# User's Manual for *GPOPS* Version 5.0:

# A MATLAB® Software for Solving Multiple-Phase Optimal Control Problems Using *hp*–Adaptive Pseudospectral Methods

Anil V. Rao
*University of Florida*
Gainesville, FL 32607

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

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

Geoffrey T. Huntington
*Blue Origin, LLC*
Seattle, WA

August 2011

## Acknowledgments

The software *GPOPS* was developed in response to a demand from the research and academic community for a MATLAB software for solving complex optimal control problems. Since the original release of *GPOPS* in the Fall of 2008, the methods and the software have undergone extensive changes. Originally the software utilized the Gauss pseudospectral method, but more research in the area of pseudospectral methods for solving optimal control has led us to the current version of the software that implements the Radau pseudospectral method. In addition, we now offer a code that implements an *hp*–adaptive mesh refinement algorithm that iteratively determines a mesh that accurately distributes the collocation points. The bulk of the changes to *GPOPS* are internal, that is, the user-interface has changed only slightly from earlier versions of the code. The authors of *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 *GPOPS* Software

It is noted that *GPOPS* has been designed to work with the nonlinear programming solver SNOPT.[1] The current version of *GPOPS* now includes a restricted version of SNOPT. Next, *GPOPS* has been re-written so that now the objective function and constraint Jacobian derivatives can be estimated using built-in finite-differencing, sparse complex-step differentiation, or forward mode automatic differentiation. In addition, *GPOPS* still retains the ability to use the forward mode automatic differentiator *INTLAB*. It is noted that *INTLAB* can be downloaded from http://www.ti3.tu-harburg.de/rump/intlab/. Commercial use of *INTLAB* requires a license which can be obtained by contacting Professor Siegfried Rump via e-mail at rumptu-harburg.de.

## Changes in *GPOPS* Version 5.0

All of the changes in *GPOPS* Version 5.0 are internal. Specifically, the mesh refinement method used in *GPOPS* has been updated to be more robust from that which was used in *GPOPS* Version 4.x. In addition, the automatic scaling routine has been revised and this modification has been found to work significantly better than the previous automatic scaling routine. It is noted that users of *GPOPS* 4.x will not see any changes in syntax to the software, but it is expected (hoped) that this new version of *GPOPS* will run more efficiently in comparison to Version 4.x

## Licensing Agreement

By downloading, using, modifying, or distributing *GPOPS* , you agree to the terms of this license agreement. This license gives you extremely GENEROUS RIGHTS, so if you do not agree to the terms of this agreement, you may not proceed further with using, using, modifying, or distributing GPOPS.

### License for GPOPS Software

This is a license for the software General Pseudospectral Optimal Control Software (*GPOPS* ). The license for *GPOPS* is based on the Simple Public License. In the same spirit as the Simple Public License, the language for the *GPOPS* License is similar to that of GPL 2.0. The words are different, but the goal is the same: to guarantee for all users the freedom to share and change software. If anyone wonders about the meaning of the *GPOPS* License, they should interpret it as consistent with GPL 2.0.

The *GPOPS* License applies to the software's source and object code and comes with any rights that I have in it (other than trademarks). You agree to the *GPOPS* License simply by downloading, copying, distributing, or making a derivative work of the software. You get the royalty-free right to

- Use the software for any purpose;
- Make derivative works of it (this is called a "Derived Work");
- Copy and distribute it and any Derived Work.

If you distribute the software or a Derived Work, you must give back to the community by

- Prominently noting the date of any changes you make;
- Leaving other people's copyright notices, warranty disclaimers, and license terms in place;
- Providing the source code, build scripts, installation scripts, and interface definitions in a form that is easy to get and best to modify;
- Licensing it to everyone under the terms of this license agreement without adding further restrictions to the rights provided;
- Conspicuously announcing that it is available under this license.

## Restrictions for Use and Distribution of *GPOPS*

*GPOPS* is a completely free software both for use and for redistribution. Furthermore, while it may be used within commercial organizations, it is not for sale or resale. The only exception to the sales restriction above is that *GPOPS* may be included as a part of a free open-source software (for example, a distribution of the Linux operating system). When distributing *GPOPS* with a free operating system, no fee beyond the price of the operating system itself may be added (that is, you cannot profit from the redistribution of *GPOPS* ). *GPOPS* is not for commercial use with the exception that it may be used by commercial organizations for internal research and development. Any use of *GPOPS* by commercial organizations that involve the presentation of results for profit-making purposes is strictly prohibited. In addition, there are some things that you must shoulder:

- You get *no warranties* of any kind;
- If the software damages you in any way, you may only recover direct damages up to the amount you paid for it (that is, you get zero if you did not pay anything for the software);
- You may not recover any other damages, including those called "consequential damages." (The state or country where you live may not allow you to limit your liability in this way, so this may not apply to you).

The *GPOPS* License continues perpetually, except that your license rights end automatically if

- You do not abide by the "give back to the community" terms (your licensees get to keep their rights if they abide);
- Anyone prevents you from distributing the software under the terms of this license agreement.
- You sell the software in any manner with the one exception listed above.

In addition, to the license given above, the authors of *GPOPS* request that the following documents be cited in any publication where *GPOPS* was used to obtain the results:

(1) Rao, A. V., Benson, D. A., Darby, C. L., Patterson, M. A., Francolin, C., Sanders, I., and Huntington, G. T., "Algorithm 902: *GPOPS* , A MATLAB Software for Solving Multiple-Phase Optimal Control Problems Using the Gauss Pseudospectral Method," *ACM Transactions on Mathematical Software*, Vol. 37, No. 2, April–June, 2010, Article 22, 39 pages.

(2) 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.

(3) Garg, D., Patterson, M. A., Darby, C. L., Francolin, C., Huntington, G. T., Hager, W. W., and Rao, A. V., "Direct Trajectory Optimization and Costate Estimation of Finite-Horizon and Infinite-Horizon Optimal Control Problems Using a Radau Pseudospectral Method," *Computational Optimization and Applications*, Vol. 49, No. 2, June 2011, pp. 335–358.

(4) Garg, D., Patterson, M. A., Hager, W. W., Rao, A. V., Benson, D. A., and Huntington, G. T., "A Unified Framework for the Numerical Solution of Optimal Control Problems Using Pseudospectral Methods," *Automatica*, Vol. 46, No. 11 November 2010, pp. 1843-1851.

(5) Garg, D., Hager, W. W., and Rao, A. V., "Pseudospectral Methods for Solving Infinite-Horizon Optimal Control Problems," *Automatica,* Vol. 47, No. 4, April 2011, pp. 829–837.

# Contents

**1 Introduction to General Pseudospectral Optimization Software (*GPOPS* ) 6**
1.1 Radau Pseudospectral Method Employed by *GPOPS* . . . 6
1.2 Organization of *GPOPS* . . . 6
1.3 Notation Used Throughout Remainder of This Manual . . . 8
1.4 Constructing an Optimal Control Problem in *GPOPS* . . . 8
1.5 Preliminary Information . . . 8

**2 Constructing an Optimal Control Problem Using *GPOPS* 8**
2.1 Syntax for Input Structure **setup** . . . 8
2.2 Syntax for Structure **setup.funcs** . . . 9
2.3 Syntax for **limits** Structure . . . 9
2.4 Syntax for **linkages** Array of Structures . . . 13
2.5 Syntax of Each Function Specified in Structure **setup.funcs** . . . 14
2.6 Syntax of Cost Functional Specified in **setup.funcs.cost** . . . 14
2.7 Syntax for Differential-Algebraic Equations Function Specified in **setup.funcs.dae** . . . 15
2.8 Syntax of Event Constraint Function Specified in **setup.funcs.event** . . . 16
2.9 Syntax of Linkage Constraint Function Specified in **setup.funcs.link** . . . 18
2.10 Specifying an Initial Guess of The Solution . . . 18
2.11 Scaling of Optimal Control Problem . . . 19

**3 Specification of Parameters for Mesh Refinement 20**

**4 Different Options for Specification of Derivatives 20**
4.1 Complex-Step Differentiation . . . 21
4.2 Analytic Differentiation . . . 21

**5 Output from an Execution of *GPOPS* 27**

**6 Useful Information for Debugging a *GPOPS* Problem 28**

**7 *GPOPS* Examples 28**
7.1 Hyper-Sensitive Problem . . . 28
7.2 Bryson-Denham Problem . . . 31
7.3 Multiple-Stage Launch Vehicle Ascent Problem . . . 34
7.4 Minimum Time-to-Climb of a Supersonic Aircraft . . . 45

**8 Concluding Remarks 52**

# 1 Introduction to General Pseudospectral Optimization Software (*GPOPS* )

*General Pseudospectral Optimization Software* (*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

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

subject to the dynamic constraint

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

the boundary conditions

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

the inequality path constraints

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

and the phase continuity (linkage) constraints

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

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 Radau Pseudospectral Method Employed by *GPOPS*

The method employed by *GPOPS* is the *Radau Pseudospectral Method* (RPM). The RPM is an orthogonal collocation method where the collocation points are the *Legendre-Gauss-Radau* points. The theory of the RPM can be found in.[2,3,4,5,6] Some of the interesting features of the RPM are as follows: (1) it is a Gaussian quadrature implicit integration scheme; (2) it has been demonstrated to converge exponentially fast for problems whose solutions are smooth; (3) an elegant connection exists between the continuous-time optimal control problem and the discrete approximation; (4) it lends itself to the *hp*–adaptive approach used in *GPOPS* .

