$\mathbb{GPOPS} - \mathbb{II}$ Version 1.0:

A General-Purpose MATLAB Toolbox for Solving Optimal Control Problems Using the Radau Pseudospectral Method

Michael A. Patterson
Anil V. Rao

University of Florida
Gainesville, FL 32611-6250
USA

January 2013

Preface

$\mathbb{GPOPS} - \mathbb{II}$ is a general-pupose software for solving nonlinear optimal control problems that arise in a wide variety applications including engineering, economics, and medicine. $\mathbb{GPOPS} - \mathbb{II}$ uses some of the latest advancements in the area of pseudospectral methods for solving optimal control problems. $\mathbb{GPOPS} - \mathbb{II}$ employs an *hp*-adaptive Radau pseudospectral Gaussian quadrature method where the collocation is performed at the Legendre-Gauss-Radau quadrature points. $\mathbb{GPOPS} - \mathbb{II}$ has been designed to work with the nonlinear programming (NLP) solvers SNOPT and IPOPT, and MATLAB mex files for both SNOPT and IPOPT are included with the software. $\mathbb{GPOPS} - \mathbb{II}$ employs sparse finite-differencing to estimate all first and second derivatives required by the NLP solver. The software has been designed to be extremely flexible, allowing a user to formulate an optimal control problem in a way that makes sense for the problem being solved. Few, if any, restrictions have been placed on the manner in which a problem needs to be modeled. As stated, the software is *general-purpose*, that is, it has not been developed for any specific type of problem. While the developers of $\mathbb{GPOPS} - \mathbb{II}$ make no guarantee as to the fitness of the software for any particular purpose, it is certainly hoped that software is useful for a variety of applications.

Complete Overhaul from Previous Versions of GPOPS

$\mathbb{GPOPS} - \mathbb{II}$ represents a complete overhaul from the GPOPS software that was released between 2008 and 2012. Specifically, this new software, $\mathbb{GPOPS} - \mathbb{II}$, is organized in a completely different manner from GPOPS and has significantly more functionality from GPOPS. Furthermore, $\mathbb{GPOPS} - \mathbb{II}$ does *not* maintain backward compatibility with GPOPS. While the authors of $\mathbb{GPOPS} - \mathbb{II}$ realize that the lack of backward compatibility may be inconvenient for some users, the increased power and functionality of $\mathbb{GPOPS} - \mathbb{II}$ will make it worth the short term inconvenience of the transition. In order to have as smooth a transition as possible to the new software, the authors of $\mathbb{GPOPS} - \mathbb{II}$ are happy to assist users of GPOPS in rewriting their code for $\mathbb{GPOPS} - \mathbb{II}$.

Acknowledgments

Disclaimer

The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. Furthermore, the contents of this document and the corresponding software are provided "as is" without any merchantability or fitness for any particular application. Neither authors nor their employers (past, present, or future) assume any responsibility whatsoever from any harm resulting from the software. The authors do, however, hope that users will find this software useful for research and other purposes.

Licensing Agreement

By downloading, using, modifying, or distributing $\mathbb{GPOPS}2$, 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 $\mathbb{GPOPS}2$.

License for $\mathbb{GPOPS} - \mathbb{II}$ Software

PLEASE READ THE FOLLOWING TERMS AND CONDITIONS CAREFULLY BEFORE PROCEEDING TO DOWNLOAD $\mathbb{GPOPS} - \mathbb{II}$!

This page constitutes the official license for the General-Purpose Pseudospectral Optimal Control Software (hereafter referred to as $\mathbb{GPOPS-II}$). By downloading, using, modifying, or distributing $\mathbb{GPOPS-II}$ you are agreeing to all of the the terms and conditions described henceforth. Do not proceed with the download of $\mathbb{GPOPS-II}$ you do not agree with any of the terms and conditions in this agreement.

If you are either a student or employee at an academic institution (that is, K-12 or degree-granting higher-learning institution), an employee at a not-for-profit, or an employee of the U.S. Federal Government or U.S. State Government, $\mathbb{GPOPS-II}$ is available free-of-charge provided you are using it for non-commercial research and development. This free-of-charge non-commercial use license comes with the following rules:

- You may use the software for any non-commercial purpose

- You may make derivative works of the software (called a Derived Work), and you may copy and distribute any such Derived Work provided that the Derived Work is also for non-commercial purposes

- If you distribute $\mathbb{GPOPS-II}$ or any Derived Work, you are required to provide any changes made to the source code to the authors of $\mathbb{GPOPS-II}$ so that we may include these changes in a future release if we find the changes useful

- You must leave all existing copyright notices, warranty disclaimers, and license terms in place in any changes that you make

- You may not add any restrictions beyond that of this license agreement in any Derived Work

- You must conspicuously announce that the changes you have made are available under the original license for $\mathbb{GPOPS-II}$

- You must provide assurance that you will not make a profit of any kind from the distribution or re-distribution of $\mathbb{GPOPS-II}$

If you are an employee at a profit-making institution, the download and use of $\mathbb{GPOPS-II}$ requires a licensing fee, and this fee structure applies regardless of the kind of work being performed at your profit-making institution. The commercial licensing fee structure is as follows:

- Individual License: \$1000.00 for first-time purchase of any version of $\mathbb{GPOPS-II}$; \$200.00 for the purchase of any upgrade to a future major release.

- Site-Wide Commercial License (Use at a Single Location): \$5000.00 for first time purchase of any version of $\mathbb{GPOPS-II}$; \$1000.00 for the purchase of any upgrade to a future major release.

- Company-Wide Commercial License: \$10000.00 for first time purchase of any version of $\mathbb{GPOPS-II}$; \$2000.00 for the purchase of any upgrade to a future major release.

DISCLAIMER: $\mathbb{GPOPS-II}$ is provided as-is with no warranties, none of any kind. Furthermore, the authors may not be held liable for any use of $\mathbb{GPOPS-II}$, and you may not recover any damages for any reason whatsoever.

Distribution of $\mathbb{GPOPS-II}$

$\mathbb{GPOPS-II}$ 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 $\mathbb{GPOPS-II}$ may be included as a part of a free open-source software (for example, a distribution of the Linux operating system). When distributing $\mathbb{GPOPS-II}$ 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 $\mathbb{GPOPS-II}$). $\mathbb{GPOPS-II}$ is not for commercial use with the exception that it may be used by commercial organizations for internal research and development. Any use of $\mathbb{GPOPS-II}$ 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 $\mathbb{GPOPS} - \mathbb{II}$ 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.

$\mathbb{GPOPS} - \mathbb{II}$ *is provided "as is" without warranty of any kind, expressed or implies, including but not limited to the warranties of merchantability, fitness for a particular purpose, and non-infringement. In no event shall the authors or copyright holders be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the software or the use or dealings in the software.*

Contents

1   Introduction to the General-Purpose Software $\mathbb{GPOPS} - \mathbb{II}$

A P-phase optimal control problem can be stated in the following general form. Determine the state, $\mathbf{y}^{(p)}(t) \in \mathbb{R}^{n_y^{(p)}}$, control, $\mathbf{u}^{(p)}(t) \in \mathbb{R}^{n_u^{(p)}}$, initial time, $t_0^{(p)} \in \mathbb{R}$, final time, $t_f^{(p)} \in \mathbb{R}$, integrals, $\mathbf{q}^{(p)} \in \mathbb{R}^{n_q^{(p)}}$, in each phase $p \in [1, \ldots, P]$, and the static parameters, $\mathbf{s} \in \mathbb{R}^{n_s}$, that minimize the cost functional

$$J = \phi \left[\mathbf{y}^{(1)}(t_0^{(1)}), \ldots, \mathbf{y}^{(P)}(t_0^{(P)}), t_0^{(1)}, \ldots, t_0^{(P)}, \mathbf{y}^{(1)}(t_f^{(1)}), \ldots, \mathbf{y}^{(P)}(t_f^{(P)}), t_f^{(1)}, \ldots, t_f^{(P)}, \mathbf{q}^{(1)}, \ldots, \mathbf{q}^{(P)}, \mathbf{s} \right] \quad (1)$$

subject to the dynamic constraints

$$\dot{\mathbf{y}}^{(p)} = \mathbf{a}^{(p)} \left[\mathbf{y}^{(p)}, \mathbf{u}^{(p)}, t^{(p)}, \mathbf{s} \right], \qquad (p = 1, \ldots, P), \quad (2)$$

the event constraints

$$\mathbf{b}_{\min}^{(g)} \le \mathbf{b} \left[\mathbf{y}^{(1)}(t_0^{(1)}), \ldots, \mathbf{y}^{(P)}(t_0^{(P)}), t_0^{(1)}, \ldots, t_0^{(P)}, \mathbf{y}^{(1)}(t_f^{(1)}), \ldots, \mathbf{y}^{(P)}(t_f^{(P)}), t_f^{(1)}, \ldots, t_f^{(P)}, \mathbf{q}^{(1)}, \ldots, \mathbf{q}^{(P)}, \mathbf{s} \right] \le \mathbf{b}_{\max}^{(g)},$$
$$(g = 1, \ldots, G), \quad (3)$$

the inequality path constraints

$$\mathbf{c}_{\min}^{(p)} \le \mathbf{c}^{(p)} \left[\mathbf{y}^{(p)}, \mathbf{u}^{(p)}, t^{(p)}, \mathbf{s} \right] \le \mathbf{c}_{\max}^{(p)}, \qquad (p = 1, \ldots, P), \quad (4)$$

and the integral constraints

$$\mathbf{q}_{\min}^{(p)} \le \mathbf{q}^{(p)} \le \mathbf{q}_{\max}^{(p)}, \quad (p = 1, \ldots, P) \quad (5)$$

where

$$q_i^{(p)} = \int_{t_0^{(p)}}^{t_f^{(p)}} Q_i \left[\mathbf{y}^{(p)}, \mathbf{u}^{(p)}, t^{(p)}, \mathbf{s} \right] dt, \quad (i = 1, \ldots, n_q^{(p)}; p = 1, \ldots, P). \quad (6)$$

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).

1.1   Radau Pseudospectral Method Employed by $\mathbb{GPOPS} - \mathbb{II}$

The method employed by $\mathbb{GPOPS} - \mathbb{II}$ is an *hp*-adaptive version of the *Radau pseudospectral method*. The Radau pseudospectral method is an orthogonal collocation Gaussian quadrature implicit integration method where collocation is performed at the *Legendre-Gauss-Radau* points. The theory behind the Radau pseudospectral method used in $\mathbb{GPOPS} - \mathbb{II}$ can be found in Refs. 1, 2, 3, and 4.

1.2   Organization of $\mathbb{GPOPS} - \mathbb{II}$

$\mathbb{GPOPS} - \mathbb{II}$ is organized as follows. In order to specify the optimal control problem that is to be solved, the user must write the following MATLAB functions: (1) an endpoint function; and (2) a continuous function. The endpoint function defines how the the start and/or terminus in any of the phases in the problem, the integrals in any phase of the problem and the static parameters are related to one another. The endpoint function also defines the cost to be minimized. The *continuous* function defines the evolution of the dynamics in any phase of the problem, the integrands that are required to compute any integrals in any phase of the problem, and any path constraints in any phase of the problem. Next, the user must specify the lower and upper limits on the following quantities:

(1) the time at the start and terminus of a phase;

(2) the state at the start of a phase, during a phase, and at the terminus of a phase;

(3) the control during a phase;

(4) the path constraints

(5) the event constraints;

(6) the static parameters.

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 Color Highlighting Throughout Document

The following notation is adopted for use throughout the remainder of this document. First, all user-specified names will be denoted by *red slanted* characters. Second, any item denoted by **blue boldface** characters are pre-defined and cannot be changed by the user. Users who do not have color rendering capability will see only slanted and boldface characters, respectively.

2 Constructing an Optimal Control Problem Using $\mathbb{GPOPS-II}$

We now proceed to describe the constructs required to specify an optimal control problem in $\mathbb{GPOPS-II}$. We note that the key MATLAB programming elements used in constructing an optimal control problem in $\mathbb{GPOPS-II}$ are *structure* and *arrays of structures.* In this Section we provide the details of constructing a problem using $\mathbb{GPOPS-II}$. First, the call to $\mathbb{GPOPS-II}$ s given as

$$output=\textbf{gpops2}(input),$$

where *input* is a user-defined structure that contains all of the information about the optimal control problem to be solved and *output* is a structure that contains the information obtained by solving the optimal control problem. In this section we describe the contents of the structures *input* and *output*.

2.1 Syntax for Input Structure **setup**

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

- **name**: a string *with no blank spaces* that contains the name of the problem;

- **functions**: a structure that contains the name of the continuous function and the endpoint function (see Section 2.2 for further details);

- **bounds**: an structure that contains the information about the lower and upper bounds on the different variables and constraints in the problem (see Section 2.3 for further details);

- **guess**: an structure that contains a guess of the time, state, control, integrals, and static parameters in the problem (see Section 2.6 for further details);

In addition to the above required fields in the structure *setup*, *optional* fields in the *setup* structure may be specified (these fields may be provided by the user if it may of benefit for a particular problem of interest). The optional fields in the structure *setup* are given as follows along with the list of possible values and the default values:

- **auxdata**: a structure containing auxiliary data that may be used by different functions in the problem. Including **auxdata** eliminates any need to specify global variables for use in the problem. The following table provided the possible values and their defaults for the field *setup*.**auxdata**:

Field	Possible Values	Default
setup.**auxdata**	Any Problem-Specific Data	Not Provided

- **derivatives**: a structure that specifies the derivative approximation to be used by the NLP solver and the derivative order ('first' or 'second') to be used by the NLP solver. The field *setup*.**derivatives** contains three fields **supplier**, **derivativelevel**, and **dependencies** where the field *setup*.**derivatives**.**supplier** contains the type of derivative approximation , the field *setup*.**derivatives**.**derivativelevel** contains the derivative order, while the field *setup*.**derivatives**.**dependencies** determines how the dependencies are found. The following table provided the possible values and their defaults for the field *setup*.**derivatives**:

Field	Possible Values	Default
setup.**derivatives**.**supplier**	'sparseFD', 'sparseBD' or 'sparseCD'	'sparseFD'
setup.**derivatives**.**derivativelevel**	'first' or 'second'	'first'
setup.**derivatives**.**dependencies**	'full', 'sparse' or 'sparseNaN'	'sparseNaN'

- **scales**: a structure that specifies the type of scaling to be used when solving the problem. [**Possible Values:** 'none' or 'automatic-bounds'; **Default:** 'none'];

Field	Possible Values	Default
setup.**scales**	'none' or 'automatic-bounds'	'none'

- **mesh**: a structure that specifies the information as to the type of mesh refinement method to be used and the mesh refinement accuracy tolerance, as well as the initial mesh. The structure *setup*.**mesh** contains the fields **method**, **tolerance**, **maxiteration**, and **phase**. The field *setup*.**mesh**.**method** is a string that specified the particular mesh refinement method to be used, while the field *setup*.**mesh**.**tolerance** contains the desired accuracy tolerance of the mesh, while the field *setup*.**mesh**.**maxiterations** contains the maximum number of allowed mes iterations.

Field	Possible Values	Default
setup.**mesh**.**method**	'hp' or 'hp1'	'hp1'
setup.**mesh**.**tolerance**	Positive Number Between 0 and 1	10^{-3}
setup.**mesh**.**maxiteration**	Non-Negative Integer	10

The field **mesh**.**phase** specifies the initial mesh intervals in a given phase and the number of collocation (Radau) points in each mesh interval. The field *setup*.**mesh**.**phase**(p).**fraction** contains the mesh intervals for each phase $p = 1, \ldots, P$, where the mesh intervals are specified in a row vector that provides the fraction of a scaled interval $[0, 1]$ that corresponds to each mesh interval. The field *setup*.**mesh**.**phase**(p).**colpoints** contains the number of collocation points in each phase $p = 1, \ldots, P$, where the number of collocation points in each mesh interval is also specified as a row vector such that the ith entry in *setup*.**mesh**.**phase**(p).**colpoints** corresponds to the ith entry in *setup*.**mesh**.**phase**(p).**fraction**.

Field	Possible Values	Default
setup.**mesh**.**phase**(p).**fraction**	Row Vector of Length $M \geq 1$ of Positive Numbers > 0 and < 1 that Sum to Unity	0.1*ones(1,10)
setup.**mesh**.**phase**(p).**colpoints**	Row Vector of Length $M \geq 1$ of Positive Integers > 1 and < 10 (M is the same as in *setup*.**mesh**.**phase**(p).**fraction**)	4*ones(1,10)

It is important to note that $\mathbb{GPOPS - II}$ has been designed so that the independent variable must be monotonically *increasing* in each phase of the problem.

2.2   Syntax for Structure *setup*.**functions**

The syntax for specifying the names of the MATLAB functions given in *setup*.**functions** given as follows:

$$
\begin{aligned}
setup.\textbf{functions}.\textbf{continuous} &= @continuousfun.m \\
setup.\textbf{functions}.\textbf{endpoint} &= @endpointfun.m
\end{aligned}
$$

The details of the syntax for each function are provided in Sections 2.4 and 2.5.

