CGPOPS Version 1.0:
A General-Purpose C++ Toolbox for Solving Optimal Control Problems Using the Radau Collocation Method

Yunus M. Agamawi
Anil V. Rao

University of Florida
Gainesville, FL 32611-6250
USA

October 2020

Preface

$\mathbb{CGPOPS}$ is a general-purpose software for solving nonlinear optimal control problems that arise in a wide variety of applications including engineering, economics, and medicine. $\mathbb{CGPOPS}$ uses some of the latest advancements in the area of direct orthogonal collocation methods for solving optimal control problems. $\mathbb{CGPOPS}$ employs an *hp*-adaptive Radau Gaussian quadrature method where the collocation is performed at the Legendre-Gauss-Radau quadrature points. $\mathbb{CGPOPS}$ has been designed to work with the nonlinear programming (NLP) solver IPOPT, and C++ source code files for IPOPT are included with the software. $\mathbb{CGPOPS}$ employs hyper-dual derivative approximations, bicomplex-step derivative approximations, or central finite-differencing to estimate all first and second derivatives required by the NLP solver. The software has been designed to be extremely flexible, allowing a user to formulate an optimal control problem in a way that makes sense for the problem being solved. Few, if any, restrictions have been placed on the manner in which a problem needs to be modeled. As stated, the software is *general-purpose*, that is, it has not been developed for any specific type of problem. While the developers of $\mathbb{CGPOPS}$ 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.

Reformulation of Optimal Control Framework from Previous Versions of $\mathbb{GPOPS-II}$

$\mathbb{CGPOPS}$ represents a more computationally efficient and portable C++ implementation of the optimal control framework utilized by the MATLAB software $\mathbb{GPOPS-II}$ first made available in 2013. Specifically, $\mathbb{CGPOPS}$ is organized much differently than $\mathbb{GPOPS-II}$ so as to facilitate computational efficiency of the optimal control framework in a C++ software implementation. Moreover, $\mathbb{CGPOPS}$ is capable of producing m-script output files of the solution which may then be plotted in MATLAB in order to allow for direct comparison between $\mathbb{CGPOPS}$ and $\mathbb{GPOPS-II}$ solutions. Although solution interpretation is not as convenient in C++ as it is in MATLAB, the reduced computational time and increased availability of the software (no subscription required) will make the software extremely useful for solving optimal control problems in a variety of applications. In order to have as smooth a transition as possible to the new software, the authors of $\mathbb{CGPOPS}$ are happy to assist users of $\mathbb{GPOPS-II}$ in rewriting their code for $\mathbb{CGPOPS}$.

Acknowledgments

Disclaimer

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

Licensing Agreement

By downloading, using, modifying, or distributing $\mathbb{CGPOPS}$, you agree to the terms of the license agreement as stated on the $\mathbb{GPOPS-II}$ website http://www.gpops2.com/License/License.html. In other words, the $\mathbb{CGPOPS}$ licensing agreement follows the licensing agreement of $\mathbb{GPOPS-II}$. **Please read the license terms and conditions carefully before proceeding to download, install, or use of $\mathbb{CGPOPS}$.**

Contents

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

The general multiple-phase optimal control problem that can be solved by $\mathbb{CGPOPS}$ is given as follows. First let $p \in \{1, \dots, P\}$ be the phase number where P is the total number of phases. Determine the state $\mathbf{x}^{(p)}(t) \in \mathbb{R}^{n_x^{(p)}}$, the control $\mathbf{u}^{(p)}(t) \in \mathbb{R}^{n_u^{(p)}}$, the integrals $\mathbf{q}^{(p)}(t) \in \mathbb{R}^{n_q^{(p)}}$, the start times $t_0^{(p)} \in \mathbb{R}$, and the terminus times $t_f^{(p)} \in \mathbb{R}$ in all phases p, along with static parameters $\mathbf{s} \in \mathbb{R}^{n_s}$ that minimize the objective functional

$$\mathcal{J} = \phi(\mathcal{E}^{(1)}, \dots, \mathcal{E}^{(P)}, \mathbf{s}) \ , \tag{1}$$

subject to the dynamic constraints

$$\dot{\mathbf{x}}^{(p)} = \mathbf{a}^{(p)}(\mathbf{x}^{(p)}, \mathbf{u}^{(p)}, t^{(p)}, \mathbf{s}) \ , \quad \forall p \in \{1, \dots, P\} \ , \tag{2}$$

the event constraints

$$\mathbf{b}_{\min} \leq \mathbf{b}(\mathcal{E}^{(1)}, \dots, \mathcal{E}^{(P)}, \mathbf{s}) \leq \mathbf{b}_{\max} \ , \tag{3}$$

the inequality path constraints

$$\mathbf{c}_{\min}^{(p)} \leq \mathbf{c}^{(p)}(\mathbf{x}^{(p)}, \mathbf{u}^{(p)}, t^{(p)}, \mathbf{s}) \leq \mathbf{c}_{\max}^{(p)} \ , \quad \forall p \in \{1, \dots, P\} \ , \tag{4}$$

the integral constraints

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

and the static parameter constraints

$$\mathbf{s}_{\min} \leq \mathbf{s} \leq \mathbf{s}_{\max} \ , \tag{6}$$

where

$$\mathcal{E}^{(p)} = \left[\mathbf{x}^{(p)}(t_0^{(p)}), t_0^{(p)}, \mathbf{x}^{(p)}(t_f^{(p)}), t_f^{(p)}, \mathbf{q}^{(p)} \right] \ , \quad \forall p \in \{1, \dots, P\} \ , \tag{7}$$

and the integrals in each phase are defined as

$$q_j^{(p)} = \int_{t_0^{(p)}}^{t_f^{(p)}} g_j^{(p)}(\mathbf{x}^{(p)}, \mathbf{u}^{(p)}, t^{(p)}, \mathbf{s})dt \ , \quad \forall j \in \{1, \dots, n_q^{(p)}\}, \ \forall p \in \{1, \dots, P\} \ , \tag{8}$$

It is important to note that the event constraints of Eq.(3) contain functions which can relate information at the start and/or terminus of any phase (including any relationships involving any integral or static parameters), with phases not needing to be in sequential order to be linked. Moreover, it is noted that the approach to linking phases is based on well-known formulations in the literature such as those given in Ref. 1 and 2. Furthermore, each phase may be further refined into $K^{(p)}$ multiple-intervals such that Eq. (8) becomes

$$q_j^{(p)} = \sum_{k=1}^{K^{(p)}} \int_{(t_0^{(k)})^{(p)}}^{(t_f^{(k)})^{(p)}} g_j^{(p)}(\mathbf{x}^{(p)}, \mathbf{u}^{(p)}, t^{(p)}, \mathbf{s})dt \ , \quad \forall j \in \{1, \dots, n_q^{(p)}\}, \ \forall p \in \{1, \dots, P\}. \tag{9}$$

1.1   Radau Collocation Method Employed by $\mathbb{CGPOPS}$

The method employed by $\mathbb{CGPOPS}$ is an *hp*-adaptive version of the *Legendre-Gauss-Radau* (LGR) collocation method. The LGR collocation method is a direct orthogonal collocation Gaussian quadrature implicit integration method where collocation is performed at the *Legendre-Gauss-Radau* points. The theory behind the LGR collocation method used in $\mathbb{CGPOPS}$ can be found in Refs. 3, 4, 5, 6, and 7.

1.2   Organization of $\mathbb{CGPOPS}$

$\mathbb{CGPOPS}$ is organized as follows. In order to specify the optimal control problem that is to be solved, the user must write the following C++ functions: (1) objective function, (2) dynamics function, (3) path function, (4) Lagrange function, and (5) event function. The objective function defines how the endpoint variables of each phase and any static parameters of the problem contribute to the overall cost functional of the optimal

control problem. The dynamics function defines the evolution of the dynamics in any phase of the problem. The path function defines the path constraints in any phase of the problem. The Lagrange function defines the integrands of any Lagrange cost components of the cost functional or integral constraints of the problem. The event function defines how the endpoint variables of all phases and any static parameters relate to one another. 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 C++ 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. Finally, type declarations (such as double, int, char, etc.) will be denoted by <u>purple underlined</u> characters. Users who do not have color rendering capability will see only *slanted*, **boldface**, and <u>underlined</u> characters, respectively.

2   Constructing an Optimal Control Problem Using ℂ𝔾ℙ𝕆ℙ𝕊

We now proceed to describe the constructs required to specify an optimal control problem in ℂ𝔾ℙ𝕆ℙ𝕊. We note that the key C++ programming elements used in constructing an optimal control problem in ℂ𝔾ℙ𝕆ℙ𝕊 are *structure* and *arrays of structures* defined or created using C++ object-oriented programming. In this Section we provide the details of constructing a problem using ℂ𝔾ℙ𝕆ℙ𝕊.

2.1   Calling ℂ𝔾ℙ𝕆ℙ𝕊

First, the call to ℂ𝔾ℙ𝕆ℙ𝕊 is made by an executable file *run_cgpops* that is produced using a Makefile to compile the necessary object code files of the user-defined .hpp and .cpp files that define the optimal control problem and LGR transcription process. The user-defined .hpp files are

- **cgpops_gov.hpp**,

- **cgpops_main.hpp**,

- **cgpopsAuxDec.hpp**,

- **cgpopsAuxExt.hpp**,

while the .cpp files are

- **cgpops_gov.cpp**,

- **cgpops_main.cpp**,

- **cgpops_wrapper.cpp**.

The header files **cgpopsAuxExt.hpp** and **cgpopsAuxExt.cpp** are used to declare and define any auxiliary data appearing in the optimal control problem. The files **cgpops_gov.hpp** and **cgpops_gov.cpp** are where the governing equations of the optimal control problem are declared and defined, respectively (i.e. objective function, dynamic functions, path constraints, Lagrange functions, event constraints). The files **cgpops_main.hpp** and **cgpops_main.cpp** are where the problem specifications and bounds are provided, as well as the initialization of the mesh used for LGR transcription. The file **cgpops_wrapper.cpp** is where the settings for the mesh refinement process are specified.

Addionally, Section 2.5 describes how the Makefile used to compile the C++ files described in Sections 2.2 through 2.4 must be modified to include the appropriate directory paths and then called in order to produce the executable binary *run_cgpops* that is executed to obtain a solution to the specified optimal control problem using LGR collocation.

2.2 Classes used in cgpops_gov files

The following code from **src/OCPFunctions.hpp** defines the classes which are inherited from and used in the **cgpops_gov.hpp** and **cgpops_gov.cpp** files in order to modularly define the governing equations of a given optimal control problem. Each class ultimately inherits from the Ipopt::ReferencedObject base class such that the Ipopt::SmartPtr class can be used to automatically take care of memory allocation. The public virtual **eval_eq** method of each class in **src/OCPFunctions.hpp** is overloaded to use the double, HyperDual, and Bicomplex types in order to facilitate the usage of the three available derivative supplier methods within ℂ𝔾ℙ𝕆ℙ𝕊 (i.e., central finite differencing, hyper-dual derivative estimation, and bicomplex derivative estimation). The private template **eq_def** method of each class is called by the overloaded **eval_eq** method; this is where the governing equation for each class type is actually defined.

```cpp
// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// OcpFunctions.hpp
// CGPOPS Software
// Declares OCP function classes
//


#ifndef __OCP_FUNCTIONS_HPP__
#define __OCP_FUNCTIONS_HPP__


#include "IpReferenced.hpp"
#include "HyperDual.hpp"
#include "Bicomplex.hpp"


/*---------------------------------EndPointFunction---------------------------------*/

class EndPointFunction : public Ipopt::ReferencedObject
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
```

```
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

class EndPointParameterEq : public EndPointFunction
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

class EventConstraintEq : public EndPointFunction
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

class ObjectiveEq : public EndPointFunction
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

/*--------------------------------PhasePointFunction--------------------------------*/

class PhasePointFunction : public Ipopt::ReferencedObject
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
```

```
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class PhasePointParameterEq : public PhasePointFunction
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class OrdinaryDifferentialEq : public PhasePointFunction
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class PathConstraintEq : public PhasePointFunction
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class IntegrandEq : public PhasePointFunction
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
```

```
                          HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                          Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};


#endif
```

The classes inheriting from the EndPointFunction class are used to define the equations that have to do with the endpoints of phases (i.e., the objective and event constraints). Analogously, the classes inheriting from the PhasePointFunction class are used to define the equations that have to do with interior points of a phase (i.e., the dynamics, path constraints, and integrands). The arguments used by the **eval_eq** and **eq_def** methods of the EndPointFunction and PhasePointFunction class types are described in Tables 1 and 2, respectively.

Table 1: Description of arguments used by EndPointFunction class types.

Argument	Role	Description
lhs	Output	lhs value for endpoint function to be computed
x0	Input	Pointer to array of initial state values in each phase
xf	Input	Pointer to array of final state values in each phase
q	Input	Pointer to array of integral values in each phase
t0	Input	Pointer to array of initial time value in each phase
tf	Input	Pointer to array of final time value in each phase
s	Input	Array of static parameter values in problem
e	Input	Array of endpoint parameter values in problem

Table 2: Description of arguments used by PhasePointFunction class types.

Argument	Role	Description
lhs	Output	lhs value for phase function to be computed
x	Input	Array of state values at point in phase
u	Input	Array of control values at point in phase
t	Input	Value of time at point in phase
s	Input	Array of static parameter values in problem
p	Input	Array of phase point parameter values at point in phase

The **e** endpoint parameters referred to in Table 1 are simply intermediary terms utilized for computing the governing equations at the endpoints of each phase (i.e., the functions defining the objective and event constraints). Similarly, the **p** phase parameters referred to in Table 2 are simply intermediary terms utilized for computing the governing equations within each phase (i.e., the functions defining the dynamics, path constraints, and integrands).

2.2.1   Inheriting from the ObjectiveEq class

The purpose of this class is to compute the objective function of the transcribed NLP. Classes inheriting from ObjectiveEq should return the value of the objective function, where it is noted that the objective function may be dependent on the variables appearing in the endpoint vectors of all phases as well as the static parameters vector and endpoint parameters vector.

Example of defining the objective class of a single phase problem with the following objective:

$$\mathcal{J}(\mathbf{y}(t_f)) = x_1(t_f)^2 + x_2(t_f)^2 \tag{10}$$

```
// Declaration of objective class in cgpops_gov.hpp
class MixedObjective : public ObjectiveEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

// Definition of objective class methods in cgpops_gov.cpp
template <class T> void MixedObjective::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
                                               T** tf, T* s, T* e)
{
    lhs = xf[0][0]*xf[0][0] + xf[0][1]*xf[0][1];
}
void MinParameter::eval_eq(double& lhs, double** x0, double** xf, double** q,
                           double** t0, double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinParameter::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                           HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinParameter::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                           Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
```

2.2.2   Inheriting from the EventConstraintEq class

The purpose of this class is to compute the event constraints of the transcribed NLP. Classes inheriting from EventConstraintEq should return the value of the event constraint functions, where it is noted that the event constraint function may be dependent on the variables appearing in the endpoint vectors of all phases as well as the static parameters vector and endpoint parameters vector.

Example of defining the event constraint class of a two phase problem with the following event constraints:

$$\begin{array}{rcl} b_1 & = & x_1^{(1)}(t_f^{(1)}) - x_1^{(2)}(t_0^{(2)}) \ , \\ b_2 & = & x_2^{(1)}(t_f^{(1)}) - x_2^{(2)}(t_0^{(2)}) \ , \\ b_3 & = & t_f^{(1)} - t_0^{(2)} \ . \end{array} \tag{11}$$

```
// Declaration of event constraint class in cgpops_gov.hpp
class Event1 : public EventConstraintEq
{
```

```
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};
class Event2 : public EventConstraintEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};
class Event3 : public EventConstraintEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

// Definition of event constraint class methods in cgpops_gov.cpp
template <class T> void Event1::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
                                       T** tf, T* s, T* e)
{
    lhs = xf[0][0] - x0[1][0];
}
template <class T> void Event2::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
                                       T** tf, T* s, T* e)
{
    lhs = xf[0][1] - x0[1][1];
}
template <class T> void Event3::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
                                       T** tf, T* s, T* e)
{
    lhs = tf[0][0] + t0[1][0];
```

```
}
void Event1::eval_eq(double& lhs, double** x0, double** xf, double** q,
                            double** t0, double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void Event1::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                            HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void Event1::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                            Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void Event2::eval_eq(double& lhs, double** x0, double** xf, double** q,
                            double** t0, double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void Event2::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                            HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void Event2::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                            Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void Event3::eval_eq(double& lhs, double** x0, double** xf, double** q,
                            double** t0, double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void Event3::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                            HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void Event3::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                            Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
```

2.2.3   Inheriting from the EndPointParameterEq class

The purpose of this class is to compute the endpoint parameters which are essentially intermediary terms used in the objective and/or event constraints. Classes inheriting from EndPointParameterEq should return the value of the endpoint parameter, where it is noted that the endpoint parameter may be dependent on the variables appearing in the endpoint vectors of all phases as well as the static parameters vector and endpoint parameters vector. Note that endpoint parameters are computed in sequential order as appearing in the **eppEqVecG** vector defined in **cgpops_main.cpp** such that the equation used to compute component i of the endpoint parameter vector should only use components of the endpoint parameter vector $< i$.

Example of defining the endpoint parameter class of a two phase problem with the following endpoint parameters:

$$
\begin{array}{llll}
e_1 &=& x_1^{(1)}(t_f^{(1)})x_2^{(2)}(t_f^{(2)}) \,, & \quad e_2 &=& q_1^{(1)}s_1 \,, \\
e_3 &=& e_1 - e_2 \,, & \quad e_4 &=& e_3 x_2^{(2)}(t_0^{(2)})
\end{array}
\tag{12}
$$

```
// Declaration of endpoint parameter class in cgpops_gov.hpp
class InterTerm1 : public EndPointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                        double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                        HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
```

```
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                        Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};
class InterTerm2 : public EndPointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                        double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                        HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                        Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};
class InterTerm3 : public EndPointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                        double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                        HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                        Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};
class InterTerm4 : public EndPointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                        double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                        HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                        Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

// Definition of endpoint parameter class methods in cgpops_gov.cpp
template <class T> void InterTerm1::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
                                           T** tf, T* s, T* e)
{
    lhs = xf[0][0]*xf[1][1];
}
template <class T> void InterTerm2::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
```

```
                                                T** tf, T* s, T* e)
{
    lhs = q[0][0]*s[0];
}
template <class T> void InterTerm3::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
                                        T** tf, T* s, T* e)
{
    lhs = e[0] - e[1];
}
template <class T> void InterTerm4::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
                                        T** tf, T* s, T* e)
{
    lhs = e[2]*x0[1][1];
}
void InterTerm1::eval_eq(double& lhs, double** x0, double** xf, double** q,
                              double** t0, double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void InterTerm1::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                              HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void InterTerm1::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                              Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void InterTerm2::eval_eq(double& lhs, double** x0, double** xf, double** q,
                              double** t0, double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void InterTerm2::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                              HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void InterTerm2::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                              Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void InterTerm3::eval_eq(double& lhs, double** x0, double** xf, double** q,
                              double** t0, double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void InterTerm3::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                              HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void InterTerm3::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                              Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void InterTerm4::eval_eq(double& lhs, double** x0, double** xf, double** q,
                              double** t0, double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void InterTerm4::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                              HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void InterTerm4::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                              Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
```

Example of defining the objective class of a single phase problem with the following objective:

$$\mathcal{J}(\mathbf{y}(t_f)) = x_1(t_f)^2 + x_2(t_f)^2 \tag{13}$$

```
// Declaration of objective class in cgpops_gov.hpp
class MixedObjective : public ObjectiveEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

// Definition of objective class methods in cgpops_gov.cpp
template <class T> void MixedObjective::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
                                               T** tf, T* s, T* e)
{
    lhs = xf[0][0]*xf[0][0] + xf[0][1]*xf[0][1];
}
void MinParameter::eval_eq(double& lhs, double** x0, double** xf, double** q,
                           double** t0, double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinParameter::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                           HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinParameter::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                           Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
```

2.2.4   Inheriting from the OrdinaryDifferentialEq class

The purpose of this class is to compute the dynamics functions of the transcribed NLP. Classes inheriting from OrdinaryDifferentialEq should return the value of the dynamics functions, where it is noted that the dynamics functions of each phase are treated as if independent of the variables of the other phases.

Example of defining the dynamics class of a single phase problem with the following objective:

$$\begin{array}{rcl} a_1(\mathbf{x}, \mathbf{u}, t, \mathbf{s}) &=& x_2 \ , \\ a_2(\mathbf{x}, \mathbf{u}, t, \mathbf{s}) &=& u_1 \ . \end{array} \tag{14}$$

```
// Declaration of dynamics class in cgpops_gov.hpp
class Dyn1 : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
```

```
                              double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};
class Dyn2 : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

// Definition of dynamics class methods in cgpops_gov.cpp
template <class T> void Dyn1::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = x[1];
}
template <class T> void Dyn2::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = u[0];
}
void Dyn1::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void Dyn1::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                   HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Dyn1::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                   Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void Dyn2::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void Dyn2::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                   HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Dyn2::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                   Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
```

2.2.5   Inheriting from the **PathConstraintEq** class

The purpose of this class is to compute the path constraints functions of the transcribed NLP. Classes inheriting from PathConstraintEq should return the value of the path constraints functions, where it is noted that the path constraints functions of each phase are treated as if independent of the variables of the

other phases.

Example of defining the path constraints class of a three phase problem with the following path constraints:

$$
\begin{array}{rcl}
c_1^{(1)}(\mathbf{x}^{(1)}, \mathbf{u}^{(1)}, t^{(1)}, \mathbf{s}) & = & u_1^{(1)} + u_2^{(1)} \ , \\
c_1^{(3)}(\mathbf{x}^{(3)}, \mathbf{u}^{(3)}, t^{(3)}, \mathbf{s}) & = & u_1^{(3)} - u_2^{(3)} \ .
\end{array}
\tag{15}
$$

```cpp
// Declaration of path constraints class in cgpops_gov.hpp
class Path1 : public PathConstraintEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};
class Path2 : public PathConstraintEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

// Definition of path constraints class methods in cgpops_gov.cpp
template <class T> void Path1::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = u[0] + u[1];
}
template <class T> void Path2::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = u[0] - u[1];
}
void Path1::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void Path1::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                    HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Path1::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                    Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void Path2::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void Path2::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
```

```
                       HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Path2::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                       Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
```

2.2.6 Inheriting from the IntegrandEq class

The purpose of this class is to compute the integrands of the transcribed NLP. Classes inheriting from IntegrandEq should return the value of the integrands, where it is noted that the integrands of each phase are treated as if independent of the variables of the other phases.

Example of defining the integrands class of a two phase problem with the following integrands:

$$
\begin{array}{rcl}
g_1^{(1)}(\mathbf{x}^{(1)}, \mathbf{u}^{(1)}, t^{(1)}, \mathbf{s}) & = & x_1^{(1)} + u_1^{(1)} , \\
g_1^{(2)}(\mathbf{x}^{(2)}, \mathbf{u}^{(2)}, t^{(2)}, \mathbf{s}) & = & x_1^{(2)} - u_1^{(2)} .
\end{array}
\tag{16}
$$

```
// Declaration of integrands class in cgpops_gov.hpp
class Integrand1 : public IntegrandEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};
class Integrand2 : public IntegrandEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

// Definition of integrands class methods in cgpops_gov.cpp
template <class T> void Integrand1::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = x[0] + u[0];
}
template <class T> void Integrand2::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = x[0] - u[0];
}
```

```
void Integrand1::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p)
{eq_def(lhs,x,u,t,s,p);}
void Integrand1::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Integrand1::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void Integrand2::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p)
{eq_def(lhs,x,u,t,s,p);}
void Integrand2::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Integrand2::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
```

2.2.7   Inheriting from the PhasePointParameterEq class

The purpose of this class is to compute the phase parameters which are essentially intermediary terms used in the dynamics, path constraints, and/or integrands. Classes inheriting from PhasePointParameterEq should return the value of the phase parameter, where it is noted that the integrands of each phase are treated as if independent of the variables of the other phases. Note that phase parameters are computed in sequential order as appearing in the **pppEqVecG** vector defined in **cgpops_main.cpp** such that the equation used to compute component i of the phase parameter vector should only use components of the phase parameter vector $< i$.

Example of defining the phase point parameter class of a single phase problem with the following phase parameters:

$$
\begin{array}{rcl}
p_1(\mathbf{x}, \mathbf{u}, t, \mathbf{s}, \mathbf{p}) & = & x_1 + u_2 + s_3 \ , \\
p_2(\mathbf{x}, \mathbf{u}, t, \mathbf{s}, \mathbf{p}) & = & x_2 u_1 \ , \\
p_3(\mathbf{x}, \mathbf{u}, t, \mathbf{s}, \mathbf{p}) & = & p_1 p_2 \ .
\end{array}
\tag{17}
$$

```
// Declaration of integrands class in cgpops_gov.hpp
class InterTerm1 : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};
class InterTerm2 : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
```

```
                            double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};
class InterTerm3 : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};


// Definition of integrands class methods in cgpops_gov.cpp
template <class T> void InterTerm1::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = x[0] + u[1] + s[2];
}
template <class T> void InterTerm2::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = x[1]*u[0];
}
template <class T> void InterTerm3::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = p[0]*p[1];
}
void InterTerm1::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                                      double* p)
{eq_def(lhs,x,u,t,s,p);}
void InterTerm1::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                                      HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void InterTerm1::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                                      Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void InterTerm2::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                                      double* p)
{eq_def(lhs,x,u,t,s,p);}
void InterTerm2::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                                      HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void InterTerm2::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                                      Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
```

```
void InterTerm3::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                                      double* p)
{eq_def(lhs,x,u,t,s,p);}
void InterTerm3::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                                      HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void InterTerm3::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                                      Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
```

2.2.8   Defining setGlobalTabularData Function

The purpose of this function is to define any tabular data defined by class structures that are used within the governing functions of the transcribed NLP. The function prototype is as follows:

```
void setGlobalTabularData(void)
```

The function **setGlobalTabularData** should set the values of any global variable class structures used for storing auxiliary tabular data declared in the **cgpopsAuxDec.hpp** and **cgpopsAuxExt.hpp** files described in Section 2.2.9.

Example of setting global tabular data for global doubleMat class structure *tabularDataG*, where the tabular data is

$$\begin{bmatrix} 2/7 \ , & 15 \ , & e^5 \ , & 10^{-4} \ , & \pi \end{bmatrix} \tag{18}$$

```
void setGlobalTabularData(void)
{
    tabularDataG.val[0] = 2.0/7.0;
    tabularDataG.val[1] = 15;
    tabularDataG.val[2] = exp(5);
    tabularDataG.val[3] = 1e-4;
    tabularDataG.val[4] = M_PI;
}
```

2.2.9   Defining cgpopsAux files

The header files **cgpopsAuxDec.hpp** and **cgpopsAuxExt.hpp** are used to make the initial and external declarations, respectively, of user-defined global variables. The authors of $\mathbb{CGPOPS}$ encourage users to define global variables for auxiliary constants or tabular data of the optimal control problem, where the values of class structures may be set using the **setGlobalTabularData** function discussed in the previous Section 2.3.3.

