User's Manual for $\mathcal{GPOPS}$ Version 2.3:

A MATLAB® Software for Solving Multiple-Phase Optimal Control Problems Using the Gauss Pseudospectral Method

Anil V. Rao
University of Florida
Gainesville, FL 32607

David Benson
The Charles Stark Draper Laboratory, Inc.
Cambridge, MA 02139

Christopher L. Darby
Camila Francolin
Michael Patterson
Ilyssa Sanders
University of Florida
Gainesville, FL 32607

Geoffrey T. Huntington
Blue Origin, LLC
Seattle, WA

August 2009

THIS PAGE IS INTENTIONALLY LEFT BLANK

Acknowledgments

The software $\mathcal{GPOPS}$ was developed in response to a demand from the research and academic community for a customizable implementation of the Gauss pseudospectral method for solving optimal control problems. Indeed, while the authors of $\mathcal{GPOPS}$ have published extensively in the open literature on the theory and application of the Gauss pseudospectral method, no effort has been made to date to provide source code of actual software that can be utilized and adapted by a broad community. The $\mathcal{GPOPS}$ software is an attempt to fill that void and enable researchers, educators, and others involved in solving complex optimal control problems to take advantage of a code that can be customized to one's particular needs. The authors of $\mathcal{GPOPS}$ hope sincerely that the code is useful.

Disclaimer

This software is provided “as is” and free-of-charge. Neither the authors nor their employers assume any responsibility for any harm resulting from the use of this software. The authors do, however, hope that users will find this software useful for research and other purposes.

Preface to The $\mathcal{GPOPS}$ Software

It is noted that $\mathcal{GPOPS}$ has been designed to work with the nonlinear programming solver SNOPT (Gill, et. al., 2007). SNOPT can be obtained from one of the following three sources: (1) for government, commercial, or institutional academic use from Stanford Business Software, Inc.; (2) for individual research use by contacting Professor Philip Gill via e-mail at pgill@ucsd.edu; (3) For redistribution by contacting the Office of Technology Licensing at Stanford University. Finally, $\mathcal{GPOPS}$ can be adapted to work with other NLP solvers, but these implementations have not as of yet been developed.

Next, $\mathcal{GPOPS}$ has the option of using either forward-mode automatic differentiation or complex-step differentiation for computing the objective function gradient and the constraint Jacobian. Both complex-step differentiation and forward-mode automatic differentiation are now *built-in options* in $\mathcal{GPOPS}$ and can be invoked using the options as specified later in this manual In addition, it is noted that earlier versions of $\mathcal{GPOPS}$ had the ability to use the following third-party automatic differentiators:

- Interval Laboratory (INTLAB): http://www.ti3.tu-harburg.de/rump/intlab/
- Matlab Automatic Differentiation (MAD): http://matlabad.com

INTLAB is developed by Prof. Siegfried Rump of The University of Hamburg (Germany) while MAD is a commercial product.

Changes in User Interface from *GPOPS* Versions 1.x

The user should be aware of the fact that the interface to *GPOPS* has been revised significantly from *GPOPS* Version 1.x. In particular, *GPOPS* 2.x uses a *structure-based* interface as opposed to an interface based on cell arrays. The new input format has been chosen because it is more intuitive than the previous input format. Any applications that the user has coded using the old input format will need to be changed to the new format. These changes should only take a small effort.

Licensing Agreement

The software *GPOPS* is distributed under the following license agreement. Regardless of how you obtain a copy of *GPOPS* , you must abide by the following terms:

Contents

Chapter 1

Introduction to Gauss Pseudospectral Optimization Software ($\mathcal{GPOPS}$)

Gauss Pseudospectral Optimization Software ($\mathcal{GPOPS}$) is a software program written in MATLAB[1]® for solving multiple-phase optimal control problems of the following form. Given a set of P phases (where $p = 1, \ldots, P$), minimize the cost functional

$$J = \sum_{p=1}^{P} J^{(p)} = \sum_{p=1}^{P} \left[\Phi^{(p)}(\mathbf{x}^{(p)}(t_0), t_0, \mathbf{x}^{(p)}(t_f), t_f; \mathbf{q}^{(p)}) + \mathcal{L}^{(p)}(\mathbf{x}^{(p)}(t), \mathbf{u}^{(p)}(t), t; \mathbf{q}^{(p)}) dt \right] \tag{1–1}$$

subject to the dynamic constraint

$$\dot{\mathbf{x}}^{(p)} = \mathbf{f}^{(p)}(\mathbf{x}^{(p)}, \mathbf{u}^{(p)}, t; \mathbf{q}^{(p)}), \qquad (p = 1, \ldots, P), \tag{1–2}$$

the boundary conditions

$$\phi_{\min} \leq \phi^{(p)}(\mathbf{x}^{(p)}(t_0), t_0^{(p)}, \mathbf{x}^{(p)}(t_f), t_f^{(p)}; \mathbf{q}^{(p)}) \leq \phi_{\max}, \qquad (p = 1, \ldots, P), \tag{1–3}$$

the inequality path constraints

$$\mathbf{C}^{(p)}(\mathbf{x}^{(p)}(t), \mathbf{u}^{(p)}(t), t; \mathbf{q}^{(p)}) \leq \mathbf{0}, \qquad (p = 1, \ldots, P), \tag{1–4}$$

and the phase continuity (linkage) constraints

$$\mathbf{P}^{(s)}(\mathbf{x}^{(p_l^s)}(t_f), t_f^{(p_l^s)}; \mathbf{q}^{(p_l^s)}, \mathbf{x}^{(p_u^s)}(t_0), t_0^{(p_u^s)}; \mathbf{q}^{(p_u^s)}) = \mathbf{0}, \qquad (p_l, p_u \in [1, \ldots, P], s = 1, \ldots, L) \tag{1–5}$$

where $\mathbf{x}^{(p)}(t) \in \mathbb{R}^{n_p}$, $\mathbf{u}^{(p)}(t) \in \mathbb{R}^{m_p}$, $\mathbf{q}^{(p)} \in \mathbb{R}^{q_p}$, and $t \in \mathbb{R}$ are, respectively, the state, control, static parameters, and time in phase $p \in [1, \ldots, P]$, L is the number of phases to be linked, $p_l^s \in [1, \ldots, P]$, $(s = 1, \ldots, L)$ are the "left" phase numbers, and $p_u^s \in [1, \ldots, P]$, $(s = 1, \ldots, L)$ are the "right" phase numbers.

While much of the time a user may want to solve a problem consisting of multiple phases, it is important to note that the phases *need not be sequential*. To the contrary, any two phases may be linked provided that the independent variable does not change direction (i.e., the independent variable moves in the same direction during each phase that is linked). A schematic of how phases can potentially be linked is given in Fig. 1.1.

1.1 Gauss Pseudospectral Method Employed by $\mathcal{GPOPS}$

The method employed by $\mathcal{GPOPS}$ is the *Gauss Pseudospectral Method* (GPM). The GPM is an orthogonal collocation method where the collocation points are the *Legendre-Gauss* points. The theory of the GPM can

[1]MATLAB is a registered trademark of The Mathworks, Inc., One Apple Hill, Natick, MA

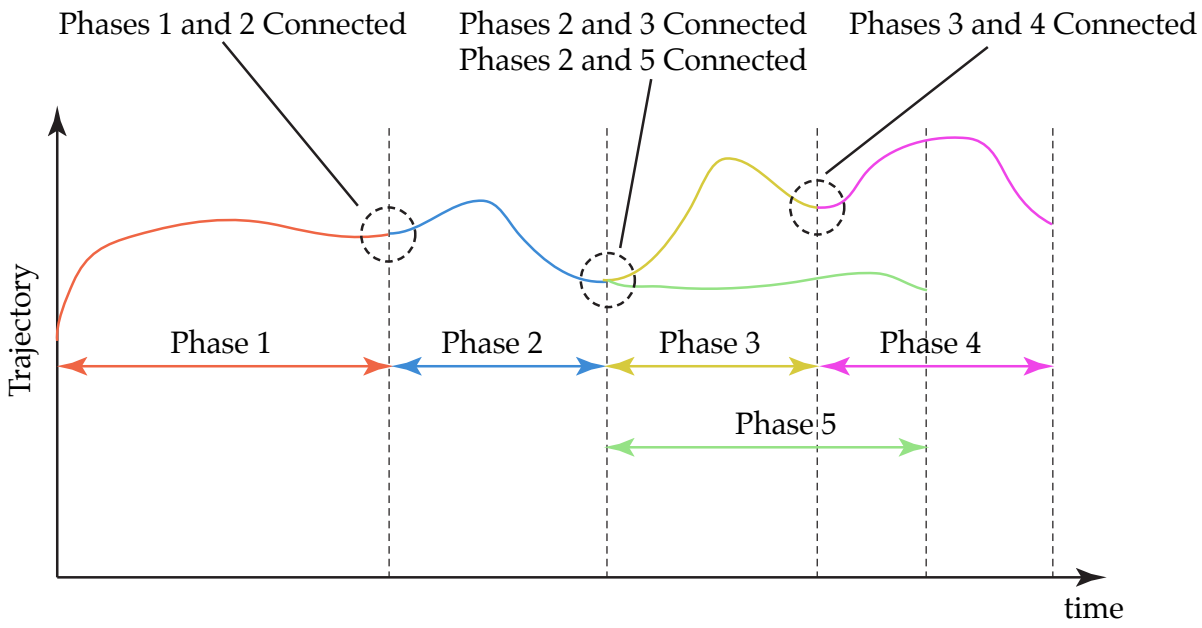

Figure 1.1 Schematic of linkages for multiple-phase optimal control problem. The example shown in the picture consists of five phases where the ends of phases 1, 2, and 3 are linked to the starts of phases 2, 3, and 4, respectively, while the end of phase 3 is linked to the start of phase 5.

be found in (Benson, 2004; Benson, et al., 2006; Huntington, 2007) while applications of the GPM can be found in (Huntington, et al., 2005a; Huntington, et al., 2005b; Huntington, et al., 2005c; Huntington and Rao, 2007a). Strictly speaking, no knowledge of the GPM is required for using $\mathcal{GPOPS}$. However, for completeness, in this section we describe the mathematics of the GPM. It is noted that this section is taken largely from Benson, et al. (2006).

1.1.1 Continuous Bolza Problem

Without loss of generality, consider the following optimal control problem. Determine the state, $\mathbf{x}(\tau) \in \mathbb{R}^n$, control, $\mathbf{u}(\tau) \in \mathbb{R}^m$, initial time, t_0, and final time, t_f, that minimize the cost functional

$$J = \Phi(\mathbf{x}(-1), t_0, \mathbf{x}(1), t_f) + \frac{t_f - t_0}{2} \int_{-1}^{1} g(\mathbf{x}(\tau), \mathbf{u}(\tau), \tau; t_0, t_f) d\tau \tag{1–6}$$

subject to the constraints

$$\frac{d\mathbf{x}}{d\tau} = \frac{t_f - t_0}{2} \mathbf{f}(\mathbf{x}(\tau), \mathbf{u}(\tau), \tau; t_0, t_f) \tag{1–7}$$

$$\boldsymbol{\phi}(\mathbf{x}(-1), t_0, \mathbf{x}(1), t_f) = \mathbf{0} \tag{1–8}$$

$$\mathbf{C}(\mathbf{x}(\tau), \mathbf{u}(\tau), \tau; t_0, t_f) \leq \mathbf{0} \tag{1–9}$$

The optimal control problem of Eqs. (1–6)–(1–9) will be referred to as the *continuous Bolza problem*. It is noted that the optimal control problem of Eqs. (1–6)–(1–9) can be transformed from the time interval $\tau \in [-1, 1]$ to the time interval $t \in [t_0, t_f]$ via the affine transformation

$$t = \frac{t_f - t_0}{2} \tau + \frac{t_f + t_0}{2} \tag{1–10}$$

1.2 Gauss Pseudospectral Discretization of Continuous Bolza Problem

The direct approach to solving the continuous Bolza optimal control problem of Section 1.1.1 is to discretize and transcribe Eqs. (1–6)–(1–9) to a nonlinear programming problem (NLP). The Gauss pseudospectral method, like Legendre and Chebyshev methods, is based on approximating the state and control trajectories using interpolating polynomials. The state is approximated using a basis of $N+1$ Lagrange interpolating polynomials, L,

$$\mathbf{x}(\tau) \approx \mathbf{X}(\tau) = \sum_{i=0}^{N} \mathbf{X}(\tau_i) L_i(\tau) \tag{1–11}$$

where $L_i(\tau)$ $(i = 0, \ldots, N)$ are defined as

$$L_i(\tau) = \prod_{j=0, j\neq i}^{N} \frac{\tau - \tau_j}{\tau_i - \tau_j} \tag{1–12}$$

Additionally, the control is approximated using a basis of N Lagrange interpolating polynomials $L_i^*(\tau),\ (i = 1, \ldots, N)$ as

$$\mathbf{u}(\tau) \approx \mathbf{U}(\tau) = \sum_{i=1}^{N} \mathbf{U}(\tau_i) L_i^*(\tau) \tag{1–13}$$

where

$$L_i^*(\tau) = \prod_{j=1, j\neq i}^{N} \frac{\tau - \tau_j}{\tau_i - \tau_j} \tag{1–14}$$

It can be seen from Eqs. (1–12) and (1–14) that $L_i(\tau)$ $(i = 0, \ldots, N)$ and $L_i^*(\tau)$ $(i = 1, \ldots, N)$ satisfy the properties

$$L_i(\tau_j) = \begin{cases} 1 & , \quad i = j \\ 0 & , \quad i \neq j \end{cases} \tag{1–15}$$

$$L_i^*(\tau_j) = \begin{cases} 1 & , \quad i = j \\ 0 & , \quad i \neq j \end{cases} \tag{1–16}$$

Differentiating the expression in Eq. (1–11), we obtain

$$\dot{\mathbf{x}}(\tau) \approx \dot{\mathbf{X}}(\tau) = \sum_{i=0}^{N} x(\tau_i) \dot{L}_i(\tau) \tag{1–17}$$

The derivative of each Lagrange polynomial at the LG points can be represented in a differential approximation matrix, $D \in \mathbb{R}^{N \times N+1}$. The elements of the differential approximation matrix are determined offline as follows:

$$D_{ki} = \dot{L}_i(\tau_k) = \sum_{l=0}^{N} \frac{\prod_{j=0, j\neq i, l}^{N} (\tau_k - \tau_j)}{\prod_{j=0, j\neq i}^{N} (\tau_i - \tau_j)} \tag{1–18}$$

where $k = 1, \ldots, N$ and $i = 0, \ldots, N$. The dynamic constraint is transcribed into algebraic constraints via the differential approximation matrix as follows:

$$\sum_{i=0}^{N} D_{ki} \mathbf{X}_i - \frac{t_f - t_0}{2} \mathbf{f}(\mathbf{X}_k, \mathbf{U}_k, \tau_k; t_0, t_f) = \mathbf{0} \quad (k = 1, \ldots, N) \tag{1–19}$$

where $\mathbf{X}_k \equiv \mathbf{X}(\tau_k) \in \mathbb{R}^n$ and $\mathbf{U}_k \equiv \mathbf{U}(\tau_k) \in \mathbb{R}^m$ $(k = 1, \ldots, N)$. Note that the dynamic constraint is collocated only at the LG points and *not* at the boundary points (this form of collocation differs from other well known pseudospectral methods such as those found in (Elnagar, et al., 1995) and (Elnagar and Kazemi, 1998)). Additional variables in the discretization are defined as follows: $\mathbf{X}_0 \equiv \mathbf{X}(-1)$, and $\mathbf{X}_f$, where $\mathbf{X}_f$ is defined in terms of $\mathbf{X}_k$, $(k = 0, \ldots, N)$ and $\mathbf{U}(\tau_k)$ $(k = 1, \ldots, N)$ via the Gauss quadrature (Davis, 1975)

$$\mathbf{X}_f \equiv \mathbf{X}_0 + \frac{t_f - t_0}{2} \sum_{k=1}^{N} w_k \mathbf{f}(\mathbf{X}_k, \mathbf{U}_k, \tau_k; t_0, t_f) \tag{1–20}$$

The continuous cost function of Eq. (1–6) is approximated using a Gauss quadrature(Davis, 1975) as

$$J = \Phi(\mathbf{X}_0, t_0, \mathbf{X}_f, t_f) + \frac{t_f - t_0}{2} \sum_{k=1}^{N} w_k g(\mathbf{X}_k, \mathbf{U}_k, \tau_k; t_0, t_f) \tag{1–21}$$

where w_k are the Gauss weights. Next, the boundary constraint of Eq. (1–8) is expressed as

$$\boldsymbol{\phi}(\mathbf{X}_0, t_0, \mathbf{X}_f, t_f) = \mathbf{0} \tag{1–22}$$

Furthermore, the path constraint of Eq. (1–9) is evaluated at the LG points as

$$\mathbf{C}(\mathbf{X}_k, \mathbf{U}_k, \tau_k; t_0, t_f) \leq \mathbf{0} \quad (k = 1, \ldots, N) \tag{1–23}$$

The cost function of Eq. (1–21) and the algebraic constraints of Eqs. (1–19), (1–20), (1–22), and (1–23) define an NLP whose solution is an approximate solution to the continuous Bolza problem. Finally, it is noted that the above discretization can be employed in multiple-phase problems by transcribing the problem in each phase using the above discretization and connecting the phases by *linkage* constraints (as described above).

1.3 KKT Conditions of the NLP

The first-order optimality conditions (i.e., the Karush-Kuhn-Tucker (KKT) conditions) of the NLP can be obtained using the augmented cost function or Lagrangian. The augmented cost function is formed using the Lagrange multipliers $\tilde{\boldsymbol{\Lambda}}_k \in \mathbb{R}^n$, $\tilde{\boldsymbol{\mu}}_k \in \mathbb{R}^c$, $k = 1, \ldots, N$, $\tilde{\boldsymbol{\Lambda}}_{\boldsymbol{F}} \in \mathbb{R}^n$, and $\tilde{\boldsymbol{\nu}} \in \mathbb{R}^q$ as

$$\begin{aligned} J_a = {} & \Phi(\mathbf{X}_0, t_0, \mathbf{X}_f, t_f) + \frac{t_f - t_0}{2} \sum_{k=1}^{N} w_k g(\mathbf{X}_k, \mathbf{U}_k, \tau_k; t_0, t_f) - \tilde{\boldsymbol{\nu}}^T \boldsymbol{\phi}(\mathbf{X}_0, t_0, \mathbf{X}_f, t_f) \\ & - \sum_{k=1}^{N} \tilde{\boldsymbol{\mu}}_k^T \mathbf{C}(\mathbf{X}_k, \mathbf{U}_k, \tau_k; t_0, t_f) - \sum_{k=1}^{N} \tilde{\boldsymbol{\Lambda}}_k^T \left(\sum_{i=0}^{N} D_{ki} \mathbf{X}_i - \frac{t_f - t_0}{2} \mathbf{f}(\mathbf{X}_k, \mathbf{U}_k, \tau_k; t_0, t_f) \right) \\ & - \tilde{\boldsymbol{\Lambda}}_{\boldsymbol{F}}{}^T \left(\mathbf{X}_f - \mathbf{X}_0 - \frac{t_f - t_0}{2} \sum_{k=1}^{N} w_k \mathbf{f}(\mathbf{X}_k, \mathbf{U}_k, \tau_k; t_0, t_f) \right) \end{aligned} \tag{1–24}$$

The KKT conditions are found by setting equal to zero the derivatives of the Lagrangian with respect to $\mathbf{X}_0$, $\mathbf{X}_k$, $\mathbf{X}_f$, $\mathbf{U}_k$, $\tilde{\boldsymbol{\Lambda}}_k$, $\tilde{\boldsymbol{\mu}}_k$, $\tilde{\boldsymbol{\Lambda}}_{\boldsymbol{F}}$, $\tilde{\boldsymbol{\nu}}$, t_0, and t_f. The solution to the NLP of Section (1.2) must satisfy the following KKT conditions:

$$\sum_{i=0}^{N} \mathbf{X}_i D_{ki} = \frac{t_f - t_0}{2} \mathbf{f}_k \tag{1–25}$$

$$\begin{aligned} & \sum_{i=1}^{N} \left(\frac{\tilde{\boldsymbol{\Lambda}}_i^T}{w_i} + \tilde{\boldsymbol{\Lambda}}_F^T \right) D_{ki}^{\dagger} + \tilde{\boldsymbol{\Lambda}}_F^T D_{kN+1}^{\dagger} = \\ & \frac{t_f - t_0}{2} \left(-\frac{\partial g_k}{\partial \mathbf{X}_k} - \left(\frac{\tilde{\boldsymbol{\Lambda}}_k^T}{w_k} + \tilde{\boldsymbol{\Lambda}}_F^T \right) \frac{\partial \mathbf{f}_k}{\partial \mathbf{X}_k} + \frac{2}{t_f - t_0} \frac{\tilde{\boldsymbol{\mu}}_k^T}{w_k} \frac{\partial \mathbf{C}_k}{\partial \mathbf{X}_k} \right) \end{aligned} \tag{1–26}$$

$$\mathbf{0} = \frac{\partial g_k}{\partial \mathbf{U}_k} + \left(\frac{\tilde{\mathbf{\Lambda}}_k^T}{w_k} + \tilde{\mathbf{\Lambda}}_F^T \right) \frac{\partial \mathbf{f}_k}{\partial \mathbf{U}_k} - \frac{2}{t_f - t_0} \frac{\tilde{\boldsymbol{\mu}}_k^T}{w_k} \frac{\partial \mathbf{C}_k}{\partial \mathbf{U}_k} \tag{1–27}$$

$$\boldsymbol{\phi}(\mathbf{X}_0, t_0, \mathbf{X}_f, t_f) = \mathbf{0} \tag{1–28}$$

$$\tilde{\mathbf{\Lambda}}_0^T = -\frac{\partial \Phi}{\partial \mathbf{X}_0} + \tilde{\boldsymbol{\nu}}^T \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{X}_0} \tag{1–29}$$

$$\tilde{\mathbf{\Lambda}}_F^T = \frac{\partial \Phi}{\partial \mathbf{X}_f} - \tilde{\boldsymbol{\nu}}^T \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{X}_f} \tag{1–30}$$

$$-\frac{t_f - t_0}{2} \sum_{k=1}^{N} w_k \frac{\partial \tilde{\mathcal{H}}_k}{\partial t_0} + \frac{1}{2} \sum_{k=1}^{N} w_k \tilde{\mathcal{H}}_k = \frac{\partial \Phi}{\partial t_0} - \tilde{\boldsymbol{\nu}}^T \frac{\partial \boldsymbol{\phi}}{\partial t_0} \tag{1–31}$$

$$\frac{t_f - t_0}{2} \sum_{k=1}^{N} w_k \frac{\partial \tilde{\mathcal{H}}_k}{\partial t_f} + \frac{1}{2} \sum_{k=1}^{N} w_k \tilde{\mathcal{H}}_k = -\frac{\partial \Phi}{\partial t_f} + \tilde{\boldsymbol{\nu}}^T \frac{\partial \boldsymbol{\phi}}{\partial t_f} \tag{1–32}$$

$$\mathbf{C}_k \leq \mathbf{0} \tag{1–33}$$

$$\tilde{\mu}_{jk} = 0, \text{ when } \quad C_{jk} < 0 \tag{1–34}$$

$$\tilde{\mu}_{jk} \leq 0, \text{ when } \quad C_{jk} = 0 \tag{1–35}$$

$$\mathbf{X}_f = \mathbf{X}_0 + \frac{(t_f - t_0)}{2} \sum_{k=1}^{N} w_k \mathbf{f}_k \tag{1–36}$$

$$\tilde{\mathbf{\Lambda}}_F = \tilde{\mathbf{\Lambda}}_0 + \frac{t_f - t_0}{2} \sum_{k=1}^{N} w_k \left(-\frac{\partial g_k}{\partial \mathbf{X}_k} - \left(\frac{\tilde{\mathbf{\Lambda}}_k^T}{w_k} + \tilde{\mathbf{\Lambda}}_F^T \right) \frac{\partial \mathbf{f}_k}{\partial \mathbf{X}_k} + \frac{2}{t_f - t_0} \frac{\tilde{\boldsymbol{\mu}}_k^T}{w_k} \frac{\partial \mathbf{C}_k}{\partial \mathbf{X}_k} \right) \tag{1–37}$$

where the shorthand notation $g_k \equiv g(\mathbf{X}_k, \mathbf{U}_k, \tau_k; t_0, t_f)$, $\mathbf{f}_k \equiv \mathbf{f}(\mathbf{X}_k, \mathbf{U}_k, \tau_k; t_0, t_f)$, $\mathcal{H}_k \equiv \mathcal{H}(\mathbf{X}_k, \mathbf{\Lambda}_k, \boldsymbol{\mu}_k, \mathbf{U}_k, \tau_k; t_0, t_f)$, and $C_{jk} \equiv C_j(\mathbf{X}_k, \mathbf{U}_k, \tau_k; t_0, t_f)$ is used. Note that the augmented Hamiltonian, $\tilde{\mathcal{H}}_k$, is defined as

$$\tilde{\mathcal{H}}_k \equiv g_k + \left(\frac{\tilde{\mathbf{\Lambda}}_k^T}{w_k} + \tilde{\mathbf{\Lambda}}_F^T \right) \mathbf{f}_k - \frac{2}{t_f - t_0} \frac{\tilde{\boldsymbol{\mu}}_k^T}{w_k} \mathbf{C}_k \tag{1–38}$$

and $\tilde{\mathbf{\Lambda}}_0$ is defined as

$$\tilde{\mathbf{\Lambda}}_0^T \equiv -\frac{\partial \Phi}{\partial \mathbf{X}_0} + \tilde{\boldsymbol{\nu}}^T \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{X}_0} \tag{1–39}$$