2.3   Syntax for bounds Structure

Once the user-defined structure *input* has been defined, the next step in setting up a problem for use with $\mathbb{GPOPS} - \mathbb{II}$ is to create the structure *input*.**bounds**. The structure **bounds** contains the following *three* fields: **phase**, **parameters**, and **eventgroup**. The field *input*.**bounds**.**phase** is an array of structures of length P (where P is the number of phases) that specifies the bounds on the time, state, control, path constraints, and integrals in each phase $p = 1, \ldots, P$ of the problem. The field *input*.**bounds**.**parameters** contains the lower and upper bounds on the static parameters in the problem. The field *input*.**bounds**.**eventgroup** is an array of structures of length G, where G is the number of event groups in the problem. The pth element in the array of structures *input*.**bounds**.**phase** contains the following fields:

- **bounds.phase(p).initialtime.lower** and **bounds.phase(p).initialtime.upper**: scalars that contain the information about the lower and upper bounds on the initial time in phase $p \in [1, \ldots, P]$. The scalars **bounds.phase(p).initialtime.lower** and **bounds.phase(p).initialtime.upper** have the following form:

$$\begin{aligned} \textbf{bounds.phase}(p)\textbf{.initialtime.lower} &= t_0^{\text{lower}} \\ \textbf{bounds.phase}(p)\textbf{.initialtime.upper} &= t_0^{\text{upper}} \end{aligned}$$

- **bounds.phase(p).finaltime.lower** and **bounds(p).finaltime.upper**: scalars that contain the information about the lower and upper bounds on the final time in phase $p \in [1, \ldots, P]$. The scalars **bounds.phase(p).finaltime.lower** and **bounds.phase(p).finaltime.upper** have the following form:

$$\begin{aligned} \textbf{bounds.phase}(p)\textbf{.finaltime.lower} &= t_f^{\text{lower}} \\ \textbf{bounds.phase}(p)\textbf{.finaltime.upper} &= t_f^{\text{upper}} \end{aligned}$$

- **bounds.phase(p).initialstate.lower** and **bounds.phase(p).initialstate.upper**: row vectors of length $n_y^{(p)}$ that contain the lower and upper bounds on the initial state in phase $p \in [1, \ldots, P]$. The row vectors **bounds.phase(p).initialstate.lower** and **bounds.phase(p).initialstate.upper** have the following form:

$$\begin{aligned} \textbf{bounds.phase}(p)\textbf{.initialstate.lower} &= \begin{bmatrix} y_{0,1}^{\text{lower}} & \cdots & y_{0,n_y^{(p)}}^{\text{lower}} \\ y_{0,1}^{\text{upper}} & \cdots & y_{0,n_y^{(p)}}^{\text{upper}} \end{bmatrix} \\ \textbf{bounds.phase}(p)\textbf{.initialstate.upper} &= \end{aligned}$$

- **bounds.phase(p).state.lower** and **bounds.phase(p).state.upper**: row vectors of length $n_y^{(p)}$ that contain the lower and upper bounds on the state during phase $p \in [1, \ldots, P]$. The row vectors **bounds.phase(p).state.lower** and **bounds.phase(p).state.upper** have the following form:

$$\begin{aligned} \textbf{bounds.phase}(p)\textbf{.state.lower} &= \begin{bmatrix} y_1^{\text{lower}} & \cdots & y_{n_y^{(p)}}^{\text{lower}} \\ y_1^{\text{upper}} & \cdots & y_{n_y^{(p)}}^{\text{upper}} \end{bmatrix} \\ \textbf{bounds.phase}(p)\textbf{.state.upper} &= \end{aligned}$$

- **bounds.phase(p).finalstate.lower** and **bounds.phase(p).finalstate.upper**: row vectors of length $n_y^{(p)}$ that contain the lower and upper bounds on the final state in phase $p \in [1, \ldots, P]$. The row vectors **bounds.phase(p).finalstate.lower** and **bounds.phase(p).finalstate.upper** have the following form:

$$\begin{aligned} \textbf{bounds.phase}(p)\textbf{.finalstate.lower} &= \begin{bmatrix} y_{f,1}^{\text{lower}} & \cdots & y_{f,n_y^{(p)}}^{\text{lower}} \\ y_{f,1}^{\text{upper}} & \cdots & y_{f,n_y^{(p)}}^{\text{upper}} \end{bmatrix} \\ \textbf{bounds.phase}(p)\textbf{.finalstate.upper} &= \end{aligned}$$

- **bounds.phase(p).control.lower** and **bounds.phase(p).control.upper**: row vectors of length $n_y^{(p)}$ that contain the lower and upper bounds on the control during phase $p \in [1, \ldots, P]$. The row vectors **bounds.phase(p).control.lower** and **bounds.phase(p).control.upper** have the following form:

$$\begin{aligned} \textbf{bounds.phase}(p)\textbf{.control.lower} &= \begin{bmatrix} u_1^{\text{lower}} & \cdots & u_{n_u^{(p)}}^{\text{lower}} \\ u_1^{\text{upper}} & \cdots & u_{n_u^{(p)}}^{\text{upper}} \end{bmatrix} \\ \textbf{bounds.phase}(p)\textbf{.control.upper} &= \end{aligned}$$

- **bounds.phase(p).path.lower** and **bounds.phase(p).path.upper**: row vectors of length $n_c^{(p)}$ that contain the lower and upper bounds on the path constraints during phase $p \in [1, \ldots, P]$. The row vectors **bounds.phase(p).path.lower** and **bounds.phase(p).path.upper** have the following form:

$$
\begin{aligned}
\textbf{bounds.phase(p).path.lower} &= \begin{bmatrix} c_1^{\text{lower}} & \cdots & c_{n_u^{(p)}}^{\text{lower}} \end{bmatrix} \\
\textbf{bounds.phase(p).path.upper} &= \begin{bmatrix} c_1^{\text{upper}} & \cdots & c_{n_u^{(p)}}^{\text{upper}} \end{bmatrix}
\end{aligned}
$$

- **bounds.phase(p).integral.lower** and **bounds.phase(p).integral.upper**: row vectors of length $n_q^{(p)}$ that contain the lower and upper bounds on the integrals in phase $p \in [1, \ldots, P]$. The row vectors **bounds.phase(p).integral.lower** and **bounds.phase(p).integral.upper** have the following form:

$$
\begin{aligned}
\textbf{bounds.phase(p).integral.lower} &= \begin{bmatrix} q_1^{\text{lower}} & \cdots & q_{n_q^{(p)}}^{\text{lower}} \end{bmatrix} \\
\textbf{bounds.phase(p).integral.upper} &= \begin{bmatrix} q_1^{\text{upper}} & \cdots & q_{n_q^{(p)}}^{\text{upper}} \end{bmatrix}
\end{aligned}
$$

- **bounds.parameters.lower** and **bounds.parameters.upper**: row vectors of length n_s that contain the lower and upper bounds on the static parameters in the problem. The row vectors **bounds.parameters.lower** and **bounds.parameters.upper** have the following form:

$$
\begin{aligned}
\textbf{bounds.parameters.lower} &= \begin{bmatrix} s_1^{\text{lower}} & \cdots & s_{n_s}^{\text{lower}} \end{bmatrix} \\
\textbf{bounds.parameters.upper} &= \begin{bmatrix} s_1^{\text{upper}} & \cdots & s_{n_s}^{\text{upper}} \end{bmatrix}
\end{aligned}
$$

- **bounds.eventgroup(g).lower** and **bounds.eventgroup(g).upper**: row vectors of length $n_b^{(g)}$ that contain the lower and upper bounds on the group $g = 1, \ldots, G$ of event constraints. The row vectors **bounds.eventgroup(g).lower** and **bounds.eventgroup(g).upper** have the following form:

$$
\begin{aligned}
\textbf{bounds.eventgroup(g).lower} &= \begin{bmatrix} b_1^{\text{lower}} & \cdots & b_{n_b^{(g)}}^{\text{lower}} \end{bmatrix} \\
\textbf{bounds.eventgroup(g).upper} &= \begin{bmatrix} b_1^{\text{upper}} & \cdots & b_{n_b^{(g)}}^{\text{upper}} \end{bmatrix}
\end{aligned}
$$

Note: any fields that do not apply to a problem (for example, a problem with no path constraints) should be omitted completely.

2.4 Syntax of Endpoint Function *setup*.**functions.endpoint**

The syntax used to evaluate the user-defined endpoint function defined by the function handle *setup*.**functions**.**endpoint** is given as follows:

function output=endpointfun(input)

The input *input* is a structure that contains the fields **phase**, **auxdata**, and **parameter** if the problem has parameters. The field *input*.**phase** is an array of structures of length P (where P is the number of phases) such that the pth element of *input*.**phase** contains the following fields:

- *input*.**phase(p).initialtime**: a scalar that contains the initial time in phase $p = 1, \ldots, P$;

- *input*.**phase(p).finaltime**: a scalar that contains the final time in phase $p = 1, \ldots, P$;

- *input*.**phase(p).initialstate**: a row vector of length $n_y^{(p)}$ that contains the initial state in phase $p = 1, \ldots, P$;

- *input*.**phase(p).finalstate**: a row vector of length $n_y^{(p)}$ that contains the final state in phase $p = 1, \ldots, P$;

- *input*.**phase(p).integral**: a row vector of length $n_d^{(p)}$ that contains the integrals in phase $p = 1, \ldots, P$;

- *input*.**parameter**: a row vector of length n_s that contains the static parameters in phase $p = 1, \ldots, P$;

The field *input*.**auxdata** contains the same information as the field *input*.**auxdata** that was specified in the structure *input* that was used to specify the information for the entire problem. The output *output* is a structure that contains the fields **objective** and **eventgroup**. The fields *output*.**objective** and *output*.**eventgroup** are given as follows:

- *output*.**objective**: a scalar that contains the result of computing the objective function on the current call to *input*.**functions.endpoint**;

- *output*.**eventgroup**: an array of structures of length G (where G is the number of event groups) such that the gth element in *output*.**eventgroup** is a row vector of length $n_b^{(g)}$ that contains the result of evaluating g^{th} group of event constraints at the values given in the call to the function *input*.**functions.endpoint**;

2.5   Syntax for Continuous Function *setup*.**functions.continuous**

The syntax used to evaluate the continuous functions defined by the function handle *setup*.**functions**.**continuous** is given as follows:

function output=continuousfun(input)

The input *input* is a structure that contains the fields **phase** and **auxdata**. The field *input*.**phase** is an array of structures of length P (where P is the number of phases) such that the pth element of *input*.**phase** contains the following fields:

- *input*.**phase**(p).time: a column vector of length $N^{(p)}$, where $N^{(p)}$ is the number of collocation points in phase $p = 1, \ldots, P$.

- *input*.**phase**(p).**time**: a matrix of size $N^{(p)} \times n_y^{(p)}$, where $N^{(p)}$ and $n_y^{(p)}$ are, respectively, the number of collocation points and the dimension of the state in phase $p = 1, \ldots, P$;

- *input*.**phase**(p).**control**: a matrix of size $N^{(p)} \times n_u^{(p)}$, where $N^{(p)}$ and $n_u^{(p)}$ are, respectively, the number of collocation points and the dimension of the control in phase $p = 1, \ldots, P$;

- *input*.**phase**(p).**parameter**: a matrix of size $N^{(p)} \times n_s$, where $N^{(p)}$ is the number of collocation points in phase $p = 1, \ldots, P$ and and n_s is the dimension of the static parameter. [**Note:** see below for the reason why the static parameter has a size $N^{(p)} \times n_s$];

Finally, *output* is an array of structures of length P (where P is the number of phases) such that the pth element of *output* contains the following fields:

- *output*.**dynamics**: a matrix of size $N^{(p)} \times n_y^{(p)}$, where $N^{(p)}$ and $n_y^{(p)}$ are, respectively, the number of collocation points and the dimension of the state in phase $p = 1, \ldots, P$;

- *output*.**path**: a matrix of size $N^{(p)} \times n_c^{(p)}$, where $N^{(p)}$ and $n_c^{(p)}$ are, respectively, the number of collocation points and the number of path constraints in phase $p = 1, \ldots, P$;

- *output*.**integrand**: a matrix of size $N^{(p)} \times n_d^{(p)}$, where $N^{(p)}$ and $n_d^{(p)}$ are, respectively, the number of collocation points and the number of integrals in phase $p = 1, \ldots, P$;

IMPORTANT NOTE: While it may seem a bit odd, the field *input*.**phase**(p).**parameter** is actually specified as if it were phase-dependent while it actually does not depend upon the phase because the static parameters themselves are independent of the phase. Furthermore, while the static parameters are defined as a single row vector, the arrays *input*.**phase**(p).**parameter** are actually matrices of size $N^{(p)} \times n_s$, where $N^{(p)}$ is the number of collocation points in each phase. The reason for making the static parameters phase dependent and providing them as an array with $N^{(p)}$ rows is to improve the efficiency with which the NLP derivatives are computed.

2.6   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 a then structure that contains the fields **phase** and **parameter**. Assume that $M^{(p)}$ is the number of values used in the guess for the time, state, and control in phase $p = 1, \ldots, P$. The field *setup*.**guess**.**phase** is an array of structures of length P such that the pth element of *setup*.**guess**.**phase** contains the following fields:

- *setup*.**guess**.**phase**(p).**time**: a column vector of length $M^{(p)}$ in phase $p = 1, \ldots, P$;

- *setup*.**guess**.**phase**(p).**state**: a matrix of size $M^{(p)} \times n_y^{(p)}$, where $n_y^{(p)}$ is the dimension of the state in phase $p = 1, \ldots, P$;

- *setup*.**guess**.**phase**(p).**control**: a matrix of size $M^{(p)} \times n_u^{(p)}$, where $n_u^{(p)}$ is the dimension of the control in phase $p = 1, \ldots, P$;

- *setup*.**guess**.**phase**(p).**integral**: a row vector of length $n_d^{(p)}$, where $n_d^{(p)}$ is the number of integrals in phase $p = 1, \ldots, P$;

- *setup*.**guess**.**parameter**: a row vector of length size n_s, where n_s is the number of static parameters in the problem.

It is noted that the column vector of time points specified in each phase $p = 1, \ldots, P$ in the field *setup*.**guess**.**phase**(p).**time** must be monotonically increasing.

2.7   Scaling of Optimal Control Problem

It is always preferable for the user to scale an optimal control problem of interest based on knowledge and insight of the problem. It is understood, however, that manually scaling a problem may be an iterative process and consume a great deal of time and effort. While it is beyond the scope of the software to provide a general procedure for scaling, in an attempt to reduce the burden on the user an automatic scaling procedure has been implemented in $\mathbb{GPOPS-II}$. This automatic scaling procedure is based on the method provided in[5] and scales both the NLP variables and NLP constraints using the information about the optimal control problem. First, using the user-supplied bounds on the time, state, control, and static parameters, the NLP variables are scaled to lie between -0.5 and 0.5. As a result, it is essential that the user provide *sensible* bounds on all quantities (that is, do *not* provide unreasonably large bounds as this will result in a poorly scaled problem). Next, the NLP constraints are scaled to make the row norms of the Jacobians of the functions in the optimal control problem near 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. Finally, the automatic scaling procedure in $\mathbb{GPOPS-II}$ can be employed simply by setting the field *setup*.**scales**.**method**='automatic-bounds'.

3   Output from an Execution of $\mathbb{GPOPS-II}$

The output of an execution of $\mathbb{GPOPS-II}$ is the structure *output*, where *output* contains the following fields:

- **result**: a structure that contains the following fields:

  - **solution**: the optimal time, state, and control, in each phase and the optimal value of the static parameter vector. The optimal time, state, and control are stored, respectively, in the fields **solution**.**phase**(p).**time**, **solution**.**phase**(p).**state**, and **solution**.**phase**(p).**control**, while the static parameter is stored in the field **solution**.**parameter**;

  - **objective**: the optimal value of the objective function of the optimal control problem;

- **result.setup**: the setup structure that produced the result found in **result** with $\mathbb{GPOPS-II}$;

- **meshhistory**: the solution and error estimate for each mesh on which the NLP was solved (only if mesh refinement is used);

- **meshiterations**: the number of mesh refinement iterations that were taken by $\mathbb{GPOPS} - \mathbb{II}$ (only if mesh refinement is used);

4 Useful Information for Debugging a $\mathbb{GPOPS} - \mathbb{II}$ Problem

One aspect of $\mathbb{GPOPS} - \mathbb{II}$ 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 $\mathbb{GPOPS} - \mathbb{II}$ 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 $\mathbb{GPOPS} - \mathbb{II}$) only evaluates the functions at the Legendre-Gauss-Radau points. Do not try to "fool" $\mathbb{GPOPS} - \mathbb{II}$ 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 $\mathbb{GPOPS} - \mathbb{II}$, see the references on the Radau pseudospectral method as given in the bibliography at the end of this document.

5 $\mathbb{GPOPS} - \mathbb{II}$ Examples

In this Chapter we provide seven examples of using $\mathbb{GPOPS} - \mathbb{II}$. Each of the examples are problems that have been studied extensively in the open literature and the solutions to these problems are well known. The first example is the hyper-sensitive optimal control problem from Ref. 6. The second example is a multiple-stage launch vehicle ascent problem taken from Refs. 7, 8, and 5. The third example is a tumor anti-angiogenesis optimal control problem from Refs. 9 and 5. The fourth example is the reusable launch vehicle entry problem taken from Ref. 5. The fifth example is the minimum time-to-climb of a supersonic aircraft taken from Refs. 10 and 5. The sixth example is the optimal control of a hang glider and is taken from Ref. 11. Finally, the seventh example is the optimal control of a two-strain tuberculosis model and is taken from Ref. 12. For each example the optimal control problem is described quantitatively, the $\mathbb{GPOPS} - \mathbb{II}$ code is provided, and the solution obtained using $\mathbb{GPOPS} - \mathbb{II}$ is provided. For reference, all examples were solved on a 2.5 GHz Core i7 MacBook Pro with 16 GB of RAM running Mac OS-X 10.7.5 (Lion). Finally, in the cases where IPOPT was used as the NLP solver, the IPOPT MATLAB mex files available on the IPOPT website were used. These IPOPT mex files were compiled with the linear solver MUMPS.

5.1 Hyper-Sensitive Problem

Consider the following *hyper-sensitive*[13, 6, 14, 15] optimal control problem adapted from Ref.[6] Minimize the cost functional

$$J = \frac{1}{2} \int_0^{t_f} (x^2 + u^2) dt \tag{7}$$

subject to the dynamic constraint

$$\dot{x} = -x^3 + u \tag{8}$$

and the boundary conditions

$$x(0) \quad = \quad 1.5 \quad , \quad x(t_f) \quad = \quad 1 \tag{9}$$

where t_f is fixed. It is known that for sufficiently large values of t_f that the solution to this example exhibits a so called "take-off", "cruise", and "landing" structure where the interesting behavior occurs near the initial and final time (see Ref.[6] for details). In particular, the "cruise" segment of this trajectory is constant (that is, the segment where the state and control are, interestingly, both zero) becomes an increasingly large percentage of the total trajectory time as t_f increases. Given the structure of the solution, one would expect

that the majority of collocation points would be placed in the "take-off" and "landing" segments while few collocation points would be placed in the "cruise" segment.