Example of initial declaration, external declaration, and setting of global variables *gG*, *ReG*, and *CLG* corresponding to the parameters

$$\begin{array}{rcl} g & = & 9.81 \\ R_e & = & 6378166 \\ C_L & = & \{1.2, 0.5, 3.4\} \end{array} \tag{19}$$

respectively, as well as the doubleMat class structure *tabularDataG* set in the previous example of Section 2.3.3.

```
#ifndef __CGPOPS_AUX_DEC_HPP__
#define __CGPOPS_AUX_DEC_HPP__
```

```
// Auxiliary data
double gG = 9.81;
double ReG = 6378166;
double CLG[] = {1.2,0.5,3.4};
doubleMat tabularDataG(5);

#endif

.
.
.

#ifndef __CGPOPS_AUX_EXT_HPP__
#define __CGPOPS_AUX_EXT_HPP__

// Auxiliary data
extern double gG;
extern double ReG;
extern double CLG[];
extern doubleMat tabularDataG;

#endif
```

2.3 Required Functions in cgpops_main files

Table 3 lists and describes the parameters used by the functions called in the **cgpops_main.cpp** to provide the optimal control problem specifications. The <u>doubleMat</u> structure is defined by the class declaration as follows.

```
class doubleMat
{
public:
    doubleMat();                                    // Default Constructor
    doubleMat(int N);                               // Parameterized Constructor
    doubleMat(const doubleMat& sourceObj);          // Copy Constructor
    doubleMat operator=(const doubleMat& sourceObj);   // Assignment Operator
    ~doubleMat();                                   // Default Destructor
    double *val;
    int Len;
};
```

Table 3: Description of parameters used by called functions in **cgpops_main.cpp**.

Parameter	Type	Role	Description
cgpopsResults	doubleMat&	Input/ Output	Address of doubleMat structure for returning results output
PG	double	Global	Number of phases
nsG	double	Global	Number of static parameters
nbG	double	Global	Number of event constraints
nepG	double	Global	Number of endpoint parameters
nxG	double*	Global	Array containing number of state components
nuG	double*	Global	Array containing number of control components
nqG	double*	Global	Array containing number of integral components
ncG	double*	Global	Array containing number of path constraints
nppG	double*	Global	Array containing number of phase parameters
numintervalG	int	Global	Default number of mesh intervals used per phase
initcolptsG	int*	Global	Default number of collocation points per interval
phasenum	int	Input	Phase number of mesh being initialized (C indexing)
numinterval	int	Input	Number of mesh intervals used in a phase
fraction	double*	Input	Array containing fraction values of a scaled interval $[0, 1]$
colpoints	int*	Input	Array containing a number of collocation points
x0_l	double*	Input	Array containing lower bounds of initial state
x0_u	double*	Input	Array containing upper bounds of initial state
xf_l	double*	Input	Array containing lower bounds of final state
xf_u	double*	Input	Array containing upper bounds of final state
x_l	double*	Input	Array containing lower bounds of state
x_u	double*	Input	Array containing upper bounds of state
u_l	double*	Input	Array containing lower bounds of control
u_u	double*	Input	Array containing upper bounds of control
q_l	double*	Input	Array containing lower bounds of integral constraints
q_u	double*	Input	Array containing upper bounds of integral constraints
c_l	double*	Input	Array containing lower bounds of path constraints
c_u	double*	Input	Array containing upper bounds of path constraints
t0_l	double	Input	Lower bound of initial time
t0_u	double	Input	Upper bound of initial time
tf_l	double	Input	Lower bound of final time
tf_u	double	Input	Upper bound of final time
s_l	double*	Input	Array containing lower bounds of static parameters
s_u	double*	Input	Array containing upper bounds of static parameters
b_l	double*	Input	Array containing lower bounds of event constraints
b_u	double*	Input	Array containing upper bounds of event constraints
x0_g	double*	Input	Array containing initial guess of initial state
xf_g	double*	Input	Array containing initial guess of final state
u0_g	double*	Input	Array containing initial guess of initial control
uf_g	double*	Input	Array containing initial guess of final control
q_g	double*	Input	Array containing initial guess of integral vector
t0_g	double	Input	Initial guess of initial time
tf_g	double	Input	Initial guess of final time
s_g	double*	Input	Array containing initial guess of static parameters

Table 4: Description of additional parameters used by called functions in **cgpops_main.cpp**.

Parameter	Type	Role	Description
ContinuityEnforceG	intMat&	Global	Address of intMat structure indicating if continuity constraints are automatically enforced
ContinuityLBG	doubleMatMat&	Global	Address of doubleMatMat structure holding lower bounds of continuity constraints
ContinuityUBG	doubleMatMat&	Global	Address of doubleMatMat structure holding upper bounds of continuity constraints

2.3.1   Calling initGlobalVars Function

The purpose of this function is to initialize the global variables of ℂ𝔾ℙ𝕆ℙ𝕊 that are dependent on the problem specifications. The function prototype is as follows:

```
void initGlobalVars(void)
```

The function **initGlobalVars** should be called after first setting the number of phases, static parameters, event constraints, and endpoint parameters in the problem via the global variables **PG**, **nsG**, **nbG**, and **nepG**, respectively.

Example of initializing the global variables of a three phase problem with three static parameters, ten event constraints, and four endpoint parameters.

```
void cgpops_go(doubleMat& cgpopsResults)
{
    // Define Global Variables used to determine problem size
    PG      = 1;    // Number of phases in problem
    nsG     = 0;    // Number of static parameters in problem
    nbG     = 0;    // Number of event constraints in problem
    nepG    = 0;    // Number of endpoint parameters in problem

    // Allocate memory for problem specifications
    initGlobalVars();
.
.
.
```

2.3.2   Setting ContinuityEnforceG variable

The purpose of this variable is to enable automatic enforcement of continuity constraints for the state and time between phases. The variable prototype is as follows:

```
intMat ContinuityEnforceG(PG-1);
```

where it is assumed that the number of state components is the same in all phases. The variable **ContinuityEnforceG** should be set after calling **initGlobalVars** and before calling **setInfoNLPG** as described in Sections 2.3.1 and 2.3.5, respectively. It is noted that the default lower and upper bounds of the continuity constraints is zero when enforced, but may be customized using the **ContinuityLBG** and **ContinuityUBG** global variables, as described in Sections 2.3.6.

Example of setting the global variable **ContinuityEnforceG** for a three phase problem where continuity is required between all three phases.

```
void cgpops_go(doubleMat& cgpopsResults)
{

⋮

    // Allocate memory for each phase
    initGlobalVars();

⋮

    // Set automatic continuity enforcement between phases
    ContinuityEnforceG.val[0] = 1; // Between phases 1 and 2
    ContinuityEnforceG.val[1] = 1; // Between phases 2 and 3

⋮

    // Set information for transcribed NLP resulting from LGR collocation
    setInfoNLPG();

⋮
```

2.3.3 Calling setGlobalTabularData Function

The purpose of this function is to set the tabular data of any tables used by the governing functions of the **cgpops_gov** files. The function prototype is as follows:

```
void setGlobalTabularData(void)
```

The function **setGlobalTabularData** should be called after first setting the number of state components, control components, integral components, path constraints, and phase parameters used in each phase via the global variables **nxG**, **nuG**, **nqG**, **ncG**, and **nppG** (must call **initGlobalVars** prior to doing this). The authors encourage the users to define global variables whenever using tabular data, as this can significantly reduce the computation time required. It is noted that if there is no global tabular data, the phase specifications must still be defined, but the call to **setGlobalTabularData** may not be necessary (but shouldn't hurt).

Example of initializing the global variables of a two phase problem with four state components, two control components, one integral component, three path constraints, and five phase parameters in each phase.

```
void cgpops_go(doubleMat& cgpopsResults)
{

⋮

    initGlobalVars();

    // Define number of components for each parameter in problem phases
    // Phase 1 specifications
    nxG[0]  = 4;    // Number of state components in phase 1
    nuG[0]  = 2;    // Number of control components in phase 1
    nqG[0]  = 1;    // Number of integral constraints in phase 1
    ncG[0]  = 3;    // Number of path constraints in phase 1
    nppG[0] = 5;    // Number of phase parameters in phase 1
```

```
    // Phase 2 specifications
    nxG[1]  = 4;     // Number of state components in phase 2
    nuG[1]  = 2;     // Number of control components in phase 2
    nqG[1]  = 1;     // Number of integral constraints in phase 2
    ncG[1]  = 3;     // Number of path constraints in phase 2
    nppG[1] = 5;     // Number of phase parameters in phase 2

    setGlobalTabularData(); // Set any global tabular data used in problem
 .
 .
 .
```

2.3.4   Calling setRPMDG Function

The purpose of this function is to initialize the global RPMD class structures that define the LGR transcription process used in each phase. The function prototype is as follows:

`void setRPMDG(int p, int K, double* fraction, int* colpoints)`

The function **setRPMDG** should be called after defining each of the phase specifications as described in Section 2.3.3. If function **setRPMDG** is not called by the user for each phase, a default mesh of **numintervalsG** intervals with **initcolptsG** collocation points in each interval will be initialized for each phase for the LGR transcription.

Example of initializing the global RPMD class structure variables of a two phase problem for a mesh consisting of ten equally spaced intervals with four collocation points in each interval for both phases.

```
void cgpops_go(doubleMat& cgpopsResults)
{
 .
 .
 .

    setGlobalTabularData(); // Set any global tabular data used in problem

    // Define mesh grids for each phase in problem for LGR collocation
    // Phase 1 mesh grid
    int K = 10;     // Number of intervals used in phase 1 mesh
    int Nk = 5;      // Number of collocation points in each interval
    double fraction[K];      // Allocate memory for fraction vector
    int colpoints[K];        // Allocate memory for colpoints vector
    for (int k=0; k<K; k++)
    {
        fraction[k] = 1.0/((double) K);
        colpoints[k] = Nk;
    }
    for (int p=0; p<PG; p++)
    {
        setRPMDG(p,K,fraction,colpoints);
    }
 .
 .
 .
```

2.3.5   Calling setInfoNLPG Function

The purpose of this function is to initialize the global infoNLP class structure that defines the form of the transcribed NLP resulting from the LGR transcription. The function prototype is as follows:

```
void setInfoNLPG(void);
```

The function **setInfoNLPG** should be called after calling **setRPMDG** for each phase in the problem. If **setRPMDG** is not called prior to **setInfoNLPG**, a default mesh will be initialized for each phase, as described in Section 2.3.4.

Example of initializing the global [infoNLP](#) class structure variable.

```
void cgpops_go(doubleMat& cgpopsResults)
{
    .
    .
    .

    for (int p=0; p<PG; p++)
    {
        setRPMDG(p,K,fraction,colpoints);
    }
    .
    .
    .

    // Set information for transcribed NLP resulting from LGR transcription
    setInfoNLPG();
    .
    .
    .
```

2.3.6   Setting ContinuityLBG and ContinuityUBG variables

The purpose of these variables is to enable custom lower and upper bounds for the automatically enforced continuity constraints of the state and time between phases, as described in Section 2.3.2. The variable prototypes are as follows:

```
doubleMatMat ContinuityLBG(PG-1);
doubleMatMat ContinuityUBG(PG-1);
for (int i=0;i<PG-1;i++)
{
ContinuityLBG.mat[i] = getDoubleMat(nxG[0]+1);
ContinuityUBG.mat[i] = getDoubleMat(nxG[0]+1);
}
```

where it is assumed that the number of state components is the same in all phases. The variables **ContinuityLBG** and **ContinuityUBG** should be set after calling **setInfoNLPG** and before calling **setNLPWBG** as described in Sections 2.3.5 and 2.3.8, respectively. It is noted that the default lower and upper bounds of the continuity constraints are zero when enforced.

Example of setting the global variables **ContinuityLBG** and **ContinuityUBG** for a three phase problem where custom bounds for the continuity constraints of 25 and 100 are enforced, respectively, for the first and second state components between the second and third phases.

```
void cgpops_go(doubleMat& cgpopsResults)
{
    .
    .
    .

    // Set information for transcribed NLP resulting from LGR collocation
    setInfoNLPG();
```

```
    ⋮

    // Set custom continuity bounds for state components 1 and 2 between phases 2 and 3
    ContinuityLBG.mat[1].val[0] = 25;
    ContinuityLBG.mat[1].val[1] = 100;
    ContinuityUBG.mat[1].val[0] = 25;
    ContinuityUBG.mat[1].val[1] = 100;

    ⋮

    // Set parameterized constructor for NLP Whole Bounds Class
    setNLPWBG(sl,su,bl,bu);

    ⋮
```

2.3.7 Calling setNLPPBG Function

The purpose of this function is to initialize the global NLPPB class structures that define the variable and constraints bounds for each phase. The function prototype is as follows:

```
void setNLPPBG(int p, double* x0_l, double* x0_u, double* xf_l, double* xf_u,
                      double* x_l, double* x_u, double* u_l, double* u_u,
                      double* q_l, double* q_u,double* c_l, double* c_u,
                      double t0_l , double t0_u, double tf_l,double tf_u)
```

The function **setNLPPBG** should be called for each phase of the problem and is necessary for providing the bounds within each phase for the transcribed NLP.

Example of initializing the global NLPPB class structures of a single phase problem that has two state components, one control component, one integral component, and two path constraints with the following bounds:

$$
\begin{array}{rclcrrcccl}
1 & \leq & x_1(t_0) & \leq & 1 \,, & \quad 0 & \leq & x_1(t_f) & \leq & 1 \,, \\
0 & \leq & x_2(t_0) & \leq & 1 \,, & \quad 0 & \leq & x_2(t_f) & \leq & 0 \,, \\
0 & \leq & x_1 & \leq & 1 \,, & \quad 0 & \leq & x_2 & \leq & 1 \,, \\
-1 & \leq & u_1 & \leq & 1 \,, & \quad -10 & \leq & q_1 & \leq & 10 \,, \\
1 & \leq & c_1 & \leq & 1 \,, & \quad 0 & \leq & c_2 & \leq & 0 \,, \\
0 & \leq & t_0 & \leq & 0 \,, & \quad 0 & \leq & t_f & \leq & 100 \,.
\end{array}
\tag{20}
$$

```
void cgpops_go(doubleMat& cgpopsResults)
{

    ⋮

    // Define Bounds of Optimal Control Problem
    // Phase bounds
    int phasenum = 0;
    double x0_l[2],   x0_u[2];
    double xf_l[2],   xf_u[2];
    double x_l[2],    x_u[2];
    double u_l[1],    u_u[1];
    double q_l[1],    q_u[1];
    double c_l[2],    c_u[2];
    double t0_l,    t0_u;
    double tf_l,    tf_u;
```

```
    x0_l[0] = 1;
    x0_l[1] = 0;
    x0_u[0] = 1;
    x0_u[1] = 1;
    xf_l[0] = 0;
    xf_l[1] = 0;
    xf_u[0] = 1;
    xf_u[1] = 0;
    x_l[0]  = 0;
    x_l[1]  = 0;
    x_u[0]  = 1;
    x_u[1]  = 1;
    u_l[0]  = -1;
    u_u[0]  = 1;
    q_l[0]  = -10;
    q_u[0]  = 10;
    c_l[0]  = 1;
    c_l[1]  = 0;
    c_u[0]  = 1;
    c_u[1]  = 0;
    t0_l    = 0;
    t0_u    = 0;
    tf_l    = 0;
    tf_u    = 100;
    // Set parameterized constructor for NLP Phase Bounds Class
    setNLPPBG(phasenum,x0_l,x0_u,xf_l,xf_u,x_l,x_u,u_l,u_u, \
              q_l,q_u,c_l,c_u,t0_l,t0_u,tf_l,tf_u);

⋮
```

2.3.8   Calling setNLPWBG Function

The purpose of this function is to initialize the global NLPWB class structure that defines the bounds of the static parameters and event constraints of the problem. The function prototype is as follows:

```
void setNLPWBG(double* s_l, double* s_u, double* b_l, double* b_u)
```

The function **setNLPWBG** is necessary for providing the bounds for any static parameters or event constraints appearing in the transcribed NLP.

Example of initializing the global NLPWB class structure of a two phase problem that has two static parameters, and three events constraints with the following bounds:

$$
\begin{array}{ccccc}
-10 & \leq & s_1 & \leq & 10 \ , \\
0 & \leq & s_2 & \leq & 1 \ , \\
0 & \leq & b_1 & \leq & \pi \ , \\
0 & \leq & b_2 & \leq & 0 \ , \\
0 & \leq & b_3 & \leq & 10 \ .
\end{array}
\tag{21}
$$

```
void cgpops_go(doubleMat& cgpopsResults)
{

⋮
```

```
    // Whole problem bounds
    double s_l[2], s_u[2];
    double b_l[3], b_u[3];
    s_l[0] = -10;
    s_l[1] = 0;
    s_u[0] = 10;
    s_u[1] = 1;
    b_l[0] = 0;
    b_l[1] = 0;
    b_l[2] = 0;
    b_u[0] = M_PI;
    b_u[1] = 0;
    b_u[2] = 10;
    // Set parameterized constructor for NLP Whole Bounds Class
    setNLPWBG(s_l,s_u,b_l,b_u);
.
.
.
```

2.3.9   Calling setNLPPGG Function

The purpose of this function is to initialize the global NLPPG class structures that define the initial guess of the variables used for each phase. The function prototype is as follows:

```
void setNLPPGG(int phasenum, double* x0_g, double* xf_g, double* u0_g, double* uf_g, \
               double* q_g, double t0_g, double tf_g)
```

The function **setNLPPGG** should be called for each phase of the problem and is necessary for providing the initial guess of the variables within each phase for the transcribed NLP. It is noted that linear interpolation of the initial and final guesses of the state and control is used to initialize the values for all interior collocation points.

Example of initializing the global NLPPG class structures of a single phase problem that has two state components, one control component, and one integral component with the following guesses:

$$
\begin{aligned}
x_1(t_0) &= 0, & x_1(t_f) &= 1, \\
x_2(t_0) &= 1, & x_2(t_f) &= 0, \\
u_1(t_0) &= 0, & u_1(t_f) &= 1, \\
q_1 &= 10, \\
t_0 &= 0, & t_f &= 5.
\end{aligned}
\tag{22}
$$

```
void cgpops_go(doubleMat& cgpopsResults)
{

.
.
.

    // Provide initial guess for NLP Solver
    // Phase 1 guess
    int phasenum = 0;
    double x0_g[2], xf_g[2];
    double u0_g[1], uf_g[1];
    double q_g[1];
    double t0_g, tf_g;
    x0_g[0] = 0;
    x0_g[1] = 1;
```

```
    xf_g[0]  = 1;
    xf_g[1]  = 0;
    u0_g[0]  = 0;
    uf_g[0]  = 1;
    q_g[0]   = 10;
    t0_g     = 0;
    tf_g     = 5;
    // Set parameterized constructor for NLP Phase Guess Class
    setNLPPGG(phasenum,x0_g,xf_g,u0_g,uf_g,q_g,t0_g,tf_g);

⋮
```

2.3.10   Calling setNLPWGG Function

The purpose of this function is to initialize the global NLPWG class structure that define the guess of the static parameters of the problem. The function prototype is as follows:

`void setNLPWGG(double* s_g)`

The function **setNLPWGG** is necessary for providing the initial guess for any static parameters appearing in the transcribed NLP.

Example of initializing the global NLPWG class structure of a two phase problem that has two static parameters:
$$s_1 \;\; = \;\; 0.5 \;\; , \;\; s_2 \;\; = \;\; 2 \tag{23}$$

```
void cgpops_go(doubleMat& cgpopsResults)
{

⋮

    // Whole problem guess
    double s_g[nsG];
    s_g[0] = 0.5;
    s_g[1] = 2;
    // Set parameterized constructor for NLP Phase Guess Class
    setNLPWGG(s_g);

⋮
```

2.3.11   Setting Global Variables of Vectors of SmartPtrs to Classes Inheriting from EndPoint-Function and PhasePointFunction

The purpose of assembling the global vectors of **SmartPtrs** to classes inheriting from **EndPointFunction** and **PhasePointFunction** is to assemble the optimal control problem using the classes that compute the equations defining the problem. The global variables that should be set are declared in **src/nlpGlobVarDec.hpp** and are as follows:

```
std::vector<SmartPtr<EndPointParameterEq>> eppEqVecG;
std::vector<SmartPtr<EventConstraintEq>> eveEqVecG;
SmartPtr<ObjectiveEq> objEqG;
std::vector<std::vector<SmartPtr<PhasePointParameterEq>>> pppEqVecG;
std::vector<std::vector<SmartPtr<OrdinaryDifferentialEq>>> odeEqVecG;
std::vector<std::vector<SmartPtr<PathConstraintEq>>> pthEqVecG;
std::vector<std::vector<SmartPtr<IntegrandEq>>> igdEqVecG;
```

The **eppEqVecG** variable does not need to be set if there are no classes inheriting from EndPointParameterEq. The **eveEqVecG** variable does not need to be set if there are no classes inheriting from EventConstraintEq. The **pppEqVecG** variable does not need to be set if there are no classes inheriting from PhasePointParameterEq in any phase. The **pthEqVecG** variable does not need to be set if there are no classes inheriting from PathConstraintEq in any phase. The **igdEqVecG** variable does not need to be set if there are no classes inheriting from IntegrandEq in any phase. The **objEqG** variable must always be set to a class inheriting from ObjectiveEq. The **OrdinaryDifferentialEq** variable must always be set to a vector of classes inheriting from OrdinaryDifferentialEq. All of the aforementioned variables must be set prior to calling the function **CGPOPS_IPOPT_caller**.

Example of setting global variables of vectors of **SmartPtrs** to classes inheriting from **EndPoint-Function** and **PhasePointFunction** for a two phase problem with two endpoint parameters, three event constraints, two phase point parameters in each phase, two dynamic equations in each phase, two path constraints in the first phase, one path constraint in the second phase, one integrand in the first phase, and two integrands in the second phase:

```
void cgpops_go(doubleMat& cgpopsResults)
{

    ⋮

    objEqG = new Objective;
    eppEqVecG = {new ETerm1, new ETerm2};
    eveEqVecG = {new Event1, new Event2, new Event3};
    pppEqVecG = {{new PTerm1_1, new PTerm2_1},{new PTerm1_2, new PTerm2_2}};
    odeEqVecG = {{new Dyn1_1, new Dyn2_1}, {new Dyn1_2, new Dyn2_2}};
    pthEqVecG = {{new Path1_1, new Path2_1},{new Path1_2}};
    igdEqVecG = {{new Integrand1_1},{new Integrand1_2, new Integrand2_2}};

    ⋮

    // Make call to CGOPS handler for IPOPT using user provided output settings
    CGPOPS_IPOPT_caller(cgpopsResults);
}
```

2.3.12   Calling **CGPOPS_IPOPT_caller** Function

The purpose of this function is to call the IPOPT application to solve the resulting NLP from the LGR transcription of the optimal control problem defined by the functions described in Sections 2.3.1 through 2.3.10. The function prototype is as follows:

```
void CGPOPS_IPOPT_caller(doubleMat& cgpopsResults)
```

The function **CGPOPS_IPOPT_caller** should call the IPOPT application to solve the resulting NLP of the LGR transcription defined by the user. The output of **CGPOPS_IPOPT_caller** is the doubleMat& structure cgpopsResults that holds the number of NLP solver iterations, IPOPT NLP derivative supplier computation time, CGPOPS computation time, and NLP objective value for the obtained NLP solution.

Example of calling the IPOPT application:

```
void cgpops_go(doubleMat& cgpopsResults)
{

    ⋮
```

```
    // Make call to CGOPS handler for IPOPT using user provided output settings
    CGPOPS_IPOPT_caller(cgpopsResults);
}
```

2.4   Required Functions in `cgpops_wrapper.cpp` file

Table 5 lists and describes the parameters used by the functions called in the **cgpops_wrapper.cpp** to provide the optimal control problem specifications.

Table 5: Description of parameters used by called functions in **cgpops_wrapper.cpp**.

Parameter	Type	Role	Description
cgpopsResults	doubleMat&	Output	Address of doubleMat structure for returning results output of ℂ𝔾ℙ𝕆ℙ𝕊
derivativeSupplierG	int	Global	Selects derivative supplier used by NLP solver
scaledG	int	Global	Selects if automatic scaling is used for NLP variables and functions
numintervalsG	int	Global	Sets default number of mesh intervals
initcolptsG	int	Global	Sets default number of collocation points
meshRefineTypeG	int	Global	Sets technique used for mesh refinement
minColPtsG	int	Global	Sets minimum number of collocation points for mesh refinement
maxColPtsG	int	Global	Sets maximum number of collocation points for mesh refinement
maxMeshIterG	int	Global	Sets maximum number of mesh refinements
meshTolG	double	Global	Sets mesh accuracy tolerance
saveIPOPTFlagG	int	Global	Selects if IPOPT solution is saved as m-script
saveMeshRefineFlagG	int	Global	Selects if mesh refinement data is saved as m-scripts
runIPOPTFlagG	int	Global	Selects if IPOPT application is executed to solve transcribed NLP
NLPtolG	double	Global	Sets NLP solver tolerance
NLPmaxiterG	int	Global	Sets maximum number of NLP solver iterations
useLTIHDDG	int	Global	Indicates usage of bang-bang control detection

2.4.1   **cgpops_go** Function

The purpose of this function is to setup the specifications for the optimal control problem to be solved via LGR collocation and make the call to the NLP solver application to solve the resulting NLP. The function prototype is as follows:

```
void cgpops_go(doubleMat& cgpopsResults)
```

The function **cgpops_go** should return *cgpopsResults*, a doubleMat class structure as shown in Section 2.3 containing an array with the number of NLP solver iterations, computation time per iteration, total computation time, and the NLP objective for the solution of the transcribed NLP.

Example of making the call to setup the specifications for the optimal control problem and resulting NLP to be solved:

```
int main(void)
{
```

```
    doubleMat cgpopsResults;

    cgpops_go(cgpopsResults);

    return 0;
}
```

2.4.2   Global variables for settings modification

The purpose of the parameters with the role of global, as shown in Table 5 is to allow the user to modify the default settings of $\mathbb{CGPOPS}$ and select custom preferences for the mesh initialization, mesh refinement, derivative supplier, NLP solver, and output.

Example of changing the mesh initialization settings from the default values of ten and four for the default number of mesh intervals per phase and number of collocation points per interval, respectively, to eight and three, respectively.

```
int main(void)
{
    doubleMat cgpopsResults;

    // Mesh initialization settings
    numintervalsG  = 10;   // Initial number of mesh intervals per phase (default=10)
    initcolptsG    = 4;    // Initial number of collocation points per interval \
                                (default=4)

    cgpops_go(cgpopsResults);

    return 0;
}
```

Example of changing the mesh refinement settings from the default values of one, four, ten, 20, and 10^{-7} for the default mesh refinement technique enumerate, minimum number of collocation points per interval, maximum number of collocation points per interval, maximum number of mesh iterations, to four, three, eight, 15, and 10^{-6}, respectively. The mesh refinement technique enumerates correspond to four hp-Adaptive mesh refinement techniques described in Refs. 8, 9, 10, and 11, referred to as the hp-Darby, hp-Patterson , hp-Liu, and hp-Legendre methods, respectively, which are as follows.

$1 = hp$-Patterson (default)

$2 = hp$-Darby

$3 = hp$-Liu

$4 = hp$-Legendre

```
int main(void)
{
    doubleMat cgpopsResults;

    // Mesh refinement settings
    meshRefineTypeG = 4;   // Select mesh refinement technique to be used (default=1)
    minColPtsG      = 3;   // Minimum number of collocation points per interval
```

```
                                    (default=4)
    maxColPtsG      = 8;     // Maximum number of collocation points per interval
                                    (default=10)
    maxMeshIterG    = 15;   // Maximum number of mesh iterations (default=20)
    meshTolG        = 1e-6; // Mesh tolerance (default=1e-7)

    cgpops_go(cgpopsResults);

    return 0;
}
```

Example of changing the bang-bang mesh refinement settings from the default values of zero, 0.05, two, and five for the bang-bang mesh refinement indicator, variable mesh point bracket fraction, number of segments per phase, and number of collocation points per segment to one, 0.04, three, and six, respectively. The bang-bang mesh refinement indicator calls the bang-bang optimal control mesh refinement method described in Ref. [*cite something*], which indicates as follows.

