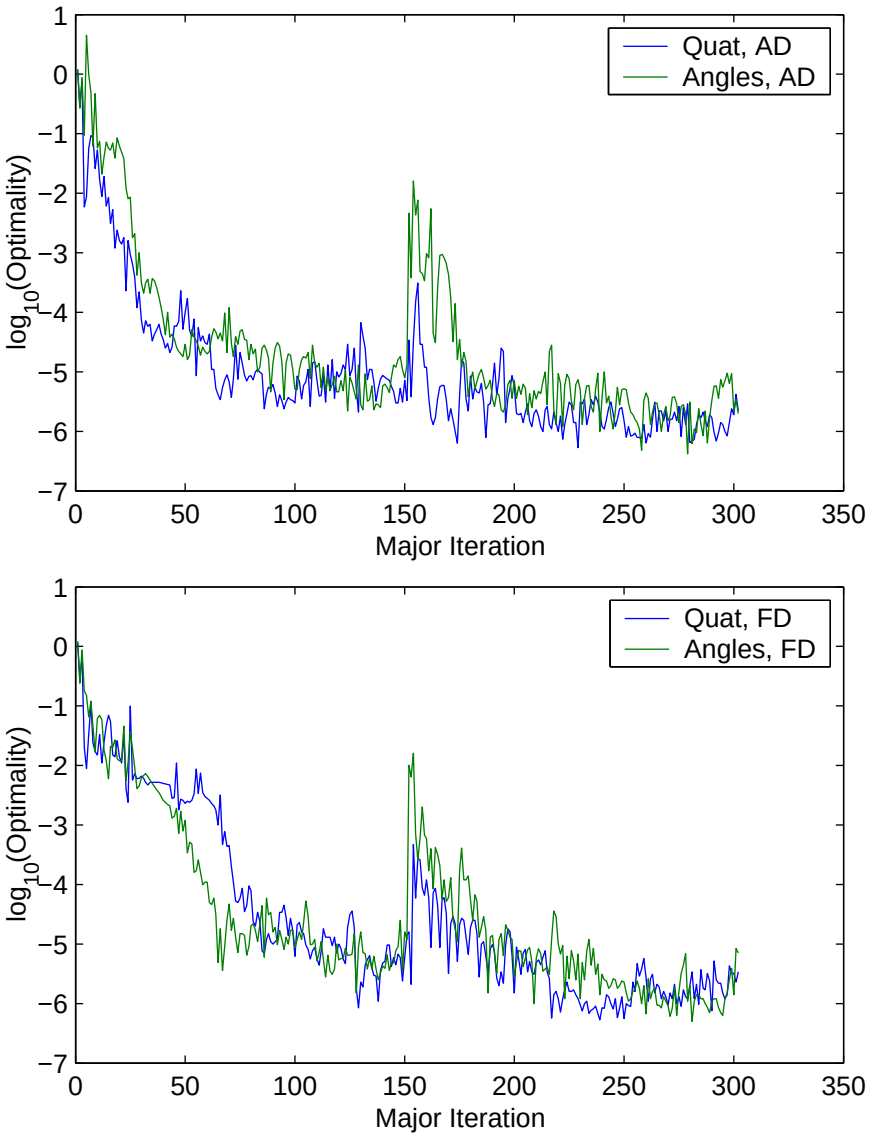

Figure 5.19: Control Variable Impact on Optimality Convergence

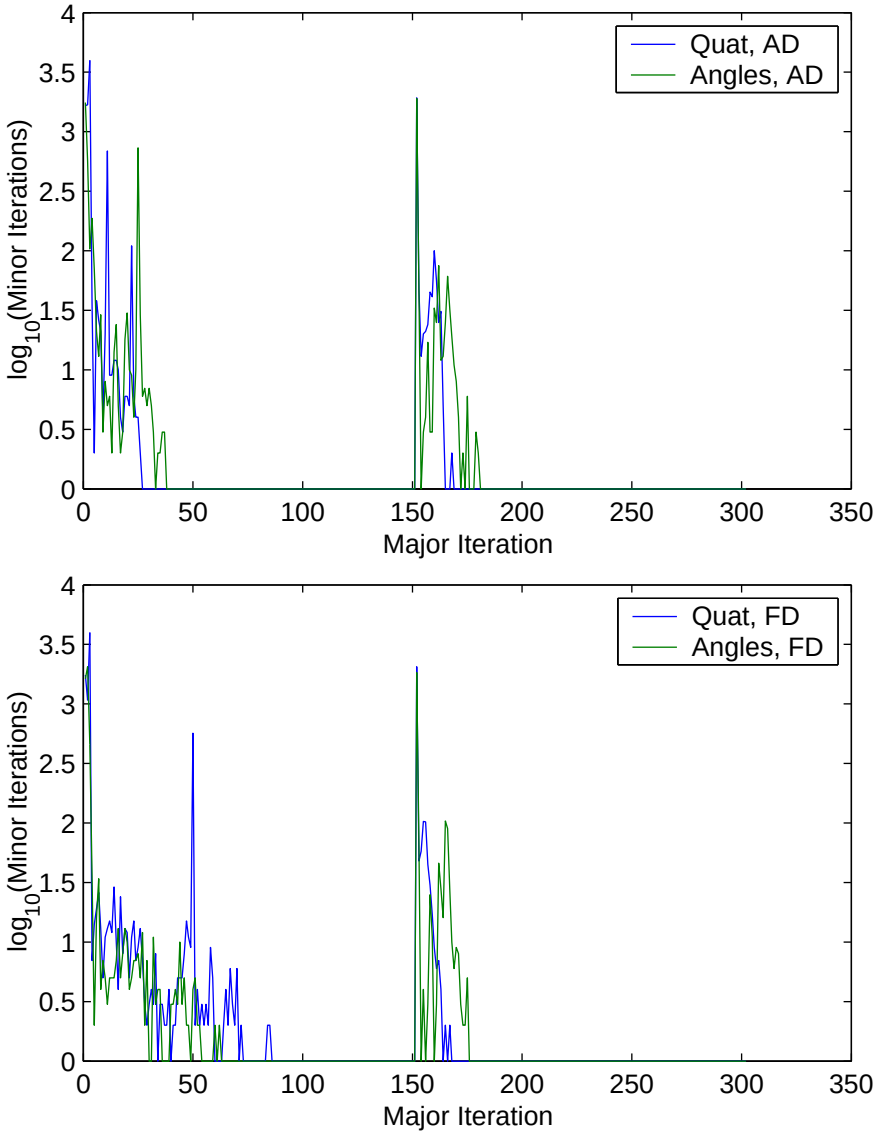

Figure 5.20: Control Variable Impact on Minor Iterations

of the higher dimension of the problem compared to the lunar launch problem the constraint Jacobian is slightly less sparse for the SSTO problem.

Table 5.4 presents timing results for the SSTO problem using automatic differentiation. In spite of exploiting sparsity the program is still fairly slow, taking over half an hour to solve the problems. A significant portion of this time was spent in the first several major iterations on each mesh, where SNOPT was establishing a good basis to work from. Generating a new basis estimate when the mesh is refined could improve the run time somewhat, although the code is still fairly slow. The constraint functions are a relatively small portion of the overall runtime, as was the case with the larger meshes for the lunar launch problem.

Table 5.3: NLP Problem Sizes

Control Variable	Mesh	NLP Dimensions		Jacobian Nonzeros	
		Variables	Constraints	Number	%
Euler Angles	1	1887	1960	54740	1.5
Euler Angles	2	3427	3640	100835	.8
Quaternions	1	2058	2134	61798	1.4
Quaternions	2	3738	3954	113738	.8

Table 5.4: Solution Times

Control Variable	Mesh	CPU Time (s)			% of Total Time		Constraint Time (s)
		SNOPT	Constraints	Total	SNOPT	Constraints	
Euler Angles	1	363	84	448	81	19	.558
Euler Angles	2	1291	158	1451	89	11	1.056
Quaternions	1	481	85	567	85	15	.567
Quaternions	2	1461	158	1621	90	10	1.052

Chapter 6

Summary, Conclusions, and Recommendations

6.1 Summary and Conclusions

The use of automatic differentiation to provide machine-precision derivatives for use in general-purpose trajectory optimization codes was demonstrated. The derivatives were coupled with a direct multiple shooting approach which provided improved convergence behavior over single-shooting approaches and more accurate adjoints than fourth-order approaches based on Hermite collocation. Trajectories were computed for two different example problems, verifying the accuracy of the code, testing the impact of the improved derivatives on the solution of a complex problem, and demonstrating the flexibility of automatic differentiation.

For the lunar launch test problem, the accuracy of the multiple shooting scheme was verified and derivative computation was found to have some effect on convergence of the problem. While noticeable, the effect did not appear until the problem had converged far beyond the level of the accuracy which would typically be expected in ordinary use. For ordinary use there was no significant difference between the solutions generated using finite differences and automatic differentiation. In retrospect this is not surprising since the

dynamics of the example problem are not very complicated.