The hyper-sensitive optimal control problem of Eqs. (7)–(9) was solved using $\mathbb{GPOPS} - \mathbb{II}$ witht the NLP solver IPOPT and a mesh refinement tolerance $\epsilon = 10^{-7}$. In order to solve this problem using $\mathbb{GPOPS} - \mathbb{II}$, the continuous function, hyperSensitiveContinuous.m, was written to compute both the right-hand side of the differential equations and the integrand of the Lagrange cost. The result of integrating the integral specified in hyperSensitiveContinuous is then the field 'integral' of the structure *input* to the endpoint function hyperSensitiveEndpoint.m. The complete MATLAB code that was written to solve the hyper-sensitive optimal control problem of Eqs. (7)–(9) is given below.

```
%----------------------- Hyper-Sensitive Problem ------------------------%
% This example is taken from the following reference:               %
% Rao, A. V., and Mease, K. D., "Eigenvector Approximate Dichotomic Basis %
% Methods for Solving Hyper-Sensitive Optimal Control Problems," Optimal  %
% Control Applications and Methods, Vol. 21, No. 1., January-February,    %
% 2000, pp. 1-17.                                                   %
%------------------------------------------------------------------------%
clear all; clc


%------------------------------------------------------------------------%
%----------------- Provide All Bounds for Problem -----------------------%
%------------------------------------------------------------------------%
t0 = 0;
tf = 10000;
x0 = 1.5;
xf = 1;
xMin = -50;
xMax = +50;
uMin = -50;
uMax = +50;


%------------------------------------------------------------------------%
%----------------------- Setup for Problem Bounds -----------------------%
%------------------------------------------------------------------------%
bounds.phase.initialtime.lower = t0;
bounds.phase.initialtime.upper = t0;
bounds.phase.finaltime.lower = tf;
bounds.phase.finaltime.upper = tf;
bounds.phase.initialstate.lower = x0;
bounds.phase.initialstate.upper = x0;
bounds.phase.state.lower = xMin;
bounds.phase.state.upper = xMax;
bounds.phase.finalstate.lower = xf;
bounds.phase.finalstate.upper = xf;
bounds.phase.control.lower = uMin;
bounds.phase.control.upper = uMax;
bounds.phase.integral.lower = 0;
bounds.phase.integral.upper = 10000;


%------------------------------------------------------------------------%
%--------------------- Provide Guess of Solution ------------------------%
%------------------------------------------------------------------------%
guess.phase.time    = [t0; tf];
```

```
guess.phase.state    = [x0; xf];
guess.phase.control  = [0; 0];
guess.phase.integral = 0;


%------------------------------------------------------------------------%
%----------Provide Mesh Refinement Method and Initial Mesh --------------%
%------------------------------------------------------------------------%
mesh.method          = 'hp1';
mesh.tolerance       = 1e-6;
mesh.maxiteration    = 45;
mesh.colpointsmax    = 4;
mesh.colpointsmin    = 10;
mesh.phase.colpoints = 4*ones(1,10);
mesh.phase.fraction  =  0.1*ones(1,10);


%------------------------------------------------------------------------%
%------------- Assemble Information into Problem Structure --------------%
%------------------------------------------------------------------------%
setup.name = 'Hyper-Sensitive-Problem';
setup.functions.continuous = @hyperSensitiveContinuous;
setup.functions.endpoint = @hyperSensitiveEndpoint;
setup.bounds = bounds;
setup.guess = guess;
setup.mesh = mesh;
setup.nlp.solver = 'snopt';
setup.derivatives.supplier = 'sparseCD';
setup.derivatives.derivativelevel = 'second';
setup.method = 'RPMintegration';


%------------------------------------------------------------------------%
%--------------------- Solve Problem Using GPOPS2 -----------------------%
%------------------------------------------------------------------------%
output = gpops2(setup);

%-----------------------------------%
% BEGIN: hyperSensitiveContinuous.m %
%-----------------------------------%
function phaseout = hyperSensitiveContinuous(input)


t = input.phase.time;
x = input.phase.state;
u = input.phase.control;

xdot = -x.^3+u;
phaseout.dynamics = xdot;
phaseout.integrand = 0.5*(x.^2+u.^2);


%---------------------------------%
% END: hyperSensitiveContinuous.m %
%---------------------------------%

%---------------------------------%
% BEGIN: hyperSensitiveEndpoint.m %
```

```
%---------------------------------%
function output = hyperSensitiveEndpoint(input)

q  = input.phase.integral;
output.objective = q;
%---------------------------------%
% BEGIN: hyperSensitiveEndpoint.m %
%---------------------------------%
```

The state, $x(t)$, control, $u(t)$, and mesh refinement history that arise from the execution of $\mathbb{GPOPS-II}$ with the above code and the NLP solver IPOPT is summarized in Figs. 1a–1c, while a table showing the
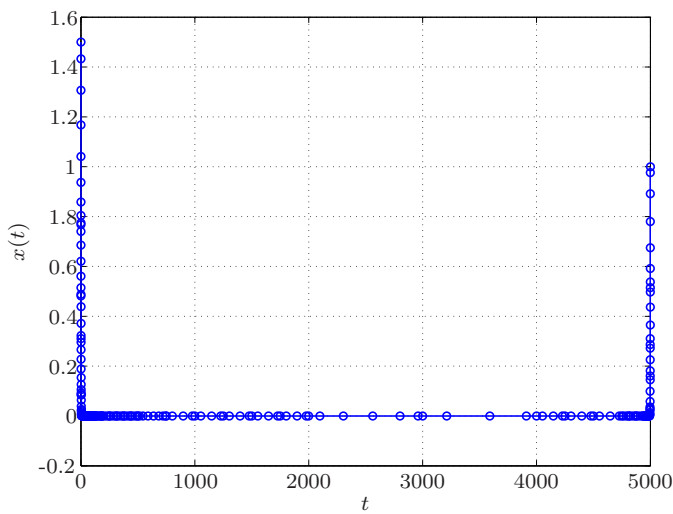
(a) $x(t)$ vs. t.
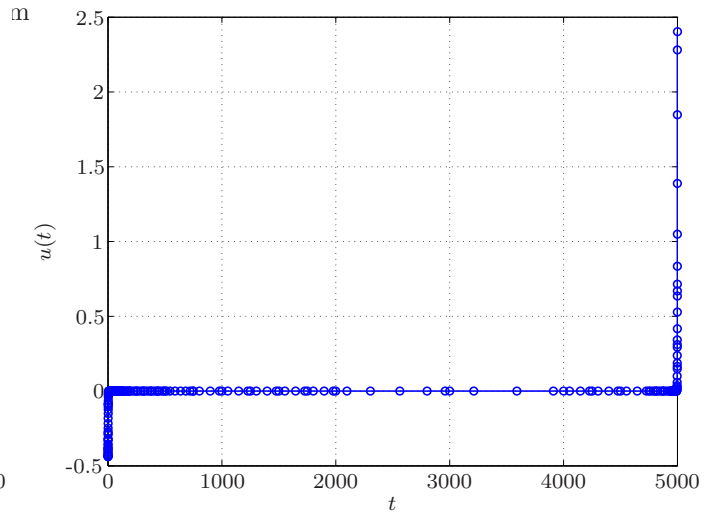
(b) $u(t)$ vs. t.
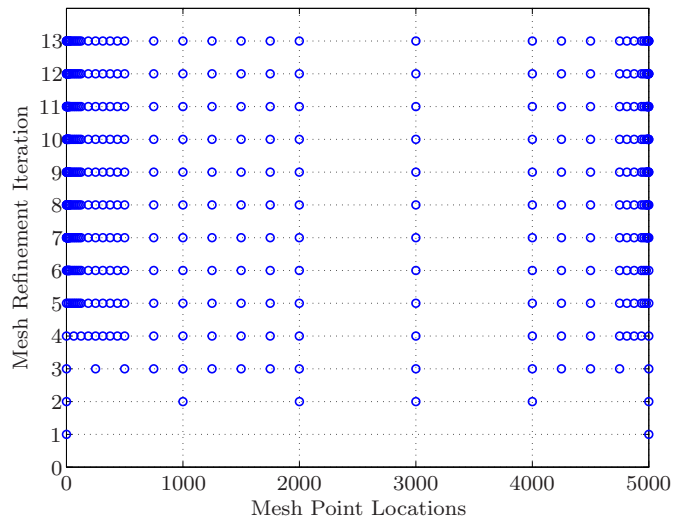
(c) Mesh Refinement History.

Figure 1: Solution to Hyper-Sensitive Problem Obtained Using $\mathbb{GPOPS-II}$ with the NLP Solver IPOPT and a Mesh Refinement Tolerance of 10^{-7}.

Table 1: Relative Error Estimate vs. Mesh Refinement Iteration for Hyper-Sensitive Problem.

Mesh Refinement Iteration	Relative Error Estimate
1	6.0129×10^{-1}
2	1.1855×10^{0}
3	2.9973×10^{-1}
4	7.6195×10^{-2}
5	1.7983×10^{-2}
6	4.7709×10^{-3}
7	6.0427×10^{-4}
8	2.6922×10^{-5}
9	7.6867×10^{-5}
10	3.2845×10^{-7}
11	1.1056×10^{-7}
12	4.9712×10^{-6}
13	4.0017×10^{-8}

5.2 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 is found verbatim in Refs. 7, 8, and 5.

5.2.1 Vehicle Properties

The goal of this launch vehicle ascent problem is to steer the vehicle from launch to a geostationary transfer orbit (GTO). The motion of the ehicle is divided into *four* distinct phases. Phase 1 starts with the vehicle on the ground and terminates when the fuel of the first set of solid rocket boosters is depleted. Upon termination of Phase 1 the first set of solid rocket boosters are dropped. Phase 2 starts where Phase 1 terminates and terminates when the fuel of the second set of solid rockets boosters is depleted. Phase 3 starts when Phase 2 terminates and termintes when the fuel of the first main engine fuel is depleted. Finally, Phase 4 starts where Phase 3 terminates and terminates when the vehicle reaches the final GTO. The vehicle data for this problem is taken verbatim from Ref. 8 or 5 and is shown in Table 2.

Table 2: Vehicle Properties for Multiple-Stage Launch Vehicle Ascent Problem.

	Solid Motors	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

5.2.2 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

$$
\begin{aligned}
\dot{\mathbf{r}} &= \mathbf{v} \\
\dot{\mathbf{v}} &= -\frac{\mu}{\|\mathbf{r}\|^3}\mathbf{r} + \frac{T}{m}\mathbf{u} + \frac{\mathbf{D}}{m} \\
\dot{m} &= -\frac{T}{g_0 I_{sp}}
\end{aligned}
\tag{10}
$$

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

$$
\mathbf{D} = -\frac{1}{2}C_D A_{ref}\rho\|\mathbf{v}_{\text{rel}}\|\mathbf{v}_{\text{rel}}
\tag{11}
$$

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

$$
\mathbf{v}_{\text{rel}} = \mathbf{v} - \boldsymbol{\omega} \times \mathbf{r}
\tag{12}
$$

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

$$
\rho = \rho_0 \exp[-h/h_0]
\tag{13}
$$

where ρ_0 is the atmospheric density at sea level, $h = \|\mathbf{r}\| - R_e$ is the altitude, R_e is the equatorial radius of the Earth, and h_0 is the density scale height. The numerical values for these constants can be found in Table 3.

Table 3: Constants used in the launch vehicle example.

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

5.2.3   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

$$
\begin{aligned}
\mathbf{r}(t_0) &= \mathbf{r}_0 = \begin{bmatrix} 5605.2 & 0 & 3043.4 \end{bmatrix}^T \quad \text{km} \\
\mathbf{v}(t_0) &= \mathbf{v}_0 = \begin{bmatrix} 0 & 0.4076 & 0 \end{bmatrix}^T \quad \text{km/s} \\
m(t_0) &= m_0 = 301454 \quad \text{kg}
\end{aligned}
\tag{14}
$$

The terminal constraints define the target geosynchronous transfer orbit (GTO), which is defined in orbital elements as

$$
\begin{aligned}
a_f &= 24361.14\,\text{km}, \\
e_f &= 0.7308, \\
i_f &= 28.5\,\text{deg}, \\
\Omega_f &= 269.8\,\text{deg}, \\
\omega_f &= 130.5\,\text{deg}
\end{aligned}
\tag{15}
$$

The orbital elements, a, e, i, Ω, and ω represent the semi-major axis, eccentricity, inclination, right ascension of the ascending node (RAAN), and argument of perigee, respectively. Note that the true anomaly, ν, is left undefined since the exact location within the orbit is not constrained. These orbital elements can be transformed into ECI coordinates via the transformation, T_{o2c}, where T_{o2c} is given in.[16]

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

$$
|\mathbf{r}| \geq R_r
\tag{16}
$$

where R_e is the radius of the Earth, as seen in Table 3. Next, a path constraint is imposed on the control to guarantee that the control vector is unit length, so that

$$
\|\mathbf{u}\|_2^2 = u_1^2 + u_2^2 + u_3^2 = 1
\tag{17}
$$

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

$$
\begin{aligned}
\mathbf{r}^{(p)}(t_f) - \mathbf{r}^{(p+1)}(t_0) &= \mathbf{0}, \\
\mathbf{v}^{(p)}(t_f) - \mathbf{v}^{(p+1)}(t_0) &= \mathbf{0}, \qquad (p = 1, \ldots, 3) \\
m^{(p)}(t_f) - m_{dry}^{(p)} - m^{(p+1)}(t_0) &= 0
\end{aligned}
\tag{18}
$$

where the superscript (p) represents the phase number.

The optimal control problem is then to find the control, **u**, that minimizes the cost function

$$J = -m^{(4)}(t_f) \tag{19}$$

subject to the conditions of Eqs. (10), (14), (15), (16), and (17).

The MATLAB code that solves the multiple-stage launch vehicle ascent problem using $\mathbb{GPOPS-II}$ 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 the following reference:                  %
% 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 No. 22, pp. 1-39.             %
% ----------------------------------------------------------------------%
clear all; clc

%------------------------------------------------------------------------%
%--------------- Provide All Physical Data for Problem -------------------%
%------------------------------------------------------------------------%
earthRadius        = 6378145;
gravParam          = 3.986012e14;
initialMass        = 301454;
earthRotRate       = 7.29211585e-5;
seaLevelDensity    = 1.225;
densityScaleHeight = 7200;
g0                 = 9.80665;

scales.length       = 1;
scales.speed        = 1;
scales.time         = 1;
scales.acceleration = 1;
scales.mass         = 1;
scales.force        = 1;
scales.area         = 1;
scales.volume       = 1;
scales.density      = 1;
scales.gravparam    = 1;

if 0,
scales.length       = earthRadius;
scales.speed        = sqrt(gravParam/scales.length);
scales.time         = scales.length/scales.speed;
scales.acceleration = scales.speed/scales.time;
scales.mass         = initialMass;
scales.force        = scales.mass*scales.acceleration;
scales.area         = scales.length^2;
```

```
scales.volume       = scales.area.*scales.length;
scales.density      = scales.mass/scales.volume;
scales.gravparam    = scales.acceleration*scales.length^2;
end

omega               = earthRotRate*scales.time;
auxdata.omegaMatrix = omega*[0 -1 0;1 0 0;0 0 0];
auxdata.mu          = gravParam/scales.gravparam;
auxdata.cd          = 0.5;
auxdata.sa          = 4*pi/scales.area;
auxdata.rho0        = seaLevelDensity/scales.density;
auxdata.H           = densityScaleHeight/scales.length;
auxdata.Re          = earthRadius/scales.length;
auxdata.g0          = g0/scales.acceleration;

lat0                = 28.5*pi/180;
x0                  = auxdata.Re*cos(lat0);
y0                  = 0;
z0                  = auxdata.Re*sin(lat0);
r0                  = [x0 y0 z0];
v0                  = r0*auxdata.omegaMatrix.';

btSrb = 75.2/scales.time;
btFirst = 261/scales.time;
btSecond = 700/scales.time;

t0 = 0/scales.time;
t1 = 75.2/scales.time;
t2 = 150.4/scales.time;
t3 = 261/scales.time;
t4 = 961/scales.time;

mTotSrb      = 19290/scales.mass;
mPropSrb     = 17010/scales.mass;
mDrySrb      = mTotSrb-mPropSrb;
mTotFirst    = 104380/scales.mass;
mPropFirst   = 95550/scales.mass;
mDryFirst    = mTotFirst-mPropFirst;
mTotSecond   = 19300/scales.mass;
mPropSecond  = 16820/scales.mass;
mDrySecond   = mTotSecond-mPropSecond;
mPayload     = 4164/scales.mass;
thrustSrb    = 628500/scales.force;
thrustFirst  = 1083100/scales.force;
thrustSecond = 110094/scales.force;
mdotSrb      = mPropSrb/btSrb;
ispSrb       = thrustSrb/(auxdata.g0*mdotSrb);
mdotFirst    = mPropFirst/btFirst;
ispFirst     = thrustFirst/(auxdata.g0*mdotFirst);
mdotSecond   = mPropSecond/btSecond;
ispSecond    = thrustSecond/(auxdata.g0*mdotSecond);

af = 24361140/scales.length;
```

```
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,auxdata.mu);
rout = rout';
vout = vout';


m10 = mPayload+mTotSecond+mTotFirst+9*mTotSrb;
m1f = m10-(6*mdotSrb+mdotFirst)*t1;
m20 = m1f-6*mDrySrb;
m2f = m20-(3*mdotSrb+mdotFirst)*(t2-t1);
m30 = m2f-3*mDrySrb;
m3f = m30-mdotFirst*(t3-t2);
m40 = m3f-mDryFirst;
m4f = mPayload;


auxdata.thrustSrb    = thrustSrb;
auxdata.thrustFirst  = thrustFirst;
auxdata.thrustSecond = thrustSecond;
auxdata.ispSrb       = ispSrb;
auxdata.ispFirst     = ispFirst;
auxdata.ispSecond    = ispSecond;


rmin = -2*auxdata.Re;
rmax = -rmin;
vmin = -10000/scales.speed;
vmax = -vmin;



%-----------------------------------------------------------------------%
%---------- Provide Bounds and Guess in Each Phase of Problem -----------%
%-----------------------------------------------------------------------%
iphase = 1;
bounds.phase(iphase).initialtime.lower = [t0];
bounds.phase(iphase).initialtime.upper = [t0];
bounds.phase(iphase).finaltime.lower = [t1];
bounds.phase(iphase).finaltime.upper = [t1];
bounds.phase(iphase).initialstate.lower = [r0(1:3),v0(1:3),m10];
bounds.phase(iphase).initialstate.upper = [r0(1:3),v0(1:3),m10];
bounds.phase(iphase).state.lower = [rmin*ones(1,3),vmin*ones(1,3),m1f];
bounds.phase(iphase).state.upper = [rmax*ones(1,3),vmax*ones(1,3),m10];
bounds.phase(iphase).finalstate.lower = [rmin*ones(1,3),vmin*ones(1,3),m1f];
bounds.phase(iphase).finalstate.upper = [rmax*ones(1,3),vmax*ones(1,3),m10];
bounds.phase(iphase).control.lower = -ones(1,3);
bounds.phase(iphase).control.upper = +ones(1,3);
bounds.phase(iphase).path.lower  = 1;
bounds.phase(iphase).path.upper  = 1;
```