0 = off (default)

1 = on

```
int main(void)
{
    doubleMat cgpopsResults;

    // Mesh refinement settings
    useLTIHDDG = 1;
    varMeshPtsFracG = 0.04;
    varMeshPtsNumSegsG = 3;
    varMeshPtsNumColPtsG = 6;

    cgpops_go(cgpopsResults);

    return 0;
}
```

Example of changing the derivative supplier settings from the default values of zero for the default derivative supplier enumerate to two. The derivative supplier enumerates correspond to three Taylor series-based derivative approximation techniques described in Refs. 12, 13, and 14, referred to as hyper-dual derivative approximation, bicomplex-step derivative approximation, and central finite-differencing, respectively, which are as follows.

0 = Hyper-dual derivative approximation (default)

1 = Bicomplex-step derivative approximation

2 = Central finite-differencing

3 = Automatic differentiation

```
int main(void)
{
```

```
    doubleMat cgpopsResults;

    // Derivative supplier settings
    derivativeSupplierG = 0;    // Derivative supplier (default=0)

    cgpops_go(cgpopsResults);

    return 0;
}
```

Example of changing the NLP solver settings from the default values of one, 10^{-7}, 3000 and one for the default run IPOPT NLP solver indicator, NLP solver tolerance, maximum number of NLP solver iterations, and automatic scaling indicator, respectively, to zero, 10^{-8}, 2000, and zero, respectively. The run IPOPT NLP solver indicator corresponds as follows.

0 = Don't call NLP solver IPOPT

1 = Call NLP solver IPOPT (default)

Additionally, the automatic scaling indicator corresponds as follows.

0 = No autamatic scaling

1 = Autamatic scaling (default)

```
int main(void)
{
    doubleMat cgpopsResults;

    // NLP solver settings
    runIPOPTFlagG   = 0;    // Run IPOPT NLP solver indicator (default=1)
    NLPtolG         = 1e-8; // NLP solver tolerance (default=1e-7)
    NLPmaxiterG     = 2000; // Maximum number of NLP solver iterations (default=3000)
    scaledG         = 1;    // Autamtic scaling flag indicator (default=1)

    cgpops_go(cgpopsResults);

    return 0;
}
```

Example of changing the output settings from the default values of one, and zero for the default save IPOPT solution enumerate and save mesh refinement history indicator, respectively, to two and one, respectively. The save IPOPT solution enumerate corresponds as follows.

0 = Don't save IPOPT solutions

1 = Save IPOPT solution for final mesh refinement iteration (default)

2 = Save IPOPT solutions for all mesh refinement iteration

Additionally, the save mesh refinement history indicator corresponds as follows.

0 = Don't save mesh refinement history (default)

```
 1 = Save mesh refinement history

int main(void)
{
    doubleMat cgpopsResults;

    // Output save settings
    saveIPOPTFlagG      = 2;   // Save IPOPT solution (default=1)
    saveMeshRefineFlagG = 1;   // Save mesh refinement history (default=0)

    cgpops_go(cgpopsResults);

    return 0;
}
```

2.5   Required Paths in Makefile file

$\mathbb{CGPOPS}$ is compiled using a Makefile executable that requires full paths to the directories where the $\mathbb{CGPOPS}$ functions and IPOPT application are located. The user must modify the Makefile template for the appropriate operating system (macOS or Windows) to include the full paths.

Example of modifying the Makefile to include the appropriate paths, where the resulting executable binary from the Makefile execution is called *run_cgpops*.
.
.
.

```
# CHANGEME: This should be the name of your executable
EXE = run_cgpops
IPOPT_DIR = relative/path/to/IPOPT/installation/directory
CGPOPS_DIR = relative/path/to/CGPOPS/installation/directory
```

.
.
.

Example of compiling code using Makefile and then executing the resulting executable binary *run_cgpops*.

```
$ make -j
$ ./run_cgpops
```

3   Examples of Using $\mathbb{CGPOPS}$

In this section we provide seven examples of using $\mathbb{CGPOPS}$. 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. 15. The second example is a multiple-stage launch vehicle ascent problem taken from Refs. 16, 17, and 2. The third example is a tumor anti-angiogenesis optimal control problem from Refs. 18 and 2. The fourth example is the reusable launch vehicle entry problem taken from Ref. 2. The fifth example is the minimum time-to-climb of a supersonic aircraft taken from Refs. 19 and 2. The sixth example is the optimal control of a hang glider and is taken from Ref. 20. Finally, the seventh example is the optimal control of a two-strain tuberculosis model and is taken from Ref. 21. For each example the optimal control problem is described quantitatively, the $\mathbb{CGPOPS}$

code is provided, and the solution obtained using $\mathbb{CGPOPS}$ is provided. For reference, all examples were solved on a 2.9 GHz Intel Core i7 MacBook Pro running MAC OS-X version 10.13.6 (High Sierra) with 16GB 2133MHz LPDDR3 of RAM. C++ files were compiled using Apple LLVM version 9.1.0 (clang-1000.10.44.2). All plots were created using MATLAB Version R2017B (build 9.3.0.713579). The NLP solver utilized to solve the transcribed NLP is IPOPT,[22] which is freely available at https://projects.coin-or.org/Ipopt.

3.1 Hyper-Sensitive Problem

Consider the following *hyper-sensitive*[15, 23–25] optimal control problem adapted from Ref. 15. Minimize the cost functional

$$J = \frac{1}{2} \int_0^{t_f} \left(x^2 + u^2 \right) \, dt \, , \tag{24}$$

subject to the dynamic constraint

$$\dot{x} = -x^3 + u \, , \tag{25}$$

and the boundary conditions

$$x(0) \;\; = \;\; 1.5 \, , \qquad x(t_f) \;\; = \;\; 1 \, , \tag{26}$$

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. 15 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. (24)–(26) was solved using $\mathbb{CGPOPS}$ witht the NLP solver IPOPT and a mesh refinement tolerance of $\epsilon = 10^{-7}$. In order to solve this problem using $\mathbb{CGPOPS}$, the complete C++ code that was written to solve the hyper-sensitive optimal control problem of Eqs. (24)–(26) is given below.

```
// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpopsAuxDec.hpp
// CGPOPS NLP Variables
// First declaration or global variables used in CGPOPS for NLP
//


#ifndef __CGPOPS_AUX_DEC_HPP__
#define __CGPOPS_AUX_DEC_HPP__



// Soft Lunar Landing problem parameters
double gG = 1.5;



#endif
```

```
// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpopsAuxExt.hpp
// CGPOPS Auxiliary Variables
// External declaration of global variables used in CGPOPS for NLP
//

#ifndef __CGPOPS_AUX_EXT_HPP__
#define __CGPOPS_AUX_EXT_HPP__


// Soft Lunar Landing problem parameters
extern double gG;


#endif




// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_gov.hpp
// CGPOPS Tool Box
// Declare governing equations of continouos time optimal control problem here
// Hyper sensitive problem


#ifndef __CGPOPS_GOV_HPP__
#define __CGPOPS_GOV_HPP__


#include "nlpGlobVarExt.hpp"
#include "cgpopsFuncDec.hpp"
#include "cgpopsAuxExt.hpp"


// $#$#$

/*--------------------------------EndPointFunction--------------------------------*/

class MinIntegral : public ObjectiveEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
```

```
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s, T* e);
};

/*---------------------------------PhasePointFunction---------------------------------*/

class XDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class IntegralCost : public IntegrandEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

#endif


// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_gov.cpp
// CGPOPS Tool Box
// Define governing equations of continouos time optimal control problem here
// Hyper sensitive problem


#include "cgpops_gov.hpp"

#define x_s        x[0]
#define u_c        u[0]
#define integral   q[0][0]

template <class T> void MinIntegral::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
                                            T** tf, T* s, T* e)
{
    lhs = integral;
```

```
}
template <class T> void XDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = -pow(x_s,3.0) + u_c;
}
template <class T> void IntegralCost::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = 0.5*(x_s*x_s + u_c*u_c);
}


void MinIntegral::eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                          double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinIntegral::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                          HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinIntegral::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                          Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void XDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void XDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                   HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void XDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                   Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void IntegralCost::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                           double* p)
{eq_def(lhs,x,u,t,s,p);}
void IntegralCost::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                           HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void IntegralCost::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                           Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}




// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main.hpp
// CGPOPS Tool Box
// Declare main continouos time optimal control problem here
//


#ifndef __CGPOPS_MAIN_HPP__
#define __CGPOPS_MAIN_HPP__

#include "cgpopsFuncDec.hpp"
```

```cpp
#include "nlpGlobVarExt.hpp"
#include "cgpopsAuxExt.hpp"


// $#$#$


void cgpops_go(doubleMat& cgpopsResults);


#endif


// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main_test.cpp
// Test Main
// Test CGPOPS functions
//

#include "cgpops_main.hpp"


void cgpops_go(doubleMat& cgpopsResults)
{
    // Define Global Variables used to determine problem size
    PG      = 1;    // Number of phases in problem
    nsG     = 0;    // Number of static parameters in problem
    nbG     = 0;    // Number of event constraints in problem
    nepG    = 0;    // Number of endpoint parameters in problem

    // Allocate memory for each phase
    initGlobalVars();

    // Define number of components for each parameter in problem phases
    // Phase 1 parameters
    nxG[0]  = 1;    // Number of state components in phase 1
    nuG[0]  = 1;    // Number of control components in phase 1
    nqG[0]  = 1;    // Number of integral constraints in phase 1
    ncG[0]  = 0;    // Number of path constraints in phase 1
    nppG[0] = 0;    // Number of phase parameters in phase 1

//    setGlobalTabularData(); // Set any global tabular data used in problem

    // Define mesh grids for each phase in problem for LGR collocation
    // Phase 1 mesh grid
    int M1 = numintervalsG;        // Number of intervals used in phase 1 mesh
    int initcolpts = initcolptsG;  // Initial number of collocation points in each
                                   // interval
    double fraction1[M1];          // Allocate memory for fraction vector for phase 1
                                   // mesh
    int colpoints1[M1];            // Allocate memory for colpoints vector for phase 1
                                   // mesh
```

```
    for (int m=0; m<M1; m++)
    {
        fraction1[m] = 1.0/((double) M1);
        colpoints1[m] = initcolpts;
    }
    setRPMDG(0,M1,fraction1,colpoints1);

    // Set information for transcribed NLP resulting from LGR collocation using defined
    // mesh grid
    setInfoNLPG();

    /*-----------------------------Provide Problem Bounds----------------------------*/
    // Phase 1
    double t0min = 0,  t0max = 0;          // minimum and maximum initial times
    double tfmin = 10000,  tfmax = 10000;  // minimum and maximum final times
    double x0 = 1,   xf = 1.5;             // initial and final values for height
    double xmin = -50, xmax = 50;          // minimum and maximum height
    double umin = -50, umax = 50;          // minimum and maximum control
    double qmin = 0,   qmax = 100000;      // minimum and maximum integral values
//    double cmin = NAN, cmax = NAN;  // minimum and maximum path constraint values (N/A)

    // Throughout
//    double smin = NAN, smax = NAN;  // minimum and maximum static parameter values (N/A)
//    double bmin = NAN, bmax = NAN;  // minimum and maximum event constraint values (N/A)

    // Define Bounds of Optimal Control Problem
    // Phase 1 bounds
    int phase1 = 0;
    double x0l1[nxG[phase1]],   x0u1[nxG[phase1]];
    double xfl1[nxG[phase1]],   xfu1[nxG[phase1]];
    double xl1[nxG[phase1]],    xu1[nxG[phase1]];
    double ul1[nuG[phase1]],    uu1[nuG[phase1]];
    double ql1[nqG[phase1]],    qu1[nqG[phase1]];
    double cl1[ncG[phase1]],    cu1[ncG[phase1]];
    double t0l1,    t0u1;
    double tfl1,    tfu1;
    x0l1[0] = x0;
    x0u1[0] = x0;
    xfl1[0] = xf;
    xfu1[0] = xf;
    xl1[0]  = xmin;
    xu1[0]  = xmax;
    ul1[0]  = umin;
    uu1[0]  = umax;
    ql1[0]  = qmin;
    qu1[0]  = qmax;
    t0l1    = t0min;
    t0u1    = t0max;
    tfl1    = tfmin;
    tfu1    = tfmax;

    // Set parameterized constructor for NLP Phase Bounds Class
    setNLPPBG(phase1,x0l1,x0u1,xfl1,xfu1,xl1,xu1,ul1,uu1,ql1,qu1,cl1,cu1,t0l1,t0u1,tfl1,
```

```
            tfu1);

    // Whole problem bounds
    double sl[nsG], su[nsG];
    double bl[nbG], bu[nbG];

    // Set parameterized constructor for NLP Whole Bounds Class
    setNLPWBG(sl,su,bl,bu);

    // Provide initial guess for NLP Solver
    // Phase 1 guess
    double x0g1[nxG[phase1]],   xfg1[nxG[phase1]];
    double u0g1[nuG[phase1]],   ufg1[nuG[phase1]];
    double qg1[nqG[phase1]];
    double t0g1,     tfg1;
    x0g1[0] = x0;
    xfg1[0] = xf;
    u0g1[0] = umin;
    ufg1[0] = umin;
    qg1[0]  = 0;
    t0g1    = t0min;
    tfg1    = tfmax;

    // Set parameterized constructor for NLP Phase Guess Class
    setNLPPGG(phase1,x0g1,xfg1,u0g1,ufg1,qg1,t0g1,tfg1);

    // Whole problem guess
    double sg[nsG];

    // Set parameterized constructor for NLP Phase Guess Class
    setNLPWGG(sg);

    // Set ocp functions
    objEqG = new MinIntegral;
    odeEqVecG = {{new XDot}};
    igdEqVecG = {{new IntegralCost}};

    // Make call to CGOPS handler for IPOPT using user provided output settings
    CGPOPS_IPOPT_caller(cgpopsResults);

}



// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main_test.cpp
// Test Main
// Test CGPOPS functions
//

#include "nlpGlobVarDec.hpp"
#include "cgpopsAuxDec.hpp"
```

```
#include "cgpops_main.hpp"


int main(void)
{
    printf("\nCGPOPS TESTING\n\n\n");

    int numTestRuns = 1;
    int numDS = 1;
    int DSSelect = 0;
    doubleMatMat cgpopsResultsMatMat(numTestRuns*numDS);
    doubleMat cgpopsResults;

    // Derivative supplier settings
    derivativeSupplierG = 0;     // Derivative supplier (default=0)
    scaledG             = 1;     // Scaling flag (default 1=on)

    // Mesh initialization settings
    numintervalsG    = 10;   // Initial number of mesh intervals per phase (default=10)
    initcolptsG      = 3;    // Initial number of collocation points per interval
                             // (default=4)

    // Mesh refinement settings
    meshRefineTypeG = 4;     // Select mesh refinement technique to be used (default=1)
    minColPtsG      = 3;     // Mininum number of collocation points used in an interval
                             // (default=4)
    maxColPtsG      = 10;    // Maximum number of collocation points used in an interval
                             // (default=10)
    maxMeshIterG    = 30;    // Maximum number of mesh iterations (default=20)
    meshTolG        = 1e-6;  // Mesh tolerance (default=1e-7)

    // Output save settings
    saveIPOPTFlagG       = 1;    // Save IPOPT solution (default=1)
    saveMeshRefineFlagG  = 1;    // Save mesh refinement history (default=0)
    saveHamiltonianG     = 0;    // Save Hamiltonian values (default=0)
    saveLTIHG            = 0;    // Save linear terms in Hamiltonian values (default=0)

    // IPOPT settings
    runIPOPTFlagG   = 1;     // Run IPOPT (default=1)
    NLPtolG         = 1e-7;  // NLP Solver tolerance (default=1e-7)
    NLPmaxiterG     = 3000;  // Maximum number of iterations allowed for NLP solver
    useLTIHDDG = 1;

    /*----------------------Changes to global parameter settings----------------------*/

    for (int ds=0; ds<numDS; ds++)
    {
        derivativeSupplierG = DSSelect;
        if (derivativeSupplierG==DSSelect)
        {
            for (int ni=0; ni<numTestRuns; ni++)
            {
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns] = getDoubleMat(6);
```

```
                printf("\nDerivativeSupplier = %d",derivativeSupplierG);
                printf("\nNumIntervals = %d",numintervalsG);
                cgpops_go(cgpopsResults);
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[0] = derivativeSupplierG;
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[1] = numintervalsG;
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[2] = cgpopsResults.val[0];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[3] = cgpopsResults.val[1];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[4] = cgpopsResults.val[2];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[5] = cgpopsResults.val[3];
            }
        }
    }

    printf4MSCRIPT("cgpopsResultsMatMat",cgpopsResultsMatMat);
    printf("\n\n\n\n\n");

    return 0;
}
```

The state $x(t)$, control $u(t)$, and mesh refinement history that arise from the execution of $\mathbb{CGPOPS}$ 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 6.

Table 6: Relative Error Estimate vs. Mesh Refinement Iteration for Hyper-Sensitive Problem.

Mesh Refinement Iteration	Relative Error Estimate
1	9.5699×10^{1}
2	1.2305×10^{1}
3	1.7686×10^{0}
4	2.5441×10^{-1}
5	4.4309×10^{-3}
6	4.4803×10^{-5}
7	6.8441×10^{-8}

3.2   Multiple-Stage Launch Vehicle Ascent Problem

The problem considered in this section is the ascent of a multiple-stage launch vehicle. The objective is to maneuver the launch vehicle from the ground to the target orbit while maximizing the remaining fuel in the upper stage. It is noted that this example is is found verbatim in Refs. 16, 17, and 2.

3.2.1   Vehicle Properties

The goal of this launch vehicle ascent problem is to steer the vehicle from launch to a geostationary transfer orbit (GTO). The motion of the 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. 17 or 2 and is shown in Table 7.

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

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

(c) Mesh Refinement History.
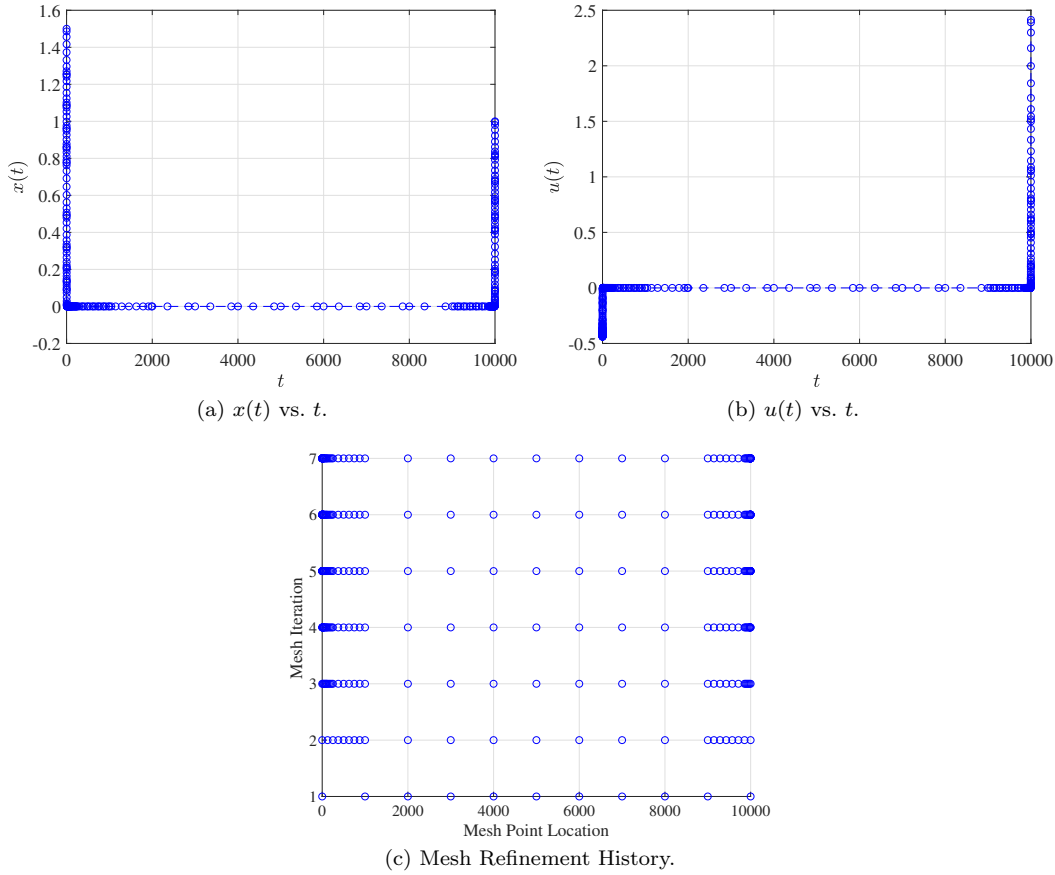
Figure 1: Solution to Hyper-Sensitive Problem Obtained Using $\mathbb{CGPOPS}$ with the NLP Solver IPOPT using the Hyper-Dual Derivative Supplier and a Mesh Refinement Tolerance of 10^{-7}.

3.2.2   Dynamic Model

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

$$
\begin{aligned}
\dot{\mathbf{r}} &= \mathbf{v} \ , \\
\dot{\mathbf{v}} &= -\frac{\mu}{\|\mathbf{r}\|^3}\mathbf{r} + \frac{T}{m}\mathbf{u} + \frac{\mathbf{D}}{m} \ , \\
\dot{m} &= -\frac{T}{g_0 I_{sp}} \ ,
\end{aligned}
\tag{27}
$$

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

$$
\mathbf{D} = -\frac{1}{2}C_D A_{ref}\rho\|\mathbf{v}_{\text{rel}}\|\mathbf{v}_{\text{rel}} \ ,
\tag{28}
$$

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

$$
\mathbf{v}_{\text{rel}} = \mathbf{v} - \boldsymbol{\omega} \times \mathbf{r} \ ,
\tag{29}
$$

Table 7: 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

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

$$\rho = \rho_0 \exp[-h/h_0] \ , \tag{30}$$

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 8.

Table 8: Constants used in the launch vehicle example.

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

3.2.3 Constraints

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

$$\begin{aligned}
\mathbf{r}(t_0) &= \mathbf{r}_0 = \begin{bmatrix} 5605.2 & 0 & 3043.4 \end{bmatrix}^T \quad \text{km} \ , \\
\mathbf{v}(t_0) &= \mathbf{v}_0 = \begin{bmatrix} 0 & 0.4076 & 0 \end{bmatrix}^T \quad \text{km/s} \ , \\
m(t_0) &= m_0 = 301454 \quad \text{kg} \ ,
\end{aligned} \tag{31}$$

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

$$\begin{aligned}
a_f &= 24361.14 \ \text{km} \ , \\
e_f &= 0.7308 \ , \\
i_f &= 28.5 \ \text{deg} \ , \\
\Omega_f &= 269.8 \ \text{deg} \ , \\
\omega_f &= 130.5 \ \text{deg} \ .
\end{aligned} \tag{32}$$

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.[26]

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

$$|\mathbf{r}| \geq R_r \ , \tag{33}$$

where R_e is the radius of the Earth, as seen in Table 8. Next, a path constraint is imposed on the control to guarantee that the control vector is unit length, so that

$$\|\mathbf{u}\|_2^2 = u_1^2 + u_2^2 + u_3^2 = 1 \ . \tag{34}$$

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

$$
\begin{aligned}
\mathbf{r}^{(p)}(t_f) - \mathbf{r}^{(p+1)}(t_0) &= \mathbf{0} \ , \\
\mathbf{v}^{(p)}(t_f) - \mathbf{v}^{(p+1)}(t_0) &= \mathbf{0} \ , \qquad (p = 1, \ldots, 3) \ , \\
m^{(p)}(t_f) - m_{dry}^{(p)} - m^{(p+1)}(t_0) &= 0 \ ,
\end{aligned}
\tag{35}
$$

where the superscript (p) represents the phase number. The optimal control problem is then to find the control, $\mathbf{u}$, that minimizes the cost function

$$J = -m^{(4)}(t_f) \ , \tag{36}$$

subject to the conditions of Eqs. (27), (31), (32), (33), and (34).

The C++ code that solves the multiple-stage launch vehicle ascent problem using $\mathbb{CGPOPS}$ is omitted for brevity, but may be found in the examples/launch folder. 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 that were then normalized using the Earth's radius, the gravitational parameter for Earth, and the initial mass of the launch vehicle, and the built-in autoscaling procedure was used. The output of the code from $\mathbb{CGPOPS}$ is summarized in the following three plots that contain the altitude, speed, and controls.

3.3 Tumor-Antiangiogenesis Optimal Control Problem

Consider the following cancer treatment optimal control problem taken from Ref. 18. The objective is to minimize

$$p(t_f) \ , \tag{37}$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{p}(t) &= -\xi p(t) \ln\left(\frac{p(t)}{q(t)}\right) \ , \\
\dot{q}(t) &= q(t)\left[b - \mu - dp^{2/3}(t) - Gu(t)\right] \ ,
\end{aligned}
\tag{38}
$$

with the initial conditions

$$
\begin{aligned}
p(0) &= p_0 \ , \\
q(0) &= q_0 \ ,
\end{aligned}
\tag{39}
$$

and the integral constraint

$$\int_0^{t_f} u(\tau)d\tau \leq A \ . \tag{40}$$

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.