## 1.2 Organization of *GPOPS*

*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)

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

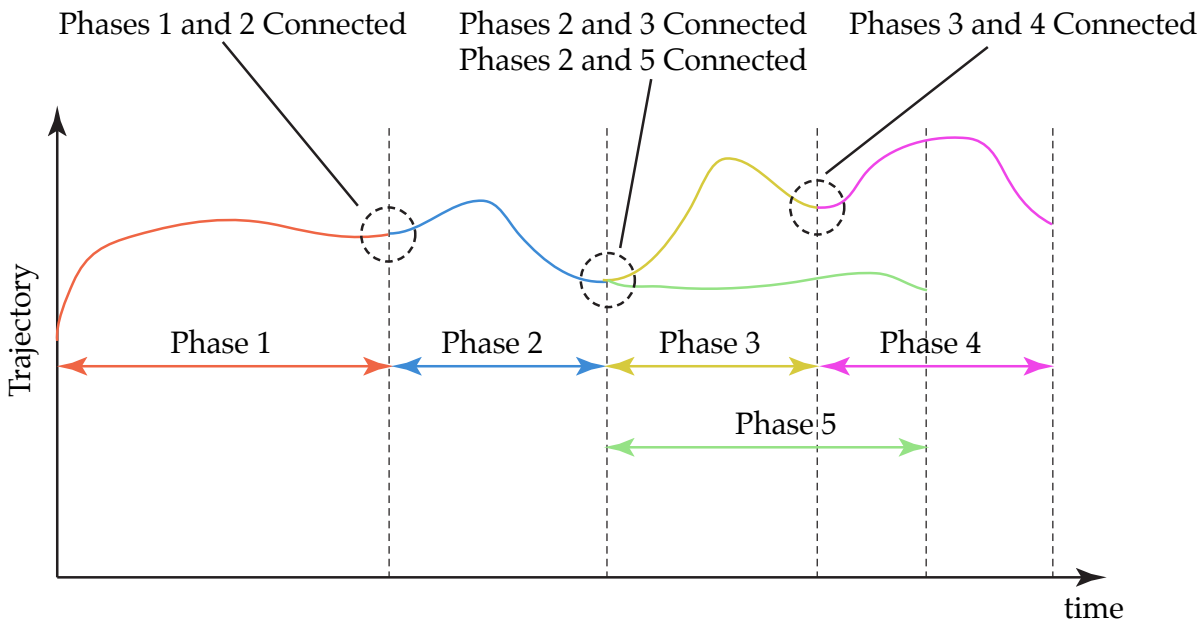



Figure 1: Schematic of linkages for multiple-phase optimal control problem. The example shown in the picture consists of five phases where the ends of phases 1, 2, and 3 are linked to the starts of phases 2, 3, and 4, respectively, while the end of phase 3 is linked to the start of phase 5.

(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.

## 1.3 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**.

## 1.4 Constructing an Optimal Control Problem in *GPOPS*

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

## 1.5 Preliminary Information

Before proceeding to the details of setting up a problem in *GPOPS* , the following few preliminary details are useful. First, it is important to understand that the *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 *GPOPS* uses the vectorization capabilities of MATLAB. In this vein all matrices and vectors in *GPOPS* are oriented ***column-wise*** for maximum efficiency. As you read through manual, please keep in mind the column-wise orientation of all matrices used in *GPOPS* .

# 2 Constructing an Optimal Control Problem Using *GPOPS*

In this Section we provide the details of constructing a problem using *GPOPS* . First, the call to *GPOPS* is deceptively simple and is given as follows:

**[output,gpopsHistory]=gpops(setup)**

The input *setup* is a user-defined structure that contains all of the information about the optimal control problem to be solved [2]. Finally, the variables *output* and *gpopsHistory* are a structure and an array of structures that contain, respectively, the information on the final run of the mesh refinement (*output*) and a complete history of the solutions on every mesh on which the problem was solved (*gpopsHistory*).[3].

## 2.1 Syntax for Input Structure setup

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 2.2 below).
- **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 2.3 below).
- **guess**: an array of structures that contains contains a guess of the solution in each phase of the problem (see Section 2.10 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 2.4 below).

[2] see the detailed description of *setup* in Section 2.1
[3] See the detailed description of the output in Section 5.

- **mesh**: Specifies the parameters to be used by the *hp*–adaptive method refinement algorithm that is implemented in *GPOPS* (see Section 3 below).
- **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 2.11 below).
- **derivatives**: a string indicating differentiation method to be used. Possible values for this string are "finite-difference", "complex", "automatic", "automatic-INTLAB", "analytic" (default="finite-difference") (see Section 4 below).
- **checkDerivatives**: a flag to check user defined analytic derivatives (default="0") (see Section 4 below).
- **maxIterations**: a positive integer indicating the maximum number of iterations that can be taken by the NLP solver.
- **printoff**: a flag that will supress all printing from *GPOPS* to the screen (default="0").
- **tolerances**: two element array specifiying the NLP solver Optimality and Feasibility Tolerances (default="[1e-6, 2e-6]").

Furthermore, it is important to note that *GPOPS* has been designed so that the independent variable must be monotonically *increasing'* across each phase of the trajectory.

## 2.2 Syntax for Structure setup.funcs

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 *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 *GPOPS* is given as follows:

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

## 2.3 Syntax for limits Structure

Once the user-defined structure *setup* has been defined, the next step in setting up a problem for use with *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**($p$)**.meshPoints**: a monotonically increasing row vector of length $M_p$, ($p \in [1, \ldots, P]$), where each entry in the vector is on the domain $[-1, +1]$, that contains a set of mesh points for the initial run of *GPOPS* . If the user does not have an estimate of the mesh point locations, this field should be left blank.

- **limits($p$).nodesPerInterval**: a row vector of length $M_p - 1$, $(p \in [1, \ldots, P])$, where each entry in the vector is a positive integer that contains the number of collocation points in each mesh interval for the initial run of *GPOPS* . If the user does not have an estimate of the number of collocation points in each mesh interval, this entry should be left blank.

- **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:

$$\begin{aligned} \textbf{limits}(p).\textbf{time.min} &= \begin{bmatrix} t_0^{\min} & t_f^{\min} \end{bmatrix} \\ \textbf{limits}(p).\textbf{time.max} &= \begin{bmatrix} t_0^{\max} & t_f^{\max} \end{bmatrix} \end{aligned}$$

- **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:

$$\textbf{limits}(p).\textbf{state.min} = \begin{bmatrix} x_{10}^{\min} & x_1^{\min} & x_{1f}^{\min} \\ \vdots & \vdots & \vdots \\ x_{n0}^{\min} & x_n^{\min} & x_{nf}^{\min} \end{bmatrix}$$

$$\textbf{limits}(p).\textbf{state.max} = \begin{bmatrix} x_{10}^{\max} & x_1^{\max} & x_{1f}^{\max} \\ \vdots & \vdots & \vdots \\ x_{n0}^{\max} & x_n^{\max} & x_{nf}^{\max} \end{bmatrix}$$

- **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:

$$\textbf{limits}(p).\textbf{control.min} = \begin{bmatrix} u_1^{\min} \\ \vdots \\ u_m^{\min} \end{bmatrix}$$

$$\textbf{limits}(p).\textbf{control.max} = \begin{bmatrix} u_1^{\max} \\ \vdots \\ u_m^{\max} \end{bmatrix}$$

- **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:

$$\textbf{limits}(p).\textbf{parameter.min} = \begin{bmatrix} q_1^{\min} \\ \vdots \\ q_{q_p}^{\min} \end{bmatrix}$$

$$\textbf{limits}(p).\textbf{parameter.max} = \begin{bmatrix} q_1^{\max} \\ \vdots \\ q_{q_p}^{\max} \end{bmatrix}$$

- **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:

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

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

- **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:

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

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

- **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:

$$\textbf{limits}(p).\textbf{duration.min} = T^{\min}$$
$$\textbf{limits}(p).\textbf{duration.max} = T^{\max}$$

- **limits($p$).dependencies**: (optional) Matrix of size ($n_p$+$r_p$ x $n_p$+$m_p$) which defines the dependencies of the dae functions on the state and control in phase $p \in [1, \ldots, P]$. An entry of 1 indicates that ode / path constraint corresponding to the row depends on (i.e. has a non-zero derivative) the state / control corresponding to the column. An entry of 0 indicates no dependence. User specification of this matrix reduces the number of values in the non-linear sparsity problem and can improve the solution time. A simple finite-difference check is performed to avoid specifying no dependence if a dependence actually exists. (default=all-ones)

**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 Structure**

As an example of setting up a limits structure in *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. Furthermore, suppose in phase 1 that we choose to initialize *GPOPS* with four mesh points) at the locations $(-1, -1/3, 1/3, +1)$ with 3, 4, and 5 collocation points, respectively, in the first, second, and third mesh intervals. In addition, suppose 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 that we choose to initialize *GPOPS* with a mesh consisting of six mesh points $(-1, -0.75, -0.5, 0, 0.5, 0.75, +1)$ with the 2, 4, 4, 3, and 2 collocation points in the first through fifth mesh intervals, respectively. In addition, 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

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

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

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

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

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

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

```
iphase = 1; % Set the phase number to 1
limits(iphase).meshPoints = [-1 -1/3 1/3 +1];
limits(iphase).nodesPerInterval = [3 4 5];
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).meshPoints = [-1 -0.75 -0.5 0.5 0.75 1];
limits(iphase).nodesPerInterval = [2 4 4 3 2];
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.

## 2.4 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.

## 2.5 Syntax of Each Function Specified in Structure setup.funcs

Now that we know *which* functions *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.

## 2.6 Syntax of Cost Functional Specified in setup.funcs.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 LGR 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 $1 \times 1$
- *Lagrange*: a *column* vector of size $N \times 1$

### 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; (6)

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., finite-difference or automatic differentiation) is as follows:

$$\boxed{\text{Lagrange=zeros(size(}\textit{solcost}\textbf{.time)};} \tag{7}$$

The user is urged to use the syntax of Eq. (7) 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 Syntax for Differential-Algebraic Equations Function Specified in setup.funcs.dae

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 LGR 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 $N$ LGR 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];
```

## 2.8 Syntax of Event Constraint Function Specified in setup.funcs.event

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 structure 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 structure *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;
```

## 2.9 Syntax of Linkage Constraint Function Specified in setup.funcs.link

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;
```

## 2.10 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 monotonically *increasing*. 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.