```
guess.phase(iphase).time = [t0; t1];
guess.phase(iphase).state(:,1) = [r0(1); r0(1)];
guess.phase(iphase).state(:,2) = [r0(2); r0(2)];
guess.phase(iphase).state(:,3) = [r0(3); r0(3)];
guess.phase(iphase).state(:,4) = [v0(1); v0(1)];
guess.phase(iphase).state(:,5) = [v0(2); v0(2)];
guess.phase(iphase).state(:,6) = [v0(3); v0(3)];
guess.phase(iphase).state(:,7) = [m10; m1f];
guess.phase(iphase).control(:,1) = [0; 0];
guess.phase(iphase).control(:,2) = [1; 1];
guess.phase(iphase).control(:,3) = [0; 0];

iphase = 2;
bounds.phase(iphase).initialtime.lower = [t1];
bounds.phase(iphase).initialtime.upper = [t1];
bounds.phase(iphase).finaltime.lower = [t2];
bounds.phase(iphase).finaltime.upper = [t2];
bounds.phase(iphase).initialstate.lower = [rmin*ones(1,3),vmin*ones(1,3),m2f];
bounds.phase(iphase).initialstate.upper = [rmax*ones(1,3),vmax*ones(1,3),m20];
bounds.phase(iphase).state.lower = [rmin*ones(1,3),vmin*ones(1,3),m2f];
bounds.phase(iphase).state.upper = [rmax*ones(1,3),vmax*ones(1,3),m20];
bounds.phase(iphase).finalstate.lower = [rmin*ones(1,3),vmin*ones(1,3),m2f];
bounds.phase(iphase).finalstate.upper = [rmax*ones(1,3),vmax*ones(1,3),m20];
bounds.phase(iphase).control.lower = -ones(1,3);
bounds.phase(iphase).control.upper = +ones(1,3);
bounds.phase(iphase).path.lower  = 1;
bounds.phase(iphase).path.upper  = 1;
guess.phase(iphase).time = [t1; t2];
guess.phase(iphase).state(:,1) = [r0(1); r0(1)];
guess.phase(iphase).state(:,2) = [r0(2); r0(2)];
guess.phase(iphase).state(:,3) = [r0(3); r0(3)];
guess.phase(iphase).state(:,4) = [v0(1); v0(1)];
guess.phase(iphase).state(:,5) = [v0(2); v0(2)];
guess.phase(iphase).state(:,6) = [v0(3); v0(3)];
guess.phase(iphase).state(:,7) = [m10; m1f];
guess.phase(iphase).control(:,1) = [0; 0];
guess.phase(iphase).control(:,2) = [1; 1];
guess.phase(iphase).control(:,3) = [0; 0];

iphase = 3;
bounds.phase(iphase).initialtime.lower = [t2];
bounds.phase(iphase).initialtime.upper = [t2];
bounds.phase(iphase).finaltime.lower = [t3];
bounds.phase(iphase).finaltime.upper = [t3];
bounds.phase(iphase).initialstate.lower = [rmin*ones(1,3),vmin*ones(1,3),m3f];
bounds.phase(iphase).initialstate.upper = [rmax*ones(1,3),vmax*ones(1,3),m30];
bounds.phase(iphase).state.lower = [rmin*ones(1,3),vmin*ones(1,3),m3f];
bounds.phase(iphase).state.upper = [rmax*ones(1,3),vmax*ones(1,3),m30];
bounds.phase(iphase).finalstate.lower = [rmin*ones(1,3),vmin*ones(1,3),m3f];
bounds.phase(iphase).finalstate.upper = [rmax*ones(1,3),vmax*ones(1,3),m30];
bounds.phase(iphase).control.lower = -ones(1,3);
bounds.phase(iphase).control.upper = +ones(1,3);
bounds.phase(iphase).path.lower  = 1;
```

```
bounds.phase(iphase).path.upper  = 1;
guess.phase(iphase).time        = [t2; t3];
guess.phase(iphase).state(:,1) = [rout(1); rout(1)];
guess.phase(iphase).state(:,2) = [rout(2); rout(2)];
guess.phase(iphase).state(:,3) = [rout(3); rout(3)];
guess.phase(iphase).state(:,4) = [vout(1); vout(1)];
guess.phase(iphase).state(:,5) = [vout(2); vout(2)];
guess.phase(iphase).state(:,6) = [vout(3); vout(3)];
guess.phase(iphase).state(:,7) = [m30; m3f];
guess.phase(iphase).control(:,1) = [0; 0];
guess.phase(iphase).control(:,2) = [1; 1];
guess.phase(iphase).control(:,3) = [0; 0];

iphase = 4;
bounds.phase(iphase).initialtime.lower = [t3];
bounds.phase(iphase).initialtime.upper = [t3];
bounds.phase(iphase).finaltime.lower = [t3];
bounds.phase(iphase).finaltime.upper = [t4];
bounds.phase(iphase).initialstate.lower = [rmin*ones(1,3),vmin*ones(1,3),m4f];
bounds.phase(iphase).initialstate.upper = [rmax*ones(1,3),vmax*ones(1,3),m40];
bounds.phase(iphase).state.lower = [rmin*ones(1,3),vmin*ones(1,3),m4f];
bounds.phase(iphase).state.upper = [rmax*ones(1,3),vmax*ones(1,3),m40];
bounds.phase(iphase).finalstate.lower = [rmin*ones(1,3),vmin*ones(1,3),m4f];
bounds.phase(iphase).finalstate.upper = [rmax*ones(1,3),vmax*ones(1,3),m40];
bounds.phase(iphase).control.lower = -ones(1,3);
bounds.phase(iphase).control.upper = +ones(1,3);
bounds.phase(iphase).path.lower  = 1;
bounds.phase(iphase).path.upper  = 1;
guess.phase(iphase).time     = [t3; t4];
guess.phase(iphase).state(:,1) = [rout(1) rout(1)];
guess.phase(iphase).state(:,2) = [rout(2) rout(2)];
guess.phase(iphase).state(:,3) = [rout(3) rout(3)];
guess.phase(iphase).state(:,4) = [vout(1) vout(1)];
guess.phase(iphase).state(:,5) = [vout(2) vout(2)];
guess.phase(iphase).state(:,6) = [vout(3) vout(3)];
guess.phase(iphase).state(:,7) = [m40; m4f];
guess.phase(iphase).control(:,1) = [0; 0];
guess.phase(iphase).control(:,2) = [1; 1];
guess.phase(iphase).control(:,3) = [0; 0];

%-------------------------------------------------------------------------%
%------------- Set up Event Constraints That Link Phases -----------------%
%-------------------------------------------------------------------------%
bounds.eventgroup(1).lower = [zeros(1,6), -6*mDrySrb, 0];
bounds.eventgroup(1).upper = [zeros(1,6), -6*mDrySrb, 0];
bounds.eventgroup(2).lower = [zeros(1,6), -3*mDrySrb, 0];
bounds.eventgroup(2).upper = [zeros(1,6), -3*mDrySrb, 0];
bounds.eventgroup(3).lower = [zeros(1,6), -mDryFirst, 0];
bounds.eventgroup(3).upper = [zeros(1,6), -mDryFirst, 0];

%-------------------------------------------------------------------------%
%----------- Set up Event Constraints That Define Final Orbit -----------%
%-------------------------------------------------------------------------%
```

```matlab
bounds.eventgroup(4).lower = [af, ef, incf, Omf, omf];
bounds.eventgroup(4).upper = [af, ef, incf, Omf, omf];

%-------------------------------------------------------------------------%
%-------------- Provide an Initial Mesh in Each Phase --------------------%
%-------------------------------------------------------------------------%
for i=1:4
  meshphase(i).colpoints = 4*ones(1,10);
  meshphase(i).fraction = 0.1*ones(1,10);
end

%-------------------------------------------------------------------------%
%----------- Assemble All Information into Setup Structure ---------------%
%-------------------------------------------------------------------------%
setup.name = 'Launch-Vehicle-Ascent-Problem';
setup.functions.continuous = @launchContinuous;
setup.functions.endpoint = @launchEndpoint;
setup.mesh.phase = meshphase;
setup.nlp.solver = 'snopt';
setup.bounds = bounds;
setup.guess = guess;
setup.auxdata = auxdata;
setup.derivatives.supplier = 'sparseFD';
setup.derivatives.derivativelevel = 'second';
setup.derivatives.dependencies = 'sparseNaN';
setup.scales.method = 'automatic-bounds';
setup.mesh.method = 'hp1';
setup.mesh.tolerance = 1e-6;
setup.method = 'RPMintegration';

%-------------------------------------------------------------------------%
%---------------------- Solve Problem using GPOPS2 -----------------------%
%-------------------------------------------------------------------------%
totaltime = tic;
output = gpops2(setup);
totaltime = toc(totaltime);



%-------------------------------------------------------------------------%
%------------------ BEGIN Function launchContinuous.m -------------------%
%-------------------------------------------------------------------------%
function phaseout = launchContinuous(input)

%--------------------%
% Dynamics in Phase 1 %
%--------------------%
t1 = input.phase(1).time;
x1 = input.phase(1).state;
u1 = input.phase(1).control;
r1 = x1(:,1:3);
v1 = x1(:,4:6);
m1 = x1(:,7);
```

```
rad1 = sqrt(sum(r1.*r1,2));
omegaMatrix = input.auxdata.omegaMatrix;
omegacrossr = r1*omegaMatrix.';
vrel1 = v1-omegacrossr;
speedrel1 = sqrt(sum(vrel1.*vrel1,2));
h1 = rad1-input.auxdata.Re;
rho1 = input.auxdata.rho0*exp(-h1/input.auxdata.H);
bc1  = (rho1./(2*m1)).*input.auxdata.sa*input.auxdata.cd;
bcspeed1 = bc1.*speedrel1;
bcspeedmat1 = repmat(bcspeed1,1,3);
Drag1 = -bcspeedmat1.*vrel1;
muoverradcubed1 = input.auxdata.mu./rad1.^3;
muoverradcubedmat1 = [muoverradcubed1 muoverradcubed1 muoverradcubed1];
grav1 = -muoverradcubedmat1.*r1;

TSrb1   = 6*input.auxdata.thrustSrb*ones(size(t1));
TFirst1 = input.auxdata.thrustFirst*ones(size(t1));
TTot1   = TSrb1+TFirst1;
m1dot1  = -TSrb1./(input.auxdata.g0*input.auxdata.ispSrb);
m2dot1  = -TFirst1./(input.auxdata.g0*input.auxdata.ispFirst);
mdot1   = m1dot1+m2dot1;

path1 = [sum(u1.*u1,2)];
Toverm1 = TTot1./m1;
Tovermmat1 = [Toverm1 Toverm1 Toverm1];
thrust1 = Tovermmat1.*u1;
rdot1 = v1;
vdot1 = thrust1+Drag1+grav1;
phaseout(1).dynamics = [rdot1 vdot1 mdot1];
phaseout(1).path = path1;

%---------------------%
% Dynamics in Phase 2 %
%--------------------%
t2 = input.phase(2).time;
x2 = input.phase(2).state;
u2 = input.phase(2).control;
r2 = x2(:,1:3);
v2 = x2(:,4:6);
m2 = x2(:,7);

rad2 = sqrt(sum(r2.*r2,2));
omegaMatrix = input.auxdata.omegaMatrix;
omegacrossr = r2*omegaMatrix.';
vrel2 = v2-omegacrossr;
speedrel2 = sqrt(sum(vrel2.*vrel2,2));
h2 = rad2-input.auxdata.Re;
rho2 = input.auxdata.rho0*exp(-h2/input.auxdata.H);
bc2  = (rho2./(2*m2)).*input.auxdata.sa*input.auxdata.cd;
bcspeed2 = bc2.*speedrel2;
bcspeedmat2 = repmat(bcspeed2,1,3);
Drag2 = -bcspeedmat2.*vrel2;
muoverradcubed2 = input.auxdata.mu./rad2.^3;
```

```
muoverradcubedmat2 = [muoverradcubed2 muoverradcubed2 muoverradcubed2];
grav2 = -muoverradcubedmat2.*r2;
TSrb2 = 3*input.auxdata.thrustSrb*ones(size(t2));
TFirst2 = input.auxdata.thrustFirst*ones(size(t2));
TTot2 = TSrb2+TFirst2;
m1dot2 = -TSrb2./(input.auxdata.g0*input.auxdata.ispSrb);
m2dot2 = -TFirst2./(input.auxdata.g0*input.auxdata.ispFirst);
mdot2 = m1dot2+m2dot2;
path2 = [sum(u2.*u2,2)];
Toverm2 = TTot2./m2;
Tovermmat2 = [Toverm2 Toverm2 Toverm2];
thrust2 = Tovermmat2.*u2;
rdot2 = v2;
vdot2 = thrust2+Drag2+grav2;
phaseout(2).dynamics = [rdot2 vdot2 mdot2];
phaseout(2).path = path2;

%--------------------%
% Dynamics in Phase 3 %
%--------------------%
t3 = input.phase(3).time;
x3 = input.phase(3).state;
u3 = input.phase(3).control;
r3 = x3(:,1:3);
v3 = x3(:,4:6);
m3 = x3(:,7);

rad3 = sqrt(sum(r3.*r3,2));
omegaMatrix = input.auxdata.omegaMatrix;
omegacrossr = r3*omegaMatrix.';
vrel3 = v3-omegacrossr;
speedrel3 = sqrt(sum(vrel3.*vrel3,2));
h3 = rad3-input.auxdata.Re;
rho3 = input.auxdata.rho0*exp(-h3/input.auxdata.H);
bc3  = (rho3./(2*m3)).*input.auxdata.sa*input.auxdata.cd;
bcspeed3 = bc3.*speedrel3;
bcspeedmat3 = repmat(bcspeed3,1,3);
Drag3 = -bcspeedmat3.*vrel3;
muoverradcubed3 = input.auxdata.mu./rad3.^3;
muoverradcubedmat3 = [muoverradcubed3 muoverradcubed3 muoverradcubed3];
grav3 = -muoverradcubedmat3.*r3;
TTot3 = input.auxdata.thrustFirst*ones(size(t3));
mdot3 = -TTot3./(input.auxdata.g0*input.auxdata.ispFirst);
path3 = [sum(u3.*u3,2)];
Toverm3 = TTot3./m3;
Tovermmat3 = [Toverm3 Toverm3 Toverm3];
thrust3 = Tovermmat3.*u3;
rdot3 = v3;
vdot3 = thrust3+Drag3+grav3;
phaseout(3).dynamics = [rdot3 vdot3 mdot3];
phaseout(3).path = path3;

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

```
% Dynamics in Phase 4 %
%---------------------%
t4 = input.phase(4).time;
x4 = input.phase(4).state;
u4 = input.phase(4).control;
r4 = x4(:,1:3);
v4 = x4(:,4:6);
m4 = x4(:,7);
rad4 = sqrt(sum(r4.*r4,2));
omegacrossr = r4*input.auxdata.omegaMatrix.';
vrel4 = v4-omegacrossr;
speedrel4 = sqrt(sum(vrel4.*vrel4,2));
h4 = rad4-input.auxdata.Re;
rho4 = input.auxdata.rho0*exp(-h4/input.auxdata.H);
bc4  = (rho4./(2*m4)).*input.auxdata.sa*input.auxdata.cd;
bcspeed4 = bc4.*speedrel4;
bcspeedmat4 = repmat(bcspeed4,1,3);
Drag4 = -bcspeedmat4.*vrel4;
muoverradcubed4 = input.auxdata.mu./rad4.^3;
muoverradcubedmat4 = [muoverradcubed4 muoverradcubed4 muoverradcubed4];
grav4 = -muoverradcubedmat4.*r4;
TTot4 = input.auxdata.thrustSecond*ones(size(t4));
mdot4 = -TTot4/(input.auxdata.g0*input.auxdata.ispSecond);
path4 = [sum(u4.*u4,2)];
Toverm4 = TTot4./m4;
Tovermmat4 = [Toverm4 Toverm4 Toverm4];
thrust4 = Tovermmat4.*u4;
rdot4 = v4;
vdot4 = thrust4+Drag4+grav4;
phaseout(4).dynamics = [rdot4 vdot4 mdot4];
phaseout(4).path = path4;

%------------------------------------------------------------------------------%
%--------------------- END Function launchContinuous.m ------------------------%
%------------------------------------------------------------------------------%


%------------------------------------------------------------------------------%
%--------------------- BEGIN Function launchEndpoint.m ------------------------%
%------------------------------------------------------------------------------%
function output = launchEndpoint(input)

% Variables at Start and Terminus of Phase 1
t0{1} = input.phase(1).initialtime;
tf{1} = input.phase(1).finaltime;
x0{1} = input.phase(1).initialstate;
xf{1} = input.phase(1).finalstate;
% Variables at Start and Terminus of Phase 2
t0{2} = input.phase(2).initialtime;
tf{2} = input.phase(2).finaltime;
x0{2} = input.phase(2).initialstate;
xf{2} = input.phase(2).finalstate;
% Variables at Start and Terminus of Phase 3
t0{3} = input.phase(3).initialtime;
```

```matlab
tf{3} = input.phase(3).finaltime;
x0{3} = input.phase(3).initialstate;
xf{3} = input.phase(3).finalstate;
% Variables at Start and Terminus of Phase 2
t0{4} = input.phase(4).initialtime;
tf{4} = input.phase(4).finaltime;
x0{4} = input.phase(4).initialstate;
xf{4} = input.phase(4).finalstate;

% Event Group 1:  Linkage Constraints Between Phases 1 and 2
output.eventgroup(1).event = [x0{2}(1:7)-xf{1}(1:7), t0{2}-tf{1}];
% Event Group 2:  Linkage Constraints Between Phases 2 and 3
output.eventgroup(2).event = [x0{3}(1:7)-xf{2}(1:7), t0{3}-tf{2}];
% Event Group 3:  Linkage Constraints Between Phases 3 and 4
output.eventgroup(3).event = [x0{4}(1:7)-xf{3}(1:7), t0{4}-tf{3}];
% Event Group 4:  Constraints on Terminal Orbit
orbitalElements = launchrv2oe(xf{4}(1:3).',xf{4}(4:6).',input.auxdata.mu);
output.eventgroup(4).event = orbitalElements(1:5).';
output.objective = -xf{4}(7);

%-------------------------------------------------------------------------%
%-------------------- END Function launchEndpoint.m --------------------%
%-------------------------------------------------------------------------%


%-------------------------------------------------------------------------%
%------------------- BEGIN Function launchrv2oe.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;
e  = sqrt(ev.'*ev);
a  = p/(1-e*e);
i  = acos(hv(3)/sqrt(h2));
Om = acos(nv(1)/n);
if (nv(2)<0-eps),
  Om = 2*pi-Om;
end;
om = acos(nv.'*ev/n/e);
if (ev(3)<0),
  om = 2*pi-om;
end;
nu = acos(ev.'*rv/e/r);
if (rv.'*vv<0),
  nu = 2*pi-nu;
end;
```

```
oe = [a; e; i; Om; om; nu];

%------------------------------------------------------------------------%
%--------------------- END Function launchrv2oe.m ----------------------%
%------------------------------------------------------------------------%



%------------------------------------------------------------------------%
%------------------ BEGIN Function launchoe2rv.m -----------------------%
%------------------------------------------------------------------------%
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;

%------------------------------------------------------------------------%
%--------------------- END Function launchoe2rv.m ----------------------%
%------------------------------------------------------------------------%
```

The output of the above code from $\mathbb{GPOPS-II}$ is summarized in the following three plots that contain the altitude, speed, and controls.
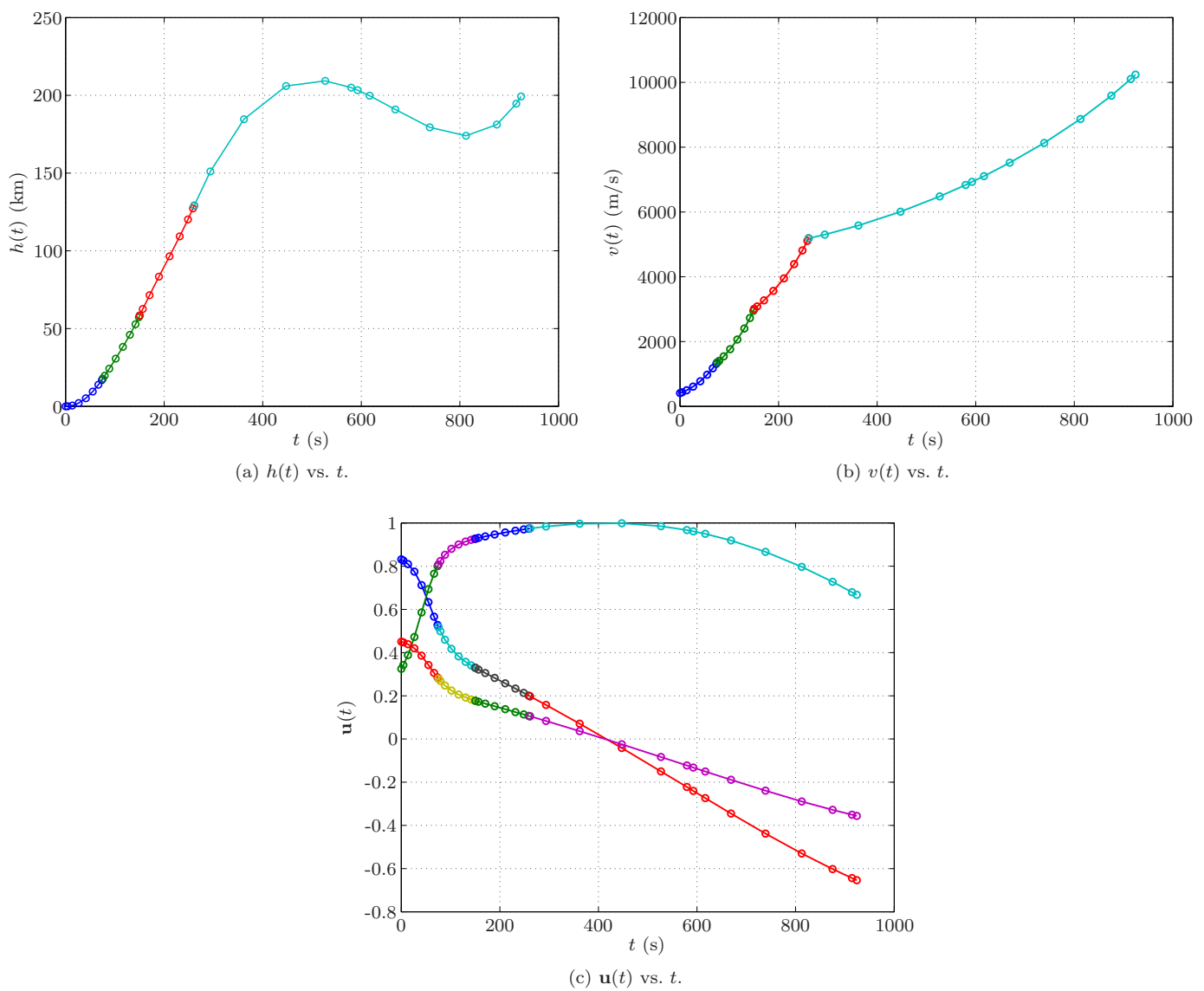
(a) $h(t)$ vs. t.