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

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

(c) $\mathbf{u}(t)$ vs. t.
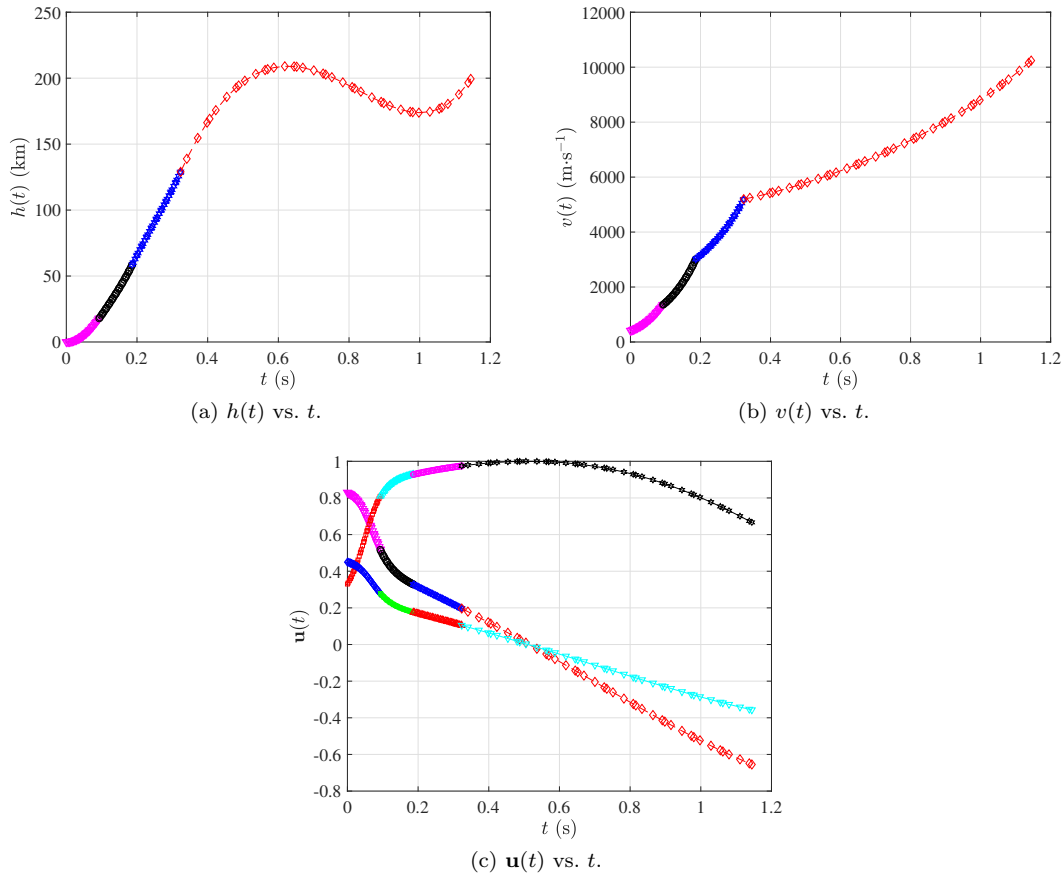
Figure 2: Solution to Launch Vehicle Ascent Problem Using $\mathbb{CGPOPS}$ with the NLP Solver IPOPT using the Hyper-Dual Derivative Supplier and a Mesh Refinement Tolerance of 10^{-6}.

```cpp
// cgpopsAuxDec.hpp
// CGPOPS NLP Variables
// First declaration or global variables used in CGPOPS for NLP
// By Yunus Agamawi
// 11/5/18
//


#ifndef __CGPOPS_AUX_DEC_HPP__
#define __CGPOPS_AUX_DEC_HPP__


// Tumor Anti-Angiogenesis problem parameters
double zetaG = 0.084;    // per day
double bG = 5.85;        // per day
double dG = 0.00873;     // per mm^2 per day
double GG = 0.15;        // per mg of dose per day
double muG = 0.02;       // per day


#endif
```

```
// cgpopsAuxExt.hpp
// CGPOPS Auxiliary Variables
// External declaration of global variables used in CGPOPS for NLP
// By Yunus Agamawi
// 11/5/18
//

#ifndef __CGPOPS_AUX_EXT_HPP__
#define __CGPOPS_AUX_EXT_HPP__


// Tumor Anti-Angiogenesis problem parameters
extern double zetaG;    // per day
extern double bG;       // per day
extern double dG;       // per mm^2 per day
extern double GG;       // per mg of dose per day
extern double muG;      // per day


#endif




// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_gov.hpp
// CGPOPS Tool Box
// Declare governing equations of continouos time optimal control problem here
// Tumor anti-angiogenesis


#ifndef __CGPOPS_GOV_HPP__
#define __CGPOPS_GOV_HPP__


#include "nlpGlobVarExt.hpp"
#include "cgpopsFuncDec.hpp"
#include "cgpopsAuxExt.hpp"


// $#$#$
```

```
/*---------------------------------EndPointFunction----------------------------------*/

class MinFinalP : public ObjectiveEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

/*--------------------------------PhasePointFunction----------------------------------*/

class PDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class QDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class ControlCost : public IntegrandEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
```

```
                                Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

#endif


// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_gov.cpp
// CGPOPS Tool Box
// Define governing equations of continouos time optimal control problem here
// Tumor anti-angiogenesis


#include "cgpops_gov.hpp"

#define p_s x[0]
#define q_s x[1]

#define u_c u[0]

#define pf  xf[PG-1][0]

template <class T> void MinFinalP::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf,
                                          T* s, T* e)
{
    lhs = pf;
}
template <class T> void PDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = -zetaG*p_s*log(p_s/q_s);
}
template <class T> void QDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = q_s*(bG-(muG+(dG*(pow(p_s,(2.0/3.0)))) + GG*u_c));
}
template <class T> void ControlCost::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = u_c;
}

void MinFinalP::eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                        double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinFinalP::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                        HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinFinalP::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                        Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void PDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
```

```
{eq_def(lhs,x,u,t,s,p);}
void PDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                   HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void PDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                   Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void QDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void QDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                   HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void QDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                   Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void ControlCost::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                          double* p)
{eq_def(lhs,x,u,t,s,p);}
void ControlCost::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                          HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void ControlCost::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                          Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}




// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main.hpp
// CGPOPS Tool Box
// Declare main continouos time optimal control problem here
//


#ifndef __CGPOPS_MAIN_HPP__
#define __CGPOPS_MAIN_HPP__

#include "cgpopsFuncDec.hpp"
#include "nlpGlobVarExt.hpp"
#include "cgpopsAuxExt.hpp"


// $#$#$


void cgpops_go(doubleMat& cgpopsResults);


#endif
```

```cpp
// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main_test.cpp
// Test Main
// Test CGPOPS functions
//

#include "cgpops_main.hpp"


void cgpops_go(doubleMat& cgpopsResults)
{
    // Define Global Variables used to determine problem size
    PG      = 1;    // Number of phases in problem
    nsG     = 0;    // Number of static parameters in problem
    nbG     = 0;    // Number of event constraints in problem
    nepG    = 0;    // Number of endpoint parameters in problem

    // Allocate memory for each phase
    initGlobalVars();

    // Define number of components for each parameter in problem phases
    // Phase 1 parameters
    nxG[0]  = 2;    // Number of state components in phase 1
    nuG[0]  = 1;    // Number of control components in phase 1
    nqG[0]  = 1;    // Number of integral constraints in phase 1
    ncG[0]  = 0;    // Number of path constraints in phase 1
    nppG[0] = 0;    // Number of phase parameters in phase 1

//    setGlobalTabularData(); // Set any global tabular data used in problem

    // Define mesh grids for each phase in problem for LGR collocation
    // Phase 1 mesh grid
    int M1 = numintervalsG;        // Number of intervals used in phase 1 mesh
    int initcolpts = initcolptsG;  // Initial number of collocation points in each
                                   // interval
    double fraction1[M1];          // Allocate memory for fraction vector for phase 1
                                   // mesh
    int colpoints1[M1];            // Allocate memory for colpoints vector for phase 1
                                   // mesh
    for (int m=0; m<M1; m++)
    {
        fraction1[m] = 1.0/((double) M1);
        colpoints1[m] = initcolpts;
    }
    setRPMDG(0,M1,fraction1,colpoints1);

    // Set information for transcribed NLP resulting from LGR collocation using defined
    // mesh grid
    setInfoNLPG();

    /*---------------------------Provide Problem Bounds---------------------------*/
    // Phase 1
```

```
    double a = 75, A = 15;
    double t0min = 0,  t0max = 0;               // minimum and maximum initial times
    double tfmin = 0.1,  tfmax = 5;             // minimum and maximum final times
    double xpmin = 0.1;                         // minimum state p values
    double xpmax = pow(((bG-muG)/dG),1.5);      // maximum state p values
    double xqmin = xpmin, xqmax = xpmax;        // minimum and maximum state q values
    double xp0 = xpmax*0.5;
    double xq0 = xqmax*0.25;
    double umin = 0, umax = a;                  // minimum and maximum control
    double qmin = 0,   qmax = A;                // minimum and maximum integral values


    // Throughout


    // Define Bounds of Optimal Control Problem
    // Phase 1 bounds
    int phase1 = 0;
    double x0l1[nxG[phase1]],   x0u1[nxG[phase1]];
    double xfl1[nxG[phase1]],   xfu1[nxG[phase1]];
    double xl1[nxG[phase1]],    xu1[nxG[phase1]];
    double ul1[nuG[phase1]],    uu1[nuG[phase1]];
    double ql1[nqG[phase1]],    qu1[nqG[phase1]];
    double cl1[ncG[phase1]],    cu1[ncG[phase1]];
    double t0l1,    t0u1;
    double tfl1,    tfu1;
    x0l1[0] = xp0;
    x0l1[1] = xq0;
    x0u1[0] = xp0;
    x0u1[1] = xq0;
    xfl1[0] = xpmin;
    xfl1[1] = xqmin;
    xfu1[0] = xpmax;
    xfu1[1] = xqmax;
    xl1[0]  = xpmin;
    xl1[1]  = xqmin;
    xu1[0]  = xpmax;
    xu1[1]  = xqmax;
    ul1[0]  = umin;
    uu1[0]  = umax;
    ql1[0]  = qmin;
    qu1[0]  = qmax;
    t0l1    = t0min;
    t0u1    = t0max;
    tfl1    = tfmin;
    tfu1    = tfmax;


    // Set parameterized constructor for NLP Phase Bounds Class
    setNLPPBG(phase1,x0l1,x0u1,xfl1,xfu1,xl1,xu1,ul1,uu1,ql1,qu1,cl1,cu1,t0l1,t0u1,tfl1,
            tfu1);


    // Whole problem bounds
    double sl[nsG], su[nsG];
    double bl[nbG], bu[nbG];
```

```
    // Set parameterized constructor for NLP Whole Bounds Class
    setNLPWBG(sl,su,bl,bu);

    // Provide initial guess for NLP Solver
    // Phase 1 guess
    double x0g1[nxG[phase1]],   xfg1[nxG[phase1]];
    double u0g1[nuG[phase1]],   ufg1[nuG[phase1]];
    double qg1[nqG[phase1]];
    double t0g1,    tfg1;
    x0g1[0] = xp0;
    x0g1[1] = xq0;
    xfg1[0] = xpmax;
    xfg1[1] = xqmax;
    u0g1[0] = umax;
    ufg1[0] = umax;
    qg1[0]  = 7.5;
    t0g1    = 0;
    tfg1    = 1;

    // Set parameterized constructor for NLP Phase Guess Class
    setNLPPGG(phase1,x0g1,xfg1,u0g1,ufg1,qg1,t0g1,tfg1);

    // Whole problem guess
    double sg[nsG];

    // Set parameterized constructor for NLP Phase Guess Class
    setNLPWGG(sg);

    // Set ocp functions
    objEqG = new MinFinalP;
    odeEqVecG = {{new PDot, new QDot}};
    igdEqVecG = {{new ControlCost}};

    // Make call to CGOPS handler for IPOPT using user provided output settings
    CGPOPS_IPOPT_caller(cgpopsResults);

}


// cgpops_main_test.cpp
// Test Main
// Test CGPOPS functions
// By Yunus Agamawi
// 4/21/18
// Version 16
/*
Capabilities:
 - TBD
*/

#include "nlpGlobVarDec.hpp"
#include "cgpopsAuxDec.hpp"
#include "cgpops_main.hpp"
```

```
int main(void)
{
    printf("\nCGPOPS TESTING\n\n\n");

    int numTestRuns = 1;
    int numDS = 1;
    int DSSelect = 0;
    doubleMatMat cgpopsResultsMatMat(numTestRuns*numDS);
    doubleMat cgpopsResults;

    // Derivative supplier settings
    derivativeSupplierG = 0;     // Derivative supplier (default=0)
    scaledG             = 1;     // Scaling flag (default 1=on)

    // Mesh initialization settings
    numintervalsG   = 10;   // Initial number of mesh intervals per phase (default=10)
    initcolptsG     = 5;    // Initial number of collocation points per interval
                            // (default=4)

    // Mesh refinement settings
    meshRefineTypeG = 1;     // Select mesh refinement technique to be used (default=1)
    minColPtsG      = 3;     // Mininum number of collocation points used in an interval
                             // (default=4)
    maxColPtsG      = 10;    // Maximum number of collocation points used in an interval
                             // (default=10)
    maxMeshIterG    = 3;     // Maximum number of mesh iterations (default=20)
    meshTolG        = 1e-6; // Mesh tolerance (default=1e-7)

    // Output save settings
    saveIPOPTFlagG       = 1;   // Save IPOPT solution (default=1)
    saveMeshRefineFlagG  = 0;   // Save mesh refinement history (default=0)
    saveHamiltonianG     = 0;   // Save Hamiltonian values (default=0)
    saveLTIHG            = 0;   // Save linear terms in Hamiltonian values (default=0)

    // IPOPT settings
    runIPOPTFlagG   = 1;    // Run IPOPT (default=1)
    NLPtolG         = 1e-9; // NLP Solver tolerance (default=1e-7)
    NLPmaxiterG     = 3000; // Maximum number of iterations allowed for NLP solver
    useLTIHDDG = 1;

    /*----------------------Changes to global parameter settings---------------------*/

    for (int ds=0; ds<numDS; ds++)
    {
        derivativeSupplierG = DSSelect;
        if (derivativeSupplierG==DSSelect)
        {
            for (int ni=0; ni<numTestRuns; ni++)
            {
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns] = getDoubleMat(6);
                printf("\nDerivativeSupplier = %d",derivativeSupplierG);
```

```
                printf("\nNumIntervals = %d",numintervalsG);
                cgpops_go(cgpopsResults);
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[0] = derivativeSupplierG;
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[1] = numintervalsG;
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[2] = cgpopsResults.val[0];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[3] = cgpopsResults.val[1];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[4] = cgpopsResults.val[2];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[5] = cgpopsResults.val[3];
            }
        }
    }

//    printf4MSCRIPT("cgpopsResultsMatMat",cgpopsResultsMatMat);
    printf("\n\n\n\n\n");

    return 0;
}
```

The solution obtained using $\mathbb{CGPOPS}$ using the NLP solver IPOPT with a mesh refinement error tolerance of 10^{-6} is shown in Figs. 3a–3c. Note that in this example we have also provided the costate of the optimal control problem, where the costate is estimated using the Radau pseudospectral costate estimation method described in Refs. 3, 4, and 5.

3.4   Reusable Launch Vehicle Entry

Consider the following optimal control problem of maximizing the crossrange during the atmospheric entry of a reusable launch vehicle and taken verbatim from Ref. 2. Minimize the cost functional

$$J = -\phi(t_f) \ , \tag{41}$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{r} &= v \sin \gamma \ , \\
\dot{\theta} &= \frac{v \cos \gamma \sin \psi}{r \cos \phi} \ , \\
\dot{\phi} &= \frac{v \cos \gamma \cos \psi}{r} \ , \\
\dot{v} &= -\frac{F_d}{m} - F_g \sin \gamma \ , \\
\dot{\gamma} &= \frac{F_l \cos \sigma}{mv} - \left(\frac{F_g}{v} - \frac{v}{r} \right) \cos \gamma \ , \\
\dot{\psi} &= \frac{F_l \sin \sigma}{mv \cos \gamma} + \frac{v \cos \gamma \sin \psi \tan \phi}{r} \ ,
\end{aligned} \tag{42}
$$

and the boundary conditions

$$
\begin{aligned}
r(0) &= 79248 + R_e \text{ m} &, & & r(t_f) &= 24384 + R_e \text{ m} \ , \\
\theta(0) &= 0 \text{ deg} &, & & \theta(t_f) &= \text{Free} \ , \\
\phi(0) &= 0 \text{ deg} &, & & \phi(t_f) &= \text{Free} \ , \\
v(0) &= 7803 \text{ m/s} &, & & v(t_f) &= 762 \text{ m/s} \ , \\
\gamma(0) &= -1 \text{ deg} &, & & \gamma(t_f) &= -5 \text{ deg} \ , \\
\psi(0) &= 90 \text{ deg} &, & & \psi(t_f) &= \text{Free} \ .
\end{aligned} \tag{43}
$$

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

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

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

(d) Mesh Refinement History.
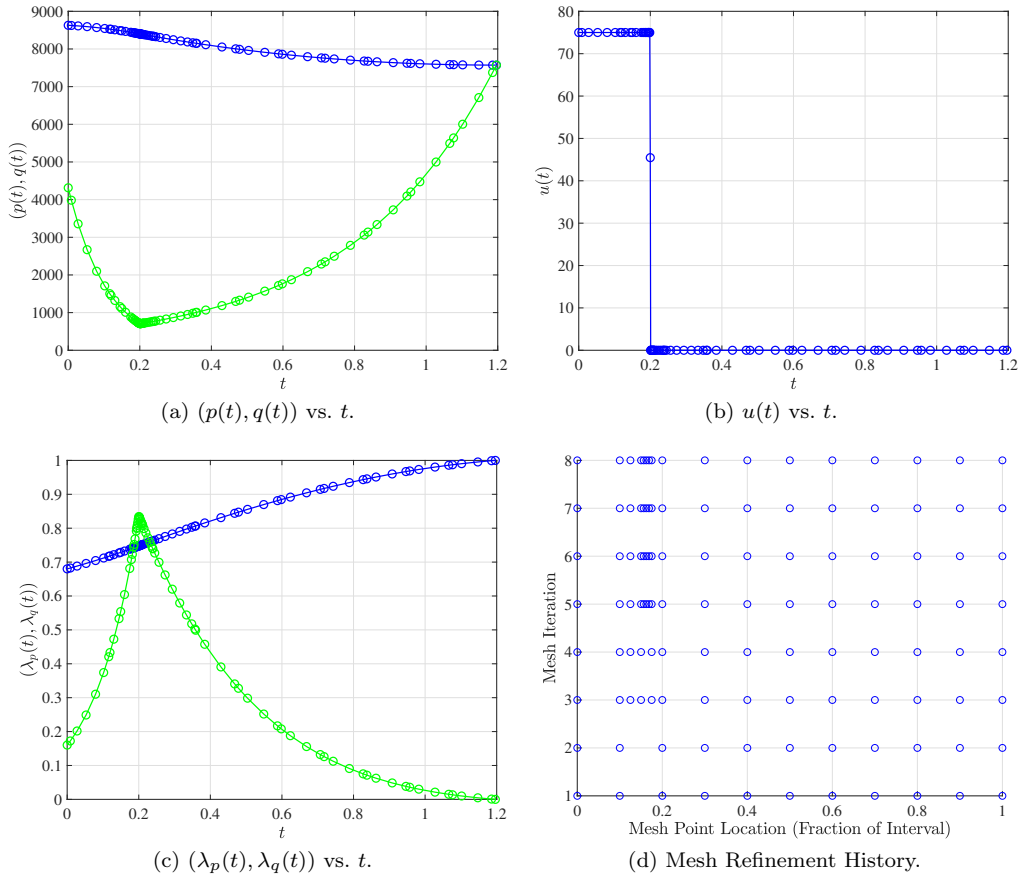
Figure 3: Solution to Tumor Anti-Angiogenesis Optimal Control Problem Using *CGPOPS* with the NLP Solver IPOPT using the Hyper-Dual Derivative Supplier and a Mesh Refinement Tolerance of 10^{-6}.

Further details of this problem, including the aerodynamic model, can be found in Ref. 2. The code for solving this problem is shown below.

```
// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpopsAuxDec.hpp
// CGPOPS NLP Variables
// First declaration or global variables used in CGPOPS for NLP
//

#ifndef __CGPOPS_AUX_DEC_HPP__
#define __CGPOPS_AUX_DEC_HPP__


// RLV Entry problem parameters
double ReG  = 6371203.92;              //  Equatorial Radius of Earth (m)
double SG   = 249.9091776;             //  Vehicle Reference Area (m^2)
double cl0G = -0.2070;                 //  Parameters for Lift Coefficient
double cl1G = 1.6756;                  //  Parameters for Lift Coefficient
double cd0G = 0.0785;                  //  Parameters for Drag Coefficient
```

```
double cd1G  = -0.3529;                   //  Parameters for Drag Coefficient
double cd2G  = 2.0400;                    //  Parameters for Drag Coefficient
double b0G   = 0.07854;                   //  Parameters for Heat Rate Model
double b1G   = -0.061592;                 //  Parameters for Heat Rate Model
double b2G   = 0.00621408;                //  Parameters for Heat Rate Model
double HG    = 7254.24;                   //  Density Scale Height (m)
double al0G  = -0.20704;                  //  Parameters for Heat Rate Model
double al1G  = 0.029244;                  //  Parameters for Heat Rate Model
double rho0G = 1.225570827014494;         //  Sea Level Atmospheric Density (kg/m^3)
double muG   = 3.986031954093051e14;      //  Earth Gravitational Parameter (m^3/s^2)
double massG = 92079.2525560557;          //  Vehicle Mass (kg)



#endif









// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpopsAuxExt.hpp
// CGPOPS Auxiliary Variables
// External declaration of global variables used in CGPOPS for NLP
//

#ifndef __CGPOPS_AUX_EXT_HPP__
#define __CGPOPS_AUX_EXT_HPP__



// RLV Entry problem parameters
extern double ReG;      //  Equatorial Radius of Earth (m)
extern double SG;       //  Vehicle Reference Area (m^2)
extern double cl0G;     //  Parameters for Lift Coefficient
extern double cl1G;     //  Parameters for Lift Coefficient
extern double cd0G;     //  Parameters for Drag Coefficient
extern double cd1G;     //  Parameters for Drag Coefficient
extern double cd2G;     //  Parameters for Drag Coefficient
extern double b0G;      //  Parameters for Heat Rate Model
extern double b1G;      //  Parameters for Heat Rate Model
extern double b2G;      //  Parameters for Heat Rate Model
extern double HG;       //  Density Scale Height (m)
extern double al0G;     //  Parameters for Heat Rate Model
extern double al1G;     //  Parameters for Heat Rate Model
extern double rho0G;    //  Sea Level Atmospheric Density (kg/m^3)
extern double muG;      //  Earth Gravitational Parameter (m^3/s^2)
extern double massG;    //  Vehicle Mass (kg)
```

```
#endif




// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_gov.hpp
// CGPOPS Tool Box
// Declare governing equations of continouos time optimal control problem here
// RLV entry problem


#ifndef __CGPOPS_GOV_HPP__
#define __CGPOPS_GOV_HPP__


#include "nlpGlobVarExt.hpp"
#include "cgpopsFuncDec.hpp"
#include "cgpopsAuxExt.hpp"


// $#$#$

/*---------------------------------EndPointFunction---------------------------------*/

class MaxCrossRange : public ObjectiveEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

/*---------------------------------PhasePointFunction---------------------------------*/

class Altitude : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
```

```
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class Rho : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class DynPres : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class DragCoef : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class LiftCoef : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
```

```
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class DragForce : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class LiftForce : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class Gravity : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class RadDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
```

```
};

class LonDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class LatDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class VelDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class FpaDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};
```

```
class AziDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};


#endif


// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_gov.cpp
// CGPOPS Tool Box
// Define governing equations of continouos time optimal control problem here
// RLV entry problem


#include "cgpops_gov.hpp"

#define alt     p[0]
#define rho     p[1]
#define dynpres p[2]
#define cd      p[3]
#define cl      p[4]
#define drag    p[5]
#define lift    p[6]
#define grav    p[7]

#define rad     x[0]
#define lon     x[1]
#define lat     x[2]
#define vel     x[3]
#define fpa     x[4]
#define azi     x[5]

#define aoa     u[0]
#define bank    u[1]

#define latf    xf[0][2]

template <class T> void MaxCrossRange::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
                                              T** tf, T* s, T* e)
{
    lhs = -latf;
}
```

```
template <class T> void Altitude::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = rad - ReG;
}
template <class T> void Rho::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = rho0G*exp(-alt/HG);
}
template <class T> void DynPres::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs =  0.5*rho*vel*vel;
}
template <class T> void DragCoef::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = cd0G+cd1G*aoa+cd2G*aoa*aoa;
}
template <class T> void LiftCoef::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = cl0G+cl1G*aoa;
}
template <class T> void DragForce::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = dynpres*SG*cd/massG;
}
template <class T> void LiftForce::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = dynpres*SG*cl/massG;
}
template <class T> void Gravity::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = muG/(rad*rad);
}
template <class T> void RadDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = vel*sin(fpa);
}
template <class T> void LonDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = vel*cos(fpa)*sin(azi)/(rad*cos(lat));
}
template <class T> void LatDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = vel*cos(fpa)*cos(azi)/rad;
}
template <class T> void VelDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = -drag-grav*sin(fpa);
}
template <class T> void FpaDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = (lift*cos(bank)-cos(fpa)*(grav-vel*vel/rad))/vel;
}
template <class T> void AziDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
```

```
{
    lhs = (lift*sin(bank)/cos(fpa)+vel*vel*cos(fpa)*sin(azi)*tan(lat)/rad)/vel;
}

void MaxCrossRange::eval_eq(double& lhs, double** x0, double** xf, double** q,
                               double** t0, double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MaxCrossRange::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                               HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MaxCrossRange::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                               Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void Altitude::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void Altitude::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Altitude::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void Rho::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void Rho::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                    HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Rho::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                    Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void DynPres::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void DynPres::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                        HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void DynPres::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                        Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void DragCoef::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void DragCoef::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void DragCoef::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void LiftCoef::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void LiftCoef::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void LiftCoef::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
```

```
void DragForce::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                        double* p)
{eq_def(lhs,x,u,t,s,p);}
void DragForce::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                        HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void DragForce::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                        Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void LiftForce::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                        double* p)
{eq_def(lhs,x,u,t,s,p);}
void LiftForce::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                        HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void LiftForce::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                        Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void Gravity::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void Gravity::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                       HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Gravity::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                       Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void RadDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void RadDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                      HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void RadDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                      Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void LonDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void LonDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                      HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void LonDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                      Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void LatDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void LatDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                      HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void LatDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                      Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void VelDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void VelDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
```

```
                                  HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void VelDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                     Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void FpaDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void FpaDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                     HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void FpaDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                     Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void AziDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void AziDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                     HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void AziDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                     Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}




// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main.hpp
// CGPOPS Tool Box
// Declare main continouos time optimal control problem here
//


#ifndef __CGPOPS_MAIN_HPP__
#define __CGPOPS_MAIN_HPP__

#include "cgpopsFuncDec.hpp"
#include "nlpGlobVarExt.hpp"
#include "cgpopsAuxExt.hpp"


// $#$#$


void cgpops_go(doubleMat& cgpopsResults);


#endif


// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main_test.cpp
```

```cpp
// Test Main
// Test CGPOPS functions
//

#include "cgpops_main.hpp"


void cgpops_go(doubleMat& cgpopsResults)
{
    // Define Global Variables used to determine problem size
    PG      = 1;    // Number of phases in problem
    nsG     = 0;    // Number of static parameters in problem
    nbG     = 0;    // Number of event constraints in problem
    nepG    = 0;    // Number of endpoint parameters in problem

    // Allocate memory for each phase
    initGlobalVars();

    // Define number of components for each parameter in problem phases
    // Phase 1 parameters
    nxG[0]  = 6;    // Number of state components in phase 1
    nuG[0]  = 2;    // Number of control components in phase 1
    nqG[0]  = 0;    // Number of integral constraints in phase 1
    ncG[0]  = 0;    // Number of path constraints in phase 1
    nppG[0] = 8;    // Number of phase parameters in phase 2

//   setGlobalTabularData(); // Set any global tabular data used in problem

    // Define mesh grids for each phase in problem for LGR collocation
    // Phase 1 mesh grid
    int M1 = numintervalsG;        // Number of intervals used in phase 1 mesh
    int initcolpts = initcolptsG;  // Initial number of collocation points in each
                                   // interval
    double fraction1[M1];          // Allocate memory for fraction vector for phase 1
                                   // mesh
    int colpoints1[M1];            // Allocate memory for colpoints vector for phase 1
                                   // mesh
    for (int m=0; m<M1; m++)
    {
        fraction1[m] = 1.0/((double) M1);
        colpoints1[m] = initcolpts;
    }
    setRPMDG(0,M1,fraction1,colpoints1);

    // Set information for transcribed NLP resulting from LGR collocation using defined
    // mesh grid
    setInfoNLPG();

    /*----------------------------Provide Problem Bounds----------------------------*/
    // Phase 1
    double t0 = 0;                          // initial time value
    double alt0 = 79248;
    double rad0 = alt0+ReG;
```

```
    double altf   = 24384;
    double radf   = altf+ReG;
    double lon0   = 0;
    double lat0   = 0;
    double speed0 = 7802.88;
    double speedf = 762;
    double fpa0   = -1*M_PI/180;
    double fpaf   = -5*M_PI/180;
    double azi0   = 90*M_PI/180;
    double azif   = -90*M_PI/180;
    double tfmin = 0,                tfmax = 3000;         // final time minimum and
                                                          // maximum values
    double radmin = ReG,             radmax = rad0;        // minimum and maximum rad
                                                          // state component
    double lonmin = -M_PI,           lonmax = M_PI;        // minimum and maximum lon
                                                          // state component
    double latmin = -70*M_PI/180,    latmax = 70*M_PI/180; // minimum and maximum lat
                                                          // state component
    double speedmin = 10,            speedmax = 45000;     // minimum and maximum y
                                                          // state component
    double fpamin = -80*M_PI/180,    fpamax = 80*M_PI/180; // minimum and maximum x
                                                          // state component
    double azimin = -M_PI,           azimax = M_PI;        // minimum and maximum y
                                                          // state component
    double aoamin = -M_PI/2,         aoamax = M_PI/2;      // minimum and maximum u1
                                                          // control component
    double bankmin = -M_PI/2,        bankmax = 1*M_PI/180; // minimum and maximum u1
                                                          // control component

    // Throughout

    // Define Bounds of Optimal Control Problem
    // Phase 1 bounds
    int phase1 = 0;
    double x0l1[nxG[phase1]],   x0u1[nxG[phase1]];
    double xfl1[nxG[phase1]],   xfu1[nxG[phase1]];
    double xl1[nxG[phase1]],    xu1[nxG[phase1]];
    double ul1[nuG[phase1]],    uu1[nuG[phase1]];
    double ql1[nqG[phase1]],    qu1[nqG[phase1]];
    double cl1[ncG[phase1]],    cu1[ncG[phase1]];
    double t0l1,    t0u1;
    double tfl1,    tfu1;
    x0l1[0] = rad0;
    x0l1[1] = lon0;
    x0l1[2] = lat0;
    x0l1[3] = speed0;
    x0l1[4] = fpa0;
    x0l1[5] = azi0;
    x0u1[0] = rad0;
    x0u1[1] = lon0;
    x0u1[2] = lat0;
    x0u1[3] = speed0;
    x0u1[4] = fpa0;
    x0u1[5] = azi0;
```

```
    xfl1[0] = radf;
    xfl1[1] = lonmin;
    xfl1[2] = latmin;
    xfl1[3] = speedf;
    xfl1[4] = fpaf;
    xfl1[5] = azimin;
    xfu1[0] = radf;
    xfu1[1] = lonmax;
    xfu1[2] = latmax;
    xfu1[3] = speedf;
    xfu1[4] = fpaf;
    xfu1[5] = azimax;
    xl1[0]  = radmin;
    xl1[1]  = lonmin;
    xl1[2]  = latmin;
    xl1[3]  = speedmin;
    xl1[4]  = fpamin;
    xl1[5]  = azimin;
    xu1[0]  = radmax;
    xu1[1]  = lonmax;
    xu1[2]  = latmax;
    xu1[3]  = speedmax;
    xu1[4]  = fpamax;
    xu1[5]  = azimax;
    ul1[0]  = aoamin;
    ul1[1]  = bankmin;
    uu1[0]  = aoamax;
    uu1[1]  = bankmax;
    t0l1    = t0;
    t0u1    = t0;
    tfl1    = tfmin;
    tfu1    = tfmax;

    // Set parameterized constructor for NLP Phase Bounds Class
    setNLPPBG(phase1,x0l1,x0u1,xfl1,xfu1,xl1,xu1,ul1,uu1,ql1,qu1,cl1,cu1,t0l1,t0u1,tfl1,
              tfu1);

    // Whole problem bounds
    double sl[nsG], su[nsG];
    double bl[nbG], bu[nbG];

    // Set parameterized constructor for NLP Whole Bounds Class
    setNLPWBG(sl,su,bl,bu);

    // Provide initial guess for NLP Solver
    // Phase 1 guess
    double x0g1[nxG[phase1]],   xfg1[nxG[phase1]];
    double u0g1[nuG[phase1]],   ufg1[nuG[phase1]];
    double qg1[nqG[phase1]];
    double t0g1,    tfg1;
    x0g1[0] = rad0;
    x0g1[1] = lon0;
    x0g1[2] = lat0;
```

```
    x0g1[3] = speed0;
    x0g1[4] = fpa0;
    x0g1[5] = azi0;
    xfg1[0] = radf;
    xfg1[1] = lon0+10*M_PI/180;
    xfg1[2] = lat0+10*M_PI/180;
    xfg1[3] = speedf;
    xfg1[4] = fpaf;
    xfg1[5] = azif;
    u0g1[0] = 0;
    u0g1[1] = 0;
    ufg1[0] = 0;
    ufg1[1] = 0;
    t0g1    = 0;
    tfg1    = 1000;

    // Set parameterized constructor for NLP Phase Guess Class
    setNLPPGG(phase1,x0g1,xfg1,u0g1,ufg1,qg1,t0g1,tfg1);

    // Whole problem guess
    double sg[nsG];

    // Set parameterized constructor for NLP Phase Guess Class
    setNLPWGG(sg);

    // Set ocp functions
    objEqG = new MaxCrossRange;
    pppEqVecG = {{new Altitude, new Rho, new DynPres, new DragCoef, new LiftCoef,
        new DragForce, new LiftForce, new Gravity}};
    odeEqVecG = {{new RadDot, new LonDot, new LatDot, new VelDot, new FpaDot,
        new AziDot}};

    // Make call to CGOPS handler for IPOPT using user provided output settings
    CGPOPS_IPOPT_caller(cgpopsResults);

}



// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main_test.cpp
// Test Main
// Test CGPOPS functions
//

#include "nlpGlobVarDec.hpp"
#include "cgpopsAuxDec.hpp"
#include "cgpops_main.hpp"


int main(void)
{
    printf("\nCGPOPS TESTING\n\n\n");
```

```
    int numTestRuns = 1;
    int numDS = 1;
    int DSSelect = 0;
    doubleMatMat cgpopsResultsMatMat(numTestRuns*numDS);
    doubleMat cgpopsResults;

    // Derivative supplier settings
    derivativeSupplierG = 0;    // Derivative supplier (default=0)
    scaledG             = 1;    // Scaling flag (default 1=on)

    // Mesh initialization settings
    numintervalsG   = 10;   // Initial number of mesh intervals per phase (default=10)
    initcolptsG     = 4;    // Initial number of collocation points per interval
                            // (default=4)

    // Mesh refinement settings
    meshRefineTypeG = 1;    // Select mesh refinement technique to be used (default=1)
    minColPtsG      = 4;    // Mininum number of collocation points used in an interval
                            // (default=4)
    maxColPtsG      = 10;   // Maximum number of collocation points used in an interval
                            // (default=10)
    maxMeshIterG    = 30;   // Maximum number of mesh iterations (default=20)
    meshTolG        = 1e-7; // Mesh tolerance (default=1e-7)

    // Output save settings
    saveIPOPTFlagG      = 1;   // Save IPOPT solution (default=1)
    saveMeshRefineFlagG = 1;   // Save mesh refinement history (default=0)
    saveHamiltonianG    = 0;   // Save Hamiltonian values (default=0)
    saveLTIHG           = 0;   // Save linear terms in Hamiltonian values (default=0)

    // IPOPT settings
    runIPOPTFlagG   = 1;    // Run IPOPT (default=1)
    NLPtolG         = 1e-7; // NLP Solver tolerance (default=1e-7)
    NLPmaxiterG     = 3000; // Maximum number of iterations allowed for NLP solver
    useLTIHDDG = 1;

    /*-----------------------Changes to global parameter settings----------------------*/

    for (int ds=0; ds<numDS; ds++)
    {
        derivativeSupplierG = DSSelect;
        if (derivativeSupplierG==DSSelect)
        {
            for (int ni=0; ni<numTestRuns; ni++)
            {
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns] = getDoubleMat(6);
                printf("\nDerivativeSupplier = %d",derivativeSupplierG);
                printf("\nNumIntervals = %d",numintervalsG);
                cgpops_go(cgpopsResults);
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[0] = derivativeSupplierG;
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[1] = numintervalsG;
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[2] = cgpopsResults.val[0];
```

```
              cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[3] = cgpopsResults.val[1];
              cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[4] = cgpopsResults.val[2];
              cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[5] = cgpopsResults.val[3];
          }
      }
  }

  printf4MSCRIPT("cgpopsResultsMatMat",cgpopsResultsMatMat);
  printf("\n\n\n\n\n");

  return 0;
}
```