1.4 First-Order Optimality Conditions of Continuous Bolza Problem

The indirect approach to solving the continuous Bolza problem of Eqs. (1–6)–(1–9) in Section 1.1.1 is to apply the calculus of variations and Pontryagin's Maximum Principle (Pontryagin, et al., 1962) to obtain first-order necessary conditions for optimality (Kirk, 1970). These variational conditions are typically derived using the augmented Hamiltonian, $\mathcal{H}$, defined as

$$\mathcal{H}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}, \mathbf{u}, \tau; t_0, t_f) = g(\mathbf{x}, \mathbf{u}, \tau; t_0, t_f) + \boldsymbol{\lambda}^T(\tau) \mathbf{f}(\mathbf{x}, \mathbf{u}, \tau; t_0, t_f) - \boldsymbol{\mu}^T(\tau) \mathbf{C}(\mathbf{x}, \mathbf{u}, \tau; t_0, t_f) \tag{1–40}$$

where $\boldsymbol{\lambda}(\tau) \in \mathbb{R}^n$ is the costate and $\boldsymbol{\mu}(\tau) \in \mathbb{R}^c$ is the Lagrange multiplier associated with the path constraint. The continuous-time first-order optimality conditions can be shown to be

$$\begin{aligned}
\frac{d\mathbf{x}}{d\tau} &= \frac{t_f - t_0}{2}\mathbf{f}(\mathbf{x}, \mathbf{u}, \tau; t_0, t_f) = \frac{t_f - t_0}{2}\frac{\partial \mathcal{H}}{\partial \boldsymbol{\lambda}} \\
\frac{d\boldsymbol{\lambda}}{d\tau} &= \frac{t_f - t_0}{2}\left(-\frac{\partial g}{\partial \mathbf{x}} - \boldsymbol{\lambda}^T\frac{\partial \mathbf{f}}{\partial \mathbf{x}} + \boldsymbol{\mu}^T\frac{\partial \mathbf{C}}{\partial \mathbf{x}}\right) = -\frac{t_f - t_0}{2}\frac{\partial \mathcal{H}}{\partial \mathbf{x}} \\
\mathbf{0} &= \frac{\partial g}{\partial \mathbf{u}} + \boldsymbol{\lambda}^T\frac{\partial \mathbf{f}}{\partial \mathbf{u}} - \boldsymbol{\mu}^T\frac{\partial \mathbf{C}}{\partial \mathbf{u}} = \frac{\partial \mathcal{H}}{\partial \mathbf{u}} \\
\boldsymbol{\phi}(\mathbf{x}(\tau_0), t_0, \mathbf{x}(\tau_f), t_f) &= \mathbf{0} \\
\boldsymbol{\lambda}(\tau_0) = -\frac{\partial \Phi}{\partial \mathbf{x}(\tau_0)} + \boldsymbol{\nu}^T\frac{\partial \boldsymbol{\phi}}{\partial \mathbf{x}(\tau_0)} \quad &, \quad \boldsymbol{\lambda}(\tau_f) = \frac{\partial \Phi}{\partial \mathbf{x}(\tau_f)} - \boldsymbol{\nu}^T\frac{\partial \boldsymbol{\phi}}{\partial \mathbf{x}(\tau_f)} \\
\mathcal{H}(t_0) = \frac{\partial \Phi}{\partial t_0} - \boldsymbol{\nu}^T\frac{\partial \boldsymbol{\phi}}{\partial t_0} \quad &, \quad \mathcal{H}(t_f) = -\frac{\partial \Phi}{\partial t_f} + \boldsymbol{\nu}^T\frac{\partial \boldsymbol{\phi}}{\partial t_f} \\
\mu_j(\tau) = 0, \text{ when } \quad C_j(\mathbf{x}, \mathbf{u}, \tau; t_0, t_f) < 0, \quad j = 1, \ldots, c \\
\mu_j(\tau) \leq 0, \text{ when } \quad C_j(\mathbf{x}, \mathbf{u}, \tau; t_0, t_f) = 0, \quad j = 1, \ldots, c
\end{aligned} \tag{1–41}$$

where $\boldsymbol{\nu} \in \mathbb{R}^q$ is Lagrange multiplier associated with the boundary condition $\boldsymbol{\phi}$. It can be shown that the augmented Hamiltonian at the initial and final times can be written, respectively, as

$$\mathcal{H}(t_0) = -\frac{t_f - t_0}{2}\int_{-1}^{1}\frac{\partial \mathcal{H}}{\partial t_0}d\tau + \frac{1}{2}\int_{-1}^{1}\mathcal{H}d\tau \tag{1–42}$$

$$\mathcal{H}(t_f) = \frac{t_f - t_0}{2}\int_{-1}^{1}\frac{\partial \mathcal{H}}{\partial t_f}d\tau + \frac{1}{2}\int_{-1}^{1}\mathcal{H}d\tau \tag{1–43}$$

1.5 Gauss Pseudospectral Discretized Necessary Conditions

In order to discretize the variational conditions of Section (1.4) using the Gauss pseudospectral discretization, it is necessary to form an appropriate approximation for the costate. In this method, the costate, $\lambda(\tau)$, is approximated as follows:

$$\lambda(\tau) \approx \Lambda(\tau) = \sum_{i=1}^{N+1}\lambda(\tau_i)L_i^{\dagger}(\tau) \tag{1–44}$$

where $L_i^{\dagger}(\tau)$ $(i = 1, \ldots, N+1)$ are defined as

$$L_i^{\dagger}(\tau) = \prod_{j=1, j\neq i}^{N+1}\frac{\tau - \tau_j}{\tau_i - \tau_j} \tag{1–45}$$

It is emphasized that the costate approximation is *different* from the state approximation. In particular, the basis of $N+1$ Lagrange interpolating polynomials $L_i^{\dagger}(\tau)$ $(i = 1, \ldots, N+1)$ includes the costate at the *final* time (as opposed to the initial time which is used in the state approximation). This (non-intuitive) costate approximation is necessary in order to provide a complete mapping between the KKT conditions and the variational conditions.

Using the costate approximation of Eq. (1–44), The first-order necessary conditions of the continuous Bolza problem in Eq. (1–41) are discretized as follows. First, the state and control are approximated using Eqs. (1–11) and (1–13), respectively. Next, the costate is approximated using the basis of $N+1$ Lagrange interpolating polynomials as defined in Eq. (1–44). The continuous-time first-order optimality conditions of Eq. (1–41) are discretized using the variables $\mathbf{X}_0 \equiv \mathbf{X}(-1)$, $\mathbf{X}_k \equiv \mathbf{X}(\tau_k) \in \mathbb{R}^n$, and $\mathbf{X}_f \equiv \mathbf{X}(1)$ for the state, $\mathbf{U}_k \equiv \mathbf{U}(\tau_k) \in \mathbb{R}^m$ for the control, $\boldsymbol{\Lambda}_0 \equiv \boldsymbol{\Lambda}(-1)$, $\boldsymbol{\Lambda}_k \equiv \boldsymbol{\Lambda}(\tau_k) \in \mathbb{R}^n$, and $\boldsymbol{\Lambda}_f \equiv \boldsymbol{\Lambda}(1)$ for the costate, and $\boldsymbol{\mu}_k \equiv \boldsymbol{\mu}(\tau_k) \in \mathbb{R}^c$, for the Lagrange multiplier associated with the path constraints at the LG points

$k = 1, \ldots, N$. The other unknown variables in the problem are the initial and final times, $t_0 \in \mathbb{R}$, $t_f \in \mathbb{R}$, and the Lagrange multiplier, $\boldsymbol{\nu} \in \mathbb{R}^q$. The total number of variables is then given as $(2n + m + c)N + 4n + q + 2$. These variables are used to discretize the continuous necessary conditions of Eq. (1–41) via the Gauss pseudospectral discretization. Note that the derivative of the state is approximated using Lagrange polynomials based on $N+1$ points consisting of the N LG points and the initial time, τ_0, while the derivative of the costate is approximated using Lagrange polynomials based on $N + 1$ points consisting of the N LG points and the final time, τ_f. The resulting algebraic equations that approximate the continuous necessary conditions at the LG points are given as

$$\sum_{i=0}^{N} \mathbf{X}_i D_{ki} = \frac{t_f - t_0}{2} \mathbf{f}_k \tag{1–46}$$

$$\sum_{i=1}^{N} \boldsymbol{\Lambda}_i D^{\dagger}_{ki} + \boldsymbol{\Lambda}_f D^{\dagger}_{kN+1} = \frac{t_f - t_0}{2} \left(-\frac{\partial g_k}{\partial \mathbf{X}_k} - \boldsymbol{\Lambda}_k^T \frac{\partial \mathbf{f}_k}{\partial \mathbf{X}_k} + \boldsymbol{\mu}_k^T \frac{\partial \mathbf{C}_k}{\partial \mathbf{X}_k} \right) \tag{1–47}$$

$$\mathbf{0} = \frac{\partial g_k}{\partial \mathbf{U}_k} + \boldsymbol{\Lambda}_k^T \frac{\partial \mathbf{f}_k}{\partial \mathbf{U}_k} - \boldsymbol{\mu}_k^T \frac{\partial \mathbf{C}_k}{\partial \mathbf{U}_k} \tag{1–48}$$

$$\boldsymbol{\phi}(\mathbf{X}_0, t_0, \mathbf{X}_f, t_f) = \mathbf{0} \tag{1–49}$$

$$\boldsymbol{\Lambda}_0 = -\frac{\partial \Phi}{\partial \mathbf{X}_0} + \boldsymbol{\nu}^T \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{X}_0} \tag{1–50}$$

$$\boldsymbol{\Lambda}_f = \frac{\partial \Phi}{\partial \mathbf{X}_f} - \boldsymbol{\nu}^T \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{X}_f} \tag{1–51}$$

$$-\frac{t_f - t_0}{2} \sum_{k=1}^{N} w_k \frac{\partial \mathcal{H}_k}{\partial t_0} + \frac{1}{2} \sum_{k=1}^{N} w_k \mathcal{H}_k = \frac{\partial \Phi}{\partial t_0} - \boldsymbol{\nu}^T \frac{\partial \boldsymbol{\phi}}{\partial t_0} \tag{1–52}$$

$$\frac{t_f - t_0}{2} \sum_{k=1}^{N} w_k \frac{\partial \mathcal{H}_k}{\partial t_f} + \frac{1}{2} \sum_{k=1}^{N} w_k \mathcal{H}_k = -\frac{\partial \Phi}{\partial t_f} + \boldsymbol{\nu}^T \frac{\partial \boldsymbol{\phi}}{\partial t_f} \tag{1–53}$$

$$\mu_{jk} = 0, \text{ when } \quad C_{jk} < 0 \tag{1–54}$$

$$\mu_{jk} \leq 0, \text{ when } \quad C_{jk} = 0 \tag{1–55}$$

for $k = 1, \ldots, N$ and $j = 1, \ldots, c$. The final two equations that are required (in order to link the initial and final state and costate, respectively) are

$$\mathbf{X}_f = \mathbf{X}_0 + \frac{t_f - t_0}{2} \sum_{k=1}^{N} w_k \mathbf{f}_k \tag{1–56}$$

$$\boldsymbol{\Lambda}_f = \boldsymbol{\Lambda}_0 + \frac{t_f - t_0}{2} \sum_{k=1}^{N} w_k \left(-\frac{\partial g_k}{\partial \mathbf{X}_k} - \boldsymbol{\Lambda}_k^T \frac{\partial \mathbf{f}_k}{\partial \mathbf{X}_k} + \boldsymbol{\mu}_k^T \frac{\partial \mathbf{C}_k}{\partial \mathbf{X}_k} \right) \tag{1–57}$$

The total number of equations in set of discrete necessary conditions of Eqs. (1–46)–(1–57) is $(2n+m+c)N+4n+q+2$ (the same number of unknown variables). Solving these nonlinear algebraic equations would be an indirect solution to the optimal control problem.

1.6 Costate Estimate

One of the key features of the Gauss pseudospectral method is the ability to map the KKT multipliers of the NLP to the costates of the continuous-time optimal control problem. In particular, using the results of Sections 1.3 and 1.5, a costate estimate for the continuous Bolza problem can be obtained at the Legendre-Gauss points and the boundary points. This costate estimate is taken from (Benson, 2004) and is summarized below via the *Gauss Pseudospectral Costate Mapping Theorem:*

***Theorem 1* (Gauss Pseudospectral Costate Mapping Theorem)** *The Karush-Kuhn-Tucker (KKT) conditions of the NLP are exactly equivalent to the discretized form of the continuous first-order necessary conditions of the continuous Bolza problem when using the Gauss pseudospectral discretization. Furthermore, a costate estimate at the initial time, final time, and the Legendre-Gauss points can be found from the KKT multipliers, $\tilde{\mathbf{\Lambda}}_k$, $\tilde{\boldsymbol{\mu}}_k$, $\tilde{\mathbf{\Lambda}}_F$, and $\tilde{\boldsymbol{\nu}}$,*

$$\mathbf{\Lambda}_k = \frac{\tilde{\mathbf{\Lambda}}_k}{w_k} + \tilde{\mathbf{\Lambda}}_F \quad , \quad \boldsymbol{\mu}_k = \frac{2}{t_f - t_0}\frac{\tilde{\boldsymbol{\mu}}_k}{w_k} \quad , \quad \boldsymbol{\nu} = \tilde{\boldsymbol{\nu}}$$
$$\mathbf{\Lambda}(t_0) = \tilde{\mathbf{\Lambda}}_0 \quad , \quad \mathbf{\Lambda}(t_f) = \tilde{\mathbf{\Lambda}}_F \tag{1–58}$$

Proof of Theorem 1 *Using the substitution of Eq. (1–58), it is seen that Eqs. (1–25)–(1–37) are exactly the same as Eqs. (1–46)–(1–57).*

Theorem 1 indicates that solving the NLP derived from the Gauss pseudospectral transcription of the optimal control problem is equivalent to applying the Gauss pseudospectral discretization to the continuous-time variational conditions. Fig. 1.2 shows the solution path for both the direct and indirect methods.

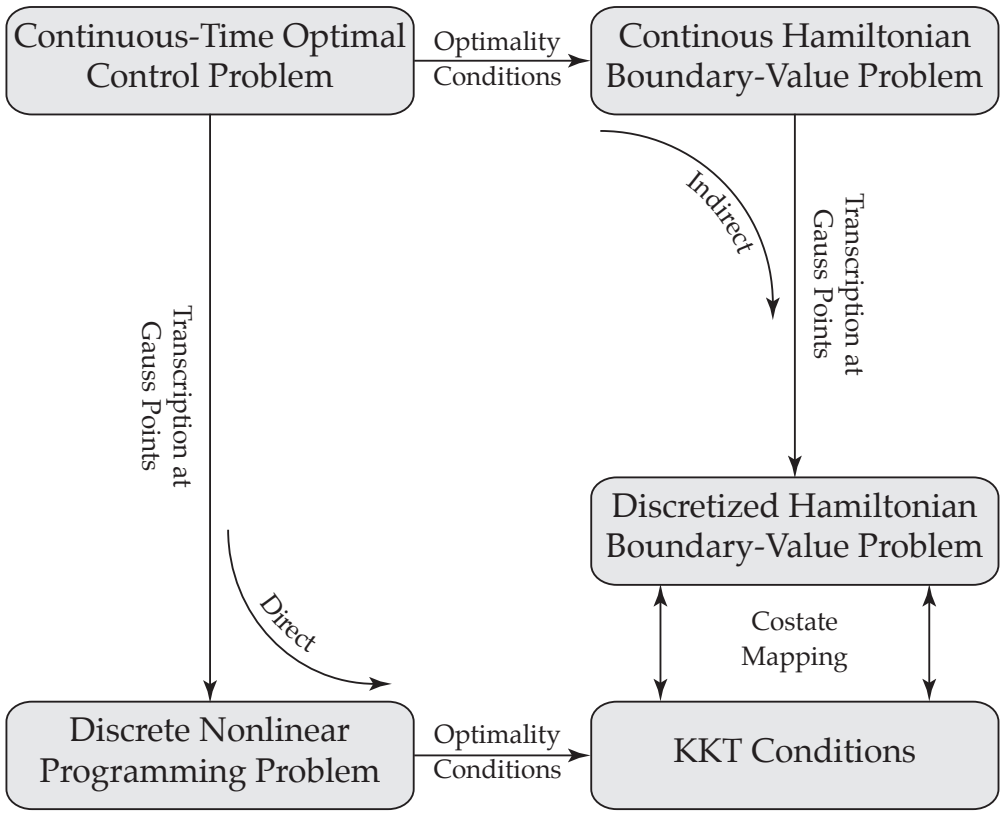

Figure 1.2 Equivalence of indirect and direct forms using the Gauss pseudospectral discretization.

1.7 Computation of Boundary Controls

It is seen in the GPM that the control is discretized only at the LG points and is *not* disctretized at either the initial or the terminal point. Consequently, the solution of the NLP defined by Eqs. (1–19), (1–20), (1–21), (1–22), and (1–23) does not produce values of the controls at the boundaries. The ability to obtain accurate initial and terminal controls can be important in many applications, particularly in guidance where real-time computation of the initial control is of vital interest.

At first glance, it may seem that the lack of control information at the boundaries can be overcome simply via extrapolation of the control at the LG points. However, multiple reasons exist as to why this is not the best approach. First, no particular functional form for the control is assumed in the GPM discretization. As a result, the best function to use for extrapolation is ambiguous. Seccond, any reasonable extrapolation of the control (e.g., linear, quadratic, cubic, or spline) may violate a path constraint which, in general, will render the extrapolated control infeasible. Third, even if the extrapolated control is feasible, it will not satisfy the required optimality conditions at the boundaries (i.e., the control will be suboptimal with respect to the NLP). Consequently, it is both practical and most rigorous to develop a systematic procedure to compute the boundary controls. We now show how to compute the boundary controls from the primal and dual solutions of the NLP arising from the Gauss pseudospectral method. It is noted that the algorithm described in this section is taken from Huntington, et. al. (2007b)

Because the approach for computing the initial and terminal control is identical, we focus on the computation of the initial control. First, recalling the augmented Hamiltonian, $\tilde{\mathcal{H}}_a$, for the continuous-time optimal control problem, we have

$$\tilde{\mathcal{H}}_a(\mathbf{x}, \mathbf{u}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \equiv g + \boldsymbol{\lambda}^T \mathbf{f} - \boldsymbol{\mu}^T \mathbf{C} \tag{1–59}$$

where shorthand notation is used. Now recall that, from the minimum principle of Pontryagin, at every instant of time the optimal control is the control $\mathbf{u}^*(\tau) \in \mathcal{U}$ that satisfies the condition

$$\tilde{\mathcal{H}}_a(\mathbf{x}^*, \mathbf{u}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) \leq \tilde{\mathcal{H}}_a(\mathbf{x}^*, \mathbf{u}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) \tag{1–60}$$

where $\mathcal{U}$ is the feasible control set. Consequently, for a given instant of time τ where $\mathbf{x}^*(\tau)$, $\boldsymbol{\lambda}^*(\tau)$, and $\boldsymbol{\mu}^*(\tau)$ are known, Eq. (1–60) is a constrained optimization problem in the $\mathbf{u}(\tau) \in \mathbb{R}^m$. In order to solve this constrained optimization problem at the *initial* time, it is necessary to know $\mathbf{x}^*(\tau_0)$, $\boldsymbol{\lambda}^*(\tau_0)$, and $\boldsymbol{\mu}^*(\tau_0)$.

Consider now the information that can be obtained by solving the NLP associated with the GPM. In particular, the primal solution to the NLP produces $\mathbf{X}(\tau_0)$ while the dual solution to the NLP can be manipulated algebraically to obtain the initial costate, $\boldsymbol{\Lambda}^*(\tau_0)$. However, because the NLP does not evaluate that path constraint at the boundaries, there is no associated Lagrange multiplier $\tilde{\boldsymbol{\mu}}(\tau_0)$. This apparent impediment can be overcome by applying the minimum principle in a manner somewhat different from that given in Eq. (1–60). In particular, suppose we let $\mathcal{H}$ be the *Hamiltonian* (not the augmented Hamiltonian), where $\mathcal{H}$ is defined as

$$\mathcal{H}(\mathbf{x}, \mathbf{u}, \boldsymbol{\lambda}) \equiv g + \boldsymbol{\lambda}^T \mathbf{f} \tag{1–61}$$

It is seen in Eq. (1–61) that the term involving the path constraint is not included. The path constraint is instead incorporated into the feasible control set. In particular, suppose we let $\mathcal{V}_0$

$$\mathcal{V}_0 = \mathcal{U} \bigcap \mathcal{C}_0 \tag{1–62}$$

where $\mathcal{V}_0$ is the intersection of the original set of feasible controls at time τ_0, denoted $\mathcal{U}$, with the set of all controls at time τ_0 that satisfy the inequality constraint of Eq. (1–23), denoted $\mathcal{C}_0$. Then, using the values of $\mathbf{X}(\tau_0)$ and $\boldsymbol{\Lambda}(\tau_0)$, the following modified optimization problem in m variables $\mathbf{U}(\tau_0) \in \mathbb{R}^m$ can be solved to obtain the initial control, $\mathbf{U}(\tau_0)$:

$$\begin{array}{ll} \text{minimize} & \mathcal{H}(\mathbf{X}(t_0), \mathbf{U}(\tau_0), \boldsymbol{\Lambda}(\tau_0), \tau_0; t_0, t_f) \\ \mathbf{U}(\tau_0) \in \mathcal{V}_0 & \end{array} \tag{1–63}$$

It is noted that, because $\mathcal{V}_0$ is restricted by the inequality path constraint at τ_0, the solution of $\mathbf{U}(\tau_0)$ is equivalent to the solution of the following problem:

$$\begin{array}{ll} \text{minimize} & \mathcal{H}(\mathbf{X}(\tau_0), \mathbf{U}(\tau_0), \boldsymbol{\Lambda}(\tau_0), \tau_0; t_0, t_f) \\ \mathbf{U}(\tau_0) \in \mathcal{U} & \\ \text{subject to} & \\ & \mathbf{C}(\mathbf{X}(\tau_0), \mathbf{U}(\tau_0), \tau_0; t_0, t_f) \leq \mathbf{0} \end{array} \tag{1–64}$$

Interestingly, if the constraint is *active*, then the initial path constraint multiplier, $\tilde{\boldsymbol{\mu}}(\tau_0)$, will also be determined by the minimization problem of Eq. (1–64). Finally, as alluded to above, the control at the terminal

time, $\mathbf{U}(\tau_f)$, can be obtained by solving the minimization problem of Eq. (1–64) at $\tau = \tau_f$, i.e.,

$$\begin{array}{rl} \text{minimize} & \mathcal{H}(\mathbf{X}(\tau_f), \mathbf{U}(\tau_f), \mathbf{\Lambda}(\tau_f), \tau_f; t_0, t_f) \\ \mathbf{U}(t_f) \in \mathcal{U} & \\ \text{subject to} & \\ & \mathbf{C}(\mathbf{X}(\tau_f), \mathbf{U}(\tau_f), \tau_f; t_0, t_f) \leq \mathbf{0} \end{array} \tag{1–65}$$

1.8 Organization of $\mathcal{GPOPS}$

$\mathcal{GPOPS}$ is organized as follows. In order to specify the optimal control problem that is to be solved, the user must write MATLAB functions that define the following functions in each phase of the problem:

(1) the cost functional

(2) the right-hand side of the differential equations and the path constraints(i.e., the differential-algebraic equations)

(3) the boundary conditions (i.e., event conditions)

(4) the linkage constraints (i.e., how the phases are connected)

In addition, the user must also specify the lower and upper limits on every component of the following quantities:

(1) initial and terminal time of the phase

(2) the state at the following points in time:

- at the beginning of the phase
- during the phase
- at the end of the phase

(3) the control

(4) the static parameters

(5) the path constraints

(6) the boundary conditions

(7) the phase duration (i.e., total length of phase in time)

(8) the linkage constraints (i.e., phase-connect conditions)

It is noted that each of the functions must be defined for each phase of the problem. The remainder of this document is devoted to describing in detail the MATLAB® syntax for describing the optimal control problem and each of the constituent functions.

1.9 Notation Used Throughout Remainder of This Manual

The following notation is adopted for use throughout the remainder of this manual. First, all user-specified names will be denoted by *slanted* characters (not *italic*, but *slanted*). Second, any item denoted by **boldface characters** are pre-defined and cannot be changed by the user. Finally, users with color capability will see the slanted characters in *red* and will see the boldface characters in **blue**.

Chapter 2

Constructing an Optimal Control Problem in $\mathcal{GPOPS}$

We now proceed to describe the constructs required to specify an optimal control problem in $\mathcal{GPOPS}$. We note that the key MATLAB programming elements used in constructing an optimal control problem in $\mathcal{GPOPS}$ are *structure* and *arrays of structures*.

2.1 Preliminary Information

Before proceeding to the details of setting up a problem in $\mathcal{GPOPS}$, the following few preliminary details are useful. First, it is important to understand that the $\mathcal{GPOPS}$ interface is laid out in ***phases***. Using a phase-based approach, it is possible to describe each segment of the problem independently of the other segments. The segments are then *linked* together using linkage conditions (or phase-connect conditions). Second, it is important to note that $\mathcal{GPOPS}$ uses the vectorization capabilities of MATLAB. In this vein all matrices and vectors in $\mathcal{GPOPS}$ are oriented ***column-wise*** for maximum efficiency. As you read through this chapter, please keep in mind the column-wise orientation of all matrices used in $\mathcal{GPOPS}$.

2.2 Call to $\mathcal{GPOPS}$

The call to $\mathcal{GPOPS}$ is deceptively simple and is given as follows:

output=gpops(setup)

The input *setup* is a user-defined structure that contains all of the information about the optimal control problem to be solved [1]. Finally, the variable *output* is a structure that contains all of the information from the original problem *plus* the information from the run itself (i.e., the solution)[2]. The remainder of this chapter is devoted to describing the fields in the structure *setup*.

2.3 Syntax for Setup Structure

The user-defined structure *setup* contains required fields and optional fields. The required fields in the structure *setup* are as follows:

- **name**: a string containing the name of the problem.
- **funcs**: a structure whose elements contain the names of the user-defined function in the problem (see Section 2.4 below).

[1] see the detailed description of *setup* in Section 2.3
[2] See the detailed description of the output in Section 2.11.

- **limits**: an array of structures that contains the information about the lower and upper limits on the variables and constraints in each phase of the problem (see Section 2.5 below).
- **guess**: an array of structures that contains contains a guess of the solution in each phase of the problem (see Section 2.8 below).