(b) $v(t)$ vs. t.

(c) $\mathbf{u}(t)$ vs. t.

Figure 2: Solution to Launch Vehicle Ascent Problem Using $\mathbb{GPOPS-II}$ with the NLP Solver SNOPT and a Mesh Refinement Tolerance of 10^{-7}.

5.3   Tumor-Antiangiogenesis Optimal Control Problem

Consider the following cancer treatment optimal control problem taken from Ref. 9. The objective is to minimize

$$p(t_f) \tag{20}$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{p}(t) &= -\xi p(t) \ln\left(\frac{p(t)}{q(t)}\right), \\
\dot{q}(t) &= q(t)\left[b - \mu - dp^{2/3}(t) - Gu(t)\right],
\end{aligned}
\tag{21}
$$

with the initial conditions

$$
\begin{aligned}
p(0) &= p_0, \\
q(0) &= q_0,
\end{aligned}
\tag{22}
$$

and the integral constraint

$$\int_0^{t_f} u(\tau)d\tau \le A. \tag{23}$$

This problem describes a treatment process called anti-angiogenesis where it is desired to reverse the direction of growth of a tumor by cutting of the blood supply to the tumor. The code for solving this problem is shown below.

```
%--------------- Tumor Anti-Angiogenesis Problem -------------------%
% This example is taken from the following reference:              %
% Ledzewicz, U. and Schattler, H, "Analysis of Optimal Controls for a %
% Mathematical Model of Tumour Anti-angiogenesis," Optimal Control   %
% Applications and Methods, Vol. 29, 2008, pp. 41-57.              %
%------------------------------------------------------------------%
clear all
clc

% Parameters:
%------------------------------------------------------------------%
%------------------- Data Required by Problem -------------------%
%------------------------------------------------------------------%
auxdata.zeta = 0.084;       % per day
auxdata.b = 5.85;           % per day
auxdata.d = 0.00873;        % per mm^2 per day
auxdata.G = 0.15;           % per mg of dose per day
auxdata.mu = 0.02;          % per day
a = 75;
A = 15;


%------------------------------------------------------------------%
%--------------------- Boundary Conditions ---------------------%
%------------------------------------------------------------------%
pMax = ((auxdata.b-auxdata.mu)/auxdata.d)^(3/2);
pMin = 0.1;
qMax = pMax;
qMin = pMin;
yMax = A;
yMin = 0;
uMax = a;
uMin = 0;
```

```
t0Max = 0;
t0Min = 0;
tfMax = 5;
tfMin = 0.1;
p0 = pMax/2;
q0 = qMax/4;
y0 = 0;

bounds.phase.initialtime.lower = t0Min;
bounds.phase.initialtime.upper = t0Max;
bounds.phase.finaltime.lower = tfMin;
bounds.phase.finaltime.upper = tfMax;
bounds.phase.initialstate.lower = [p0, q0];
bounds.phase.initialstate.upper = [p0, q0];
bounds.phase.state.lower = [pMin, qMin];
bounds.phase.state.upper = [pMax, qMax];
bounds.phase.finalstate.lower = [pMin, qMin];
bounds.phase.finalstate.upper = [pMax, qMax];
bounds.phase.control.lower = uMin;
bounds.phase.control.upper = uMax;
bounds.phase.integral.lower = [0];
bounds.phase.integral.upper = [A];
guess.phase.time    = [0; 1];
guess.phase.state   = [[p0; pMax],[q0; qMax]];
guess.phase.control = [uMax; uMax];
guess.phase.integral = [7.5];


%-------------------------------------------------------------------------%
%------------------------- Problem Setup ------------------------%
%-------------------------------------------------------------------------%
setup.name = 'Tumor-Anti-Angiogenesis-Problem';
setup.functions.continuous = @tumorAntiAngiogenesisContinuous;
setup.functions.endpoint = @tumorAntiAngiogenesisEndpoint;
setup.auxdata = auxdata;
setup.bounds = bounds;
setup.guess = guess;
setup.nlp.solver = 'snopt';
setup.derivatives.supplier = 'sparseCD';
setup.derivatives.derivativelevel = 'second';
setup.scales.method = 'automatic-bounds';
setup.mesh.method = 'hp1';
setup.mesh.tolerance = 1e-6; % default 1e-3
setup.mesh.phase.colpoints = 4*ones(1,10);
setup.mesh.phase.fraction = 0.1*ones(1,10);


%-------------------------------------------------------------------------%
%------------------- Solve Problem Using GPOPS2 -------------------%
%-------------------------------------------------------------------------%
output = gpops2(setup);
output.result.nlptime
solution = output.result.solution;
```

```
function phaseout = tumorAntiAngiogenesisContinuous(input)

zeta = input.auxdata.zeta;
b    = input.auxdata.b;
mu   = input.auxdata.mu;
d    = input.auxdata.d;
G    = input.auxdata.G;


t = input.phase(1).time;
x = input.phase(1).state;
u = input.phase(1).control;


p = x(:,1);
q = x(:,2);
pdot = -zeta.*p.*log(p./q);
qdot = q.*(b-(mu+(d*(p.^(2/3)))+G.*u));

phaseout.dynamics = [pdot qdot];
phaseout.integrand = u;

function output = tumorAntiAngiogenesisEndpoint(input)

pf = input.phase.finalstate(1);
output.objective = pf;

```

The solution obtained using $\mathbb{GPOPS}-\mathbb{II}$ using the NLP solver IPOPT with a mesh refinement error tolerance of 10^{-6} is shown in Figs. 3a–3c. Note that in this example we have also provided the costate of the optimal control problem, where the costate is estimated using the Radau pseudospectral costate estimation method described in Refs. 1, 2, and 3.

(a) $(p(t), q(t))$ vs. t.

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

(c) $(\lambda_p(t), \lambda_q(t), \lambda_y(t))$ vs. t.
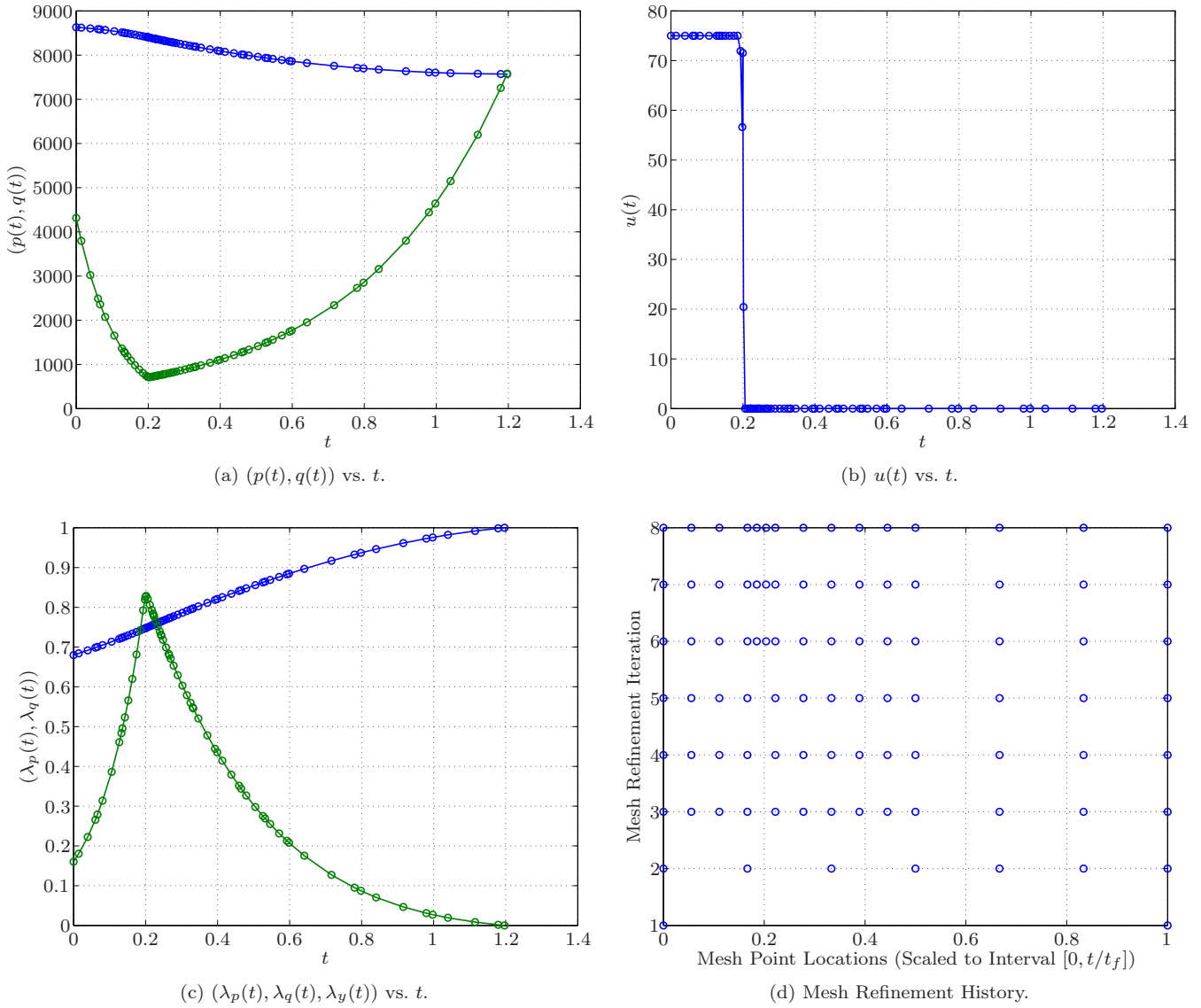

(d) Mesh Refinement History.

Figure 3: Solution to Tumor Anti-Angiogenesis Optimal Control Problem Using $\mathbb{GPOPS-II}$ with the NLP Solver SNOPT and a Mesh Refinement Tolerance of 10^{-6}.

5.4   Reusable Launch Vehicle Entry

Consider the following optimal control problem of maximizing the crossrange during the atmospheric entry of a reusable launch vehicle and taken verbatim from Ref.[5] Minimize the cost functional

$$J = -\phi(t_f) \tag{24}$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{r} &= v \sin\gamma, \\
\dot{\theta} &= \frac{v \cos\gamma \sin\psi}{r \cos\phi}, \\
\dot{\phi} &= \frac{v \cos\gamma \cos\psi}{r}, \\
\dot{v} &= -\frac{F_d}{m} - F_g \sin\gamma, \\
\dot{\gamma} &= \frac{F_l \cos\sigma}{mv} - \left(\frac{F_g}{v} - \frac{v}{r}\right)\cos\gamma, \\
\dot{\psi} &= \frac{F_l \sin\sigma}{mv \cos\gamma} + \frac{v \cos\gamma \sin\psi \tan\phi}{r},
\end{aligned}
\tag{25}
$$

and the boundary conditions

$$
\begin{array}{rclcrcl}
r(0) &=& 79248 + R_e \text{ m} &,& r(t_f) &=& 24384 + R_e \text{ m}, \\
\theta(0) &=& 0 \text{ deg} &,& \theta(t_f) &=& \text{Free}, \\
\phi(0) &=& 0 \text{ deg} &,& \phi(t_f) &=& \text{Free}, \\
v(0) &=& 7803 \text{ m/s} &,& v(t_f) &=& 762 \text{ m/s}, \\
\gamma(0) &=& -1 \text{ deg} &,& \gamma(t_f) &=& -5 \text{ deg}, \\
\psi(0) &=& 90 \text{ deg} &,& \psi(t_f) &=& \text{Free}.
\end{array}
\tag{26}
$$

Further details of this problem, including the aerodynamic model, can be found in Ref. 5. The code for solving this problem is shown below.

```
% ----------- Reusable Launch Vehicle Entry Example ------------%
% This example is taken verbatim from the following reference:  %
% Betts, J. T., Practical Methods for Optimal Control Using     %
% Nonlinear Programming, SIAM Press, Philadelphia, 2009.        %
% -------------------------------------------------------------%
%close all
clear all
clc

cft2m = 0.3048;
cft2km = cft2m/1000;
cslug2kg = 14.5939029;
%------------------------------------%
%              Problem Setup         %
%------------------------------------%
auxdata.Re = 20902900*cft2m;              % Equatorial Radius of Earth (m)
auxdata.S  = 2690*cft2m^2;                % Vehicle Reference Area (m^2)
auxdata.cl(1) = -0.2070;                  % Parameters for lift coefficient
auxdata.cl(2) = 1.6756;
auxdata.cd(1) = 0.0785;
auxdata.cd(2) = -0.3529;
```

```
auxdata.cd(3) = 2.0400;
auxdata.b(1)  = 0.07854;
auxdata.b(2)  = -0.061592;
auxdata.b(3)  = 0.00621408;
auxdata.H     = 23800*cft2m;                  % Density Scale Height (m)
auxdata.al(1) = -0.20704;
auxdata.al(2) = 0.029244;
auxdata.rho0  = 0.002378*cslug2kg/cft2m^3;% Sea Level Atmospheric Density (slug/ft^3)
auxdata.mu    = 1.4076539e16*cft2m^3;     % Earth Gravitational Parameter (ft^^3/s^2)
auxdata.mass  = 6309.433*cslug2kg;


% inital conditions
t0 = 0;
alt0 = 260000*cft2m;
rad0 = alt0+auxdata.Re;
lon0 = 0;
lat0 = 0;
speed0 = 25600*cft2m;
fpa0   = -1*pi/180;
azi0   = 90*pi/180;


% terminal conditions
altf = 80000*cft2m;
radf = altf+auxdata.Re;
speedf = 2500*cft2m;
fpaf   = -5*pi/180;
azif   = -90*pi/180;


%-------------------------------------------------------%
% Lower and Upper Limits on Time, State, and Control %
%-------------------------------------------------------%
tfMin = 0;            tfMax = 3000;
radMin = auxdata.Re;  radMax = rad0;
lonMin = -pi;         lonMax = -lonMin;
latMin = -70*pi/180;  latMax = -latMin;
speedMin = 10;        speedMax = 45000;
fpaMin = -80*pi/180;  fpaMax =  80*pi/180;
aziMin = -180*pi/180; aziMax =  180*pi/180;
aoaMin = -90*pi/180;  aoaMax = -aoaMin;
bankMin = -90*pi/180; bankMax =   1*pi/180;


bounds.phase.initialtime.lower = t0;
bounds.phase.initialtime.upper = t0;
bounds.phase.finaltime.lower = tfMin;
bounds.phase.finaltime.upper = tfMax;
bounds.phase.initialstate.lower = [rad0, lon0, lat0, speed0, fpa0, azi0];
bounds.phase.initialstate.upper = [rad0, lon0, lat0, speed0, fpa0, azi0];
bounds.phase.state.lower = [radMin, lonMin, latMin, speedMin, fpaMin, aziMin];
bounds.phase.state.upper = [radMax, lonMax, latMax, speedMax, fpaMax, aziMax];
bounds.phase.finalstate.lower = [radf, lonMin, latMin, speedf, fpaf, aziMin];
bounds.phase.finalstate.upper = [radf, lonMax, latMax, speedf, fpaf, aziMax];
bounds.phase.control.lower = [aoaMin, bankMin];
bounds.phase.control.upper = [aoaMax, bankMax];
```

```
%----------------------%
% Set up Initial Guess %
%----------------------%
tGuess = [0; 1000];
radGuess = [rad0; radf];
lonGuess = [lon0; lon0+10*pi/180];
latGuess = [lat0; lat0+10*pi/180];
speedGuess = [speed0; speedf];
fpaGuess = [fpa0; fpaf];
aziGuess = [azi0; azif];
aoaGuess = [0; 0];
bankGuess = [0; 0];

guess.phase.state   = [radGuess, lonGuess, latGuess, speedGuess, fpaGuess, aziGuess];
guess.phase.control = [aoaGuess, bankGuess];
guess.phase.time    = tGuess;

%---------------------%
% Set up Initial Mesh %
%---------------------%
meshphase.colpoints = 4*ones(1,10);
meshphase.fraction = 0.1*ones(1,10);

setup.name = 'Reusable-Launch-Vehicle-Entry-Problem';
setup.functions.continuous = @rlvEntryContinuous;
setup.functions.endpoint   = @rlvEntryEndpoint;
setup.auxdata = auxdata;
setup.mesh.phase = meshphase;
setup.bounds = bounds;
setup.guess = guess;
setup.nlp.solver = 'ipopt';
setup.derivatives.supplier = 'sparseCD';
setup.derivatives.derivativelevel = 'second';
setup.scales.method = 'automatic-bounds';
%setup.method = 'RPMintegration';
setup.mesh.method = 'hp1';
setup.mesh.tolerance = 1e-6; % default 1e-3
setup.mesh.colpointsmin = 4;
setup.mesh.colpointsmax = 16;

%---------------------------------%
% Solve Problem Using OptimalPrime %
%---------------------------------%
output = gpops2(setup);

function phaseout = rlvEntryContinuous(input)

% input
% input.phase(phasenumber).state
% input.phase(phasenumber).control
% input.phase(phasenumber).time
% input.phase(phasenumber).parameter
%
```

```
% input.auxdata = auxiliary information
%
% output
% phaseout(phasenumber).dynamics
% phaseout(phasenumber).path
% phaseout(phasenumber).integrand

rad = input.phase.state(:,1);
lon = input.phase.state(:,2);
lat = input.phase.state(:,3);
speed = input.phase.state(:,4);
fpa = input.phase.state(:,5);
azimuth = input.phase.state(:,6);
aoa = input.phase.control(:,1);
bank = input.phase.control(:,2);


cd0 = input.auxdata.cd(1);
cd1 = input.auxdata.cd(2);
cd2 = input.auxdata.cd(3);
cl0 = input.auxdata.cl(1);
cl1 = input.auxdata.cl(2);
mu  = input.auxdata.mu;
rho0 = input.auxdata.rho0;
H = input.auxdata.H;
S = input.auxdata.S;
mass = input.auxdata.mass;
altitude = rad - input.auxdata.Re;


CD = cd0+cd1*aoa+cd2*aoa.^2;


rho = rho0*exp(-altitude/H);
CL = cl0+cl1*aoa;
gravity = mu./rad.^2;
dynamic_pressure = 0.5*rho.*speed.^2;
D = dynamic_pressure.*S.*CD./mass;
L = dynamic_pressure.*S.*CL./mass;
slon = sin(lon);
clon = cos(lon);
slat = sin(lat);
clat = cos(lat);
tlat = tan(lat);
sfpa = sin(fpa);
cfpa = cos(fpa);
sazi = sin(azimuth);
cazi = cos(azimuth);
cbank = cos(bank);
sbank = sin(bank);


raddot   = speed.*sfpa;
londot   = speed.*cfpa.*sazi./(rad.*clat);
latdot   = speed.*cfpa.*cazi./rad;
speeddot = -D-gravity.*sfpa;
fpadot   = (L.*cbank-cfpa.*(gravity-speed.^2./rad))./speed;
```

```
azidot   = (L.*sbank./cfpa + speed.^2.*cfpa.*sazi.*tlat./rad)./speed;

phaseout.dynamics  = [raddot, londot, latdot, speeddot, fpadot, azidot];

function output = rlvEntryEndpoint(input)

% Inputs
% input.phase(phasenumber).initialstate -- row
% input.phase(phasenumber).finalstate -- row
% input.phase(phasenumber).initialtime -- scalar
% input.phase(phasenumber).finaltime -- scalar
% input.phase(phasenumber).integral -- row
%
% input.parameter -- row

% input.auxdata = auxiliary information

% Output
% output.objective -- scalar
% output.eventgroup(eventnumber).event -- row
latf = input.phase.finalstate(3);

% cost
output.objective = -latf;
```

