

GPOPS-ii

**A General-Purpose MATLAB Software for
Solving Multiple-Phase Optimal Control Problems**

Version 2.3

Michael A. Patterson, Ph.D.
Anil V. Rao, Ph.D.

December 2016

Copyright © 2013–2016 Michael A. Patterson and Anil V. Rao. All Rights Reserved.

Preface

This document serves a user’s guide for Version 2.0 of the MATLAB optimal control software `GPOPS – III`: a general-purpose software for solving optimal control problems using variable-order adaptive orthogonal collocation methods together with sparse nonlinear programming. The class of problems that `GPOPS – III` can solve is very broad and include applications in engineering, economics, and medicine. `GPOPS – III` uses some of the latest advancements in the area of orthogonal collocation methods along with the state-of-the-art in nonlinear optimization. Specifically, `GPOPS – III` employs a variable-order adaptive Legendre-Gauss-Radau quadrature collocation method. `GPOPS – III` has been designed to work with the nonlinear programming (NLP) solvers SNOPT and IPOPT, and MATLAB mex files for IPOPT are included with the software (users must obtain SNOPT on their own, but the complete interface for using SNOPT is included with `GPOPS – III`). 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 `GPOPS – III` 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.

The new version of `GPOPS – III` described in this user’s guide differs from the first release of May 2013 in the follows ways. First, some of the input syntax has been modified to be more elegant than it was previously. Second, the software now includes a complete input checker that provides errors and warnings regarding user-defined inputs. Third, from an algorithmic standpoint the major change is the way in which NLP solver derivatives can now be obtained. In addition to the original sparse finite-differencing method (which is still included), the user now has the choice of generating derivative source code using the open-source automatic differentiation software *ADiGator*¹ (written by Matthew J. Weinstein and available at <http://sourceforge.net/projects/adigator>) or by providing user-supplied (analytic) derivatives. These improvements have been made to increase both computational efficiency and reliability of the software.

Acknowledgments

The authors of `GPOPS – III` have many people to thank along the way for their contributions to the research that has made `GPOPS – III` possible. While it is impossible to acknowledge everyone who has contributed to the development of the underlying methods that are implemented in `GPOPS – III`, here is a partial (hopefully complete) list of people and their contributions:

- Dr. Divya Garg: mathematical theory showing the equivalence between the differential and integral forms of the methods along with the development of the transformed adjoint systems as described in Refs. 2, 3, and 4.
- Dr. Christopher L. Darby: *hp* adaptive mesh refinement methods as described in Refs. 5 and 6.
- Dr. Camila Francolin: costate estimation as described in Ref. 7.
- Matthew Weinstein: development of the open-source automatic differentiation package *ADiGator* as found at <http://www.sourceforge.net/projects/adigator> and described in Ref. 1.
- Fengjin Liu: development of the *hp* adaptive mesh refinement methods as described in Refs. 8 and 9.
- Professor William W. Hager: for the honor of being able to collaborate over the past several years with one of the world’s most distinguished applied mathematicians on our optimal control research and for his contributions to all of the aforementioned *hp* adaptive mesh refinement methods as described in Refs. 5, 6, 8–10.

Disclaimer

The contents of this document and the corresponding software `GPOPS – III` are provided “as is” without any merchantability or fitness for any particular application. In particular,

- 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).

Neither authors nor their employers (past, present, or future) assume any responsibility whatsoever from any harm resulting from the software.

GPOPS – III is provided "as is" without warranty of any kind, expressed or implied, 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.

Licensing Agreement

By downloading, using, modifying, or distributing GPOPS2, you agree to the terms of the license agreement as stated on the website <http://www.gpops2.com/License/License.html>. **Please read the license terms and conditions carefully before proceeding to download, install, or use GPOPS – III**

Installation Instructions

GPOPS – III can be installed using any version of MATLAB after R2011a on any computer that runs Microsoft Windows 32-bit and 64-bit, Linux 64-bit, or Mac OS-X 64-bit. The installation instructions are as follows:

- (1) Unzip the zip archive from the download link that was supplied when you registered or purchased GPOPS – III. Once the file is unzipped, place the license file you received in the directory \$GPOPS2/license (where \$GPOPS2 is the root directory where GPOPS – III was unzipped).
- (2) Start MATLAB, change to the directory \$GPOPS2, and execute the file "gpopsMatlabPathSetup.m" from the MATLAB command prompt. The execution of "gpopsMatlabPathSetup.m" will attempt to write to the master path definition file "pathdef.m". If "pathdef.m" cannot be written to (for example, if you are not the administrator of the computer or if you otherwise do not have permission to write to "pathdef.m"), then the GPOPS – III directories will not be saved when MATLAB is closed. If the MATLAB path cannot be saved and you want to include the GPOPS – III directories each time MATLAB starts, you will need to add the GPOPS – III paths using the "addpath" command in a "startup.m" file and place the "startup.m" file in your MATLAB home working directory.

For completeness, if you do not have permission to write to the "pathdef.m" file, then the following addpath commands should be included in the "startup.m" file so that you have a functioning GPOPS – III installation each time MATLAB is started:

```
addpath('$GPOPS2/license');
addpath('$GPOPS2/nlp/ipopt');
addpath('$GPOPS2/gpopsUtilities');
addpath('$GPOPS2/lib/gpopsCommon');
addpath('$GPOPS2/lib/gpopsMeshRefinement');
addpath('$GPOPS2/lib/gpopsAutomaticScaling');
addpath('$GPOPS2/lib/gpopsADiGator');
addpath('$GPOPS2/lib/gpopsFiniteDifference');
addpath('$GPOPS2/lib/gpopsRPMDifferentiation');
addpath('$GPOPS2/lib/gpopsRPMDifferentiation/gpopsIpoptRPM');
addpath('$GPOPS2/lib/gpopsRPMDifferentiation/gpopsSnoptRPM');
```

```
addpath('$GPOPS2/lib/gpopsRPMIntegration');  
addpath('$GPOPS2/lib/gpopsRPMIntegration/gpopsIoptRPMI');  
addpath('$GPOPS2/lib/gpopsRPMIntegration/gpopsSnoptRPMI');
```

Finally, it is noted that the following information is displayed upon execution of “gpops2License”:

GPOPS-II License Information

```
License Owner   : Academic Institution, Not-for-Profit Corporation, or Commercial Institution  
License Type    : Academic, Not-for-Profit, or Commercial  
License Class   : Single-User, Department-Wide, or Institution-Wide
```

```
License Expiration Date [year, month, day]      : [Expiration-Year, Expiration-Month, Expiration-Day]  
License Valid For The Following MATLAB license(s) : List-of-Valid-MATLAB-License-Numbers
```

The above installation instructions do not include the installation of the open-source automatic differentiation software *ADiGator*.¹ If you would like to use *ADiGator* with GPOPS – III, please visit the *ADiGator* website <http://sourceforge.net/projects/adigator> and follow the setup instructions included with *ADiGator*. If you have any questions about installing GPOPS – III, please send an e-mail to support@gpops2.com.

Contents

1	Introduction to the General-Purpose Software GPOPS – II	6
1.1	Radau Orthogonal Collocation Method Employed by GPOPS – II	6
1.2	Organization of GPOPS – II	6
1.3	Color Highlighting Throughout Document	7
2	Constructing an Optimal Control Problem Using GPOPS – II	7
2.1	Syntax for Input Structure <code>setup</code>	7
2.2	Syntax for Structure <code>setup.functions</code>	10
2.3	Syntax for <code>bounds</code> Structure	10
2.4	Syntax of Endpoint Function <code>setup.functions.endpoint</code>	12
2.5	Syntax for Continuous Function <code>setup.functions.continuous</code>	12
2.6	Specifying an Initial Guess of The Solution	13
2.7	Use of <i>ADiGator</i> ¹ to Generate NLP Solver Derivatives	14
2.8	Scaling the Optimal Control Problem	15
3	Output from an Execution of GPOPS – II	15
4	Useful Information for Debugging a GPOPS – II Problem	16
5	GPOPS – II Examples	16
5.1	Hyper-Sensitive Problem	16
5.2	Low-Thrust Orbit Transfer Problem	20
5.3	Initial Guess Generation	21
6	Results	22
6.1	Multiple-Stage Launch Vehicle Ascent Problem	31
6.2	Tumor-Antiangiogenesis Optimal Control Problem	43
6.3	Reusable Launch Vehicle Entry	47
6.4	Minimum Time-to-Climb of a Supersonic Aircraft	52
6.5	Two-Strain Tuberculosis Optimal Control Problem	64
7	Concluding Remarks	70

1 Introduction to the General-Purpose Software GPOPS – III

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 $p \in [1, \dots, P]$, and the static parameters, $\mathbf{s} \in \mathbb{R}^{n_s}$, that minimize the cost functional

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

subject to the dynamic constraints

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

the event constraints

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

the inequality path constraints

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

and the integral constraints

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

where

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

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 Orthogonal Collocation Method Employed by GPOPS – III

The method employed by GPOPS – III is an hp -adaptive version of the *Legendre-Gauss-Radau (LGR) orthogonal collocation method*. The LGR orthogonal collocation method is a Gaussian quadrature implicit integration method where collocation is performed at LGR points. The theory behind the collocation method used in GPOPS – III can be found Refs. 2, 3, 4, and 11.

1.2 Organization of GPOPS – III

GPOPS – III 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 GPOPS – III

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

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

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 5.3 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
<i>setup.auxdata</i>	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
<code>setup.derivatives.supplier</code>	'sparseFD' or 'sparseBD' or 'sparseCD' or 'adigator'	'sparseFD'
<code>setup.derivatives.derivativelevel</code>	'first' or 'second'	'first'
<code>setup.derivatives.dependencies</code>	'full', 'sparse' or 'sparseNaN'	'sparseNaN'

Note that the option `setup.derivatives.supplier` is ignored when the derivative option is set to either 'analytic' or 'adigator'.

- **scales**: a structure that specifies how the problem to be solved is scaled. The field `scales` itself contains a field `method` that can be set to one of the following:

Field	Possible Values	Default
<code>setup.scales.method</code>	'none' or 'automatic-bounds' or 'automatic-guess' 'automatic-guessUpdate' or 'automatic-hybrid' or 'automatic-hybridUpdate' or 'defined'	'none'

The option 'automatic-bounds' scales the problem from the user-supplied bounds on the variables. The option 'automatic-guess' scales the problem once using the initial guess of the solution supplied by the user. The option 'automatic-guessUpdate' scales the problem from the initial guess on the first mesh and from the solution obtained on every subsequent mesh during the mesh refinement. The option 'automatic-hybrid' scales the problem from the user supplied bounds on the variables on the first mesh and from the solution obtained on the initial mesh for every subsequent mesh in the mesh refinement. The option 'automatic-hybridUpdate' scales the problem from the bounds on the initial mesh and from the solution obtained on every subsequent mesh during the mesh refinement. It is noted that the field `scales.method` is changed to 'defined' when using any scaling method where the scales are not changed between every mesh iteration (in other words, when using 'automatic-bounds', 'automatic-guess', or 'automatic-hybrid'). Finally, other fields in the structure `setup.scales` are created during an execution of GPOPS – III but the user need not in general be concerned with these additional fields. If a user desires an explanation of these other fields that are created, please contact the authors.

- **method**: a string that defines the version of the collocation to be used when solving the problem. Valid options are

Field	Possible Values	Default
<code>setup.method</code>	'RPM-Differentiation' or 'RPM-Integration'	'RPM-Differentiation'

- **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`, `maxiterations`, `colpointsmin`, `colpointsmax`, `splitmult`, `curveratio`, `R`, `sigma`, and `phase`. The field `setup.mesh.method` is a string that specified the particular mesh refinement method to be used. The field `setup.mesh.tolerance` contains the desired accuracy tolerance of the mesh. The field `setup.mesh.maxiterations` contains the maximum number of allowed mes iterations. The fields `setup.mesh.colpointsmin` and `setup.mesh.colpointsmax` are integers (both greater than two and such that `colpointsmax` is greater `colpointsmin`) that specifies the minimum and maximum allowable number of collocation points, respectively, in a mesh interval. The

fields `setup.mesh.curveratio` and `setup.mesh.splitmult` are used with the method “hp-DarbyRao” only (see below) and are defined as follows. First, `setup.mesh.curveratio` provide a threshold for the maximum-to-average curvature ratio of the solution in a mesh interval to determine whether to increase the polynomial degree within a mesh interval or create new mesh intervals. Second, the field `setup.mesh.curveratio` provides a scale factor that determines the number of mesh intervals to create when dividing a mesh interval into subintervals. The field `setup.mesh.R` is used with the method “hp-LiuRao” only (see below) and provides a derivative ratio that determines whether to increase the polynomial degree within a mesh interval or to create new mesh intervals based on the maximum-to-average value of the derivative of the solution in a mesh interval. Finally, the field `setup.mesh.sigma` is used with the method “hp-LiuRao-Legendre” (see below) and determines whether to increase the polynomial degree within a mesh interval or to create new mesh intervals based on the decay rate of a Legendre polynomial coefficient expansion of the solution in a mesh interval. All of these parameters are described in Refs. 5,6,8 and 10 (the appropriate reference being that which corresponds to the method being used).

Field	Possible Values	Default
<code>setup.mesh.method</code>	'hp-PattersonRao' or 'hp-DarbyRao' or 'hp-LiuRao' or 'hp-LiuRao-Legendre'	'hp-PattersonRao'
<code>setup.mesh.tolerance</code>	Positive Number Between 0 and 1	10^{-3}
<code>setup.mesh.maxiterations</code>	Non-Negative Integer	10
<code>setup.mesh.colpointsmin</code>	Integer > 2	3
<code>setup.mesh.colpointsmax</code>	Integer \geq <code>colpointsmin</code>	10
<code>setup.mesh.splitmult</code>	Real Number > 1	1.2
<code>setup.mesh.curveratio</code>	Real Number > 1	2
<code>setup.mesh.R</code>	Real Number > 1	1.2
<code>setup.mesh.sigma</code>	Real Number > 0	0.5

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 $p = 1, \dots, P$, 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 $p = 1, \dots, P$, where the number of collocation points in each mesh interval is also specified as a row vector such that the i^{th} entry in `setup.mesh.phase(p).colpoints` corresponds to the i^{th} entry in `setup.mesh.phase(p).fraction`.

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

- `nlp`: a structure that specifies the NLP solver to be used and the options to be used within the chosen NLP solver. `setup.nlp` contains the field `solver` and `options`. The field `solver` contains a string indicating the NLP solver to be used. The fields `ipoptoptions` and `snoptoptions` are structures that themselves contains fields with options for the NLP solvers IPOPT and SNOPT, respectively, which can be set from GPOPS – III.

Field	Possible Values	Default
<code>setup.nlp.solver</code>	'snopt' or 'ipopt'	'ipopt'

Field `nlp.ipoptoptions` for the NLP solver IPOPT:

Field	Possible Values	Default
<code>setup.nlp.ipoptoptions.linear_solver</code>	'mumps' or 'ma57'	'mumps'
<code>setup.nlp.ipoptoptions.tolerance</code>	Positive Real Number	10^{-7}
<code>setup.nlp.ipoptoptions.maxiterations</code>	Positive Integer	2000

Field `nlp.snoptoptions` for the NLP solver SNOPT:

<code>setup.nlp.snoptoptions.tolerance</code>	Positive Real Number	10^{-6}
<code>setup.nlp.snoptoptions.maxiterations</code>	Positive Integer	2000

- **displaylevel**: a integer that takes on the values 0, 1, or 2 and provides the amount of output that is sent to the MATLAB command window during an execution of `GPOPS – III`. The following table provided the possible values and their defaults for the field `setup.auxdata`:

Field	Possible Values	Default
<code>setup.displaylevel</code>	0, 1, or 2	2

A display level of zero suppresses all output. A display level of one provides only mesh refinement output. Finally, a display level of two provides both NLP solver iteration output and mesh refinement output.