The optional fields (and their default values) are as follows:

- **linkages**: an array of structures that contains the information about the lower and upper limits of the linkage constraints (see Section 2.6 below).
- **direction**: a string that indicates the direction of the independent variable. The two possible values for this string are "increasing" and "decreasing". (default="increasing")
- **autoscale**: a string that indicates whether or not the user would like the optimal control problem to be scaled automatically before it is solved. (default="off") (see Section 2.9 below).
- **derivatives**: a string indicating differentiation method to be used. Possible values for this string are "numerical", "complex", "automatic", "automatic-INTLAB", "automatic-MAD", "analytic" (default="numerical") (see Section 2.10 below).
- **checkDerivatives**: a flag to check user defined analytic derivatives (default="0") (see Section 2.10 below).
- **maxIterations**: a positive integer indicating the maximum number of iterations that can be taken by the NLP solver.

2.4 Specifying Function Names Used in Optimal Control Problem

The syntax for specifying the names of the MATLAB functions is done by setting the fields in the structure **FUNCS** and is given as follows:

$$\begin{aligned} setup.\mathbf{funcs.cost} &= `costfun.m' \\ setup.\mathbf{funcs.dae} &= `daefun.m' \\ setup.\mathbf{funcs.event} &= `eventfun.m' \\ setup.\mathbf{funcs.link} &= `linkfun.m' \end{aligned}$$

Example of Specifying Function Names for Use in $\mathcal{GPOPS}$

Suppose we have a problem whose cost functional, differential-algebraic equations, event constraints, and linkage constraints are defined, respectively, via the *user-defined* functions *mycostfun.m*, *mydaefun.m*, *myeventfun.m*, and *mylinkfun.m*. Then the syntax for specifying these functions for use in $\mathcal{GPOPS}$ is given as follows:

```
setup.funcs.cost    = 'mycostfun';
setup.funcs.dae     = 'mydaefun';
setup.funcs.event   = 'myeventfun';
setup.funcs.link    = 'mylinkfun';
```

2.5 Syntax for limits Structure

Once the user-defined structure *setup* has been defined, the next step in setting up a problem for use with $\mathcal{GPOPS}$ is to create an array of structures of length P (where P is the number of phases) called **limits**, where **limits** is a field of the structure *setup*. The array of structures **limits** is specified as follows:

- **limits(p).nodes**: scalar value specifying the number of nodes in phase $p \in [1, \ldots, P]$.
- **limits(p).time.min** and **limits(p).time.max**: row vectors, each of length two, that contain the information about the lower and upper limits, respectively, on the initial and terminal time in phase $p \in [1, \ldots, P]$. The row vectors **limits(p).time.min** and **limits(p).time.max** have the following form:

$$\begin{array}{rcl} \textbf{limits}(p).\textbf{time.min} & = & \left[\begin{array}{cc} t_0^{\min} & t_f^{\min} \end{array} \right] \\ \textbf{limits}(p).\textbf{time.max} & = & \left[\begin{array}{cc} t_0^{\max} & t_f^{\max} \end{array} \right] \end{array}$$

- **limits(p).state.min** and **limits(p).state.max**: matrices, each of size $n_p \times 3$, that contain the lower and upper limits, respectively, on the state in phase $p \in [1, \ldots, P]$. Each of the columns of the matrices **limits(p).state.min** and **limits(p).state.max** are given as follows:
  - **limits(p).state.min(:,1)**: a column vector containing the lower (upper) limits on the state at the *start* of phase $p \in [1, \ldots, P]$.
  - **limits(p).state.min(:,2)**: a column vector containing the lower (upper) limits on the state at the *during* phase $p \in [1, \ldots, P]$.
  - **limits(p).state.min(:,3)**: a column vector containing the lower (upper) limits on the state at the *terminus* of phase $p \in [1, \ldots, P]$.

  The matrices **limits(p).state.min** and **limits(p).state.max** then have the following form:

$$\textbf{limits}(p).\textbf{state.min} = \left[\begin{array}{ccc} x_{10}^{\min} & x_1^{\min} & x_{1f}^{\min} \\ \vdots & \vdots & \vdots \\ x_{n0}^{\min} & x_n^{\min} & x_{nf}^{\min} \end{array} \right]$$

$$\textbf{limits}(p).\textbf{state.max} = \left[\begin{array}{ccc} x_{10}^{\max} & x_1^{\max} & x_{1f}^{\max} \\ \vdots & \vdots & \vdots \\ x_{n0}^{\max} & x_n^{\max} & x_{nf}^{\max} \end{array} \right]$$

- **limits(p).control.min** and **limits(p).control.max**: column vectors, each of length m_p, that contain the lower and upper limits, respectively, on the controls in phase $p \in [1, \ldots, P]$. The column vectors **limits(p).control.min** and **limits(p).control.max** have the following form:

$$\textbf{limits}(p).\textbf{control.min} = \left[\begin{array}{c} u_1^{\min} \\ \vdots \\ u_m^{\min} \end{array} \right]$$

$$\textbf{limits}(p).\textbf{control.max} = \left[\begin{array}{c} u_1^{\max} \\ \vdots \\ u_m^{\max} \end{array} \right]$$

- **limits(p).parameter.min** and **limits(p).parameter.max**: column vectors, each of length q_p, that contain the lower and upper limits, respectively, on the static parameters in phase $p \in [1, \ldots, P]$. The column vectors **limits(p).parameter.min** and **limits(p).parameters.max** have the following form:

$$\textbf{limits}(p).\textbf{parameter.min} = \left[\begin{array}{c} q_1^{\min} \\ \vdots \\ q_{q_p}^{\min} \end{array} \right]$$

$$\textbf{limits}(p).\textbf{parameter.max} = \left[\begin{array}{c} q_1^{\max} \\ \vdots \\ q_{q_p}^{\max} \end{array} \right]$$

- **limits(p).path.min** and **limits(p).path.max**: column vectors, each of length r_p, that contain the lower and upper limits, respectively, on the path constraints in phase $p \in [1, \ldots, P]$. The column vectors **limits(p).path.min** and **limits(p).path.max** have the following form:

$$\mathbf{limits}(p).\mathbf{path.min} = \begin{bmatrix} c_1^{\min} \\ \vdots \\ c_{r_p}^{\min} \end{bmatrix}$$

$$\mathbf{limits}(p).\mathbf{path.max} = \begin{bmatrix} c_1^{\max} \\ \vdots \\ c_{r_p}^{\max} \end{bmatrix}$$

- **limits(p).event.min** and **limits(p).event.max**: column vectors, each of length e_p, that contain the lower and upper limits on the event constraints in phase $p \in [1, \ldots, P]$. The column vectors **limits(p).event.min** and **limits(p).event.max** have the following form:

$$\mathbf{limits}(p).\mathbf{event.min} = \begin{bmatrix} \phi_1^{\min} \\ \vdots \\ \phi_{e_p}^{\min} \end{bmatrix}$$

$$\mathbf{limits}(p).\mathbf{event.max} = \begin{bmatrix} \phi_1^{\min} \\ \vdots \\ \phi_{e_p}^{\min} \end{bmatrix}$$

- **limits(p).duration.min** and **limits(p).duration.max**: scalars that contain the lower and upper limits on the duration of phase $p \in [1, \ldots, P]$. The scalars **limits(p).duration.min** and **limits(p).duration.max** have the following form:

$$\begin{aligned} \mathbf{limits}(p).\mathbf{duration.min} &= T^{\min} \\ \mathbf{limits}(p).\mathbf{duration.max} &= T^{\max} \end{aligned}$$

Note: any fields that do not apply to a problem (i.e. a problem without event constraints, path constraints, etc.) may be omitted or left as empty matrices ("[]").

Example of Setting Up a Limits Cell Array

As an example of setting up a limits cell array in $\mathcal{GPOPS}$, consider the following two-phase optimal control problem. In particular, suppose that *phase 1* of the problem has 3 states, 2 controls, 2 path constraints, and 5 event constraints. Suppose further that the lower and upper limits on the initial and terminal time in the first phase are given as

$$\begin{aligned} 0 &\leq t_0^{(1)} \leq 0 \\ 50 &\leq t_f^{(1)} \leq 100 \end{aligned}$$

Next, suppose that the lower and upper limits on the states at the *start* of the first phase are given, respectively, as

$$\begin{aligned} 1 &\leq x_1(t_0^{(1)}) \leq 1 \\ -3 &\leq x_2(t_0^{(1)}) \leq 0 \\ 0 &\leq x_2(t_0^{(1)}) \leq 5 \end{aligned}$$

Similarly, suppose that the lower and upper limits on the states *during* the first phase are given, respectively, as

$$\begin{aligned} 1 &\leq x_1(t^{(1)}) \leq 10 \\ -50 &\leq x_2(t^{(1)}) \leq 50 \\ -20 &\leq x_2(t^{(1)}) \leq 20 \end{aligned}$$

Finally, suppose that the lower and upper limits on the states at the *terminus* of the first phase are given, respectively, as

$$\begin{aligned} 5 &\leq x_1(t_f^{(1)}) \leq 7 \\ 2 &\leq x_2(t_f^{(1)}) \leq 2.5 \\ -\pi &\leq x_2(t_f^{(1)}) \leq \pi \end{aligned}$$

Next, suppose that the lower and upper limits on the controls *during* the first phase are given, respectively, as

$$\begin{aligned} -50 &\leq u_1(t^{(1)}) \leq 50 \\ -100 &\leq u_2(t^{(1)}) \leq 100 \end{aligned}$$

Next, suppose that the lower and upper limits on the path constraints *during* the first phase are given, respectively, as

$$\begin{aligned} -10 &\leq p_1(t^{(1)}) \leq 10 \\ 1 &\leq p_2(t^{(1)}) \leq 1 \end{aligned}$$

Next, suppose that the lower and upper limits on the event constraints of the first phase are given, respectively, as

$$\begin{aligned} 0 &\leq \phi_1^{(1)} \leq 1 \\ -2 &\leq \phi_2^{(1)} \leq 4 \\ 8 &\leq \phi_3^{(1)} \leq 20 \\ 3 &\leq \phi_4^{(1)} \leq 3 \\ 10 &\leq \phi_5^{(1)} \leq 10 \end{aligned}$$

In a similar manner, suppose that *phase 2* of the problem contains the following information: 4 states, 3 controls, 1 path constraint, and 4 event constraints. Also, suppose now that the lower and upper limits on the initial and terminal time in the first phase are given, respectively, as

$$\begin{aligned} 50 &\leq t_0^{(2)} \leq 100 \\ 100 &\leq t_f^{(2)} \leq 200 \end{aligned}$$

Next, suppose that the lower and upper limits on the states at the *start* of the second phase are given, respectively, as

$$\begin{aligned} 3 &\leq x_1(t_0^{(2)}) \leq 3 \\ -10 &\leq x_2(t_0^{(2)}) \leq 4 \\ 7 &\leq x_3(t_0^{(2)}) \leq 18 \\ 25 &\leq x_4(t_0^{(2)}) \leq 75 \end{aligned}$$

Similarly, suppose that the lower and upper limits on the states *during* the second phase are given, respectively, as

$$\begin{aligned} -200 &\leq x_1(t^{(2)}) \leq 200 \\ -50 &\leq x_2(t^{(2)}) \leq 50 \\ -20 &\leq x_3(t^{(2)}) \leq 20 \\ -80 &\leq x_4(t^{(2)}) \leq 80 \end{aligned}$$

Finally, suppose that the lower and upper limits on the states at the *terminus* of the second phase are given, respectively, as

$$\begin{aligned} 12 &\leq x_1(t_f^{(2)}) \leq 12 \\ -60 &\leq x_2(t_f^{(2)}) \leq 30 \\ -90 &\leq x_3(t_f^{(2)}) \leq 10 \\ 100 &\leq x_4(t_f^{(2)}) \leq 500 \end{aligned}$$

Next, suppose that the lower and upper limits on the controls *during* the second phase are given, respectively, as

$$\begin{aligned} -90 &\leq u_1(t^{(2)}) \leq 90 \\ -120 &\leq u_2(t^{(2)}) \leq 120 \end{aligned}$$

Next, suppose that the lower and upper limits on the path constraints *during* the second phase are given, respectively, as

$$\begin{array}{rcccl} -10 & \leq & p_1(t^{(2)}) & \leq & 10 \\ 1 & \leq & p_2(t^{(2)}) & \leq & 1 \end{array}$$

Finally, suppose that the lower and upper limits on the events constraints of the second phase phase are given, respectively, as

$$\begin{array}{rcccl} 0 & \leq & \phi_1^{(2)} & \leq & 1 \\ -2 & \leq & \phi_2^{(2)} & \leq & 4 \\ 8 & \leq & \phi_3^{(2)} & \leq & 20 \\ 3 & \leq & \phi_4^{(2)} & \leq & 3 \end{array}$$

Then a MATLAB code that would generate the above specification is given as follows:

```
iphase = 1; % Set the phase number to 1
limits(iphase).nodes = 10;
limits(iphase).time.min = [0 50];
limits(iphase).time.max = [0 100];
limits(iphase).state.min = [1 1 5; -3 -50 2; 0 -20 -pi];
limits(iphase).state.max = [1 10  7; 0  50 2.5; 5 20 pi];
limits(iphase).control.min = [-50; -100];
limits(iphase).control.max = [ 50;  100];
limits(iphase).parameter.min = [];
limits(iphase).parameter.max = [];
limits(iphase).path.min = [-10; 1];
limits(iphase).path.max = [10; 1];
limits(iphase).event.min = [0; -2; 8; 3; 10];
limits(iphase).event.max = [1; 4; 20; 3; 10];

iphase = 2; % Set the phase number to 2
limits(iphase).nodes = 10;
limits(iphase).time.min = [50 100];
limits(iphase).time.max = [100 200];
limits(iphase).state.min = [3 -200 12; -10 -50 -60; 7 -20 -90; 25 -80 100];
limits(iphase).state.max = [3 200 12; 4 50 30; 18 20 10; 75 80 500];
limits(iphase).control.min = [-90; -120];
limits(iphase).control.max = [ 90;  120];
limits(iphase).parameter.min = [];
limits(iphase).parameter.max = [];
limits(iphase).path.min = [-10; 10];
limits(iphase).path.max = [1; 1];
limits(iphase).event.min = [0; -2; 8; 3];
limits(iphase).event.max = [1; 4; 20; 3];

setup.limits = limits;
```

Note: in order to make the coding easier, we have introduced the auxiliary integer variable**iphase** so that the user can more easily reuse code from phase to phase.

2.6 Syntax for linkages Array of Structures

Another required field in the structure *setup* is an array of structures called **linkages** that defines the way that the phases are to be linked. If there is only one phase in the problem, then *setup*.**linkages** may be set to "[]". If the problem contains more than a single phase, then **linkages** is an array of structures of length L (where L is the number of pairs of phases to be linked). The array of structures **linkages** is specified as follows:

- **linkages(s).min**: a column vector of length l_s containing the lower limits on the s^{th} pair of linkages.
- **linkages(s).max**: a column vector of length l_s containing the upper limits on the s^{th} pair of linkages.
- **linkages(s).left.phase**: an integer containing the "left" phase in the pair of phases to be connected
- **linkages(s).right.phase**: an integer containing the "right" phase in the pair of phases to be connected

Note that we use the terminology "left" and "right" in the sense of viewing a graph of the trajectory on a page where time is increasing to the right. Thus, the "left" phase corresponds to the terminus of a phase while the "right" phase corresponds to the start of a phase.

2.7 Syntax of Each Function in Optimal Control Problem

Now that we know *which* functions $\mathcal{GPOPS}$ will use, the next step is to discuss the syntax of each of these functions. In general, the syntax for each function will differ because the quantities being evaluated are different in nature. In this section we will explain the syntax of each function.

2.7.1 Syntax of Function Used to Evaluate Cost

The syntax used to evaluate a user-defined cost functional is given as follows:

function [Mayer,Lagrange]=mycostfun(solcost);

where *mycostfun.m* is the name of the MATLAB function, *solcost* is the input to the function, and *Mayer* and *Lagrange* are the outputs. The input *solcost* is a structure while the outputs *Mayer* and *Lagrange* are the endpoint cost and the integrand of the integrated cost, respectively. The input structure *solcost* has the following fields (note that N=number of LG points which are on the interior of the time interval):

- *solcost*.**phase**: the phase number
- *solcost*.**initial.time**: the initial time in phase *solcost*.**phase**
- *solcost*.**initial.state**: the initial state in phase *solcost*.**phase**
- *solcost*.**terminal.time**: the terminal time in phase *solcost*.**phase**
- *solcost*.**terminal.state**: the terminal state in phase *solcost*.**phase**
- *solcost*.**time**: a column vector of length N that contains the time (excluding the initial and terminal points) in phase *solcost*.**phase**
- *solcost*.**state**: a matrix of size $N \times n$ (where n is the number of states) that contains the values of the state (excluding the initial and terminal points) in phase *solcost*.**phase**
- *solcost*.**control**: a matrix of size $N \times m$ (where m is the number of controls) that contains the values of the control (excluding the initial and terminal points) in phase *solcost*.**phase**
- *solcost*.**parameter**: a column vector of length q that contains the values of the static parameters in phase *solcost*.**phase**

Finally, the outputs of *mycostfun* are as follows:

- *Mayer*: a *scalar*, i.e., size 1×1
- *Lagrange*: a *column* vector of size $N \times 1$

2.7.2 Warning About Outputs to Cost Function

For many optimal control problems the output *Lagrange* in the user-defined cost function *mycostfun* is ***zero***. As such, it is appealing to set *Lagrange* to zero by the MATLAB command

```
Lagrange=0;
```
(2–1)

However, ***the integrand cannot be set to a scalar value!***. Instead, the integrand ***must*** be set to a ***column vector of zeros!***. The way to set the integrand to zero and that ***will work in all cases*** (i.e., numerical or automatic differentiation) is as follows:

```
Lagrange=zeros(size(solcost.time);
```
(2–2)

The user is urged to use the syntax of Eq. (2–2) whenever the integrand is identically zero.

Example of a Cost Functional

Suppose we have a two-phase optimal control problem that uses a cost functional named "mycostfun.m". Suppose further that the dimension of the state in each phase is 2 while the dimension of the control in each phase is 2. Also, suppose that the endpoint and integrand cost in phase 1 are given, respectively, as

$$\begin{aligned}\Phi^{(1)}(\mathbf{x}^{(1)}(t_0),t_0^{(1)},\mathbf{x}^{(1)}(t_f),t_f^{(1)}) &= \mathbf{x}^T(t_f)\mathbf{S}\mathbf{x}(t_f)\\ \mathcal{L}^{(1)}(\mathbf{x}^{(1)}(t),\mathbf{u}^{(1)}(t),t) &= \mathbf{x}^T\mathbf{Q}\mathbf{x}+\mathbf{u}^T\mathbf{R}\mathbf{u}\end{aligned}$$

while the endpoint and integrand in phase 2 are given, respectively, as

$$\begin{aligned}\Phi^{(2)}(\mathbf{x}^{(2)}(t_0^{(2)}),t_0^{(2)},\mathbf{x}^{(2)}(t_f^{(2)}),t_f^{(2)}) &= \mathbf{x}^T(t_f)\mathbf{x}(t_f)\\ \mathcal{L}^{(2)}(\mathbf{x}^{(2)}(t),\mathbf{u}^{(2)}(t),t) &= \mathbf{u}^T\mathbf{R}\mathbf{u}\end{aligned}$$

Then the syntax of the above cost functional is given as follows:

```
function [endpoint,integrand]=mycostfun(solcost);

Q = [5 0; 0 2];
R = [1 0; 0 3];
S = [1 5; 5 1];
iphase = solcost.phase;
t0 = solcost.initial.time;
x0 = solcost.initial.state;
tf = solcost.terminal.time;
xf = solcost.terminal.state;
t  = solcost.time;
x  = solcost.state;
u  = solcost.control;
p  = solcost.parameter;

if iphase==1,
  Mayer  = dot(xf',S*xf');
  Lagrange = dot(x,x*Q',2)+dot(u,u*R',2); % Note transposes
elseif iphase==2,
  Mayer  = dot(xf,xf);
  Lagrange = dot(u,u*R',2); % Note transposes
end;
```

It is noted in the above function call that the third argument in the command **dot** takes the dot product across the *rows*, thereby producing a *column vector*.

2.7.3 Syntax of Function Used to Evaluate Differential-Algebraic Equations

The calling syntax used evaluate the right-hand side of a user-defined vector of differential equations is given as follows:

function dae=mydaefun(soldae);

where *mydaefun.m* is the name of the MATLAB function, *soldae* is the input to the function, and *dae* is the output (i.e., the right-hand side of the differential equations and the values of the path constraints). The input *soldae* is a structure while the output *dae* is a matrix of size $N \times (n+c)$ where n is the number of differential equations, c is the number of path constraints, and N is the number of LG points. The input structure *soldae* has the following fields:

- *soldae*.**phase**: the phase number
- *soldae*.**time**: a column vector of length N that contains the time (excluding the initial and terminal points) in phase *soldae*.**phase**
- *soldae*.**state**: a matrix of size $N \times n$ (where n is the number of states) that contains the values of the state (excluding the initial and terminal points) in phase *soldae*.**phase**
- *soldae*.**control**: a matrix of size $N \times m$ (where m is the number of controls) that contains the values of the control (excluding the initial and terminal points) in phase *soldae*.**phase**
- *soldae*.**parameter**: a column vector of length q that contains the values of the static parameters in phase *soldae*.**phase**

Finally, the output of *myodefun* are as follows:

- *dae*: a *matrix* of size $N \times (n+c)$ containing the values of the right-hand side of the n differential equations and the c path constraints evaluated at the N LG points

Example of a Differential-Algebraic Equation

Suppose we have a two-phase optimal control problem that uses a differential equation function called "mydaefun.m". Suppose further that the dimension of the state in each phase is 2, the dimension of the control in each phase is 2. Furthermore, suppose that there are no path constraints in phase 1 and one path constraint in phase 2. Next, suppose that the differential equations in phase 1 are given as

$$\begin{aligned} \dot{x}_1 &= -x_1^2 - x_2^2 + u_1 u_2 \\ \dot{x}_2 &= -x_1 x_2 + 2(u_1 + u_2) \end{aligned}$$

Also, suppose that the differential equations in phase 2 are given as

$$\begin{aligned} \dot{x}_1 &= \sin(x_1^2 + x_2^2) + u_1 u_2^2 \\ \dot{x}_2 &= -\sin x_1 \cos x_2 + 2u_1 u_2 \end{aligned}$$

Finally, suppose that the path constraint in phase 2 is given as

$$u_1^2 + u_2^2 = 1$$

Then a MATLAB code that will evaluate the above system of differential-algebraic equations is given as follows:

```
function dae = mydaefun(soldae);

iphase = soldae.phase;
t = soldae.time;
x = soldae.state;
```

```
u = soldae.control;
p = soldae.parameter;

if iphase==1,
  x1dot = -x(:,1).^2-x(:,2).^2 + u(:,1).*u(:,2);
  x2dot = -x(:,1).*x(:,2) + 2*(u(:,1)+u(:,2));
  path = [];
elseif iphase==2,
  x1dot = sin(x(:,1).^2 + x(:,2).^2) + u(:,1).*u(:,2).^2;
  x2dot = -sin(x(:,1)).*cos(x(:,2))+2*u(:,1).*u(:,2);
  path  = u(:,1).^2+u(:,2).^2;
end;
dae = [x1dot x2dot path];
```

2.7.4 Syntax of Function Used to Evaluate Event Constraints

The syntax used to evaluate a user-defined vector of event constraints is given as follows:

function events=myeventfun(solevents,iphase);

where *myeventfun.m* is the name of the MATLAB function, *solevents* and *iphase* are the inputs to the function, and *event* is the output (i.e., the value of the event constraints). The inputs *solevents* and *iphase* are a cell array and an integer, respectively, while the output *event* is a *column vector* of length e where e is the number of event constraints. The input cell array *solevents* has the following elements:

- *solevents*.**phase**: the phase number
- *solevents*.**initial.time**: the time at the start of the phase
- *solevents*.**initial.state**: the state at the start of the phase
- *solevents*.**terminal.time**: the time at the terminus of the phase
- *solevents*.**terminal.state**: the state at the terminus of the phase
- *solevents*.**parameter**: the static parameters in the phase

Example of Event Constraints

Suppose we have a one-phase optimal control problem that has two initial event constraints and three terminal event constraints. Suppose further that the number of states in the phase is six and that the function that computes the values of these constraints is called "myeventfun.m". Finally, let the two initial event constraints be given as

$$\begin{aligned}\phi_{01} &= x_1(t_0)^2 + x_2(t_0)^2 + x_3(t_0)^2\\ \phi_{02} &= x_4(t_0)^2 + x_5(t_0)^2 + x_6(t_0)^2\end{aligned}$$

while the three terminal event constraints are given as

$$\begin{aligned}\phi_{f1} &= \sin(x_1(t_f))\cos(x_2(t_f) + x_3(t_f))\\ \phi_{f2} &= \tan(x_4^2(t_f) + x_5^2(t_f) + x_6^2(t_f))\\ \phi_{f3} &= x_4(t_f) + x_5(t_f) + x_6(t_f)\end{aligned}$$

Then the syntax of the above event function is given as

```
function events = myeventfun(solevents);

iphase = solevents.phase;
t0 = solevents.initial.time;
x0 = solevents.initial.state;
tf = solevents.terminal.time;
xf = solevents.terminal.state;

ei1 = dot(x0(1:3),x0(1:3));
ei2 = dot(x0(4:6),x0(4:6));
ef1 = sin(xf(1))*cos(xf(2)+xf(3));
ef2 = tan(dot(xf(4:6),xf(4:6)));
ef3 = xf(4)+xf(5)+xf(6);

events = [ei1;ei2;ef1;ef2;ef3];
```

Finally, it is noted that each event constraint need not be a function of either the initial or the terminal state, but can also be functions that contain *both* the initial and terminal state and/or the initial and terminal time. As an example of an event constraint that contains both the initial and terminal state, consider the following example.