This example was solved using $\mathbb{GPOPS-II}$ using the NLP solver IPOPT with a mesh refinement tolerance of 10^{-6} and the solution is shown in Figs. 4a–4f.
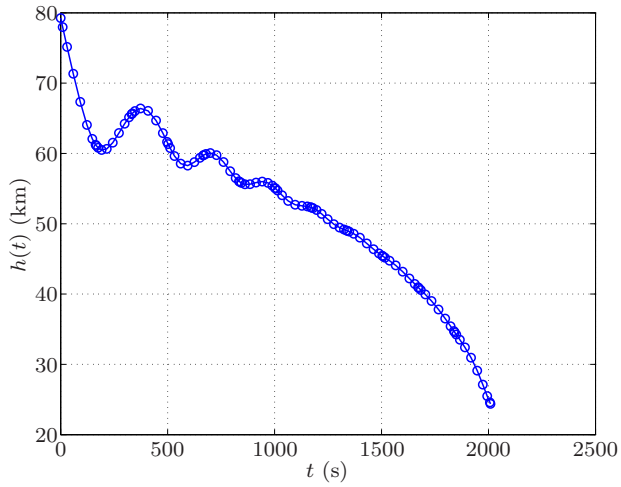
(a) $h(t)$ vs. t.
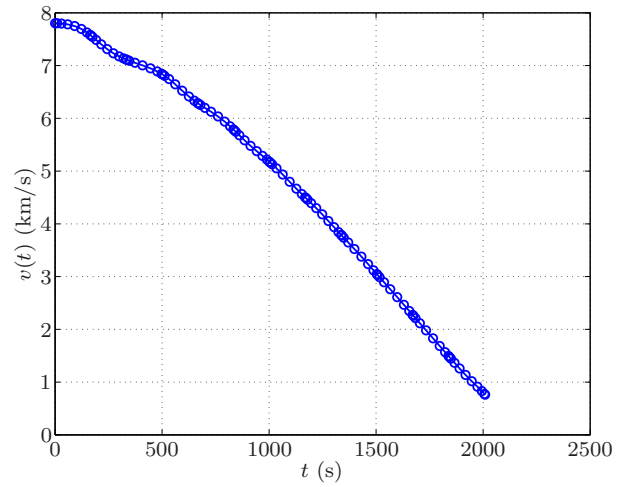
(b) $v(t)$ vs. t.
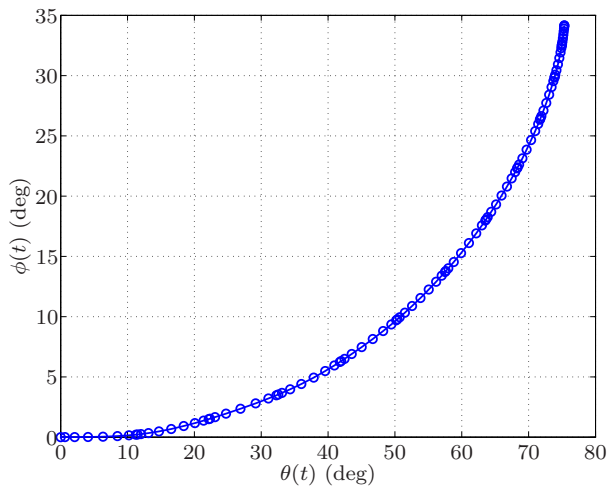
(c) $\phi(t)$ vs. $\theta(t)$.
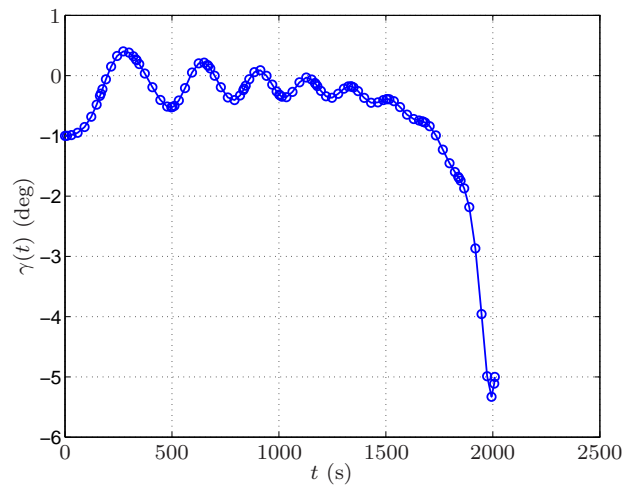
(d) $\gamma(t)$ vs. t.
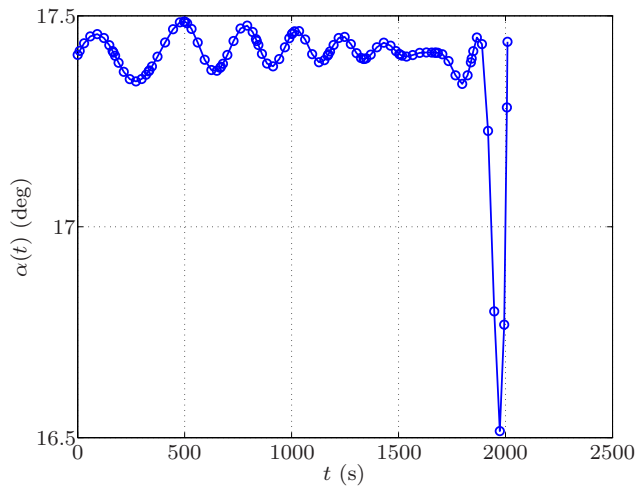
(e) $\alpha(t)$ vs. t.
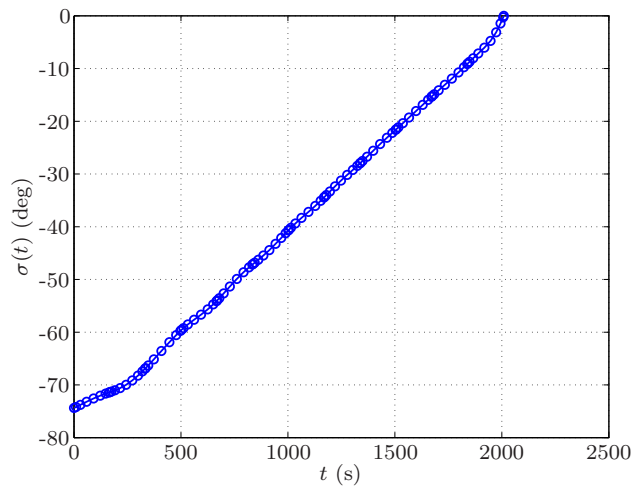
(f) $\sigma(t)$ vs. t.

Figure 4: Solution to Reusable Launch Vehicle Entry Problem Using $\mathbb{GPOPS} - \mathbb{II}$ with the NLP Solver IPOPT and a Mesh Refinement Tolerance of 10^{-6}.

5.5   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. 17, but the model used in this study was taken from Ref. 5 with the exception that a linear extrapolation of the thrust data as found in Ref. 5 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

$$J = t_f \tag{27}$$

subject to the dynamic constraints

$$
\begin{align}
\dot{h} &= v \sin\alpha \tag{28} \\
\dot{v} &= \frac{T\cos\alpha - D}{m} \tag{29} \\
\dot{\gamma} &= \frac{T\sin\alpha + L}{mv} + \left(\frac{v}{r} - \frac{\mu}{vr^2}\right)\cos\gamma \tag{30} \\
\dot{m} &= -\frac{T}{g_0 I_{sp}} \tag{31}
\end{align}
$$

and the boundary conditions

$$
\begin{align}
h(0) &= 0 \text{ ft} \tag{32} \\
v(0) &= 129.3144 \text{ m/s} \tag{33} \\
\gamma(0) &= 0 \text{ rad} \tag{34} \\
h(t_f) &= 19994.88 \text{ m} \tag{35} \\
v(t_f) &= 295.092 \text{ ft/s} \tag{36} \\
\gamma(t_f) &= 0 \text{ rad} \tag{37}
\end{align}
$$

where h is the altitude, v is the speed, γ 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. 17. In this example $\mathbb{GPOPS-II}$ 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:      %
% Bryson, A. E., Desai, M. N. and Hoffman, W. C., "Energy-State     %
% Approximation in Performance Optimization of Supersonic           %
% Aircraft," Journal of Aircraft, Vol. 6, No. 6, November-December, %
% 1969, pp. 481-488.                                                %
%------------------------------------------------------------------%
clear all
close all
clc


%------------------------------------------------------------------%
%---------- Initialize all of the data for the problem ------------%
%------------------------------------------------------------------%
load minimumTimeToClimbAeroData.mat;
```

```
%------------------------------------------------------------------------%
%----------------- U.S. 1976 Standard Atmosphere  ------------------%
%------------------------------------------------------------------------%
% Format of Data:                                                        %
%   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];
```

```
%-------------------------------------------------------------------------%
%--------------- Propulsion Data for Bryson Aircraft --------------%
%-------------------------------------------------------------------------%
% The thrust depends for the aircraft considered by Bryson in 1969  %
% depends upon the Mach number and the altitude.  This data is taken%
% verbatim from the 1969 Journal of Aircraft paper (see reference   %
% above) and is copied for use in this example.  The data are stored%
% in the following variables:                                       %
%    - Mtab: a vector of values of Mach number                      %
%    - alttab: a vector of altitude values                          %
%    - Ttab: is a table of aircraft thrust values                   %
% After conversion, the altitude given in meters.                   %
% After conversion, the thrust given in Newtons.                    %
%-------------------------------------------------------------------------%
Mtab   = [0; 0.2; 0.4; 0.6; 0.8; 1; 1.2; 1.4; 1.6; 1.8];
alttab = 304.8*[0 5 10 15 20 25 30 40 50 70];
Ttab   = 4448.222*[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];

%-------------------------------------------------------------------------%
%-------------- Aerodynamic Data for Bryson Aircraft --------------%
%-------------------------------------------------------------------------%
%  M2 is a vector of Mach number values                             %
%  Clalphatab is a vector of coefficient of lift values             %
%  CD0tab is a vector of zero-lift coefficient of drag values       %
%   - etatab is a vector of load factors                            %
%-------------------------------------------------------------------------%
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];


%-------------------------------------------------------------------------%
%---- All Data Required by User Functions is Stored in AUXDATA -----%
%-------------------------------------------------------------------------%
auxdata.CDdat    = CDdat;
auxdata.CLdat    = CLdat;
auxdata.etadat   = etadat;
auxdata.M        = Mtab;
auxdata.M2       = M2;
auxdata.alt      = alttab;
auxdata.T        = Ttab;
```

```
auxdata.Clalpha   = Clalphatab;
auxdata.CD0       = CD0tab;
auxdata.eta       = etatab;
auxdata.ppCLalpha = polyfit(auxdata.M2,auxdata.Clalpha,8);
auxdata.ppCD0     = polyfit(auxdata.M2,auxdata.CD0,8);
auxdata.ppeta     = polyfit(auxdata.M2,auxdata.eta,8);
auxdata.Re        = 6378145;
auxdata.mu        = 3.986e14;
auxdata.S         = 49.2386;
auxdata.g0        = 9.80665;
auxdata.Isp       = 1600;
auxdata.H         = 7254.24;
auxdata.rho0      = 1.225;
auxdata.us1976    = us1976;
[aa,mm]           = meshgrid(alttab,Mtab);
auxdata.aa        = aa;
auxdata.mm        = mm;

%-----------------------------------------------------------------------%
%--------------------- Boundary Conditions ----------------------%
%-----------------------------------------------------------------------%
t0     = 0;          % Initial time (sec)
alt0   = 0;          % Initial altitude (meters)
altf   = 19994.88;   % Final altitude (meters)
speed0 = 129.314;    % Initial speed (m/s)
speedf = 295.092;    % Final speed (m/s)
fpa0   = 0;          % Initial flight path angle (rad)
fpaf   = 0;          % Final flight path angle (rad)
mass0  = 19050.864;  % Initial mass (kg)

%-----------------------------------------------------------------------%
%---------------------- Limits on Variables ----------------------%
%-----------------------------------------------------------------------%
tfmin    = 100;         tfmax    = 800;
altmin   = 0;           altmax   = 21031.2;
speedmin = 5;           speedmax = 1000;
fpamin   = -40*pi/180;  fpamax   = 40*pi/180;
massmin  = 22;          massmax  = 20410;
alphamin = -pi/4;       alphamax = pi/4;

iphase = 1;
%-----------------------------------------------------------------------%
%--------------- Set Up Problem Using Data Provided Above ----------%
%-----------------------------------------------------------------------%
%mesh(iphase).colpoints                 = 20;
bounds.phase(iphase).initialtime.lower  = t0;
bounds.phase(iphase).initialtime.upper  = t0;
bounds.phase(iphase).finaltime.lower    = tfmin;
bounds.phase(iphase).finaltime.upper    = tfmax;
bounds.phase(iphase).initialstate.lower = [alt0, speed0, fpa0, mass0];
bounds.phase(iphase).initialstate.upper = [alt0, speed0, fpa0, mass0];
bounds.phase(iphase).state.lower        = [altmin, speedmin, fpamin, massmin];
bounds.phase(iphase).state.upper        = [altmax, speedmax, fpamax, massmax];
```

```
bounds.phase(iphase).finalstate.lower   = [altf, speedf, fpaf, massmin];
bounds.phase(iphase).finalstate.upper   = [altf, speedf, fpaf, massmax];
bounds.phase(iphase).control.lower      = alphamin;
bounds.phase(iphase).control.upper      = alphamax;
guess.phase(iphase).time                = [0; 1000];
guess.phase(iphase).state(:,1)          = [alt0; altf];
guess.phase(iphase).state(:,2)          = [speed0; speedf];
guess.phase(iphase).state(:,3)          = [fpa0; fpaf];
guess.phase(iphase).state(:,4)          = [mass0; mass0];
guess.phase(iphase).control             = [20; -20]*pi/180;


%------------------------------------------------------------------%
%---------- Configure Setup Using the information provided ---------%
%------------------------------------------------------------------%
setup.name                          = 'Bryson-Minimum-Time-to-Climb-Problem';
setup.functions.continuous          = 'minimumTimeToClimbContinuous';
setup.functions.endpoint            = 'minimumTimeToClimbEndpoint';
%setup.mesh                           = mesh;
setup.nlp.solver                    = 'snopt';
setup.bounds                        = bounds;
setup.guess                         = guess;
setup.auxdata                       = auxdata;
setup.derivatives.supplier          = 'sparseCD';
setup.derivatives.derivativelevel   = 'first';
setup.scales.method                 = 'automatic-bounds';
setup.mesh.method                   = 'hp1';
setup.mesh.tolerance                = 1e-7;
setup.mesh.colpointsmax             = 4;
setup.mesh.colpointsmin             = 16;
setup.method                        = 'RPMintegration';


%------------------------------------------------------------------%
%------------------- Solve Problem Using GPOPS2 -------------------%
%------------------------------------------------------------------%
output = gpops2(setup);



%-----------------------------------------------%
% Begin Function:  minimumTimeToClimbContinuous.m %
%-----------------------------------------------%
function phaseout = minimumTimeToClimbContinuous(input)

CONSTANTS = input.auxdata;
us1976 = CONSTANTS.us1976;
Ttab = CONSTANTS.T;

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

x = input.phase(1).state;
```

```
u = input.phase(1).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]=minimumTimeToClimbCompute(Mach,CONSTANTS);
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);

phaseout.dynamics = [hdot, vdot, fpadot, mdot];

%----------------------------------------------------%
% End Function:  minimumTimeToClimbContinuous.m    %
%----------------------------------------------------%

%----------------------------------------------------%
% Begin Function:  minimumTimeToClimbEndpoint.m %
%----------------------------------------------------%
function output = brysonMinimumClimbEndpoint(input);

tf = input.phase(1).finaltime;
output.objective = tf;
%----------------------------------------------------%
% End Function:  minimumTimeToClimbEndpoint.m    %
%----------------------------------------------------%

%----------------------------------------------------%
% Begin Function:  minimumTimeToClimbCompute.m %
%----------------------------------------------------%
function [CD,CL,eta]=minimumTimeToClimbCompute(Mach,CONSTANTS)

CDdat  = CONSTANTS.CDdat;
CLdat  = CONSTANTS.CLdat;
etadat = CONSTANTS.etadat;

ii     = find(Mach>=0.8);
jj     = find(Mach<0.8);
mpoly  = Mach(ii);
CD  = zeros(length(Mach),1);
```

```
CL = zeros(length(Mach),1);
eta = zeros(length(Mach),1);
if ~isempty(ii)
    CD(ii)  = ppval(CDdat,mpoly);
    CL(ii)  = ppval(CLdat,mpoly);
    eta(ii) = ppval(etadat,mpoly);
end

if ~isempty(jj)
    CD(jj)  = 0.013*ones(length(jj),1);
    CL(jj)  = 3.44*ones(length(jj),1);
    eta(jj) = 0.54*ones(length(jj),1);
end

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

The components of the state and the control obtained from running the above $\mathbb{GPOPS} - \mathbb{II}$ code is summarized in Figs. 5a–5d.