It is important to note that `GPOPS – III` 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:

```

setup.functions.continuous = @continuousfun.m
setup.functions.endpoint   = @endpointfun.m

```

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 `GPOPS – III` 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 $p = 1, \dots, P$ 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 p^{th} 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 $p \in [1, \dots, P]$. The scalars **bounds.phase(p).initialtime.lower** and **bounds.phase(p).initialtime.upper** have the following form:

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

- **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 $p \in [1, \dots, P]$. The scalars **bounds.phase(p).finaltime.lower** and **bounds.phase(p).finaltime.upper** have the following form:

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

- **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 $p \in [1, \dots, P]$. The row vectors **bounds.phase(p).initialstate.lower** and **bounds.phase(p).initialstate.upper** have the following form:

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

- **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 $p \in [1, \dots, P]$. The row vectors **bounds.phase(p).state.lower** and **bounds.phase(p).state.upper** have the following form:

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

- **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 $p \in [1, \dots, P]$. The row vectors **bounds.phase(p).finalstate.lower** and **bounds.phase(p).finalstate.upper** have the following form:

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

- **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 $p \in [1, \dots, P]$. The row vectors **bounds.phase(p).control.lower** and **bounds.phase(p).control.upper** have the following form:

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

- **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 $p \in [1, \dots, P]$. The row vectors **bounds.phase(p).path.lower** and **bounds.phase(p).path.upper** have the following form:

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

- **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 $p \in [1, \dots, P]$. The row vectors **bounds.phase(p).integral.lower** and **bounds.phase(p).integral.upper** have the following form:

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

- **bounds.phase(p).duration.lower** and **bounds.phase(p).duration.upper**: scalars that contain the lower and upper bounds on the duration of a phases $p \in [1, \dots, P]$. The duration is the difference between the final time of the phase and the initial time of the phase, $t_f^{(p)} - t_0^{(p)}$.

- **bounds.parameter.lower** and **bounds.parameter.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:

$$\begin{aligned} \mathbf{bounds.parameter.lower} &= \begin{bmatrix} s_1^{\text{lower}} & \dots & s_{n_s}^{\text{lower}} \end{bmatrix} \\ \mathbf{bounds.parameter.upper} &= \begin{bmatrix} s_1^{\text{upper}} & \dots & s_{n_s}^{\text{upper}} \end{bmatrix} \end{aligned}$$

- `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 $g = 1, \dots, G$ of event constraints. The row vectors `bounds.eventgroup(g).lower` and `bounds.eventgroup(g).upper` have the following form:

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

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 p^{th} element of `input.phase` contains the following fields:

- `input.phase(p).initialtime`: a scalar that contains the initial time in phase $p = 1, \dots, P$;
- `input.phase(p).finaltime`: a scalar that contains the final time in phase $p = 1, \dots, P$;
- `input.phase(p).initialstate`: a row vector of length $n_y^{(p)}$ that contains the initial state in phase $p = 1, \dots, P$;
- `input.phase(p).finalstate`: a row vector of length $n_y^{(p)}$ that contains the final state in phase $p = 1, \dots, P$;
- `input.phase(p).integral`: a row vector of length $n_d^{(p)}$ that contains the integrals in phase $p = 1, \dots, P$;
- `input.parameter`: a row vector of length n_s that contains the static parameters in phase $p = 1, \dots, P$;

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 g^{th} 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 p^{th} 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 $p = 1, \dots, P$.
- **input.phase(p).state**: 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 $p = 1, \dots, P$;
- **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 $p = 1, \dots, P$;
- **input.phase(p).parameter**: a matrix of size $N^{(p)} \times n_s$, where $N^{(p)}$ is the number of collocation points in phase $p = 1, \dots, P$ 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 p^{th} 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 $p = 1, \dots, P$;
- **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 $p = 1, \dots, P$;
- **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 $p = 1, \dots, P$;

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 $p = 1, \dots, P$. The field **setup.guess.phase** is an array of structures of length P such that the p^{th} element of **setup.guess.phase** contains the following fields:

- **setup.guess.phase(p).time**: a column vector of length $M^{(p)}$ in phase $p = 1, \dots, P$;
- **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 $p = 1, \dots, P$;
- **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 $p = 1, \dots, P$;
- **setup.guess.phase(p).integral**: a row vector of length $n_d^{(p)}$, where $n_d^{(p)}$ is the number of integrals in phase $p = 1, \dots, P$;
- **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 $p = 1, \dots, P$ in the field **setup.guess.phase(p).time** must be monotonically increasing.

2.7 Use of *ADiGator*¹ to Generate NLP Solver Derivatives

As stated earlier, one of the options for generating derivatives required by the NLP solver is to use the open-source automatic MATLAB differentiation software *ADiGator*.¹ The software *ADiGator* is described in detail in Ref. 1 and can be found at <http://sourceforge.net/projects/adigator>. The option for using *ADiGator* is invoked by setting `setup.derivatives.method = 'adigator'`. Assuming the user has obtained *ADiGator* from <http://sourceforge.net/projects/adigator> and the option `setup.derivatives.method` has been set to 'adigator', the first and/or second derivative functions (depending upon which NLP solver is being used and which derivative level is chosen) of the user-supplied optimal control functions `setup.functions.continuous` and `setup.functions.endpoint` are obtained simply by executing GPOPS – III on the user setup. At the start of the GPOPS – III run, screen output will be displayed that provides a status of the derivative file generation. After generating the derivative files, GPOPS – III will run in the usual mode, this time using the derivative code generated by *ADiGator* instead of using the default sparse finite-difference method.

An important aspect of using *ADiGator* is that the user may only want to generate derivative code once and then re-use the derivative functions in future runs of GPOPS – III. In order to avoid regeneration of the user `continuous` and `endpoint` functions, it is necessary that the user provide these derivative function names in the `setup` structure that will be used in the future execution of GPOPS – III. First, let `myproblem` be the prefix for the continuous and endpoint function names (that is, the continuous and endpoint functions are m-files named `myproblemContinuous` and `myproblemEndpoint`). The first derivative functions are then m-files named `myproblemContinuousADiGatorGrd` and `myproblemEndpointADiGatorGrd.m` while the second derivative functions are m-files named `myproblemContinuousADiGatorHes` and `myproblemEndpointADiGatorHes`. Corresponding to the first and second derivative function names created by *ADiGator*, the following fields must be added to the `setup` structure used in a future run of GPOPS – III:

- `setup.adigatordrd.continuous = @myproblemContinuousADiGatorGrd`
- `setup.adigatordrd.endpoint = @myproblemEndpointADiGatorGrd`
- `setup.adigatorhes.continuous = @myproblemContinuousADiGatorHes`
- `setup.adigatorhes.endpoint = @myproblemEndpointADiGatorHes`

Finally, a few important points should be noted when using *ADiGator* with GPOPS – III. The first important point pertains to when the derivative files are regenerated. Specifically, it is recommended to regenerate the derivative files if the user has any doubt as to whether any changes have been made to *anything* in the problem setup or the associated function files. For example, if the user continuous or endpoint function depends upon another function and the user has made changes in the dependent function (but not in the actual user functions themselves), then it will be necessary to regenerate the derivative functions using *ADiGator*. The easiest way to know that the correct derivative files are used in subsequent executions of GPOPS – III is to delete any or all of the files created by *ADiGator* before executing GPOPS – III using the derivative option `setup.derivatives.method = 'adigator'`.

A second important point regarding *ADiGator* pertains to the situation when GPOPS – III is unable to generate derivative files. As with any automatic differentiation software, *ADiGator* has limitations on function code that can be differentiated. If a GPOPS – III user function cannot be differentiated by *ADiGator*, GPOPS – III will produce the following *generic* error:

GPOPS-II ERROR: ADiGator could not produce derivative files

Note that the above error does not provide the reason why *ADiGator* cannot produce derivative files. In order for the user to ascertain the reason why *ADiGator* cannot produce derivative files, it is necessary to run the function `adigatordGenFiles4gpop2` on the user `setup` structure. In fact, derivative files can be generated *without* running GPOPS – III simply by executing the command

`adigatordfilenames = adigatordGenFiles4gpop2(setup)`

It is also noted that *ADiGator* is updated regularly when bugs are found. For more information about using *ADiGator*, the user is referred to the *ADiGator* user's guide as found on the *ADiGator* project page <http://sourceforge.net/projects/adigator>.

2.8 Scaling the Optimal Control Problem

It is always preferable for the user to scale an optimal control problem of interest based on an understanding of the problem itself. In many cases, however, it may be difficult to determine a suitable scaling of the optimal control problem. While it is beyond the scope of the software to provide a general procedure for scaling, several different scaling options have been provided in GPOPS – III that may prove useful to the user in lieu of manually scaling the problem. We now explain these options in more detail than were explained earlier in this user’s guide.

Setting `setup.scales.method` to ‘automatic-bounds’ scales the problems based on the variable bounds and by computing scale factors based on randomly sampling the first derivatives of the optimal control problem functions.¹² Setting `setup.scales.method` to ‘automatic-guess’ scales the problem once using the initial guess supplied by the user. Note that the option ‘automatic-guess’ does not re-scale the problem on each mesh refinement iteration, but only scales the problem a *single* time based purely on whatever guess the user provides. Setting `setup.scales.method` to ‘automatic-guessUpdate’ scales the problem on the user-supplied initial guess on the first mesh and from the solution generated by GPOPS – III on every mesh during the mesh refinement. As a result, the option ‘automatic-guessUpdate’ performs a new scaling of the optimal control problem for every mesh on which the problem is solved. Setting `setup.scales.method` to ‘automatic-hybrid’ provides a hybrid scaling that uses the same method as ‘automatic-bounds’ on the first mesh and then performs a *single* scaling from the solution obtained on the first mesh. Setting `setup.scales.method` to ‘automatic-hybridUpdate’ scales the problem using the method ‘automatic-bounds’ on the initial mesh and using the method ‘automatic-guessUpdate’ on every subsequent mesh during the mesh refinement. In order to maximize repeatability, `scales.method` is changed to ‘defined’ when using any scaling method where the scales are not changed between mesh iterations (in other words, the scales are set to ‘defined’ when using ‘automatic-bounds’, ‘automatic-guess’, or ‘automatic-hybrid’). Finally, other fields in the structure `setup.scales` are created during an execution of GPOPS – III but the user need not in general be concerned with these additional fields. If a user desires an explanation of these other fields that are created, please contact the authors.

3 Output from an Execution of GPOPS – III

The output of an execution of GPOPS – III 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 GPOPS – III;
- **result.nextsetup**: the setup structure that would be used for the next mesh refinement iteration had the mesh refinement process not terminated on the previous mesh (due to the mesh refinement accuracy tolerance not having been satisfied).
- **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 GPOPS – III (only if mesh refinement is used);

4 Useful Information for Debugging a GPOPS – III Problem

One aspect of GPOPS – III that may appear confusing when debugging code pertains to the dimensions of the arrays and the corresponding time values. It is important to remember that GPOPS – III uses collocation at *Legendre-Gauss-Radau* points. Because the Legendre-Gauss-Radau points include the initial point but do not include the final point, the dynamics, path constraints, and integrand cost are computed only at the Legendre-Gauss-Radau points. While this may appear to be a bit strange, the fundamental point here is that Legendre-Gauss-Radau quadrature (which is used in GPOPS – III) only evaluates the functions at the Legendre-Gauss-Radau points. Do not try to “fool” GPOPS – III by adding the endpoints to the computation of the dynamics, path constraints, or integrand cost. If you do this, you will get an error because the dimensions are incorrect. For a more complete mathematical description of the collocation method used in GPOPS – III, see the references on the Radau orthogonal collocation method as given in the bibliography at the end of this document.

5 GPOPS – III Examples

In this Section several examples of using GPOPS – III are provided. 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. 13. The second example is a multiple-stage launch vehicle ascent problem taken from Refs. 14, 15, and 12. The third example is a tumor anti-angiogenesis optimal control problem from Refs. 16 and 12. The fourth example is the reusable launch vehicle entry problem taken from Ref. 12. The fifth example is the minimum time-to-climb of a supersonic aircraft taken from Refs. 17 and 12. The sixth example is the optimal control of a hang glider and is taken from Ref. 18. Finally, the seventh example is the optimal control of a two-strain tuberculosis model and is taken from Ref. 19. For each example the optimal control problem is described quantitatively, the GPOPS – III code is provided, and the solution obtained using GPOPS – III 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*^{13,20–22} optimal control problem adapted from Ref. 13. Minimize the cost functional

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

subject to the dynamic constraint

$$\dot{x} = -x^3 + u \quad (8)$$

and the boundary conditions

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

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. 13 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 GPOPS – III with the NLP solver IPOPT and a mesh refinement tolerance $\epsilon = 10^{-7}$. In order to solve this problem using GPOPS – III, 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 = 1000;
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 = 100000;

%-----%
%----- 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 = 'hp-LiuRao-Legendre';
mesh.tolerance = 1e-7;
mesh.maxiterations = 45;
mesh.colpointsmin = 2;
mesh.colpointsmax = 14;
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.displaylevel             = 2;
setup.bounds                   = bounds;
setup.guess                    = guess;
setup.mesh                     = mesh;
setup.nlp.solver               = 'ipopt';
setup.nlp.snoptoptions.tolerance = 1e-10;
setup.nlp.snoptoptions.maxiterations = 20000;
setup.nlp.ipoptoptions.linear_solver = 'ma57';
setup.nlp.ipoptoptions.tolerance = 1e-10;
setup.derivatives.supplier     = 'adigator';
setup.derivatives.derivativelevel = 'second';
setup.method                   = 'RPM-Differentiation';

%-----%
%----- Solve Problem Using GPOPS2 -----%
%-----%

tic
output = gpops2(setup);
toc

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

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

% xdot = -x.^3+u;
xdot = -x+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 GPOPS – III with the above code and the NLP solver IPOPT is summarized in Figs. 1a–1c, while a table showing the estimate of the relative error as a function of the mesh refinement iteration is shown in Table 1.

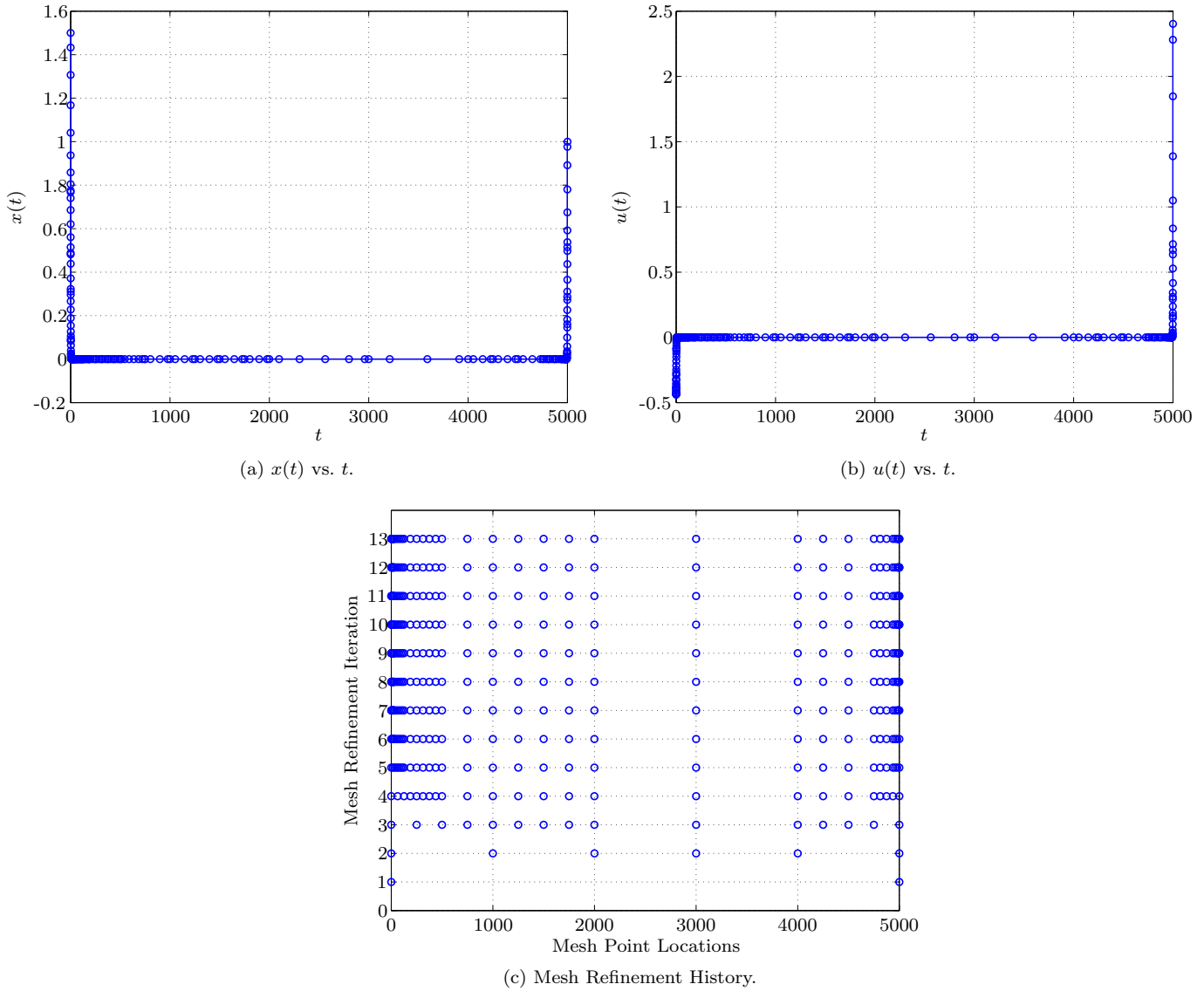


Figure 1: Solution to Hyper-Sensitive Problem Obtained Using GPOPS – III with the NLP Solver IPOPT and a Mesh Refinement Tolerance of 10^{-7} .