Example of Event Constraint Containing Both Initial and Terminal State

Suppose we have a one-phase optimal control problem that contains only a single state. Furthermore, suppose that the problem contains a single event constraint on the *difference* between the terminal value of the state and the initial value of the state. Finally, suppose that the function that computes the values of these constraints is called "myeventfun.m". Then the event constraint is evaluated as

$$\phi = x(t_f) - x(t_0)$$

Then the syntax of the above event function is given as

```
function events = myeventfun(solevents);

t0 = solevents.initial.time;
x0 = solevents.initial.state;
tf = solevents.terminal.time;
xf = solevents.terminal.state;

events = xf-x0;
```

2.7.5 Syntax of Function Used to Evaluate Linkage Constraints

The syntax used to define the user defined vector of linkage constraints between two phases is given as follows:

function links=mylinkfun(sollink);

where *mylinkfun.m* is the name of the MATLAB function, *sollink* is the input to the function, and *links* is the output (i.e., the value of the linkage constraints). The input *sollink* is a structure while the output *links* is a *column vector* of length l, where l is the number of event constraints. The input structure *sollink* has the following fields:

- *sollink*.**left.phase**: the left phase of the pair of phases to be linked
- *sollink*.**right.phase**: the right phase of the pair of phases to be linked
- *sollink*.**left.state**: the state at the terminus of phase *sollink*.**left.phase**
- *sollink*.**right.state**: the state at the start of phase *sollink*.**right.phase**
- *sollink*.**left.parameter**: the static parameters in phase *sollink*.**left.phase**
- *sollink*.**right.state**: the static parameters in phase *sollink*.**right.phase**

The terms *left* and *right* are conventions adopted to help the user orient the phases on a page from left to right.

Example of Linkage Constraint

Suppose we have a multiple phase optimal control problem with a simple link between the phases, i.e. the state of the end of the phase is equal to the state at the beginning of the next phase.

$$\mathbf{P} = x^l(t_f) - x^r(t_0)$$

Then the syntax of the above linkage is given as

```
function links = mylinkagefun(sollink);

left_phase = sollink.left.phase;
right_phase = sollink.right.phase;
xf_left = sollink.left.state;
p_left  = sollink.left.parameter;
x0_left = sollink.right.phase;
p_left  = sollink.right.parameter;

links = xf_left - x0_right;
```

2.8 Specifying an Initial Guess of The Solution

The field **guess** of the user-defined structure *setup* contains the initial guess for the problem. The field **guess** is an array of structures of length P (where P is the number of phases in the problem). The p^{th} element of the array of structures **guess** contains the initial guess of the problem in phase $p \in [1, \ldots, P]$. The fields of each element of array of structures **guess** are given as follows:

- **guess(*p*).time**: a *column* vector of length s where s is the number of time points used in the guess
- **guess(*p*).state**: a matrix of size $s \times n$ where s is the number of time points and n is the number of states in the phase
- **guess(*p*).control**: a matrix of size $s \times m$ where s is the number of time points and m is the number of controls in the phase
- **guess(*p*).parameter**: a column vector of length q where q is the number of static parameters in the phase

It is noted that the element **guess(*p*).time** must be monotonic and in the same direction as that specified by the field **direction** of the structure *setup*. Schematically, in each phase of the problem the guess for the time,

states, controls, and parameters is structured as follows:

$$\textbf{guess}(p).\textbf{time} = \begin{bmatrix} t_0 \\ t_1 \\ t_2 \\ \cdots \\ t_s \end{bmatrix}$$

$$\textbf{guess}(p).\textbf{state} = \begin{bmatrix} x_{10} & x_{20} & \cdots & x_{n0} \\ x_{11} & x_{21} & \cdots & x_{n1} \\ \vdots & \vdots & \vdots & \vdots \\ x_{1s} & x_{2s} & \cdots & x_{ns} \end{bmatrix}$$

$$\textbf{guess}(p).\textbf{control} = \begin{bmatrix} u_{10} & x_{20} & \cdots & x_{m0} \\ u_{11} & u_{21} & \cdots & x_{m1} \\ \vdots & \vdots & \vdots & \vdots \\ u_{1s} & u_{2s} & \cdots & u_{ms} \end{bmatrix}$$

$$\textbf{guess}(p).\textbf{parameter} = \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_q \end{bmatrix}$$

Example of Specifying an Initial Guess

Suppose we have a two-phase problem that has three states and two controls in phase 1 while it has two states and one control in phase 2. Furthermore, suppose that we choose five time points for the guess in phase 1 while we choose 3 time points for the guess in phase 2. A MATLAB code that would create such an initial guess is given below.

```
iphase = 1;
guess(iphase).time  = [0; 1; 3; 5; 7];
guess(iphase).state(:,1) = [1.27; 3.1; 5.8; 9.6; -13.7272];
guess(iphase).state(:,2) = [-4.2; -9.6; 8.5; 25.73; 100.00];
guess(iphase).state(:,3) = [18.727; 1.827; 25.272; -14.272; 26.84];
guess(iphase).control(:,1) = [8.4; -13.7; -26.5; 19; 87];
guess(iphase).control(:,2) = [-1.2; 5.8; -3.77; 14; 19.787];
guess(iphase).parameter = [];

iphase = 2;
guess(iphase).time = [7; 7.5; 8];
guess(iphase).state(:,1) = [0.5; 1.5; 8];
guess(iphase).state(:,2) = [-0.5; -2.5; 19];
guess(iphase).control(:,1) = [8.4; -13.7; -26.5; 19; 87];
guess(iphase).parameter = [];

setup.guess = guess;
```

It is noted again that, for the above example, auxiliary integer variables were used to minimize the cumbersomeness of coding and to minimize the chance of error.

2.9 Scaling of Optimal Control Problem

As with any numerical optimization procedure, the approach employed by $\mathcal{GPOPS}$ requires a well-scaled optimal control problem. In general, it is recommended that the user scale the problem in accordance with any known large discrepancies either in the sizes of various quantities (i.e., state, control) or the sizes of the derivatives of such quantities. While it is beyond the scope of this user's manual to provide a general procedure for scaling, in an attempt to reduce the burden on the user an automatic scaling procedure has been developed for use in $\mathcal{GPOPS}$. This procedure is based on the scaling algorithm developed in (Betts, 2001). In order to invoke the automatic scaling routine, the user must set the field **autoscale** in the user-defined structure *setup* to the string "on".

The automatic scaling procedure operates as follows. The bounds on the variables are used to scale all components of the state, control, parameters, and time to lie between -1 and 1. As a result, it is essential that the user provide *sensible* bounds on all quantities (e.g., do not provide unreasonably large bounds as this will result in a poorly scaled problem). Next, the constraints are scaled to make the row norms of the Jacobians of the respective functions approximately unity. The automatic scaling procedure is by no means foolproof, but it has been found in practice to work well on many problems that otherwise would require scaling by hand. The advice given here is to try the automatic scaling procedure, but not to use it for too long if it is proving to be unsuccessful.

2.10 Different Options for Specification of Derivatives

The user has six choices for the computation of the derivatives of the objective function gradient and the constraint Jacobian for use within the NLP solver. As stated above, the choices for **derivatives** are "numerical", "complex", "automatic", "automatic-INTLAB", "automatic-MAD", and "analytic" and correspond to the following differentiation methods:

- *setup*.**derivatives**="numerical": default finite-differencing algorithm within SNOPT is used.
- *setup*.**derivatives**="complex": the *built-in* complex-step differentiation method is used.
- *setup*.**derivatives**="automatic", the *built-in* automatic differentiator is used.
- *setup*.**derivatives**="automatic-INTLAB": automatic differentiation using the third-party program *INTLAB* is used (if the program INTLAB is installed on your computer).
- *setup*.**derivatives**="automatic-MAD": Matlab Automatic Differentiation (MAD) is used (if the program MAD is installed on your computer)
- *setup*.**derivatives**="analytic": analytic derivatives (supplied by the user) are used.

It is noted that INTLAB can be obtained from Prof. Siegfried Rump by visiting the URL `http://www.ti3.tu-harburg.de/rump/intlab/` while MAD can be obtained for a nominal charge by visiting `www.tomopt.com/`. Note that the functionality for MAD has been coded in $\mathcal{GPOPS}$ however, it is not officially supported and therefore may not continue to function in future versions of $\mathcal{GPOPS}$ and/or MAD. The authors recommend using either the built-in automatic differentiator or INTLAB if automatic differentiation is desired.

2.10.1 Complex-Step Differentiation

Of the differention methods given above, either the built-in automatic differentiator or the complex-step differentiator most preferred because these two methods provide highly accurate derivatives and are both included as part of the $\mathcal{GPOPS}$ software (i.e., the user does not have to obtain any third-party software). One drawback with complex-step differentiation, however, is that certain functions need to be handled with great care. In particular, the functions **min**, **max**, **abs**, and **dot** need to be redefined for use in complex-step differentiation (see Ref. Martins, et al. (2003) and the URL `http://mdolab.utias.utoronto.ca/`

resources/complex-step/complexify.f90 for details). Finally, the transpose operator must be replaced with a dot-transpose (i.e., a *real transpose*) because the standard transpose in MATLAB produces a complex conjugate transpose and it is necessary to maintain a real transpose when computing derivatives via complex-step differentiation.

2.10.2 Analytic Differentiation

Analytic differentiation has the advantage that it is the fasted and most accurate of the four methods, however, it is by far the most complex for the user to compute, code, and verify. The derivatives for the objective function gradient and the constraint Jacobian are computed from the user defined analytic derivatives. These derivatives are supplied as an additional output of the user functions for the cost, dae functions, event constraints, and linkage constraints (if applicable). The user defined derivatives can be checked relative to a finite-difference approximation by setting the flag *setup*.**checkDerivatives** equal to one. Upon execution of $\mathcal{GPOPS}$, the derivatives will be computed at the user supplied initial guess using a finite-difference approximation and compared to the analytic derivatives with the results printed to the screen. It is recommended that the user run the derivative checking algorithm a least one time to verify that the derivatives are correct, however, it should be noted that the algorithm is not guaranteed to find any incorrect derivatives. The user must take special care to ensure that the analytic derivatives are coded correctly in order to take advantage of the speed and accuracy of analytic differentiation.

Syntax of Function Used to Evaluate Cost with Derivatives

The syntax used to evaluate the user-defined cost derivatives is given as follows:

function [Mayer,Lagrange,DerivMayer,DerivLagrange]=mycostfun(solcost);

See Section 2.7.1 for the definition of the regular inputs/outputs. The additional outputs of *mycostfun* are as follows:

- *DerivMayer*: a *row* vector of size $1 \times (2n + 2 + q)$
- *DerivLagrange*: a *matrix* of size $N \times (n + m + q + 1)$

where n is the number of states, m is the number of controls, q is the number of parameters, and N is the number of LG points in the phase. The row vector *DerivMayer* defines the partial derivatives of the Mayer cost with respect to the initial state, initial time, final state, final time, and finally the parameters:

$$\textbf{DerivMayer} = \left[\frac{\partial \Phi}{\partial \mathbf{x}(t_0)}, \quad \frac{\partial \Phi}{\partial t_0}, \quad \frac{\partial \Phi}{\partial \mathbf{x}(t_f)}, \quad \frac{\partial \Phi}{\partial t_f}, \quad \frac{\partial \Phi}{\partial p}\right]$$

The matrix *DerivLagrange* defines the partial derivatives of the Lagrange cost with respect to the state, control, parameters, and time at each of the N LG points:

$$\textbf{DerivLagrange} = \left[\frac{\partial \mathcal{L}}{\partial \mathbf{x}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{u}}, \quad \frac{\partial \mathcal{L}}{\partial p}, \quad \frac{\partial \mathcal{L}}{\partial t}\right]$$

It is important to provide all the derivatives in the correct order *even if* they are zero.

Example of a Cost Functional with Derivatives

Suppose we have a two-phase optimal control problem that uses a cost functional named "mycostfun.m". Suppose further that the dimension of the state in each phase is 2 while the dimension of the control in each phase is 2. Also, suppose that the endpoint and integrand cost in phase 1 are given, respectively, as

$$\begin{aligned}\Phi^{(1)}(\mathbf{x}^{(1)}(t_0), t_0^{(1)}, \mathbf{x}^{(1)}(t_f), t_f^{(1)}) &= \mathbf{x}^T(t_f)\mathbf{S}\mathbf{x}(t_f)\\ \mathcal{L}^{(1)}(\mathbf{x}^{(1)}(t), \mathbf{u}^{(1)}(t), t) &= \mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{u}^T\mathbf{R}\mathbf{u}\end{aligned}$$

while the endpoint and integrand in phase 2 are given, respectively, as

$$\begin{aligned}\Phi^{(2)}(\mathbf{x}^{(2)}(t_0^{(2)}),t_0^{(2)},\mathbf{x}^{(2)}(t_f^{(2)}),t_f^{(2)}) &= \mathbf{x}^T(t_f)\mathbf{x}(t_f)\\ \mathcal{L}^{(2)}(\mathbf{x}^{(2)}(t),\mathbf{u}^{(2)}(t),t) &= \mathbf{u}^T\mathbf{R}\mathbf{u}\end{aligned}$$

Then the syntax of the above cost functional is given as follows:

```
function [Mayer,Lagrange,DerivMayer,DerivLagrange]=mycostfun(solcost,iphase);

Q = [5 0; 0 2];
R = [1 0; 0 3];
S = [1 5; 5 1];
t0 = solcost.initial.time;
x0 = solcost.initial.state;
tf = solcost.terminal.time;
xf = solcost.terminal.state;
t  = solcost.time;
x  = solcost.state;
u  = solcost.control;
p  = solcost.parameter;

if iphase==1,
  Mayer  = dot(xf',S*xf');
  Lagrange = dot(x,x*Q',2)+dot(u,u*R',2); % Note transposes
  DerivMayer = [zeros(1,length(x0)), zeros(1,length(t0)), ...
                xf'*S, zeros(1,length(tf), zeros(1,length(p))];
  DerivLagrange = [x*Q', u*R', ...
                   zeros(length(t),length(p)), zeros(size(t))];
elseif iphase==2,
  Mayer  = dot(xf,xf);
  Lagrange = dot(u,u*R',2); % Note transposes
  DerivMayer = [zeros(1,length(x0)), zeros(1,length(t0)), ...
                xf', zeros(1,length(tf), zeros(1,length(p))];
  DerivLagrange = [zeros(size(x)), u*R', ...
                   zeros(length(t),length(p)), zeros(size(t))];
end;
```

It is noted in the above function call that the third argument in the command **dot** takes the dot product across the *rows*, thereby producing a *column vector*.

Syntax of Function Used to Evaluate Differential-Algebraic Equations with Derivatives

The calling syntax used evaluate the derivatives of the right-hand side of a user-defined vector of differential equations is given as follows:

function [dae,Derivdae]=mydaefun(soldae);

See Section 2.7.3 for the definition of the regular inputs/outputs. The additional output of *myodefun* is as follows:

- *Derivdae*: a *matrix* of size $N(n+c)\times(n+m+q+1)$

where n is the number of states, m is the number of controls, q is the number of parameters, c is the number of path constraints, and N is the number of LG points in the phase. The matrix *Derivdae* defines the partial derivatives of the differential equations and path constraints with respect to the state, control, parameters, and time at each of the N LG points:

$$\textbf{Derivdae} = \begin{bmatrix} \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}}, & \frac{\partial \mathbf{f}_1}{\partial \mathbf{u}}, & \frac{\partial \mathbf{f}_1}{\partial p}, & \frac{\partial \mathbf{f}_1}{\partial t} \\ \vdots, & \vdots, & \vdots, & \vdots \\ \frac{\partial \mathbf{f}_n}{\partial \mathbf{x}}, & \frac{\partial \mathbf{f}_n}{\partial \mathbf{u}}, & \frac{\partial \mathbf{f}_n}{\partial p}, & \frac{\partial \mathbf{f}_n}{\partial t} \\ \frac{\partial \mathbf{C}_1}{\partial \mathbf{x}}, & \frac{\partial \mathbf{C}_1}{\partial \mathbf{u}}, & \frac{\partial \mathbf{C}_1}{\partial p}, & \frac{\partial \mathbf{C}_1}{\partial t} \\ \vdots, & \vdots, & \vdots, & \vdots \\ \frac{\partial \mathbf{C}_r}{\partial \mathbf{x}}, & \frac{\partial \mathbf{C}_r}{\partial \mathbf{u}}, & \frac{\partial \mathbf{C}_r}{\partial p}, & \frac{\partial \mathbf{C}_r}{\partial t} \end{bmatrix}$$

where $\mathbf{f}_i$, $(i = 1, \ldots, n)$ is the right-hand side of the i^{th} differential equation, and $\mathbf{C}_j$, $(j = 1, \ldots, r)$ is the j^{th} path constraint. It is important to provide all the derivatives in the correct order *even if* they are zero.

Example of a Differential-Algebraic Equation with Derivatives

Suppose we have a two-phase optimal control problem that uses a differential equation function called "mydaefun.m". Suppose further that the dimension of the state in each phase is 2, the dimension of the control in each phase is 2. Furthermore, suppose that there are no path constraints in phase 1 and one path constraint in phase 2. Next, suppose that the differential equations in phase 1 are given as

$$\begin{aligned} \dot{x}_1 &= -x_1^2 - x_2^2 + u_1 u_2 \\ \dot{x}_2 &= -x_1 x_2 + 2(u_1 + u_2) \end{aligned}$$

Also, suppose that the differential equations in phase 2 are given as

$$\begin{aligned} \dot{x}_1 &= \sin(x_1^2 + x_2^2) + u_1 u_2^2 \\ \dot{x}_2 &= -\sin x_1 \cos x_2 + 2u_1 u_2 \end{aligned}$$

Finally, suppose that the path constraint in phase 2 is given as

$$u_1^2 + u_2^2 = 1$$

Then a MATLAB code that will evaluate the above system of differential-algebraic equations is given as follows:

```
function [dae, Derivdae] = mydaefun(soldae);

iphase = soldae.phase;
t = soldae.time;
x = soldae.state;
u = soldae.control;
p = soldae.parameter;

if iphase==1,
  x1dot = -x(:,1).^2-x(:,2).^2 + u(:,1).*u(:,2);
  x2dot = -x(:,1).*x(:,2) + 2*(u(:,1)+u(:,2));
  path = [];
  df1_dx1 = -2*x(:,1);
  df1_dx2 = -2*x(:,2);
  df1_du1 = u(:,2);
  df1_du2 = u(:,1);
```

```
  df2_dx1 = -x(:,2);
  df2_dx2 = -x(:,1);
  df2_du1 = 2*ones(size(t));
  df2_du2 = 2*ones(size(t));
  dpath_dx1 = [];
  dpath_dx2 = [];
  dpath_du1 = [];
  dpath_du2 = [];
  dpath_dp = [];
  dpath_dt = [];
elseif iphase==2,
  x1dot = sin(x(:,1).^2 + x(:,2).^2) + u(:,1).*u(:,2).^2;
  x2dot = -sin(x(:,1)).*cos(x(:,2)) + 2*u(:,1).*u(:,2);
  path  = u(:,1).^2+u(:,2).^2;
  df1_dx1 = 2*x(:,1)*cos(x(:,1).^2 + x(:,2).^2);
  df1_dx2 = 2*x(:,2)*cos(x(:,1).^2 + x(:,2).^2);
  df1_du1 = u(:,2).^2;
  df1_du2 = 2*u(:,1).*u(:,2);
  df2_dx1 = -cos(x(:,1)).*cos(x(:,2));
  df2_dx2 = sin(x(:,1)).*sin(x(:,2));
  df2_du1 = 2*u(:,2);
  df2_du2 = 2*u(:,1);
  dpath_dx1 = zeros(size(x(:,1)));
  dpath_dx2 = zeros(size(x(:,2)));
  dpath_du1 = 2*u(:,1);
  dpath_du2 = 2*u(:,2);
  dpath_dp = zeros(length(t),length(p));
  dpath_dt = zeros(size(t));
end;
df1_dp = zeros(length(t),length(p));
df1_dt = zeros(size(t));
df2_dp = zeros(length(t),length(p));
df2_dt = zeros(size(t));

dae = [x1dot x2dot path];

Derivdae = [df1_dx1,   df1_dx2,   df1_du1,   df1_du2,   df1_dp,   df1_dt; ...
            df2_dx1,   df2_dx2,   df2_du1,   df2_du2,   df2_dp,   df2_dt; ...
          dpath_dx1, dpath_dx2, dpath_du1, dpath_du2, dpath_dp, dpath_dt];
```

Syntax of Function Used to Evaluate Event Constraints with Derivatives

The syntax used to evaluate the derivative of a user-defined vector of event constraints is given as follows:

function [events, Derivevents]=myeventfun(solevents);

See Section 2.7.4 for the definition of the regular inputs/outputs. The additional output of *myeventfun* is as follows:

- *Derivevents*: a *matrix* of size $e \times (2n + 2 + q)$

where n is the number of states, q is the number of parameters, and e is the number of event constraints in the phase. The matrix *Derivevents* defines the partial derivatives of each event constraint with respect to the initial state, initial time, final state, final time, and parameters:

$$\textbf{Derivevents} = \begin{bmatrix} \frac{\partial \phi_1}{\partial \mathbf{x}(t_0)}, & \frac{\partial \phi_1}{\partial t_0}, & \frac{\partial \phi_1}{\partial \mathbf{x}(t_f)}, & \frac{\partial \phi_1}{\partial t_f}, & \frac{\partial \phi_1}{\partial p} \\ \vdots, & \vdots, & \vdots, & \vdots, & \vdots \\ \frac{\partial \phi_e}{\partial \mathbf{x}(t_0)}, & \frac{\partial \phi_e}{\partial t_0}, & \frac{\partial \phi_e}{\partial \mathbf{x}(t_f)}, & \frac{\partial \phi_e}{\partial t_f}, & \frac{\partial \phi_e}{\partial p} \end{bmatrix}$$

where ϕ_i, $(i = 1, \ldots, e)$ is the i^{th} event constraint. It is important to provide all the derivatives in the correct order *even if* they are zero.

Example of Event Constraints with Derivatives

Suppose we have a one-phase optimal control problem that has two initial event constraints and three terminal event constraints. Suppose further that the number of states in the phase is six and that the function that computes the values of these constraints is called "myeventfun.m". Finally, let the two initial event constraints be given as

$$\begin{aligned} \phi_{01} &= x_1(t_0)^2 + x_2(t_0)^2 + x_3(t_0)^2 \\ \phi_{02} &= x_4(t_0)^2 + x_5(t_0)^2 + x_6(t_0)^2 \end{aligned}$$

while the three terminal event constraints are given as

$$\begin{aligned} \phi_{f1} &= \sin(x_1(t_f))\cos(x_2(t_f) + x_3(t_f)) \\ \phi_{f2} &= \tan(x_4^2(t_f) + x_5^2(t_f) + x_6^2(t_f)) \\ \phi_{f3} &= x_4(t_f) + x_5(t_f) + x_6(t_f) \end{aligned}$$

Then the syntax of the above event function is given as

```
function [events, Derivevents] = myeventfun(solevents);

iphase = solevents.phase;
t0 = solevents.initial.time;
x0 = solevents.initial.state;
tf = solevents.terminal.time;
xf = solevents.terminal.state;

ei1 = dot(x0(1:3),x0(1:3));
ei2 = dot(x0(4:6),x0(4:6));
ef1 = sin(xf(1))*cos(xf(2)+xf(3));
ef2 = tan(dot(xf(4:6),xf(4:6)));
ef3 = xf(4)+xf(5)+xf(6);

events = [ei1;ei2;ef1;ef2;ef3];

dei1_dx0 = [2*x0(1:3).' zeros(1,3)];
dei1_dt0 = 0;
dei1_dxf = zeros(1,6);
dei1_dtf = 0;
dei1_dp = [];
dei1_dt = 0;
dei2_dx0 = [zeros(1,3), 2*x0(4:6).'];
dei2_dt0 = 0;
dei2_dxf = zeros(1,6);
dei2_dtf = 0;
dei2_dp = [];
def1_dx0 = zeros(1,6);
```

```
def1_dt0 = 0;
def1_dxf = [cos(xf(1))*cos(xf(2)+xf(3)), -sin(xf(1))*sin(xf(2)+xf(3)), ...
            -sin(xf(1))*sin(xf(2)+xf(3)), zeros(1,3)];
def1_dtf = 0;
def1_dp = [];
def2_dx0 = zeros(1,6);
def2_dt0 = 0;
def2_dxf = [zeros(1,3), 2*xf(4:6).']/(cos(dot(xf(4:6),xf(4:6))))^2;
def2_dtf = 0;
def2_dp = [];
def3_dx0 = zeros(1,6);
def3_dt0 = 0;
def3_dxf = [zeros(1,3), ones(1,3)];
def3_dtf = 0;
def3_dp = [];

Derivevents = [dei1_dx0, dei1_dt0, dei1_dxf, dei1_dtf, dei1_dp, dei1_dt; ...
               dei2_dx0, dei2_dt0, dei1_dxf, dei2_dtf, dei2_dp, dei2_dt; ...
               def1_dx0, def1_dt0, def1_dxf, def1_dtf, def1_dp, def1_dt; ...
               def2_dx0, def2_dt0, def2_dxf, def2_dtf, def2_dp, def2_dt; ...
               def3_dx0, def3_dt0, def3_dxf, def3_dtf, def3_dp, def3_dt];
```

Syntax of Function Used to Evaluate Linkage Constraints with Derivatives

The syntax used to define the user defined vector of linkage constraints between two phases is given as follows:

function [links,Derivlinks]=mylinkfun(sollink);

See Section 2.7.5 for the definition of the regular inputs/outputs. The additional output of *mylinkfun* is as follows:

- *Derivlinks*: a *matrix* of size $l \times (n^l + q^l + n^r + q^r)$

where l is the number of linkages in the constraint, n^l is the number of states in the left phase, q^l is the number of parameters in the left phase, n^r is the number of states in the right phase, and q^r is the number of parameters in the right phase. The matrix *Derivlinks* defines the partial derivatives of each linkage with respect to the left state, left parameters, right state, and right parameters:

$$\textbf{Derivlinks} = \begin{bmatrix} \frac{\partial \mathbf{P}_1}{\partial \mathbf{x}^l(t_f)}, & \frac{\partial \mathbf{P}_1}{\partial p^l}, & \frac{\partial \mathbf{P}_1}{\partial \mathbf{x}^r(t_0)}, & \frac{\partial \mathbf{P}_1}{\partial p^r} \\ \vdots, & \vdots, & \vdots, & \vdots \\ \frac{\partial \mathbf{P}_l}{\partial \mathbf{x}^l(t_f)}, & \frac{\partial \mathbf{P}_l}{\partial p^l}, & \frac{\partial \mathbf{P}_l}{\partial \mathbf{x}^r(t_0)}, & \frac{\partial \mathbf{P}_l}{\partial p^r} \end{bmatrix}$$

where $\mathbf{P}_i$, $(i = 1, \ldots, l)$ is the i^{th} linkage constraint. It is important to provide all the derivatives in the correct order *even if* they are zero.

