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

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. ℂGℙOℙS uses some of the latest advancements in the area of direct orthogonal collocation methods for solving optimal control problems. ℂGℙOℙS employs an *hp*-adaptive Radau Gaussian quadrature method where the collocation is performed at the Legendre-Gauss-Radau quadrature points. 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. 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 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 ℍℙOℙS − 𝕀𝕀

ℂGℙOℙS represents a more computationally efficient and portable C++ implementation of the optimal control framework utilized by the MATLAB software ℍℙOℙS − 𝕀𝕀 first made available in 2013. Specifically, ℂGℙOℙS is organized much differently than ℍℙOℙS − 𝕀𝕀 so as to facilitate computational efficiency of the optimal control framework in a C++ software implementation. Moreover, 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 ℂGℙOℙS and ℍℙOℙS − 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 ℂGℙOℙS are happy to assist users of ℍℙOℙS − 𝕀𝕀 in rewriting their code for ℂGℙOℙS.

Acknowledgments

The authors gratefully acknowledge support for this research from the U.S. Office of Naval Research under Grant N00014-15-1-2048 and from the U.S. National Science Foundation under Grants DMS-1522629 and CMMI-1563225.

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 ℂGℙOℙS, you agree to the terms of the license agreement as stated on the ℍℙOℙS − 𝕀𝕀 website http://www.gpops2.com/License/License.html. In other words, the ℂGℙOℙS licensing agreement follows the licensing agreement of ℍℙOℙS − 𝕀𝕀. **Please read the license terms and conditions carefully before proceeding to download, install, or use of** ℂGℙOℙS.

Contents

1 Introduction to the General-Purpose Software ℂ𝔾ℙ𝕆ℙ𝕊

The general multiple-phase optimal control problem that can be solved by ℂ𝔾ℙ𝕆ℙ𝕊 is given as follows. First let $p \in \{1, \ldots, P\}$ 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

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

subject to the dynamic constraints

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

the event constraints

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

the inequality path constraints

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

the integral constraints

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

and the static parameter constraints

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

where

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

and the integrals in each phase are defined as

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

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. **?** and **?**. Furthermore, each phase may be further refined into $K^{(p)}$ multiple-intervals such that Eq. (8) becomes

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

1.1 Radau Collocation Method Employed by ℂ𝔾ℙ𝕆ℙ𝕊

The method employed by 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 ℂ𝔾ℙ𝕆ℙ𝕊 can be found in Refs. **?**, **?**, **?**, **?**, and **?**.

1.2 Organization of ℂ𝔾ℙ𝕆ℙ𝕊

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 purple underlined characters. Users who do not have color rendering capability will see only *slanted*, **boldface**, and underlined 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 $\mathbb{CGPOPS}$ (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.

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

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

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

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:

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

```
// 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:

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

```
// 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:

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

```
// 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:

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

```
// 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:

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

```
// 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:

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

```
// 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

$$\left[2/7\,,\quad 15\,,\quad e^{5}\,,\quad 10^{-4}\,,\quad \pi\right] \tag{18}$$

```
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 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

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

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 doubleMat 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:

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

```
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:

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

```
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:

$$s_1 = 0.5 \quad , \quad s_2 = 2 \tag{23}$$

```
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 EndPointFunction 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 **EndPointFunction** 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, const std::string& linear_solver="ma57")
```

The function **CGPOPS_IPOPT_caller** should call the IPOPT application to solve the resulting NLP of the LGR transcription defined by the user. The first argument is used as the output of **CGPOPS_IPOPT_caller** and 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. The second argument **linear_solver** is an optional input to specify the linear solver used by IPOPT.

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, "mumps");
}
```

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 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. **?**, **?**, **?**, and **?**, 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. **?**, **?**, and **?**, 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

CGPOPS is compiled using a Makefile executable that requires full paths to the directories where the 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
```

If on Linux, paths to shared libraries may need to be added to the environmental variable **LD_LIBRARY_PATH**.

Adding to environmental variable **LD_LIBRARY_PATH** for single bash session.

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/libs:/path/to/more/libs
```

To permanently add to **LD_LIBRARY_PATH** for all future sessions, add above to the local **~/.bashrc** file.

2.6 CMake Build Process

A CMake build process for compiling a libcgpops binary and all examples is available for CMake versions 3.13 and newer. The following commands in the ℂ𝔾ℙ𝕆ℙ𝕊 installation directory are typically sufficient:

```
$ mkdir build && cd build
$ cmake ..
$ cmake --build .
$ cmake --install .
```

Executables binaries are installed in **~/build/bin**. Library binaries are installed in **~/build/lib**. Header files are installed in **~/build/include**.

NOTE: The IPOPT installation directory is assumed to be installed in the same directory as ℂ𝔾ℙ𝕆ℙ𝕊 or in a **~/cgpops/thirdparty** directory under the name **Ipopt-3.12.13**, **ipopt-3.12.13**, **Ipopt**, or **ipopt**.