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 Low-Thrust Orbit Transfer Problem

Consider the following low-thrust orbital transfer optimal control problem taken from Ref. 12. The state of the system is given in modified equinoctial elements while the control is given in radial-transverse-normal coordinates. The goal is to determine the state

$$\mathbf{x} = (p, f, g, h, k, L, w), \quad (10)$$

the control

$$\mathbf{u} = (u_r, u_\theta, u_h), \quad (11)$$

and the throttle parameter, τ , that transfer the spacecraft from an initial orbit to a final orbit while maximizing the final weight of the spacecraft. The spacecraft starts in a circular low-Earth orbit with inclination $i(t_0) = 28.5$ deg and terminates in a highly elliptic low periapsis orbit with inclination $i(t_f) = 63.4$ deg. The continuous-time optimal control problem corresponding to this orbital transfer problem can be stated in Mayer form as follows. Minimize the cost functional

$$J = -w(t_f) \quad (12)$$

subject to the dynamic constraints

$$\dot{\mathbf{x}} = \mathbf{A}(\mathbf{x})\mathbf{\Delta} + \mathbf{b}, \quad (13)$$

$$\dot{w} = -\frac{T(1 + 0.01\tau)}{I_{sp}}, \quad (14)$$

the path constraint

$$\|\mathbf{u}\| = 1, \quad (15)$$

the parameter constraint

$$-50 \leq \tau \leq 0, \quad (16)$$

and the boundary conditions

$$\begin{aligned} p(t_0) &= 21837080.052835 \text{ ft}, & p(t_f) &= 40007346.015232 \text{ ft}, \\ f(t_0) &= 0, & \sqrt{f^2(t_f) + g^2(t_f)} &= 0.73550320568829, \\ g(t_0) &= 0, & \sqrt{h^2(t_f) + k^2(t_f)} &= 0.61761258786099, \\ h(t_0) &= -0.25396764647494, & f(t_f)h(t_f) + g(t_f)k(t_f) &= 0, \\ k(t_0) &= 0, & g(t_f)h(t_f) - k(t_f)f(t_f) &\leq 0, \\ L(t_0) &= \pi \text{ rad}, & w(t_0) &= 1 \text{ lbm}, \\ i(t_0) &= 28.5 \text{ deg}, & i(t_f) &= 63.4 \text{ deg}. \end{aligned} \quad (17)$$

The matrix $\mathbf{A}(\mathbf{x})$ in Eq. (13) is given as

$$\mathbf{A} = \begin{bmatrix} 0 & \frac{2p}{q} \sqrt{\frac{p}{\mu}} & 0 \\ \sqrt{\frac{p}{\mu}} \sin(L) & \sqrt{\frac{p}{\mu}} \frac{1}{q} ((q+1) \cos(L) + f) & -\sqrt{\frac{p}{\mu}} \frac{g}{q} (h \sin(L) - k \cos(L)) \\ -\sqrt{\frac{p}{\mu}} \cos(L) & \sqrt{\frac{p}{\mu}} \frac{1}{q} ((q+1) \sin(L) + g) & \sqrt{\frac{p}{\mu}} \frac{f}{q} (h \sin(L) - k \cos(L)) \\ 0 & 0 & \sqrt{\frac{p}{\mu}} \frac{s^2 \cos(L)}{2q} \\ 0 & 0 & \sqrt{\frac{p}{\mu}} \frac{s^2 \sin(L)}{2q} \\ 0 & 0 & \sqrt{\frac{p}{\mu}} (h \sin(L) - k \cos(L)) \end{bmatrix} \quad (18)$$

while the vector is

$$\mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \sqrt{\mu p} \left(\frac{q}{p}\right)^2 \end{bmatrix}, \quad (19)$$

where

$$\begin{aligned} q &= 1 + f \cos(L) + g \sin(L), & r &= p/q, \\ \alpha^2 &= h^2 - k^2, & \chi &= \sqrt{h^2 + k^2}, \\ s^2 &= 1 + \chi^2. \end{aligned} \quad (20)$$

The spacecraft acceleration is modeled as

$$\mathbf{\Delta} = \mathbf{\Delta}_g + \mathbf{\Delta}_T, \quad (21)$$

where $\mathbf{\Delta}_g$ is the acceleration due to the oblateness of the Earth while $\mathbf{\Delta}_T$ is the thrust specific force. The acceleration due to Earth oblateness is expressed in rotating radial coordinates as

$$\mathbf{\Delta}_g = \mathbf{Q}_r^T \delta \mathbf{g}, \quad (22)$$

where \mathbf{Q}_r is the transformation from rotating radial coordinates to Earth centered inertial coordinates. The matrix \mathbf{Q}_r is given column-wise as

$$\mathbf{Q}_r = [\mathbf{i}_r \quad \mathbf{i}_\theta \quad \mathbf{i}_h], \quad (23)$$

where the basis vectors \mathbf{i}_r , \mathbf{i}_θ , and \mathbf{i}_h are given as

$$\mathbf{i}_r = \frac{\mathbf{r}}{\|\mathbf{r}\|}, \quad \mathbf{i}_h = \frac{\mathbf{r} \times \mathbf{v}}{\|\mathbf{r} \times \mathbf{v}\|}, \quad \mathbf{i}_\theta = \mathbf{i}_h \times \mathbf{i}_r. \quad (24)$$

Furthermore, the vector $\delta \mathbf{g}$ is defined as

$$\delta \mathbf{g} = \delta g_n \mathbf{i}_n - \delta g_r \mathbf{i}_r, \quad (25)$$

where \mathbf{i}_n is the local North direction and is defined as

$$\mathbf{i}_n = \frac{\mathbf{e}_n - (\mathbf{e}_n^T \mathbf{i}_r) \mathbf{i}_r}{\|\mathbf{e}_n - (\mathbf{e}_n^T \mathbf{i}_r) \mathbf{i}_r\|} \quad (26)$$

and $\mathbf{e}_n = (0, 0, 1)$. The oblate earth perturbations are then expressed as

$$\delta g_r = -\frac{\mu}{r^2} \sum_{k=2}^4 (k+1) \left(\frac{R_e}{r}\right)^k P_k(s) J_k, \quad (27)$$

$$\delta g_n = -\frac{\mu \cos(\phi)}{r^2} \sum_{k=2}^4 \left(\frac{R_e}{r}\right)^k P'_k(s) J_k, \quad (28)$$

where R_e is the equatorial radius of the earth, $P_k(s)$ ($s \in [-1, +1]$) is the k^{th} -degree Legendre polynomial, P'_k is the derivative of P_k with respect to s , $s = \sin(\phi)$, and J_k represents the zonal harmonic coefficients for $k = (2, 3, 4)$. Next, the acceleration due to thrust is given as

$$\mathbf{\Delta}_T = \frac{g_0 T (1 + 0.01 \tau)}{w} \mathbf{u}. \quad (29)$$

Finally, the physical constants used in the problem are given as

$$\begin{aligned} I_{sp} &= 450 \text{ s}, & T &= 4.446618 \times 10^{-3} \text{ lbf}, \\ g_0 &= 32.174 \text{ ft/s}^2, & \mu &= 1.407645794 \times 10^{16} \text{ ft}^3/\text{s}^2, \\ R_e &= 20925662.73 \text{ ft}, & J_2 &= 1082.639 \times 10^{-6}, \\ J_3 &= -2.565 \times 10^{-6}, & J_4 &= -1.608 \times 10^{-6}. \end{aligned} \quad (30)$$

5.3 Initial Guess Generation

The initial guess was obtained using a variable step ordinary differential equation solver by propagating the modified equinoctial dynamics from the initial conditions to a final time of 9×10^4 s with a throttle parameter

of $\tau = -25$. In computing this initial guess the control direction was assumed to lie along the direction of inertial velocity, that is,

$$\mathbf{u} = \mathbf{Q}_r^T \frac{\mathbf{v}}{\|\mathbf{v}\|}. \quad (31)$$

The values at the desired collocation points were found using interpolation to form the initial guess. These collocation points generated from an initial mesh of M intervals each containing a specified number of Radau collocation points. The number of intervals M was calculated by

$$M = \text{ceil}(L_f) - 1. \quad (32)$$

The intervals were then evenly spaced on the domain of the longitude L as

$$L_m = L_i + \frac{m-1}{M-1}(L_f - L_i) \quad (33)$$

for $m = 1, \dots, M+1$ and where L_i and L_f are the initial and terminal values of the true longitude L . Finally, the evenly spaced intervals on the L domain were converted to corresponding intervals on the time t domain. The desired number of collocation points were then used within each mesh interval. It is noted that the approach described here for obtaining the initial initial mesh provides an increased density of collocation points near periapsis of the trajectory.

6 Results

The MATLAB code that solves the low-thrust orbit transfer problem using GPOPS – III 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 the problem. Solutions to the low-thrust orbital transfer problem are shown in Figs. 2 and 3. Furthermore, a three-dimensional view of the optimal orbital transfer is shown in Fig. 4.

```

% ----- Low-Thrust Orbit Transfer 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. %
% -----%
clc
clear all
close all

% -----%
% Constants and initial conditions %
% -----%
T = 4.446618e-3; % [lb]
Isp = 450; % [s]
mu = 1.407645794e16; % [ft^3/s^2]
gs = 32.174; % [ft/s^2]
Re = 20925662.73; % [ft]
J2 = 1082.639e-6;
J3 = -2.565e-6;
J4 = -1.608e-6;

p0 = 21837080.052835; % [ft]
f0 = 0;
g0 = 0;
h0 = -0.25396764647494;
k0 = 0;
L0 = pi;
w0 = 1;

```

```

tau = -25;

t0 = 0;
tf = 90000;

% ----- %
%           Set up auxiliary data for problem           %
% ----- %
auxdata.Isp = Isp; % [s]
auxdata.mu = mu; % [ft^3/s^2]
auxdata.gs = gs; % [ft/s^2]
auxdata.T = T; % [lb]
auxdata.Re = Re; % [ft]
auxdata.J2 = J2;
auxdata.J3 = J3;
auxdata.J4 = J4;

% ----- %
%           Generate initial guess for problem           %
% ----- %
initial.p0 = p0;
initial.f0 = f0;
initial.g0 = g0;
initial.h0 = h0;
initial.k0 = k0;
initial.L0 = L0;
initial.w0 = w0;
initial.t0 = t0;

guess.tau = tau;
guess.tf = tf;

initialguess = lowThrustPropagate(auxdata,initial,guess);

% ----- %
%           Set up bounds on state, control, and time           %
% ----- %
t0 = 0; tmin = t0; tfmin = 50000; tmax = 100000;
p0 = 21837080.052835; pf = 40007346.015232;
f0 = 0;
g0 = 0;
h0 = -0.25396764647494;
k0 = 0;
L0 = pi;
w0 = 1;
pmin = 20000000; pmax = 60000000;
fmin = -1; fmax = +1;
gmin = -1; gmax = +1;
hmin = -1; hmax = +1;
kmin = -1; kmax = +1;
Lmin = L0; Lmax = 9*2*pi;
wmin = 0.1; wmax = w0;
urmin = -1; urmax = +1;
utmin = -1; utmax = +1;
uhmin = -1; uhmax = +1;
taumin = -50; taumax = 0;

bounds.phase.initialtime.lower = t0;
bounds.phase.initialtime.upper = t0;
bounds.phase.finaltime.lower = tfmin;
bounds.phase.finaltime.upper = tmax;

bounds.phase.initialstate.lower = [p0,f0,g0,h0,k0,L0,w0];
bounds.phase.initialstate.upper = [p0,f0,g0,h0,k0,L0,w0];
bounds.phase.state.lower = [pmin,fmin,gmin,hmin,kmin,Lmin,wmin];
bounds.phase.state.upper = [pmax,fmax,gmax,hmax,kmax,Lmax,wmax];

```

```

bounds.phase.finalstate.lower = [pf, fmin, gmin, hmin, kmin, Lmin, wmin];
bounds.phase.finalstate.upper = [pf, fmax, gmax, hmax, kmax, Lmax, wmax];

bounds.phase.control.lower = [urmin, utmin, uhmin];
bounds.phase.control.upper = [urmax, utmax, uhmax];

bounds.parameter.lower = taumin;
bounds.parameter.upper = taumax;

bounds.phase.path.lower = 0;
bounds.phase.path.upper = 0;

bounds.eventgroup.lower = [0.73550320568829^2, 0.61761258786099^2, 0, -3];
bounds.eventgroup.upper = [0.73550320568829^2, 0.61761258786099^2, 0, 0];

% ----- %
% Generate an Initial Guess by Propagating the Dynamics %
% ----- %
guess = initialguess;

% ----- %
% Set Up an Initial Mesh Using the Result from the Propagated Guess %
% ----- %
mesh.method = 'hp-LiuRao-Legendre';
mesh.tolerance = 1e-5;
mesh.maxiterations = 10;
mesh.colpointsmax = 6;
mesh.colpointsmmin = 4;
N = length(guess.fraction);
mesh.phase.colpoints = 4*ones(1,N);
mesh.phase.fraction = guess.fraction;

% ----- %
% Set up solver %
% ----- %
setup.name = 'Betts-Low-Thrust-Orbit-Transfer';
setup.functions.continuous = @lowThrustContinuous;
setup.functions.endpoint = @lowThrustEndpoint;
setup.nlp.solver = 'ipopt';
setup.nlp.ipoptoptions.linear_solver = 'ma57';
setup.auxdata = auxdata;
setup.bounds = bounds;
setup.guess = guess;
setup.derivatives.supplier = 'adigator';
setup.derivatives.derivativelevel = 'second';
setup.derivatives.dependencies = 'sparseNaN';
setup.derivatives.stepsize = 1e-6;
setup.scales.method = 'automatic-bounds';
setup.mesh = mesh;
setup.method = 'RPM-Differentiation';

% ----- %
% Solve problem and extract solution %
% ----- %
tic;
output = gpops2(setup);
gpopsCPU = toc;

solution = output.result.solution;

function phaseout = lowThrustContinuous(input)

% ----- %
% Extract auxiliary data for problem %
% ----- %