Example of Linkage Constraint with Derivatives

Suppose we have a multiple phase optimal control problem with a simple link between the phases, i.e. the state of the end of the phase is equal to the state at the beginning of the next phase.

$$\mathbf{P} = x^l(t_f) - x^r(t_0)$$

Then the syntax of the above linkage is given as

```
function [links, Derivlinks] = mylinkagefun(sollink,left_phase,right_phase);

xf_left = sollink.left.state;
p_left  = sollink.left.parameter;
x0_right = sollink.right.state;
p_right  = sollink.right.parameter;

links = xf_left - x0_right;

nlink = length(xf_left); %number of linkages
Derivlinks = [ eye(nlink), zeros(nlink,length(p_left)), ...
              -eye(nlink), zeros(nlink,length(p_right))];
```

2.11 Output of Execution of $\mathcal{GPOPS}$

Upon execution of $\mathcal{GPOPS}$, new fields are created in the output structure *output*. In particular, upon completion of the execution of $\mathcal{GPOPS}$, the following new fields are created (in addition to the fields that were created prior to running $\mathcal{GPOPS}$ on the problem):

- **solution:** an array of structures of length P (where P is the number of phases) containing the solution in each phase

The p^{th} element in the array of structures **solution** contains the solution in phase $p \in [1, \ldots, P]$. The fields of the array of structures **solution** are as follows:

- **solution(*p*).time:** a column vector of size $M \times 1$ containing the time at each point along the trajectory (where $M = N + 2$ is the number of time points and N is the number of LG points)
- **solution(*p*).state:** a matrix of size $M \times n$ such that the rows contain the state at the time points along the trajectory
- **solution(*p*).control:** a matrix of size $M \times m$ such that the rows contain the state at the time points along the trajectory
- **solution(*p*).parameter:** a column vector of length q containing the static parameters
- **solution(*p*).costate:** a matrix of size $M \times n$ such that the rows contain the costate at each time point along the trajectory
- **solution(*p*).pathmult:** a structure containing the Lagrange multipliers of the path constraints
- **solution(*p*).Hamiltonian:** a column vector of size $M \times 1$ that contains the Hamiltonian at each time point along the trajectory
- **solution(*p*).Mayer_cost:** The Mayer part of the cost along the trajectory
- **solution(*p*).Lagrange_cost:** The Lagrange (integrated) cost along the trajectory

2.12 Useful Information for Debugging a $\mathcal{GPOPS}$ Problem

2.12.1 Debugging Code when Using Automatic Differentiation

As stated above, one of the options in $\mathcal{GPOPS}$ is the use automatic differentiation, either the built-in code, INTLAB or Matlab automatic Differentiation (MAD). As is often the case, the user may want to break in

the various functions to ensure proper coding of the functions. In the case where numerical derivatives are used, the variables can be printed in the MATLAB command window by breaking in the function and typing the appropriate variable name. However, when automatic differentiation is being used, all variables are defined as custom MATLAB objects. The objects are not a real-valued variable, but is a MATLAB object that contains information about both the *value* of the variables and the *derivatives* of the variables. If a user wants to obtain the value of a variable, additional commands are necessary. In the built-in automatic differentiator, the value is obtained using ".value". For example the value of a variable named "y" is obtained by typing the command "y.value" (and not simply by typing "y"). When using INTLAB, the value is obtained using ".x" (see INTLAB documentation), i.e. the command "y.x" will return the value for variable named "y". Finally in MAD, the command to get a value is "getvalue" (see the TOMLAB/MAD documentation). For example the value of a variable named "y" is obtained by typing the command "getvalue(y)". The user is urged to keep this in mind when using automatic differentiation to compute derivatives.

2.12.2 Dimensions of Arrays When Debugging $\mathcal{GPOPS}$ Code

One aspect of $\mathcal{GPOPS}$ that may appear confusing when debugging code pertains to the dimensions of the arrays and the corresponding time values. It is important to remember that $\mathcal{GPOPS}$ uses collocation at *Legendre-Gauss* points. Because the Legendre-Gauss points lie on the interior of the time interval of interest, the dynamics, path constraints, and integrand cost are computed only at the Legendre-Gauss points. While this may appear to be a bit strange, the fundamental point here is that Gaussian quadrature (which is used in $\mathcal{GPOPS}$) only evaluates the functions at the Legendre-Gauss points. Do not try to "fool" $\mathcal{GPOPS}$ by adding the endpoints to the computation of the dynamics, path constraints, or integrand cost. If you do this, you will get an error because the dimensions are incorrect. For a more complete mathematical description of the collocation method used in *GPOP*, see either Chapter 1 of this manual or the references contained in the bibliography at the end of this manual.

Chapter 3

Examples of Using $\mathcal{GPOPS}$

In this Chapter we provide three complete examples of using $\mathcal{GPOPS}$. For each example the optimal control problem is first described quantitatively, then the $\mathcal{GPOPS}$ code is provided.

3.1 Hyper-Sensitive Problem

Consider the following optimal control problem. Minimize the cost functional

$$J = \frac{1}{2}\int_0^{t_f} \left[x^2 + u^2\right] dt \tag{3–1}$$

subject to the dynamic constraint

$$\dot{x} = -x^3 + u \tag{3–2}$$

and the boundary conditions

$$\begin{array}{rcl} x(0) & = & 1.5 \\ x(t_f) & = & 1 \end{array} \tag{3–3}$$

with $t_f = 50$. It is noted that this problem is taken from Rao (2000). The $\mathcal{GPOPS}$ code that solves this problem is shown below. In particular, the following three MATLAB functions are defined:

- hyperSensitiveMain.m: MATLAB m-file (main driver) for problem
- hyperSensitiveCost.m: MATLAB function that evaluates the cost functional
- hyperSensitiveDae.m: MATLAB function that evaluates the differential-algebraic equations

The beginning and end of each function is labeled by a MATLAB comment.

```
%------------------------------------
% BEGIN: script hyperSensitiveMain.m
%------------------------------------
% This m-file is the main file for the following optimal control
% problem:
%   minimize
%     J = 0.5*(x^2+u^2)
%   subject to
%     dx/dt = -x^3 + u
%     x(0) = 1.5
%     x(tf) = 1
%--------------------------------------------------------------------
% This example is taken from the following reference:
% Rao, A. V. and Mease, K. D., "Eigenvector Approximate Dichotomic
```

```
% Basis Method for Solving Hypersensitive Optimal Control
% Problems," Optimal Control Applications and Methods, Vol. 21,
% No. 1, 2000, pp. 1-19.
%--------------------------------------------------------------------

clear setup limits guess

x0 = 1.5;
xf = 1;
xmin = -50;
xmax =  50;
umin = -50;
umax =  50;

iphase = 1;
limits(iphase).nodes = 50;
limits(iphase).time.min = [0 50];
limits(iphase).time.max = [0 50];
limits(iphase).state.min = [x0 xmin xf];
limits(iphase).state.max = [x0 xmax xf];
limits(iphase).control.min = umin;
limits(iphase).control.max = umax;
limits(iphase).parameter.min = [];
limits(iphase).parameter.max = [];
limits(iphase).path.min = [];
limits(iphase).path.max = [];
limits(iphase).event.min = [];
limits(iphase).event.max = [];
guess(iphase).time = [0; 20];
guess(iphase).state(:,1) = [x0; x0];
guess(iphase).control = [0; 0];
guess(iphase).parameter = [];

clear x0 xf xmin xmax umin umax

setup.name  = 'HyperSensitive-Problem';
setup.funcs.cost = 'hyperSensitiveCost';
setup.funcs.dae = 'hyperSensitiveDae';
setup.linkages = [];
setup.limits = limits;
setup.guess = guess;
setup.derivatives = 'automatic';
setup.checkDerivatives = 0;
setup.direction = 'increasing';
setup.autoscale = 'off';

output = gpops(setup);

%-----------------------------------
% END: script hyperSensitiveMain.m
%-----------------------------------

%-----------------------------------
% BEGIN: function hyperSensitiveCost.m
%-----------------------------------
function [Mayer,Lagrange,DMayer,DLagrange] = hyperSensitiveCost(sol);

t0 = sol.initial.time;
x0 = sol.initial.state;
tf = sol.terminal.time;
xf = sol.terminal.state;
t  = sol.time;
x  = sol.state;
u  = sol.control;
p  = sol.parameter;
Mayer = zeros(size(t0));
Lagrange = 0.5*(x.^2+u.^2);
```

```
if nargout == 4
    % DMayer = [           dM/dx0,              dM/dt0,              dM/dxf,
    DMayer = [zeros(1,length(x0)), zeros(1,length(t0)), zeros(1,length(xf)), ...
    ... %                  dM/dtf,              dM/dp]
             zeros(1,length(tf)), zeros(1,length(p))];

    % DLagrange = [ dL/dx, dL/du,                        dL/dp,          dL/dt]
    DLagrange =[        x,     u,  zeros(length(t),length(p)), zeros(size(t))];
end

%-------------------------------------
% END: function hyperSensitiveCost.m
%-------------------------------------

%-------------------------------------
% BEGIN: function hyperSensitiveDae.m
%-------------------------------------
function [dae Ddae] = hyperSensitiveDae(sol);

t = sol.time;
x = sol.state;
u = sol.control;
p = sol.parameter;

dae = -x.^3+u;

if nargout == 2
    % Ddae = [df/dx,         df/du,                       df/dp,          df/dt]
    Ddae = [-3*x.^2, ones(size(u)), zeros(length(t),length(p)), zeros(size(t))];
end

%-----------------------------------
% END: function hyperSensitiveDae.m
%-----------------------------------
```

The output of the above code from $\mathcal{GPOPS}$ is summarized in the following three plots that contain the state (x), costate (λ), and the Hamiltonian (H), respectively, for the problem (where $H = L + \lambda f$ where $L = 0.5(x^2 + u^2)$ and $f = -x^3 + u$).

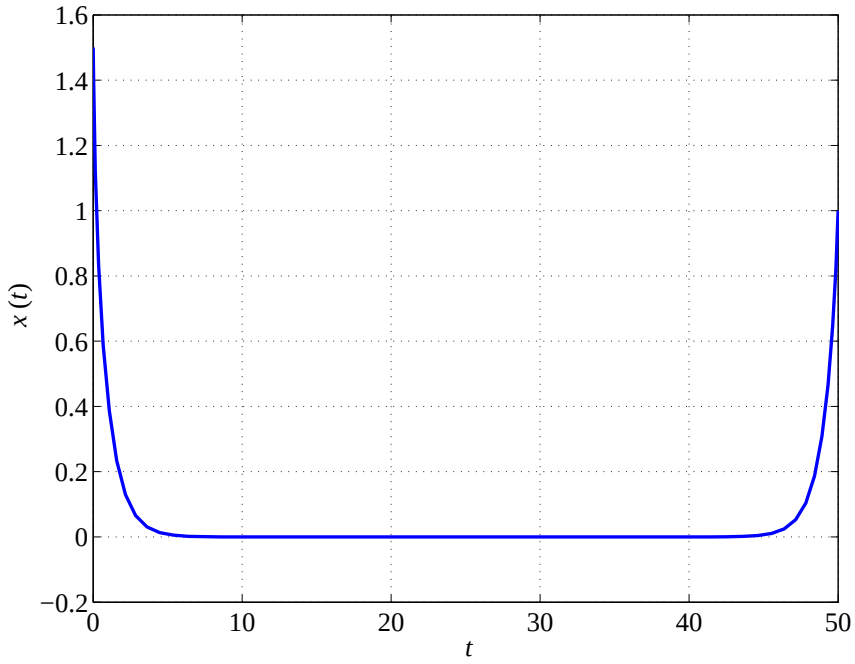

Figure 3.1 $x(t)$ vs. t for one-dimensional problem.

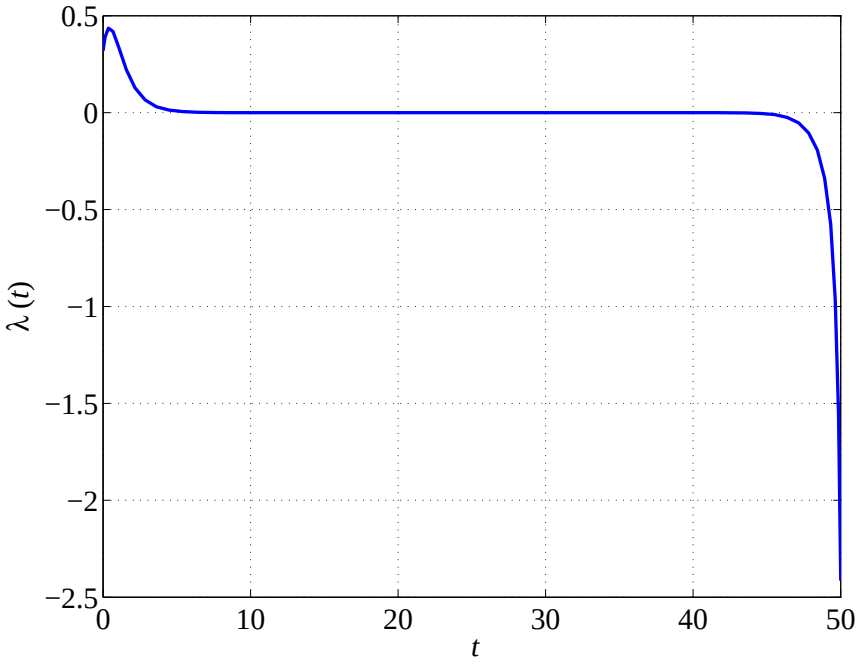

Figure 3.2 $\lambda(t)$ vs. t for one-dimensional problem.

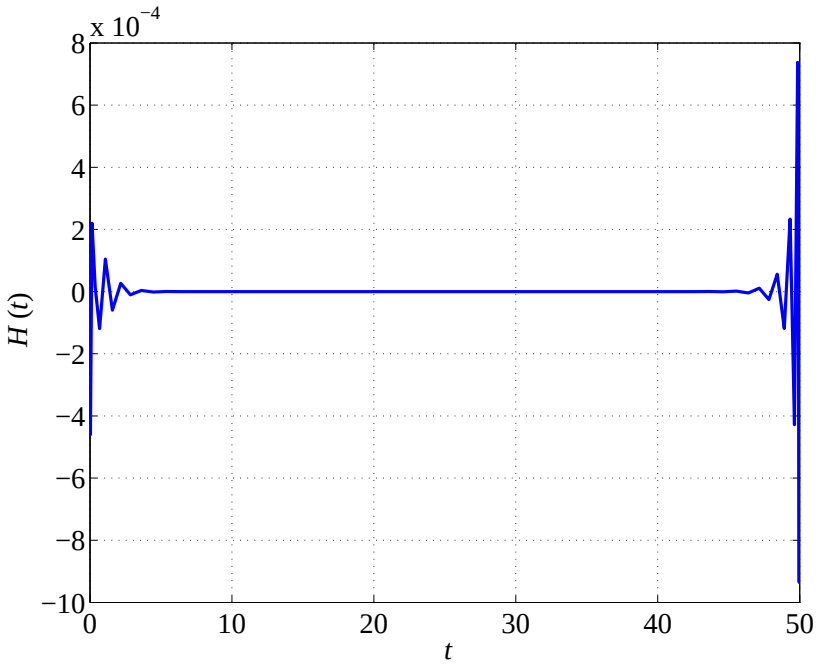

Figure 3.3 H vs. t for one-dimensional problem.

3.2 Bryson-Denham Problem

Consider the following optimal control problem. Minimize the cost functional

$$J = x_3(t_f) \tag{3–4}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= u \\ \dot{x}_3 &= \tfrac{1}{2}u^2 \end{aligned} \tag{3–5}$$

the path constraint

$$0 \leq x_1(t) \leq 1/9 \tag{3–6}$$

and the boundary conditions

$$\begin{aligned} x_1(0) &= 0 \\ x_2(0) &= 1 \\ x_3(0) &= 0 \\ x_1(t_f) &= 0 \\ x_2(t_f) &= -1 \end{aligned} \tag{3–7}$$

The above problem was originally formulated by Bryson and Denham (Bryson, et al., 1963) and is referred to as the *Bryson-Denham* problem. The $\mathcal{GPOPS}$ code that solves the Bryson-Denham problem is shown below. In particular, the following four MATLAB files are defined:

- brysonDenhamMain.m: MATLAB m-file (main driver) for problem
- brysonDenhamCost.m: MATLAB function that evaluates the cost functional
- brysonDenhamDae.m: MATLAB function that evaluates the differential-algebraic equation
- brysonDenhamEvent.m: MATLAB function that evaluates the event constraints

The beginning and end of each function is labeled by a MATLAB comment. It is noted that while all five boundary conditions are simple bounds (and are, thus, linear, they are treated as general event constraints in order to demonstrate the proper use of an event function.

```
% ----------------------------
% Bryson-Denham Example Problem
% ----------------------------
% ------------------------------------------------
% This example is taken from the following reference:
% ------------------------------------------------
% Bryson, A. E., Denham, W. F., and Dreyfus, S. E., "Optimal
% Programming Problems with Inequality Constraints.  I: Necessary
% Conditions for Extremal Solutions, AIAA Journal, Vol. 1, No. 11,
% November, 1963, pp. 2544-2550.
clear setup limits guess

x10 = 0;
x20 = 1;
x30 = 0;
x1f = 0;
x2f = -1;
x1min = -10;
x1max = 10;
x2min = x1min;
x2max = x1max;
x3min = x1min;
x3max = x1max;

param_min = [];
param_max = [];
path_min = 0;
path_max = 1/9;
event_min = [x10; x20; x30; x1f; x2f];
event_max = [x10; x20; x30; x1f; x2f];
duration_min = [];
duration_max = [];

iphase = 1;
limits(iphase).nodes = 50;
limits(iphase).time.min = [0 0];
limits(iphase).time.max = [0 50];
limits(iphase).state.min(1,:) = [x1min x1min x1min];
limits(iphase).state.max(1,:) = [x1max x1max x1max];
limits(iphase).state.min(2,:) = [x2min x2min x2min];
limits(iphase).state.max(2,:) = [x2max x2max x2max];
limits(iphase).state.min(3,:) = [x3min x3min x3min];
limits(iphase).state.max(3,:) = [x3max x3max x3max];
limits(iphase).control.min    = -5000;
limits(iphase).control.max    =  5000;
limits(iphase).parameter.min  = param_min;
limits(iphase).parameter.max  = param_max;
limits(iphase).path.min       = path_min;
limits(iphase).path.max       = path_max;
limits(iphase).event.min      = event_min;
limits(iphase).event.max      = event_max;
limits(iphase).duration.min    = [];
limits(iphase).duration.max    = [];
guess(iphase).time             = [0; 0.5];
guess(iphase).state(:,1)      = [x10; x1f];
guess(iphase).state(:,2)      = [x20; x2f];
guess(iphase).state(:,3)      = [x30; x30];
guess(iphase).control         = [0; 0];
guess(iphase).parameter       = [];

clear x10 x20 x30 x1f x2f x1min x1max x2min x2max x3min x3max
clear param_min param_max path_min path_max event_min event_max
```

```
clear duration_min duration_max iphase

setup.name  = 'Bryson-Denham-Problem';
setup.funcs.cost = 'brysonDenhamCost';
setup.funcs.dae = 'brysonDenhamDae';
setup.funcs.event = 'brysonDenhamEvent';
setup.limits = limits;
setup.guess = guess;
setup.linkages = [];
setup.derivatives = 'automatic';
setup.direction = 'increasing';
setup.autoscale = 'off';

output = gpops(setup);
solution = output.solution;
%-------------------------------------
% END: script brysonDenhamMain.m
%-------------------------------------
plotresults
close all

%-------------------------------------
% BEGIN: function brysonDenhamCost.m
%-------------------------------------
function [Mayer,Lagrange,DMayer,DLagrange]=brysonDenhamCost(sol);

t0 = sol.initial.time;
x0 = sol.initial.state;
tf = sol.terminal.time;
xf = sol.terminal.state;
t  = sol.time;
x  = sol.state;
u  = sol.control;
p  = sol.parameter;

Mayer = xf(3);
Lagrange = zeros(size(t));

if nargout == 4
    DMayer = [0 0 0 0 0 0 1 0];
    DLagrange = [zeros(size(t)) zeros(size(t)) zeros(size(t)) zeros(size(t)) zeros(size(t))];
end

%-------------------------------------
% END: function brysonDenhamCost.m
%-------------------------------------

%-------------------------------------
% BEGIN: function brysonDenhamDae.m
%-------------------------------------
function [dae,Ddae] = brysonDenhamDae(sol);

t = sol.time;
x = sol.state;
u = sol.control;
x1dot = x(:,2);
x2dot = u;
x3dot = u.^2/2;
path = x(:,1);
dae = [x1dot x2dot x3dot path];
if nargout == 2
    x1dotx1 = zeros(size(t));
    x1dotx2 = ones(size(t));
    x1dotx3 = zeros(size(t));
    x1dotu  = zeros(size(t));
    x1dott  = zeros(size(t));
    x2dotx1 = zeros(size(t));
```

```
    x2dotx2 = zeros(size(t));
    x2dotx3 = zeros(size(t));
    x2dotu  = ones(size(t));
    x2dott  = zeros(size(t));
    x3dotx1 = zeros(size(t));
    x3dotx2 = zeros(size(t));
    x3dotx3 = zeros(size(t));
    x3dotu  = u;
    x3dott  = zeros(size(t));
    pathx1 = ones(size(t));
    pathx2 = zeros(size(t));
    pathx3 = zeros(size(t));
    pathu  = zeros(size(t));
    patht  = zeros(size(t));
    Ddae = [x1dotx1 x1dotx2 x1dotx3 x1dotu x1dott;
            x2dotx1 x2dotx2 x2dotx3 x2dotu x2dott;
        x3dotx1 x3dotx2 x3dotx3 x3dotu x3dott;
        pathx1  pathx2  pathx3  pathu  patht];
end

%-------------------------------------
% END: function brysonDenhamDae.m
%-------------------------------------
```

The output from $\mathcal{GPOPS}$ of the Bryson-Denham problem coded above is summarized in the following plots that contain the components of the state, the component of the costate, and the control, respectively.

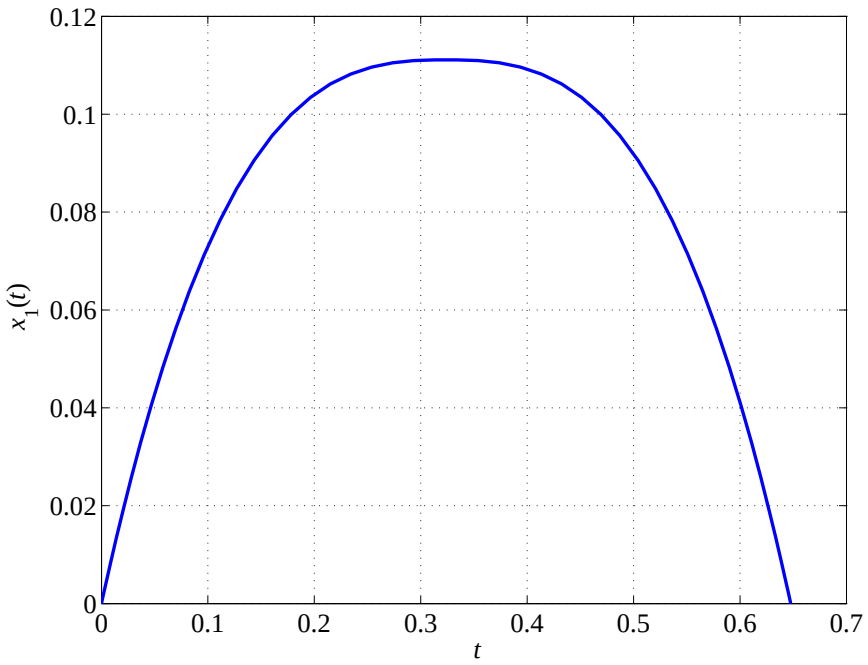

Figure 3.4 $x_1(t)$ vs. t for Bryson-Denham problem.

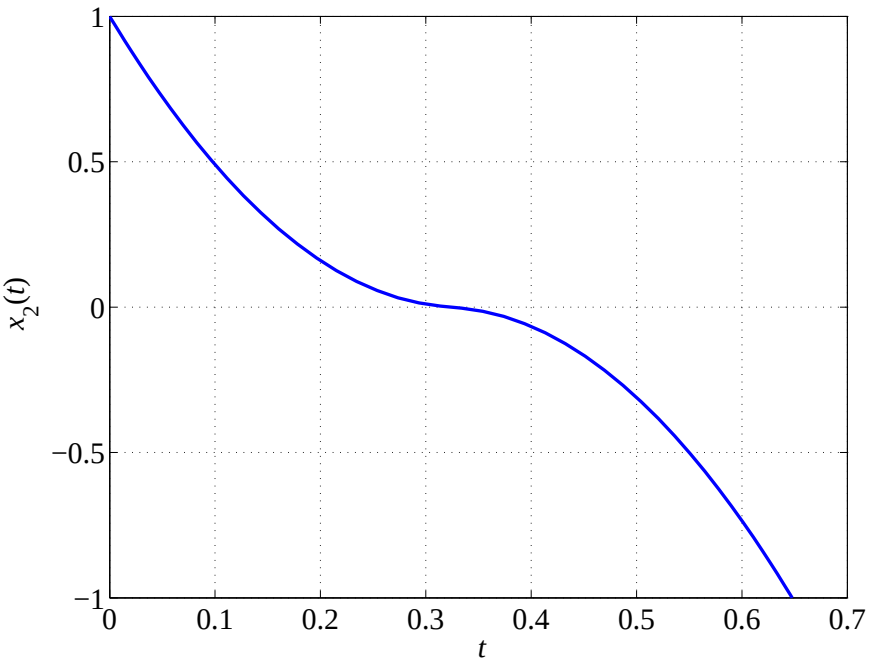

Figure 3.5 $x_2(t)$ vs. t for Bryson-Denham problem.