With the SSTO ascent problem both derivative accuracy and control variable choices were tested. No significant difference was observed in the final level of convergence when either the type of derivative computation or the type of control variables were changed, although for lower accuracy levels (10^{-4}), automatic differentiation provided faster convergence then finite differencing. Neither form of derivative computation allowed the solution to converge as tightly as the lunar launch problem. The convergence behavior was essentially the same, regardless of the form of derivative computation. Neither automatic differentiation nor finite differencing held a clear lead in the number of iterations required for convergence, and neither type of derivative computation consistently allowed tighter convergence than the other.

The principle impact control variable choice had on computation was the length of the startup period, where the NLP code establishes the set of variables which will be adjusted to find an optimum. Using quaternion elements as controls allowed the NLP solver to establish a basis faster than when Euler angles were controls.

For the SSTO problem there were 61 scalar quantities, 30 vector quantities, and 14 coordinate transformations which were computed at each node. Each coordinate transformation contained a quaternion, direction cosine matrix, and Euler angle set for the transformation, so over 370 quantities were computed at each node. While not all were available as states or controls, most were available for use in path and point constraints. The code was writ-

ten without any special derivative computation functions, aside from the routines for the automatic-differentiation data structures themselves. The function evaluation source code required was the same as that which was required for finite difference derivative computations, although the style was somewhat different because of language limitations. Switching control variables was accomplished simply by changing the appropriate guesses, bounds, and lines declaring what the control variables were in each phase. Although not demonstrated here, the capability exists to change the coordinate transformation used to specify vehicle attitude. Again, the only changes which must be made are a few lines in the problem specification. This demonstrates automatic differentiation can match the versatility of finite differencing in general-purpose trajectory optimization software such as POST, OTIS, and GTS.

Some limitations on the use of automatic differentiation for this work do exist. One, due to the choice of Java as the source language for this project, is the requirement to break a one-line calculation into its sequence of one- and two-function operations. If the code had been written in C++, or Fortran 90 or 95, operator overloading could have been used to avoid this. Alternatively, a preprocessor of some sort could have been used to generate the code lists automatically from the original expression.

A more serious issue which is independent of the type of computer language is the need for access to the function source code. If the objective or constraint function involves calls to precompiled code for which the source code is not available, then automatic differentiation cannot be used.

Also, while automatic differentiation can provide machine-precision derivatives, finite-differencing can still be used to check the values of the derivative and ensure the derivative code is work as intended. In fact, SNOPT includes an option to do precisely this with the user-supplied derivatives.

Given the results of this work and the issues mentioned above, the use of automatic differentiation can improve results obtained with a trajectory optimization code, but not to the degree to warrant a major retrofit effort to provide accurate gradients in an existing code. On the other hand, if a new program is being developed, or an existing one is being rewritten anyway, automatic differentiation could provide an easy incremental improvement provided the code is written in a language which provides operator overloading, or which is supported by an automatic differentiation preprocessor.

6.2 Recommendations for Future Work

There are several avenues for future work. The example problems chosen here are perhaps not the best problems, one being too simple and the other too complex for conclusively highlighting the benefits of exact derivatives in trajectory optimization codes. Tests on a broader range of trajectory problems could better highlight the impact of accurate derivative computation.

Single-shooting methods, or codes which use a combination of multiple shooting and marching solutions will likely have different sensitivities to the derivative computation. Tests using a variety of shooting approaches might

indicate trends such as greater derivative sensitivity when the code includes a significant amount of explicit forward integration of the states. Perhaps a single-shooting code such as POST or GTS is more sensitive to derivative accuracy than a parallel shooting methods such as the one investigated by Enright[19] and Betts[9].

The multiple shooting method used here is inappropriate for stiff problems because of the explicit integration rule used. One such stiff problem is computing optimal six-degree-of-freedom vehicle trajectories. In these problems both translational and rotational dynamics are modeled, and they would likely also have different sensitivity to the accuracy of the derivative computation.

Higher-order integration schemes, such as the ones investigated by Herman[38], would allow more accurate solutions without as many grid points as the fourth-order method used here. Higher integration accuracy could be obtained before reaching memory limits. On the other hand, gains in state integration accuracy might come at the expense of accuracy in the adjoints and controls.

Variable elimination would involve removing the intermediate stage values for the states from the NLP problem, along with the corresponding intermediate stage defect constraints. This would significantly reduce the size of the NLP problem, and could allow larger problems to be solved given a certain amount of computer memory.

Appendices

Appendix A

Coordinate Systems

A.1 ECI Frame

The ECI frame is an inertial frame with the origin fixed at the center of the Earth, the x-axis pointing to the first point of Aries, and the z-axis pointing through the North geographic pole. The y-axis completes a right-hand coordinate system.