This example was solved using $\mathbb{CGPOPS}$ using the NLP solver IPOPT with a mesh refinement tolerance of 10^{-6} and the solution is shown in Figs. 4a–4f.
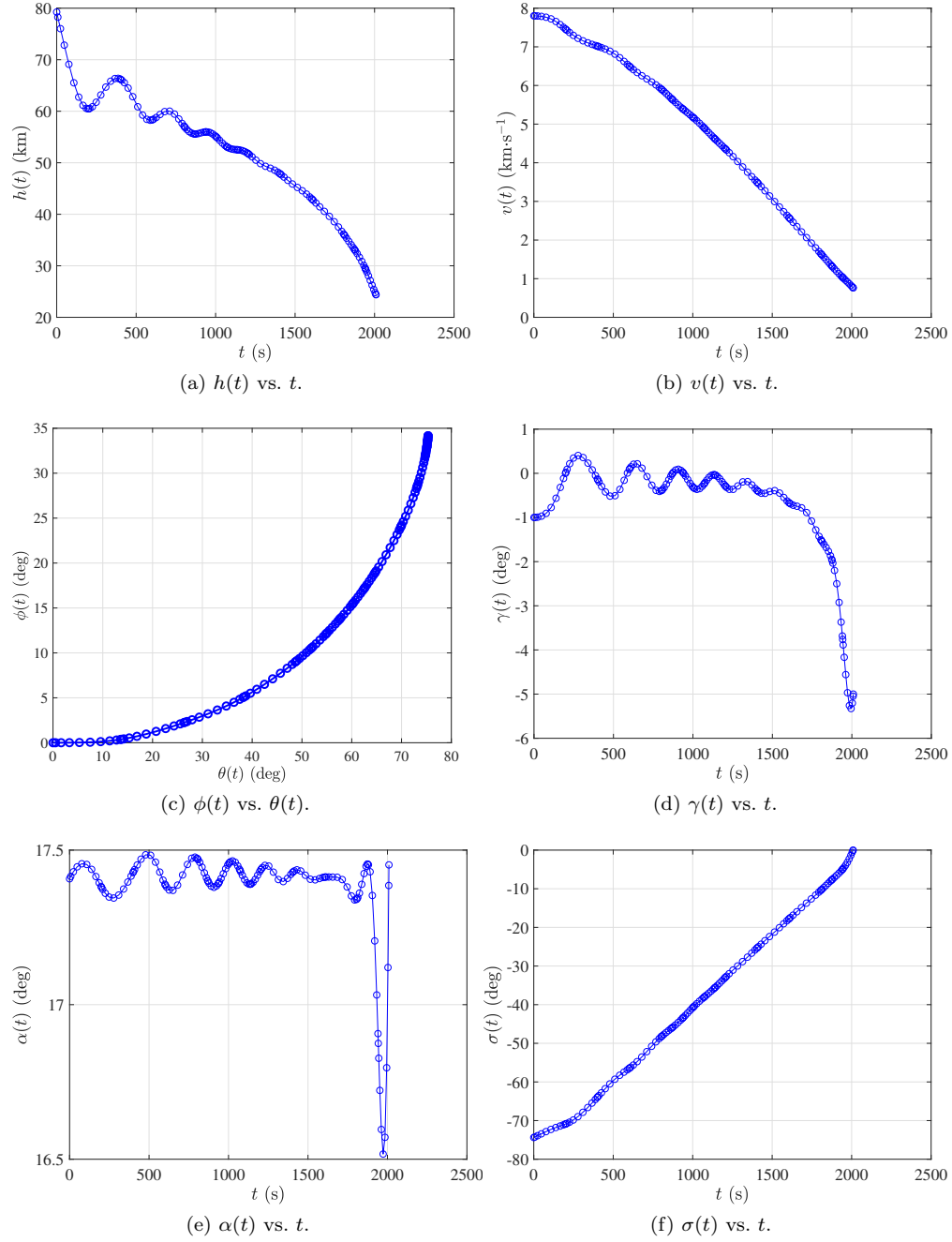
Figure 4: Solution to Reusable Launch Vehicle Entry Problem Using $\mathbb{CGPOPS}$ with the NLP Solver IPOPT using the Hyper-Dual Derivative Supplier and a Mesh Refinement Tolerance of 10^{-6}.

3.5   Minimum Time-to-Climb of a Supersonic Aircraft

The problem considered in this section is the classical minimum time-to-climb of a supersonic aircraft. The objective is to determine the minimum-time trajectory and control from take-off to a specified altitude and speed. This problem was originally stated in the open literature in the work of Ref. 27, but the model used in this study was taken from Ref. 2 with the exception that a linear extrapolation of the thrust data as found in Ref. 2 was performed in order to fill in the "missing" data points.

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

$$J = t_f \; , \tag{44}$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{h} &= v \sin\alpha \; , \\
\dot{v} &= \frac{T\cos\alpha - D}{m} \; , \\
\dot{\gamma} &= \frac{T\sin\alpha + L}{mv} + \left(\frac{v}{r} - \frac{\mu}{vr^2}\right)\cos\gamma \; , \\
\dot{m} &= -\frac{T}{g_0 I_{sp}} \; ,
\end{aligned}
\tag{45}
$$

and the boundary conditions

$$
\begin{aligned}
h(0) &= 0 \text{ ft} \; , \\
v(0) &= 129.3144 \text{ m/s} \; , \\
\gamma(0) &= 0 \text{ rad} \; , \\
h(t_f) &= 19994.88 \text{ m} \; , \\
v(t_f) &= 295.092 \text{ ft/s} \; , \\
\gamma(t_f) &= 0 \text{ rad} \; ,
\end{aligned}
\tag{46}
$$