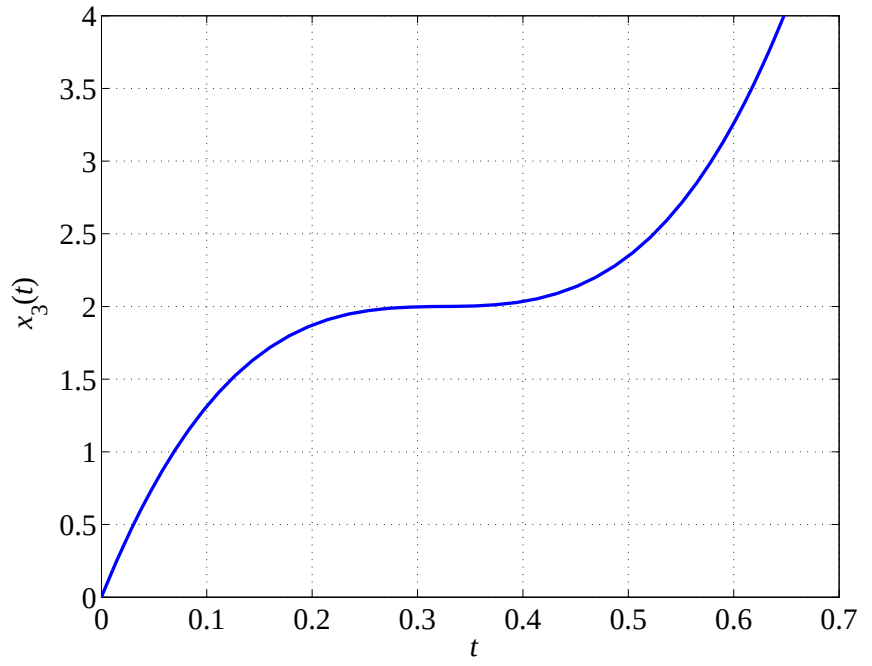

Figure 3.6 $x_3(t)$ vs. t for Bryson-Denham problem.

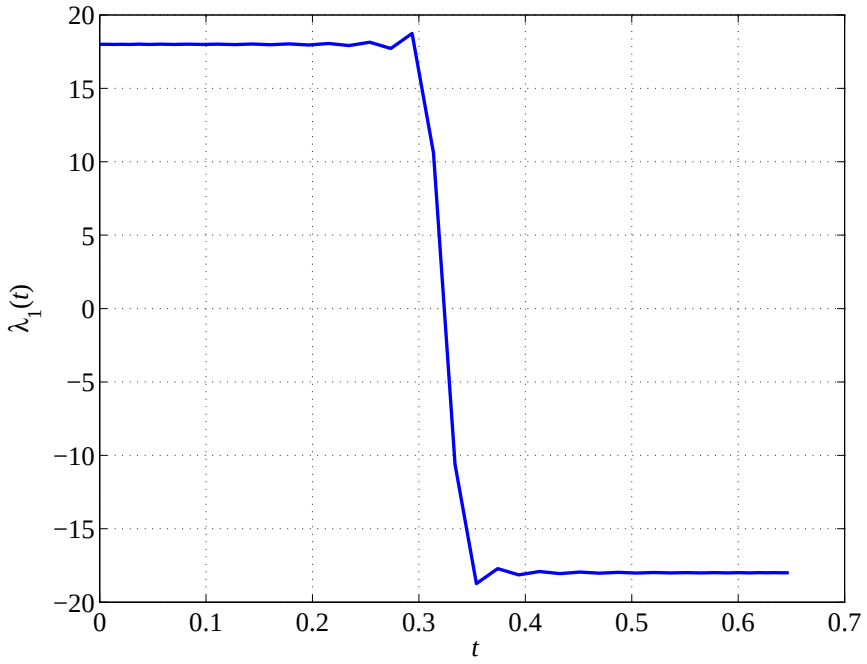

Figure 3.7 $\lambda_1(t)$ vs. t for Bryson-Denham problem.

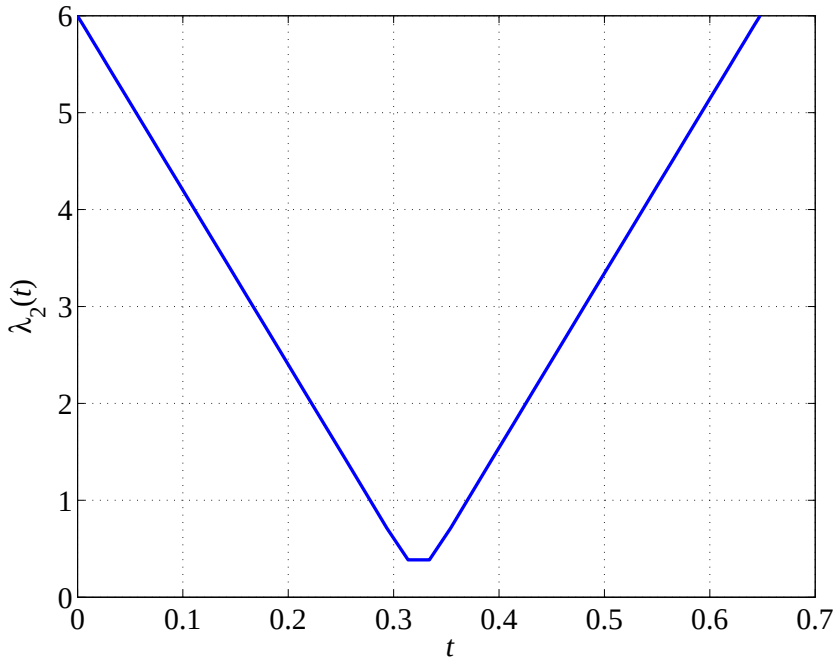

Figure 3.8 $\lambda_2(t)$ vs. t for Bryson-Denham problem.

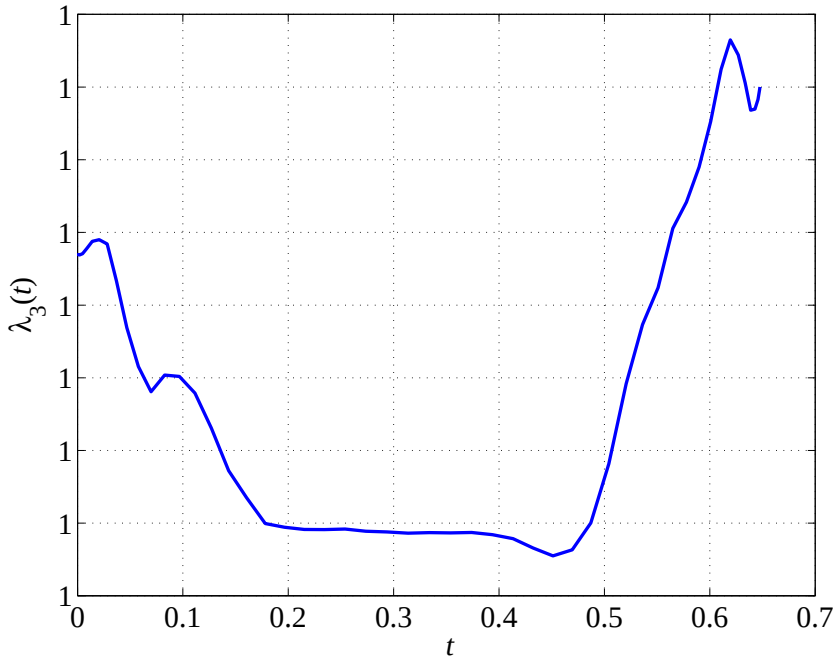

Figure 3.9 $\lambda_3(t)$ vs. t for Bryson-Denham problem.

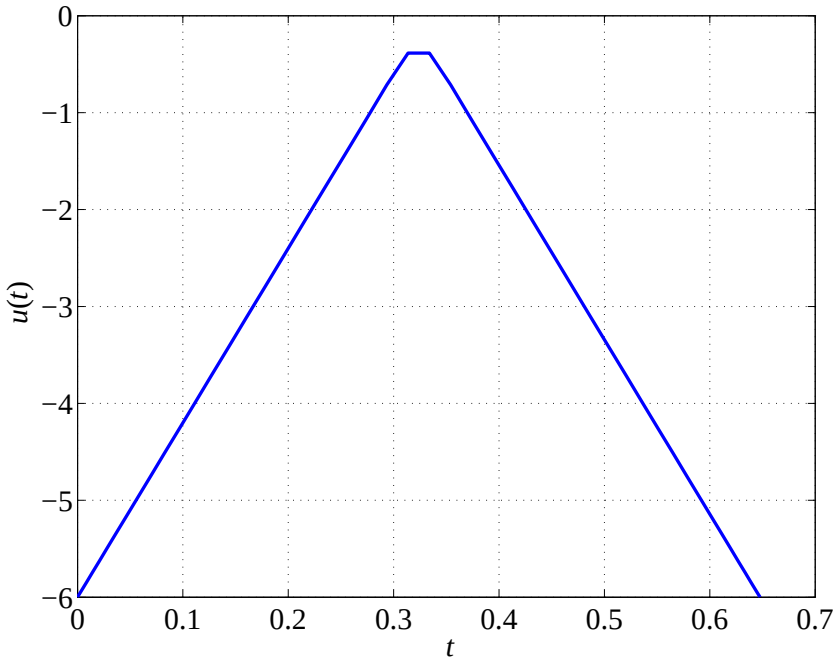

Figure 3.10 $u(t)$ vs. t for Bryson-Denham problem.

3.3 Multiple-Stage Launch Vehicle Ascent Problem

The problem considered in this section is the ascent of a multiple-stage launch vehicle. The objective is to maneuver the launch vehicle from the ground to the target orbit while maximizing the remaining fuel in

the upper stage. It is noted that this example is taken verbatim from Benson (2004).

3.3.1 Vehicle Properties

The launch vehicle considered in this example has two main stages along with nine strap-on solid rocket boosters. The flight of the vehicle can be divided into *four* distinct phases. The first phase begins with the rocket at rest on the ground and at time t_0, the main engine and six of the nine solid boosters ignite. When the boosters are depleted at time t_1, their remaining dry mass is jettisoned. The final three boosters are then ignited, and along with the main engine, represent the thrust for the second phase of flight. These three remaining boosters are jettisoned when their fuel is exhausted at time t_2, and the main engine alone creates the thrust for the third phase. The fourth phase begins when the main engine fuel has been exhausted (MECO) and the dry mass associated with the main engine is ejected at time t_3. The thrust during phase four is from a second stage, which burns until the target orbit has been reached (SECO) at time t_4, thus completing the trajectory. The specific characteristics of these rocket motors can be seen in Table 3.1. Note that the solid boosters and main engine burn for their entire duration (meaning t_1, t_2, and t_3 are fixed), while the second stage engine is shut off when the target orbit is achieved (t_4 is free).

Table 3.1 Mass and propulsion properties of the launch vehicle ascent problem.

	Solid Boosters	Stage 1	Stage 2
Total Mass (kg)	19290	104380	19300
Propellant Mass (kg)	17010	95550	16820
Engine Thrust (N)	628500	1083100	110094
Isp (sec)	284	301.7	462.4
Number of Engines	9	1	1
Burn Time (sec)	75.2	261	700

3.3.2 Dynamic Model

The equations of motion for a non-lifting point mass in flight over a spherical rotating planet are expressed in Cartesian Earth centered inertial (ECI) coordinates as

$$\begin{aligned} \dot{\mathbf{r}} &= \mathbf{v} \\ \dot{\mathbf{v}} &= -\frac{\mu}{\|\mathbf{r}\|^3}\mathbf{r} + \frac{T}{m}\mathbf{u} + \frac{\mathbf{D}}{m} \\ \dot{m} &= -\frac{T}{g_0 I_{sp}} \end{aligned} \tag{3–8}$$

where $\mathbf{r}(t) = \begin{bmatrix} x(t) & y(t) & z(t) \end{bmatrix}^T$ is the position, $\mathbf{v} = \begin{bmatrix} v_x(t) & v_y(t) & v_z(t) \end{bmatrix}^T$ is the Cartesian ECI velocity, μ is the gravitational parameter, T is the vacuum thrust, m is the mass, g_0 is the acceleration due to gravity at sea level, I_{sp} is the specific impulse of the engine, $\mathbf{u} = \begin{bmatrix} u_x & u_y & u_z \end{bmatrix}^T$ is the thrust direction, and $\mathbf{D} = \begin{bmatrix} D_x & D_y & D_z \end{bmatrix}^T$ is the drag force. The drag force is defined as

$$\mathbf{D} = -\frac{1}{2} C_D A_{ref} \rho \|\mathbf{v}_{rel}\| \mathbf{v}_{rel} \tag{3–9}$$

where C_D is the drag coefficient, A_{ref} is the reference area, ρ is the atmospheric density, and $\mathbf{v}_{rel}$ is the Earth relative velocity, where $\mathbf{v}_{rel}$ is given as

$$\mathbf{v}_{rel} = \mathbf{v} - \boldsymbol{\omega} \times \mathbf{r} \tag{3–10}$$

where $\boldsymbol{\omega}$ is the angular velocity of the Earth relative to inertial space. The atmospheric density is modeled as the exponential function

$$\rho = \rho_0 \text{exp}[-h/h_0] \tag{3–11}$$

where ρ_0 is the atmospheric density at sea level, $h = \|\mathbf{r}\| - R_e$ is the altitude, R_e is the equatorial radius of the Earth, and h_0 is the density scale height. The numerical values for these constants can be found in Table 3.2.

Table 3.2 Constants used in the launch vehicle example.

Constant	Value
Payload Mass (kg)	4164
A_{ref} (m^2)	4π
C_d	0.5
ρ_0 (kg/m^3)	1.225
h_0 (km)	7.2
t_1 (s)	75.2
t_2 (s)	150.4
t_3 (s)	261
R_e (km)	6378.14
V_E (km/s)	7.905

3.3.3 Constraints

The launch vehicle starts on the ground at rest (relative to the Earth) at time t_0, so that the ECI initial conditions are

$$\begin{aligned} \mathbf{r}(t_0) &= \mathbf{r}_0 = \begin{bmatrix} 5605.2 & 0 & 3043.4 \end{bmatrix}^T \quad \text{km} \\ \mathbf{v}(t_0) &= \mathbf{v}_0 = \begin{bmatrix} 0 & 0.4076 & 0 \end{bmatrix}^T \quad \text{km/s} \\ m(t_0) &= m_0 = 301454 \quad \text{kg} \end{aligned} \tag{3–12}$$

which corresponds to the Cape Canaveral launch site. The terminal constraints define the target geosynchronous transfer orbit (GTO), which is defined in orbital elements as

$$\begin{aligned} a_f &= 24361.14 \text{ km}, \\ e_f &= 0.7308, \\ i_f &= 28.5 \deg, \\ \Omega_f &= 269.8 \deg, \\ \omega_f &= 130.5 \deg \end{aligned} \tag{3–13}$$

The orbital elements, a, e, i, Ω, and ω represent the semi-major axis, eccentricity, inclination, right ascension of the ascending node (RAAN), and argument of perigee, respectively. Note that the true anomaly, ν, is left undefined since the exact location within the orbit is not constrained. These orbital elements can be transformed into ECI coordinates via the transformation, T_{o2c}, where T_{o2c} is given in (Bate, et al., 2001).

In addition to the boundary constraints, there exists both a state path constraint and a control path constraint in this problem. A state path constraint is imposed to keep the vehicle's altitude above the surface of the Earth, so that

$$|\mathbf{r}| \geq R_r \tag{3–14}$$

where R_e is the radius of the Earth, as seen in Table 3.2. Next, a path constraint is imposed on the control to guarantee that the control vector is unit length, so that

$$|\mathbf{u}| = 1 \tag{3–15}$$

Lastly, each of the four phases in this trajectory is linked to the adjoining phases by a set of linkage conditions. These constraints force the position and velocity to be continuous and also account for the mass

ejections, as

$$\begin{aligned} \mathbf{r}^{(p)}(t_f) - \mathbf{r}^{(p+1)}(t_0) &= \mathbf{0}, \\ \mathbf{v}^{(p)}(t_f) - \mathbf{v}^{(p+1)}(t_0) &= \mathbf{0}, \qquad (p = 1, \ldots, 3) \\ m^{(p)}(t_f) - m^{(p)}_{dry} - m^{(p+1)}(t_0) &= 0 \end{aligned} \tag{3–16}$$

where the superscript (p) represents the phase number.

The optimal control problem is then to find the control, $\mathbf{u}$, that minimizes the cost function

$$J = -m^{(4)}(t_f) \tag{3–17}$$

subject to the conditions of Eqs. (3–8), (3–12), (3–13), (3–14), and (3–15).