## 2.11 Scaling of Optimal Control Problem

As with any numerical optimization procedure, the approach employed by *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 *GPOPS* . This procedure is based on the scaling algorithm developed in.[7] 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.

# 3 Specification of Parameters for Mesh Refinement

An *hp*–adaptive mesh refinement algorithm is now included as part of *GPOPS* . While the user does not need to provide any parameters in order to use this algorithm, supplying values for these parameters is recommended. The mesh refinement algorithm parameters are specified in the structure *setup*.**mesh** and are given as follows (with the default values shown in the parentheses):

- **tolerance**: a scalar real number containing the mesh refinement tolerance (**default: mesh.tolerance**=$10^{-3}$).
- **iteration**: a positive integer containing the number of mesh refinement iterations to perform (**default: mesh.iteration**=10).
- **nodesPerInterval**: a structure containing the fields **min** and **max**, where **min** and **max** are integers containing the minimum and maximum number of allowable collocation points in a mesh interval (**defaults: nodesPerInterval.min**=4 and **nodesPerInterval.max**=12).
- **splitmult**: a real number greater than or equal to unity that specified how quickly to increase the total number of segments in the mesh (**default: mesh.splitmult**=1.2).

It is noted that the user should use the default values until the problem under consideration is reasonably well understood. Finally, if one wants to solve a problem with no mesh refinement and with a specified mesh distribution, it is necessary to set *setup*.**mesh.iteration**=0 and provide the necessary information in fields **meshPoints** and **nodesPerInterval** of the **limits** structure as described in Section 2.3.

# 4 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 "finite-difference", "complex", "automatic", "automatic-INTLAB", and "analytic" and correspond to the following differentiation methods:

- *setup*.**derivatives**="finite-difference": default internal sparse finite-differencing algorithm 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**="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/`.

## 4.1 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 *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. 8 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.

## 4.2 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 *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.

### 4.2.1 Syntax of Cost Function Using Analytic Derivatives

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

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

See Section 2.6 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 LGR 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:

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

where

$$\begin{aligned} \frac{\partial \Phi}{\partial \mathbf{x}(t_0)} &\in \mathbb{R}^{1\times n} \\ \frac{\partial \Phi}{\partial t_0} &\in \mathbb{R} \\ \frac{\partial \Phi}{\partial \mathbf{x}(t_f)} &\in \mathbb{R}^{1\times n} \\ \frac{\partial \Phi}{\partial t_f} &\in \mathbb{R} \\ \frac{\partial \Phi}{\partial \mathbf{p}} &\in \mathbb{R}^{1\times q} \end{aligned} \tag{8}$$

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

$$\mathbf{DerivLagrange} = \begin{bmatrix} \dfrac{\partial \mathcal{L}}{\partial \mathbf{x}} & \dfrac{\partial \mathcal{L}}{\partial \mathbf{u}} & \dfrac{\partial \mathcal{L}}{\partial t} & \dfrac{\partial \mathcal{L}}{\partial p} \end{bmatrix}$$

where

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{x}} &\in \mathbb{R}^{N\times n} \\ \frac{\partial \mathcal{L}}{\partial \mathbf{u}} &\in \mathbb{R}^{N\times m} \\ \frac{\partial \mathcal{L}}{\partial t} &= \mathbb{R}^{N\times 1} \\ \frac{\partial \mathcal{L}}{\partial \mathbf{p}} &= \mathbb{R}^{N\times q} \end{aligned} \tag{9}$$

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(size(t)), zeros(length(t),length(p))];
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*.

### 4.2.2 Syntax of Differential-Algebraic Equations Function Using Analytic 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 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 LGR 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 $N$ LGR points:

$$\mathbf{Derivdae} = \begin{bmatrix} \frac{\partial f_1}{\partial \mathbf{x}} & \frac{\partial f_1}{\partial \mathbf{u}} & \frac{\partial f_1}{\partial t} & \frac{\partial f_1}{\partial \mathbf{p}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_2}{\partial \mathbf{x}} & \frac{\partial f_2}{\partial \mathbf{u}} & \frac{\partial f_2}{\partial t} & \frac{\partial f_2}{\partial \mathbf{p}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_n}{\partial \mathbf{x}} & \frac{\partial f_n}{\partial \mathbf{u}} & \frac{\partial f_n}{\partial t} & \frac{\partial f_n}{\partial \mathbf{p}} \\ \frac{\partial C_1}{\partial \mathbf{x}} & \frac{\partial C_1}{\partial \mathbf{u}} & \frac{\partial C_1}{\partial t} & \frac{\partial C_1}{\partial \mathbf{p}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial C_r}{\partial \mathbf{x}} & \frac{\partial C_r}{\partial \mathbf{u}} & \frac{\partial C_r}{\partial t} & \frac{\partial C_r}{\partial \mathbf{p}} \end{bmatrix}$$

where $f_i$, $(i = 1, \ldots, n)$ is the right-hand side of the $i^{th}$ differential equation, and $C_j$, $(j = 1, \ldots, r)$ is the $j^{th}$ path constraint. Each of the elements of *Derivdae* have the following sizes:

$$\begin{aligned} \frac{\partial f_i}{\partial \mathbf{x}} &\in \mathbb{R}^{N\times n}, \quad (i = 1, \ldots, n) \\ \frac{\partial f_i}{\partial \mathbf{u}} &\in \mathbb{R}^{N\times m}, \quad (i = 1, \ldots, n) \\ \frac{\partial f_i}{\partial t} &\in \mathbb{R}^{N\times 1}, \quad (i = 1, \ldots, n) \\ \frac{\partial f_i}{\partial \mathbf{p}} &\in \mathbb{R}^{N\times q}, \quad (i = 1, \ldots, n) \\ \frac{\partial C_i}{\partial \mathbf{x}} &\in \mathbb{R}^{N\times n}, \quad (i = 1, \ldots, r) \\ \frac{\partial C_i}{\partial \mathbf{u}} &\in \mathbb{R}^{N\times m}, \quad (i = 1, \ldots, r) \\ \frac{\partial C_i}{\partial t} &\in \mathbb{R}^{N\times 1}, \quad (i = 1, \ldots, r) \\ \frac{\partial C_i}{\partial \mathbf{p}} &\in \mathbb{R}^{N\times q}, \quad (i = 1, \ldots, r) \end{aligned} \tag{10}$$

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_dt,   df1_dp; ...
            df2_dx1,   df2_dx2,   df2_du1,   df2_du2,   df2_dt    df2_dp,; ...
          dpath_dx1, dpath_dx2, dpath_du1, dpath_du2, dpath_dt,  dpath_dp,];
```

### 4.2.3 Syntax of Event Constraint Function Using Analytic 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 2.8 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 $\phi_i$, $(i = 1, \ldots, e)$ is the $i^{th}$ event constraint. The sizes of each of the entries in *Derivevents* are as follows:

$$\begin{aligned} \frac{\partial \phi_i}{\partial \mathbf{x}(t_0)} &\in \mathbb{R}^{1\times n} \\ \frac{\partial \phi_i}{\partial t_0} &\in \mathbb{R} \\ \frac{\partial \phi_i}{\partial \mathbf{x}(t_f)} &\in \mathbb{R}^{1\times n}, \quad (i = 1, \ldots, e) \\ \frac{\partial \phi_i}{\partial t_f} &\in \mathbb{R} \\ \frac{\partial \Phi}{\partial \mathbf{p}} &\in \mathbb{R}^{1\times q} \end{aligned} \tag{11}$$

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_dt, dei1_dp; ...
               dei2_dx0, dei2_dt0, dei1_dxf, dei2_dtf, dei2_dt, dei2_dp; ...
               def1_dx0, def1_dt0, def1_dxf, def1_dtf, def1_dt, def1_dp; ...
               def2_dx0, def2_dt0, def2_dxf, def2_dtf, def2_dt, def2_dp; ...
               def3_dx0, def3_dt0, def3_dxf, def3_dtf, def3_dt, def3_dp];
```

### 4.2.4 Syntax of Linkage Constraint Function Using Analytic 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 2.9 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:

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

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))];
```

# 5 Output from an Execution of *GPOPS*

Upon execution of *GPOPS* , new fields are created in the output structure *output*. In particular, upon completion of the execution of *GPOPS* , the following new fields are created (in addition to the fields that were created prior to running *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
- **solutionPlot:** an array of structures of length $P$ (where $P$ is the number of phases) containing a solution obtained using Lagrange polynomial interpolation that can be used for a graphical display of the solution (often because the structure **solution** is obtained on a coarse grid due to the high accuracy of the pseudospectral method).

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

- **solution($p$).time:** a column vector containing the time at each discretization point along the trajectory.
- **solution($p$).state:** an array whose rows contain the state at each time point in **solution($p$).time**.
- **solution($p$).control:** an array whose rows contain the control at each time point in **solution($p$).time**.
- **solution($p$).parameter:** a column vector containing the static parameters.
- **solution($p$).costate:** an array whose rows contain the costate at each time point in **solution($p$).time**.
- **solution($p$).pathmult:** an array whose rows contain the path constraint multipliers at each time point in
- **solution($p$).Hamiltonian:** a column vector whose rows contain the Hamiltonian at each time point in **solution($p$).time**.
- **solution($p$).Mayer_cost:** The Mayer part of the cost along the trajectory
- **solution($p$).Lagrange_cost:** The Lagrange (integrated) cost along the trajectory

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

- **solution(*p*).time:** a column vector containing the time at each discretization point along the trajectory.
- **solution(*p*).state:** an array whose rows contain the state at each time point in **solution(*p*).time**.
- **solution(*p*).control:** an array whose rows contain the control at each time point in **solution(*p*).time**.
- **solution(*p*).costate:** an array whose rows contain the costate at each time point in **solution(*p*).time**.

It is noted that the field **parameter** would be the same in **solutionPlot** as it is in **solution** and thus, **parameter** is omitted from **solutionPlot**.

# 6 Useful Information for Debugging a *GPOPS* Problem

One aspect of *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 *GPOPS* uses collocation at *Legendre-Gauss-Radau* points. Because the Legendre-Gauss-Radau points include the initial point but do not include the final point, the dynamics, path constraints, and integrand cost are computed only at the Legendre-Gauss-Radau points. While this may appear to be a bit strange, the fundamental point here is that Legendre-Gauss-Radau quadrature (which is used in *GPOPS* ) only evaluates the functions at the Legendre-Gauss-Radau points. Do not try to "fool" *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 *GPOPS* , see the references on the Radau pseudospectral method as given in the bibliography at the end of this manual.

# 7 *GPOPS* Examples

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

## 7.1 Hyper-Sensitive Problem

Consider the following optimal control problem. Minimize the cost functional

$$J = \frac{1}{2}\int_0^{t_f} \left(x^2 + u^2\right) dt \tag{12}$$

subject to the dynamic constraint

$$\dot{x} = -x^3 + u \tag{13}$$

and the boundary conditions

$$\begin{array}{rcl} x(0) & = & 1.5 \\ x(t_f) & = & 1 \end{array} \tag{14}$$

with $t_f = 50$. It is noted that this problem is taken from Ref. 9. The *GPOPS* code that solves this problem is shown below. In particular, the following three MATLAB functions are defined:

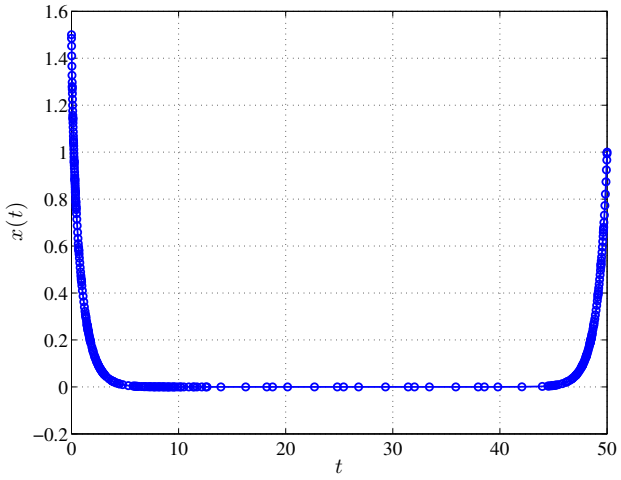
- hyperSensitiveMain.m: MATLAB m-file (main driver) for problem
- hyperSensitiveCost.m: MATLAB function that evaluates the cost functional
- hyperSensitiveDae.m: MATLAB function that evaluates the differential-algebraic equations

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

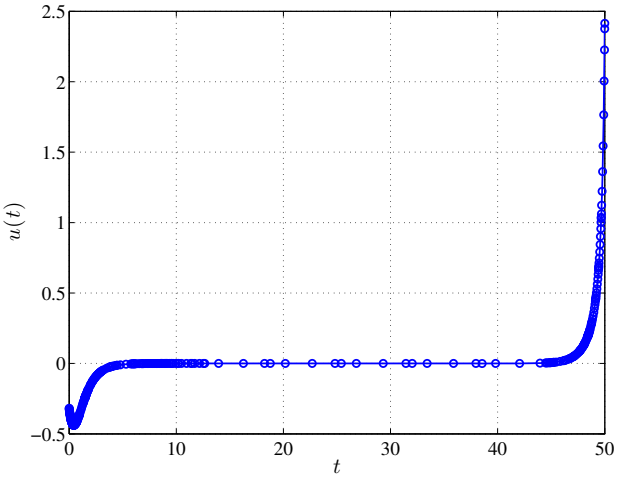
```
%-------------------------------------------------------------------%
% BEGIN: script hyperSensitiveMain.m                                %
%-------------------------------------------------------------------%
% This example is taken from the following reference:               %
% Rao, A. V. and 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.                                            %
% The optimal control problem is described as follows:              %
% Minimize J = 0.5*(x^2+u^2)                                        %
% subject to the dynamic constraint                                 %
%   xdot = -x^3 + u                                                 %
% and the boundary conditions                                       %
%   x(0) = 1.5                                                      %
%   x(tf) = 1                                                       %
%-------------------------------------------------------------------%
clear all
clc

t0 = 0; tf = 5000; x0 = 1.5; xf = 1;
xmin = -10; xmax =  10; umin = -10; umax =  10;

iphase = 1;
limits(iphase).time.min = [t0 tf];
limits(iphase).time.max = [t0 tf];
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 = [t0; tf];
guess(iphase).state(:,1) = [x0; xf];
guess(iphase).control = [-1; 1];
guess(iphase).parameter = [];

setup.name  = 'HyperSensitive-Problem';
setup.funcs.cost = 'hyperSensitiveCost';
setup.funcs.dae = 'hyperSensitiveDae';
setup.linkages = [];
setup.limits = limits;
setup.guess = guess;
setup.derivatives = 'finite-difference';
setup.checkDerivatives = 0;
setup.autoscale = 'off';
setup.mesh.tolerance = 1e-6;
setup.mesh.iteration = 20;
setup.mesh.nodesPerInterval.min = 4;
setup.mesh.nodesPerInterval.max = 10;

[output,gpopsHistory] = gpops(setup);
%---------------------------------%
% END: script hyperSensitiveMain.m %
%---------------------------------%

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

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

%--------------------------------------%
```

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

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

dae = -x.^3+u;

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

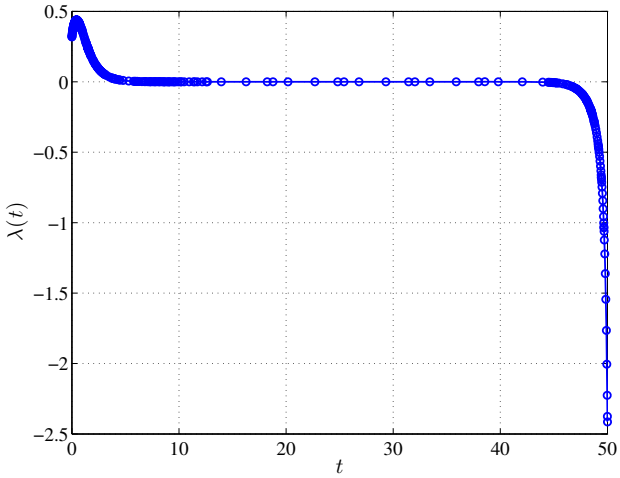
The state, $x(t)$, control, $u(t)$, and costate, $\lambda(t)$ resulting from the execution of *GPOPS* using the above code is summarized in Figs. 2a–2c.




(a) State, $x(t)$, vs. $t$.




(b) Control, $u(t)$, vs. $t$.




(c) Costate, $\lambda(t)$, vs. $t$.

Figure 2: State,Control, and Costate for Hyper-Sensitive Optimal Control Problem

## 7.2 Bryson-Denham Problem

Consider the following optimal control problem. Minimize the cost functional

$$J = x_3(t_f) \tag{15}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= u \\ \dot{x}_3 &= \tfrac{1}{2}u^2 \end{aligned} \tag{16}$$

the path constraint

$$0 \leq x_1(t) \leq 1/9 \tag{17}$$

and the boundary conditions

$$\begin{aligned} x_1(0) &= 0 \\ x_2(0) &= 1 \\ x_3(0) &= 0 \\ x_1(t_f) &= 0 \\ x_2(t_f) &= -1 \end{aligned} \tag{18}$$

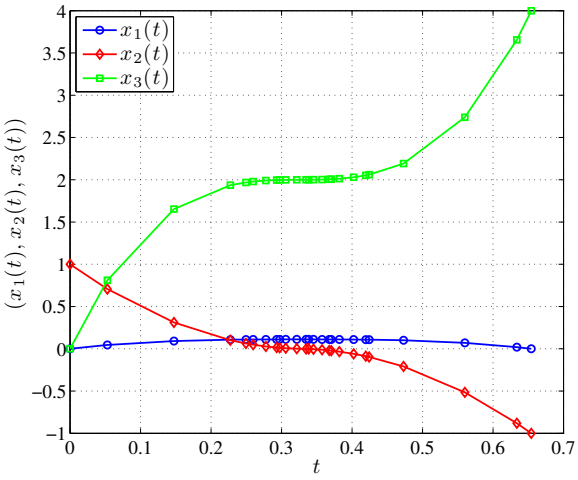
The above problem was originally formulated by Bryson and Denham[10] and is referred to as the *Bryson-Denham* problem. The *GPOPS* code that solves the Bryson-Denham problem is shown below. In particular, the following four MATLAB files are defined:

- 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

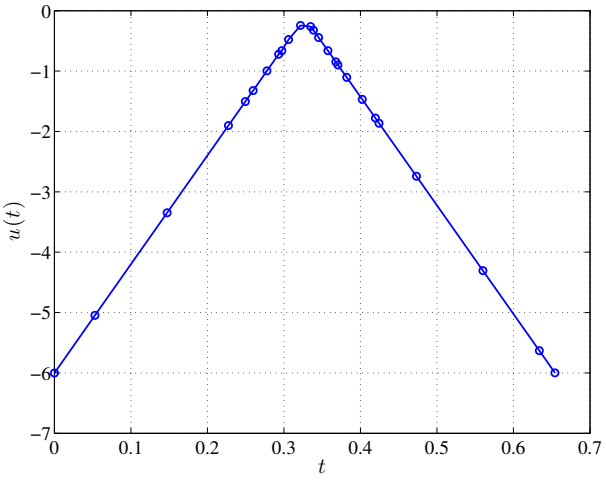
The beginning and end of each function is labeled by a MATLAB comment. It is noted that while all five boundary conditions are simple bounds (and are, thus, linear, they are treated as general event constraints in order to demonstrate the proper use of an event function.