```



```

Isp = input.auxdata.Isp; % [s]
mu = input.auxdata.mu; % [ft^3/s^2]
gs = input.auxdata.gs; % [ft/s^2]
T = input.auxdata.T; % [lb]
Re = input.auxdata.Re; % [ft]
J2 = input.auxdata.J2;
J3 = input.auxdata.J3;
J4 = input.auxdata.J4;

% ----- %
% Extract state, control, and parameter for problem
% ----- %
p = input.phase.state(:,1);
f = input.phase.state(:,2);
g = input.phase.state(:,3);
h = input.phase.state(:,4);
k = input.phase.state(:,5);
L = input.phase.state(:,6);
w = input.phase.state(:,7);

ur = input.phase.control(:,1);
ut = input.phase.control(:,2);
uh = input.phase.control(:,3);

tau = input.phase.parameter(:,1);

% ----- %
% Gravitational disturbing acceleration
% ----- %
q = 1+f.*cos(L)+g.*sin(L);
r = p./q;
alpha2 = h.*h-k.*k;
chi = sqrt(h.*h+k.*k);
s2 = 1+chi.*chi;

rX = (r./s2).*(cos(L)+alpha2.*cos(L)+2*h.*k.*sin(L));
rY = (r./s2).*(sin(L)-alpha2.*sin(L)+2*h.*k.*cos(L));
rZ = (2*r./s2).*(h.*sin(L)-k.*cos(L));
rVec = [rX rY rZ];
rMag = sqrt(rX.^2+rY.^2+rZ.^2);
rXZMag = sqrt(rX.^2+rZ.^2);

vX = -(1./s2).*sqrt(mu./p).*(sin(L)+alpha2.*sin(L)-2*h.*k.*cos(L)+g-2*f.*h.*k+alpha2.*g);
vY = -(1./s2).*sqrt(mu./p).*(-cos(L)+alpha2.*cos(L)+2*h.*k.*sin(L)-f+2*g.*h.*k+alpha2.*f);
vZ = (2./s2).*sqrt(mu./p).*(h.*cos(L)+k.*sin(L)+f.*h+g.*k);
vVec = [vX vY vZ];

rCrossv = cross(rVec,vVec,2);
rCrossvMag = sqrt(rCrossv(:,1).^2+rCrossv(:,2).^2+rCrossv(:,3).^2);
rCrossvCrossr = cross(rCrossv,rVec,2);

ir1 = rVec(:,1)./rMag;
ir2 = rVec(:,2)./rMag;
ir3 = rVec(:,3)./rMag;
ir = [ir1 ir2 ir3];

it1 = rCrossvCrossr(:,1)./(rCrossvMag.*rMag);
it2 = rCrossvCrossr(:,2)./(rCrossvMag.*rMag);
it3 = rCrossvCrossr(:,3)./(rCrossvMag.*rMag);
it = [it1 it2 it3];

ih1 = rCrossv(:,1)./rCrossvMag;
ih2 = rCrossv(:,2)./rCrossvMag;
ih3 = rCrossv(:,3)./rCrossvMag;
ih = [ih1 ih2 ih3];

enir = ir3;

```

```

enirir1 = enir.*ir1;
enirir2 = enir.*ir2;
enirir3 = enir.*ir3;

enenirir1 = 0-enirir1;
enenirir2 = 0-enirir2;
enenirir3 = 1-enirir3;
enenirMag = sqrt(enenirir1.^2+enenirir2.^2+enenirir3.^2);

in1 = enenirir1./enenirMag;
in2 = enenirir2./enenirMag;
in3 = enenirir3./enenirMag;

% Geocentric latitude
sinphi = rZ./rXZMag;
cosphi = sqrt(1-sinphi.^2);

% Legendre polynomials
P2 = (3*sinphi.^2-2)./2;
P3 = (5*sinphi.^3-3*sinphi)./2;
P4 = (35*sinphi.^4-30*sinphi.^2+3)./8;
dP2 = 3*sinphi;
dP3 = (15*sinphi-3)./2;
dP4 = (140*sinphi.^3-60*sinphi)./8;

% Oblate earth perturbations
sumn = (Re./r).^2.*dP2.*J2+(Re./r).^3.*dP3.*J3+(Re./r).^4.*dP4.*J4;
sumr = (2+1)*(Re./r).^2.*P2.*J2+(3+1)*(Re./r).^3.*P3.*J3+(4+1)*(Re./r).^4.*P4.*J4;
dgn = -(mu*cosphi./(r.^2)).*sumn;
dgr = -(mu./(r.^2)).*sumr;

dgnin1 = dgn.*in1;
dgnin2 = dgn.*in2;
dgnin3 = dgn.*in3;

dgrir1 = dgr.*ir1;
dgrir2 = dgr.*ir2;
dgrir3 = dgr.*ir3;

dgl = dgnin1 - dgrir1;
dg2 = dgnin2 - dgrir2;
dg3 = dgnin3 - dgrir3;

Deltag1 = ir(:,1).*dgl+ir(:,2).*dg2+ir(:,3).*dg3;
Deltag2 = it(:,1).*dgl+it(:,2).*dg2+it(:,3).*dg3;
Deltag3 = ih(:,1).*dgl+ih(:,2).*dg2+ih(:,3).*dg3;

% ----- %
% Thrust acceleration
% ----- %
DeltaT1 = ((gs*T*(1+0.01*tau))./w).*ur;
DeltaT2 = ((gs*T*(1+0.01*tau))./w).*ut;
DeltaT3 = ((gs*T*(1+0.01*tau))./w).*uh;

% ----- %
% Total acceleration
% ----- %
Delta1 = Deltag1+DeltaT1;
Delta2 = Deltag2+DeltaT2;
Delta3 = Deltag3+DeltaT3;

% ----- %
% Differential equations of motion
% ----- %
dp = (2*p./q).*sqrt(p./mu).*Delta2;

```

```

df = sqrt(p./mu).*sin(L).*Delta1 ...
    +sqrt(p./mu.*(1./q).*(q+1).*cos(L)+f).*Delta2 ...
    -sqrt(p./mu.*(g./q).*(h.*sin(L)-k.*cos(L)).*Delta3;

dg = -sqrt(p./mu).*cos(L).*Delta1 ...
    +sqrt(p./mu.*(1./q).*(q+1).*sin(L)+g).*Delta2 ...
    +sqrt(p./mu.*(f./q).*(h.*sin(L)-k.*cos(L)).*Delta3;

dh = sqrt(p./mu).*(s2.*cos(L)./(2*q)).*Delta3;

dk = sqrt(p./mu).*(s2.*sin(L)./(2*q)).*Delta3;

dL = sqrt(p./mu).*(1./q).*(h.*sin(L)-k.*cos(L)).*Delta3...
    +sqrt(mu.*p).*(q./p).^2);

dw = -(T*(1+0.01*tau)/Isp);

% ----- %
% Declare phaseout
% ----- %
phaseout.dynamics = [dp,df,dg,dh,dk,dL,dw];
phaseout.path = ur.^2+ut.^2+uh.^2-1;

function output = lowThrustEndpoint(input)

ff = input.phase.finalstate(2);
gf = input.phase.finalstate(3);
hf = input.phase.finalstate(4);
kf = input.phase.finalstate(5);

wf = input.phase.finalstate(7);

% Declare objective (minimize final weight)
output.objective = -wf;

% Declare event constraints
output.eventgroup.event = [ff^2+gf^2 hf^2+kf^2 ff*hf+gf*kf gf*hf-kf*ff];

```

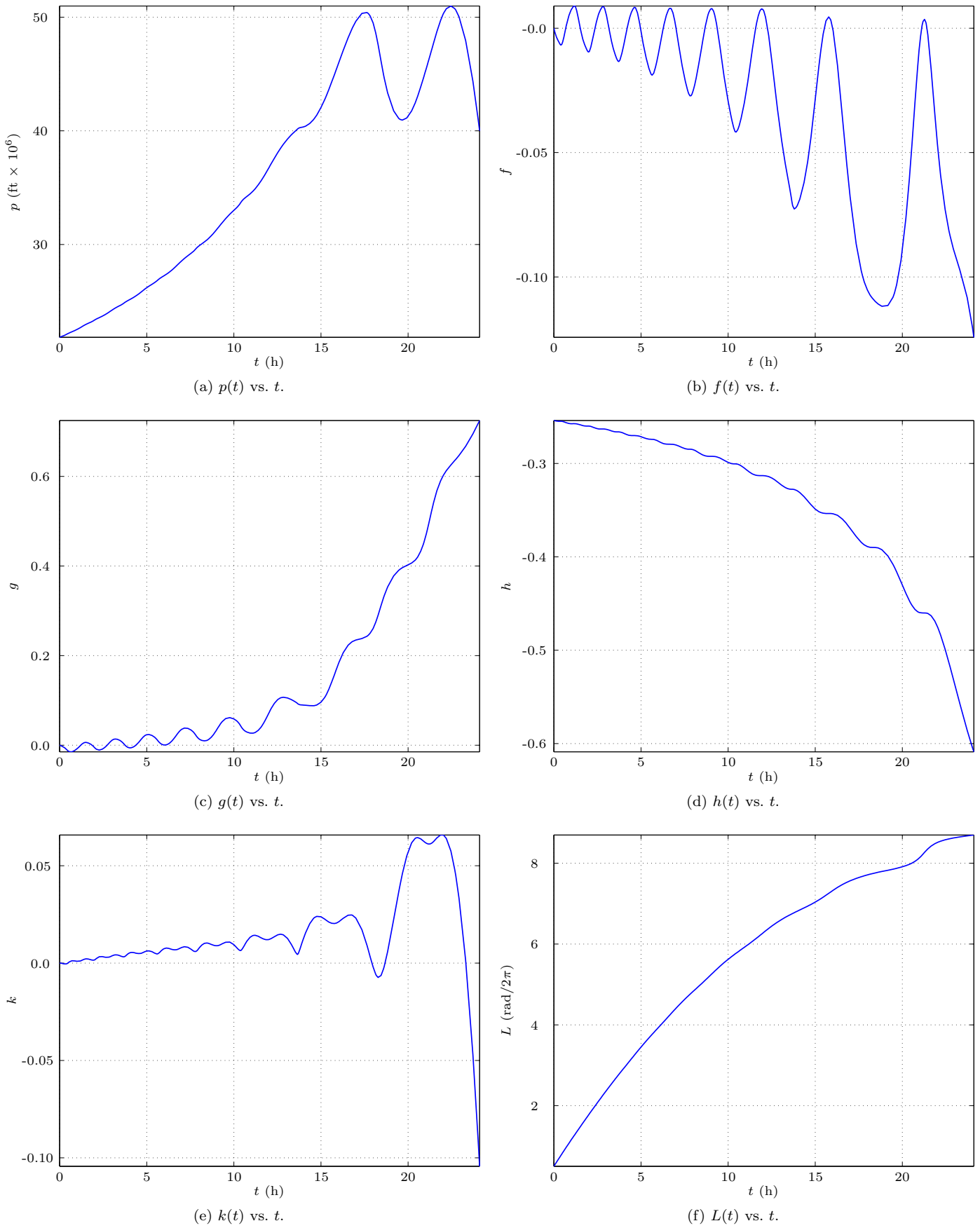


Figure 2: Low-thrust transfer-state variables.

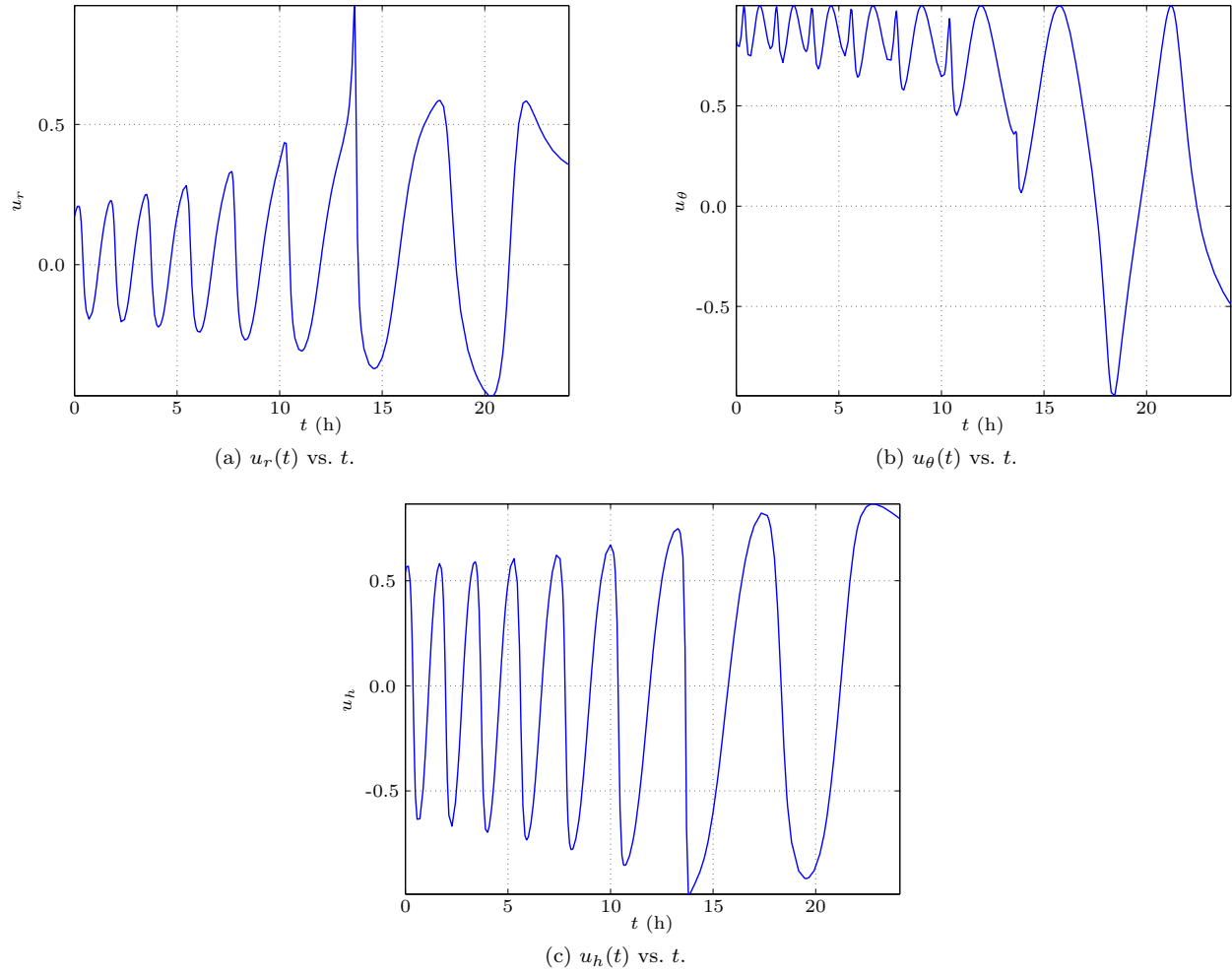


Figure 3: Low-thrust transfer-control variables.

