User's Manual for $\mathcal{DMG}$ Version 1.0:

A MATLAB® Software for Solving Multiple-Phase Optimal Control Problems based on $\mathcal{GPOPS}$

David Morante González
Universidad Carlos III de Madrid
Avenida de la Universidad 30, Leganés, Madrid 28911

Acknowledgments

The software $\mathcal{DMG}$ was developed in response to a demand from the research and academic community for a complete open-source tool for solving optimal control problems.

This distribution is a derived work from the open-source project "$\mathcal{GPOPS}$ 2.2" (Gauss Pseudospectral Optimization Software). The $\mathcal{GPOPS}$ version was downloaded in October 2009 from SourceForge. It was developed at MIT, Draper Laboratory, and The University of Florida, and was published under the Simple Public License. However, the open-source version of $\mathcal{GPOPS}$ was designed to interface with the NLP solver SNOPT, which is a proprietary software. Therefore, we have modified $\mathcal{GPOPS}$ in such a way that now is able to work with the open-source solver IPOPT. Additionally, the authors have included the Hermite-Simpson collocation scheme to further expand $\mathcal{GPOPS}$ features.

The $\mathcal{DMG}$ software follows the same philosophy than $\mathcal{GPOPS}$, which is an attempt to fill that void and enable researchers, educators, and others involved in solving complex optimal control problems to take advantage of a code that can be customized to one's particular needs. The authors of $\mathcal{DMG}$ hope sincerely that the code is useful.

Disclaimer

Preface

This Manual is only intended to serve as a quick usage guide. It is highly based on the $\mathcal{GPOPS}$ Manual, which can be found in this folder, but it contains the changes that has been included within the $\mathcal{DMG}$ version.

The main $\mathcal{GPOPS}$ features are:

- Implements Gauss Pseudospectral Collocation Method for Transcribing the continuous optimal control problem into an Nonlinear Optimization Problem.
- Interfaces with SNOPT
- Includes Automatic, Forward Numerical, Complex-Step and Analytical Derivatives for gradient and jacobian computations.

The new features $\mathcal{DMG}$ includes:

- Implements Hermite Simpson Collocation Method for Transcribing the continuous optimal control problem into a Nonlinear Optimization Problem.
- Interfaces with IPOPT
- Multi-core computation of Forward numerical derivatives and Complex Step differentiation.

For further information about $\mathcal{GPOPS}$ and about the test cases, see the $\mathcal{GPOPS}$ manual.

Licensing Agreement

Contents

Chapter 1

Constructing an Optimal Control Problem

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

1.1 Organization

In order to specify the optimal control problem that is to be solved, the user must write MATLAB functions that define the following functions in each phase of the problem:

(1) the cost functional

(2) the right-hand side of the differential equations and the path constraints(i.e., the differential-algebraic equations)

(3) the boundary conditions (i.e., event conditions)

(4) the linkage constraints (i.e., how the phases are connected)

In addition, the user must also specify the lower and upper limits on every component of the following quantities:

(1) initial and terminal time of the phase

(2) the state at the following points in time:
- at the beginning of the phase
- during the phase
- at the end of the phase

(3) the control

(4) the static parameters

(5) the path constraints

(6) the boundary conditions

(7) the phase duration (i.e., total length of phase in time)

(8) the linkage constraints (i.e., phase-connect conditions)

It is noted that each of the functions must be defined for each phase of the problem. The remainder of this document is devoted to describing in detail the MATLAB® syntax for describing the optimal control problem and each of the constituent functions.

1.2 Notation Used Throughout Remainder of This Manual

The following notation is adopted for use throughout the remainder of this manual. First, all user-specified names will be denoted by *slanted* characters (not *italic*, but *slanted*). Second, any item denoted by **boldface characters** are pre-defined and cannot be changed by the user. Finally, users with color capability will see the slanted characters in *red* and will see the boldface characters in **blue**.