```
%-------------------------------------------------------%
% 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 all
clc

x10 = 0;
x20 = 1;
x30 = 0;
x1f = 0;
x2f = -1;
x1min = 0;
x1max = 1/9;
x2min = -10;
x2max = 10;
x3min = -10;
x3max =  10;

param_min = [];
param_max = [];
path_min = [];
path_max = [];
event_min = [];
event_max = [];
duration_min = [];
```

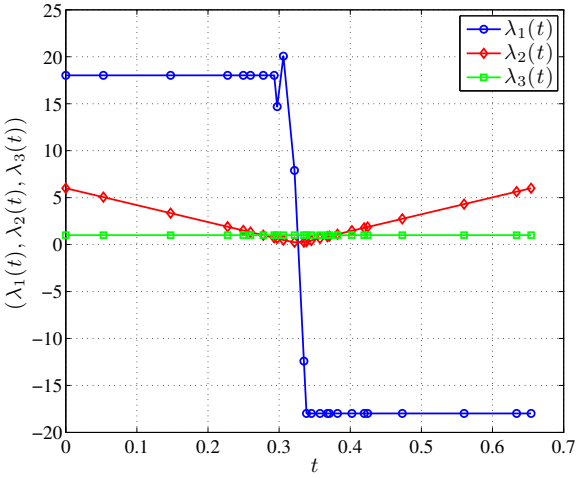
```
duration_max = [];

iphase = 1;
limits(iphase).time.min = [0 0];
limits(iphase).time.max = [0 50];
limits(iphase).state.min(1,:) = [x10 x1min x1f];
limits(iphase).state.max(1,:) = [x10 x1max x1f];
limits(iphase).state.min(2,:) = [x20 x2min x2f];
limits(iphase).state.max(2,:) = [x20 x2max x2f];
limits(iphase).state.min(3,:) = [x30 x3min x3min];
limits(iphase).state.max(3,:) = [x30 x3max x3max];
limits(iphase).control.min    = -5;
limits(iphase).control.max    =  10;
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; 1];
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       = [];

setup.name  = 'Bryson-Denham-Problem';
setup.funcs.cost = 'brysonDenhamCost';
setup.funcs.dae = 'brysonDenhamDae';
setup.limits = limits;
setup.guess = guess;
setup.linkages = [];
setup.derivatives = 'finite-difference';
setup.checkDerivatives = 0;
setup.autoscale = 'off';
setup.mesh.tolerance = 1e-6;
setup.mesh.iteration = 10;
setup.mesh.nodesPerInterval.min = 4;
setup.mesh.nodesPerInterval.max = 10;
[output,gpopsHistory] = gpops(setup);
solution = output.solution;
solutionPlot = output.solutionPlot;

%---------------------------------%
% END: script brysonDenhamMain.m %
%---------------------------------%


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

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

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

t = sol.time;
x = sol.state;
u = sol.control;
```




(a) State vs. Time.




(b) Control vs. Time.




(c) Costate vs. Time.

```
x1dot = x(:,2);
x2dot = u;
x3dot = u.^2/2;
dae = [x1dot x2dot x3dot];

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

The output obtained by solving the Bryson-Denham problem using the *GPOPS* code above is summarized in Figs. 3a–3c.

## 7.3 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 Ref. 11.

### 7.3.1 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 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 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 |

### 7.3.2 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

$$\begin{aligned} \dot{\mathbf{r}} &= \mathbf{v} \\ \dot{\mathbf{v}} &= -\frac{\mu}{\|\mathbf{r}\|^3}\mathbf{r} + \frac{T}{m}\mathbf{u} + \frac{\mathbf{D}}{m} \\ \dot{m} &= -\frac{T}{g_0 I_{sp}} \end{aligned} \tag{19}$$

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, $\mu$ 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

$$\mathbf{D} = -\frac{1}{2} C_D A_{ref} \rho \|\mathbf{v}_{rel}\| \mathbf{v}_{rel} \tag{20}$$

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

$$\mathbf{v}_{rel} = \mathbf{v} - \boldsymbol{\omega} \times \mathbf{r} \tag{21}$$

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

$$\rho = \rho_0 \exp[-h/h_0] \tag{22}$$

where $\rho_0$ 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 2.

Table 2: Constants used in the launch vehicle example.

| Constant | Value |
|---|---|
| Payload Mass (kg) | 4164 |
| $A_{ref}$ ($m^2$) | $4\pi$ |
| $C_d$ | 0.5 |
| $\rho_0$ ($kg/m^3$) | 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 |

### 7.3.3 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

$$\begin{aligned} \mathbf{r}(t_0) &= \mathbf{r}_0 = \begin{bmatrix} 5605.2 & 0 & 3043.4 \end{bmatrix}^T \quad \text{km} \\ \mathbf{v}(t_0) &= \mathbf{v}_0 = \begin{bmatrix} 0 & 0.4076 & 0 \end{bmatrix}^T \quad \text{km/s} \\ m(t_0) &= m_0 = 301454 \quad \text{kg} \end{aligned} \tag{23}$$

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

$$\begin{aligned} a_f &= 24361.14 \text{ km}, \\ e_f &= 0.7308, \\ i_f &= 28.5 \text{ deg}, \\ \Omega_f &= 269.8 \text{ deg}, \\ \omega_f &= 130.5 \text{ deg} \end{aligned} \tag{24}$$

The orbital elements, $a, e, i, \Omega$, and $\omega$ represent the semi-major axis, eccentricity, inclination, right ascension of the ascending node (RAAN), and argument of perigee, respectively. Note that the true anomaly, $\nu$, 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.[12]

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

$$|\mathbf{r}| \geq R_r \tag{25}$$

where $R_e$ is the radius of the Earth, as seen in Table 2. Next, a path constraint is imposed on the control to guarantee that the control vector is unit length, so that

$$|\mathbf{u}| = 1 \tag{26}$$

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

$$\begin{aligned} \mathbf{r}^{(p)}(t_f) - \mathbf{r}^{(p+1)}(t_0) &= \mathbf{0}, \\ \mathbf{v}^{(p)}(t_f) - \mathbf{v}^{(p+1)}(t_0) &= \mathbf{0}, \qquad (p = 1, \ldots, 3) \\ m^{(p)}(t_f) - m^{(p)}_{dry} - m^{(p+1)}(t_0) &= 0 \end{aligned} \tag{27}$$

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

$$J = -m^{(4)}(t_f) \tag{28}$$

subject to the conditions of Eqs. (19), (23), (24), (25), and (26).

The MATLAB code that solves the multiple-stage launch vehicle ascent problem using *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 guess limits 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).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) = [0; 0];
guess(iphase).control(:,2) = [1; 1];
guess(iphase).control(:,3) = [0; 0];
guess(iphase).parameter    = [];

iphase = 2;
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) = [0; 0];
guess(iphase).control(:,2) = [1; 1];
guess(iphase).control(:,3) = [0; 0];
guess(iphase).parameter    = [];

iphase = 3;
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) = [1; 1];
guess(iphase).control(:,3) = [0; 0];
guess(iphase).parameter    = [];

iphase = 4;
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) = [1; 1];
guess(iphase).control(:,3) = [0; 0];
guess(iphase).parameter    = [];

ipair = 1; % First pair of phases to link
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 link
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 link
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.name = 'Launch-Vehicle-Ascent';
setup.funcs.cost = 'launchCost';
```

```
setup.funcs.dae = 'launchDae';
setup.funcs.event = 'launchEvent';
setup.funcs.link = 'launchLink';
setup.derivatives = 'finite-difference';
setup.checkDerivatives = 0;
setup.limits = limits;
setup.guess = guess;
setup.linkages = linkages;
setup.autoscale = 'on';

if isequal(setup.derivatives,'automatic-intlab'),
    CONSTANTS.derivatives = 'automatic-intlab';
else
    CONSTANTS.derivatives = [];
end;
setup.mesh.tolerance = 1e-6;
setup.mesh.iteration = 10;
setup.mesh.nodesPerInterval.min = 4;
setup.mesh.nodesPerInterval.max = 12;

[output,gpopsHistory] = gpops(setup);
solutionPlot= output.solutionPlot;
solution = output.solution;

% ------------------------%
% End File:  launchMain.m %
% ------------------------%


% --------------------------%
% Begin File:  launchCost.m %
% --------------------------%
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==4,
        % 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

% ------------------------%
% End File:  launchCost.m %
% ------------------------%


% -------------------------%
% Begin File:  launchDae.m %
% -------------------------%
```

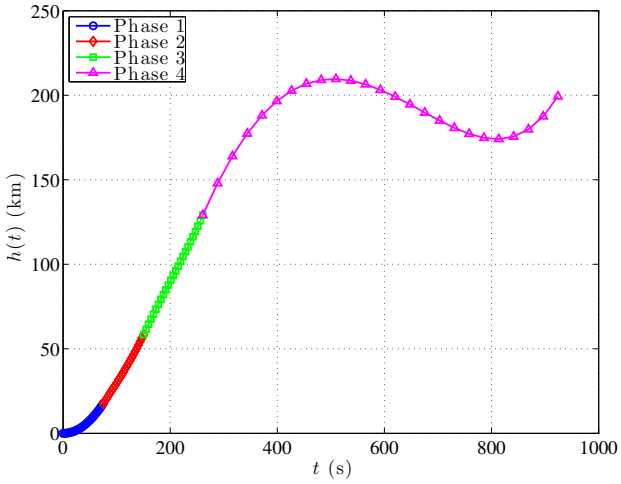
```
function [dae Ddae] = launchDae(sol);