Table 4: Relative Error Estimate vs. Mesh Refinement Iteration for Minimum Time-to-Climb Problem.

Mesh Refinement Iteration	Relative Error Estimate
1	5.776×10^{-3}
2	2.3717×10^{-3}
3	3.0679×10^{-5}
4	6.2216×10^{-6}
5	8.861×10^{-6}
6	2.3224×10^{-6}
7	1.3708×10^{-6}
8	3.8553×10^{-6}
9	5.1621×10^{-6}
10	7.0515×10^{-6}
11	2.5598×10^{-6}
12	1.0775×10^{-6}
13	7.8122×10^{-7}

(a) $h(t)$ vs. t.

(b) $h(t)$ vs. $v(t)$.

(c) $\gamma(t)$ vs. t.

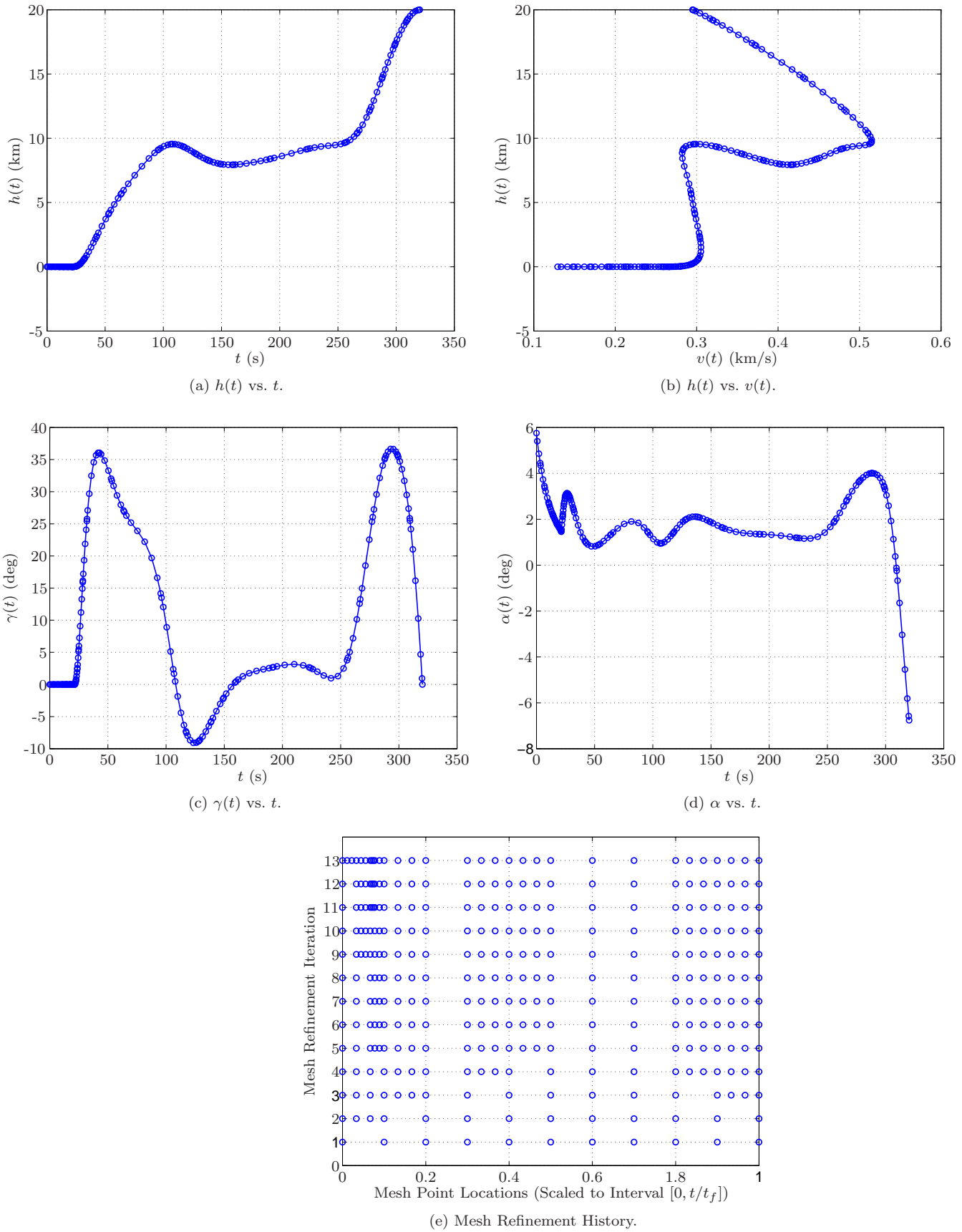(d) α vs. t.

(e) Mesh Refinement History.

Figure 5: Solution to Minimum Time-to-Climb Problem Using $\mathbb{GPOPS} - \mathbb{II}$ with the NLP Solver SNOPT and a Mesh Refinement Tolerance of 10^{-6}.

Dynamic Soaring Problem

The following optimal control problem considers optimizing the motion of a hang glider in the presence of known wind force. The probem was originally described in Ref.[11] and the problem considered here is identical to that of Ref.[11] The objective is to minimize the average wind gradient slope β, that is, minimize

$$J = \beta \tag{38}$$

subject to the hang glider dynamics

$$
\begin{array}{llll}
\dot{x} &= v\cos\gamma\sin\psi + W_x &, & m\dot{v} &= -D - mg\sin\gamma - m\dot{W}_x\cos\gamma\sin\psi \\
\dot{y} &= v\cos\gamma\cos\psi &, & mv\dot{\gamma} &= L\cos\sigma - mg\cos\gamma + m\dot{W}_x\sin\gamma\sin\psi, \\
\dot{h} &= v\sin\gamma &, & mv\cos\gamma\dot{\psi} &= L\sin\sigma - m\dot{W}_x\cos\psi
\end{array}
\tag{39}
$$

and the boundary conditions

$$
\begin{array}{lll}
(x(0), y(0), h(0)) &=& (x(t_f), y(t_f), h(t_f)) = (0,0,0), \\
(v(t_f) - v(0), \gamma(t_f) - \gamma(0), \psi(t_f) + 2\pi - \psi(0)) &=& (0,0,0),
\end{array}
\tag{40}
$$

where W_x is the wind component along the East direction, m is the glider mass, v is the air-relative speed, ψ is the azimuth angle (measured clockwise from the North), γ is the air-relative flight path angle, h is the altitude, (x, y) are (East, North) position, σ is the glider bank angle, D is the drag force, and L is the lift force. The drag and lift forces are computed using a standard drag polar aerodynamic model

$$
\begin{array}{ll}
D = & qSC_D, \\
L = & qSC_L,
\end{array}
\tag{41}
$$

where $q = \rho v^2/2$ is the dynamic pressure, S is the vehicle reference area, $C_D = C_{D0} + KC_L^2$ is the coefficient of drag, and C_L is the coefficient of lift (where $0 \leq C_L \leq C_{L,\max}$). The constants for this problem ar taken verbatim from Ref. 11 and are given as $C_{D0} = 0.00873$, $K = 0.045$, and $C_{L,\max} = 1.5$. Finally, it is noted that C_L and σ are the controls.

   This example was posed in English units, but was solved using the automatic scaling procedure in $\mathbb{GPOPS - II}$ with the NLP solver IPOPT using second sparse finite-difference approximations for the NLP derivatives and with a mesh refinement tolerance of 10^{-7}. The code used to solve this problem is shown below and the solution to this problem is shown in Fig. 6.

```
%------------------- Dynamic Soaring Problem ----------------------------%
% This example is taken from the following reference:                   %
% Zhao, Y. J., "Optimal Pattern of Glider Dynamic Soaring," Optimal      %
% Control Applications and Methods, Vol. 25, 2004, pp. 67-89.           %
%-----------------------------------------------------------------------%
clear all
clc

auxdata.rho=0.002378;
auxdata.CD0 = 0.00873;
auxdata.K= 0.045;
auxdata.g=32.2;
auxdata.m=5.6;
auxdata.S=45.09703;
auxdata.mu = 3.986e14;

auxdata.mgos=auxdata.m*auxdata.g/auxdata.S;
auxdata.Emax=(1/(4*auxdata.K*auxdata.CD0))^0.5;
auxdata.W0=0;
```

```
auxdata.lmin = -2;
auxdata.lmax = 5;

x0 = 0;
y0 = 0;
z0 = 0;
r0 = 0;
rf = 0;
v0=100;
v0=100;


xmin      = -1000;
xmax      = +1000;
ymin      = -1000;
ymax      = +1000;
zmin      =   0;
zmax      = +1000;
vmin      = +10;
vmax      = +350;
gammamin = -75*pi/180;
gammamax =   75*pi/180;
psimin    = -3*pi;
psimax    = +pi/2;
betamin   = 0.005;
betamax   = 0.15;
CLmin     = -0.5;
CLmax     = 1.5;
Phimin    = -75/180*pi;
Phimax    =   75/180*pi;
t0        = 0;
tfmin     = 1;
tfmax     = 30;

% Phase 1 Information
iphase = 1;
bounds.phase(iphase).initialtime.lower  = t0;
bounds.phase(iphase).initialtime.upper  = t0;
bounds.phase(iphase).finaltime.lower    = tfmin;
bounds.phase(iphase).finaltime.upper    = tfmax;
bounds.phase(iphase).initialstate.lower = [x0, y0, z0, vmin, gammamin, psimin];
bounds.phase(iphase).initialstate.upper = [x0, y0, z0, vmax, gammamax, psimax];
bounds.phase(iphase).state.lower        = [xmin, ymin, zmin, vmin, gammamin, psimin];
bounds.phase(iphase).state.upper        = [xmax, ymax, zmax, vmax, gammamax, psimax];
bounds.phase(iphase).finalstate.lower   = [x0, y0, z0, vmin, gammamin, psimin];
bounds.phase(iphase).finalstate.upper   = [x0, y0, z0, vmax, gammamax, psimax];
bounds.phase(iphase).control.lower      = [CLmin, Phimin];
bounds.phase(iphase).control.upper      = [CLmax, Phimax];
bounds.phase(iphase).path.lower         = auxdata.lmin;
bounds.phase(iphase).path.upper         = auxdata.lmax;
bounds.eventgroup(1).lower              = zeros(1,3);
bounds.eventgroup(1).upper              = zeros(1,3);
bounds.parameter.lower                  = betamin;
bounds.parameter.upper                  = betamax;
```

```
N                           = 100;
CL0                         = CLmax;
basetime                    = linspace(0,24,N).';
xguess                      = 500*cos(2*pi*basetime/24)-500;
yguess                      = 300*sin(2*pi*basetime/24);
zguess                      = -400*cos(2*pi*basetime/24)+400;
vguess                      = 0.8*v0*(1.5+cos(2*pi*basetime/24));
gammaguess                  = pi/6*sin(2*pi*basetime/24);
psiguess                    = -1-basetime/4;
CLguess                     = CL0*ones(N,1)/3;
phiguess                    = -ones(N,1);
betaguess                   = 0.08;
guess.phase(iphase).time    = [basetime];
guess.phase(iphase).state   = [xguess, yguess, zguess, vguess, gammaguess, psiguess];
guess.phase(iphase).control = [CLguess, phiguess];
guess.parameter             = [betaguess];

setup.name                        = 'Dynamic-Soaring-Problem';
setup.functions.continuous        = @dynamicSoaringContinuous;
setup.functions.endpoint          = @dynamicSoaringEndpoint;
setup.nlp.solver                  = 'snopt';
setup.bounds                      = bounds;
setup.guess                       = guess;
setup.auxdata                     = auxdata;
setup.derivatives.supplier        = 'sparseCD';
setup.derivatives.derivativelevel = 'second';
setup.scales.method               = 'automatic-bounds';
%setup.method = 'RPMintegration';
setup.mesh.method          = 'hp1';
setup.mesh.tolerance = 1e-6;

output = gpops2(setup);
solution = output.result.solution;

%-------------------------------------------%
% BEGIN: function dynamicSoaringContinuous.m %
%-------------------------------------------%
function phaseout = dynamicSoaringContinuous(input)

t         = input.phase(1).time;
s         = input.phase(1).state;
u         = input.phase(1).control;
p         = input.phase(1).parameter;
x         = s(:,1);
y         = s(:,2);
z         = s(:,3);
v         = s(:,4);
gamma     = s(:,5);
psi       = s(:,6);
CL        = u(:,1);
phi       = u(:,2);
beta      = p(:,1);
singamma  = sin(gamma);
```

```
cosgamma       = cos(gamma);
sinpsi         = sin(psi);
cospsi         = cos(psi);
sinphi         = sin(phi);
cosphi         = cos(phi);
rho            = input.auxdata.rho;
S              = input.auxdata.S;
CD0            = input.auxdata.CD0;
K              = input.auxdata.K;
g              = input.auxdata.g;
m              = input.auxdata.m;
W0             = input.auxdata.W0;
wx             = (beta.*z+W0);
DWxDt          = beta.*v.*singamma;
vcosgamma      = v.*cosgamma;
DWxDtsinpsi    = DWxDt.*sinpsi;

xdot           = vcosgamma.*sinpsi+wx;
ydot           = vcosgamma.*cospsi;
zdot           = v.*singamma;
term1          = rho*S/2/m;
term2          = 1;
term3          = g*term2;
CLsq           = CL.^2;
vsq            = v.^2;
vdot           = -term1*(CD0+K*CLsq).*vsq-(term3)*singamma-term2*DWxDtsinpsi.*cosgamma;
gammadot       = term1*CL.*v.*cosphi-(term3)*cosgamma./v+term2*DWxDtsinpsi.*singamma./v;
psidot         = (term1*CL.*v.*sinphi-term2*DWxDt.*cospsi./v)./cosgamma;
ngconstant     = (0.5*rho*S/m/g);
ng             = ngconstant.*CL.*v.^2;

phaseout.dynamics  = [xdot, ydot, zdot, vdot, gammadot, psidot];
phaseout.path = ng;

%----------------------------------------%
% END: function dynamicSoaringContinuous.m %
%----------------------------------------%

function output = dynamicSoaringEndpoint(input)

t0    = input.phase(1).initialtime;
tf    = input.phase(1).finaltime;
x0    = input.phase(1).initialstate;
xf    = input.phase(1).finalstate;
beta  = input.parameter;
output.eventgroup(1).event = [xf(4)-x0(4), (xf(5)-x0(5)), xf(6)+2*pi-x0(6)];
output.objective = 7*beta;
```
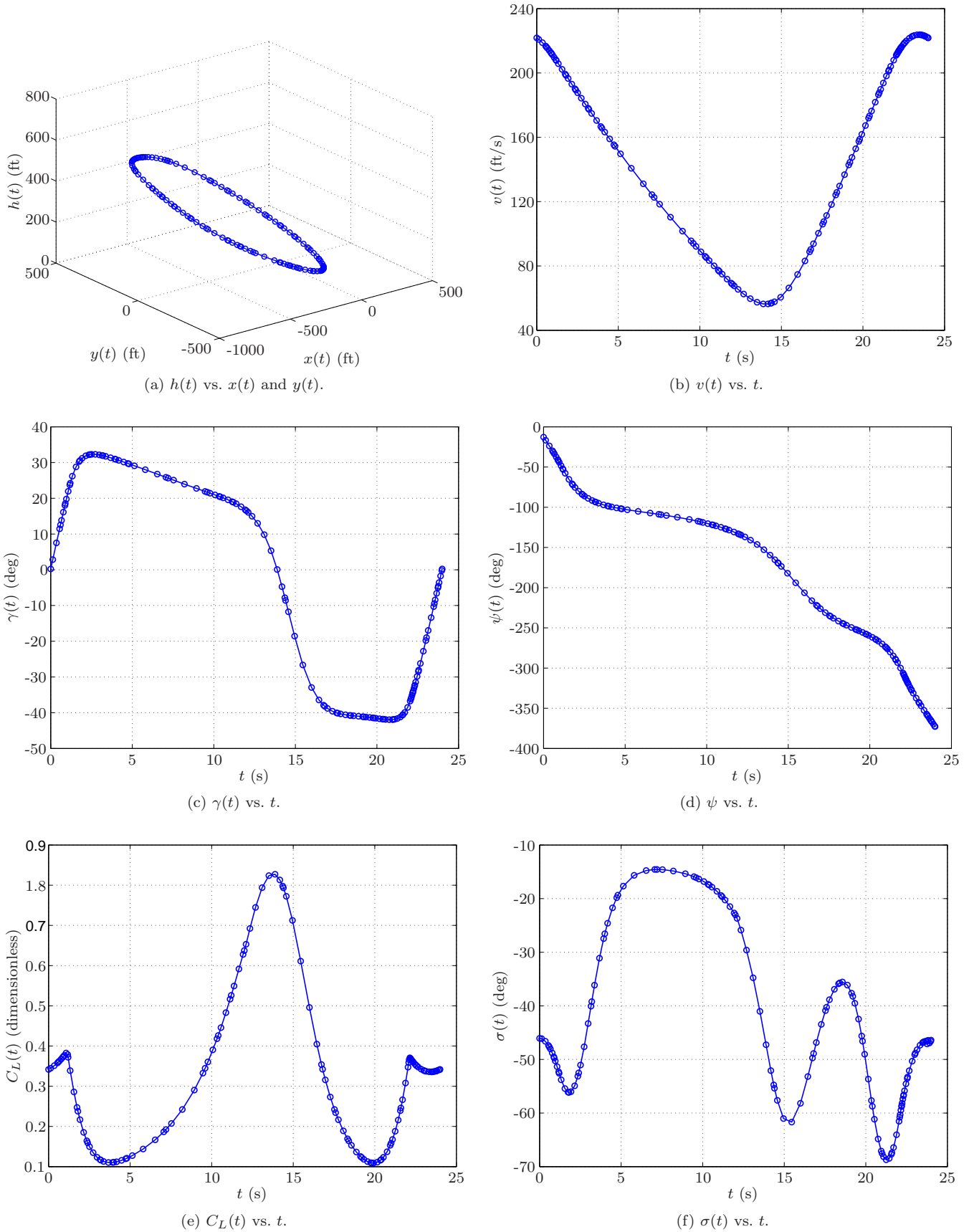
(a) $h(t)$ vs. $x(t)$ and $y(t)$.