1.3 Preliminary Information

Before proceeding to the details of setting up a problem, the following few preliminary details are useful. First, it is important to understand that the interface is laid out in ***phases***. Using a phase-based approach, it is possible to describe each segment of the problem independently of the other segments. The segments are then *linked* together using linkage conditions (or phase-connect conditions). Second, it is important to note that it uses the vectorization capabilities of MATLAB. In this vein all matrices and vectors are oriented ***column-wise*** for maximum efficiency. As you read through this chapter, please keep in mind the column-wise orientation of all matrices used.

1.4 Call to $\mathcal{DMG}$

The call to $\mathcal{DMG}$ is deceptively simple and is given as follows:

output=DMG(setup)

The input *setup* is a user-defined structure that contains all of the information about the optimal control problem to be solved [1]. Finally, the variable *output* is a structure that contains all of the information from the original problem *plus* the information from the run itself (i.e., the solution)[2]. The remainder of this chapter is devoted to describing the fields in the structure *setup*.

1.5 Syntax for Setup Structure

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

- **name**: a string containing the name of the problem.
- **funcs**: a structure whose elements contain the names of the user-defined function in the problem (see Section 1.6 below).
- **limits**: an array of structures that contains the information about the lower and upper limits on the variables and constraints in each phase of the problem (see Section 1.7 below).
- **guess**: an array of structures that contains contains a guess of the solution in each phase of the problem (see Section 1.10 below).

The optional fields (and their default values) are as follows:

- **linkages**: an array of structures that contains the information about the lower and upper limits of the linkage constraints (see Section 1.8 below).
- **direction**: a string that indicates the direction of the independent variable. The two possible values for this string are "increasing" and "decreasing". (default="increasing")
- **autoscale**: a string that indicates whether or not the user would like the optimal control problem to be scaled automatically before it is solved. (default="off") (see Section 1.11 below).

[1]see the detailed description of *setup* in Section 1.5
[2]See the detailed description of the output in Section 1.13.

- **derivatives**: a string indicating differentiation method to be used. Possible values for this string are "numerical", "complex", "automatic", "analytic" (default="numerical") (see Section 1.12 below).
- **checkDerivatives**: a flag to check user defined analytic derivatives (default="0") (see Section 1.12 below).
- **maxIterations**: a positive integer indicating the maximum number of iterations that can be taken by the NLP solver.
- **method**: a string indicating transcription method to be used. Possible values for this string are "hermite-simpson", "pseudospectral'.
- **solver**: a string indicating nonlinear-programming solver to be used. Possible values for this string are "ipopt", and "snopt'.
- **parallel**: a string indicating if multi-core computation of derivatives is to be used. Possible values for this string are "yes', and "now'. This option only applies when numerical or complex derivatives options are selected.

1.6 Specifying Function Names Used in Optimal Control Problem

The syntax for specifying the names of the MATLAB functions is done by setting the fields in the structure **FUNCS** and is given as follows:

$$\begin{array}{lcl} setup.\textbf{funcs.cost} & = & `costfun.m' \\ setup.\textbf{funcs.dae} & = & `daefun.m' \\ setup.\textbf{funcs.event} & = & `eventfun.m' \\ setup.\textbf{funcs.link} & = & `linkfun.m' \end{array}$$

Example of Specifying Function Names

Suppose we have a problem whose cost functional, differential-algebraic equations, event constraints, and linkage constraints are defined, respectively, via the *user-defined* functions *mycostfun.m*, *mydaefun.m*, *myeventfun.m*, and *mylinkfun.m*. Then the syntax for specifying these functions is given as follows:

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

1.7 Syntax for limits Structure

Once the user-defined structure *setup* has been defined, the next step in setting up a problem is to create an array of structures of length P (where P is the number of phases) called **limits**, where **limits** is a field of the structure *setup*. The array of structures **limits** is specified as follows:

- **limits(p).nodes**: scalar value specifying the number of nodes in phase $p \in [1, \ldots, P]$.
- **limits(p).time.min** and **limits(p).time.max**: row vectors, each of length two, that contain the information about the lower and upper limits, respectively, on the initial and terminal time in phase $p \in [1, \ldots, P]$. The row vectors **limits(p).time.min** and **limits(p).time.max** have the following form:

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

- **limits(p).state.min** and **limits(p).state.max**: matrices, each of size $n_p \times 3$, that contain the lower and upper limits, respectively, on the state in phase $p \in [1, \ldots, P]$. Each of the columns of the matrices **limits(p).state.min** and **limits(p).state.max** are given as follows:
  - **limits(p).state.min(:,1)**: a column vector containing the lower (upper) limits on the state at the *start* of phase $p \in [1, \ldots, P]$.
  - **limits(p).state.min(:,2)**: a column vector containing the lower (upper) limits on the state at the *during* phase $p \in [1, \ldots, P]$.
  - **limits(p).state.min(:,3)**: a column vector containing the lower (upper) limits on the state at the *terminus* of phase $p \in [1, \ldots, P]$.

  The matrices **limits(p).state.min** and **limits(p).state.max** then have the following form:

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

- **limits(p).control.min** and **limits(p).control.max**: column vectors, each of length m_p, that contain the lower and upper limits, respectively, on the controls in phase $p \in [1, \ldots, P]$. The column vectors **limits(p).control.min** and **limits(p).control.max** have the following form:

$$\begin{aligned}
\text{limits}(p).\text{control.min} &= \begin{bmatrix} u_1^{\text{min}} \\ \vdots \\ u_m^{\text{min}} \end{bmatrix} \\
\text{limits}(p).\text{control.max} &= \begin{bmatrix} u_1^{\text{max}} \\ \vdots \\ u_m^{\text{max}} \end{bmatrix}
\end{aligned}$$

- **limits(p).parameter.min** and **limits(p).parameter.max**: column vectors, each of length q_p, that contain the lower and upper limits, respectively, on the static parameters in phase $p \in [1, \ldots, P]$. The column vectors **limits(p).parameter.min** and **limits(p).parameters.max** have the following form:

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

- **limits(p).path.min** and **limits(p).path.max**: column vectors, each of length r_p, that contain the lower and upper limits, respectively, on the path constraints in phase $p \in [1, \ldots, P]$. The column vectors

limits(p).path.min and **limits(p).path.max** have the following form:

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

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

- **limits(p).event.min** and **limits(p).event.max**: column vectors, each of length e_p, that contain the lower and upper limits on the event constraints in phase $p \in [1, \ldots, P]$. The column vectors **limits(p).event.min** and **limits(p).event.max** have the following form:

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

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

- **limits(p).duration.min** and **limits(p).duration.max**: scalars that contain the lower and upper limits on the duration of phase $p \in [1, \ldots, P]$. The scalars **limits(p).duration.min** and **limits(p).duration.max** have the following form:

$$\begin{aligned} \texttt{limits}(p)\texttt{.duration.min} &= T^{\min} \\ \texttt{limits}(p)\texttt{.duration.max} &= T^{\max} \end{aligned}$$

Note: any fields that do not apply to a problem (i.e. a problem without event constraints, path constraints, etc.) may be omitted or left as empty matrices ("[]").

Example of Setting Up a Limits Cell Array

As an example of setting up a limits cell array, consider the following two-phase optimal control problem. In particular, suppose that *phase 1* of the problem has 3 states, 2 controls, 2 path constraints, and 5 event constraints. Suppose further that the lower and upper limits on the initial and terminal time in the first phase are given as

$$\begin{array}{rcccl} 0 & \leq & t_0^{(1)} & \leq & 0 \\ 50 & \leq & t_f^{(1)} & \leq & 100 \end{array}$$

Next, suppose that the lower and upper limits on the states at the *start* of the first phase are given, respectively, as

$$\begin{array}{rcccl} 1 & \leq & x_1(t_0^{(1)}) & \leq & 1 \\ -3 & \leq & x_2(t_0^{(1)}) & \leq & 0 \\ 0 & \leq & x_2(t_0^{(1)}) & \leq & 5 \end{array}$$

Similarly, suppose that the lower and upper limits on the states *during* the first phase are given, respectively, as

$$\begin{array}{rcccl} 1 & \leq & x_1(t^{(1)}) & \leq & 10 \\ -50 & \leq & x_2(t^{(1)}) & \leq & 50 \\ -20 & \leq & x_2(t^{(1)}) & \leq & 20 \end{array}$$