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;
omegacrossr = r*omega_matrix.';
vrel = v-omegacrossr;
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

dspeedreldv1 = (v(:,1)-omegacrossr(:,1))./speedrel;
dspeedreldv2 = (v(:,2)-omegacrossr(:,2))./speedrel;
dspeedreldv3 = (v(:,3)-omegacrossr(:,3))./speedrel;
dDrag1_dv1 = -bc.*((v(:,1)-omegacrossr(:,1)).^2+speedrel.^2)./speedrel;
% dDrag1_dv1 = -bc.*(dspeedreldv1.*v(:,1)+speedrel)./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.*(dspeedreldv2.*v(:,2)+speedrel);
dDrag2_dv2 = -bc.*((v(:,2)-omegacrossr(:,2)).^2+speedrel.^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.*(dspeedreldv3.*v(:,3)+speedrel);
dDrag3_dv3 = -bc.*((v(:,3)-omegacrossr(:,3)).^2+speedrel.^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
```

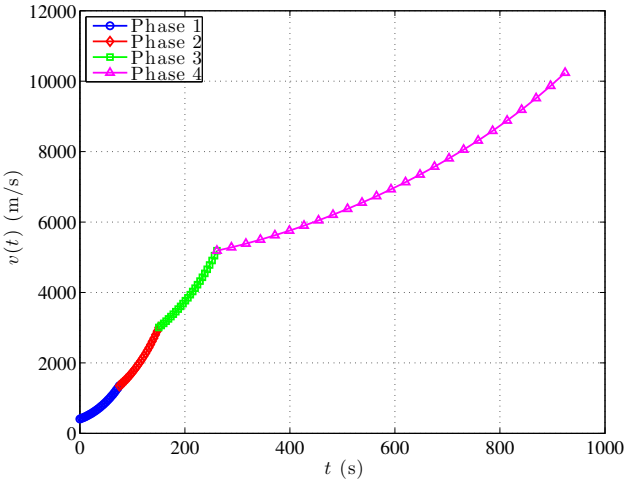
```
    % mass dynamics independant of State
    % Ddae(6*N+1:7*N,:) = 0

    %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

% ------------------------%
% End File:  launchDae.m %
% ------------------------%

% ---------------------------%
% Begin File:  launchEvent.m %
% ---------------------------%
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

% ------------------------%
% End File:  launchEvent.m %
% ------------------------%

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;
```

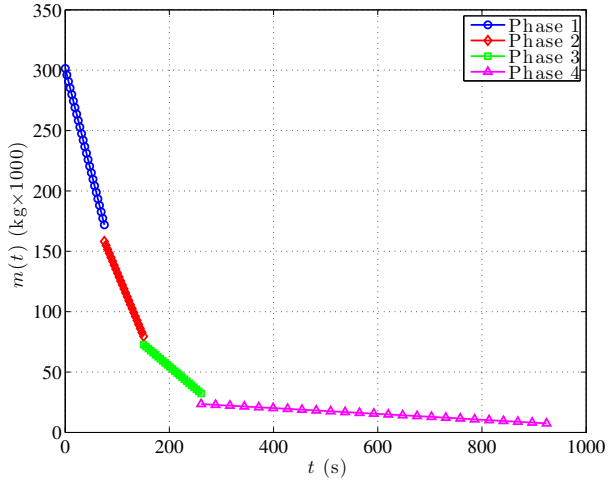
The output of the above code from *GPOPS* is summarized in the following three plots that contain the altitude, speed, and controls.



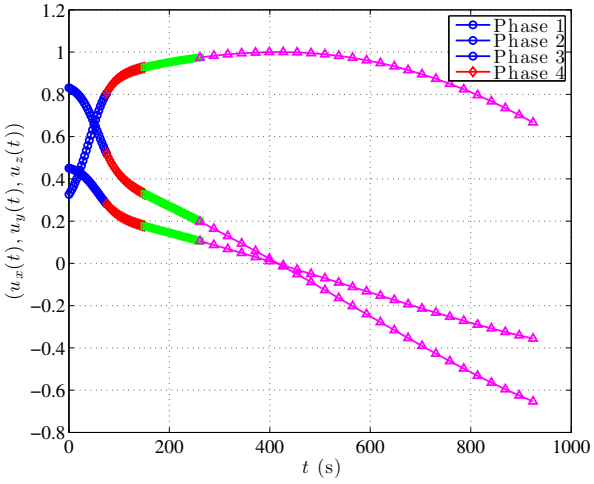

(a) Altitude vs. Time.




(b) Speed vs. Time.




(c) Mass vs. Time.




(d) Control vs. Time.

## 7.4 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 Ref. 13, but the model used in this study was taken from Ref. 7 with the exception that a linear extrapolation of the thrust data as found in Ref. 7 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

$$J = t_f \tag{29}$$

subject to the dynamic constraints

$$\dot{h} = v \sin\alpha \tag{30}$$

$$\dot{v} = \frac{T\cos\alpha - D}{m} \tag{31}$$

$$\dot{\gamma} = \frac{T\sin\alpha + L}{mv} + \left(\frac{v}{r} - \frac{\mu}{vr^2}\right)\cos\gamma \tag{32}$$

$$\dot{m} = -\frac{T}{g0 I_{sp}} \tag{33}$$

and the boundary conditions

$$h(0) = 0 \text{ ft} \tag{34}$$

$$v(0) = 129.3144 \text{ m/s} \tag{35}$$

$$\gamma(0) = 0 \text{ rad} \tag{36}$$

$$h(t_f) = 19994.88 \text{ m} \tag{37}$$

$$v(t_f) = 295.092 \text{ ft/s} \tag{38}$$

$$\gamma(t_f) = 0 \text{ rad} \tag{39}$$

where $h$ is the altitude, $v$ is the speed, $\gamma$ is the flight path angle, $m$ is the vehicle mass, $T$ is the magnitude of the thrust force, and $D$ is the magnitude of the drag force. It is noted that this example uses table data obtained from Ref. 13. In this example *GPOPS* is implemented using the finite-difference derivative option (that is, using the option *setup*.**derivatives**='finite-difference') together with an interpolation of the table data with the MATLAB functions 'interp1' and 'interp2'. The MATLAB code that solves the minimum time-to-climb of a supersonic aircraft is shown below.

```
% ------------------------------------------------
% Minimum Time-to-Climb of a Supersonic Aircraft
% ------------------------------------------------
% This example is taken verbatim from the following reference:
% Betts, J. T., Practical Methods for Optimal Control Using
% Nonlinear Programming, SIAM Press, Philadelphia, 2001.
clear setup limits guess

% Initialize all of the data for the problem
load brysonMinimumClimbAeroData.mat;
global CONSTANTS;

% U.S. 1976 Standard Atmosphere Table
% Reference:  U.S. 1976 Standard Atmosphere, National Oceanographic
% and Atmospheric Administration, 1976.
% Column 1:  Altitude (m)
% Column 2:  Atmospheric Density (kg/m^3)
% Column 3:  Speed of Sound (m/s)
us1976 = [-2000    1.478e+00     3.479e+02
    0    1.225e+00     3.403e+02
    2000    1.007e+00     3.325e+02
    4000    8.193e-01     3.246e+02
    6000    6.601e-01     3.165e+02
    8000    5.258e-01     3.081e+02
    10000    4.135e-01     2.995e+02
    12000    3.119e-01     2.951e+02
```

```
    14000    2.279e-01     2.951e+02
    16000    1.665e-01     2.951e+02
    18000    1.216e-01     2.951e+02
    20000    8.891e-02     2.951e+02
    22000    6.451e-02     2.964e+02
    24000    4.694e-02     2.977e+02
    26000    3.426e-02     2.991e+02
    28000    2.508e-02     3.004e+02
    30000    1.841e-02     3.017e+02
    32000    1.355e-02     3.030e+02
    34000    9.887e-03     3.065e+02
    36000    7.257e-03     3.101e+02
    38000    5.366e-03     3.137e+02
    40000    3.995e-03     3.172e+02
    42000    2.995e-03     3.207e+02
    44000    2.259e-03     3.241e+02
    46000    1.714e-03     3.275e+02
    48000    1.317e-03     3.298e+02
    50000    1.027e-03     3.298e+02
    52000    8.055e-04     3.288e+02
    54000    6.389e-04     3.254e+02
    56000    5.044e-04     3.220e+02
    58000    3.962e-04     3.186e+02
    60000    3.096e-04     3.151e+02
    62000    2.407e-04     3.115e+02
    64000    1.860e-04     3.080e+02
    66000    1.429e-04     3.044e+02
    68000    1.091e-04     3.007e+02
    70000    8.281e-05     2.971e+02
    72000    6.236e-05     2.934e+02
    74000    4.637e-05     2.907e+02
    76000    3.430e-05     2.880e+02
    78000    2.523e-05     2.853e+02
    80000    1.845e-05     2.825e+02
    82000    1.341e-05     2.797e+02
    84000    9.690e-06     2.769e+02
    86000    6.955e-06     2.741e+02];

% Mtab is a table of Mach number values
Mtab   = [0; 0.2; 0.4; 0.6; 0.8; 1; 1.2; 1.4; 1.6; 1.8];
% alttab is a table of altitude values (in ft)
alttab = [0 5000 10000 15000 20000 25000 30000 40000 50000 70000];
% Convert altitude table to meters
alttab = 0.3048*alttab;
% Ttab is a table of aircraft thrust values (in lbf)
% Ttab is taken from Bryson's 1969 Journal of Aircraft paper (also
% Betts' book), but the table has been extended via linear
% extrapolation to fill in the "missing" data points.
Ttab = 1000*[24.2 24.0  20.3 17.3 14.5 12.2 10.2 5.7 3.4 0.1;
    28.0 24.6 21.1 18.1 15.2 12.8 10.7 6.5 3.9 0.2;
    28.3 25.2 21.9 18.7 15.9 13.4 11.2 7.3 4.4 0.4;
    30.8 27.2 23.8 20.5 17.3 14.7 12.3 8.1 4.9 0.8;
    34.5 30.3 26.6 23.2 19.8 16.8 14.1 9.4 5.6 1.1;
    37.9 34.3 30.4 26.8 23.3 19.8 16.8 11.2 6.8 1.4;
    36.1 38.0 34.9 31.3 27.3 23.6 20.1 13.4 8.3 1.7;
    36.1 36.6 38.5 36.1 31.6 28.1 24.2 16.2 10.0 2.2;
    36.1 35.2 42.1 38.7 35.7 32.0 28.1 19.3 11.9 2.9;
    36.1 33.8 45.7 41.3 39.8 34.6 31.1 21.7 13.3 3.1];
% Convert Thrust to Newtons
Ttab = 4.448222*Ttab;