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. 27. In this example $\mathbb{CGPOPS}$ is implemented using the bicomplex-step derivative supplier (that is, using the option **derivativeSupplierG**=1) together with an interpolation of the table data with the C++ functions 'EvaluatePiecewisePolynomial', 'EvaluateCubicInterpNAKModel', and 'BicubicInterpolationWrapper' defined in the $\mathbb{CGPOPS}$ functions library. The C++ code that solves the minimum time-to-climb of a supersonic aircraft is shown below.

```
// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpopsAuxDec.hpp
// CGPOPS NLP Variables
// First declaration or global variables used in CGPOPS for NLP
//

#ifndef __CGPOPS_AUX_DEC_HPP__
#define __CGPOPS_AUX_DEC_HPP__

#include "cgpopsClassDef.hpp"


// Minimum Time to Climb problem parameters
int US1976LenG = 45;     // Length of US 1976 tabular data
// Altitude values for 1976 U.S. Standard Atmosphere data
double altitudeUS1976G[] = {-2000, 0, 2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000,
    18000, 20000, 22000, 24000, 26000, 28000, 30000, 32000, 34000, 36000, 38000, 40000,
    42000, 44000, 46000, 48000, 50000, 52000, 54000, 56000, 58000, 60000, 62000, 64000,
    66000, 68000, 70000, 72000, 74000, 76000, 78000, 80000, 82000, 84000, 86000};
```

```
// Air density values for 1976 U.S. Standard Atmosphere data
double densityUS1976G[] = {1.478e+00, 1.225e+00, 1.007e+00, 8.193e-01, 6.601e-01,
    5.258e-01, 4.135e-01, 3.119e-01, 2.279e-01, 1.665e-01, 1.216e-01, 8.891e-02,
    6.451e-02, 4.694e-02, 3.426e-02, 2.508e-02, 1.841e-02, 1.355e-02, 9.887e-03,
    7.257e-03, 5.366e-03, 3.995e-03, 2.995e-03, 2.259e-03, 1.714e-03, 1.317e-03,
    1.027e-03, 8.055e-04, 6.389e-04, 5.044e-04, 3.962e-04, 3.096e-04, 2.407e-04,
    1.860e-04, 1.429e-04, 1.091e-04, 8.281e-05, 6.236e-05, 4.637e-05, 3.430e-05,
    2.523e-05, 1.845e-05, 1.341e-05, 9.690e-06, 6.955e-06};
// Speed of sound values for 1976 U.S. Standard Atmosphere data
double sosUS1976G[] = {3.479e+02, 3.403e+02, 3.325e+02, 3.246e+02, 3.165e+02, 3.081e+02,
    2.995e+02, 2.951e+02, 2.951e+02, 2.951e+02, 2.951e+02, 2.951e+02, 2.964e+02,
    2.977e+02, 2.991e+02, 3.004e+02, 3.017e+02, 3.030e+02, 3.065e+02, 3.101e+02,
    3.137e+02, 3.172e+02, 3.207e+02, 3.241e+02, 3.275e+02, 3.298e+02, 3.298e+02,
    3.288e+02, 3.254e+02, 3.220e+02, 3.186e+02, 3.151e+02, 3.115e+02, 3.080e+02,
    3.044e+02, 3.007e+02, 2.971e+02, 2.934e+02, 2.907e+02, 2.880e+02, 2.853e+02,
    2.825e+02, 2.797e+02, 2.769e+02, 2.741e+02};
doubleMat MtabG(10);     // Mach speeds, xvals for thrust table
doubleMat alttabG(10);   // Altitude in meters, yvals for thrust table
doubleMatMat TtabG(10);  // Thrust values in Newtons, doublemat[xind].val[yind]
doubleMatMat TtabXG(10);    // Thrust value derivatives wrt Mach,
                            // doublemat[xind].val[yind]
doubleMatMat TtabYG(10);    // Thrust value derivatives wrt altitude,
                            // doublemat[xind].val[yind]
doubleMatMat TtabXYG(10);   // Thrust value derivatives wrt Mach and altitude,
                            // doublemat[xind].val[yind]
doubleMat CLdatBreaksG(7);  // Break points for polynomial approximation for computing
                            // coefficient of lift in dynamics
doubleMat CDdatBreaksG(7);  // Break points for polynomial approximation for computing
                            // coefficient of drag in dynamics
doubleMat etadatBreaksG(7); // Break points for polynomial approximation for computing
                            // eta term in coefficient of drag in dynamics
doubleMatMat CLdatCoefsG(6);    // Coefficients of polynomial approximation for computing
                                // coefficient of lift in dynamics
doubleMatMat CDdatCoefsG(6);    // Coefficients of polynomial approximation for computing
                                // coefficient of drag in dynamics
doubleMatMat etadatCoefsG(6);   // Coefficients of polynomial approximation for computing
                                // eta term in coefficient of drag equation in dynamics
double ReG      = 6378145;  // Radius of earth [m]
double muG      = 3.986e14; // Gravitational parameter [m^3/s^2]
double SG       = 49.2386;  // Surface area [m^2]
double g0G      = 9.80665;  // Gravitational acceleration [m/s^2]
double IspG     = 1600;     // Specific thrusts]
// Cubic interpolating model for density tabular data
cubicInterpNAKModel rhoCubicInterpG;
// Cubic interpolating model for sos tabular data
cubicInterpNAKModel sosCubicInterpG;




#endif
```

```
// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpopsAuxExt.hpp
// CGPOPS Auxiliary Variables
// External declaration of global variables used in CGPOPS for NLP
//

#ifndef __CGPOPS_AUX_EXT_HPP__
#define __CGPOPS_AUX_EXT_HPP__


// Minimum Time to Climb problem parameters
extern int US1976LenG;                        // Length of US 1976 tabular data
// Altitude values for 1976 U.S. Standard Atmosphere data
extern double altitudeUS1976G[];
// Air density values for 1976 U.S. Standard Atmosphere data
extern double densityUS1976G[];
// Speed of sound values for 1976 U.S. Standard Atmosphere data
extern double sosUS1976G[];
extern doubleMat MtabG; // Mach speeds, xvals for thrust table
extern doubleMat alttabG;   // Altitude in meters, yvals for thrust table
extern doubleMatMat TtabG;  // Thrust values in Newtons, doublemat[xind].val[yind]
extern doubleMatMat TtabXG; // Thrust value derivatives wrt Mach,
                            // doublemat[xind].val[yind]
extern doubleMatMat TtabYG; // Thrust value derivatives wrt altitude,
                            // doublemat[xind].val[yind]
extern doubleMatMat TtabXYG;    // Thrust value derivatives wrt Mach and altitude,
                                // doublemat[xind].val[yind]
extern doubleMat CLdatBreaksG;  // Break points for polynomial approximation for
                                // computing coefficient of lift in dynamics
extern doubleMat CDdatBreaksG;  // Break points for polynomial approximation for
                                // computing coefficient of drag in dynamics
extern doubleMat etadatBreaksG; // Break points for polynomial approximation for
                                // computing eta term in coefficient of drag in dynamics
extern doubleMatMat CLdatCoefsG;    // Coefficients of polynomial approximation for
                                    // computing coefficient of lift in dynamics
extern doubleMatMat CDdatCoefsG;    // Coefficients of polynomial approximation for
                                    // computing coefficient of drag in dynamics
extern doubleMatMat etadatCoefsG;   // Coefficients of polynomial approximation for
                                    // computing eta term in coefficient of drag equation
                                    // in dynamics
extern double ReG;  // Radius of earth [m]
extern double muG;  // Gravitational parameter [m^3/s^2]
extern double SG;    // Surface area [m^2]
extern double g0G;  // Gravitational acceleration [m/s^2]
extern double IspG; // Specific thrusts]
// Cubic interpolating model for density tabular data
extern cubicInterpNAKModel rhoCubicInterpG;
// Cubic interpolating model for sos tabular data
extern cubicInterpNAKModel sosCubicInterpG;
```

```
#endif




// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_gov.hpp
// CGPOPS Tool Box
// Declare governing equations of continouos time optimal control problem here
// Minimum time-to-climb problem


#ifndef __CGPOPS_GOV_HPP__
#define __CGPOPS_GOV_HPP__


#include "nlpGlobVarExt.hpp"
#include "cgpopsFuncDec.hpp"
#include "cgpopsAuxExt.hpp"


// $#$#$

/*--------------------------------EndPointFunction--------------------------------*/

class MinTf : public ObjectiveEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

/*-------------------------------PhasePointFunction-------------------------------*/

class Radius : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
```

```
                                Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class MachSpeed : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class LiftDragCoef : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class Thrust : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class DynamicPressure : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
```

```
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class HDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class VDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class FpaDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class MDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
```

```
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

void setGlobalTabularData(void); // Set global tabular data used in problem

#endif


// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_gov.cpp
// CGPOPS Tool Box
// Define governing equations of continouos time optimal control problem here
// Minimum time-to-climb problem


#include "cgpops_gov.hpp"

#define r       p[0]
#define mach    p[1]
#define clalpha p[2]
#define thrust  p[3]
#define dynpres p[4]
#define h       x[0]
#define v       x[1]
#define fpa     x[2]
#define m       x[3]
#define alpha   u[0]
#define tf_     tf[0][0]

template <class T> void MinTf::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf,
                                      T* s, T* e)
{
    lhs = tf_;
}
template <class T> void Radius::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = h+ReG;
}
template <class T> void MachSpeed::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    T sos = EvaluateCubicInterpNAKModel(h,sosCubicInterpG);
    lhs = v/sos;
}
template <class T> void LiftDragCoef::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    // Clalpha computation
    if (mach<0.8)   // If Mach speed parameter < 0.8
        lhs = 3.44;
    else
        lhs = EvaluatePiecewisePolynomial(mach,CLdatBreaksG,CLdatCoefsG);

}
```

```
template <class T> void Thrust::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = BicubicInterpolationWrapper(mach,h,MtabG,alttabG,TtabG,TtabXG,TtabYG,TtabXYG);
}
template <class T> void DynamicPressure::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    T rho = EvaluateCubicInterpNAKModel(h,rhoCubicInterpG);
    lhs = 0.5*rho*v*v;  // q = 0.5.*rho.*v.*v;
}
template <class T> void HDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = v*sin(fpa);
}
template <class T> void VDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    T cd0,eta,cd,d;
    if (mach<0.8)    // If Mach speed parameter < 0.8
    {
        cd0 = 0.013;
        eta = 0.54;
    }
    else
    {
        cd0 = EvaluatePiecewisePolynomial(mach,CDdatBreaksG,CDdatCoefsG);
        eta = EvaluatePiecewisePolynomial(mach,etadatBreaksG,etadatCoefsG);
    }
    cd = cd0 + eta*clalpha*alpha*alpha; // CD = CD0 + eta.*Clalpha.*alpha.^2;
    d = dynpres*SG*cd;  // D = q.*S.*CD;
    lhs = (thrust*cos(alpha)-d)/m - muG*sin(fpa)/(r*r);
}
template <class T> void FpaDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    T cl = clalpha*alpha;    // CL = Clalpha.*alpha;
    T l = dynpres*SG*cl;     // L = q.*S.*CL;
    lhs = (thrust*sin(alpha)+l)/(m*v)+cos(fpa)*(v/r-muG/(v*r*r));
}
template <class T> void MDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = -thrust/(g0G*IspG);
}

void MinTf::eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                    double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinTf::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                    HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinTf::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                    Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void Radius::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void Radius::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
```

```
                            HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Radius::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                     Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void MachSpeed::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                          double* p)
{eq_def(lhs,x,u,t,s,p);}
void MachSpeed::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                          HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void MachSpeed::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                          Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void LiftDragCoef::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                             double* p)
{eq_def(lhs,x,u,t,s,p);}
void LiftDragCoef::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                             HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void LiftDragCoef::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                             Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void Thrust::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void Thrust::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                       HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Thrust::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                       Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void DynamicPressure::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                                double* p)
{eq_def(lhs,x,u,t,s,p);}
void DynamicPressure::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                                HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void DynamicPressure::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                                Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void HDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void HDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                     HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void HDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                     Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void VDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void VDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                     HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
```

```
void VDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                   Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void FpaDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void FpaDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                     HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void FpaDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                     Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void MDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void MDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                   HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void MDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                   Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}


void setGlobalTabularData(void) // Set global tabular data used in problem
{
    MtabG.val[0] = 0;
    MtabG.val[1] = 0.2;
    MtabG.val[2] = 0.4;
    MtabG.val[3] = 0.6;
    MtabG.val[4] = 0.8;
    MtabG.val[5] = 1;
    MtabG.val[6] = 1.2;
    MtabG.val[7] = 1.4;
    MtabG.val[8] = 1.6;
    MtabG.val[9] = 1.8;

    alttabG.val[0] = 304.8*0;
    alttabG.val[1] = 304.8*5;
    alttabG.val[2] = 304.8*10;
    alttabG.val[3] = 304.8*15;
    alttabG.val[4] = 304.8*20;
    alttabG.val[5] = 304.8*25;
    alttabG.val[6] = 304.8*30;
    alttabG.val[7] = 304.8*40;
    alttabG.val[8] = 304.8*50;
    alttabG.val[9] = 304.8*70;

    TtabG.mat[0] = getDoubleMat(10);
    TtabG.mat[1] = getDoubleMat(10);
    TtabG.mat[2] = getDoubleMat(10);
    TtabG.mat[3] = getDoubleMat(10);
    TtabG.mat[4] = getDoubleMat(10);
    TtabG.mat[5] = getDoubleMat(10);
    TtabG.mat[6] = getDoubleMat(10);
    TtabG.mat[7] = getDoubleMat(10);
```

```
    TtabG.mat[8] = getDoubleMat(10);
    TtabG.mat[9] = getDoubleMat(10);

    TtabG.mat[0].val[0]  = 1.076469724000000e5;
    TtabG.mat[0].val[1]  = 1.067573280000000e5;
    TtabG.mat[0].val[2]  = 0.902989066000000e5;
    TtabG.mat[0].val[3]  = 0.769542406000000e5;
    TtabG.mat[0].val[4]  = 0.644992190000000e5;
    TtabG.mat[0].val[5]  = 0.542683084000000e5;
    TtabG.mat[0].val[6]  = 0.453718644000000e5;
    TtabG.mat[0].val[7]  = 0.253548654000000e5;
    TtabG.mat[0].val[8]  = 0.151239548000000e5;
    TtabG.mat[0].val[9]  = 0.004448222000000e5;

    TtabG.mat[1].val[0]  = 1.245502160000000e5;
    TtabG.mat[1].val[1]  = 1.094262612000000e5;
    TtabG.mat[1].val[2]  = 0.938574842000000e5;
    TtabG.mat[1].val[3]  = 0.805128182000000e5;
    TtabG.mat[1].val[4]  = 0.676129744000000e5;
    TtabG.mat[1].val[5]  = 0.569372416000000e5;
    TtabG.mat[1].val[6]  = 0.475959754000000e5;
    TtabG.mat[1].val[7]  = 0.289134430000000e5;
    TtabG.mat[1].val[8]  = 0.173480658000000e5;
    TtabG.mat[1].val[9]  = 0.008896444000000e5;

    TtabG.mat[2].val[0]  = 1.258846826000000e5;
    TtabG.mat[2].val[1]  = 1.120951944000000e5;
    TtabG.mat[2].val[2]  = 0.974160618000000e5;
    TtabG.mat[2].val[3]  = 0.831817514000000e5;
    TtabG.mat[2].val[4]  = 0.707267298000000e5;
    TtabG.mat[2].val[5]  = 0.596061748000000e5;
    TtabG.mat[2].val[6]  = 0.498200864000000e5;
    TtabG.mat[2].val[7]  = 0.324720206000000e5;
    TtabG.mat[2].val[8]  = 0.195721768000000e5;
    TtabG.mat[2].val[9]  = 0.017792888000000e5;

    TtabG.mat[3].val[0]  = 1.370052376000000e5;
    TtabG.mat[3].val[1]  = 1.209916384000000e5;
    TtabG.mat[3].val[2]  = 1.058676836000000e5;
    TtabG.mat[3].val[3]  = 0.911885510000000e5;
    TtabG.mat[3].val[4]  = 0.769542406000000e5;
    TtabG.mat[3].val[5]  = 0.653888634000000e5;
    TtabG.mat[3].val[6]  = 0.547131306000000e5;
    TtabG.mat[3].val[7]  = 0.360305982000000e5;
    TtabG.mat[3].val[8]  = 0.217962878000000e5;
    TtabG.mat[3].val[9]  = 0.035585776000000e5;

    TtabG.mat[4].val[0]  = 1.534636590000000e5;
    TtabG.mat[4].val[1]  = 1.347811266000000e5;
    TtabG.mat[4].val[2]  = 1.183227052000000e5;
    TtabG.mat[4].val[3]  = 1.031987504000000e5;
    TtabG.mat[4].val[4]  = 0.880747956000000e5;
    TtabG.mat[4].val[5]  = 0.747301296000000e5;
```

```
    TtabG.mat[4].val[6]  = 0.627199302000000e5;
    TtabG.mat[4].val[7]  = 0.418132868000000e5;
    TtabG.mat[4].val[8]  = 0.249100432000000e5;
    TtabG.mat[4].val[9]  = 0.048930442000000e5;

    TtabG.mat[5].val[0]  = 1.685876138000000e5;
    TtabG.mat[5].val[1]  = 1.525740146000000e5;
    TtabG.mat[5].val[2]  = 1.352259488000000e5;
    TtabG.mat[5].val[3]  = 1.192123496000000e5;
    TtabG.mat[5].val[4]  = 1.036435726000000e5;
    TtabG.mat[5].val[5]  = 0.880747956000000e5;
    TtabG.mat[5].val[6]  = 0.747301296000000e5;
    TtabG.mat[5].val[7]  = 0.498200864000000e5;
    TtabG.mat[5].val[8]  = 0.302479096000000e5;
    TtabG.mat[5].val[9]  = 0.062275108000000e5;

    TtabG.mat[6].val[0]  = 1.605808142000000e5;
    TtabG.mat[6].val[1]  = 1.690324360000000e5;
    TtabG.mat[6].val[2]  = 1.552429478000000e5;
    TtabG.mat[6].val[3]  = 1.392293486000000e5;
    TtabG.mat[6].val[4]  = 1.214364606000000e5;
    TtabG.mat[6].val[5]  = 1.049780392000000e5;
    TtabG.mat[6].val[6]  = 0.894092622000000e5;
    TtabG.mat[6].val[7]  = 0.596061748000000e5;
    TtabG.mat[6].val[8]  = 0.369202426000000e5;
    TtabG.mat[6].val[9]  = 0.075619774000000e5;

    TtabG.mat[7].val[0]  = 1.605808142000000e5;
    TtabG.mat[7].val[1]  = 1.628049252000000e5;
    TtabG.mat[7].val[2]  = 1.712565470000000e5;
    TtabG.mat[7].val[3]  = 1.605808142000000e5;
    TtabG.mat[7].val[4]  = 1.405638152000000e5;
    TtabG.mat[7].val[5]  = 1.249950382000000e5;
    TtabG.mat[7].val[6]  = 1.076469724000000e5;
    TtabG.mat[7].val[7]  = 0.720611964000000e5;
    TtabG.mat[7].val[8]  = 0.444822200000000e5;
    TtabG.mat[7].val[9]  = 0.097860884000000e5;

    TtabG.mat[8].val[0]  = 1.605808142000000e5;
    TtabG.mat[8].val[1]  = 1.565774144000000e5;
    TtabG.mat[8].val[2]  = 1.872701462000000e5;
    TtabG.mat[8].val[3]  = 1.721461914000000e5;
    TtabG.mat[8].val[4]  = 1.588015254000000e5;
    TtabG.mat[8].val[5]  = 1.423431040000000e5;
    TtabG.mat[8].val[6]  = 1.249950382000000e5;
    TtabG.mat[8].val[7]  = 0.858506846000000e5;
    TtabG.mat[8].val[8]  = 0.529338418000000e5;
    TtabG.mat[8].val[9]  = 0.128998438000000e5;

    TtabG.mat[9].val[0]  = 1.605808142000000e5;
    TtabG.mat[9].val[1]  = 1.503499036000000e5;
    TtabG.mat[9].val[2]  = 2.032837454000000e5;
    TtabG.mat[9].val[3]  = 1.837115686000000e5;
```

```
    TtabG.mat[9].val[4]  = 1.770392356000000e5;
    TtabG.mat[9].val[5]  = 1.539084812000000e5;
    TtabG.mat[9].val[6]  = 1.383397042000000e5;
    TtabG.mat[9].val[7]  = 0.965264174000000e5;
    TtabG.mat[9].val[8]  = 0.591613526000000e5;
    TtabG.mat[9].val[9]  = 0.137894882000000e5;

    CLdatBreaksG.val[0] = 0.8;
    CLdatBreaksG.val[1] = 0.9;
    CLdatBreaksG.val[2] = 1;
    CLdatBreaksG.val[3] = 1.2;
    CLdatBreaksG.val[4] = 1.4;
    CLdatBreaksG.val[5] = 1.6;
    CLdatBreaksG.val[6] = 1.8;

    CDdatBreaksG.val[0] = 0.8;
    CDdatBreaksG.val[1] = 0.9;
    CDdatBreaksG.val[2] = 1;
    CDdatBreaksG.val[3] = 1.2;
    CDdatBreaksG.val[4] = 1.4;
    CDdatBreaksG.val[5] = 1.6;
    CDdatBreaksG.val[6] = 1.8;

    etadatBreaksG.val[0] = 0.8;
    etadatBreaksG.val[1] = 0.9;
    etadatBreaksG.val[2] = 1;
    etadatBreaksG.val[3] = 1.2;
    etadatBreaksG.val[4] = 1.4;
    etadatBreaksG.val[5] = 1.6;
    etadatBreaksG.val[6] = 1.8;

    CLdatCoefsG.mat[0]  = getDoubleMat(4);
    CLdatCoefsG.mat[1]  = getDoubleMat(4);
    CLdatCoefsG.mat[2]  = getDoubleMat(4);
    CLdatCoefsG.mat[3]  = getDoubleMat(4);
    CLdatCoefsG.mat[4]  = getDoubleMat(4);
    CLdatCoefsG.mat[5]  = getDoubleMat(4);

    CDdatCoefsG.mat[0]  = getDoubleMat(4);
    CDdatCoefsG.mat[1]  = getDoubleMat(4);
    CDdatCoefsG.mat[2]  = getDoubleMat(4);
    CDdatCoefsG.mat[3]  = getDoubleMat(4);
    CDdatCoefsG.mat[4]  = getDoubleMat(4);
    CDdatCoefsG.mat[5]  = getDoubleMat(4);

    etadatCoefsG.mat[0]  = getDoubleMat(4);
    etadatCoefsG.mat[1]  = getDoubleMat(4);
    etadatCoefsG.mat[2]  = getDoubleMat(4);
    etadatCoefsG.mat[3]  = getDoubleMat(4);
    etadatCoefsG.mat[4]  = getDoubleMat(4);
    etadatCoefsG.mat[5]  = getDoubleMat(4);

    CLdatCoefsG.mat[0].val[0]  = 3.417576791808878e2;
```

```
    CLdatCoefsG.mat[0].val[1] = -0.201757679180887e2;
    CLdatCoefsG.mat[0].val[2] = 0;
    CLdatCoefsG.mat[0].val[3] = 0.034400000000000e2;

    CLdatCoefsG.mat[1].val[0] = -5.852730375426626e2;
    CLdatCoefsG.mat[1].val[1] = 0.823515358361775e2;
    CLdatCoefsG.mat[1].val[2] = 0.062175767918089e2;
    CLdatCoefsG.mat[1].val[3] = 0.035800000000000e2;

    CLdatCoefsG.mat[2].val[0] = 2.129095563139934e2;
    CLdatCoefsG.mat[2].val[1] = -0.932303754266212e2;
    CLdatCoefsG.mat[2].val[2] = 0.051296928327645e2;
    CLdatCoefsG.mat[2].val[3] = 0.044400000000000e2;

    CLdatCoefsG.mat[3].val[0] = -0.609940273037543e2;
    CLdatCoefsG.mat[3].val[1] = 0.345153583617748e2;
    CLdatCoefsG.mat[3].val[2] = -0.066133105802048e2;
    CLdatCoefsG.mat[3].val[3] = 0.034400000000000e2;

    CLdatCoefsG.mat[4].val[0] = -0.051834470989761e2;
    CLdatCoefsG.mat[4].val[1] = -0.020810580204778e2;
    CLdatCoefsG.mat[4].val[2] = -0.001264505119454e2;
    CLdatCoefsG.mat[4].val[3] = 0.030100000000000e2;

    CLdatCoefsG.mat[5].val[0] = 0.129778156996587e2;
    CLdatCoefsG.mat[5].val[1] = -0.051911262798635e2;
    CLdatCoefsG.mat[5].val[2] = -0.015808873720137e2;
    CLdatCoefsG.mat[5].val[3] = 0.028600000000000e2;

    CDdatCoefsG.mat[0].val[0] = 7.374146757679185;
    CDdatCoefsG.mat[0].val[1] = -0.637414675767918;
    CDdatCoefsG.mat[0].val[2] = 0;
    CDdatCoefsG.mat[0].val[3] = 0.013000000000000;

    CDdatCoefsG.mat[1].val[0] = -8.122440273037544;
    CDdatCoefsG.mat[1].val[1] = 1.574829351535836;
    CDdatCoefsG.mat[1].val[2] = 0.093741467576792;
    CDdatCoefsG.mat[1].val[3] = 0.014000000000000;

    CDdatCoefsG.mat[2].val[0] = 1.433660409556315;
    CDdatCoefsG.mat[2].val[1] = -0.861902730375427;
    CDdatCoefsG.mat[2].val[2] = 0.165034129692833;
    CDdatCoefsG.mat[2].val[3] = 0.031000000000000;

    CDdatCoefsG.mat[3].val[0] = -0.049274744027304;
    CDdatCoefsG.mat[3].val[1] = -0.001706484641638;
    CDdatCoefsG.mat[3].val[2] = -0.007687713310580;
    CDdatCoefsG.mat[3].val[3] = 0.041000000000000;

    CDdatCoefsG.mat[4].val[0] = 0.138438566552901;
    CDdatCoefsG.mat[4].val[1] = -0.031271331058021;
    CDdatCoefsG.mat[4].val[2] = -0.014283276450512;
    CDdatCoefsG.mat[4].val[3] = 0.039000000000000;
```

```
    CDdatCoefsG.mat[5].val[0] = -0.129479522184301;
    CDdatCoefsG.mat[5].val[1] = 0.051791808873720;
    CDdatCoefsG.mat[5].val[2] = -0.010179180887372;
    CDdatCoefsG.mat[5].val[3] = 0.036000000000000;

    etadatCoefsG.mat[0].val[0] = -2.241296928327646e2;
    etadatCoefsG.mat[0].val[1] = 0.434129692832765e2;
    etadatCoefsG.mat[0].val[2] = 0;
    etadatCoefsG.mat[0].val[3] = 0.005400000000000e2;

    etadatCoefsG.mat[1].val[0] = 0.823890784982935e2;
    etadatCoefsG.mat[1].val[1] = -0.238259385665529e2;
    etadatCoefsG.mat[1].val[2] = 0.019587030716724e2;
    etadatCoefsG.mat[1].val[3] = 0.007500000000000e2;

    etadatCoefsG.mat[2].val[0] = 0.026663822525597e2;
    etadatCoefsG.mat[2].val[1] = 0.008907849829352e2;
    etadatCoefsG.mat[2].val[2] = -0.003348122866894e2;
    etadatCoefsG.mat[2].val[3] = 0.007900000000000e2;

    etadatCoefsG.mat[3].val[0] = -0.072397610921502e2;
    etadatCoefsG.mat[3].val[1] = 0.024906143344710e2;
    etadatCoefsG.mat[3].val[2] = 0.003414675767918e2;
    etadatCoefsG.mat[3].val[3] = 0.007800000000000e2;

    etadatCoefsG.mat[4].val[0] = 0.025426621160410e2;
    etadatCoefsG.mat[4].val[1] = -0.018532423208191e2;
    etadatCoefsG.mat[4].val[2] = 0.004689419795222e2;
    etadatCoefsG.mat[4].val[3] = 0.008900000000000e2;

    etadatCoefsG.mat[5].val[0] = 0.008191126279863e2;
    etadatCoefsG.mat[5].val[1] = -0.003276450511945e2;
    etadatCoefsG.mat[5].val[2] = 0.000327645051195e2;
    etadatCoefsG.mat[5].val[3] = 0.009300000000000e2;

    TtabXG  = TabularDataXPartialNAK(MtabG,TtabG);
    TtabYG  = TabularDataYPartialNAK(alttabG,TtabG);
    TtabXYG = TabularDataXPartialNAK(MtabG,TtabYG);
    rhoCubicInterpG = ComputeCubicInterpNAKModel(US1976LenG, altitudeUS1976G,
                                                 densityUS1976G);
    sosCubicInterpG = ComputeCubicInterpNAKModel(US1976LenG,altitudeUS1976G,sosUS1976G);
}




// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main.hpp
// CGPOPS Tool Box
// Declare main continouos time optimal control problem here
//
```

```cpp
#ifndef __CGPOPS_MAIN_HPP__
#define __CGPOPS_MAIN_HPP__

#include "cgpopsFuncDec.hpp"
#include "nlpGlobVarExt.hpp"
#include "cgpopsAuxExt.hpp"


// $#$#$


void cgpops_go(doubleMat& cgpopsResults);


#endif


// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main_test.cpp
// Test Main
// Test CGPOPS functions
//

#include "cgpops_main.hpp"


void cgpops_go(doubleMat& cgpopsResults)
{
    // Define Global Variables used to determine problem size
    PG      = 1;    // Number of phases in problem
    nsG     = 0;    // Number of static parameters in problem
    nbG     = 0;    // Number of event constraints in problem
    nepG    = 0;    // Number of endpoint parameters in problem

    // Allocate memory for each phase
    initGlobalVars();

    // Define number of components for each parameter in problem phases
    // Phase 1 parameters
    nxG[0]  = 4;    // Number of state components in phase 1
    nuG[0]  = 1;    // Number of control components in phase 1
    nqG[0]  = 0;    // Number of integral constraints in phase 1
    ncG[0]  = 0;    // Number of path constraints in phase 1
    nppG[0] = 5;    // Number of phase parameters in phase 1

    setGlobalTabularData(); // Set any global tabular data used in problem

    // Define mesh grids for each phase in problem for LGR collocation
    // Phase 1 mesh grid
    int M1 = numintervalsG;        // Number of intervals used in phase 1 mesh
    int initcolpts = initcolptsG;  // Initial number of collocation points in each
                                   // interval
```

```
    double fraction1[M1];            // Allocate memory for fraction vector for phase 1
                                     // mesh
    int colpoints1[M1];             // Allocate memory for colpoints vector for phase 1
                                     // mesh
    for (int m=0; m<M1; m++)
    {
        fraction1[m] = 1.0/((double) M1);
        colpoints1[m] = initcolpts;
    }
    setRPMDG(0,M1,fraction1,colpoints1);

    // Set information for transcribed NLP resulting from LGR collocation using defined
    // mesh grid
    setInfoNLPG();

    /*---------------------------Provide Problem Bounds---------------------------*/
    // Phase 1
    double t0min = 0,             t0max = 0;              // minimum and maximum
                                                         // initial times
    double tfmin = 100,          tfmax = 800;            // minimum and maximum final
                                                         // times
    double alt0 = 0,             altf = 19994.88;        // initial and final values
                                                         // of altitude state component
    double speed0 = 129.314,     speedf = 295.092;       // initial and final values
                                                         // of speed state component
    double fpa0 = 0,             fpaf = 0;               // initial and final values
                                                         // of flight path angle state
                                                         // component
    double mass0 = 19050.864;                            // initial value of mass
                                                         // state component
    double altmin = 0,           altmax = 21031.2;       // minimum and maximum
                                                         // altitude state component
    double speedmin = 5,         speedmax = 1000;        // minimum and maximum speed
                                                         // state component
    double fpamin = -40*M_PI/180, fpamax = 40*M_PI/180;  // minimum and maximum flight
                                                         // path angle state component
    double massmin = 22,         massmax = 20410;        // minimum and maximum mass
                                                         // state component
    double alphamin = -M_PI/4,   alphamax = M_PI/4;      // minimum and maximum alpha
                                                         // control component


    // Throughout

    // Define Bounds of Optimal Control Problem
    // Phase 1 bounds
    int phase1 = 0;
    double x0l1[nxG[phase1]],   x0u1[nxG[phase1]];
    double xfl1[nxG[phase1]],   xfu1[nxG[phase1]];
    double xl1[nxG[phase1]],    xu1[nxG[phase1]];
    double ul1[nuG[phase1]],    uu1[nuG[phase1]];
    double ql1[nqG[phase1]],    qu1[nqG[phase1]];
    double cl1[ncG[phase1]],    cu1[ncG[phase1]];
    double t0l1,    t0u1;
```

```
    double tfl1,    tfu1;
   x0l1[0] = alt0;
   x0l1[1] = speed0;
   x0l1[2] = fpa0;
   x0l1[3] = mass0;
   x0u1[0] = alt0;
   x0u1[1] = speed0;
   x0u1[2] = fpa0;
   x0u1[3] = mass0;
   xfl1[0] = altf;
   xfl1[1] = speedf;
   xfl1[2] = fpaf;
   xfl1[3] = massmin;
   xfu1[0] = altf;
   xfu1[1] = speedf;
   xfu1[2] = fpaf;
   xfu1[3] = massmax;
   xl1[0]  = altmin;
   xl1[1]  = speedmin;
   xl1[2]  = fpamin;
   xl1[3]  = massmin;
   xu1[0]  = altmax;
   xu1[1]  = speedmax;
   xu1[2]  = fpamax;
   xu1[3]  = massmax;
   ul1[0]  = alphamin;
   uu1[0]  = alphamax;
   t0l1    = t0min;
   t0u1    = t0max;
   tfl1    = tfmin;
   tfu1    = tfmax;

   // Set parameterized constructor for NLP Phase Bounds Class
   setNLPPBG(phase1,x0l1,x0u1,xfl1,xfu1,xl1,xu1,ul1,uu1,ql1,qu1,cl1,cu1,t0l1,t0u1,tfl1,
             tfu1);

   // Whole problem bounds
   double sl[nsG], su[nsG];
   double bl[nbG], bu[nbG];

   // Set parameterized constructor for NLP Whole Bounds Class
   setNLPWBG(sl,su,bl,bu);

   // Provide initial guess for NLP Solver
   // Phase 1 guess
   double x0g1[nxG[phase1]],   xfg1[nxG[phase1]];
   double u0g1[nuG[phase1]],   ufg1[nuG[phase1]];
   double qg1[nqG[phase1]];
   double t0g1,    tfg1;
   x0g1[0] = alt0;
   x0g1[1] = speed0;
   x0g1[2] = fpa0;
   x0g1[3] = mass0;
```

```
    xfg1[0] = altf;
    xfg1[1] = speedf;
    xfg1[2] = fpaf;
    xfg1[3] = mass0;
    u0g1[0] = 20*M_PI/180;
    ufg1[0] = -20*M_PI/180;
    t0g1    = 0;
    tfg1    = 1000;

    // Set parameterized constructor for NLP Phase Guess Class
    setNLPPGG(phase1,x0g1,xfg1,u0g1,ufg1,qg1,t0g1,tfg1);

    // Whole problem guess
    double sg[nsG];

    // Set parameterized constructor for NLP Phase Guess Class
    setNLPWGG(sg);

    // Set ocp functions
    objEqG = new MinTf;
    pppEqVecG = {{new Radius, new MachSpeed, new LiftDragCoef, new Thrust, new DynamicPressure}};
    odeEqVecG = {{new HDot, new VDot, new FpaDot, new MDot}};

    // Make call to CGOPS handler for IPOPT using user provided output settings
    CGPOPS_IPOPT_caller(cgpopsResults);

}


// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main_test.cpp
// Test Main
// Test CGPOPS functions
//

#include "nlpGlobVarDec.hpp"
#include "cgpopsAuxDec.hpp"
#include "cgpops_main.hpp"


int main(void)
{
    printf("\nCGPOPS TESTING\n\n\n");

    int numTestRuns = 1;
    int numDS = 1;
    int DSSelect = 0;
    doubleMatMat cgpopsResultsMatMat(numTestRuns*numDS);
    doubleMat cgpopsResults;

    // Derivative supplier settings
    derivativeSupplierG = 0;     // Derivative supplier (default=0)
```

```
    scaledG                 = 1;     // Scaling flag (default 1=on)

    // Mesh initialization settings
    numintervalsG   = 10;   // Initial number of mesh intervals per phase (default=10)
    initcolptsG     = 5;    // Initial number of collocation points per interval
                            // (default=4)

    // Mesh refinement settings
    meshRefineTypeG = 1;    // Select mesh refinement technique to be used (default=1)
    minColPtsG      = 4;    // Mininum number of collocation points used in an interval
                            // (default=4)
    maxColPtsG      = 8;    // Maximum number of collocation points used in an interval
                            // (default=10)
    maxMeshIterG    = 15;   // Maximum number of mesh iterations (default=20)
    meshTolG        = 1e-7; // Mesh tolerance (default=1e-7)

    // Output save settings
    saveIPOPTFlagG      = 0;   // Save IPOPT solution (default=1)
    saveMeshRefineFlagG = 0;   // Save mesh refinement history (default=0)
    saveHamiltonianG    = 0;   // Save Hamiltonian values (default=0)
    saveLTIHG           = 0;   // Save linear terms in Hamiltonian values (default=0)

    // IPOPT settings
    runIPOPTFlagG   = 1;    // Run IPOPT (default=1)
    NLPtolG         = 1e-7; // NLP Solver tolerance (default=1e-7)
    NLPmaxiterG     = 3000; // Maximum number of iterations allowed for NLP solver
    useLTIHDDG = 1;

    /*-----------------------Changes to global parameter settings----------------------*/

    for (int ds=0; ds<numDS; ds++)
    {
//        derivativeSupplierG = DSSelect;
//        if (derivativeSupplierG==DSSelect)
        derivativeSupplierG = ds;
        if (derivativeSupplierG==ds)
        {
            for (int ni=0; ni<numTestRuns; ni++)
            {
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns] = getDoubleMat(6);
                printf("\nDerivativeSupplier = %d",derivativeSupplierG);
//                  numintervalsG = pow(2,ni+1);
                printf("\nNumIntervals = %d",numintervalsG);
                cgpops_go(cgpopsResults);
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[0] = derivativeSupplierG;
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[1] = numintervalsG;
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[2] = cgpopsResults.val[0];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[3] = cgpopsResults.val[1];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[4] = cgpopsResults.val[2];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[5] = cgpopsResults.val[3];
            }
        }
    }
```

```
//    printf4MSCRIPT("cgpopsResultsMatMat",cgpopsResultsMatMat);
      printf("\n\n\n\n\n");

      return 0;
}
```

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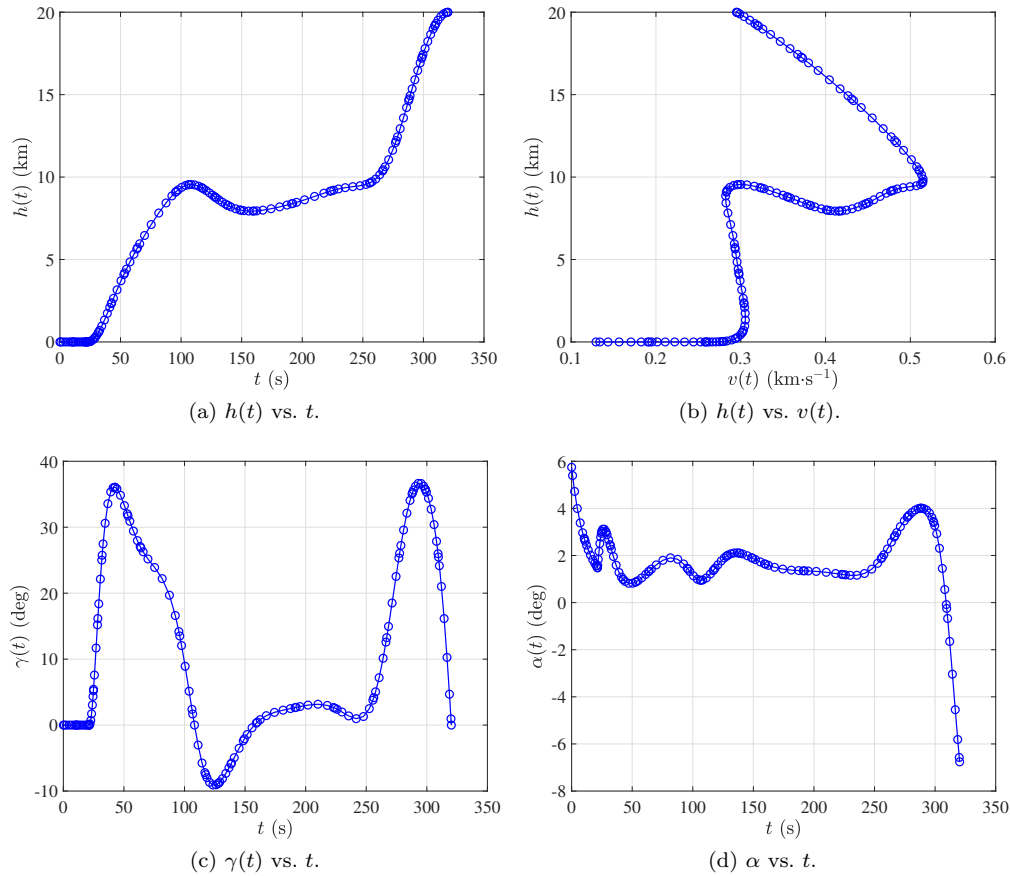

(a) $h(t)$ vs. t.

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

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

(d) α vs. t.

Figure 5: Solution to Minimum Time-to-Climb Problem Using $\mathbb{CGPOPS}$ with the NLP Solver IPOPT using the Bicomplex-step Derivative Supplier and a Mesh Refinement Tolerance of 10^{-6}.
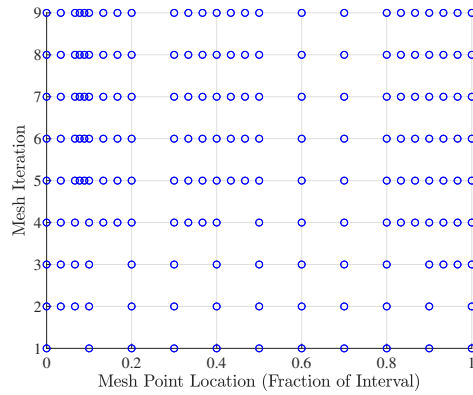
Figure 6: Mesh Refinement to Minimum Time-to-Climb Problem Using $\mathbb{CGPOPS}$ with the NLP Solver IPOPT using the Bicomplex-step Derivative Supplier and a Mesh Refinement Tolerance of 10^{-6}.