A.2 ECEF Frame

The ECEF frame is a rotating frame with the origin fixed at the center of the Earth, the x-axis in the equatorial plane pointing out through the Prime Meridian, and the z-axis through the North geographic pole. As with the ECI system, the y-axis completes the right-hand coordinate system. The ECEF frame rotates with the Earth, and the transformation of a vector $\vec{a}$ from the

ECEF system to the ECI system is given by

$$\vec{a}_R = T_{RI}\vec{a}_I \tag{A.1}$$

$$T_{RI} = \begin{pmatrix} \cos(\Omega t) & \sin(\Omega t) & 0 \\ -\sin(\Omega t) & \cos(\Omega t) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{A.2}$$

where Ω is the rotation rate of the Earth, and t is time. Note this is simply to transform coordinates from one system to another, not to relate motion relative to one system in terms of motion relative to another. The velocity relative to the ECEF frame is related to velocity in the ECI frame by

$$\vec{v}_R^R = T_{RI}\vec{v}_I + \dot{T}_{RI}\vec{r}_I \tag{A.3}$$

where $\vec{r}_I$ is the radius vector in the ECI frame and $\vec{v}_I$ is velocity in the inertial coordinate system in terms of ECI coordinates. Acceleration relative to the ECEF frame is given by

$$\vec{a}_R^R = T_{RI}\vec{a}_I + 2\dot{T}_{RI}\vec{v}_I + \ddot{T}_{RI}\vec{r}_I \tag{A.4}$$

where $\vec{a}_I$ is acceleration in the ECI frame. The superscript R indicates the velocity is relative to the rotating system while the subscript R indicates the components are expressed in terms of the ECEF coordinates.

A.3 Launch Frame

The Launch frame is a right-hand, inertial coordinate system with the origin fixed to the launch site position at the time of launch. The x-axis points in the direction of the local vertical while the z-axis points in the direction of a specified azimuth. With an azimuth of zero degrees the z-axis points North, while an azimuth of 90 degrees points the z-axis East. The y-axis completes the right-hand system. Transforming a vector from the Launch system to the ECI system is done with

$$\vec{a}_I = T_{RI}^{-1} T(\phi_0)\, T(\lambda_0)\, T(\alpha_L)\, \vec{a}_L \tag{A.5}$$

$$T(\phi_0) = \begin{pmatrix} \cos(\phi_0) & -\sin(\phi_0) & 0 \\ \sin(\phi_0) & \cos(\phi_0) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{A.6}$$

$$T(\lambda_0) = \begin{pmatrix} \cos(\lambda_0) & 0 & -\sin(\lambda_0) \\ 0 & 1 & 0 \\ \sin(\lambda_0) & 0 & \cos(\lambda_0) \end{pmatrix} \tag{A.7}$$

$$T(\alpha_L) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha_L) & \sin(\alpha_L) \\ 0 & -\sin(\alpha_L) & \cos(\alpha_L) \end{pmatrix} \tag{A.8}$$

where ϕ_0 is the longitude of the launch site, λ_0 is the latitude of the launch site, and α_L is the azimuth angle of the z-axis in the Launch frame.

A.4 Geographic Frame

The Geographic frame is a moving frame with the origin fixed to the point on the surface directly below the vehicle. The x-axis points North, the y-axis points East, and the z-axis points in the direction opposite the radius vector. The transformation from the Geographic to ECEF frame is

$$\vec{a}_R = T(\phi)\, T(\lambda)\, S \vec{a}_G \tag{A.9}$$

$$T(\phi) = \begin{pmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{A.10}$$

$$T(\lambda) = \begin{pmatrix} \cos(\lambda) & 0 & -\sin(\lambda) \\ 0 & 1 & 0 \\ \sin(\lambda) & 0 & \cos(\lambda) \end{pmatrix} S \qquad = \begin{pmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \tag{A.11}$$

where ϕ and λ are the longitude and latitude of the sub-vehicle point, respectively.