Finally, suppose that the lower and upper limits on the states at the *terminus* of the first phase are given, respectively, as

$$\begin{array}{rcccl} 5 & \leq & x_1(t_f^{(1)}) & \leq & 7 \\ 2 & \leq & x_2(t_f^{(1)}) & \leq & 2.5 \\ -\pi & \leq & x_2(t_f^{(1)}) & \leq & \pi \end{array}$$

Next, suppose that the lower and upper limits on the controls *during* the first phase are given, respectively, as

$$\begin{array}{rcccl} -50 & \leq & u_1(t^{(1)}) & \leq & 50 \\ -100 & \leq & u_2(t^{(1)}) & \leq & 100 \end{array}$$

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

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

Next, suppose that the lower and upper limits on the event constraints of the first phase are given, respectively, as

$$\begin{array}{rcccl} 0 & \leq & \phi_1^{(1)} & \leq & 1 \\ -2 & \leq & \phi_2^{(1)} & \leq & 4 \\ 8 & \leq & \phi_3^{(1)} & \leq & 20 \\ 3 & \leq & \phi_4^{(1)} & \leq & 3 \\ 10 & \leq & \phi_5^{(1)} & \leq & 10 \end{array}$$

In a similar manner, suppose that *phase 2* of the problem contains the following information: 4 states, 3 controls, 1 path constraint, and 4 event constraints. Also, suppose now that the lower and upper limits on the initial and terminal time in the first phase are given, respectively, as

$$\begin{array}{rcccl} 50 & \leq & t_0^{(2)} & \leq & 100 \\ 100 & \leq & t_f^{(2)} & \leq & 200 \end{array}$$

Next, suppose that the lower and upper limits on the states at the *start* of the second phase are given, respectively, as

$$\begin{array}{rcccl} 3 & \leq & x_1(t_0^{(2)}) & \leq & 3 \\ -10 & \leq & x_2(t_0^{(2)}) & \leq & 4 \\ 7 & \leq & x_3(t_0^{(2)}) & \leq & 18 \\ 25 & \leq & x_4(t_0^{(2)}) & \leq & 75 \end{array}$$

Similarly, suppose that the lower and upper limits on the states *during* the second phase are given, respectively, as

$$\begin{array}{rcccl} -200 & \leq & x_1(t^{(2)}) & \leq & 200 \\ -50 & \leq & x_2(t^{(2)}) & \leq & 50 \\ -20 & \leq & x_3(t^{(2)}) & \leq & 20 \\ -80 & \leq & x_4(t^{(2)}) & \leq & 80 \end{array}$$

Finally, suppose that the lower and upper limits on the states at the *terminus* of the second phase are given, respectively, as

$$\begin{array}{rcccl} 12 & \leq & x_1(t_f^{(2)}) & \leq & 12 \\ -60 & \leq & x_2(t_f^{(2)}) & \leq & 30 \\ -90 & \leq & x_3(t_f^{(2)}) & \leq & 10 \\ 100 & \leq & x_4(t_f^{(2)}) & \leq & 500 \end{array}$$

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

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

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

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

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

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

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

```
iphase = 1; % Set the phase number to 1
limits(iphase).nodes = 10;
limits(iphase).time.min = [0 50];
limits(iphase).time.max = [0 100];
limits(iphase).state.min = [1 1 5; -3 -50 2; 0 -20 -pi];
limits(iphase).state.max = [1 10  7; 0  50 2.5; 5 20 pi];
limits(iphase).control.min = [-50; -100];
limits(iphase).control.max = [ 50;  100];
limits(iphase).parameter.min = [];
limits(iphase).parameter.max = [];
limits(iphase).path.min = [-10; 1];
limits(iphase).path.max = [10; 1];
limits(iphase).event.min = [0; -2; 8; 3; 10];
limits(iphase).event.max = [1; 4; 20; 3; 10];

iphase = 2; % Set the phase number to 2
limits(iphase).nodes = 10;
limits(iphase).time.min = [50 100];
limits(iphase).time.max = [100 200];
limits(iphase).state.min = [3 -200 12; -10 -50 -60; 7 -20 -90; 25 -80 100];
limits(iphase).state.max = [3 200 12; 4 50 30; 18 20 10; 75 80 500];
limits(iphase).control.min = [-90; -120];
limits(iphase).control.max = [ 90;  120];
limits(iphase).parameter.min = [];
limits(iphase).parameter.max = [];
limits(iphase).path.min = [-10; 10];
limits(iphase).path.max = [1; 1];
limits(iphase).event.min = [0; -2; 8; 3];
limits(iphase).event.max = [1; 4; 20; 3];

setup.limits = limits;
```