Table 9: Relative Error Estimate vs. Mesh Refinement Iteration for Minimum Time-to-Climb Problem.

Mesh Refinement Iteration	Relative Error Estimate
1	4.9246×10^{-3}
2	2.0176×10^{-4}
3	2.6130×10^{-5}
4	2.0945×10^{-5}
5	7.5472×10^{-6}
6	1.8581×10^{-6}
7	1.9957×10^{-6}
8	1.1712×10^{-6}
9	9.2040×10^{-7}

3.6   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. 20 and the problem considered here is identical to that of Ref. 20. The objective is to minimize the average wind gradient slope β, that is, minimize

$$J = \beta , \tag{47}$$

subject to the hang glider dynamics

$$
\begin{aligned}
\dot{x} &= v \cos \gamma \sin \psi + W_x , \\
m\dot{v} &= -D - mg \sin \gamma - m\dot{W}_x \cos \gamma \sin \psi , \\
\dot{y} &= v \cos \gamma \cos \psi , \\
mv\dot{\gamma} &= L \cos \sigma - mg \cos \gamma + m\dot{W}_x \sin \gamma \sin \psi , \\
\dot{h} &= v \sin \gamma , \\
mv \cos \gamma \dot{\psi} &= L \sin \sigma - m\dot{W}_x \cos \psi ,
\end{aligned} \tag{48}
$$

and the boundary conditions

$$
\begin{aligned}
(x(0), y(0), h(0)) &= (x(t_f), y(t_f), h(t_f)) = (0, 0, 0) , \\
(v(t_f) - v(0), \gamma(t_f) - \gamma(0), \psi(t_f) + 2\pi - \psi(0)) &= (0, 0, 0) ,
\end{aligned} \tag{49}
$$

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

$$
\begin{aligned}
D &= qSC_D , \\
L &= qSC_L ,
\end{aligned} \tag{50}
$$

where $q = \rho v^2 / 2$ is the dynamic pressure, S is the vehicle reference area, $C_D = C_{D0} + K C_L^2$ 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. 20 and are given as $C_{D0} = 0.00873$, $K = 0.045$, and $C_{L,\max} = 1.5$. Finally, it is noted that C_L and σ are the controls.

This example was posed in English units, but was solved using the automatic scaling procedure in $\mathbb{CGPOPS}$ with the NLP solver IPOPT using the central finite-differencing derivative supplier 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. 7.

```
// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.   All Rights Reserved
//
// cgpopsAuxDec.hpp
// CGPOPS NLP Variables
// First declaration or global variables used in CGPOPS for NLP
//

#ifndef __CGPOPS_AUX_DEC_HPP__
#define __CGPOPS_AUX_DEC_HPP__


// Dynamic Soaring problem parameters
double rhoG  = 0.002378;
double CD0G  = 0.00873;
double KG    = 0.045;
double gG    = 32.2;
```

```
double mG    = 5.6;
double SG    = 45.09703;
double muG   = 3.986e14;
double mgosG = mG*gG/SG;
double EmaxG = sqrt(1.0/(4.0*KG*CD0G));
double W0G   = 0;
double lminG = -2;
double lmaxG = 5;



#endif
```

```
// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpopsAuxExt.hpp
// CGPOPS Auxiliary Variables
// External declaration of global variables used in CGPOPS for NLP
//

#ifndef __CGPOPS_AUX_EXT_HPP__
#define __CGPOPS_AUX_EXT_HPP__


// Dynamic Soaring problem parameters
extern double rhoG;
extern double CD0G;
extern double KG;
extern double gG;
extern double mG;
extern double SG;
extern double muG;
extern double mgosG;
extern double EmaxG;
extern double W0G;
extern double lminG;
extern double lmaxG;


#endif
```

```cpp
// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_gov.hpp
// CGPOPS Tool Box
// Declare governing equations of continouos time optimal control problem here
// Dynamic soaring problem


#ifndef __CGPOPS_GOV_HPP__
#define __CGPOPS_GOV_HPP__


#include "nlpGlobVarExt.hpp"
#include "cgpopsFuncDec.hpp"
#include "cgpopsAuxExt.hpp"


// $#$#$

/*---------------------------------EndPointFunction---------------------------------*/

class MinParameter : public ObjectiveEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

class VLink : public EventConstraintEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

class GammaLink : public EventConstraintEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
```

```
                                   double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

class PsiLink : public EventConstraintEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                         double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                         HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                         Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                   T* e);
};

/*--------------------------------PhasePointFunction--------------------------------*/

class Term1 : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class Term2 : public PhasePointParameterEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class Term3 : public PhasePointParameterEq
```

```
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};


class XDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};


class YDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};


class ZDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};


class VDot : public OrdinaryDifferentialEq
{
```

```
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class GammaDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class PsiDot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class GConstraint : public PathConstraintEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

void setGlobalTabularData(void); // Set global tabular data used in problem
```

```
#endif


// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_gov.cpp
// CGPOPS Tool Box
// Define governing equations of continouos time optimal control problem here
// Dynamic soaring problem


#include "cgpops_gov.hpp"

#define term1   p[0]
#define term2   p[1]
#define term3   p[2]
#define x_s     x[0]
#define y       x[1]
#define z       x[2]
#define v       x[3]
#define gamma   x[4]
#define psi     x[5]
#define cl      u[0]
#define phi     u[1]
#define beta    s[0]
#define v0      x0[0][3]
#define gamma0  x0[0][4]
#define psi0    x0[0][5]
#define vf      xf[0][3]
#define gammaf  xf[0][4]
#define psif    xf[0][5]

template <class T> void MinParameter::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
                                             T** tf, T* s, T* e)
{
    lhs = 7*beta;
}
template <class T> void VLink::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf,
                                      T* s, T* e)
{
    lhs = vf-v0;
}
template <class T> void GammaLink::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf,
                                          T* s, T* e)
{
    lhs = gammaf-gamma0;
}
template <class T> void PsiLink::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf,
                                        T* s, T* e)
{
    lhs = psif-psi0;
}
template <class T> void Term1::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
```

```
{
    lhs = 0.5*rhoG*SG/mG;
}
template <class T> void Term2::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = 1;
}
template <class T> void Term3::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = gG*term2;
}
template <class T> void XDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = v*cos(gamma)*sin(psi)+beta*z+W0G;
}
template <class T> void YDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = v*cos(gamma)*cos(psi);
}
template <class T> void ZDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = v*sin(gamma);
}
template <class T> void VDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = -term1*(CD0G+KG*cl*cl)*v*v-term3*sin(gamma)
        -term2*beta*v*sin(gamma)*sin(psi)*cos(gamma);
}
template <class T> void GammaDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = term1*cl*v*cos(phi)-term3*cos(gamma)/v
        +term2*beta*v*sin(gamma)*sin(psi)*sin(gamma)/v;
}
template <class T> void PsiDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = (term1*cl*v*sin(phi)-term2*beta*v*sin(gamma)*cos(psi)/v)/cos(gamma);
}
template <class T> void GConstraint::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = term1/gG*cl*v*v;
}


void MinParameter::eval_eq(double& lhs, double** x0, double** xf, double** q,
                           double** t0, double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinParameter::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                           HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinParameter::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                           Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void VLink::eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                    double** tf, double* s, double* e)
```

```
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void VLink::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                    HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void VLink::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                    Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void GammaLink::eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                        double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void GammaLink::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                        HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void GammaLink::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                        Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void PsiLink::eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                      double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void PsiLink::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                      HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void PsiLink::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                      Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void Term1::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void Term1::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                    HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Term1::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s, Bicomplex* p
void Term2::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void Term2::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                    HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Term2::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                    Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void Term3::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void Term3::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                    HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void Term3::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                    Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void XDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void XDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                   HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void XDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
```

```
                              Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void YDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void YDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                   HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void YDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                   Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void ZDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void ZDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                   HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void ZDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                   Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void VDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void VDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                   HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void VDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                   Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void GammaDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void GammaDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                       HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void GammaDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                       Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void PsiDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void PsiDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                     HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void PsiDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                     Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void GConstraint::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                          double* p)
{eq_def(lhs,x,u,t,s,p);}
void GConstraint::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                          HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void GConstraint::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                          Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}


void setGlobalTabularData(void) // Set global tabular data used in problem
```

```
{
    if (userScalarsG==1)
    {
        pathScalarsG    = getDoubleMatMat(PG);
        eventScalarsG   = getDoubleMat(nbG);
        for (int phase=0; phase<PG; phase++)
            pathScalarsG.mat[phase] = getDoubleMat(ncG[phase]);
        // Provide Scalars for path and event constraints
        pathScalarsG.mat[0].val[0] = 8.596357982165083e-02;
        eventScalarsG.val[0] = 7.071067811862652e-01;
        eventScalarsG.val[1] = 7.071067811862652e-01;
        eventScalarsG.val[2] = 7.071067811862652e-01;
    }
}




// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main.hpp
// CGPOPS Tool Box
// Declare main continouos time optimal control problem here
//

#ifndef __CGPOPS_MAIN_HPP__
#define __CGPOPS_MAIN_HPP__

#include "cgpopsFuncDec.hpp"
#include "nlpGlobVarExt.hpp"
#include "cgpopsAuxExt.hpp"


// $#$#$


void cgpops_go(doubleMat& cgpopsResults);


#endif


// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main_test.cpp
// Test Main
// Test CGPOPS functions
//

#include "cgpops_main.hpp"


void cgpops_go(doubleMat& cgpopsResults)
```

```
{
    // Define Global Variables used to determine problem size
    PG      = 1;    // Number of phases in problem
    nsG     = 1;    // Number of static parameters in problem
    nbG     = 3;    // Number of event constraints in problem
    nepG    = 0;    // Number of endpoint parameters in problem

    // Allocate memory for each phase
    initGlobalVars();

    // Define number of components for each parameter in problem phases
    // Phase 1 parameters
    nxG[0]  = 6;    // Number of state components in phase 1
    nuG[0]  = 2;    // Number of control components in phase 1
    nqG[0]  = 0;    // Number of integral constraints in phase 1
    ncG[0]  = 1;    // Number of path constraints in phase 1
    nppG[0] = 3;    // Number of phase parameters in phase 2

    setGlobalTabularData(); // Set any global tabular data used in problem

    // Define mesh grids for each phase in problem for LGR collocation
    // Phase 1 mesh grid
    int M1 = numintervalsG;         // Number of intervals used in phase 1 mesh
    int initcolpts = initcolptsG;   // Initial number of collocation points in each
                                    // interval
    double fraction1[M1];           // Allocate memory for fraction vector for phase 1
                                    // mesh
    int colpoints1[M1];             // Allocate memory for colpoints vector for phase 1
                                    // mesh
    for (int m=0; m<M1; m++)
    {
        fraction1[m] = 1.0/((double) M1);
        colpoints1[m] = initcolpts;
    }
    setRPMDG(0,M1,fraction1,colpoints1);

    // Set information for transcribed NLP resulting from LGR collocation using defined
    // mesh grid
    setInfoNLPG();

    /*---------------------------Provide Problem Bounds---------------------------*/
    // Phase 1
    double t0min = 0,       t0max = 0;      // minimum and maximum initial times
    double tfmin = 1,       tfmax = 30;     // minimum and maximum final times
    double x0 = 0,          xf = 0;         // initial and final values for x state
                                            // component
    double y0 = 0,          yf = 0;         // initial and final values for y state
                                            // component
    double z0 = 0,          zf = 0;         // initial and final values for z state
                                            // component
    double xmin = -1000,    xmax = 1000;    // minimum and maximum x state component
    double ymin = -1000,    ymax = 1000;    // minimum and maximum y state component
    double zmin = 0,        zmax = 1000;    // minimum and maximum z state component
```

```
   double vmin = 10,        vmax = 350;      // minimum and maximum v state component
   double gammamin = -75.0*M_PI/180.0;       // minimum gamma state component
   double gammamax = 75.0*M_PI/180.0;        // maximum gamma state component
   double psimin = -3.0*M_PI;                // minimum psi state component
   double psimax = M_PI/2.0;                 // maximum psi state component
   double CLmin = -0.5,    CLmax = 1.5;      // minimum and maximum CL control component
   double Phimin = -75.0*M_PI/180.0;         // minimum Phi control component
   double Phimax = 75.0*M_PI/180.0;          // maximum Phi control component
   double cmin = lminG,    cmax = lmaxG;     // minimum and maximum path constraint
                                             // component


   // Throughout
   double betamin = 0.005, betamax = 0.15;   // minimum and maximum beta static parameter
                                             // component


   // Define Bounds of Optimal Control Problem
   // Phase 1 bounds
   int phase1 = 0;
   double x0l1[nxG[phase1]],   x0u1[nxG[phase1]];
   double xfl1[nxG[phase1]],   xfu1[nxG[phase1]];
   double xl1[nxG[phase1]],    xu1[nxG[phase1]];
   double ul1[nuG[phase1]],    uu1[nuG[phase1]];
   double ql1[nqG[phase1]],    qu1[nqG[phase1]];
   double cl1[ncG[phase1]],    cu1[ncG[phase1]];
   double t0l1,    t0u1;
   double tfl1,    tfu1;
   x0l1[0] = x0;
   x0l1[1] = y0;
   x0l1[2] = z0;
   x0l1[3] = vmin;
   x0l1[4] = gammamin;
   x0l1[5] = psimin;
   x0u1[0] = xf;
   x0u1[1] = yf;
   x0u1[2] = zf;
   x0u1[3] = vmax;
   x0u1[4] = gammamax;
   x0u1[5] = psimax;
   xfl1[0] = x0;
   xfl1[1] = y0;
   xfl1[2] = z0;
   xfl1[3] = vmin;
   xfl1[4] = gammamin;
   xfl1[5] = psimin;
   xfu1[0] = x0;
   xfu1[1] = y0;
   xfu1[2] = z0;
   xfu1[3] = vmax;
   xfu1[4] = gammamax;
   xfu1[5] = psimax;
   xl1[0]  = xmin;
   xl1[1]  = ymin;
   xl1[2]  = zmin;
```

```
    xl1[3]   = vmin;
    xl1[4]   = gammamin;
    xl1[5]   = psimin;
    xu1[0]   = xmax;
    xu1[1]   = ymax;
    xu1[2]   = zmax;
    xu1[3]   = vmax;
    xu1[4]   = gammamax;
    xu1[5]   = psimax;
    ul1[0]   = CLmin;
    ul1[1]   = Phimin;
    uu1[0]   = CLmax;
    uu1[1]   = Phimax;
    t0l1     = t0min;
    t0u1     = t0max;
    tfl1     = tfmin;
    tfu1     = tfmax;
    cl1[0]   = cmin;
    cu1[0]   = cmax;


    // Set parameterized constructor for NLP Phase Bounds Class
    setNLPPBG(phase1,x0l1,x0u1,xfl1,xfu1,xl1,xu1,ul1,uu1,ql1,qu1,cl1,cu1,t0l1,t0u1,tfl1,
             tfu1);


    // Whole problem bounds
    double sl[nsG], su[nsG];
    double bl[nbG], bu[nbG];
    sl[0] = betamin;
    su[0] = betamax;
    bl[0] = 0;
    bl[1] = 0;
    bl[2] = -2*M_PI;
    bu[0] = 0;
    bu[1] = 0;
    bu[2] = -2*M_PI;


    // Set parameterized constructor for NLP Whole Bounds Class
    setNLPWBG(sl,su,bl,bu);


    // Provide initial guess for NLP Solver
    // Phase 1 guess
    double x0g1[nxG[phase1]],   xfg1[nxG[phase1]];
    double u0g1[nuG[phase1]],   ufg1[nuG[phase1]];
    double qg1[nqG[phase1]];
    double t0g1,    tfg1;
    x0g1[0] = 0;
    x0g1[1] = 0;
    x0g1[2] = 0;
//    x0g1[2] = 6.931764956787646e-06;
    x0g1[3] = 200;
    x0g1[4] = 0;
    x0g1[5] = -1;
    xfg1[0] = 0;
```

```
    xfg1[1] = 0;
    xfg1[2] = 0;
//    xfg1[2] = 6.931764956787646e-06;
    xfg1[3] = 200;
    xfg1[4] = 0;
    xfg1[5] = -7;
    u0g1[0] = 0.50;
    u0g1[1] = -1;
    ufg1[0] = 0.50;
    ufg1[1] = -1;
    t0g1   = 0;
    tfg1   = 24;

    // Set parameterized constructor for NLP Phase Guess Class
    setNLPPGG(phase1,x0g1,xfg1,u0g1,ufg1,qg1,t0g1,tfg1);

    // Whole problem guess
    double sg[nsG];
    sg[0] = 0.08;

    // Set parameterized constructor for NLP Phase Guess Class
    setNLPWGG(sg);

    // Set ocp functions
    objEqG = new MinParameter;
    eveEqVecG = {new VLink, new GammaLink, new PsiLink};
    pppEqVecG = {{new Term1, new Term2, new Term3}};
    odeEqVecG = {{new XDot, new YDot, new ZDot, new VDot, new GammaDot, new PsiDot}};
    pthEqVecG = {{new GConstraint}};

    // Make call to CGOPS handler for IPOPT using user provided output settings
    CGPOPS_IPOPT_caller(cgpopsResults);

}



// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main_test.cpp
// Test Main
// Test CGPOPS functions
//

#include "nlpGlobVarDec.hpp"
#include "cgpopsAuxDec.hpp"
#include "cgpops_main.hpp"


int main(void)
{
    printf("\nCGPOPS TESTING\n\n\n");

    int numTestRuns = 1;
```

```
    int numDS = 1;
    int DSSelect = 0;
    doubleMatMat cgpopsResultsMatMat(numTestRuns*numDS);
    doubleMat cgpopsResults;

    // Derivative supplier settings
    derivativeSupplierG = 0;     // Derivative supplier (default=0)
    scaledG             = 1;     // Scaling flag (default 1=on)

    // Mesh initialization settings
    numintervalsG   = 10;   // Initial number of mesh intervals per phase (default=10)
    initcolptsG     = 3;    // Initial number of collocation points per interval
                            // (default=4)

    // Mesh refinement settings
    meshRefineTypeG = 1;    // Select mesh refinement technique to be used (default=1)
    minColPtsG      = 3;    // Mininum number of collocation points used in an interval
                            // (default=4)
    maxColPtsG      = 8;    // Maximum number of collocation points used in an interval
                            //(default=10)
    maxMeshIterG    = 30;   // Maximum number of mesh iterations (default=20)
    meshTolG        = 1e-7; // Mesh tolerance (default=1e-7)

    // Output save settings
    saveIPOPTFlagG      = 1;   // Save IPOPT solution (default=1)
    saveMeshRefineFlagG = 1;   // Save mesh refinement history (default=0)
    saveHamiltonianG    = 0;   // Save Hamiltonian values (default=0)
    saveLTIHG           = 0;   // Save linear terms in Hamiltonian values (default=0)

    // IPOPT settings
    runIPOPTFlagG   = 1;    // Run IPOPT (default=1)
    NLPtolG         = 1e-7; // NLP Solver tolerance (default=1e-7)
    NLPmaxiterG     = 3000; // Maximum number of iterations allowed for NLP solver
    useLTIHDDG = 1;

    /*----------------------Changes to global parameter settings----------------------*/

    for (int ds=0; ds<numDS; ds++)
    {
        derivativeSupplierG = DSSelect;
        if (derivativeSupplierG==DSSelect)
        {
            for (int ni=0; ni<numTestRuns; ni++)
            {
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns] = getDoubleMat(6);
                printf("\nDerivativeSupplier = %d",derivativeSupplierG);
                printf("\nNumIntervals = %d",numintervalsG);
                cgpops_go(cgpopsResults);
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[0] = derivativeSupplierG;
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[1] = numintervalsG;
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[2] = cgpopsResults.val[0];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[3] = cgpopsResults.val[1];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[4] = cgpopsResults.val[2];
```

```
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[5] = cgpopsResults.val[3];
            }
        }
    }

    printf4MSCRIPT("cgpopsResultsMatMat",cgpopsResultsMatMat);
    printf("\n\n\n\n\n");

    return 0;
}
```
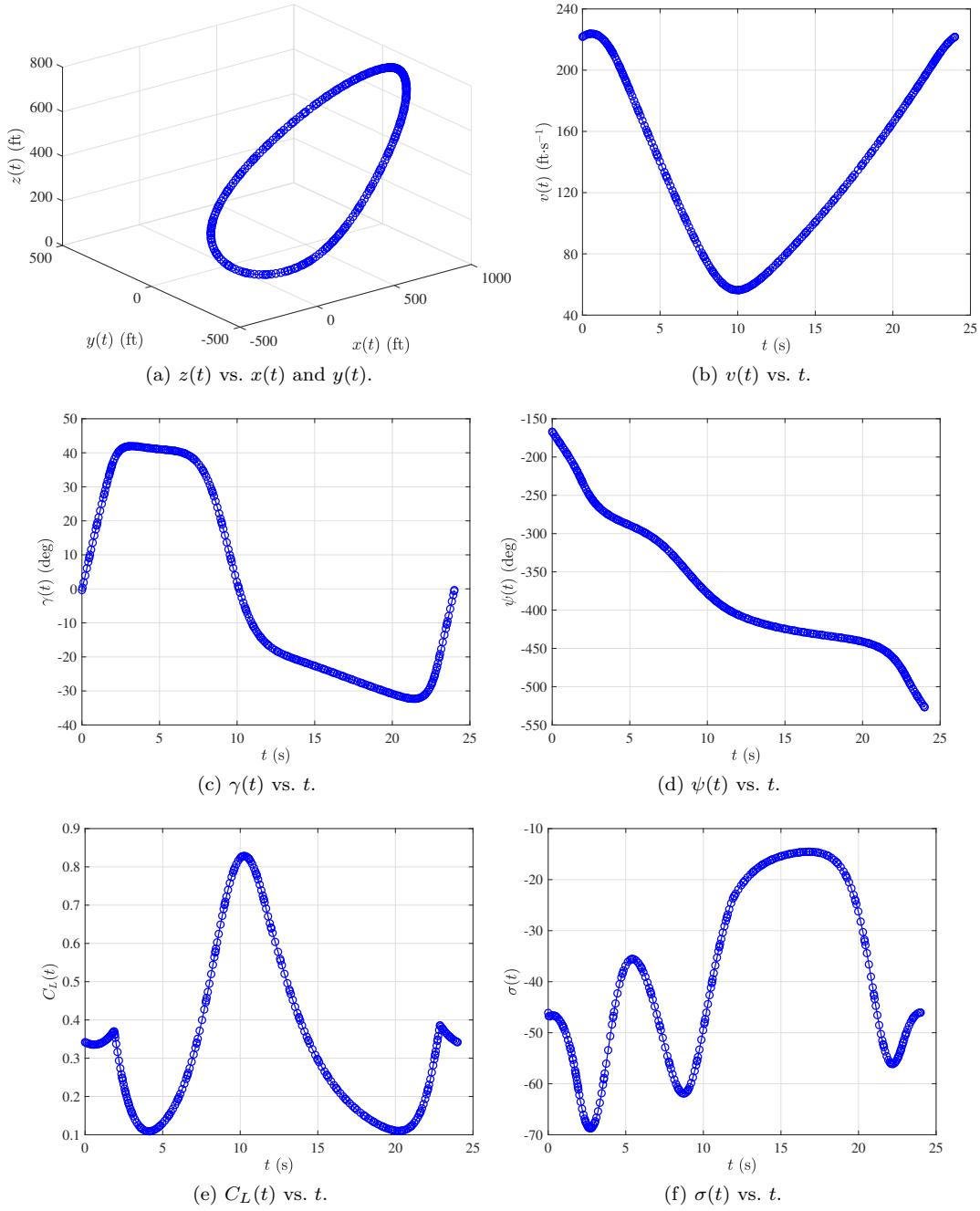
(a) $z(t)$ vs. $x(t)$ and $y(t)$.

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

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

(d) $\psi(t)$ vs. t.

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

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

Figure 7: Solution to Dynamic Soaring Problem Using $\mathbb{GPOPS-II}$ with the NLP Solver IPOPT and a Mesh Refinement Tolerance of 10^{-6}.