The MATLAB code that solves the multiple-stage launch vehicle ascent problem using $\mathcal{GPOPS}$ is shown below. In particular, this problem requires the specification of a function that computes the cost functional, the differential-algebraic equations (which, it is noted, include both the differential equations *and* the path constraints), and the event constraints in each phase of the problem along with the phase-connect (i.e., linkage) constraints. The problem was posed in SI units and the built-in autoscaling procedure was used.

```
% ------------------------------------------
% Multiple-Stage Launch Vehicle Ascent Example
% ------------------------------------------
% ------------------------------------------------------------------------
% This example can be found in one of the following three references:
%   Benson, D. A., A Gauss Pseudospectral Transcription for Optimal
%   Control, Ph.D. Thesis, Department of Aeronautics and
%   Astronautics, Massachusetts Institute of Technology, November 2004.
%
%   Huntington, G. T. Advancement and Analysis of a Gauss
%   Pseudospectral Transcription for Optimal Control, Ph.D. Thesis,
%   Department of Aeronautics and Astronautics, Massachusetts
%   Institute of Technology, May 2007.
%
%   Huntington, G. T., Benson, D. A., Kanizay, N., Darby, C. L.,
%   How, J. P., and Rao, A. V., "Computation of Boundary Controls
%   Using a Gauss Pseudospectral Method," 2007 Astrodynamics
%   Specialist Conference, Mackinac Island, Michigan, August 19-23, 2007.
% ------------------------------------------------------------------------
clear setup limits guess linkages

global CONSTANTS

omega            = 7.29211585e-5; % Earth rotation rate (rad/s)
omega_matrix  = [0 -omega 0; omega 0 0; 0 0 0];
CONSTANTS.omega_matrix = omega_matrix; % Rotation rate matrix (rad/s)
CONSTANTS.mu = 3.986012e14;       % Gravitational parameter (m^3/s^2)
CONSTANTS.cd = 0.5;               % Drag coefficient
CONSTANTS.sa = 4*pi;              % Surface area (m^2)
CONSTANTS.rho0 = 1.225;           % sea level gravity (kg/m^3)
CONSTANTS.H = 7200.0;             % Density scale height (m)
CONSTANTS.Re = 6378145.0;         % Radius of earth (m)
CONSTANTS.g0 = 9.80665;           % sea level gravity (m/s^2)

lat0 = 28.5*pi/180;               % Geocentric Latitude of Cape Canaveral
x0 = CONSTANTS.Re*cos(lat0);      % x component of initial position
z0 = CONSTANTS.Re*sin(lat0);      % z component of initial position
y0 = 0;
r0 = [x0; y0; z0];
v0 = CONSTANTS.omega_matrix*r0;

bt_srb = 75.2;
bt_first = 261;
bt_second = 700;

t0 = 0;
```

```
t1 = 75.2;
t2 = 150.4;
t3 = 261;
t4 = 961;

m_tot_srb     = 19290;
m_prop_srb    = 17010;
m_dry_srb     = m_tot_srb-m_prop_srb;
m_tot_first   = 104380;
m_prop_first  = 95550;
m_dry_first   = m_tot_first-m_prop_first;
m_tot_second  = 19300;
m_prop_second = 16820;
m_dry_second  = m_tot_second-m_prop_second;
m_payload     = 4164;
thrust_srb    = 628500;
thrust_first  = 1083100;
thrust_second = 110094;
mdot_srb      = m_prop_srb/bt_srb;
ISP_srb       = thrust_srb/(CONSTANTS.g0*mdot_srb);
mdot_first    = m_prop_first/bt_first;
ISP_first     = thrust_first/(CONSTANTS.g0*mdot_first);
mdot_second   = m_prop_second/bt_second;
ISP_second     = thrust_second/(CONSTANTS.g0*mdot_second);

af = 24361140;
ef = 0.7308;
incf = 28.5*pi/180;
Omf = 269.8*pi/180;
omf = 130.5*pi/180;
nuguess = 0;
cosincf = cos(incf);
cosOmf = cos(Omf);
cosomf = cos(omf);
oe = [af ef incf Omf omf nuguess];
[rout,vout] = launchoe2rv(oe,CONSTANTS.mu);
rout = rout';
vout = vout';

m10 = m_payload+m_tot_second+m_tot_first+9*m_tot_srb;
m1f = m10-(6*mdot_srb+mdot_first)*t1;
m20 = m1f-6*m_dry_srb;
m2f = m20-(3*mdot_srb+mdot_first)*(t2-t1);
m30 = m2f-3*m_dry_srb;
m3f = m30-mdot_first*(t3-t2);
m40 = m3f-m_dry_first;
m4f = m_payload;

CONSTANTS.thrust_srb    = thrust_srb;
CONSTANTS.thrust_first  = thrust_first;
CONSTANTS.thrust_second = thrust_second;
CONSTANTS.ISP_srb       = ISP_srb;
CONSTANTS.ISP_first     = ISP_first;
CONSTANTS.ISP_second    = ISP_second;

rmin = -2*CONSTANTS.Re;
rmax = -rmin;
vmin = -10000;
vmax = -vmin;

iphase = 1;
limits(iphase).nodes = 15;
limits(iphase).time.min = [t0 t1];
limits(iphase).time.max = [t0 t1];
limits(iphase).state.min(1,:) = [r0(1) rmin rmin];
limits(iphase).state.max(1,:) = [r0(1) rmax rmax];
limits(iphase).state.min(2,:) = [r0(2) rmin rmin];
```

```
limits(iphase).state.max(2,:) = [r0(2) rmax rmax];
limits(iphase).state.min(3,:) = [r0(3) rmin rmin];
limits(iphase).state.max(3,:) = [r0(3) rmax rmax];
limits(iphase).state.min(4,:) = [v0(1) vmin vmin];
limits(iphase).state.max(4,:) = [v0(1) vmax vmax];
limits(iphase).state.min(5,:) = [v0(2) vmin vmin];
limits(iphase).state.max(5,:) = [v0(2) vmax vmax];
limits(iphase).state.min(6,:) = [v0(3) vmin vmin];
limits(iphase).state.max(6,:) = [v0(3) vmax vmax];
limits(iphase).state.min(7,:) = [m10 m1f m1f];
limits(iphase).state.max(7,:) = [m10 m10 m10];
limits(iphase).control.min(1,:) = -1;
limits(iphase).control.max(1,:) =  1;
limits(iphase).control.min(2,:) = -1;
limits(iphase).control.max(2,:) =  1;
limits(iphase).control.min(3,:) = -1;
limits(iphase).control.max(3,:) =  1;
limits(iphase).parameter.min    = [];
limits(iphase).parameter.max    = [];
limits(iphase).path.min    = 1;
limits(iphase).path.max    = 1;
guess(iphase).time = [t0; t1];
guess(iphase).state(:,1) = [r0(1); r0(1)];
guess(iphase).state(:,2) = [r0(2); r0(2)];
guess(iphase).state(:,3) = [r0(3); r0(3)];
guess(iphase).state(:,4) = [v0(1); v0(1)];
guess(iphase).state(:,5) = [v0(2); v0(2)];
guess(iphase).state(:,6) = [v0(3); v0(3)];
guess(iphase).state(:,7) = [m10; m1f];
guess(iphase).control(:,1) = [1; 1];
guess(iphase).control(:,2) = [0; 0];
guess(iphase).control(:,3) = [0; 0];
guess(iphase).parameter    = [];

iphase = 2;
limits(iphase).nodes = 15;
limits(iphase).time.min    = [t1 t2];
limits(iphase).time.max    = [t1 t2];
limits(iphase).state.min(1,:) = [rmin rmin rmin];
limits(iphase).state.max(1,:) = [rmax rmax rmax];
limits(iphase).state.min(2,:) = [rmin rmin rmin];
limits(iphase).state.max(2,:) = [rmax rmax rmax];
limits(iphase).state.min(3,:) = [rmin rmin rmin];
limits(iphase).state.max(3,:) = [rmax rmax rmax];
limits(iphase).state.min(4,:) = [vmin vmin vmin];
limits(iphase).state.max(4,:) = [vmax vmax vmax];
limits(iphase).state.min(5,:) = [vmin vmin vmin];
limits(iphase).state.max(5,:) = [vmax vmax vmax];
limits(iphase).state.min(6,:) = [vmin vmin vmin];
limits(iphase).state.max(6,:) = [vmax vmax vmax];
limits(iphase).state.min(7,:) = [m2f m2f m2f];
limits(iphase).state.max(7,:) = [m20  m20  m20];
limits(iphase).control.min(1,:) = -1;
limits(iphase).control.max(1,:) =  1;
limits(iphase).control.min(2,:) = -1;
limits(iphase).control.max(2,:) =  1;
limits(iphase).control.min(3,:) = -1;
limits(iphase).control.max(3,:) =  1;
limits(iphase).parameter.min    = [];
limits(iphase).parameter.max    = [];
limits(iphase).path.min    = 1;
limits(iphase).path.max    = 1;
guess(iphase).time = [t1; t2];
guess(iphase).state(:,1) = [r0(1); r0(1)];
guess(iphase).state(:,2) = [r0(2); r0(2)];
guess(iphase).state(:,3) = [r0(3); r0(3)];
guess(iphase).state(:,4) = [v0(1); v0(1)];
```

```
guess(iphase).state(:,5) = [v0(2); v0(2)];
guess(iphase).state(:,6) = [v0(3); v0(3)];
guess(iphase).state(:,7) = [m20; m2f];
guess(iphase).control(:,1) = [1; 1];
guess(iphase).control(:,2) = [0; 0];
guess(iphase).control(:,3) = [0; 0];
guess(iphase).parameter    = [];

iphase = 3;
limits(iphase).nodes = 15;
limits(iphase).time.min = [t2 t3];
limits(iphase).time.max = [t2 t3];
limits(iphase).state.min(1,:) = [rmin rmin rmin];
limits(iphase).state.max(1,:) = [rmax rmax rmax];
limits(iphase).state.min(2,:) = [rmin rmin rmin];
limits(iphase).state.max(2,:) = [rmax rmax rmax];
limits(iphase).state.min(3,:) = [rmin rmin rmin];
limits(iphase).state.max(3,:) = [rmax rmax rmax];
limits(iphase).state.min(4,:) = [vmin vmin vmin];
limits(iphase).state.max(4,:) = [vmax vmax vmax];
limits(iphase).state.min(5,:) = [vmin vmin vmin];
limits(iphase).state.max(5,:) = [vmax vmax vmax];
limits(iphase).state.min(6,:) = [vmin vmin vmin];
limits(iphase).state.max(6,:) = [vmax vmax vmax];
limits(iphase).state.min(7,:) = [m3f m3f m3f];
limits(iphase).state.max(7,:) = [m30  m30  m30];
limits(iphase).control.min(1,:) = -1;
limits(iphase).control.max(1,:) =  1;
limits(iphase).control.min(2,:) = -1;
limits(iphase).control.max(2,:) =  1;
limits(iphase).control.min(3,:) = -1;
limits(iphase).control.max(3,:) =  1;
limits(iphase).parameter.min    = [];
limits(iphase).parameter.max    = [];
limits(iphase).path.min    = 1;
limits(iphase).path.max    = 1;
guess(iphase).time = [t2; t3];
guess(iphase).state(:,1) = [rout(1); rout(1)];
guess(iphase).state(:,2) = [rout(2); rout(2)];
guess(iphase).state(:,3) = [rout(3); rout(3)];
guess(iphase).state(:,4) = [vout(1); vout(1)];
guess(iphase).state(:,5) = [vout(2); vout(2)];
guess(iphase).state(:,6) = [vout(3); vout(3)];
guess(iphase).state(:,7) = [m30; m3f];
guess(iphase).control(:,1) = [0; 0];
guess(iphase).control(:,2) = [0; 0];
guess(iphase).control(:,3) = [1; 1];
guess(iphase).parameter    = [];

iphase = 4;
limits(iphase).nodes = 15;
limits(iphase).time.min = [t3 t3];
limits(iphase).time.max = [t3 t4];
limits(iphase).state.min(1,:) = [rmin rmin rmin];
limits(iphase).state.max(1,:) = [rmax rmax rmax];
limits(iphase).state.min(2,:) = [rmin rmin rmin];
limits(iphase).state.max(2,:) = [rmax rmax rmax];
limits(iphase).state.min(3,:) = [rmin rmin rmin];
limits(iphase).state.max(3,:) = [rmax rmax rmax];
limits(iphase).state.min(4,:) = [vmin vmin vmin];
limits(iphase).state.max(4,:) = [vmax vmax vmax];
limits(iphase).state.min(5,:) = [vmin vmin vmin];
limits(iphase).state.max(5,:) = [vmax vmax vmax];
limits(iphase).state.min(6,:) = [vmin vmin vmin];
limits(iphase).state.max(6,:) = [vmax vmax vmax];
limits(iphase).state.min(7,:) = [m4f m4f m4f];
limits(iphase).state.max(7,:) = [m40 m40 m40];
```

```
limits(iphase).control.min(1,:) = -1;
limits(iphase).control.max(1,:) =  1;
limits(iphase).control.min(2,:) = -1;
limits(iphase).control.max(2,:) =  1;
limits(iphase).control.min(3,:) = -1;
limits(iphase).control.max(3,:) =  1;
limits(iphase).parameter.min = [];
limits(iphase).parameter.max = [];
limits(iphase).path.min      = 1;
limits(iphase).path.max      = 1;
limits(iphase).event.min     = [af; ef; incf; Omf; omf];
limits(iphase).event.max     = [af; ef; incf; Omf; omf];
guess(iphase).time    = [t3; t4];
guess(iphase).state(:,1) = [rout(1) rout(1)];
guess(iphase).state(:,2) = [rout(2) rout(2)];
guess(iphase).state(:,3) = [rout(3) rout(3)];
guess(iphase).state(:,4) = [vout(1) vout(1)];
guess(iphase).state(:,5) = [vout(2) vout(2)];
guess(iphase).state(:,6) = [vout(3) vout(3)];
guess(iphase).state(:,7) = [m40; m4f];
guess(iphase).control(:,1) = [0; 0];
guess(iphase).control(:,2) = [0; 0];
guess(iphase).control(:,3) = [1; 1];
guess(iphase).parameter    = [];

ipair = 1; % First pair of phases to connect
linkages(ipair).left.phase = 1;
linkages(ipair).right.phase = 2;
linkages(ipair).min = [0; 0; 0; 0; 0; 0; -6*m_dry_srb];
linkages(ipair).max = [0; 0; 0; 0; 0; 0; -6*m_dry_srb];

ipair = 2; % Second pair of phases to connect
linkages(ipair).left.phase = 2;
linkages(ipair).right.phase = 3;
linkages(ipair).min = [0; 0; 0; 0; 0; 0; -3*m_dry_srb];
linkages(ipair).max = [0; 0; 0; 0; 0; 0; -3*m_dry_srb];

ipair = 3; % Third pair of phases to connect
linkages(ipair).left.phase = 3;
linkages(ipair).right.phase = 4;
linkages(ipair).min = [0; 0; 0; 0; 0; 0; -m_dry_first];
linkages(ipair).max = [0; 0; 0; 0; 0; 0; -m_dry_first];

setup.autoscale = 'on';
setup.name = 'Launch-Vehicle-Ascent';
setup.funcs.cost = 'launchCost';
setup.funcs.dae = 'launchDae';
setup.funcs.event = 'launchEvent';
setup.funcs.link = 'launchConnect';
setup.derivatives = 'automatic';
% setup.derivatives = 'analytic';
setup.checkDerivatives = 1;
setup.direction = 'increasing';
setup.limits = limits;
setup.linkages = linkages;
setup.guess = guess;

if isequal(setup.derivatives,'automatic-intlab'),
    CONSTANTS.derivatives = 'automatic-intlab';
else
    CONSTANTS.derivatives = [];
end;

output = gpops(setup);
solution = output.solution;
```

```
function [Mayer,Lagrange, DMayer, DLagrange] = launchCost(sol);

global CONSTANTS

t0 = sol.initial.time;
x0 = sol.initial.state;
tf = sol.terminal.time;
xf = sol.terminal.state;
t = sol.time;
x = sol.state;
u = sol.control;
p = sol.parameter;

Lagrange = zeros(size(t));
if sol.phase==4,
    Mayer = -xf(7);
else
    Mayer = zeros(size(t0));
end;

% avoid calc of derivs in not necessary
if nargout == 4

    if sol.phase
        % DMayer = [           dM/dx0,              dM/dt0,          dM/dxf,
        DMayer = [zeros(1,length(x0)), zeros(1,length(t0)), [zeros(1,6) -1], ...
        ... %                  dM/dtf,              dM/dp]
                  zeros(1,length(tf)), zeros(1,length(p))];
    else
        % DMayer = [           dM/dx0,              dM/dt0,              dM/dxf,
        DMayer = [zeros(1,length(x0)), zeros(1,length(t0)), zeros(1,length(xf)), ...
        ... %                  dM/dtf,              dM/dp]
                  zeros(1,length(tf)), zeros(1,length(p))];
    end

    % DLagrange = [       dL/dx,          dL/du,                       dL/dp,           dL/dt]
    DLagrange =[ zeros(size(x)), zeros(size(u)),  zeros(length(t),length(p)), zeros(size(t))];

end


function [dae Ddae] = launchDae(sol);

global CONSTANTS;
t = sol.time;
x = sol.state;
u = sol.control;
p = sol.parameter;
iphase = sol.phase;
r = x(:,1:3);
v = x(:,4:6);
m = x(:,7);

rad = sqrt(sum(r.*r,2));
omega_matrix = CONSTANTS.omega_matrix;
vrel = v-r*omega_matrix.';
speedrel = sqrt(sum(vrel.*vrel,2));
if isequal(CONSTANTS.derivatives,'automatic-intlab'),
    % to eliminate divide by zero in INTLAB deriv calc
    speedrel(logical(speedrel == 0)) = 1;
end;
altitude = rad-CONSTANTS.Re;
rho = CONSTANTS.rho0*exp(-altitude/CONSTANTS.H);
bc  = (rho./(2*m)).*CONSTANTS.sa*CONSTANTS.cd;
bcspeed = bc.*speedrel;
% bcspeedmat = [bcspeed bcspeed bcspeed];
bcspeedmat = repmat(bcspeed,1,3);
```

```
Drag = -bcspeedmat.*vrel;
muoverradcubed = CONSTANTS.mu./rad.^3;
muoverradcubedmat = [muoverradcubed muoverradcubed muoverradcubed];
grav = -muoverradcubedmat.*r;

if iphase==1,
    T_srb = 6*CONSTANTS.thrust_srb*ones(size(t));
    T_first = CONSTANTS.thrust_first*ones(size(t));
    T_tot = T_srb+T_first;
    m1dot = -T_srb./(CONSTANTS.g0*CONSTANTS.ISP_srb);
    m2dot = -T_first./(CONSTANTS.g0*CONSTANTS.ISP_first);
    mdot = m1dot+m2dot;
elseif iphase==2,
    T_srb = 3*CONSTANTS.thrust_srb*ones(size(t));
    T_first = CONSTANTS.thrust_first*ones(size(t));
    T_tot = T_srb+T_first;
    m1dot = -T_srb./(CONSTANTS.g0*CONSTANTS.ISP_srb);
    m2dot = -T_first./(CONSTANTS.g0*CONSTANTS.ISP_first);
    mdot = m1dot+m2dot;
elseif iphase==3
    T_first = CONSTANTS.thrust_first*ones(size(t));
    T_tot = T_first;
    mdot = -T_first./(CONSTANTS.g0*CONSTANTS.ISP_first);
elseif iphase==4,
    T_second = CONSTANTS.thrust_second*ones(size(t));
    T_tot = T_second;
    mdot = -T_second./(CONSTANTS.g0*CONSTANTS.ISP_second);
end;

path = sum(u.*u,2);
Toverm = T_tot./m;
Tovermmat = [Toverm Toverm Toverm];
thrust = Tovermmat.*u;

rdot = v;
vdot = thrust+Drag+grav;

dae = [rdot vdot mdot path];

% avoid calc of derivs in not necessary
if nargout == 2

    % to eliminate divide by zero in analytic deriv calc
    speedrel(logical(speedrel == 0)) = 1;

    Ddae = zeros(8*length(t),11);
    N = length(t);  %number of nodes

    % drdot/dx
    Ddae(1:N,4) = 1;        % drdot1/dv1
    Ddae(N+1:2*N,5) = 1;    % drdot2/dv2
    Ddae(2*N+1:3*N,6) = 1;  % drdot3/dv3

    % dvdot/dx
    dDrag1_dr1 = bc.*vrel(:,2).*vrel(:,1)./speedrel*CONSTANTS.omega_matrix(2,1) ...
        + bc.*speedrel.*vrel(:,1).*r(:,1)./rad./CONSTANTS.H;
    dDrag1_dr2 = -bc.*vrel(:,1).*vrel(:,1)./speedrel*CONSTANTS.omega_matrix(2,1) ...
        + bc.*speedrel.*vrel(:,1)./CONSTANTS.H.*r(:,2)./rad ...
        - bc.*speedrel*CONSTANTS.omega_matrix(2,1);
    dDrag1_dr3 = bc.*speedrel.*vrel(:,1)./CONSTANTS.H.*r(:,3)./rad;
    dDrag2_dr1 = bc.*vrel(:,2).*vrel(:,2)./speedrel*CONSTANTS.omega_matrix(2,1) ...
        + bc.*speedrel.*vrel(:,2)./CONSTANTS.H.*r(:,1)./rad ...
        + bc.*speedrel*CONSTANTS.omega_matrix(2,1);
    dDrag2_dr2 = -bc.*vrel(:,1).*vrel(:,2)./speedrel*CONSTANTS.omega_matrix(2,1) ...
        + bc.*speedrel.*vrel(:,2)./CONSTANTS.H.*r(:,2)./rad;
    dDrag2_dr3 = bc.*speedrel.*vrel(:,2)./CONSTANTS.H.*r(:,3)./rad;
```

```
    dDrag3_dr1 = bc.*vrel(:,2).*vrel(:,3)./speedrel*CONSTANTS.omega_matrix(2,1) ...
        + bc.*speedrel.*vrel(:,3)./CONSTANTS.H.*r(:,1)./rad;
    dDrag3_dr2 = -bc.*vrel(:,1).*vrel(:,3)./speedrel*CONSTANTS.omega_matrix(2,1) ...
        + bc.*speedrel.*vrel(:,3)./CONSTANTS.H.*r(:,2)./rad;
    dDrag3_dr3 = bc.*speedrel.*vrel(:,3)./CONSTANTS.H.*r(:,3)./rad;

    dgrav1_dr1 = -muoverradcubed + 3*CONSTANTS.mu.*r(:,1).^2./rad.^5;
    dgrav1_dr2 = 3*CONSTANTS.mu.*r(:,1).*r(:,2)./rad.^5;
    dgrav1_dr3 = 3*CONSTANTS.mu.*r(:,1).*r(:,3)./rad.^5;
    dgrav2_dr1 = 3*CONSTANTS.mu.*r(:,2).*r(:,1)./rad.^5;
    dgrav2_dr2 = -muoverradcubed + 3*CONSTANTS.mu.*r(:,2).^2./rad.^5;
    dgrav2_dr3 = 3*CONSTANTS.mu.*r(:,2).*r(:,3)./rad.^5;
    dgrav3_dr1 = 3*CONSTANTS.mu.*r(:,3).*r(:,1)./rad.^5;
    dgrav3_dr2 = 3*CONSTANTS.mu.*r(:,3).*r(:,2)./rad.^5;
    dgrav3_dr3 = -muoverradcubed + 3*CONSTANTS.mu.*r(:,3).^2./rad.^5;

    Ddae(3*N+1:4*N,1) = dDrag1_dr1 + dgrav1_dr1;    % dvdot1/dr1
    Ddae(3*N+1:4*N,2) = dDrag1_dr2 + dgrav1_dr2;    % dvdot1/dr2
    Ddae(3*N+1:4*N,3) = dDrag1_dr3 + dgrav1_dr3;    % dvdot1/dr3
    Ddae(4*N+1:5*N,1) = dDrag2_dr1 + dgrav2_dr1;    % dvdot2/dr1
    Ddae(4*N+1:5*N,2) = dDrag2_dr2 + dgrav2_dr2;    % dvdot2/dr2
    Ddae(4*N+1:5*N,3) = dDrag2_dr3 + dgrav2_dr3;    % dvdot2/dr3
    Ddae(5*N+1:6*N,1) = dDrag3_dr1 + dgrav3_dr1;    % dvdot3/dr1
    Ddae(5*N+1:6*N,2) = dDrag3_dr2 + dgrav3_dr2;    % dvdot3/dr2
    Ddae(5*N+1:6*N,3) = dDrag3_dr3 + dgrav3_dr3;    % dvdot3/dr3

    dDrag1_dv1 = -bc.*speedrel - bc.*vrel(:,1).^2./speedrel;
    dDrag1_dv2 = -bc.*vrel(:,1).*vrel(:,2)./speedrel;
    dDrag1_dv3 = -bc.*vrel(:,1).*vrel(:,3)./speedrel;
    dDrag2_dv1 = -bc.*vrel(:,2).*vrel(:,1)./speedrel;
    dDrag2_dv2 = -bc.*speedrel - bc.*vrel(:,2).^2./speedrel;
    dDrag2_dv3 = -bc.*vrel(:,2).*vrel(:,3)./speedrel;
    dDrag3_dv1 = -bc.*vrel(:,3).*vrel(:,1)./speedrel;
    dDrag3_dv2 = -bc.*vrel(:,3).*vrel(:,2)./speedrel;
    dDrag3_dv3 = -bc.*speedrel - bc.*vrel(:,3).^2./speedrel;

    Ddae(3*N+1:4*N,4) = dDrag1_dv1;    % dvdot1/dv1
    Ddae(3*N+1:4*N,5) = dDrag1_dv2;    % dvdot1/dv2
    Ddae(3*N+1:4*N,6) = dDrag1_dv3;    % dvdot1/dv3
    Ddae(4*N+1:5*N,4) = dDrag2_dv1;    % dvdot2/dv1
    Ddae(4*N+1:5*N,5) = dDrag2_dv2;    % dvdot2/dv2
    Ddae(4*N+1:5*N,6) = dDrag2_dv3;    % dvdot2/dv3
    Ddae(5*N+1:6*N,4) = dDrag3_dv1;    % dvdot3/dv1
    Ddae(5*N+1:6*N,5) = dDrag3_dv2;    % dvdot3/dv2
    Ddae(5*N+1:6*N,6) = dDrag3_dv3;    % dvdot3/dv3

    dDrag1_dm = -Drag(:,1)./m;
    dDrag2_dm = -Drag(:,2)./m;
    dDrag3_dm = -Drag(:,3)./m;

    Ddae(3*N+1:4*N,7) = dDrag1_dm - thrust(:,1)./m; % dvdot1/dm
    Ddae(4*N+1:5*N,7) = dDrag2_dm - thrust(:,2)./m; % dvdot2/dm
    Ddae(5*N+1:6*N,7) = dDrag3_dm - thrust(:,3)./m; % dvdot3/dm

    %dvdot/du
    Ddae(3*N+1:4*N,8) = Toverm;  % dvdot1/du1
    Ddae(4*N+1:5*N,9) = Toverm;  % dvdot2/du2
    Ddae(5*N+1:6*N,10) = Toverm; % dvdot3/du3

    % mass dynamics independant of State
    % Ddae(6*N+1:7*N,:) = 0

    %dpath/du
    Ddae(7*N+1:8*N,8) = 2*u(:,1);  % dp/du1
    Ddae(7*N+1:8*N,9) = 2*u(:,2);  % dp/du2
    Ddae(7*N+1:8*N,10) = 2*u(:,3); % dp/du3
end
```

```
function [event Devent] = launchEvent(sol);

global CONSTANTS
t0 = sol.initial.time;
x0 = sol.initial.state;
tf = sol.terminal.time;
xf = sol.terminal.state;
p  = sol.parameter;
iphase = sol.phase;

if iphase==4,
    oe = launchrv2oe(xf(1:3),xf(4:6),CONSTANTS.mu);
    event = oe(1:5);
else
    event = [];
end;

% avoid calc of derivs in not necessary
if nargout == 2

    if iphase == 4
        Doe = launchrv2oe_D(xf(1:3),xf(4:6),CONSTANTS.mu,CONSTANTS.Re);

        % Devents = [dE/dx0, dE/dt0, dE/dxf, dE/dtf, dE/dp]
        lx0 = length(x0);
        lp  = length(p);
        Devent = [zeros(5,lx0), zeros(5,1), [Doe, zeros(5,1)], zeros(5,1), zeros(5,lp)];
    else
        Devent = [];
    end
end


function oe = launchrv2oe(rv,vv,mu);

K  = [0;0;1];
hv = cross(rv,vv);
nv = cross(K,hv);
n  = sqrt(nv.'*nv);
h2 = (hv.'*hv);
v2 = (vv.'*vv);
r  = sqrt(rv.'*rv);
ev = 1/mu *( (v2-mu/r)*rv - (rv.'*vv)*vv );
p  = h2/mu;
%
% now compute the oe's
%
e  = sqrt(ev.'*ev); % eccentricity
a  = p/(1-e*e); % semimajor axis
i  = acos(hv(3)/sqrt(h2)); % inclination
Om = acos(nv(1)/n); % RAAN
if ( nv(2) < 0-eps ) % fix quadrant
    Om = 2*pi-Om;
end;
om = acos(nv.'*ev/n/e); % arg of periapsis
if ( ev(3) < 0 ) % fix quadrant
    om = 2*pi-om;
end;
nu = acos(ev.'*rv/e/r); % true anomaly
if ( rv.'*vv < 0 ) % fix quadrant
    nu = 2*pi-nu;
end;
oe = [a; e; i; Om; om; nu]; % assemble "vector"


function [ri,vi] = launchoe2rv(oe,mu)
```

```
a=oe(1); e=oe(2); i=oe(3); Om=oe(4); om=oe(5); nu=oe(6);
p = a*(1-e*e);
r = p/(1+e*cos(nu));
rv = [r*cos(nu); r*sin(nu); 0];
vv = sqrt(mu/p)*[-sin(nu); e+cos(nu); 0];
cO = cos(Om);  sO = sin(Om);
co = cos(om);  so = sin(om);
ci = cos(i);   si = sin(i);
R  = [cO*co-sO*so*ci  -cO*so-sO*co*ci  sO*si;
  sO*co+cO*so*ci  -sO*so+cO*co*ci -cO*si;
  so*si            co*si           ci];
ri = R*rv;
vi = R*vv;
```

The output of the above code from $\mathcal{GPOPS}$ is summarized in the following three plots that contain the altitude, speed, and controls.

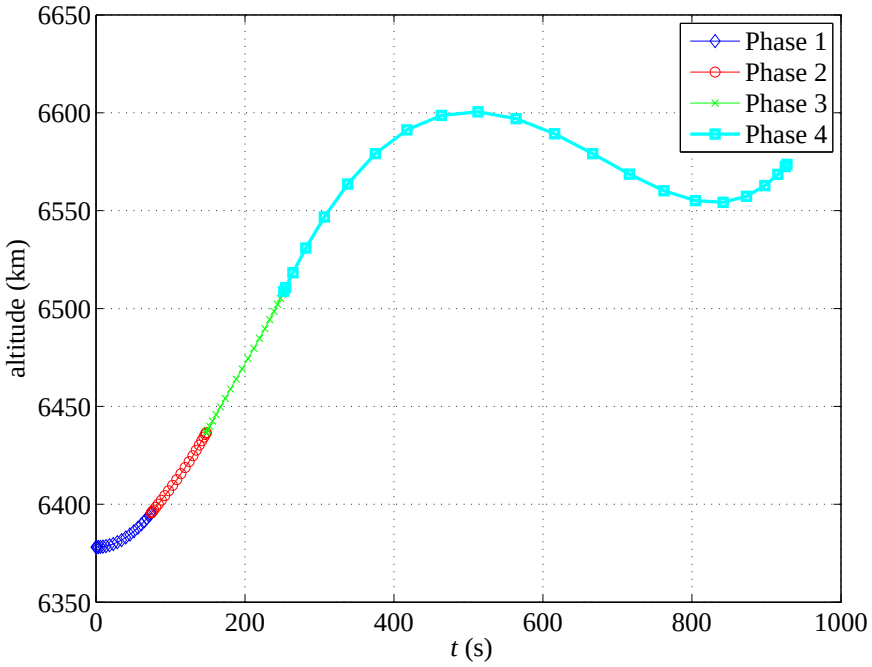

Figure 3.11 Altitude vs. time for the launch vehicle ascent problem.

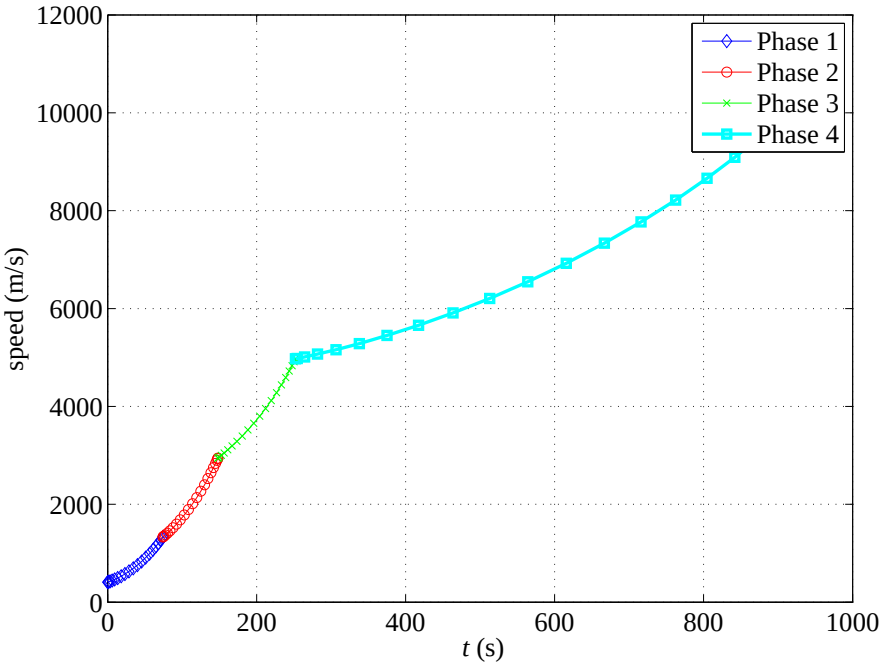

Figure 3.12 Inertial speed vs. time for the launch vehicle ascent problem.

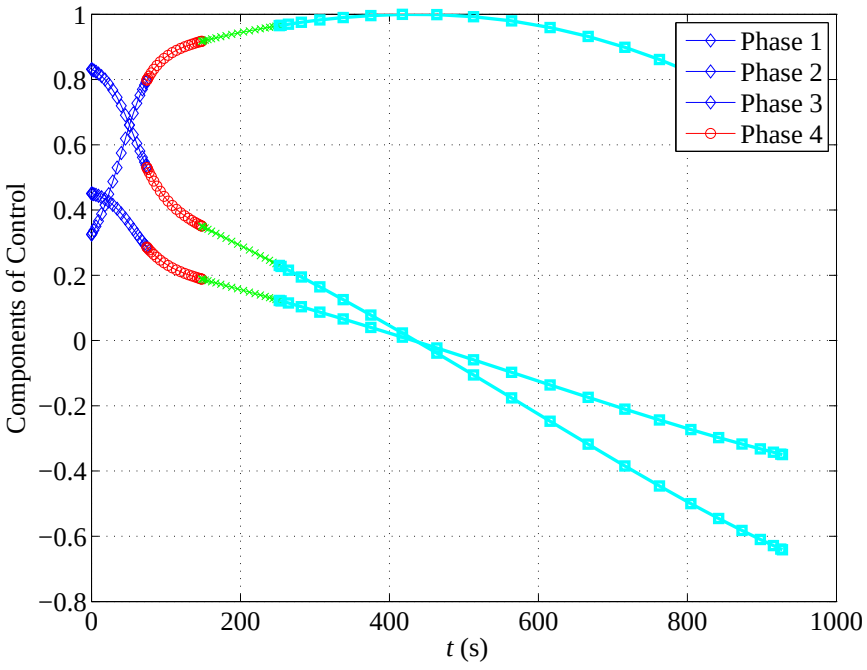

Figure 3.13 Controls vs. time for the launch vehicle ascent problem.