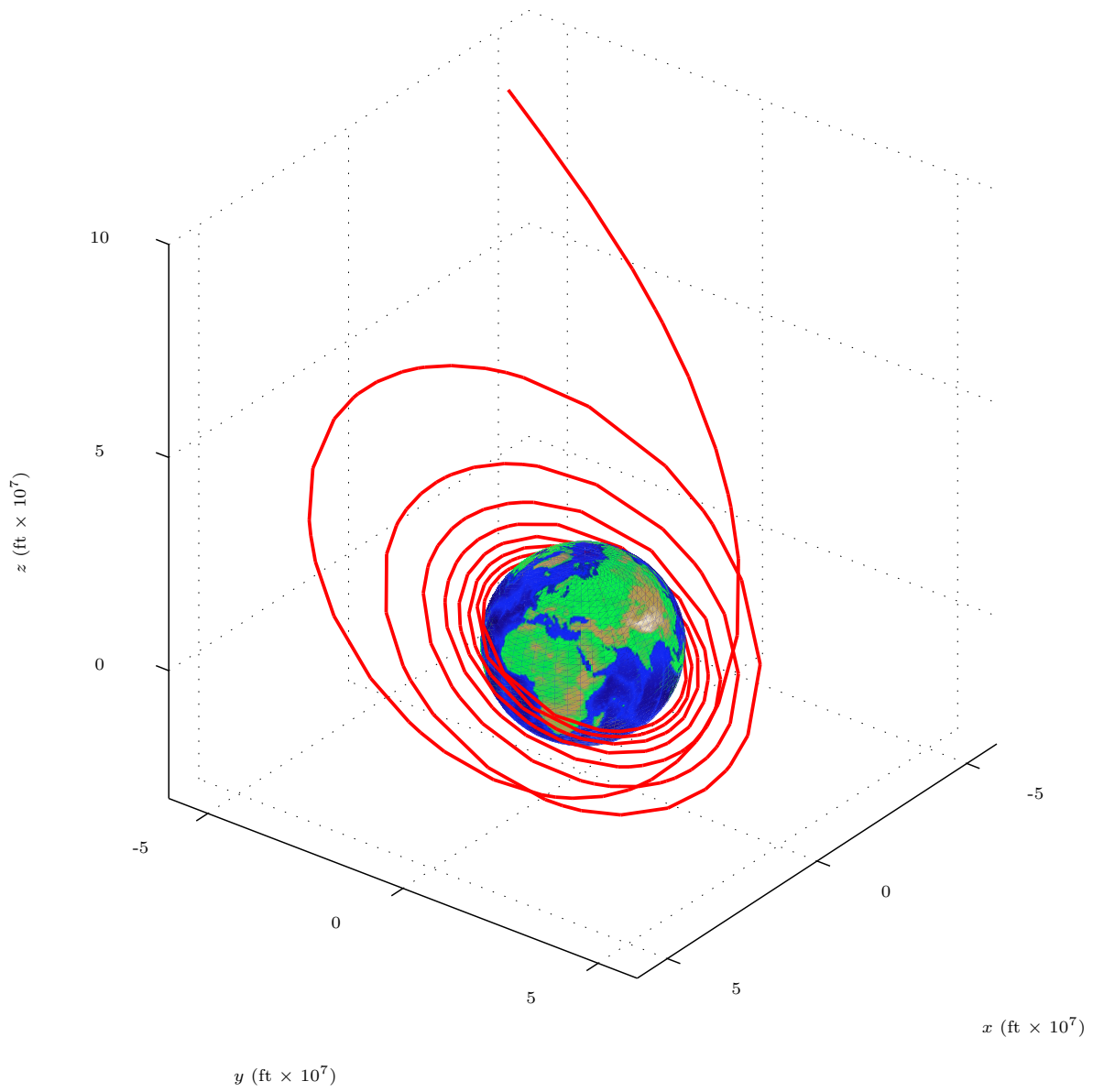


Figure 4: Optimal low-thrust transfer.

6.1 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 found verbatim in Refs. 14, 15, and 12.

6.1.1 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 vehicle 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 terminates 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. 15 or 12 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

6.1.2 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

$$\begin{aligned}
 \dot{\mathbf{r}} &= \mathbf{v} \\
 \dot{\mathbf{v}} &= -\frac{\mu}{\|\mathbf{r}\|^3}\mathbf{r} + \frac{T}{m}\mathbf{u} + \frac{\mathbf{D}}{m} \\
 \dot{m} &= -\frac{T}{g_0 I_{sp}}
 \end{aligned} \tag{34}$$

where $\mathbf{r}(t) = [x(t) \ y(t) \ z(t)]^T$ is the position, $\mathbf{v} = [v_x(t) \ v_y(t) \ v_z(t)]^T$ 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, $\mathbf{u} = [u_x \ u_y \ u_z]^T$ is the thrust direction, and $\mathbf{D} = [D_x \ D_y \ D_z]^T$ is the drag force. The drag force is defined as

$$\mathbf{D} = -\frac{1}{2}C_D A_{ref} \rho \|\mathbf{v}_{rel}\| \mathbf{v}_{rel} \tag{35}$$

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

$$\mathbf{v}_{rel} = \mathbf{v} - \boldsymbol{\omega} \times \mathbf{r} \tag{36}$$

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

$$\rho = \rho_0 \exp[-h/h_0] \tag{37}$$

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

6.1.3 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

$$\begin{aligned} \mathbf{r}(t_0) &= \mathbf{r}_0 = [5605.2 \quad 0 \quad 3043.4]^T \text{ km} \\ \mathbf{v}(t_0) &= \mathbf{v}_0 = [0 \quad 0.4076 \quad 0]^T \text{ km/s} \\ m(t_0) &= m_0 = 301454 \text{ kg} \end{aligned} \quad (38)$$

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

$$\begin{aligned} a_f &= 24361.14 \text{ km}, \\ e_f &= 0.7308, \\ i_f &= 28.5 \text{ deg}, \\ \Omega_f &= 269.8 \text{ deg}, \\ \omega_f &= 130.5 \text{ deg} \end{aligned} \quad (39)$$

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.²³

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

$$|\mathbf{r}| \geq R_r \quad (40)$$

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

$$\|\mathbf{u}\|_2^2 = u_1^2 + u_2^2 + u_3^2 = 1 \quad (41)$$

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

$$\begin{aligned} \mathbf{r}^{(p)}(t_f) - \mathbf{r}^{(p+1)}(t_0) &= \mathbf{0}, \\ \mathbf{v}^{(p)}(t_f) - \mathbf{v}^{(p+1)}(t_0) &= \mathbf{0}, \quad (p = 1, \dots, 3) \\ m^{(p)}(t_f) - m_{dry}^{(p)} - m^{(p+1)}(t_0) &= 0 \end{aligned} \quad (42)$$

where the superscript (p) represents the phase number.

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

$$J = -m^{(4)}(t_f) \quad (43)$$

subject to the conditions of Eqs. (34), (38), (39), (40), and (41).

The MATLAB code that solves the multiple-stage launch vehicle ascent problem using GPOPS – III 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 1,
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.pressure  = scales.force/scales.area;
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.';
unitr0             = r0/norm(r0,2);
speedrel0         = 5/scales.speed;
v0                 = v0 + speedrel0*unitr0;

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 = -10*ones(1,3);
bounds.phase(iphase).control.upper = +10*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 = -10*ones(1,3);
bounds.phase(iphase).control.upper = +10*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 = -10*ones(1,3);
bounds.phase(iphase).control.upper = +10*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) = [1; 1];
guess.phase(iphase).control(:,2) = [0; 0];
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 = -10*ones(1,3);
bounds.phase(iphase).control.upper = +10*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) = [1; 1];
guess.phase(iphase).control(:,2) = [0; 0];
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 -----%
%-----%
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 = 'ipopt';
setup.nlp.snoptoptions.tolerance = 1e-7;
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 = 'hp-PattersonRao';
setup.mesh.tolerance = 1e-6;
setup.method = 'RPM-Differentiation';

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

%-----%
%----- 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.';
vrell = v1-omegacrossr;
speedrell = sqrt(sum(vrell.*vrell,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.*speedrell;

```

```

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;
mldot1 = -TSrb1./(input.auxdata.g0*input.auxdata.ispSrb);
m2dot1 = -TFirst1./(input.auxdata.g0*input.auxdata.ispFirst);
mdot1 = mldot1+m2dot1;
q1 = 1/2*rho1.*speedrel1.^2;
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;
mldot2 = -TSrb2./(input.auxdata.g0*input.auxdata.ispSrb);
m2dot2 = -TFirst2./(input.auxdata.g0*input.auxdata.ispFirst);
mdot2 = mldot2+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
t01 = input.phase(1).initialtime;
tf1 = input.phase(1).finaltime;
x01 = input.phase(1).initialstate;
xf1 = input.phase(1).finalstate;
% Variables at Start and Terminus of Phase 2
t02 = input.phase(2).initialtime;
tf2 = input.phase(2).finaltime;
x02 = input.phase(2).initialstate;
xf2 = input.phase(2).finalstate;
% Variables at Start and Terminus of Phase 3
t03 = input.phase(3).initialtime;
tf3 = input.phase(3).finaltime;
x03 = input.phase(3).initialstate;
xf3 = input.phase(3).finalstate;
% Variables at Start and Terminus of Phase 4
t04 = input.phase(4).initialtime;
tf4 = input.phase(4).finaltime;
x04 = input.phase(4).initialstate;
xf4 = input.phase(4).finalstate;

% Event Group 1: Linkage Constraints Between Phases 1 and 2
output.eventgroup(1).event = [x02(1:7)-xf1(1:7), t02-tf1];
% Event Group 2: Linkage Constraints Between Phases 2 and 3
output.eventgroup(2).event = [x03(1:7)-xf2(1:7), t03-tf2];
% Event Group 3: Linkage Constraints Between Phases 3 and 4
output.eventgroup(3).event = [x04(1:7)-xf3(1:7), t04-tf3];
% Event Group 4: Constraints on Terminal Orbit
orbitalElements = launchrv2oe(xf4(1:3).',xf4(4:6).',input.auxdata.mu);
output.eventgroup(4).event = orbitalElements(1:5).';
output.objective = -xf4(7);

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

% -----%
% Begin File: launchEvents.m %
% -----%
function output = launchEvents(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;
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 4
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 File: launchEvents.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 = [c0*co-s0*so*ci  -c0*so-s0*co*ci  s0*si;
      s0*co+c0*so*ci  -s0*so+c0*co*ci  -c0*si;
      s0*si            co*si            ci];
ri = R*rv;
vi = R*vv;

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

```

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

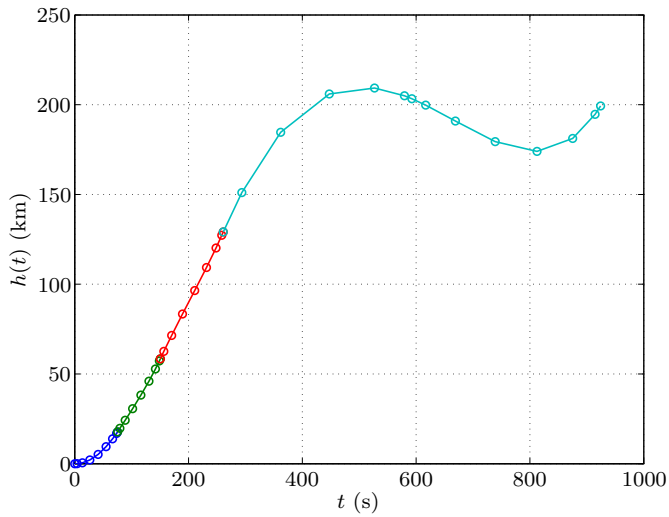
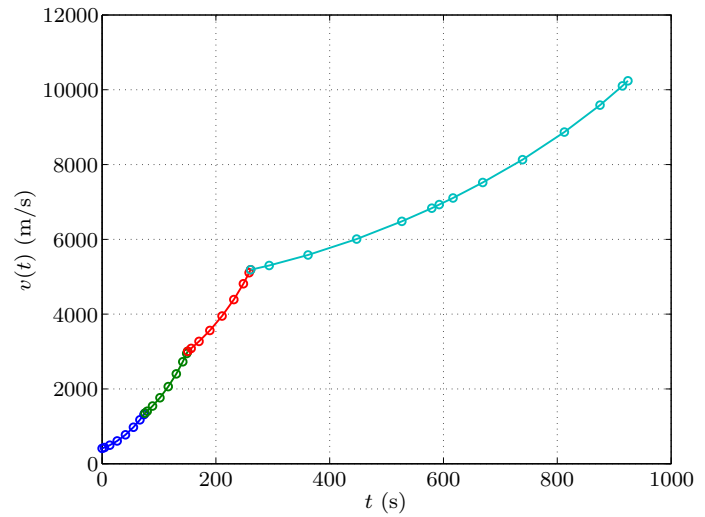
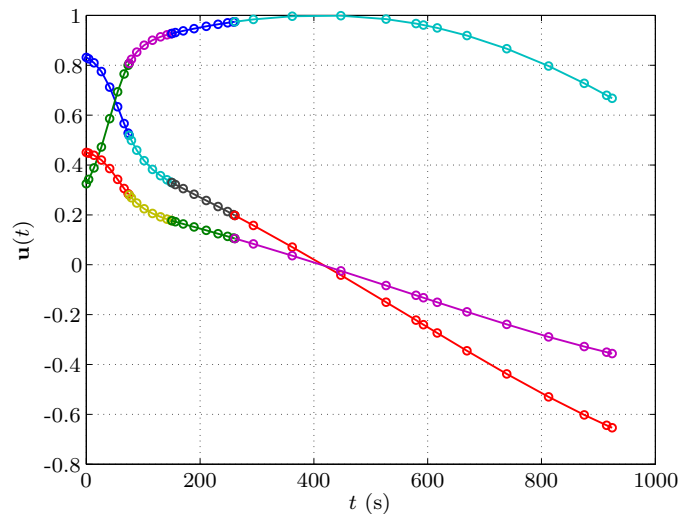
(a) $h(t)$ vs. t .(b) $v(t)$ vs. t .(c) $\mathbf{u}(t)$ vs. t .

Figure 5: Solution to Launch Vehicle Ascent Problem Using `GPOPS – III` with the NLP Solver `SNOPT` and a Mesh Refinement Tolerance of 10^{-7} .

6.2 Tumor-Antiangiogenesis Optimal Control Problem

Consider the following cancer treatment optimal control problem taken from Ref. 16. The objective is to minimize

$$p(t_f) \quad (44)$$

subject to the dynamic constraints

$$\begin{aligned} \dot{p}(t) &= -\xi p(t) \ln\left(\frac{p(t)}{q(t)}\right), \\ \dot{q}(t) &= q(t) [b - \mu - dp^{2/3}(t) - Gu(t)], \end{aligned} \quad (45)$$

with the initial conditions

$$\begin{aligned} p(0) &= p_0, \\ q(0) &= q_0, \end{aligned} \quad (46)$$

and the integral constraint

$$\int_0^{t_f} u(\tau) d\tau \leq A. \quad (47)$$

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];

%-----Provide Mesh Refinement Method and Initial Mesh -----%
mesh.method = 'hp-LiuRao-Legendre';
mesh.tolerance = 1e-6;
mesh.phase.colpoints = 4*ones(1,10);
mesh.phase.fraction = 0.1*ones(1,10);

%----- Problem Setup -----%
setup.name = 'Tumor-Anti-Angiogenesis-Problem';
setup.functions.continuous = @tumorAntiAngiogenesisContinuous;
setup.functions.endpoint = @tumorAntiAngiogenesisEndpoint;
setup.displaylevel = 2;
setup.auxdata = auxdata;
setup.bounds = bounds;
setup.guess = guess;
setup.mesh = mesh;
setup.nlp.solver = 'ipopt';
setup.derivatives.supplier = 'sparseCD';
setup.derivatives.derivativelevel = 'second';
setup.scales.method = 'none';
setup.method = 'RPM-Differentiation';

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

function phaseout = tumorAntiAngiogenesisContinuous(input)

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

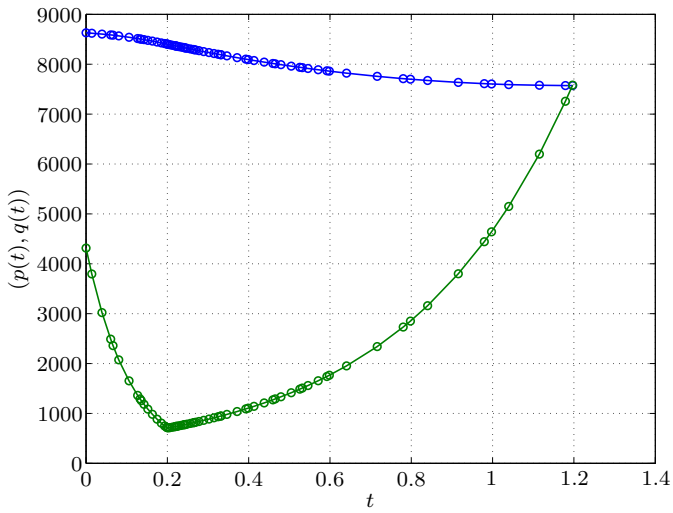
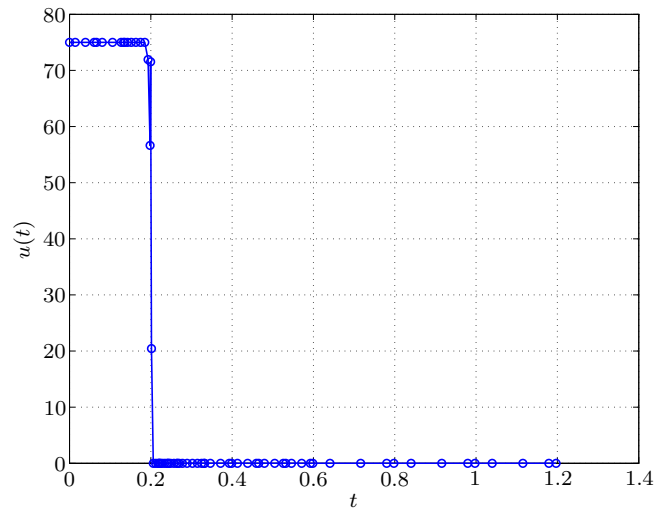
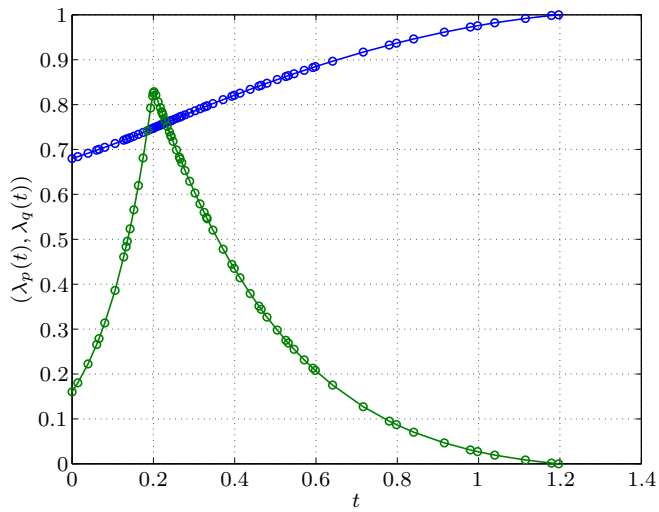
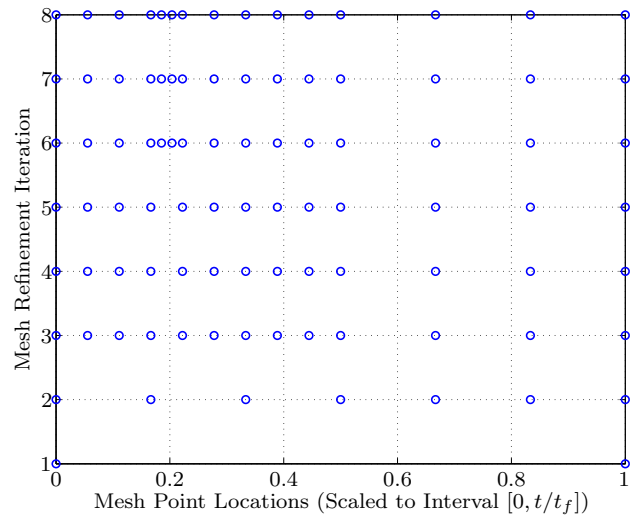
t = input.phase(1).time;
x = input.phase(1).state;
u = input.phase(1).control;

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 `GPOPS-III` using the NLP solver IPOPT with a mesh refinement error tolerance of 10^{-6} is shown in Figs. 6a–6c. Note that in this example we have also provided the costate of the optimal control problem, where the costate is estimated using the Radau orthogonal collocation costate estimation method described in Refs. 2, 3, and 4.

(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 6: Solution to Tumor Anti-Angiogenesis Optimal Control Problem Using GPOPS-III with the NLP Solver SNOPT and a Mesh Refinement Tolerance of 10^{-6} .

6.3 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. ¹² Minimize the cost functional

$$J = -\phi(t_f) \quad (48)$$

subject to the dynamic constraints

$$\begin{aligned} \dot{r} &= v \sin \gamma, \\ \dot{\theta} &= \frac{v \cos \gamma \sin \psi}{r \cos \phi}, \\ \dot{\phi} &= \frac{v \cos \gamma \cos \psi}{r}, \\ \dot{v} &= -\frac{F_d}{m} - F_g \sin \gamma, \\ \dot{\gamma} &= \frac{F_l \cos \sigma}{mv} - \left(\frac{F_g}{v} - \frac{v}{r} \right) \cos \gamma, \\ \dot{\psi} &= \frac{F_l \sin \sigma}{mv \cos \gamma} + \frac{v \cos \gamma \sin \psi \tan \phi}{r}, \end{aligned} \quad (49)$$

and the boundary conditions

$$\begin{aligned} r(0) &= 79248 + R_e \text{ m} & , & & r(t_f) &= 24384 + R_e \text{ m}, \\ \theta(0) &= 0 \text{ deg} & , & & \theta(t_f) &= \text{Free}, \\ \phi(0) &= 0 \text{ deg} & , & & \phi(t_f) &= \text{Free}, \\ v(0) &= 7803 \text{ m/s} & , & & v(t_f) &= 762 \text{ m/s}, \\ \gamma(0) &= -1 \text{ deg} & , & & \gamma(t_f) &= -5 \text{ deg}, \\ \psi(0) &= 90 \text{ deg} & , & & \psi(t_f) &= \text{Free}. \end{aligned} \quad (50)$$

Further details of this problem, including the aerodynamic model, can be found in Ref. ¹². 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;

%----- Provide Auxiliary Data for Problem -----%
%-----%
auxdata.Re = 6371203.92; % Equatorial Radius of Earth (m)
auxdata.S = 249.9091776; % Vehicle Reference Area (m^2)
auxdata.cl = [-0.2070 1.6756]; % Parameters for Lift Coefficient
auxdata.cd = [0.0785 -0.3529 2.0400]; % Parameters for Drag Coefficient
auxdata.b = [0.07854 -0.061592 0.00621408]; % Parameters for Heat Rate Model
auxdata.H = 7254.24; % Density Scale Height (m)
auxdata.al = [-0.20704 0.029244]; % Parameters for Heat Rate Model
auxdata.rho0 = 1.225570827014494; % Sea Level Atmospheric Density (kg/m^3)
auxdata.mu = 3.986031954093051e14; % Earth Gravitational Parameter (m^3/s^2)
auxdata.mass = 92079.2525560557; % Vehicle Mass (kg)
```

```

%-----%
%----- Boundary Conditions -----%
%-----%
t0      = 0;
alt0    = 79248;
rad0    = alt0+auxdata.Re;
altf    = +24384;
radf    = altf+auxdata.Re;
lon0    = 0;
lat0    = 0;
speed0  = +7802.88;
speedf  = +762;
fpa0    = -1*pi/180;
fpaf    = -5*pi/180;
azi0    = +90*pi/180;
azif    = -90*pi/180;

%-----%
%----- Limits on Variables -----%
%-----%
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;

%-----%
%----- Set Up Problem Using Data Provided Above -----%
%-----%
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];

%-----%
%----- Provide Guess of Solution -----%
%-----%
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;

%-----%
%----- Provide Mesh Refinement Method and Initial Mesh -----%
%-----%
mesh.method      = 'hp-LiuRao-Legendre';
mesh.maxiterations = 2;