3.7   Two-Strain Tuberculosis Optimal Control Problem

Quoting from Ref. 21, "[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. 21 is formulated as follows. Minimize the objective functional

$$J = \int_0^{t_f} \left(L_2 + I_2 + \frac{1}{2}B_1 u_1^2 + B_2 u_2^2 \right) \, dt \ , \tag{51}$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{S}(t) &= \Lambda - \beta_1 S \frac{I_1(t)}{N} - \beta^* S \frac{I_2(t)}{N} - \mu S(t) \ , \\
\dot{L}_1(t) &= \beta_1 S(t) \frac{I_1(t)}{N} - (\mu + k_1) L_1(t) - u_1 r_1 L_1(t) \\
&\quad + (1 - u_2(t)) p r_2 I_1(t) + \beta_2 T(t) \frac{I_1(t)}{N} - \beta^* L_1(t) \frac{I_2(t)}{N} \ , \\
\dot{I}_1(t) &= k_1 L_1(t) - (\mu + d_1) I_1(t) - r_2 I_1(t) \ , \\
\dot{L}_2(t) &= (1 - u_2(t)) q r_2 I_1(t) - (\mu + k_2) L_2(t) \\
&\quad + \beta^* (S(t) + L_1(t) + T(t)) \frac{I_2(t)}{N} \ , \\
\dot{I}_2(t) &= k_2 L_2(t) - (\mu + d_2) I_2(t) \ , \\
\dot{T}(t) &= u_1(t) r_1 L_1(t) - (1 - (1 - u_2(t)))(p + q) r_2 I_1(t) \\
&\quad - \beta_2 T(t) \frac{I_1(t)}{N} - \beta^* T(t) \frac{I_2(t)}{N} - \mu T(t) \ , \\
0 &= S + L_1 + I_1 + L_2 + I_2 + T - N \ ,
\end{aligned}
\tag{52}
$$

and the initial conditions

$$(S(0), L_1(0), I_1(0), L_2(0), I_2(0), T(0)) = (S_0, L_{10}, I_{10}, L_{20}, I_{20}, T_0) \ , \tag{53}$$

where details of the model can be found in Ref. 21 (and are also provided in the $\mathbb{CGPOPS}$ code shown below). The optimal control problem of Eqs. (51)–(53) is solved using $\mathbb{CGPOPS}$ with the NLP solver IPOPT 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. 8 and 9.

```
// cgpopsAuxDec.hpp
// CGPOPS NLP Variables
// First declaration or global variables used in CGPOPS for NLP
// By Yunus Agamawi
// 11/29/18
//

#ifndef __CGPOPS_AUX_DEC_HPP__
#define __CGPOPS_AUX_DEC_HPP__


// Tuberculosis problem parameters
double beta1G = 13;
double beta2G = 13;
double muG = 0.0143;
double d1G = 0;
```

```
double d2G = 0;
double k1G = 0.5;
double k2G = 1;
double r1G = 2;
double r2G = 1;
double pG = 0.4;
double qG = 0.1;
double NpopG = 30000;
double betasG = 0.029;
double B1G = 50;
double B2G = 500;
double lamG = muG*NpopG;
double m0G = 1;
double dmG = 0.0749;


#endif




// cgpopsAuxExt.hpp
// CGPOPS Auxiliary Variables
// External declaration of global variables used in CGPOPS for NLP
// By Yunus Agamawi
// 11/29/18
//

#ifndef __CGPOPS_AUX_EXT_HPP__
#define __CGPOPS_AUX_EXT_HPP__


// Tuberculosis problem parameters
extern double beta1G;
extern double beta2G;
extern double muG;
extern double d1G;
extern double d2G;
extern double k1G;
extern double k2G;
extern double r1G;
extern double r2G;
extern double pG;
extern double qG;
extern double NpopG;
extern double betasG;
extern double B1G;
```

```cpp
extern double B2G;
extern double lamG;
extern double m0G;
extern double dmG;



#endif




// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_gov.hpp
// CGPOPS Tool Box
// Declare governing equations of continouos time optimal control problem here
// Tuberculosis


#ifndef __CGPOPS_GOV_HPP__
#define __CGPOPS_GOV_HPP__



#include "nlpGlobVarExt.hpp"
#include "cgpopsFuncDec.hpp"
#include "cgpopsAuxExt.hpp"


// $#$#$

/*----------------------------------EndPointFunction--------------------------------*/

class MinIntegralCost : public ObjectiveEq
{
public:
    virtual void eval_eq(double& lhs, double** x0, double** xf, double** q, double** t0,
                        double** tf, double* s, double* e);
    virtual void eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf, HyperDual** q,
                        HyperDual** t0, HyperDual** tf, HyperDual* s, HyperDual* e);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf, Bicomplex** q,
                        Bicomplex** t0, Bicomplex** tf, Bicomplex* s, Bicomplex* e);
private:
    template <class T> void eq_def(T& lhs, T** x0, T** xf, T** q, T** t0, T** tf, T* s,
                                    T* e);
};

/*----------------------------------PhasePointFunction--------------------------------*/

class SDot : public OrdinaryDifferentialEq
{
public:
```

```
      virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                           double* p);
      virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                           HyperDual* s, HyperDual* p);
      virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                           Bicomplex* s, Bicomplex* p);
private:
      template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class TDot : public OrdinaryDifferentialEq
{
public:
      virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                           double* p);
      virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                           HyperDual* s, HyperDual* p);
      virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                           Bicomplex* s, Bicomplex* p);
private:
      template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class L1Dot : public OrdinaryDifferentialEq
{
public:
      virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                           double* p);
      virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                           HyperDual* s, HyperDual* p);
      virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                           Bicomplex* s, Bicomplex* p);
private:
      template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class L2Dot : public OrdinaryDifferentialEq
{
public:
      virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                           double* p);
      virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                           HyperDual* s, HyperDual* p);
      virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                           Bicomplex* s, Bicomplex* p);
private:
      template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class I1Dot : public OrdinaryDifferentialEq
{
public:
      virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
```

```
                                     double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class I2Dot : public OrdinaryDifferentialEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class NPopConstraint : public PathConstraintEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

class IntegralCost : public IntegrandEq
{
public:
    virtual void eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                         double* p);
    virtual void eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                         HyperDual* s, HyperDual* p);
    virtual void eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                         Bicomplex* s, Bicomplex* p);
private:
    template <class T> void eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p);
};

#endif


// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
```

```cpp
// cgpops_gov.cpp
// CGPOPS Tool Box
// Define governing equations of continouos time optimal control problem here
// Tuberculosis


#include "cgpops_gov.hpp"

#define S    x[0]
#define T_s  x[1]
#define L1   x[2]
#define L2   x[3]
#define I1   x[4]
#define I2   x[5]


#define u1   u[0]
#define u2   u[1]

template <class T> void MinIntegralCost::eq_def(T& lhs, T** x0, T** xf, T** q, T** t0,
                                                T** tf, T* s, T* e)
{
    lhs = q[0][0];
}
template <class T> void SDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = lamG-(beta1G*S*I1+betasG*S*I2)/NpopG-muG*S;
}
template <class T> void TDot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = u1*r1G*L1-muG*T_s+(1.0-(1.0-u2)*(pG+qG))*r2G*I1-(beta2G*T_s*I1+betasG*T_s*I2)/NpopG;
}
template <class T> void L1Dot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = (beta1G*S*I1+beta2G*T_s*I1-betasG*L1*I2)/NpopG
        -(muG+k1G)*L1-u1*r1G*L1+(1.0-u2)*pG*r2G*I1;
}
template <class T> void L2Dot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = (1.0-u2)*qG*r2G*I1-(muG+k2G)*L2+betasG*(S+L1+T_s)*I2/NpopG;
}
template <class T> void I1Dot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = k1G*L1-(muG+d1G)*I1-r2G*I1;
}
template <class T> void I2Dot::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = k2G*L2-(muG+d2G)*I2;
}
template <class T> void NPopConstraint::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
{
    lhs = S + T_s + L1 + L2 + I1 + I2 - NpopG;
}
template <class T> void IntegralCost::eq_def(T& lhs, T* x, T* u, T& t, T* s, T* p)
```

```
{
    lhs = L2 + I2 + B1G/2.0*u1*u1 + B2G/2.0*u2*u2;
}

void MinIntegralCost::eval_eq(double& lhs, double** x0, double** xf, double** q,
                              double** t0, double** tf, double* s, double* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinIntegralCost::eval_eq(HyperDual& lhs, HyperDual** x0, HyperDual** xf,
                              HyperDual** q, HyperDual** t0, HyperDual** tf,
                              HyperDual* s, HyperDual* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void MinIntegralCost::eval_eq(Bicomplex& lhs, Bicomplex** x0, Bicomplex** xf,
                              Bicomplex** q, Bicomplex** t0, Bicomplex** tf,
                              Bicomplex* s, Bicomplex* e)
{eq_def(lhs,x0,xf,q,t0,tf,s,e);}
void SDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void SDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                   HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void SDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                   Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void TDot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void TDot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t, HyperDual* s,
                   HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void TDot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t, Bicomplex* s,
                   Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void L1Dot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void L1Dot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                    HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void L1Dot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                    Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void L2Dot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void L2Dot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                    HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void L2Dot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                    Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void I1Dot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void I1Dot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                    HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void I1Dot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
```

```
                            Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void I2Dot::eval_eq(double& lhs, double* x, double* u, double& t, double* s, double* p)
{eq_def(lhs,x,u,t,s,p);}
void I2Dot::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                    HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void I2Dot::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                    Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void NPopConstraint::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                             double* p)
{eq_def(lhs,x,u,t,s,p);}
void NPopConstraint::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                             HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void NPopConstraint::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                             Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}
void IntegralCost::eval_eq(double& lhs, double* x, double* u, double& t, double* s,
                           double* p)
{eq_def(lhs,x,u,t,s,p);}
void IntegralCost::eval_eq(HyperDual& lhs, HyperDual* x, HyperDual* u, HyperDual& t,
                           HyperDual* s, HyperDual* p)
{eq_def(lhs,x,u,t,s,p);}
void IntegralCost::eval_eq(Bicomplex& lhs, Bicomplex* x, Bicomplex* u, Bicomplex& t,
                           Bicomplex* s, Bicomplex* p)
{eq_def(lhs,x,u,t,s,p);}




// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main.hpp
// CGPOPS Tool Box
// Declare main continouos time optimal control problem here
//


#ifndef __CGPOPS_MAIN_HPP__
#define __CGPOPS_MAIN_HPP__

#include "cgpopsFuncDec.hpp"
#include "nlpGlobVarExt.hpp"
#include "cgpopsAuxExt.hpp"


// $#$#$


void cgpops_go(doubleMat& cgpopsResults);
```

```cpp
#endif


// Copyright (c) Yunus M. Agamawi and Anil Vithala Rao.  All Rights Reserved
//
// cgpops_main_test.cpp
// Test Main
// Test CGPOPS functions
//

#include "cgpops_main.hpp"


void cgpops_go(doubleMat& cgpopsResults)
{
    // Define Global Variables used to determine problem size
    PG      = 1;    // Number of phases in problem
    nsG     = 0;    // Number of static parameters in problem
    nbG     = 0;    // Number of event constraints in problem
    nepG    = 0;    // Number of endpoint parameters in problem

    // Allocate memory for each phase
    initGlobalVars();

    // Define number of components for each parameter in problem phases
    // Phase 1 parameters
    nxG[0]  = 6;    // Number of state components in phase 1
    nuG[0]  = 2;    // Number of control components in phase 1
    nqG[0]  = 1;    // Number of integral constraints in phase 1
    ncG[0]  = 1;    // Number of path constraints in phase 1
    nppG[0] = 0;    // Number of phase parameters in phase 1

//    setGlobalTabularData(); // Set any global tabular data used in problem

    // Define mesh grids for each phase in problem for LGR collocation
    // Phase 1 mesh grid
    int M1 = numintervalsG;         // Number of intervals used in phase 1 mesh
    int initcolpts = initcolptsG;   // Initial number of collocation points in each
                                    // interval
    double fraction1[M1];           // Allocate memory for fraction vector for phase 1
                                    // mesh
    int colpoints1[M1];             // Allocate memory for colpoints vector for phase 1
                                    // mesh
    for (int m=0; m<M1; m++)
    {
        fraction1[m] = 1.0/((double) M1);
        colpoints1[m] = initcolpts;
    }
    setRPMDG(0,M1,fraction1,colpoints1);

    // Set information for transcribed NLP resulting from LGR collocation using defined
    // mesh grid
```

```
    setInfoNLPG();

    /*-----------------------------Provide Problem Bounds-----------------------------*/
    // Phase 1
    double t0min = 0,   t0max = 0;              // minimum and maximum initial times
    double tfmin = 5,   tfmax = 5;              // minimum and maximum final times
    double S0 = 76.0*NpopG/120.0;                  // initial value for S
    double T0 = NpopG/120.0;                     // initial value for T
    double L10 = 36.0*NpopG/120.0;                 // initial value for L1
    double L20 = 2.0*NpopG/120.0;                  // initial value for L2
    double I10 = 4.0*NpopG/120.0;                  // initial value for I1
    double I20 = NpopG/120.0;                    // initial value for I2
    double Nmin = 0,    Nmax =  30000;       // minimum and maximum N
    double u1min = 0.05,    u1max = 0.95;    // minimum and maximum u1
    double u2min = 0.05,    u2max = 0.95;    // minimum and maximum u2
    double qmin = 0, qmax = 10000;           // minimum and maximum integral values
    double cmin = 0, cmax = 0;               // minimum and maximum path constraint values


    // Throughout


    // Define Bounds of Optimal Control Problem
    // Phase 1 bounds
    int phase1 = 0;
    double x0l1[nxG[phase1]],   x0u1[nxG[phase1]];
    double xfl1[nxG[phase1]],   xfu1[nxG[phase1]];
    double xl1[nxG[phase1]],    xu1[nxG[phase1]];
    double ul1[nuG[phase1]],    uu1[nuG[phase1]];
    double ql1[nqG[phase1]],    qu1[nqG[phase1]];
    double cl1[ncG[phase1]],    cu1[ncG[phase1]];
    double t0l1,    t0u1;
    double tfl1,    tfu1;
    x0l1[0] = S0;
    x0l1[1] = T0;
    x0l1[2] = L10;
    x0l1[3] = L20;
    x0l1[4] = I10;
    x0l1[5] = I20;
    x0u1[0] = S0;
    x0u1[1] = T0;
    x0u1[2] = L10;
    x0u1[3] = L20;
    x0u1[4] = I10;
    x0u1[5] = I20;
    xfl1[0] = Nmin;
    xfl1[1] = Nmin;
    xfl1[2] = Nmin;
    xfl1[3] = Nmin;
    xfl1[4] = Nmin;
    xfl1[5] = Nmin;
    xfu1[0] = Nmax;
    xfu1[1] = Nmax;
    xfu1[2] = Nmax;
    xfu1[3] = Nmax;
```

```
   xfu1[4]  = Nmax;
   xfu1[5]  = Nmax;
   xl1[0]   = Nmin;
   xl1[1]   = Nmin;
   xl1[2]   = Nmin;
   xl1[3]   = Nmin;
   xl1[4]   = Nmin;
   xl1[5]   = Nmin;
   xu1[0]   = Nmax;
   xu1[1]   = Nmax;
   xu1[2]   = Nmax;
   xu1[3]   = Nmax;
   xu1[4]   = Nmax;
   xu1[5]   = Nmax;
   ul1[0]   = u1min;
   ul1[1]   = u2min;
   uu1[0]   = u1max;
   uu1[1]   = u2max;
   ql1[0]   = qmin;
   qu1[0]   = qmax;
   cl1[0]   = cmin;
   cu1[0]   = cmax;
   t0l1     = t0min;
   t0u1     = t0max;
   tfl1     = tfmin;
   tfu1     = tfmax;

   // Set parameterized constructor for NLP Phase Bounds Class
   setNLPPBG(phase1,x0l1,x0u1,xfl1,xfu1,xl1,xu1,ul1,uu1,ql1,qu1,cl1,cu1,t0l1,t0u1,tfl1,
            tfu1);

   // Whole problem bounds
   double sl[nsG], su[nsG];
   double bl[nbG], bu[nbG];

   // Set parameterized constructor for NLP Whole Bounds Class
   setNLPWBG(sl,su,bl,bu);

   // Provide initial guess for NLP Solver
   // Phase 1 guess
   double x0g1[nxG[phase1]],   xfg1[nxG[phase1]];
   double u0g1[nuG[phase1]],   ufg1[nuG[phase1]];
   double qg1[nqG[phase1]];
   double t0g1,    tfg1;
   x0g1[0] = S0;
   x0g1[1] = T0;
   x0g1[2] = L10;
   x0g1[3] = L20;
   x0g1[4] = I10;
   x0g1[5] = I20;
   xfg1[0] = S0;
   xfg1[1] = S0;
   xfg1[2] = L10;
```

```
    xfg1[3] = L20;
    xfg1[4] = I10;
    xfg1[5] = I20;
    u0g1[0] = 0.95;
    u0g1[1] = 0.95;
    ufg1[0] = 0.95;
    ufg1[1] = 0.95;
    qg1[0]  = 6000;
    t0g1    = t0min;
    tfg1    = tfmax;

    // Set parameterized constructor for NLP Phase Guess Class
    setNLPPGG(phase1,x0g1,xfg1,u0g1,ufg1,qg1,t0g1,tfg1);

    // Whole problem guess
    double sg[nsG];

    // Set parameterized constructor for NLP Phase Guess Class
    setNLPWGG(sg);

    // Set ocp functions
    objEqG = new MinIntegralCost;
    odeEqVecG = {{new SDot, new TDot, new L1Dot, new L2Dot, new I1Dot, new I2Dot}};
    pthEqVecG = {{new NPopConstraint}};
    igdEqVecG = {{new IntegralCost}};

    // Make call to CGOPS handler for IPOPT using user provided output settings
    CGPOPS_IPOPT_caller(cgpopsResults);

}



// cgpops_main_test.cpp
// Test Main
// Test CGPOPS functions
// By Yunus Agamawi
// 4/21/18
// Version 16
/*
Capabilities:
 - TBD
*/

#include "nlpGlobVarDec.hpp"
#include "cgpopsAuxDec.hpp"
#include "cgpops_main.hpp"


int main(void)
{
    printf("\nCGPOPS TESTING\n\n\n");

    int numTestRuns = 1;
```

```
    int numDS = 1;
    int DSSelect = 0;
    doubleMatMat cgpopsResultsMatMat(numTestRuns*numDS);
    doubleMat cgpopsResults;

    // Derivative supplier settings
    derivativeSupplierG = 0;     // Derivative supplier (default=0)
    scaledG             = 1;     // Scaling flag (default 1=on)

    // Mesh initialization settings
    numintervalsG   = 10;   // Initial number of mesh intervals per phase (default=10)
    initcolptsG     = 3;    // Initial number of collocation points per interval
                            // (default=4)

    // Mesh refinement settings
    meshRefineTypeG = 1;    // Select mesh refinement technique to be used (default=1)
    minColPtsG      = 3;    // Mininum number of collocation points used in an interval
                            // (default=4)
    maxColPtsG      = 10;   // Maximum number of collocation points used in an interval
                            // (default=10)
    maxMeshIterG    = 30;   // Maximum number of mesh iterations (default=20)
    meshTolG        = 1e-6; // Mesh tolerance (default=1e-7)

    // Output save settings
    saveIPOPTFlagG      = 0;   // Save IPOPT solution (default=1)
    saveMeshRefineFlagG = 0;   // Save mesh refinement history (default=0)
    saveHamiltonianG    = 0;   // Save Hamiltonian values (default=0)
    saveLTIHG           = 0;   // Save linear terms in Hamiltonian values (default=0)

    // IPOPT settings
    runIPOPTFlagG   = 1;    // Run IPOPT (default=1)
    NLPtolG         = 1e-7; // NLP Solver tolerance (default=1e-7)
    NLPmaxiterG     = 3000; // Maximum number of iterations allowed for NLP solver
    useLTIHDDG = 1;

    /*----------------------Changes to global parameter settings----------------------*/

    for (int ds=0; ds<numDS; ds++)
    {
        derivativeSupplierG = DSSelect;
        if (derivativeSupplierG==DSSelect)
        {
            for (int ni=0; ni<numTestRuns; ni++)
            {
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns] = getDoubleMat(6);
                printf("\nDerivativeSupplier = %d",derivativeSupplierG);
                printf("\nNumIntervals = %d",numintervalsG);
                cgpops_go(cgpopsResults);
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[0] = derivativeSupplierG;
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[1] = numintervalsG;
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[2] = cgpopsResults.val[0];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[3] = cgpopsResults.val[1];
                cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[4] = cgpopsResults.val[2];
```

```
                    cgpopsResultsMatMat.mat[ni+ds*numTestRuns].val[5] = cgpopsResults.val[3];
            }
        }
    }

    printf4MSCRIPT("cgpopsResultsMatMat",cgpopsResultsMatMat);
    printf("\n\n\n\n\n");

    return 0;
}
```
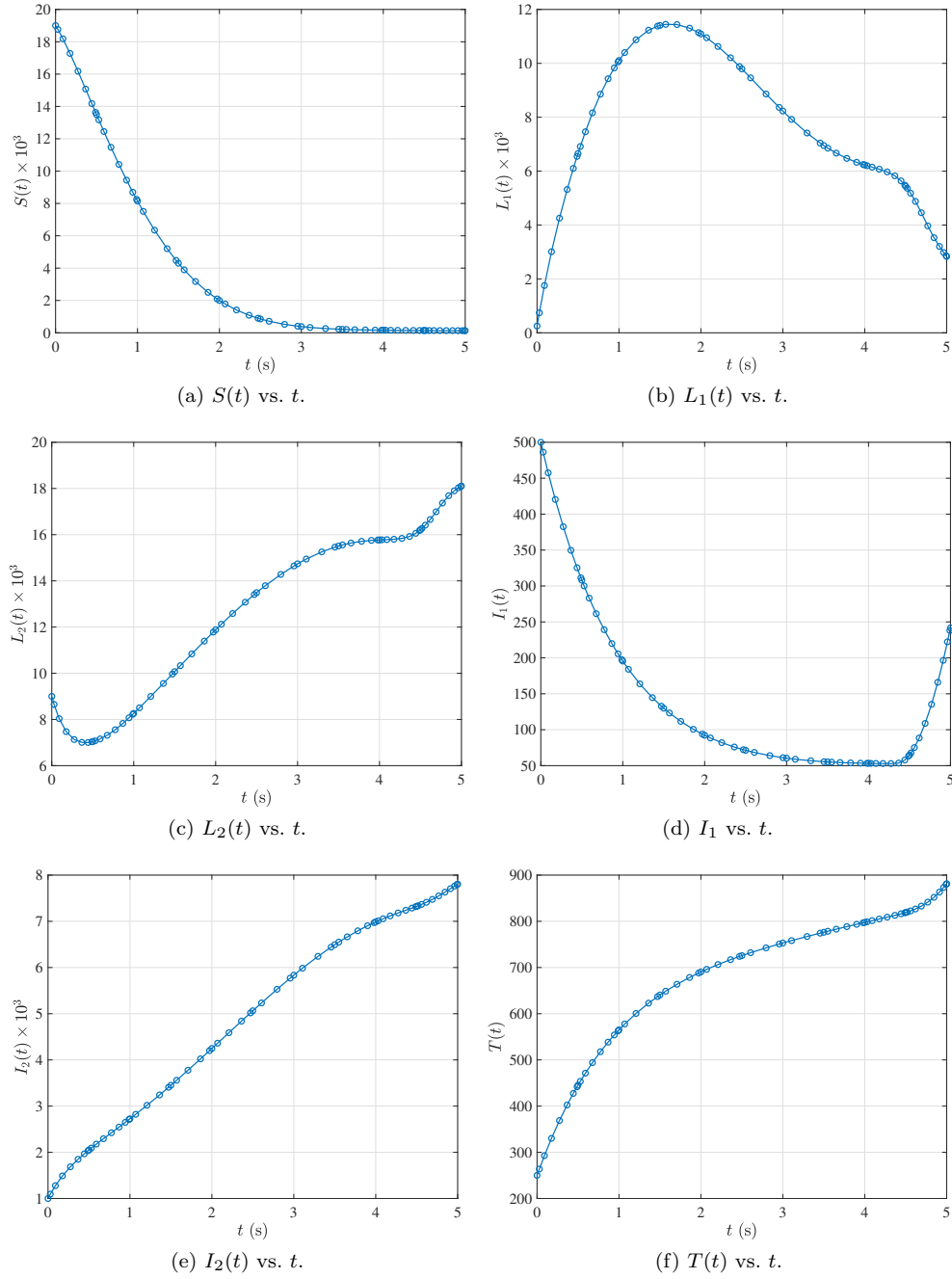
(a) $S(t)$ vs. t.

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

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

(d) I_1 vs. t.

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

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

Figure 8: Optimal State for Tuberculosis Optimal Control Problem Using $\mathbb{CGPOPS}$ with the NLP Solver IPOPT using the Hyper-Dual Derivative Supplier and a Mesh Refinement Tolerance of 10^{-6}.

(a) $u_1(t)$ vs. t. (b) $u_2(t)$ vs. t.
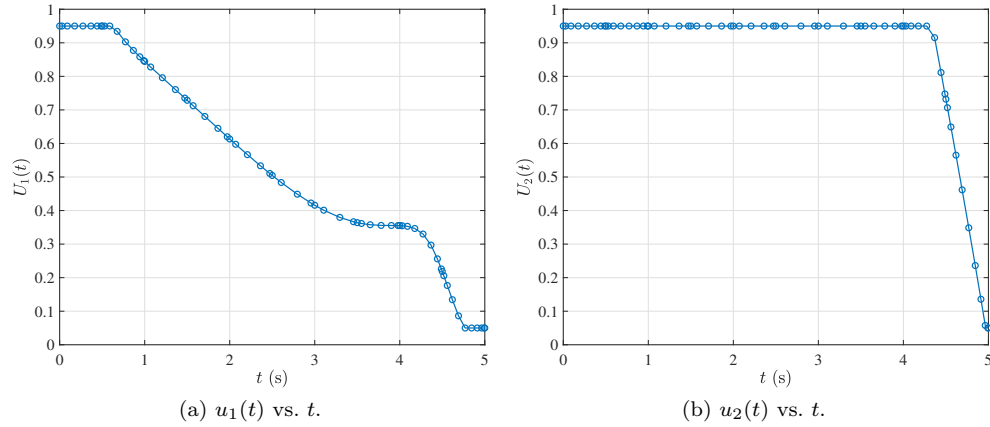
Figure 9: Optimal Control for Tuberculosis Optimal Control Problem Using $\mathbb{CGPOPS}$ with the NLP Solver IPOPT using Hyper-Dual Derivative Supplier and a Mesh Refinement Tolerance of 10^{-6}.

4 Concluding Remarks

While the authors have made the effort to make $\mathbb{CGPOPS}$ a user-friendly software, it is important to understand several aspects of computational optimal control in order to make $\mathbb{CGPOPS}$ 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 (e.g. the multi-stage launch vehicle ascent example). 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] Betts, J. T., "Sparse Jacobian Updates in the Collocation Method for Optimal Control Problems," *Journal of Guidance, Control, and Dynamics*, Vol. 13, No. 3, May–June 1990, pp. 409–415. https://doi.org/10.2514/3.25352.

[2] Betts, J. T., *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*, SIAM Press, Philadelphia, 2nd ed., 2009.

[3] Garg, D., Patterson, M. A., Darby, C. L., Francolin, C., Huntington, G. T., Hager, W. W., and Rao, A. V., "Direct Trajectory Optimization and Costate Estimation of Finite-Horizon and Infinite-Horizon Optimal Control Problems via a Radau Pseudospectral Method," *Computational Optimization and Applications*, Vol. 49, No. 2, June 2011, pp. 335–358. https://doi.org/10.1007/s10589--00--09291--0.

[4] Garg, D., Patterson, M. A., Hager, W. W., Rao, A. V., Benson, D. A., and Huntington, G. T., "A Unified Framework for the Numerical Solution of Optimal Control Problems Using Pseudospectral Methods," *Automatica*, Vol. 46, No. 11, November 2010, pp. 1843–1851. https://doi.org/10.1016/j.automatica.2010.06.048.

[5] Garg, D., Hager, W. W., and Rao, A. V., "Pseudospectral Methods for Solving Infinite-Horizon Optimal Control Problems," *Automatica*, Vol. 47, No. 4, April 2011, pp. 829–837. https://doi.org/10.1016/j.automatica.2011.01.085.

[6] 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. https://doi.org/10.2514/1.A32071.

[7] Agamawi, Y. M. and Rao, A. V., "Exploiting Sparsity in Direct Orthogonal Collocation Methods for Solving Multiple-Phase Optimal Control Problems," *2018 Space Flight Mechanics Meeting*, AIAA Paper2018-0724, Kissimmee, Florida, 8-12 January. https://doi.org/10.2514/6.2018-0724 2018.

[8] 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. https://doi.org/10.1002/oca.2114.

[9] 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. https://doi.org/10.1002/oca.957.

[10] Liu, F., Hager, W. W., and Rao, A. V., "Adaptive Mesh Refinement for Optimal Control Using Non-smoothness Detection and Mesh Size Reduction," *Journal of the Franklin Institute*, Vol. 352, No. 10, October 2015, pp. 4081–4106. https://doi.org/10.1016/j.jfranklin.2015.05.028.

[11] Liu, F., Hager, W. W., and Rao, A. V., "Adaptive Mesh Refinement for Optimal Control Using Using Decay Rates of Legendre Polynomial Coefficients," *IEEE Transactions on Control System Technology*, Vol. 26, No. 4, 2018, pp. 1475 – 1483. https://doi.org/10.1109/TCST.2017.2702122.

[12] Fike, J. and Alonso, J., "The Development of Hyper-Dual Numbers for Exact Second-Derivative Calculations," *49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, AIAA Paper 2011-886, Orlando, Florida, 4-7 January 2011. https://doi.org/10.2514/6.2011-886.

[13] Lantoine, G., Russell, R. P., and Dargent, T., "Using Multicomplex Variables for Automatic Computation of High-Order Derivatives," *ACM Transactions on Mathematical Software*, Vol. 38, April 2012, pp. 16:1–16:21. 10.1145/2168773.2168774.

[14] Gill, P. E., Murray, W., and Wright, M. H., *Practical Optimization*, Academic Press, London, 1981.

[15] 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. https://doi.org/10.1002/(SICI)1099--1514(200001/02)21:1<1::AID--OCA646>3.0.CO;2--V.

[16] 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.

[17] 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. https://doi.org/10.1145/1731022.1731032.

[18] 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. https://doi.org/10.1002/oca.814.

[19] 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, November–December 1969, pp. 481–488. https://doi.org/10.2514/3.44093.

[20] Zhao, Y. and Tsiotras, P., "Density Functions for Mesh Refinement in Numerical Optimal Control," *Journal of Guidance, Control, and Dynamics*, Vol. 34, No. 1, January–February 2011, pp. 271–277. https://doi.org/10.2514/1.45852.

[21] 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. http://dx.doi.org/10.3934/dcdsb.2002.2.473.

[22] Biegler, L. T. and Zavala, V. M., "Large-Scale Nonlinear Programming Using IPOPT: An Integrating Framework for Enterprise-Wide Optimization," *Computers and Chemical Engineering*, Vol. 33, No. 3, March 2008, pp. 575–582. https://doi.org/10.1016/j.compchemeng.2008.08.006.

[23] 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. https://doi.org/10.1016/S0005--1098(98)00161--7.

[24] 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. https://doi.org/10.2514/2.4570.

[25] 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. https://doi.org/10.2514/2.5035.

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

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