A.5 Body Frame

The Body frame is a vehicle-fixed frame with the coordinate system origin fixed to the vehicle's center of mass. The x-axis points forward out the nose of the vehicle, the y-axis points in the direction of the right wing for a pilot seated in the vehicle, and the z-axis points through the bottom of the vehicle, completing the right-hand system. The body system is related to the Launch frame through a series of three Euler angles with the transformation

given by

$$\vec{a}_L = T(\Phi_I)\, T(\Psi_I)\, T(\Theta_I)\, \vec{a}_B \tag{A.12}$$

$$T(\Phi_I) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\Phi_I) & \sin(\Phi_I) \\ 0 & -\sin(\Phi_I) & \cos(\Phi_I) \end{pmatrix} \tag{A.13}$$

$$T(\Psi_I) = \begin{pmatrix} \cos(\Psi_I) & -\sin(\Psi_I) & 0 \\ \sin(\Psi_I) & \cos(\Psi_I) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{A.14}$$

$$T(\Theta_I) = \begin{pmatrix} \cos(\Theta_I) & 0 & -\sin(\Theta_I) \\ 0 & 1 & 0 \\ \sin(\Theta_I) & 0 & \cos(\Theta_I) \end{pmatrix} \tag{A.15}$$

where Ψ_I is the inertial yaw angle, Θ_I is the inertial pitch angle, and Φ_I is the inertial roll angle. Figure A.1 depicts the relationship between the Launch and Body axes.

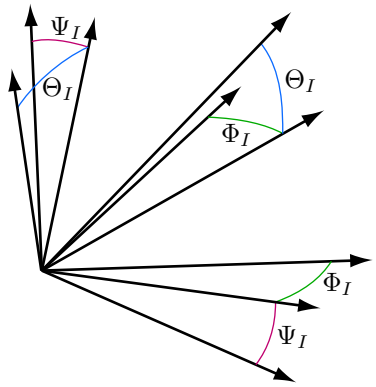

Figure A.1: Geometry of the Launch-Body Coordinate System Transformation

The Body frame has a similar transformation to the Geographic frame

$$\vec{a}_G = T(\Psi)\,T(\Theta)\,T(\Phi)\,\vec{a}_B \tag{A.16}$$

$$T(\Psi) = \begin{pmatrix} \cos(\Psi) & -\sin(\Psi) & 0 \\ \sin(\Psi) & \cos(\Psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{A.17}$$

$$T(\Theta) = \begin{pmatrix} \cos(\Theta) & 0 & -\sin(\Theta) \\ 0 & 1 & 0 \\ \sin(\Theta) & 0 & \cos(\Theta) \end{pmatrix} \tag{A.18}$$

$$T(\Phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\Phi) & \sin(\Phi) \\ 0 & -\sin(\Phi) & \cos(\Phi) \end{pmatrix} \tag{A.19}$$

where Ψ is the yaw angle, Θ is the pitch angle, and Φ is the roll angle. Figure A.2 depicts the relationship between the Geographic and Body axes.

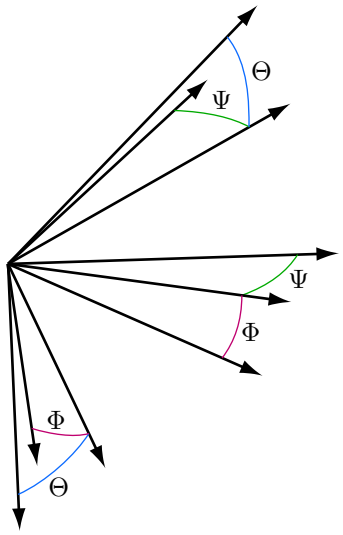

Figure A.2: Geometry of the Geographic-Body Coordinate System Transformation

A.6 Air-Path Frame

Like the Body frame, the Air-Path frame is a vehicle-fixed frame with the coordinate system origin fixed to the vehicle's center of mass. The x-axis points in the direction of the vehicle's atmosphere-relative velocity vector. It is related to the body frame through an angle-of-attack, sideslip, bank rotation

sequence:

$$\vec{a}_B = T(\alpha)\, T(\beta)\, T(\phi)\, \vec{a}_W \tag{A.20}$$

$$T(\alpha) = \begin{pmatrix} \cos(\alpha) & 0 & -\sin(\alpha) \\ 0 & 1 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix} \tag{A.21}$$

$$T(\beta) = \begin{pmatrix} \cos(\beta) & -\sin(\beta) & 0 \\ \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{A.22}$$

$$T(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{pmatrix} \tag{A.23}$$

where α is the angle of attack, β is the sideslip angle, and ϕ is the bank angle. Figure A.3 depicts the relationship between the Air-Path and Body axes.