% M2 is the Mach number used to compute the aerodynamic coefficients
M2         = [0 0.4 0.8 0.9 1.0 1.2 1.4 1.6 1.8];
Clalphatab = [3.44 3.44 3.44 3.58 4.44 3.44 3.01 2.86 2.44];
CD0tab     = [0.013 0.013 0.013 0.014 0.031 0.041 0.039 0.036 0.035];
etatab     = [0.54 0.54 0.54 0.75 0.79 0.78 0.89 0.93 0.93];

CONSTANTS.CDdat     = CDdat;
CONSTANTS.CLdat     = CLdat;
CONSTANTS.etadat    = etadat;
CONSTANTS.M         = Mtab;
CONSTANTS.M2        = M2;
CONSTANTS.alt       = alttab;
CONSTANTS.T         = Ttab;
CONSTANTS.Clalpha   = Clalphatab;
CONSTANTS.CD0       = CD0tab;
CONSTANTS.eta       = etatab;
CONSTANTS.ppCLalpha = polyfit(CONSTANTS.M2,CONSTANTS.Clalpha,8);
CONSTANTS.ppCD0     = polyfit(CONSTANTS.M2,CONSTANTS.CD0,8);
```

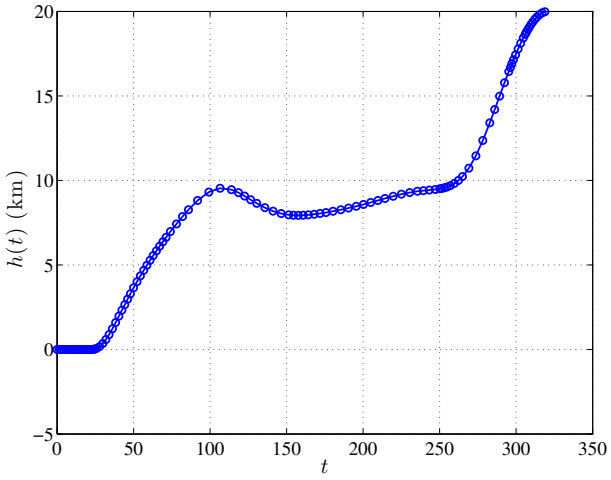
```
CONSTANTS.ppeta     = polyfit(CONSTANTS.M2,CONSTANTS.eta,8);
CONSTANTS.Re        = 6378145;
CONSTANTS.mu        = 3.986e14;
CONSTANTS.S         = 49.2386;
CONSTANTS.g0        = 9.80665;
CONSTANTS.Isp       = 1600;
CONSTANTS.H         = 7254.24;
CONSTANTS.rho0      = 1.225;
CONSTANTS.us1976    = us1976;

mass0 = 19050.864;
% Compute Scale Factors
if 0,
  scales.length = CONSTANTS.Re;
  scales.area = scales.length*scales.length;
  scales.volume = scales.area*scales.length;
  scales.speed = sqrt(CONSTANTS.mu/scales.length);
  scales.time = scales.length/scales.speed;
  scales.acceleration = scales.speed/scales.time;
  scales.mass = mass0;
  scales.force = scales.mass*scales.acceleration;
  scales.density = scales.mass/scales.volume;
  scales.gravparameter = scales.volume/scales.time^2;
else
  scales.length = 1;
  scales.area = 1;
  scales.volume = 1;
  scales.speed = 1;
  scales.time = 1;
  scales.acceleration = 1;
  scales.mass = 1;
  scales.force = 1;
  scales.density = 1;
  scales.gravparameter = 1;
end;

% Scale all quantities in problem
CONSTANTS.us1976(:,1) = us1976(:,1)/scales.length;
CONSTANTS.us1976(:,2) = us1976(:,2)/scales.density;
CONSTANTS.us1976(:,3) = us1976(:,3)/scales.speed;
CONSTANTS.alt         = CONSTANTS.alt/scales.length;
CONSTANTS.T           = CONSTANTS.T/scales.force;
CONSTANTS.Re          = CONSTANTS.Re/scales.length;
CONSTANTS.mu          = CONSTANTS.mu/scales.gravparameter;
CONSTANTS.S           = CONSTANTS.S/scales.area;
CONSTANTS.g0          = CONSTANTS.g0/scales.acceleration;
CONSTANTS.Isp         = CONSTANTS.Isp/scales.time;
CONSTANTS.H           = CONSTANTS.H/scales.length;
CONSTANTS.rho0        = CONSTANTS.rho0/scales.density;
[aa,mm]               = meshgrid(alttab,Mtab);
CONSTANTS.aa          = aa;
CONSTANTS.mm          = mm;

% Boundary conditions (taken from Betts 2001 & converted to SI units)
t0   = 0/scales.time;          % Initial time [scales.time]
alt0 = 0/scales.time;          % Initial altitude [scales.length]
altf = 19994.88/scales.length; % Final altitude [scales.length]
speed0 = 129.314/scales.speed; % Initial speed [scales.speed]
speedf = 295.092/scales.speed; % Final speed [scales.speed]
fpa0   = 0;                    % Initial flight path angle [rad]
fpaf   = 0;                    % Final flight path angle [rad]
mass0  = mass0/scales.mass;    % Initial mass [scales.mass]

% Bounds on variables (taken from Betts, 2001)
tfmin  = 100/scales.time;
tfmax  = 800/scales.time;
altmin = 0/scales.length;
altmax = 21031.2/scales.length;
speedmin = 5/scales.speed;
speedmax = 1000/scales.speed;
fpamin   = -40*pi/180;
fpamax   =  40*pi/180;
massmin  = 22/scales.mass;
massmax  = 20410/scales.mass;
% alphamin = -pi/2;
% alphamax = pi/2;
alphamin = -pi/4;
alphamax = pi/4;
```

```
iphase = 1;
% Bounds on initial and terminal values of time
% limits(iphase).meshPoints = [-1 1];
% limits(iphase).nodesPerInterval = [20];
limits(iphase).time.min = [t0 tfmin];
limits(iphase).time.max = [t0 tfmax];
limits(iphase).state.min(1,:) = [alt0 altmin altf];
limits(iphase).state.max(1,:) = [alt0 altmax altf];
limits(iphase).state.min(2,:) = [speed0 speedmin speedf];
limits(iphase).state.max(2,:) = [speed0 speedmax speedf];
limits(iphase).state.min(3,:) = [fpa0 fpamin fpaf];
limits(iphase).state.max(3,:) = [fpa0 fpamax fpaf];
limits(iphase).state.min(4,:) = [mass0 massmin massmin];
limits(iphase).state.max(4,:) = [mass0 massmax massmax];
limits(iphase).control.min = alphamin;
limits(iphase).control.max = alphamax;
limits(iphase).parameter.min = [];
limits(iphase).parameter.max = [];
guess(iphase).time = [0; 100];
guess(iphase).state(:,1) = [alt0; altf];
guess(iphase).state(:,2) = [speed0; speedf];
guess(iphase).state(:,3) = [fpa0; fpaf];
guess(iphase).state(:,4) = [mass0; mass0];
guess(iphase).control = [20; -20]*pi/180;
guess(iphase).parameter = [];

setup.name  = 'Bryson-Minimum-Time-to-Climb-Problem';
setup.funcs.cost = 'brysonMinimumClimbCost';
setup.funcs.dae = 'brysonMinimumClimbDae';
setup.funcs.link = '';
setup.limits = limits;
setup.guess = guess;
%==============================================%
% WARNING:  AT THIS TIME THIS PROBLEM CAN ONLY   %
% BE SOLVED USING NUMERICAL DIFFERENTIATION!     %
% DO NOT SET "SETUP.DERIVATIVES" TO ANYTHING BUT %
% "finite-difference".                           %
%==============================================%
setup.derivatives = 'finite-difference';
setup.autoscale = 'on';
% setup.tolerances = [1e-3 2e-3];
setup.mesh.tolerance = 1e-4;
setup.mesh.iteration = 10;
setup.mesh.nodesPerInterval.min = 4;
setup.mesh.nodesPerInterval.max = 12;

output = gpops(setup);

solution = output.solution;
solutionPlot = output.solutionPlot;
plotfigures;

%---------------------------------------%
% End Function:  brysonMinimumClimbMain.m %
%---------------------------------------%

%-----------------------------------------%
% Begin Function:  brysonMinimumClimbCost.m %
%-----------------------------------------%
function [Mayer,Lagrange] = brysonMinimumClimbCost(solcost);

tf = solcost.terminal.time;
t  = solcost.time;

Mayer = tf;
Lagrange = zeros(size(t));
%-----------------------------------------%
% End Function:  brysonMinimumClimbCost.m %
%-----------------------------------------%