Note: in order to make the coding easier, we have introduced the auxiliary integer variable**iphase** so that the user can more easily reuse code from phase to phase.

1.8 Syntax for linkages Array of Structures

Another required field in the structure *setup* is an array of structures called **linkages** that defines the way that the phases are to be linked. If there is only one phase in the problem, then *setup*.**linkages** may be set to "[]". If the problem contains more than a single phase, then **linkages** is an array of structures of length L (where L is the number of pairs of phases to be linked). The array of structures **linkages** is specified as follows:

- **linkages(s).min**: a column vector of length l_s containing the lower limits on the s^{th} pair of linkages.
- **linkages(s).max**: a column vector of length l_s containing the upper limits on the s^{th} pair of linkages.
- **linkages(s).left.phase**: an integer containing the "left" phase in the pair of phases to be connected
- **linkages(s).right.phase**: an integer containing the "right" phase in the pair of phases to be connected

Note that we use the terminology "left" and "right" in the sense of viewing a graph of the trajectory on a page where time is increasing to the right. Thus, the "left" phase corresponds to the terminus of a phase while the "right" phase corresponds to the start of a phase.

1.9 Syntax of Each Function in Optimal Control Problem

Now that we know *which* functions will be used, the next step is to discuss the syntax of each of these functions. In general, the syntax for each function will differ because the quantities being evaluated are different in nature. In this section we will explain the syntax of each function.

1.9.1 Syntax of Function Used to Evaluate Cost

The syntax used to evaluate a user-defined cost functional is given as follows:

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

where *mycostfun.m* is the name of the MATLAB function, *solcost* is the input to the function, and *Mayer* and *Lagrange* are the outputs. The input *solcost* is a structure while the outputs *Mayer* and *Lagrange* are the endpoint cost and the integrand of the integrated cost, respectively. The input structure *solcost* has the following fields (note that N=number of LG points which are on the interior of the time interval):

- *solcost*.**phase**: the phase number
- *solcost*.**initial.time**: the initial time in phase *solcost*.**phase**
- *solcost*.**initial.state**: the initial state in phase *solcost*.**phase**
- *solcost*.**terminal.time**: the terminal time in phase *solcost*.**phase**
- *solcost*.**terminal.state**: the terminal state in phase *solcost*.**phase**
- *solcost*.**time**: a column vector of length N that contains the time (excluding the initial and terminal points) in phase *solcost*.**phase**
- *solcost*.**state**: a matrix of size $N \times n$ (where n is the number of states) that contains the values of the state (excluding the initial and terminal points) in phase *solcost*.**phase**
- *solcost*.**control**: a matrix of size $N \times m$ (where m is the number of controls) that contains the values of the control (excluding the initial and terminal points) in phase *solcost*.**phase**
- *solcost*.**parameter**: a column vector of length q that contains the values of the static parameters in phase *solcost*.**phase**

Finally, the outputs of *mycostfun* are as follows:

- *Mayer*: a *scalar*, i.e., size 1×1
- *Lagrange*: a *column* vector of size $N \times 1$

1.9.2 Warning About Outputs to Cost Function

For many optimal control problems the output *Lagrange* in the user-defined cost function *mycostfun* is ***zero***. As such, it is appealing to set *Lagrange* to zero by the MATLAB command

$$\text{Lagrange=0;} \tag{1–1}$$

However, ***the integrand cannot be set to a scalar value!***. Instead, the integrand ***must*** be set to a ***column vector of zeros!***. The way to set the integrand to zero and that ***will work in all cases*** (i.e., numerical or automatic differentiation) is as follows:

$$\boxed{\text{Lagrange=zeros(size(}\textit{solcost}.\textbf{time}\text{);}} \tag{1–2}$$

The user is urged to use the syntax of Eq. (1–2) whenever the integrand is identically zero.