```



```

mesh.colpointsmmin = 3;
mesh.colpointsmmax = 20;
mesh.tolerance     = 1e-7;

%-----%
%----- Configure Setup Using the information provided -----%
%-----%
setup.name          = 'Reusable-Launch-Vehicle-Entry-Problem';
setup.functions.continuous = @rlvEntryContinuous;
setup.functions.endpoint   = @rlvEntryEndpoint;
setup.auxdata         = auxdata;
setup.bounds         = bounds;
setup.guess          = guess;
setup.mesh           = mesh;
setup.displaylevel   = 2;
setup.nlp.solver     = 'ipopt';
setup.nlp.ipoptoptions.linear_solver = 'ma57';
setup.derivatives.supplier = 'adigator';
setup.derivatives.derivativelevel = 'second';
setup.scales.method  = 'automatic-bounds';
setup.method         = 'RPM-Differentiation';

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

function phaseout = rlvEntryContinuous(input)

% -----%
% ----- Extract Each Component of the State -----%
% -----%
rad = input.phase.state(:,1);
lon = input.phase.state(:,2);
lat = input.phase.state(:,3);
v   = input.phase.state(:,4);
fpa = input.phase.state(:,5);
azi = input.phase.state(:,6);
aoa = input.phase.control(:,1);
bank = input.phase.control(:,2);

% -----%
% ----- Compute the Aerodynamic Quantities -----%
% -----%
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;
q       = 0.5*rho.*v.^2;
D       = q.*S.*CD./mass;
L       = q.*S.*CL./mass;
gravity = mu./rad.^2;

% -----%
% ----- Evaluate Right-Hand Side of the Dynamics -----%

```

```

% -----%
raddot = v.*sin(fpa);
londot = v.*cos(fpa).*sin(azi)./(rad.*cos(lat));
latdot = v.*cos(fpa).*cos(azi)./rad;
vdot = -D-gravity.*sin(fpa);
fpadot = (L.*cos(bank)-cos(fpa).*(gravity-v.^2./rad))./v;
azidot = (L.*sin(bank)./cos(fpa)+v.^2.*cos(fpa).*sin(azi).*tan(lat)./rad)./v;

phaseout.dynamics = [raddot, londot, latdot, vdot, 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 **GPOPS – III** using the NLP solver IPOPT with a mesh refinement tolerance of 10^{-6} and the solution is shown in Figs. 7a–7f.

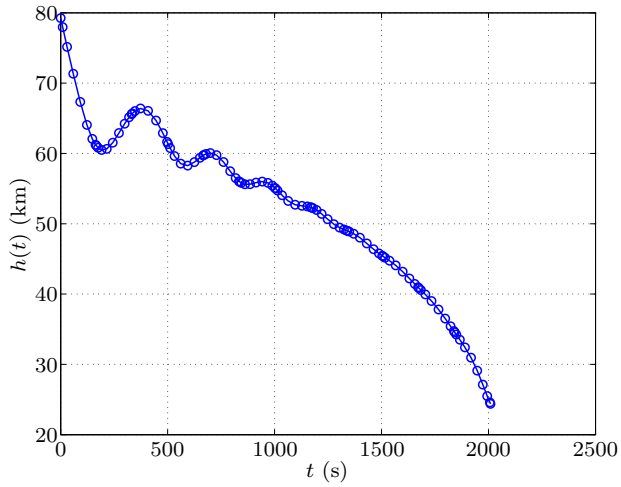
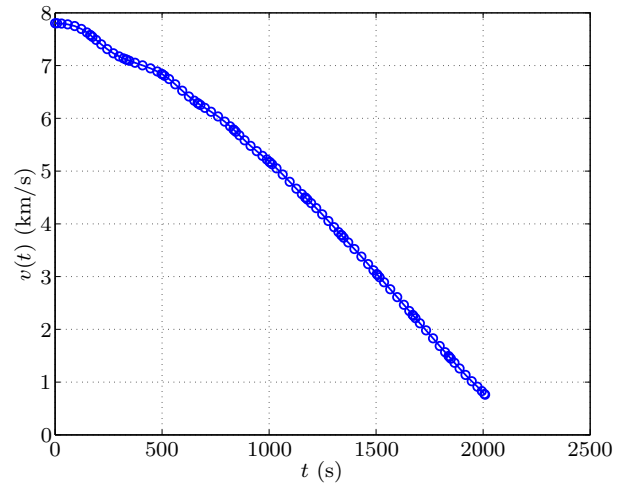
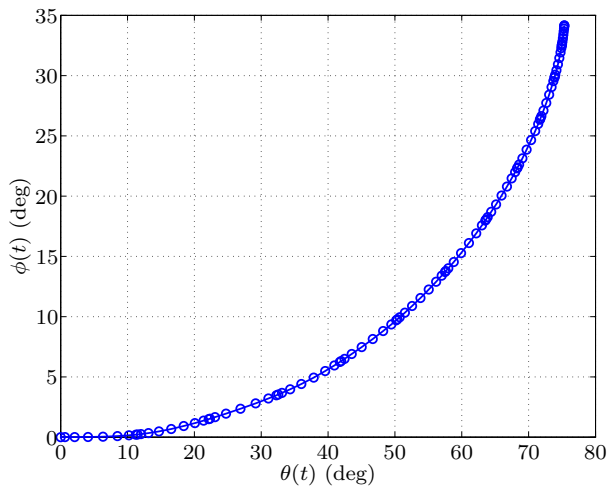
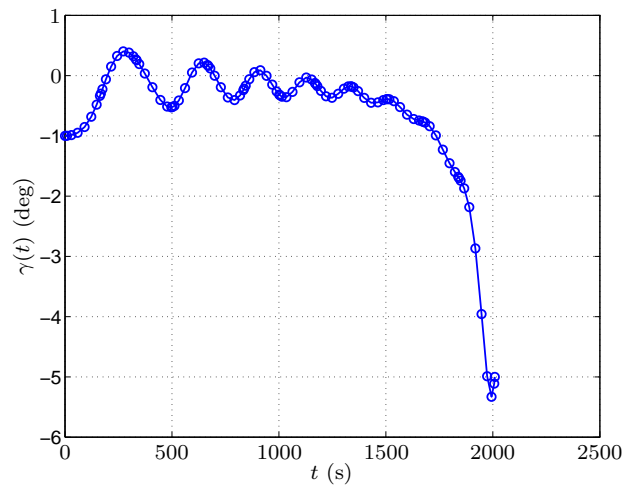
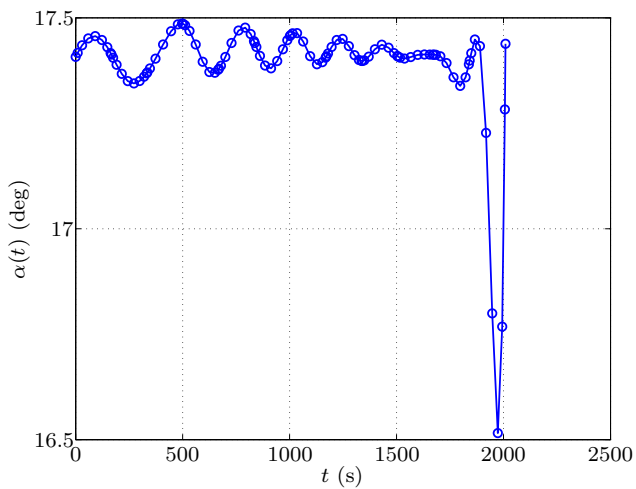
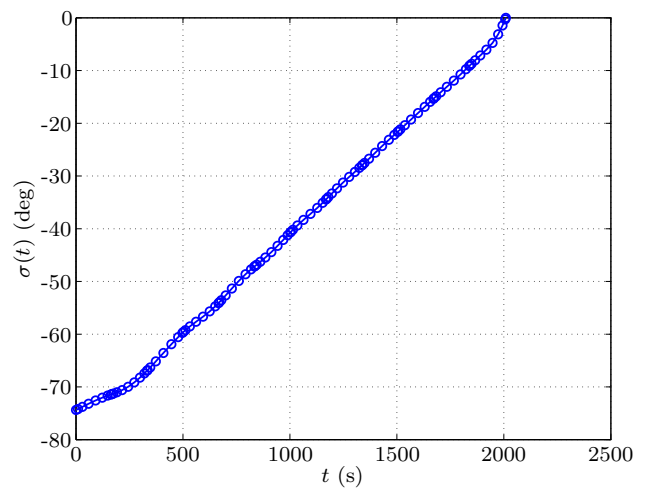
(a) $h(t)$ vs. t .(b) $v(t)$ vs. t .(c) $\phi(t)$ vs. $\theta(t)$.(d) $\gamma(t)$ vs. t .(e) $\alpha(t)$ vs. t .(f) $\sigma(t)$ vs. t .

Figure 7: Solution to Reusable Launch Vehicle Entry Problem Using GPOPS – III with the NLP Solver IPOPT and a Mesh Refinement Tolerance of 10^{-6} .

6.4 Minimum Time-to-Climb of a Supersonic Aircraft

The problem considered in this section is the classical minimum time-to-climb of a supersonic aircraft. The objective is to determine the minimum-time trajectory and control from take-off to a specified altitude and speed. This problem was originally stated in the open literature in the work of Ref. 24, but the model used in this study was taken from Ref. 12 with the exception that a linear extrapolation of the thrust data as found in Ref. 12 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

$$J = t_f \quad (51)$$

subject to the dynamic constraints

$$\dot{h} = v \sin \alpha \quad (52)$$

$$\dot{v} = \frac{T \cos \alpha - D}{m} \quad (53)$$

$$\dot{\gamma} = \frac{T \sin \alpha + L}{mv} + \left(\frac{v}{r} - \frac{\mu}{vr^2} \right) \cos \gamma \quad (54)$$

$$\dot{m} = -\frac{T}{g_0 I_{sp}} \quad (55)$$

and the boundary conditions

$$h(0) = 0 \text{ ft} \quad (56)$$

$$v(0) = 129.3144 \text{ m/s} \quad (57)$$

$$\gamma(0) = 0 \text{ rad} \quad (58)$$

$$h(t_f) = 19994.88 \text{ m} \quad (59)$$

$$v(t_f) = 295.092 \text{ ft/s} \quad (60)$$

$$\gamma(t_f) = 0 \text{ rad} \quad (61)$$

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. 24. 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 brysonMinimumTimeToClimbAeroData.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)
fpa0 = 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;

%-----%
%----- Set Up Problem Using Data Provided Above -----%
%-----%
bounds.phase.initialtime.lower = t0;
bounds.phase.initialtime.upper = t0;
bounds.phase.finaltime.lower   = tfmin;
bounds.phase.finaltime.upper   = tfmax;
bounds.phase.initialstate.lower = [alt0, speed0, fpa0, mass0];
bounds.phase.initialstate.upper = [alt0, speed0, fpa0, mass0];
bounds.phase.state.lower        = [altmin, speedmin, fpamin, massmin];
bounds.phase.state.upper        = [altmax, speedmax, fpamax, massmax];
bounds.phase.state.lower        = [altf, speedf, fpaf, massmin];
bounds.phase.state.upper        = [altf, speedf, fpaf, massmax];
bounds.phase.control.lower      = alphamin;
bounds.phase.control.upper      = alphamax;

%-----%
%----- Provide Guess of Solution -----%
%-----%
guess.phase.time                = [0; 1000];
guess.phase.state(:,1)          = [alt0; altf];
guess.phase.state(:,2)          = [speed0; speedf];
guess.phase.state(:,3)          = [fpa0; fpaf];
guess.phase.state(:,4)          = [mass0; mass0];
guess.phase.control              = [20; -20]*pi/180;

%-----%
%----- Provide Mesh Refinement Method and Initial Mesh -----%
%-----%
mesh.method                     = 'hp-LiuRao-Legendre';
mesh.tolerance                  = 1e-6;
mesh.colpointsmmin             = 4;
mesh.colpointsmmax             = 10;
mesh.sigma                      = 0.75;

% mesh.colpointsmmax = 100;
% mesh.colpoints     = 10;
% mesh.fraction      = 1;

%-----%
%----- Configure Setup Using the information provided -----%
%-----%
setup.name                      = 'Bryson-Minimum-Time-to-Climb-Problem';
setup.functions.continuous      = @brysonMinimumTimeToClimbContinuous;
setup.functions.endpoint        = @brysonMinimumTimeToClimbEndpoint;
setup.displaylevel              = 2;
setup.nlp.solver                 = 'ipopt';
setup.nlp.ipoptoptions.linear_solver = 'ma57';
setup.bounds                    = bounds;
setup.guess                     = guess;
setup.mesh                      = mesh;
setup.auxdata                   = auxdata;
setup.derivatives.supplier      = 'sparseCD';
% setup.derivatives.supplier    = 'adigator';
setup.derivatives.derivativelevel = 'second';
setup.derivatives.dependencies = 'sparseNaN';
setup.scales.method             = 'automatic-bounds';
setup.method                    = 'RPM-Differentiation';

%-----%
%----- Solve Problem Using GPOPS2 -----%
%-----%