Examples of assumed directory tree structure:

```
Software
|___ cgpops
|      |___ build
|      |      |___ bin
|      |      |___ include
|      |      |___ lib
|      |___ ... (other files/folders)
|___ Ipopt-3.12.13
|      |___ include
|      |___ lib
|      |___ ... (other files/folders)

Software
|___ cgpops
|      |___ build
|      |      |___ bin
|      |      |___ include
|      |      |___ lib
|      |___ ... (other files/folders)
|      |___ thirdparty
|      |      |___ ipopt
|      |      |      |___ build
|      |      |      |      |___ include
|      |      |      |      |___ lib
|      |      |___ ... (other files/folders)
```

If IPOPT is not installed under one of the assumed names or locations, add the following define during the CMake compilation and generation step:

```
$ cmake .. -DIPOPT_DIR=/path/to/Ipopt/installation/directory
```

3 Examples of Using CGPOPS

In this section we provide seven examples of using 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. **?**. The second example is a multiple-stage launch vehicle ascent problem taken from Refs. **?**, **?**, and **?**. The third example is a tumor anti-angiogenesis optimal control problem from Refs. **?** and **?**. The fourth example is the reusable launch vehicle entry problem taken from Ref. **?**. The fifth example is the minimum time-to-climb of a supersonic aircraft taken from Refs. **?** and **?**. The sixth example is the optimal control of a hang glider and is taken from Ref. **?**. Finally, the seventh example is the optimal control of a two-strain tuberculosis model and is taken from Ref. **?**. For each example the optimal control problem is described quantitatively, the CGPOPS code is provided, and the solution obtained using 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,[?] which is freely available at https://projects.coin-or.org/Ipopt.

3.1 Hyper-Sensitive Problem

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

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

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

The state $x(t)$, control $u(t)$, and mesh refinement history that arise from the execution of 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.

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. **?**, **?**, and **?**.

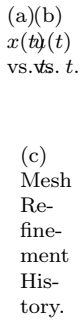

Figure 1: Solution to Hyper-Sensitive Problem Obtained Using $\mathbb{CGPOPS}$ with the NLP Solver IPOPT using the Hyper-Dual Derivative Supplier and a Mesh Refinement Tolerance of 10^{-7}.

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}

3.2.1 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. **?** or **?** and is shown in Table 7.

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

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 $\mathbf{r}(t) = \left[\begin{array}{ccc} x(t) & y(t) & z(t) \end{array} \right]$ is the position, $\mathbf{v} = \left[\begin{array}{ccc} v_x(t) & v_y(t) & v_z(t) \end{array} \right]$ 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, $\mathbf{u} = \left[\begin{array}{ccc} u_x & u_y & u_z \end{array} \right]$ is the thrust direction, and $\mathbf{D} = \left[\begin{array}{ccc} D_x & D_y & D_z \end{array} \right]$ 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

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

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

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

where ρ_0 is the atmospheric density at sea level, $h = \|\| - R_e$ 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

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

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.[?]

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 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 (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 ℂGℙOℙS is summarized in the following three plots that contain the altitude, speed, and controls.

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

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

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

Figure 2: Solution to Launch Vehicle Ascent Problem Using ℂGℙOℙS with the NLP Solver IPOPT using the Hyper-Dual Derivative Supplier and a Mesh Refinement Tolerance of 10^{-6}.

3.3 Tumor-Antiangiogenesis Optimal Control Problem

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

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

subject to the dynamic constraints

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

with the initial conditions

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

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) $(p(t), q(t))$ 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}.

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. **?**, **?**, and **?**.

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. **?**. 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

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

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

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.

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

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

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

and the boundary conditions

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

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. **?**. 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.

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

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

(c)(d)
$\gamma(t)$ 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. **?** and the problem considered here is identical to that of Ref. **?**. 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, $C_D = C_{D0} + KC_L^2$ is the coefficient of drag, and C_L is the coefficient of lift (where $0 \leq C_L \leq C_{L,\max}$). The constants for this problem are taken verbatim from Ref. **?** 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 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.

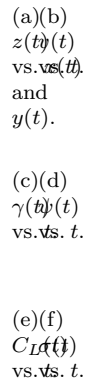

Figure 7: Solution to Dynamic Soaring Problem Using $\mathbb{GPOPS}-\mathbb{II}$ with the NLP Solver IPOPT and a Mesh Refinement Tolerance of 10^{-6}.

3.7 Two-Strain Tuberculosis Optimal Control Problem

Quoting from Ref. **?**, "[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. **?** 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

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

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. **?** (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.

(a) $S(t)$ vs. t. (b) $I_1(t)$ vs. t. (c) $L_2(t)$ vs. t. (d) $I_2(t)$ vs. t. (e) $T(t)$ vs. t. (f)

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}.

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.