3.4 Minimum Time-to-Climb of a Supersonic Aircraft

The problem considered in this section is the classical minimum time-to-climb of a supersonic aircraft. The objective is to determine the minimum-time trajectory and control from take-off to a specified altitude and

speed. This problem was originally stated in the open literature in the work of Bryson, et al. (1969), but the model used in this study was taken from Betts (2001) with the exception that a linear extrapolation of the thrust data as found in Betts (2001) was performed in order to fill in the "missing" data points.

The minimum time-to-climb problem for a supersonic aircraft is posed as follows. Minimize the cost functional

$$J = t_f \tag{3–18}$$

subject to the dynamic constraints

$$\dot{E} = \frac{v(T-D)}{mg} \tag{3–19}$$

$$\dot{h} = v\sin\gamma \tag{3–20}$$

$$\dot{\gamma} = \frac{g}{v}\left[n - \cos\gamma\right] \tag{3–21}$$

$$\tag{3–22}$$

and the boundary conditions

$$h(0) = 0 \text{ ft} \tag{3–23}$$

$$v(0) = 129.3144 \text{ m/s} \tag{3–24}$$

$$\gamma(0) = 0 \text{ rad} \tag{3–25}$$

$$h(t_f) = 19995 \text{ m} \tag{3–26}$$

$$v(t_f) = 295.09 \text{ ft/s} \tag{3–27}$$

$$\gamma(t_f) = 0 \text{ rad} \tag{3–28}$$

where E is the the energy altitude, h is the altitude, γ is the flight path angle, m is the vehicle mass, n is the load factor, T is the magnitude of the thrust force, and D is the magnitude of the drag force.

The MATLAB code that solves the minimum time-to-climb of a supersonic aircraft is shown below.

```
% Minimum Time-to-Climb Problem
% The vehicle model for this problem is taken
% from the following two references:
% Seywald, H., Clifs, E. M., and Well, K. H.,
% "Range Optimal Trajectories for an Aircraft Flying in
%  the Vertical Plane," Journal of Guidance, Control, and Dynamics,
%  Vol. 17, No. 2, March-April, 1994.
%
%  Rao, A. V., Extension of a Computational Singular Perturbation
%  Methodology to Optimal Control Problems, Ph.D. Thesis, Dept. of
%  Mechanical and Aerospace Engineering, Princeton University,
%  June 1996.

clear setup limits guess CONSTANTS
global CONSTANTS
CoF(1,:) = [2.61059846050e-2;
            -8.57043966269e-2;
            1.07863115049e-1;
            -6.44772018636e-2;
            1.64933626507e-2;
            0];
CoF(2,:) = [1.37368651246e0;
            -4.57116286752e0;
            5.72789877344e0;
            -3.25219000620e0;
            7.29821847445e-1;
            0];
CoF(3,:) = [1.23001735612e0;
            -2.97244144190e0;
            2.78009092756e0;
```

```
            -1.16227834301e0;
            1.81868987624e-1;
            0];
CoF(4,:) = [1.42392902737e1;
            -3.24759126471e1;
            2.96838643792e1;
            -1.33316812491e1;
            2.87165882405e0;
            -2.27239723756e-1];
CoF(5,:) = [0.11969995703e6;
            -0.14644656421e5;
            -0.45534597613e3;
            0.49544694509e3;
            -0.46253181596e2;
            0.12000480258e1];
CoF(6,:) = [-0.35217318620e6;
            0.51808811078e5;
            0.23143969006e4;
            -0.22482310455e4;
            0.20894683419e3;
            -0.53807416658e1];
CoF(7,:) = [0.60452159152e6;
            -0.95597112936e5;
            -0.38860323817e4;
            0.39771922607e4;
            -0.36835984294e3;
            0.94529288471e1];
CoF(8,:) = [-0.43042985701e6;
            0.83271826575e5;
            0.12357128390e4;
            -0.30734191752e4;
            0.29388870979e3;
            -0.76204728620e1];
CoF(9,:) = [0.13656937908e6;
            -0.32867923740e5;
            0.55572727442e3;
            0.10635494768e4;
            -0.10784916936e3;
            0.28552696781e1];
CoF(10,:) = [-0.16647992124e5;
            0.49102536402e4;
            -0.23591380327e3;
            -0.13626703723e3;
            0.14880019422e2;
            -0.40379767869e0];
CoFZ = [-3.48643241e-2;
        3.50991865e-3;
        -8.33000535e-5;
        1.15219733e-6];

CONSTANTS.CoF = CoF;
CONSTANTS.CoFZ = CoFZ;

g = 9.80665;
m = 37000/2.2;
feettometer = .3048;

h0 = 0*feettometer;
hf = 65600*feettometer;
v0 = 424.26*feettometer;
vf = 968.148*feettometer;
e0 = (v0^2/(2*g)+h0);
ef = (vf^2/(2*g)+hf);
fpa0 = 0;
fpaf = 0;

hmin =  0*feettometer;
```

```
hmax =  69000*feettometer;
vmin =  1*feettometer;
vmax =  2000*feettometer;
emin = (vmin^2/(2*g)+hmin);
emax = (vmax^2/(2*g)+hmax);
fpamin = -40/180*pi;
fpamax = -fpamin;
umin = -10;
umax =  10;
t0min = 0;
t0max = 0;
tfmin = 100;
tfmax = 350;

% Phase 1 Information
iphase = 1;
limits(iphase).nodes = 50;
limits(iphase).time.min = [t0min tfmin];
limits(iphase).time.max = [t0max tfmax];
limits(iphase).state.min(1,:) = [h0 hmin hf];
limits(iphase).state.max(1,:) = [h0 hmax hf];
limits(iphase).state.min(2,:) = [e0 emin ef];
limits(iphase).state.max(2,:) = [e0 emax ef];
limits(iphase).state.min(3,:) = [fpa0 fpamin fpaf];
limits(iphase).state.max(3,:) = [fpa0 fpamax fpaf];
limits(iphase).control.min = umin;
limits(iphase).control.max = umax;
limits(iphase).parameter.min = [];
limits(iphase).parameter.max = [];
limits(iphase).path.min = [];
limits(iphase).path.max = [];
limits(iphase).event.min = [];
limits(iphase).event.max = [];
limits(iphase).duration.min = [];
limits(iphase).duration.max = [];
guess(iphase).time = [t0min; tfmax];
guess(iphase).state(:,1) = [hf/2; hf/2];
guess(iphase).state(:,2) = [ef/2; ef/2];
guess(iphase).state(:,3) = [fpamax/2; fpamax/2];
guess(iphase).control = [0; 0];
guess(iphase).parameter = []; % No parameters in Phase 1

setup.name  = 'Minimum-Time-to-Climb-Problem';
setup.funcs.cost = 'minimumClimbCost';
setup.funcs.dae = 'minimumClimbDae';
setup.funcs.link = '';
setup.limits = limits;
setup.guess = guess;
setup.derivatives = 'automatic';
setup.direction = 'increasing';
setup.autoscale = 'on';

output = gpops(setup);

solution = output.solution;


function [Mayer,Lagrange]=minimumClimbCost(solcost);

t0 = solcost.initial.time;
x0 = solcost.initial.state;
tf = solcost.terminal.time;
xf = solcost.terminal.state;
t  = solcost.time;
x  = solcost.state;
u  = solcost.control;
p  = solcost.parameter;
```

```
Mayer = tf;
Lagrange = zeros(size(t));

function daeout = minimumClimbDae(soldae);

global CONSTANTS

CoF = CONSTANTS.CoF;
CoFZ = CONSTANTS.CoFZ;

t = soldae.time;
x = soldae.state;
u = soldae.control;
p = soldae.parameter;
h = x(:,1);
E = x(:,2);
fpa = x(:,3);

g = 9.80665;
m = 37000./2.2;
S = 60;
hbar = h./1000;

%rho calculation
z = CoFZ(1).*hbar+CoFZ(2).*hbar.^2+CoFZ(3).*hbar.^3+CoFZ(4).*hbar.^4;
r = 1.0228066.*exp(-z);
y = -0.12122693.*hbar+r-1.0228055;
rho = 1.225.*exp(y);
%rho calculation

%speed of sound%
theta = 292.1-8.87743.*hbar+0.193315.*hbar.^2+(3.72e-3).*hbar.^3;
a     = 20.0468.*sqrt(theta);
%speed of sound%

%Velocity and mach
Eminush = (E-h).^2;
Eminush = sqrt(Eminush);
v = sqrt(2.*g.*Eminush);

M = v./a;
%Velocity and mach

%Who are lift and drag
q = 0.5.*rho.*v.*v.*S;
L = m.*g.*u;
M0 = M.^0;
M1 = M.^1;
M2 = M.^2;
M3 = M.^3;
M4 = M.^4;
M5 = M.^5;
numeratorCD0 = CoF(1,1).*M0+CoF(1,2).*M1+CoF(1,3).*M2+CoF(1,4).*M3+CoF(1,5).*M4;
denominatorCD0 = CoF(2,1).*M0+CoF(2,2).*M1+CoF(2,3).*M2+CoF(2,4).*M3+CoF(2,5).*M4;
Cd0 = numeratorCD0./denominatorCD0;
numeratorK = CoF(3,1).*M0+CoF(3,2).*M1+CoF(3,3).*M2+CoF(3,4).*M3+CoF(3,5).*M4;
denominatorK = CoF(4,1).*M0+CoF(4,2).*M1+CoF(4,3).*M2+CoF(4,4).*M3+CoF(4,5).*M4+CoF(4,6).*M5;
K   = numeratorK./denominatorK;
D = q.*(Cd0+K.*((m.^2).*(g.^2)./(q.^2)).*(u.^2));
%Who are lift and drag

%Who is thrust
e0 = CoF(5,1).*M0+CoF(6,1).*M1+CoF(7,1).*M2+CoF(8,1).*M3+CoF(9,1).*M4+CoF(10,1).*M5;
e1 = CoF(5,2).*M0+CoF(6,2).*M1+CoF(7,2).*M2+CoF(8,2).*M3+CoF(9,2).*M4+CoF(10,2).*M5;
e2 = CoF(5,3).*M0+CoF(6,3).*M1+CoF(7,3).*M2+CoF(8,3).*M3+CoF(9,3).*M4+CoF(10,3).*M5;
e3 = CoF(5,4).*M0+CoF(6,4).*M1+CoF(7,4).*M2+CoF(8,4).*M3+CoF(9,4).*M4+CoF(10,4).*M5;
```

```
e4 = CoF(5,5).*M0+CoF(6,5).*M1+CoF(7,5).*M2+CoF(8,5).*M3+CoF(9,5).*M4+CoF(10,5).*M5;
e5 = CoF(5,6).*M0+CoF(6,6).*M1+CoF(7,6).*M2+CoF(8,6).*M3+CoF(9,6).*M4+CoF(10,6).*M5;

T = (e0.*hbar.^0+e1.*hbar.^1+e2.*hbar.^2+e3.*hbar.^3+e4.*hbar.^4+e5.*hbar.^5).*9.80665./2.2;
%Who is thrust

%***********************************************

%User Input - Give me f in dot(x) = f
%*********************

Edot = v./(m.*g).*(T-D);
hdot = v.*sin(fpa);
fpadot = g./v.*(L./(m.*g)-cos(fpa));


daeout = [hdot Edot fpadot];


function daeout = minimumClimbDae(soldae);

global CONSTANTS

CoF = CONSTANTS.CoF;
CoFZ = CONSTANTS.CoFZ;

t = soldae.time;
x = soldae.state;
u = soldae.control;
p = soldae.parameter;
h = x(:,1);
E = x(:,2);
fpa = x(:,3);

g = 9.80665;
m = 37000./2.2;
S = 60;
hbar = h./1000;

%rho calculation
z = CoFZ(1).*hbar+CoFZ(2).*hbar.^2+CoFZ(3).*hbar.^3+CoFZ(4).*hbar.^4;
r = 1.0228066.*exp(-z);
y = -0.12122693.*hbar+r-1.0228055;
rho = 1.225.*exp(y);
%rho calculation

%speed of sound%
theta = 292.1-8.87743.*hbar+0.193315.*hbar.^2+(3.72e-3).*hbar.^3;
a     = 20.0468.*sqrt(theta);
%speed of sound%

%Velocity and mach
Eminush = (E-h).^2;
Eminush = sqrt(Eminush);
v = sqrt(2.*g.*Eminush);

M = v./a;
%Velocity and mach

%Who are lift and drag
q = 0.5.*rho.*v.*v.*S;
L = m.*g.*u;
M0 = M.^0;
M1 = M.^1;
M2 = M.^2;
M3 = M.^3;
M4 = M.^4;
```

```
M5 = M.^5;
numeratorCD0 = CoF(1,1).*M0+CoF(1,2).*M1+CoF(1,3).*M2+CoF(1,4).*M3+CoF(1,5).*M4;
denominatorCD0 = CoF(2,1).*M0+CoF(2,2).*M1+CoF(2,3).*M2+CoF(2,4).*M3+CoF(2,5).*M4;
Cd0 = numeratorCD0./denominatorCD0;
numeratorK = CoF(3,1).*M0+CoF(3,2).*M1+CoF(3,3).*M2+CoF(3,4).*M3+CoF(3,5).*M4;
denominatorK = CoF(4,1).*M0+CoF(4,2).*M1+CoF(4,3).*M2+CoF(4,4).*M3+CoF(4,5).*M4+CoF(4,6).*M5;
K   = numeratorK./denominatorK;
D = q.*(Cd0+K.*((m.^2).*(g.^2)./(q.^2)).*(u.^2));
%Who are lift and drag

%Who is thrust
e0 = CoF(5,1).*M0+CoF(6,1).*M1+CoF(7,1).*M2+CoF(8,1).*M3+CoF(9,1).*M4+CoF(10,1).*M5;
e1 = CoF(5,2).*M0+CoF(6,2).*M1+CoF(7,2).*M2+CoF(8,2).*M3+CoF(9,2).*M4+CoF(10,2).*M5;
e2 = CoF(5,3).*M0+CoF(6,3).*M1+CoF(7,3).*M2+CoF(8,3).*M3+CoF(9,3).*M4+CoF(10,3).*M5;
e3 = CoF(5,4).*M0+CoF(6,4).*M1+CoF(7,4).*M2+CoF(8,4).*M3+CoF(9,4).*M4+CoF(10,4).*M5;
e4 = CoF(5,5).*M0+CoF(6,5).*M1+CoF(7,5).*M2+CoF(8,5).*M3+CoF(9,5).*M4+CoF(10,5).*M5;
e5 = CoF(5,6).*M0+CoF(6,6).*M1+CoF(7,6).*M2+CoF(8,6).*M3+CoF(9,6).*M4+CoF(10,6).*M5;

T = (e0.*hbar.^0+e1.*hbar.^1+e2.*hbar.^2+e3.*hbar.^3+e4.*hbar.^4+e5.*hbar.^5).*9.80665./2.2;
%Who is thrust

%*********************************************

%User Input - Give me f in dot(x) = f
%*********************

Edot = v./(m.*g).*(T-D);
hdot = v.*sin(fpa);
fpadot = g./v.*(L./(m.*g)-cos(fpa));


daeout = [hdot Edot fpadot];
```

The output of the above code from $\mathcal{GPOPS}$ is summarized in the following three plots that contain the altitude, speed, flight path angle, and angle of attack:

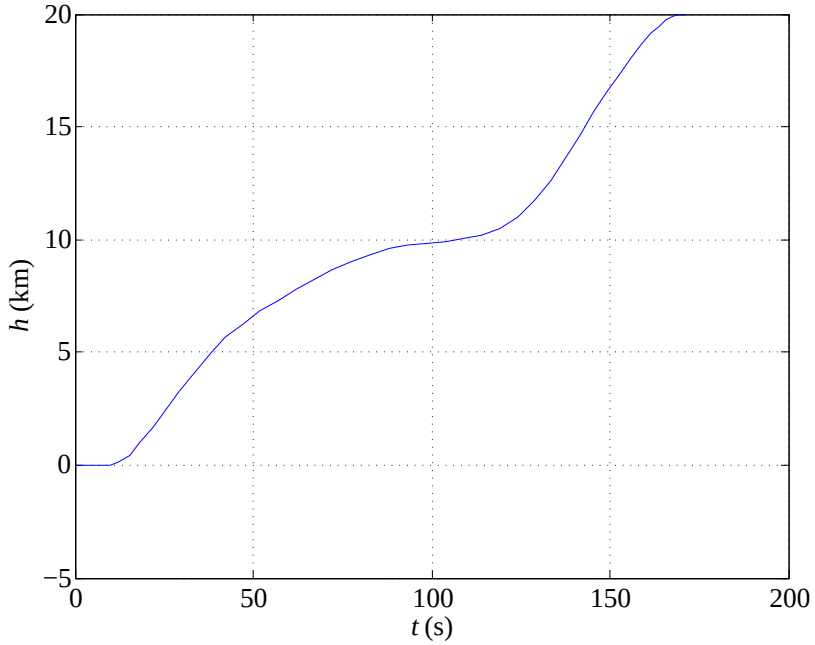

Figure 3.14 Altitude vs. Time for supersonic aircraft minimum time-to-climb.

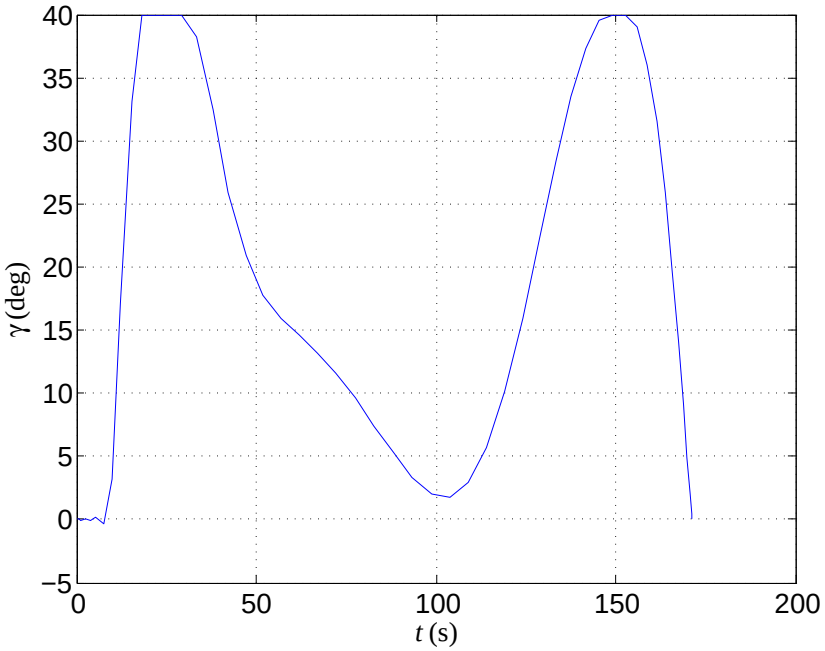

Figure 3.15 Flight path angle vs. Time for supersonic aircraft minimum time-to-climb.

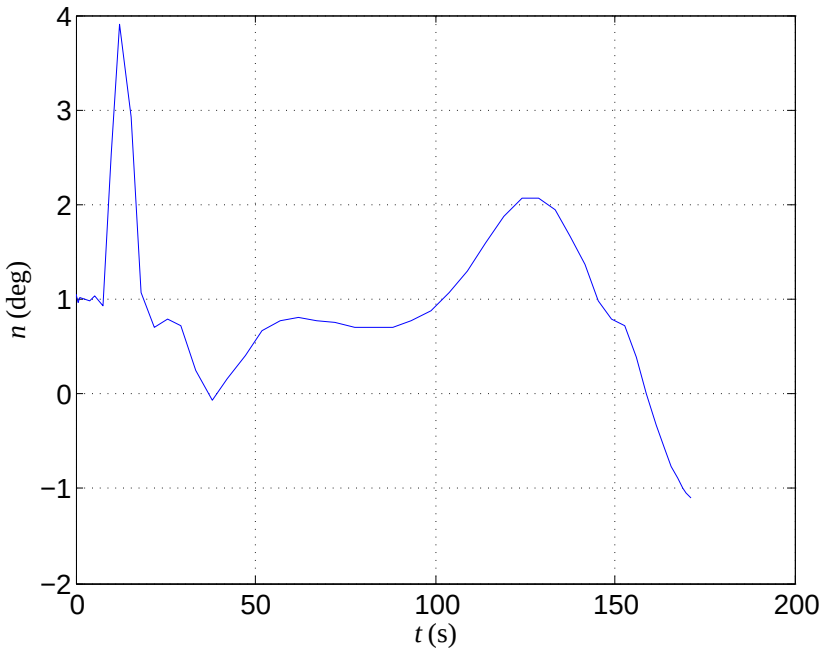

Figure 3.16 Angle of attack vs. Time for supersonic aircraft minimum time-to-climb.

3.5 Some Concluding Remarks

While $\mathcal{GPOPS}$ has been designed to take some of the cumbersomeness out of solving an optimal control problem numerically, the user must still be wary of several aspects of computational optimal control that will make it easier to use $\mathcal{GPOPS}$. First, as noted earlier, it is *highly* recommended that the user scale the problem manually because the automatic scaling procedure is by no means foolproof. Second, the choice of variables to solve an optimal control problem can make all the difference in the world as to how quickly and reliably a solution is obtained. For example, atmospheric flight problems with large lift maneuvers tend to be easier to solve if spherical coordinates (where position is parameterized using radius, longitude, and latitude, while velocity is parameterized using speed, flight path angle, and heading angle) are used as compared to Cartesian coordinates whereas launch vehicle ascent problems (which have no lift) tend to be better parameterized using Cartesian coordinates. Finally, even if the optimizer returns the result that the optimality conditions have been satisfied, it is extremely important to analyze the solution to make sure that (1) the solution is the one corresponding to the problem that the user wants to solve and (2) if the solution makes sense. In short, a great deal of time in solving optimal control problems is spent in formulation and analysis.

References

Bate, R. R., Mueller, D. D., and White, J. E., *Fundamentals of Astrodynamics*, Dover Publications, New York, 1971.

Benson, D. A., *A Gauss Pseudospectral Transcription for Optimal Control*, Ph.D. Dissertation, Department of Aeronautics and Astronautics, MIT, November 2004.

Benson, D. A., Huntington, G. T., Thorvaldsen, T. P., and Rao, A. V., "Direct Trajectory Optimization and Costate Estimation via an Orthogonal Collocation Method,", *Journal of Guidance, Control, and Dynamics*, Vol. 29, No. 6, November-December, 2006, pp. 1435–1440.

Betts, J. T., *Practical Methods for Optimal Control Using Nonlinear Programming*, SIAM Press, Philadelphia, 2001.

Bryson, A. E., Denham, W. F., and Dreyfus, S. E., "Optimal Programming Problems with Inequality Constraints. I: Necessary Conditions for Extremal Solutions", *AIAA Journal*, Vol. 1, No. 11, November 1963, pp. 2544-2550.

Bryson, A. E., Desai, M. N., and Hoffman, W. C., "Energy-State Approximation in Performance Optimization of Supersonic Aircraft," *Journal of Aircraft*, Vol. 6, No. 6, 1969, pp. 481–488.

Davis, P., *Interpolation and Approximation*, Dover Publications, 1975.

Elnagar, G., Kazemi, M., Razzaghi, M., "The Pseudospectral Legendre Method for Discretizing Optimal Control Problems," *IEEE Transactions on Automatic Control* Vol. 40, No. 10, October 1995.

Elnagar, G. and Kazemi, M., "Pseudospectral Chebyshev Optimal Control of Constrained Nonlinear Dynamical Systems," *Computational Optimization and Applications*, Vol. 11, 1998, pp. 195-217.

Gill, P. E., Murray, W., and Saunders, M. A., "User's Guide for SNOPT Version 7: Software for Large-Scale Nonlinear Programming," Report, University of California, San Diego, 24 April 2007.

Huntington, G. T. and Rao, A. V., "Optimal Spacecraft Formation Configuration Using a Gauss Pseudospectral Method," *Proceedings of the 2005 AAS/AISS Spaceflight Mechanics Meeting*, AAS Paper 05-103, Copper Mountain, Colorado, January 23–27, 2005.

Huntington, G. T., Benson, D. A., and Rao, A. V., "Post-Optimality Evaluation and Analysis of a Formation Flying Problem via a Gauss Pseudospectral Method," *Proceedings of the 2005 AAS/AIAA Astrodynamics Specialist Conference*, AAS Paper 05-339, Lake Tahoe, California, August 7–11, 2005.

Huntington, G. T. and Rao, A. V., "Optimal Reconfiguration of a Spacecraft Formation via a Gauss Pseudospectral Method," *Proceedings of the 2005 AAS/AIAA Astrodynamics Specialist Conference*, AAS Paper 05-338, Lake Tahoe, California, August 7–11, 2005.

Huntington, G. T., *Advancement and Analysis of a Gauss Pseudospectral Transcription for Optimal Control*, Ph.D. Dissertation, Department of Aeronautics and Astronautics, MIT, May 2007.

Huntington, G. T. and Rao, A. V., "Design of Optimal Tetrahedral Spacecraft Formations," *The Journal of the Astronautical Sciences*, to appear in 2007.

Huntington, G. T., Benson, D. A., How, J. P., Kanizay, N., Darby, C. L., and Rao, A. V., "Computation of Boundary Controls Using a Gauss Pseudospectral Method," *2007 Astrodynamics Specialist Conference*, Mackinac Island, Michigan, AAS Paper 07-381, August 21–24, 2007.

Kirk, D. E., *Optimal Control Theory*, Dover Publications, 1970.

Martins, J. R. R., Sturdza, P., and Alonso, J. J., "The Complex-Step Derivative Approximation," *ACM Transactions on Mathematical Software*, Vol. 29, No. 3, 2003, pp. 245–262.

Pontryagin, L.S., Boltyanskii, V., Gamkrelidze, R., Mischenko, E., *The Mathematical Theory of Optimal Processes*, New York: Interscience, 1962.

Rao, A. V. and Mease, K. D., "Eigenvector Approximate Dichotomic Basis Method for Solving Hyper-Sensitive Optimal Control Problems," *Optimal Control Applications and Methods*, Vol. 21, No. 1, 2000, pp. 1–19.