```

```

output = gpops2(setup);

%-----%
% Begin Function:  minimumTimeToClimbContinuous.m %
%-----%
function phaseout = brysonMinimumTimeToClimbContinuous(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] = brysonMinimumTimeToClimbCompute(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:  brysonMinimumTimeToClimbEndpoint.m %
%-----%
function output = brysonMinimumTimeToClimbEndpoint(input);

output.objective = input.phase(1).finaltime;;

%-----%
% End Function:  brysonMinimumTimeToClimbEndpoint.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);

ii       = Mach>=0.8;
jj       = Mach<0.8;

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

if any(jj)
    CD(jj) = 0.013;
    CL(jj) = 3.44;
    eta(jj) = 0.54;
end

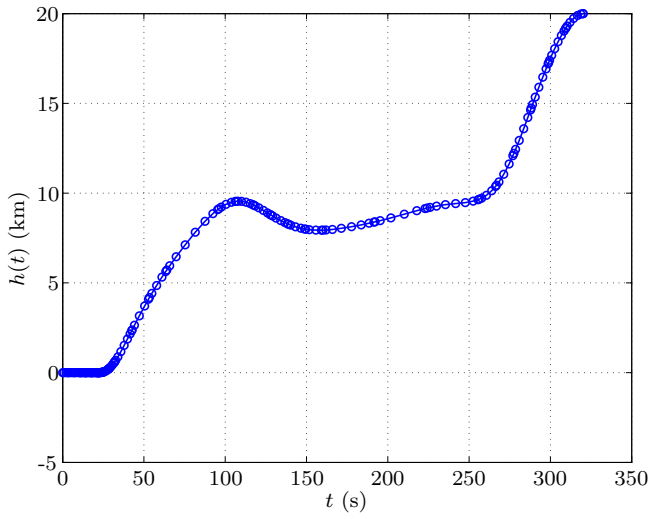
%-----%
% End Function:  minimumTimeToClimbCompute.m  %
%-----%

```

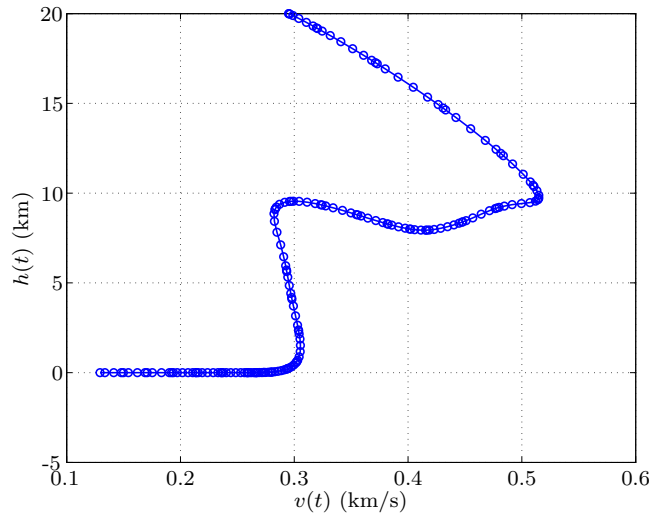
The components of the state and the control obtained from running the above GPOPS – III code is summarized in Figs. 8a–8d.

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

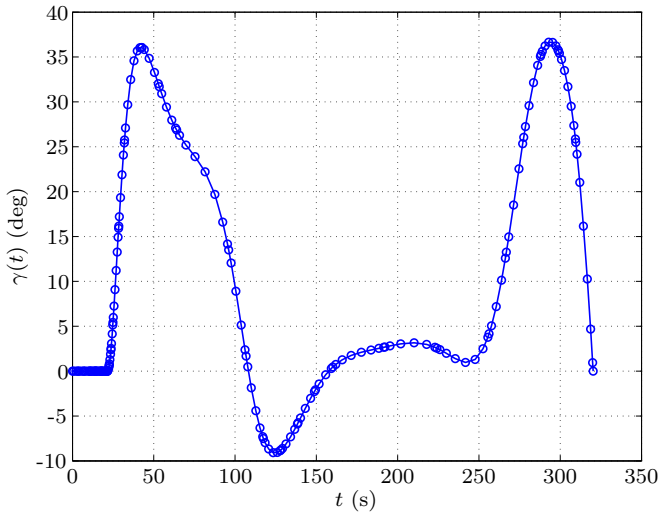
Mesh Refinement Iteration	Relative Error Estimate
1	5.776×10^{-3}
2	2.3717×10^{-3}
3	3.0679×10^{-5}
4	6.2216×10^{-6}
5	8.861×10^{-6}
6	2.3224×10^{-6}
7	1.3708×10^{-6}
8	3.8553×10^{-6}
9	5.1621×10^{-6}
10	7.0515×10^{-6}
11	2.5598×10^{-6}
12	1.0775×10^{-6}
13	7.8122×10^{-7}



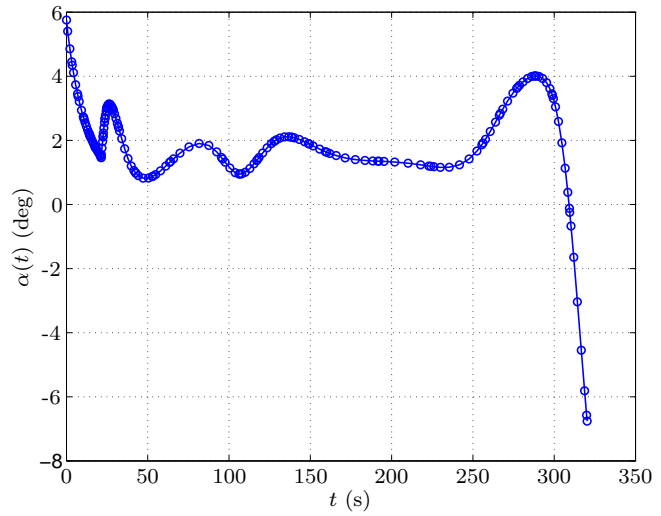
(a) $h(t)$ vs. t .



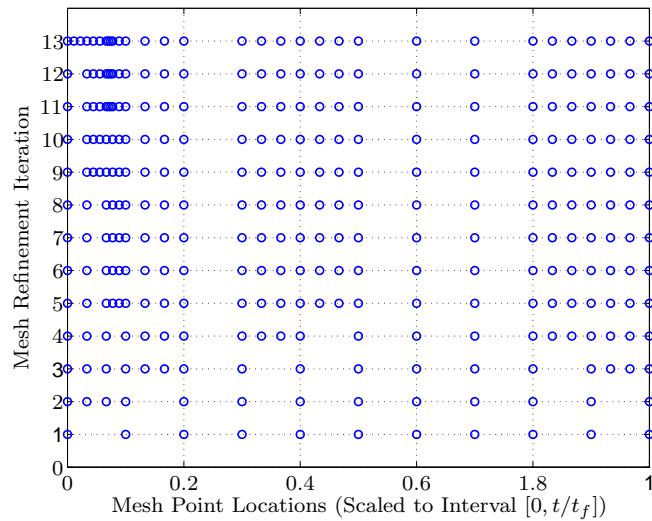
(b) $h(t)$ vs. $v(t)$.



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



(d) α vs. t .



(e) Mesh Refinement History.

Figure 8: Solution to Minimum Time-to-Climb Problem Using GPOPS – III 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 problem was originally described in Ref.¹⁸ and the problem considered here is identical to that of Ref.¹⁸ The objective is to minimize the average wind gradient slope β , that is, minimize

$$J = \beta \quad (62)$$

subject to the hang glider dynamics

$$\begin{aligned} \dot{x} &= v \cos \gamma \sin \psi + W_x & , & \quad m\dot{v} &= -D - mg \sin \gamma - m\dot{W}_x \cos \gamma \sin \psi \\ \dot{y} &= v \cos \gamma \cos \psi & , & \quad mv\dot{\gamma} &= L \cos \sigma - mg \cos \gamma + m\dot{W}_x \sin \gamma \sin \psi, \\ \dot{h} &= v \sin \gamma & , & \quad mv \cos \gamma \dot{\psi} &= L \sin \sigma - m\dot{W}_x \cos \psi \end{aligned} \quad (63)$$

and the boundary conditions

$$\begin{aligned} (x(0), y(0), h(0)) &= (x(t_f), y(t_f), h(t_f)) = (0, 0, 0), \\ (v(t_f) - v(0), \gamma(t_f) - \gamma(0), \psi(t_f) + 2\pi - \psi(0)) &= (0, 0, 0), \end{aligned} \quad (64)$$

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

$$\begin{aligned} D &= qSC_D, \\ L &= qSC_L, \end{aligned} \quad (65)$$

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 are taken verbatim from Ref. 18 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 GPOPS – III 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. 9.

```

%----- 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

%-----%
%----- Provide Auxiliary Data for Problem -----%
%-----%
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;

```

```

%-----%
%----- Boundary Conditions -----%
%-----%
t0 = 0; x0 = 0; y0 = 0; z0 = 0; v0 = 100;

%-----%
%----- Limits on Variables -----%
%-----%
tfmin   = 1;          tfmax   = 30;
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;

%-----%
%----- Set Up Problem Using Data Provided Above -----%
%-----%
bounds.phase.initialtime.lower = t0;
bounds.phase.initialtime.upper = t0;
bounds.phase.finaltime.lower   = tfmin;
bounds.phase.finaltime.upper   = tfmax;
bounds.phase.initialstate.lower = [x0, y0, z0, vmin, gammamin, psimin];
bounds.phase.initialstate.upper = [x0, y0, z0, vmax, gammamax, psimax];
bounds.phase.state.lower        = [xmin, ymin, zmin, vmin, gammamin, psimin];
bounds.phase.state.upper        = [xmax, ymax, zmax, vmax, gammamax, psimax];
bounds.phase.finalstate.lower   = [x0, y0, z0, vmin, gammamin, psimin];
bounds.phase.finalstate.upper   = [x0, y0, z0, vmax, gammamax, psimax];
bounds.phase.control.lower      = [CLmin, Phimin];
bounds.phase.control.upper      = [CLmax, Phimax];
bounds.phase.path.lower        = auxdata.lmin;
bounds.phase.path.upper        = auxdata.lmax;
bounds.eventgroup(1).lower     = [0, 0, -2*pi];
bounds.eventgroup(1).upper     = [0, 0, -2*pi];
bounds.parameter.lower         = betamin;
bounds.parameter.upper         = betamax;

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

%-----%
%----- Provide Mesh Refinement Method and Initial Mesh -----%
%-----%
mesh.maxiterations = 10;
mesh.method        = 'hp-LiuRao';

```

```

mesh.tolerance          = 1e-6;

%-----%
%----- Configure Setup Using the information provided -----%
%-----%
setup.name              = 'Dynamic-Soaring-Problem';
setup.functions.continuous = @dynamicSoaringContinuous;
setup.functions.endpoint = @dynamicSoaringEndpoint;
setup.nlp.solver        = 'ipopt';
setup.nlp.ipoptoptions.linear_solver = 'ma57';
setup.displaylevel      = 2;
setup.auxdata           = auxdata;
setup.bounds            = bounds;
setup.guess             = guess;
setup.mesh              = mesh;
setup.derivatives.supplier = 'adigator';
setup.derivatives.derivativelevel = 'second';
setup.scales.method     = 'automatic-bounds';
setup.method            = 'RPM-Differentiation';

%-----%
%----- Solve Problem Using GPOPS-II -----%
%-----%
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      = l;
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)-x0(6)];
output.objective = 7*beta;