A.7 Inertial Air-Path Frame

The Inertial Air-Path frame is oriented with respect to the inertial velocity as the Air-Path frame is oriented to the atmosphere-relative velocity. The Inertial Air-Path frame is a vehicle-fixed frame with the coordinate system origin fixed to the vehicle's center of mass. The x-axis points in the direction of the vehicle's inertial velocity vector. It is related to the body frame through

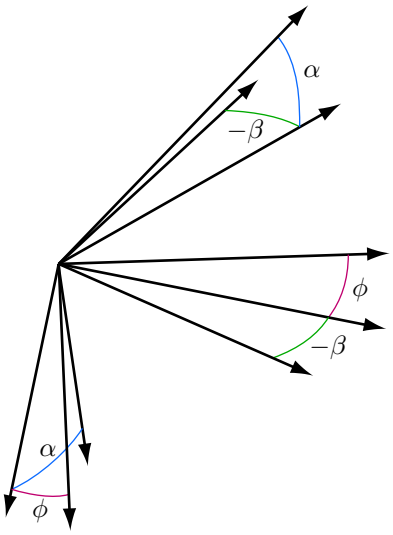

Figure A.3: Geometry of the Geographic-Body Coordinate System Transformation

an angle-of-attack, sideslip, bank rotation sequence:

$$\vec{a}_B = T(\alpha_I)\, T(\beta_I)\, T(\phi_I)\, \vec{a}_W \tag{A.24}$$

$$T(\alpha_I) = \begin{pmatrix} \cos(\alpha_I) & 0 & -\sin(\alpha_I) \\ 0 & 1 & 0 \\ \sin(\alpha_I) & 0 & \cos(\alpha_I) \end{pmatrix} \tag{A.25}$$

$$T(\beta_I) = \begin{pmatrix} \cos(\beta_I) & -\sin(\beta_I) & 0 \\ \sin(\beta_I) & \cos(\beta_I) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{A.26}$$

$$T(\phi_I) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi_I) & \sin(\phi_I) \\ 0 & -\sin(\phi_I) & \cos(\phi_I) \end{pmatrix} \tag{A.27}$$

where α_I is the angle of attack, β_I is the sideslip angle, and ϕ_I is the bank angle.

Bibliography

[1] U.S. standard atmosphere, 1962. Technical report, Committee on Extension to the Standard Atmosphere(COSEA), Washington, D.C., December 1962.

[2] Uri Ascher, J. Christiansen, and R. Russell. Collocation software for boundary value ODE's. *ACM Transactions on Mathematical Software*, 7:209–222, 1981.

[3] Uri Ascher, J. Christiansen, and R. D. Russell. COLSYS–A collocation code for boundary-value problems. In B. Childs, M. Scott, J. W. Daniel, E. Denman, and P. Nelson, editors, *Codes for Boundary Value Problems in Ordinary Differential Equations*, volume 76 of *Lecture Notes in Computer Science*, pages 164–185, Berlin, May 1979. Springer–Verlag.

[4] Uri M. Ascher, Robert M. M. Mattheij, and Robert D. Russell. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*, volume 13 of *Classics in Applied Mathematics*. SIAM, Philadelphia, 1995.

[5] G. Bader and Uri Ascher. A new basis implementation for a mixed order boundary value ODE solver. *SIAM Journal of Scientific and Statistical Computing*, 8:483–500, 1987.

[6] John T. Betts. Mesh refinement in direct transcription methods for optimal control. *Optimal Control Applications & Methods*, 19:1–21, 1998.

[7] John T. Betts. Survey of numerical methods for trajectory optimization. *Journal of Guidance, Control, and Dynamics*, 21(2):193–207, 1998.

[8] John T. Betts. *Practical Methods for Optimal Control Using Nonlinear Programming*. Advances in Design and Control. SIAM, Philadelphia, 2001.

[9] John T. Betts and William P. Huffman. Trajectory optimization on a parallel processor. *Journal of Guidance, Control, and Dynamics*, 14(2):431–439, 1991.

[10] Arthur E. Bryson Jr. and Yu-Chi Ho. *Applied Optimal Control*. Hemisphere Publishing Company, Bristol, PA, 1975.

[11] Michael D. Canon, Clifton D. Cullum Jr., and Elijah Polak. *Theory of Optimal Control and Mathematical Programming*. McGraw-Hill Series in Systems Science. McGraw-Hill, New York, 1970.

[12] Alan Carle, Lawrence L. Green, Christian H. Bischof, and Perry A. Newman. Applications of automatic differentiation in CFD. In *Proceedings of the 25th AIAA Fluid Dynamics Conference*, number AIAA 94-2197. AIAA, 1994.

[13] Thomas F. Coleman and Jorge J. Moré. Estimation of sparse jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20(1):187–209, 1983.

[14] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse jacobian matrices. *Journal of the Institute of Mathematics and its Applications*, 13:117–119, 1974.

[15] Ernst D. Dickmanns and Klaus H. Well. Approximate solution of optimal control problems using third-order hermite polynomial functions. In *Proceedings of the 6th Technical Conference on Optimization Techniques*, volume IFIP-TC7, New York, 1975. Springer-Verlag.