Example of a Cost Functional

Suppose we have a two-phase optimal control problem that uses a cost functional named "mycostfun.m". Suppose further that the dimension of the state in each phase is 2 while the dimension of the control in each phase is 2. Also, suppose that the endpoint and integrand cost in phase 1 are given, respectively, as

$$\begin{aligned} \Phi^{(1)}(\mathbf{x}^{(1)}(t_0), t_0^{(1)}, \mathbf{x}^{(1)}(t_f), t_f^{(1)}) &= \mathbf{x}^T(t_f)\mathbf{S}\mathbf{x}(t_f) \\ \mathcal{L}^{(1)}(\mathbf{x}^{(1)}(t), \mathbf{u}^{(1)}(t), t) &= \mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{u}^T\mathbf{R}\mathbf{u} \end{aligned}$$

while the endpoint and integrand in phase 2 are given, respectively, as

$$\begin{aligned} \Phi^{(2)}(\mathbf{x}^{(2)}(t_0^{(2)}), t_0^{(2)}, \mathbf{x}^{(2)}(t_f^{(2)}), t_f^{(2)}) &= \mathbf{x}^T(t_f)\mathbf{x}(t_f) \\ \mathcal{L}^{(2)}(\mathbf{x}^{(2)}(t), \mathbf{u}^{(2)}(t), t) &= \mathbf{u}^T\mathbf{R}\mathbf{u} \end{aligned}$$

Then the syntax of the above cost functional is given as follows:

```
function [endpoint,integrand]=mycostfun(solcost);

Q = [5 0; 0 2];
R = [1 0; 0 3];
S = [1 5; 5 1];
iphase = solcost.phase;
t0 = solcost.initial.time;
x0 = solcost.initial.state;
tf = solcost.terminal.time;
xf = solcost.terminal.state;
t  = solcost.time;
x  = solcost.state;
u  = solcost.control;
p  = solcost.parameter;

if iphase==1,
  Mayer   = dot(xf',S*xf');
  Lagrange = dot(x,x*Q',2)+dot(u,u*R',2); % Note transposes
elseif iphase==2,
  Mayer   = dot(xf,xf);
  Lagrange = dot(u,u*R',2); % Note transposes
end;
```

It is noted in the above function call that the third argument in the command **dot** takes the dot product across the *rows*, thereby producing a *column vector*.

1.9.3 Syntax of Function Used to Evaluate Differential-Algebraic Equations

The calling syntax used evaluate the right-hand side of a user-defined vector of differential equations is given as follows:

function dae=mydaefun(soldae);

where *mydaefun.m* is the name of the MATLAB function, *soldae* is the input to the function, and *dae* is the output (i.e., the right-hand side of the differential equations and the values of the path constraints). The input *soldae* is a structure while the output *dae* is a matrix of size $N \times (n+c)$ where n is the number of differential equations, c is the number of path constraints, and N is the number of LG points. The input structure *soldae* has the following fields:

- *soldae*.**phase**: the phase number
- *soldae*.**time**: a column vector of length N that contains the time (excluding the initial and terminal points) in phase *soldae*.**phase**
- *soldae*.**state**: a matrix of size $N \times n$ (where n is the number of states) that contains the values of the state (excluding the initial and terminal points) in phase *soldae*.**phase**
- *soldae*.**control**: a matrix of size $N \times m$ (where m is the number of controls) that contains the values of the control (excluding the initial and terminal points) in phase *soldae*.**phase**
- *soldae*.**parameter**: a column vector of length q that contains the values of the static parameters in phase *soldae*.**phase**

Finally, the output of *myodefun* are as follows:

- *dae*: a *matrix* of size $N \times (n+c)$ containing the values of the right-hand side of the n differential equations and the c path constraints evaluated at the N LG points