(b) $v(t)$ vs. t.

(c) $\gamma(t)$ vs. t.

(d) ψ vs. t.

(e) $C_L(t)$ vs. t.

(f) $\sigma(t)$ vs. t.

Figure 6: Solution to Dynamic Soaring Problem Using $\mathbb{GPOPS} - \mathbb{II}$ with the NLP Solver IPOPT and a Mesh Refinement Tolerance of 10^{-6}.

5.6   Two-Strain Tuberculosis Optimal Control Problem

Quoting from Ref. 12, "[Past] models [for Tuberculosis (TB)] did not account for time dependent control strategies...In this article we consider (time dependent) optimal control strategies associated with case holding and case finding based on a two-strain TB model...Our objective functional balances the effect of minimizing the cases of latent and infectious drug-resistant TB and minimizing the cost of implementing the control treatments." The two-strain tuberculosis optimal control problem considered in Ref. 12 is formulated as follows. Minimize the objective functional

$$J = \int_0^{t_f} \left[L_2 + I_2 + \frac{1}{2} B_1 u_1^2 + B_2 u_2^2 \right] dt \tag{42}$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{S}(t) &= \Lambda - \beta_1 S \frac{I_1(t)}{N} - \beta^* S \frac{I_2(t)}{N} - \mu S(t), \\
\dot{L}_1(t) &= \beta_1 S(t) \frac{I_1(t)}{N} - (\mu + k_1) L_1(t) - u_1 r_1 L_1(t) \\
&+ (1 - u_2(t)) p r_2 I_1(t) + \beta_2 T(t) \frac{I_1(t)}{N} - \beta^* L_1(t) \frac{I_2(t)}{N}, \\
\dot{I}_1(t) &= k_1 L_1(t) - (\mu + d_1) I_1(t) - r_2 I_1(t), \\
\dot{L}_2(t) &= (1 - u_2(t)) q r_2 I_1(t) - (\mu + k_2) L_2(t) \\
&+ \beta^* (S(t) + L_1(t) + T(t)) \frac{I_2(t)}{N}, \\
\dot{I}_2(t) &= k_2 L_2(t) - (\mu + d_2) I_2(t), \\
\dot{T}(t) &= u_1(t) r_1 L_1(t) - (1 - (1 - u_2(t)))(p + q) r_2 I_1(t) \\
&- \beta_2 T(t) \frac{I_1(t)}{N} - \beta^* T(t) \frac{I_2(t)}{N} - \mu T(t), \\
0 &= S + L_1 + I_1 + L_2 + I_2 + T - N,
\end{aligned}
\tag{43}
$$

and the initial conditions

$$(S(0), L_1(0), I_1(0), L_2(0), I_2(0), T(0)) = (S_0, L_{10}, I_{10}, L_{20}, I_{20}, T_0), \tag{44}$$

where etails of the model can be found in Ref. 12 (and are also provided in the $\mathbb{GPOPS-II}$ code shown below). The optimal control problem of Eqs. (42)–(44) is solved using $\mathbb{GPOPS-II}$ with the NLP solver SNOPT with a mesh refinement accuracy tolerance of 10^{-6}. The code used to solve this example is given below and the solution is shown in Figs. 7 and 8.

```
%------------------- Two-Strain Tuberculosis Model -----------------%
% This problem is taken from the following reference:              %
% Betts, J. T., "Practical Methods for Optimal Control and         %
% Estimation Using Nonlinear Programming," SIAM Press, Philadelphia, %
% PA, 2009.                                                        %
%-----------------------------------------------------------------%
clear all
close all
clc


%-----------------------------------------------------------------%
%-------------- Provide Auxiliary Data for Problem ---------------%
%-----------------------------------------------------------------%
auxdata.beta1 = 13;
auxdata.beta2 = 13;
auxdata.mu = 0.0143;
auxdata.d1 = 0;
auxdata.d2 = 0;
auxdata.k1 = 0.5;
auxdata.k2 = 1;
```

```
auxdata.r1 = 2;
auxdata.r2 = 1;
auxdata.p = 0.4;
auxdata.q = 0.1;
auxdata.Npop = 30000;
auxdata.betas = 0.029;
auxdata.B1 = 50;
auxdata.B2 = 500;
auxdata.lam = auxdata.mu*auxdata.Npop;
auxdata.m0 = 1;
auxdata.dm = 0.0749;


%-------------------------------------------------------------------%
%------------ Set up Bounds for Optimal Control Problem -------------%
%-------------------------------------------------------------------%
t0 = 0;
tf = 5;
S0 = 76*auxdata.Npop/120;
T0 = auxdata.Npop/120;
L10 = 36*auxdata.Npop/120;
L20 = 2*auxdata.Npop/120;
I10 = 4*auxdata.Npop/120;
I20 = 1*auxdata.Npop/120;
Nmin = 0;
Nmax = 30000;
u1min = 0.05;
u1max = 0.95;
u2min = 0.05;
u2max = 0.95;

bounds.phase.initialstate.lower = [S0, T0, L10, L20, I10, I20];
bounds.phase.initialstate.upper = [S0, T0, L10, L20, I10, I20];
bounds.phase.state.lower = [Nmin, Nmin, Nmin, Nmin, Nmin, Nmin];
bounds.phase.state.upper = [Nmax, Nmax, Nmax, Nmax, Nmax, Nmax];
bounds.phase.finalstate.lower = [Nmin, Nmin, Nmin, Nmin, Nmin, Nmin];
bounds.phase.finalstate.upper = [Nmax, Nmax, Nmax, Nmax, Nmax, Nmax];
bounds.phase.control.lower = [u1min, u2min];
bounds.phase.control.upper = [u1max, u2max];
bounds.phase.initialtime.lower = t0;
bounds.phase.initialtime.upper = t0;
bounds.phase.finaltime.lower = tf;
bounds.phase.finaltime.upper = tf;
bounds.phase.integral.lower = [0];
bounds.phase.integral.upper = [10000];
bounds.phase.path.lower  = [0];
bounds.phase.path.upper  = [0];


%-------------------------------------------------------------------%
%------------ Provide an Initial Guess of the Solution -------------%
%-------------------------------------------------------------------%
timeGuess = [t0; tf];
SGuess    = [S0; S0];
TGuess    = [T0; S0];
```

```
L1Guess   = [L10; L10];
L2Guess   = [L20; L20];
I1Guess   = [I10; I10];
I2Guess   = [I20; I20];
u1Guess   = [0.95; 0.95];
u2Guess   = [0.95; 0.95];
guess.phase.time    = [timeGuess];
guess.phase.state   = [SGuess, TGuess, L1Guess, L2Guess, I1Guess, I2Guess];
guess.phase.control = [u1Guess, u2Guess];
guess.phase.integral = 6000;


%------------------------------------------------------------------------%
%------------ Provide an Initial Mesh for the Solution ------------%
%------------------------------------------------------------------------%
N = 10;
meshphase.colpoints = 4*ones(1,N);
meshphase.fraction   = ones(1,N)/N;


setup.name = 'tuberculosisModel';
setup.functions.continuous = @tuberculosisContinuous;
setup.functions.endpoint = @tuberculosisEndpoint;
setup.auxdata = auxdata;
setup.bounds = bounds;
setup.guess = guess;
setup.mesh.phase = meshphase;
setup.nlp.solver = 'snopt';
setup.derivatives.supplier = 'sparseCD';
setup.derivatives.derivativelevel = 'second';
setup.scales.method = 'automatic-bounds';
setup.method  = 'RPMintegration';
setup.mesh.method = 'hp';
setup.mesh.tolerance = 1e-6;


%------------------------------------------------------------------------%
%--------- Solve Problem with GPOPS2 and Extract Solution ----------%
%------------------------------------------------------------------------%
output = gpops2(setup);
solution = output.result.solution;


%----------------------------------------------%
% Begin Function:  tuberculosisContinuous.m %
%----------------------------------------------%
function phaseout = tuberculosisContinuous(input)

beta1 = input.auxdata.beta1;
beta2 = input.auxdata.beta2;
mu = input.auxdata.mu;
d1 = input.auxdata.d1;
d2 = input.auxdata.d2;
k1 = input.auxdata.k1;
k2 = input.auxdata.k2;
r1 = input.auxdata.r1;
```

```
r2 = input.auxdata.r2;
p = input.auxdata.p;
q = input.auxdata.q;
Npop = input.auxdata.Npop;
betas = input.auxdata.betas;
B1 = input.auxdata.B1;
B2 = input.auxdata.B2;
lam = input.auxdata.lam;

S = input.phase.state(:,1);
T  = input.phase.state(:,2);
L1 = input.phase.state(:,3);
L2 = input.phase.state(:,4);
I1 = input.phase.state(:,5);
I2 = input.phase.state(:,6);

u1 = input.phase.control(:,1);
u2 = input.phase.control(:,2);

dS = lam-(beta1.*S.*I1+betas.*S.*I2)./Npop-mu.*S;
dT = u1.*r1.*L1-mu.*T+(1-(1-u2).*(p+q)).*r2.*I1-(beta2.*T.*I1+betas.*T.*I2)./Npop;
dL1 = (beta1.*S.*I1+beta2.*T.*I1-betas.*L1.*I2)./Npop...
      -(mu+k1).*L1-u1.*r1.*L1+(1-u2).*p.*r2.*I1;
dL2 = (1-u2).*q.*r2.*I1-(mu+k2).*L2+betas.*(S+L1+T).*I2./Npop;
dI1 = k1.*L1-(mu+d1).*I1-r2.*I1;
dI2 = k2.*L2-(mu+d2).*I2;

phaseout.dynamics  = [dS, dT, dL1, dL2, dI1, dI2];
phaseout.path = S + T + L1 + L2 + I1 + I2 - Npop;
phaseout.integrand = L2 + I2 + B1./2.*u1.^2 + B2./2.*u2.^2;

%-------------------------------------------%
% End Function:  tuberculosisContinuous.m   %
%-------------------------------------------%

%-------------------------------------------%
% Begin Function:  tuberculosisEndpoint.m %
%-------------------------------------------%
function output = tuberculosisEndpoint(input)

output.objective = input.phase.integral;

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

(a) $S(t)$ vs. t.

(b) $L_1(t)$ vs. t.

(c) $L_2(t)$ vs. t.

(d) I_1 vs. t.

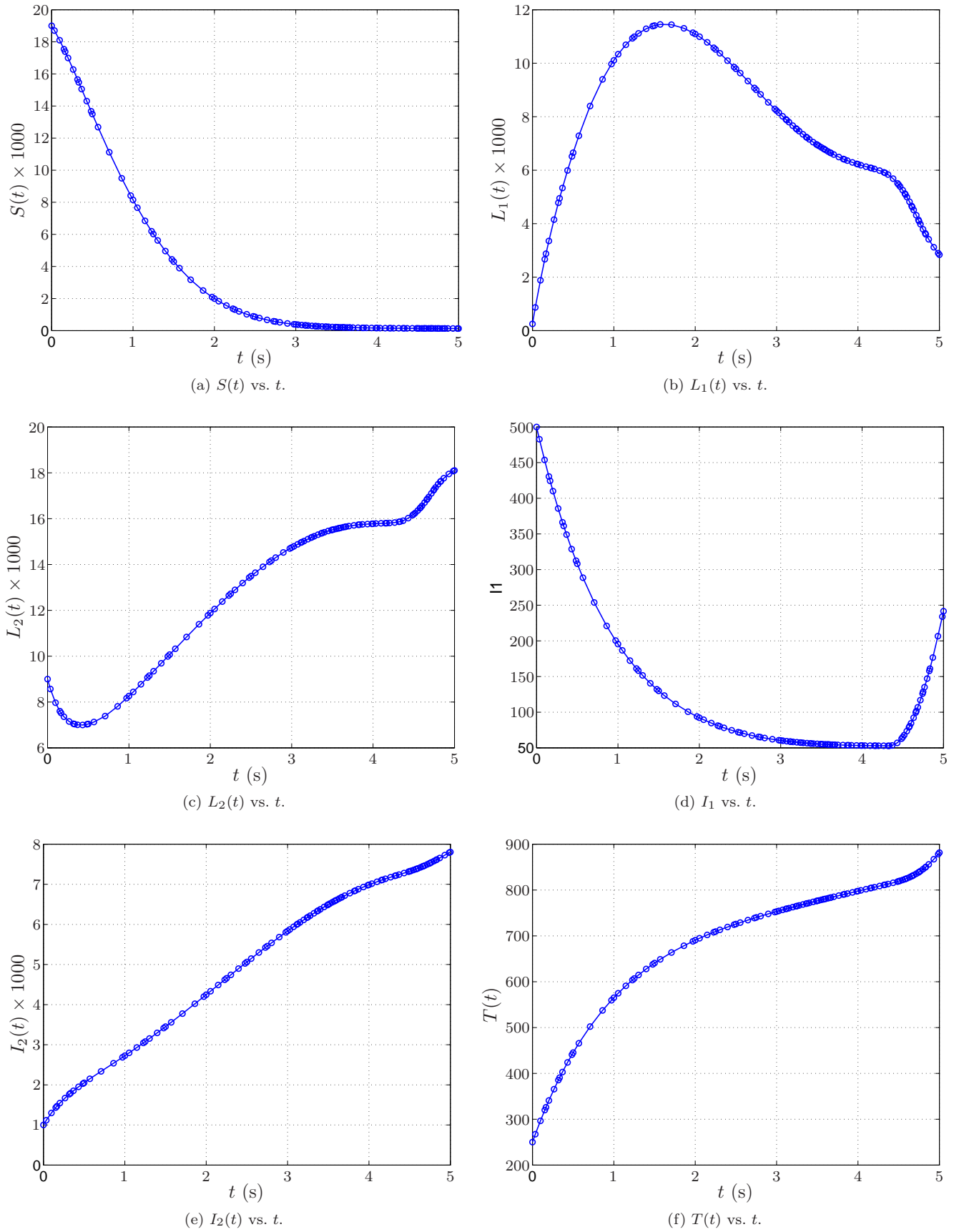(e) $I_2(t)$ vs. t.

(f) $T(t)$ vs. t.

Figure 7: Optimal State for Tuberculosis Optimal Control Problem Using $\mathbb{GPOPS} - \mathbb{II}$ with the NLP Solver SNOPT and a Mesh Refinement Tolerance of 10^{-6}.
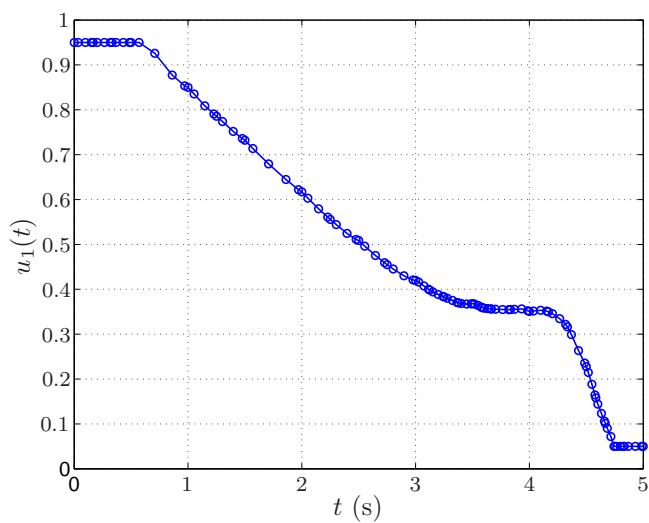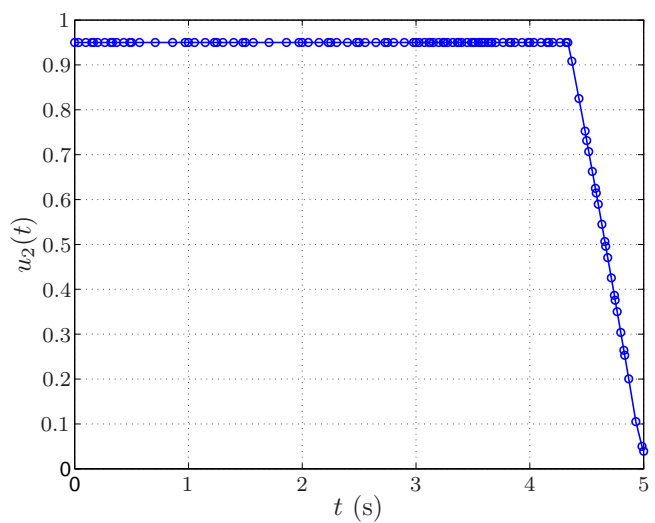
(a) $u_1(t)$ vs. t.

(b) $u_2(t)$ vs. t.

Figure 8: Optimal Control for Tuberculosis Optimal Control Problem Using $\mathbb{GPOPS-II}$ with the NLP Solver SNOPT and a Mesh Refinement Tolerance of 10^{-6}.

6   Concluding Remarks

While the authors have put for the effort to make $\mathbb{GPOPS-II}$ a user-friendly software, it is important to understand several aspects of computational optimal control in order to make $\mathbb{GPOPS-II}$ 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] 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.

[2] 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.

[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 via a Radau Pseudospectral Method," *Computational Optimization and Applications*, Vol. 49, No. 2, June 2011, pp. 335–358.

[4] Patterson, M. A. and Rao, A. V., "Exploiting Sparsity in Direct Collocation Pseudospectral Methods for Solving Continuous-Time Optimal Control Problems," *Journal of Spacecraft and Rockets,*, Vol. 49, No. 2, March–April 2012, pp. 364–377.

[5] Betts, J. T., *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*, SIAM Press, Philadelphia, 2009.

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

[7] 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.

[8] Rao, A. V., Benson, D. A., Christopher Darby, M. A. P., 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, pp. 22:1–22:39.

[9] Ledzewicz, U. and Schättler, H., "Analysis of Optimal Controls for a Mathematical Model of Tumour Anti-Angiogenesis," *Optimal Control Applications and Methods*, Vol. 29, No. 1, January–February 2008, pp. 41–57.

[10] 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.

[11] Zhao, Y. J., "Optimal Pattern of Glider Dynamic Soaring," *Optimal Control Applications and Methods*, Vol. 25, 2004, pp. 67–89.

[12] Jung, E., Lenhart, S., and Feng, Z., "Optimal Control of Treatments in a Two-Strain Tuberculosis Model," *Discrete and Continuous Dynamical Systems – Series B*, Vol. 2, 2002, pp. 479–482.

[13] Rao, A. V. and Mease, K. D., "Dichotomic Basis Approach to solving Hyper-Sensitive Optimal Control Problems," *Automatica*, Vol. 35, No. 4, April 1999, pp. 633–642.

[14] Rao, A. V., "Application of a Dichotomic Basis Method to Performance Optimization of Supersonic Aircraft," *Journal of Guidance, Control, and Dynamics*, Vol. 23, No. 3, May–June 2000, pp. 570–573.

[15] Rao, A. V., "Riccati Dichotomic Basis Method for solving Hyper-Sensitive optimal Control Problems," *Journal of Guidance, Control, and Dynamics*, Vol. 26, No. 1, January–February 2003, pp. 185–189.

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

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