```

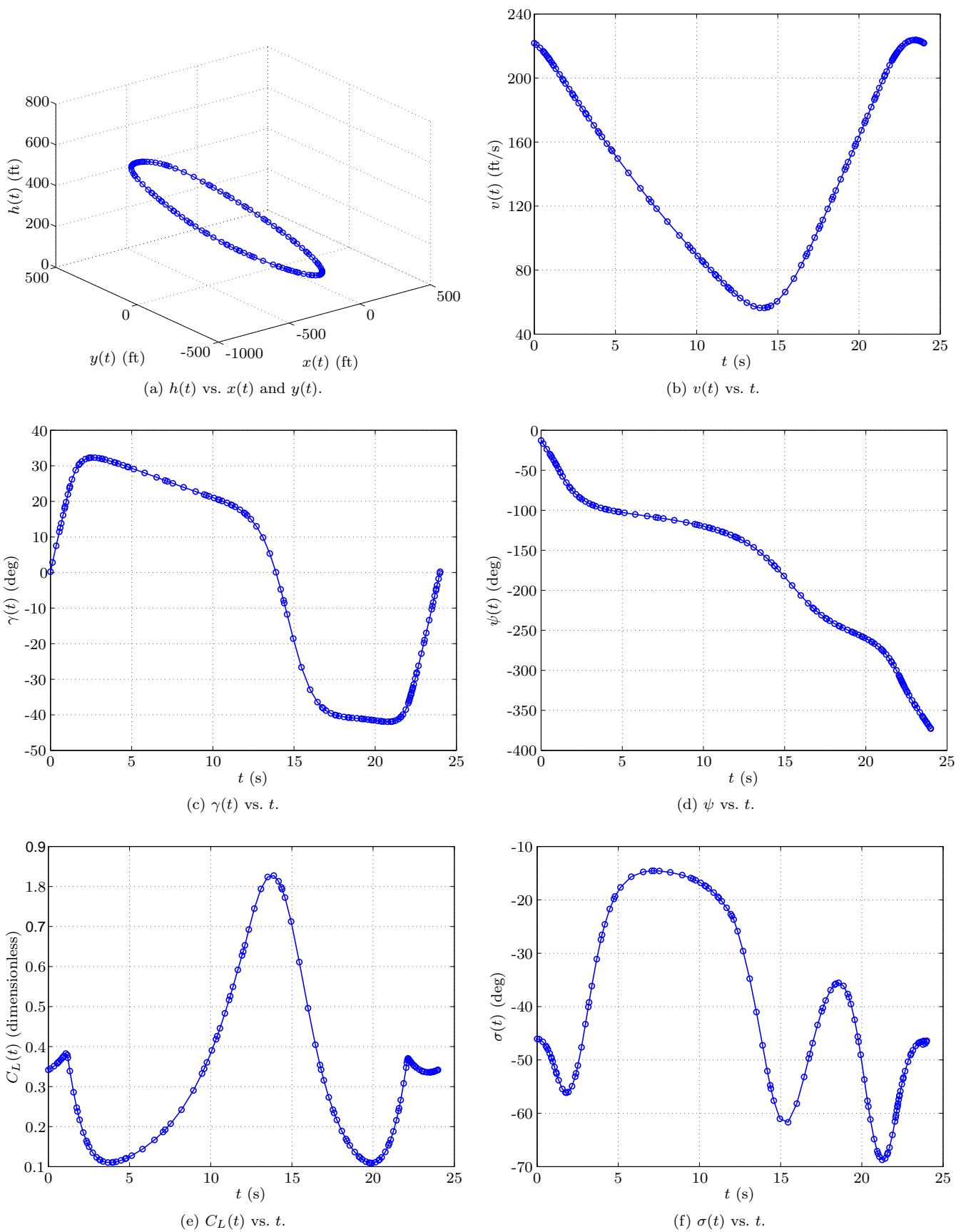


Figure 9: Solution to Dynamic Soaring Problem Using \mathbb{G} POPS – III with the NLP Solver IPOPT and a Mesh Refinement Tolerance of 10^{-6} .

6.5 Two-Strain Tuberculosis Optimal Control Problem

Quoting from Ref. 19, “[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. 19 is formulated as follows. Minimize the objective functional

$$J = \int_0^{t_f} \left[L_2 + I_2 + \frac{1}{2} B_1 u_1^2 + B_2 u_2^2 \right] dt \quad (66)$$

subject to the dynamic constraints

$$\begin{aligned} \dot{S}(t) &= \Lambda - \beta_1 S \frac{I_1(t)}{N} - \beta^* S \frac{I_2(t)}{N} - \mu S(t), \\ \dot{L}_1(t) &= \beta_1 S(t) \frac{I_1(t)}{N} - (\mu + k_1) L_1(t) - u_1 r_1 L_1(t) \\ &\quad + (1 - u_2(t)) p r_2 I_1(t) + \beta_2 T(t) \frac{I_1(t)}{N} - \beta^* L_1(t) \frac{I_2(t)}{N}, \\ \dot{I}_1(t) &= k_1 L_1(t) - (\mu + d_1) I_1(t) - r_2 I_1(t), \\ \dot{L}_2(t) &= (1 - u_2(t)) q r_2 I_1(t) - (\mu + k_2) L_2(t) \\ &\quad + \beta^* (S(t) + L_1(t) + T(t)) \frac{I_2(t)}{N}, \\ \dot{I}_2(t) &= k_2 L_2(t) - (\mu + d_2) I_2(t), \\ \dot{T}(t) &= u_1(t) r_1 L_1(t) - (1 - (1 - u_2(t))) (p + q) r_2 I_1(t) \\ &\quad - \beta_2 T(t) \frac{I_1(t)}{N} - \beta^* T(t) \frac{I_2(t)}{N} - \mu T(t), \\ 0 &= S + L_1 + I_1 + L_2 + I_2 + T - N, \end{aligned} \quad (67)$$

and the initial conditions

$$(S(0), L_1(0), I_1(0), L_2(0), I_2(0), T(0)) = (S_0, L_{10}, I_{10}, L_{20}, I_{20}, T_0), \quad (68)$$

where details of the model can be found in Ref. 19 (and are also provided in the GPOPS – III code shown below). The optimal control problem of Eqs. (66)–(68) is solved using GPOPS – III 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. 10 and 11.

```

%----- 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 -----%
%----- Set up Bounds for Optimal Control Problem -----%
%----- 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;
ulmin = 0.05;
ulmax = 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 = [ulmin, u2min];
bounds.phase.control.upper = [ulmax, u2max];
bounds.phase.initialtime.lower = t0;
bounds.phase.initialtime.upper = t0;
bounds.phase.finaftime.lower = tf;
bounds.phase.finaftime.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 -----%
%----- Provide an Initial Guess of the Solution -----%
%----- 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 -----%
%----- Provide an Initial Mesh for the Solution -----%
%----- Provide an Initial Mesh for the Solution -----%
mesh.method = 'hp-LiuRao-Legendre';
mesh.tolerance = 1e-6;
mesh.maxiterations = 10;
mesh.colpointsmmin = 3;

```

```

mesh.colpointsmax    = 10;
N = 10;
mesh.phase.colpoints = 4*ones(1,N);
mesh.phase.fraction  = ones(1,N)/N;

setup.name           = 'Tuberculosis-Optimal-Control-Problem';
setup.functions.continuous = @tuberculosisContinuous;
setup.functions.endpoint   = @tuberculosisEndpoint;
setup.displaylevel      = 2;
setup.auxdata          = auxdata;
setup.bounds           = bounds;
setup.guess            = guess;
setup.mesh             = mesh;
setup.nlp.solver       = 'ipopt';
setup.derivatives.supplier = 'adigator';
setup.derivatives.derivativelevel = 'second';
setup.scales.method    = 'automatic-bounds';
setup.method           = 'RPM-Differentiation';

%-----%
%----- Solve Problem with GPOPS2 and Extract Solution -----%
%-----%

output = gpops2(setup);

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

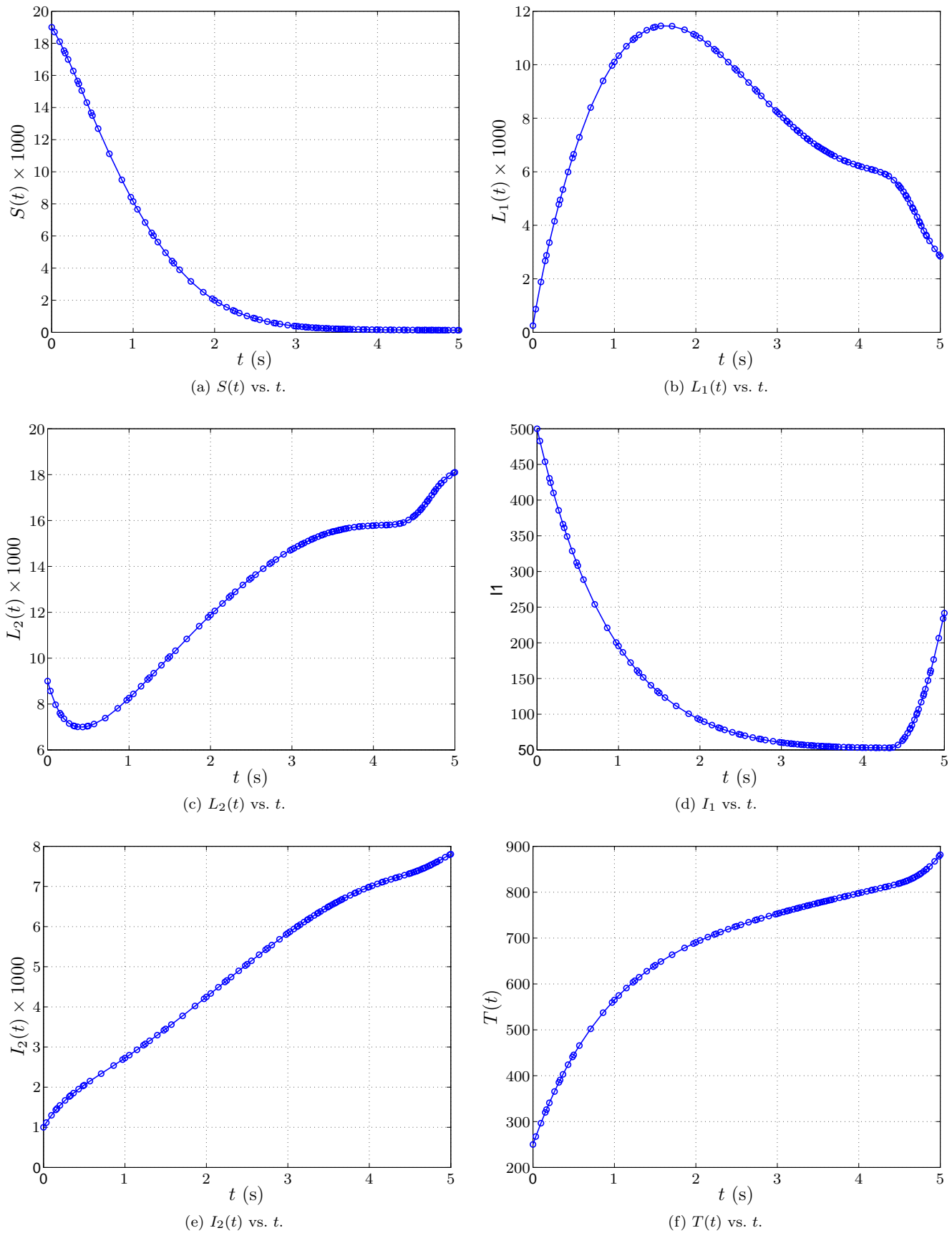


Figure 10: Optimal State for Tuberculosis Optimal Control Problem Using GPOPS – III with the NLP Solver SNOPT and a Mesh Refinement Tolerance of 10^{-6} .

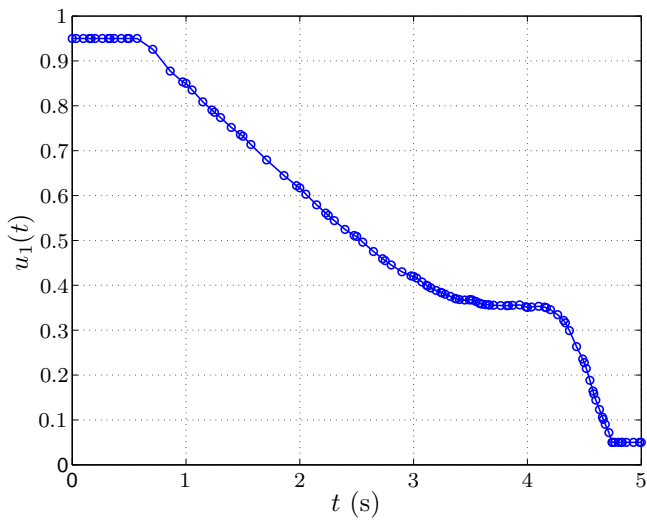
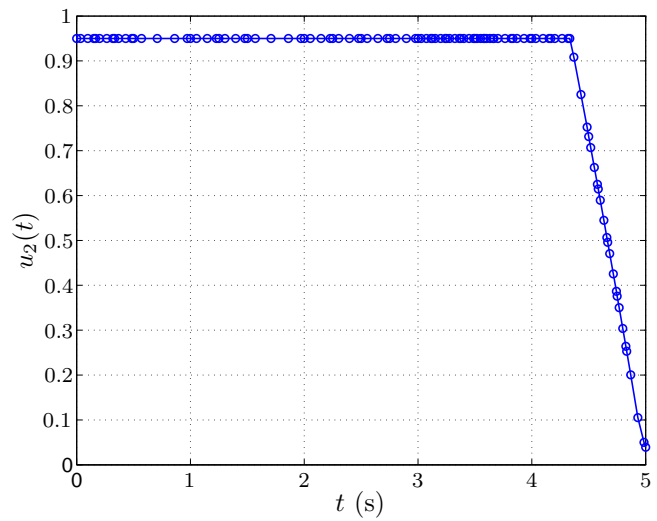
(a) $u_1(t)$ vs. t .(b) $u_2(t)$ vs. t .

Figure 11: Optimal Control for Tuberculosis Optimal Control Problem Using \mathbb{G} POPS – III with the NLP Solver SNOPT and a Mesh Refinement Tolerance of 10^{-6} .

7 Concluding Remarks

While the authors have put for the effort to make GPOPS – III a user-friendly software, it is important to understand several aspects of computational optimal control in order to make GPOPS – III 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] Weinstein, M. J. and Rao, A. V., “A Source Transformation via Operator Overloading Method for the Automatic Differentiation of Mathematical Functions in MATLAB,” *ACM Transactions on Mathematical Software*, in Revision, April 2014. Current manuscript available at <http://vdol.mae.ufl.edu/SubmittedJournalPublications/gatorad-TOMS-November-2013.pdf>.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] Darby, C. L., Hager, W. W., and Rao, A. V., “An *hp*-Adaptive Pseudospectral Method for Solving Optimal Control Problems,” *Optimal Control Applications and Methods*, Vol. 32, No. 4, July–August 2011, pp. 476–502.
- [6] Darby, C. L., Hager, W. W., and Rao, A. V., “Direct Trajectory Optimization Using a Variable Low-Order Adaptive Pseudospectral Method,” *Journal of Spacecraft and Rockets*, Vol. 48, No. 3, May–June 2011, pp. 433–445.
- [7] Francolin, C. C., Hager, W. W., and Rao, A. V., “Costate Approximation in Optimal Control Using Integral Gaussian Quadrature Collocation Methods,” *Optimal Control Applications and Methods*, Published Online in Early View, January 2014. DOI 10.1002/oca2112.
- [8] Liu, F., Hager, W. W., and Rao, A. V., “Adaptive Mesh Refinement for Optimal Control Using Nonsmoothness Detection and Mesh Size Reduction,” *Journal of the Franklin Institute*, Vol. 352, No. 10, October 2015, pp. 4081 – 4106.
- [9] Liu, F., Hager, W. W., and Rao, A. V., “Adaptive Mesh Refinement Method for Optimal Control Using Nonsmoothness Detection and Mesh Size Reduction,” *IEEE Transactions on Control Systems Technology*, October 2015.
- [10] Patterson, M. A., Hager, W. W., and Rao, A. V., “A *ph* Mesh Refinement Method for Optimal Control,” *Optimal Control Applications and Methods*, Vol. 36, No. 4, July–August 2015, pp. 398 – 421.
- [11] 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.

- [12] Betts, J. T., *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*, SIAM Press, Philadelphia, 2009.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] Zhao, Y. J., “Optimal Pattern of Glider Dynamic Soaring,” *Optimal Control Applications and Methods*, Vol. 25, 2004, pp. 67–89.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] Bate, R. R., Mueller, D. D., and White, J. E., *Fundamentals of Astrodynamics*, Dover Publications, 1971.
- [24] Bryson, A. E. and Ho, Y.-C., *Applied Optimal Control*, Hemisphere Publishing, New York, 1975.