%-----------------------------------------%
% Begin Function:  brysonMinimumClimbDae.m %
%-----------------------------------------%
function dae = brysonMinimumClimbDae(sol)

global CONSTANTS

us1976 = CONSTANTS.us1976;
Ttab = CONSTANTS.T;
```

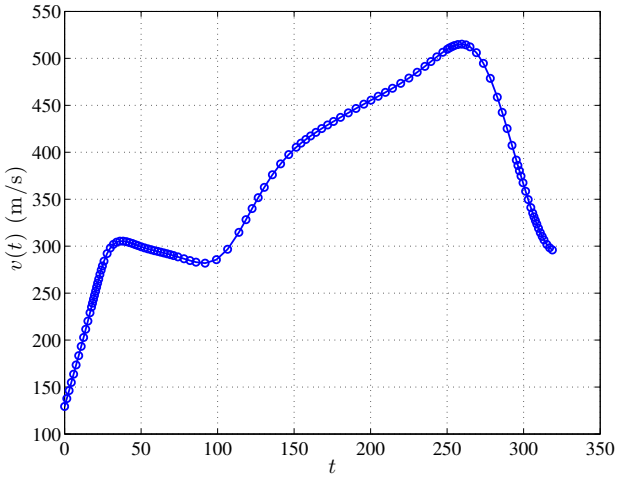
```
mu = CONSTANTS.mu;
S  = CONSTANTS.S;
g0 = CONSTANTS.g0;
Isp = CONSTANTS.Isp;
Re   = CONSTANTS.Re;

x = sol.state;
u = sol.control;

h = x(:,1);
v = x(:,2);
fpa = x(:,3);
mass = x(:,4);
alpha = u(:,1);

r   = h+Re;
rho = interp1(us1976(:,1),us1976(:,2),h,'spline');
sos = interp1(us1976(:,1),us1976(:,3),h,'spline');
Mach = v./sos;
[CD0,Clalpha,eta]=brysonMinimumClimbAeroCompute(Mach);
Thrust = interp2(CONSTANTS.aa,CONSTANTS.mm,Ttab,h,Mach,'spline');
CD = CD0 + eta.*Clalpha.*alpha.^2;
CL = Clalpha.*alpha;
q  = 0.5.*rho.*v.*v;
D  = q.*S.*CD;
L  = q.*S.*CL;
hdot = v.*sin(fpa);
vdot = (Thrust.*cos(alpha)-D)./mass - mu.*sin(fpa)./r.^2;
fpadot = (Thrust.*sin(alpha)+L)./(mass.*v)+cos(fpa).*(v./r-mu./(v.*r.^2));
mdot = -Thrust./(g0.*Isp);

dae = [hdot vdot fpadot mdot];

%----------------------------------------%
% End Function:  brysonMinimumClimbDae.m %
%----------------------------------------%

%----------------------------------------%
% Begin Function:  brysonMinimumClimbDae.m %
%----------------------------------------%
function dae = brysonMinimumClimbDae(sol)

global CONSTANTS

us1976 = CONSTANTS.us1976;
Ttab = CONSTANTS.T;

mu = CONSTANTS.mu;
S  = CONSTANTS.S;
g0 = CONSTANTS.g0;
Isp = CONSTANTS.Isp;
Re   = CONSTANTS.Re;

x = sol.state;
u = sol.control;

h = x(:,1);
v = x(:,2);
fpa = x(:,3);
mass = x(:,4);
alpha = u(:,1);

r   = h+Re;
rho = interp1(us1976(:,1),us1976(:,2),h,'spline');
sos = interp1(us1976(:,1),us1976(:,3),h,'spline');
Mach = v./sos;
[CD0,Clalpha,eta]=brysonMinimumClimbAeroCompute(Mach);
Thrust = interp2(CONSTANTS.aa,CONSTANTS.mm,Ttab,h,Mach,'spline');
CD = CD0 + eta.*Clalpha.*alpha.^2;
CL = Clalpha.*alpha;
q  = 0.5.*rho.*v.*v;
D  = q.*S.*CD;
L  = q.*S.*CL;
hdot = v.*sin(fpa);
vdot = (Thrust.*cos(alpha)-D)./mass - mu.*sin(fpa)./r.^2;
fpadot = (Thrust.*sin(alpha)+L)./(mass.*v)+cos(fpa).*(v./r-mu./(v.*r.^2));
mdot = -Thrust./(g0.*Isp);

dae = [hdot vdot fpadot mdot];
```

```
%--------------------------------------%
% End Function:  brysonMinimumClimbDae.m %
%--------------------------------------%
```

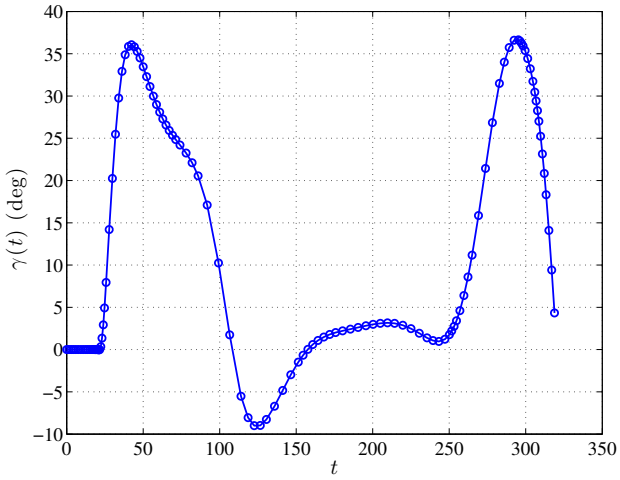
The components of the state and the control obtained from running the above *GPOPS* code is summarized in Figs. 3a–3e.



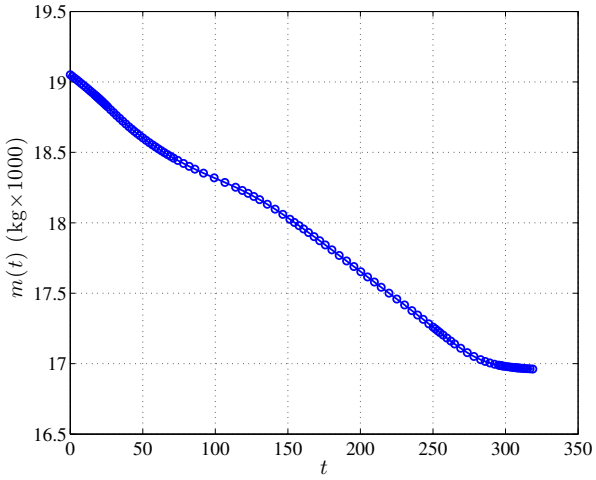

(a) Altitude vs. Time.



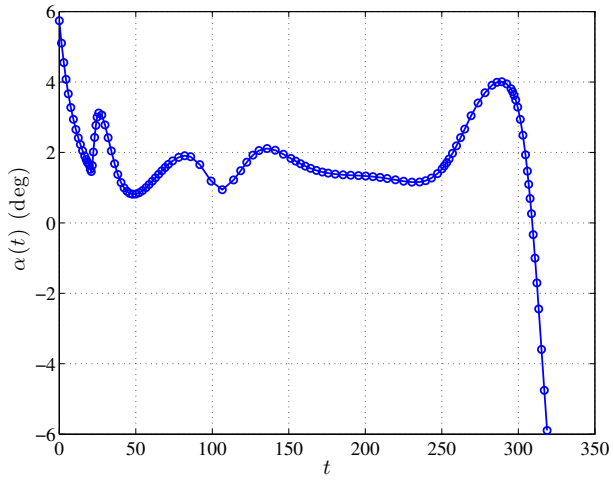

(b) Speed vs. Time.




(c) Flight Path Angle vs. Time.




(d) Mass vs. Time.




(e) Angle of Attack vs. Time.

# 8 Concluding Remarks

While the authors have put for the effort to make *GPOPS* a user-friendly software, it is important to understand several aspects of computational optimal control in order to make *GPOPS* easier to use. First, it is *highly* recommended that the user scale a problem manually using insight from the physics/mathematics of the problem because the automatic scaling procedure is by no means foolproof. Second, the particular parameterization of a problem can make all the difference with regard to obtaining a solution in a reliable manner. Finally, even if the NLP solver returns the result that the optimality conditions have been satisfied, it is important to verify the solution. In short, a great deal of time in solving optimal control problems is spent in formulation and analysis.

# References

[1] Gill, P. E., Murray, W., and Saunders, M. A., "SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization," *SIAM Review*, Vol. 47, No. 1, January 2002, pp. 99–131.

[2] Garg, D., Patterson, M. A., Darby, C. L., Francolin, C., Huntington, G. T., Hager, W. W., and Rao, A. V., "Direct Trajectory Optimization and Costate Estimation of Finite-Horizon and Infinite-Horizon Optimal Control Problems via a Radau Pseudospectral Method," *Computational Optimization and Applications*, Published Online: October 6, 2009. DOI 10.1007/s10589-009-9291-0. http://www.springerlink.com/content/n851q6n343p9k60k/.

[3] Garg, D., Patterson, M. A., Hager, W. W., Rao, A. V., Benson, D. A., and Huntington, G. T., "A Unified Framework for the Numerical Solution of Optimal Control Problems Using Pseudospectral Methods," *Automatica*, Vol. 46, No. 11, November 2010, pp. 1843–1851.

[4] Garg, D., Hager, W. W., and Rao, A. V., "Pseudospectral Methods for Solving Infinite-Horizon Optimal Control Problems," *Automatica*, Published Online March 2010. DOI:10.1016/j.automatica.2011.01.085.

[5] Garg, D., *Advances in Global Pseudospectral Methods for Optimal Control*, Ph.D. thesis, Department of Mechanical and Aerospace Engineering, University of Florida, August 2011.

[6] Patterson, M. A. and Rao, A. V., "Exploiting Sparsity in Direct Collocation Pseudospectral Methods for Solving Optimal Control Problems," *Journal of Spacecraft and Rockets*, June 2011, Accepted for Publication.

[7] Betts, J. T., "Sparse Jacobian Updates in the Collocation Method for Optimal Control Problems," *Journal of Guidance, Control, and Dynamics*, Vol. 13, No. 3, May–June 1990, pp. 409–415.

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

[9] Rao, A. V. and Mease, K. D., "A New Method for Solving Optimal Control Problems," *Proceedings of the AIAA Guidance, Navigation and Control Conference*, Baltimore, 1995, pp. 818–825.

[10] Bryson, A. E. and Ho, Y.-C., *Applied Optimal Control*, Hemisphere Publishing, New York, 1975.

[11] Benson, D. A., *A Gauss Pseudospectral Transcription for Optimal Control*, Ph.D. thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2004.

[12] Bate, R. R., Mueller, D. D., and White, J. E., *Fundamentals of Astrodynamics*, Dover Publications, 1971.

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