[16] Peter Eberhard and Christian Bischof. Automatic differentiation of numerical integration algorithms. *Mathematics of Computation*, 68(226):717–731, 1999.

[17] Gamal Elnagar, Mohammad A. Kazemi, and Mohsen Razzaghi. The pseudospectral legendre method for discretizing optimal control problems. *IEEE Transactions on Automatic Control*, 40(10):1793–1796, 1995.

[18] Gamal Elnagar and Mohsen Razzaghi. A collocation-type method for linear quadratic optimal control problems. *Optimal Control Applications and Methods*, 18:227–235, 1997.

[19] Paul Enright. *Optimal Finite-Thrust Spacecraft Trajectories Using Direct Transcription and Nonlinear Programming.* PhD thesis, University of

Illinois at Urbana-Champaign, 1991.

[20] Fariba Fahroo and I. Michael Ross. A spectral patching method for direct trajectory optimization. *The Journal of the Astronautical Sciences*, 48(2 & 3):269–286, 2000.

[21] Fariba Fahroo and I. Michael Ross. Costate estimation by a Legendre pseudospectral method. *Journal of Guidance, Control, and Dynamics*, 24(2):270–277, 2001.

[22] F. E. Fritzen. GTS: An optimization approach to trajectory performance evaluation via simulation. AIAA 86-1221. Space Systems Technology Conference, San Diego, CA, June 9-12, 1986.

[23] Philip E. Gill, Walter Murray, and Michael A. Saunders. User's guide for SNOPT 5.3: A Fortran package for large-scale nonlinear programming. Technical Report SOL Report 98-1, Stanford Systems Optimization Laboratory, Stanford, December 1998.

[24] Philip E. Gill, Walter Murray, and Michael A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Journal on Optimization*, 12(4):979–1006, 2002.

[25] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press, San Diego, 1981.

[26] D. Gottlieb, M. Y. Hussaini, and S. A. Orszag. Theory and applications of spectral methods. In R. Vought, D. Gottlieb, and M. Y. Hussaini, editors, *Spectral Methods for PDE's*, Philadelphia, 1984. SIAM.

[27] Robert G. Gottlieb. Rapid convergence to optimum solutions using a min-H strategy. *AIAA Journal*, 5(2):322–329, 1967.

[28] Robert Golden Gottlieb. *Optimal Three-Dimesional Boost Trajectories for Shuttle-Type Vehicles*. PhD thesis, University of Texas at Austin, December 1972.

[29] Lawrence L. Green, Perry A. Newman, and Kara J. Haigler. Sensitivity derivatives for advanced CFD algorithm and viscous modeling parameters via automatic differentiation. *Journal of Computational Physics*, 125:313–324, 1996.

[30] Andreas Griewank. On Automatic Differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

[31] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, volume 19 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, 2000.

[32] Andreas Griewank, Christian Bischof, George Corliss, Alan Carle, and Karen Williamson. Derivative convergence for iterative equation solvers. *Optimization Methods and Software*, 2:321–355, 1993.

[33] William W. Hager. Rates of convergence for discrete approximations to unconstrained control problems. *SIAM Journal on Numerical Analysis*, 13(4):449–472, 1976.

[34] William W. Hager. Runge-Kutta methods in optimal control and the transformed adjoint system. *Numerische Mathematik*, 87:247–282, 2000.

[35] Ernst Hairer, Gerhard Wanner, and Syvert Paul Norsett. *Solving Ordinary Differential Equations I: Nonstiff Problems*, volume 8 of *Springer Series in Comptational Mathematics*. Springer, Berlin, second revised edition edition, 2000.

[36] C. R. Hargraves and S. W. Paris. Direct trajetory optimization using nonlinear programming and collocation. *Journal of Guidance, Control, and Dynamics*, 10(4):338–342, 1987.

[37] C. R. Hargraves, S. W. Paris, and W. G. Vlases. OTIS past, present, and future. AIAA 92-4530. AIAA Guidance Navigation and Control Conference, Hilton Head Island, SC, Aug 10-12, 1992.

[38] Albert L. Herman and Bruce A. Conway. Direct optimization using collocation based on high-order Gauss-Lobatto quadrature rules. *Journal of Guidance, Control, and Dynamics*, 19(3):592–599, 1996.

[39] David G. Hull. Numerical derivatives for parameter optimization. *Journal of Guidance and Control*, 2(2):158–160, 1979.

[40] D. H. Jacobson. New second-order and first-order algorithms for determining optimal control: A differential dynamic programming approach. *Journal of Optimization Theory and Applications*, 2(6):411–440, 1968.

[41] Harriet Kagiwada, Robert Kalaba, Nima Rasakhoo, and Karl Spingarn. *Numerical Derivatives and Nonlinear Analysis*, volume 31 of *Mathematical Concepts and Methods in Science and Engineering*. Plenum Press, New York, 1986.

[42] Gershon Kedem. Automatic differentiation of computer programs. *ACM Transactions on Mathematical Software*, 6(2):150–165, 1980.

[43] Dieter Kraft. On converting optimal control problems into nonlinear programming problems. In Klaus Schittkowski, editor, *Computational Mathematical Programming*, volume F15 of *NATO ASI series*, pages 261–280, Berlin, 1985. Springer–Verlag.

[44] Rainer Mehlhorn and Gottfried Sachs. A new tool for efficient optimization by automatic differentiation and program transparency. *Optimization Methods and Software*, 4:225–242, 1994.

[45] A. Miele, R. E. Pritchard, and J. N. Damoulakis. Sequential gradient-restoration algorithm for optimal control problems. *Journal of Optimization Theory and Applications*, 5(4):235–282, 1970.

[46] H. J. Oberle and W. Grimm. BNDSCO: A program for the numerical solution of optimal control problems. Technical Report 515, DFVLR,

Hamburg, October 2001.

[47] G. M. Ostrovskii, Y. M. Volin, and W. W. Borisov. Über die berechnung von ableitungen. *Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg*, 13(4):382–384, 1971.

[48] Elijah Polak. *Computational Methods in Optimization*, volume 77 of *Mathematics in Science and Engineering*. Academic Press, New York, 1971.

[49] R. W. Powell, S. A. Striepe, P. N. Desai, R. D. Braun, G. L. Brauer, D. E. Cornick, D. W. Olson, F. M. Petersen, R. Stevenson, M. C. Engel, and S. M. Marsh. *Program To Optimize Simulated Trajectories(POST II) Volume II: Utilization Manual*, February 1998.

[50] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

[51] Rainer Schöpf and Peter Deuflhard. OCCAL: A mixed symbolic-numeric Optimal Control CALculator. In R. Bulirsch and D. Kraft, editors, *Computational Optimal Control*, volume 115 of *International Series of Numerical Mathematics*, pages 269–278, Basel, Switzerland, 1994. Birkhäuser Verlag.

[52] Adam Lowell Schwartz. *Theory and Implementation of Numerical Methods Based on Runge-Kutta Integration for Solving Optimal Control Prob-*

lems. Ph.d. dissertation, University of California at Berkeley, 1996.

[53] Laura L. Sherman, Arthur C. Taylor III, Lawrence L. Green, Perry A. Newman, Gene W. Hou, and Vamshi Mohan Korivi. First- and second-order aerodynamic sensitivity derivatives via automatic differentiation with incremental iterative methods. *Journal of Computational Physics*, 129:307–331, 1996.

[54] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Fiven by Algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, 1980.

[55] Josef Stoer and Roland Bulirsch. *Introduction to Numerical Analysis*, volume 12 of *Texts in Applied Mathematics*. Springer-Verlag, New York, 2nd edition, 1993.

[56] Paul V. Tartabini, Roger A. Lepsch, J. J. Korte, and Kathryn E. Wurster. A multidisciplinary performance analysis of a lifting-body single-stage-to-orbit vehicle. In *Proceedings of the 38th Aerospace Sciences Meeting & Exhibit*, number AIAA 2000-1045. AIAA, 2000.

[57] R. E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

[58] R. D. Wilkins. Investigation of a new analytical method for numerical derivative evaluation. *Communications of the ACM*, 7(8):465–471, 1964.

[59] Philip Wolfe. Checking the calculation of gradients. *ACM Transactions on Mathematical Software*, 8(4):337–343, 1982.

[60] J. A. Zonneveld. *Automatic Numerical Integraton*, volume 8 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 2nd edition, 1970.

Vita

David Marcelo Garza was born in McAllen, Texas on April 6, 1971, the son of Daniel Marcelo Garza and Mary Florence Boyce Garza. After graduating from Westwood High School, Austin, Texas, in 1989, he entered The University of Texas at Austin in Austin, Texas. He received his Bachelor of Science in Aerospace Engineering from The University of Texas at Austin in December 1993. In 1994, he entered The University of Tennessee Space Institute in Tullahoma, Tennessee, and graduated with his Master of Science in August 1996. The same month, he entered the Graduate School of The University of Texas at Austin.

Permanent address: 10019 Woodland Village Dr
Austin, Texas 78750

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.