Example of a Differential-Algebraic Equation

Suppose we have a two-phase optimal control problem that uses a differential equation function called "mydaefun.m". Suppose further that the dimension of the state in each phase is 2, the dimension of the control in each phase is 2. Furthermore, suppose that there are no path constraints in phase 1 and one path constraint in phase 2. Next, suppose that the differential equations in phase 1 are given as

$$\begin{array}{rcl} \dot{x}_1 & = & -x_1^2 - x_2^2 + u_1 u_2 \\ \dot{x}_2 & = & -x_1 x_2 + 2(u_1 + u_2) \end{array}$$

Also, suppose that the differential equations in phase 2 are given as

$$\begin{array}{rcl} \dot{x}_1 & = & \sin(x_1^2 + x_2^2) + u_1 u_2^2 \\ \dot{x}_2 & = & -\sin x_1 \cos x_2 + 2 u_1 u_2 \end{array}$$

Finally, suppose that the path constraint in phase 2 is given as

$$u_1^2 + u_2^2 = 1$$

Then a MATLAB code that will evaluate the above system of differential-algebraic equations is given as follows:

```
function dae = mydaefun(soldae);

iphase = soldae.phase;
t = soldae.time;
x = soldae.state;
```

```
u = soldae.control;
p = soldae.parameter;

if iphase==1,
  x1dot = -x(:,1).^2-x(:,2).^2 + u(:,1).*u(:,2);
  x2dot = -x(:,1).*x(:,2) + 2*(u(:,1)+u(:,2));
  path = [];
elseif iphase==2,
  x1dot = sin(x(:,1).^2 + x(:,2).^2) + u(:,1).*u(:,2).^2;
  x2dot = -sin(x(:,1)).*cos(x(:,2))+2*u(:,1).*u(:,2);
  path  = u(:,1).^2+u(:,2).^2;
end;
dae = [x1dot x2dot path];
```

1.9.4 Syntax of Function Used to Evaluate Event Constraints

The syntax used to evaluate a user-defined vector of event constraints is given as follows:

function events=myeventfun(solevents,iphase);

where *myeventfun.m* is the name of the MATLAB function, *solevents* and *iphase* are the inputs to the function, and *event* is the output (i.e., the value of the event constraints). The inputs *solevents* and *iphase* are a cell array and an integer, respectively, while the output *event* is a *column vector* of length e where e is the number of event constraints. The input cell array *solevents* has the following elements:

- *solevents*.**phase**: the phase number
- *solevents*.**initial.time**: the time at the start of the phase
- *solevents*.**initial.state**: the state at the start of the phase
- *solevents*.**terminal.time**: the time at the terminus of the phase
- *solevents*.**terminal.state**: the state at the terminus of the phase
- *solevents*.**parameter**: the static parameters in the phase

Example of Event Constraints

Suppose we have a one-phase optimal control problem that has two initial event constraints and three terminal event constraints. Suppose further that the number of states in the phase is six and that the function that computes the values of these constraints is called "myeventfun.m". Finally, let the two initial event constraints be given as

$$\begin{aligned}\phi_{01} &= x_1(t_0)^2 + x_2(t_0)^2 + x_3(t_0)^2 \\ \phi_{02} &= x_4(t_0)^2 + x_5(t_0)^2 + x_6(t_0)^2\end{aligned}$$

while the three terminal event constraints are given as

$$\begin{aligned}\phi_{f1} &= \sin(x_1(t_f))\cos(x_2(t_f) + x_3(t_f)) \\ \phi_{f2} &= \tan(x_4^2(t_f) + x_5^2(t_f) + x_6^2(t_f)) \\ \phi_{f3} &= x_4(t_f) + x_5(t_f) + x_6(t_f)\end{aligned}$$

Then the syntax of the above event function is given as

```
function events = myeventfun(solevents);

iphase = solevents.phase;
t0 = solevents.initial.time;
x0 = solevents.initial.state;
tf = solevents.terminal.time;
xf = solevents.terminal.state;

ei1 = dot(x0(1:3),x0(1:3));
ei2 = dot(x0(4:6),x0(4:6));
ef1 = sin(xf(1))*cos(xf(2)+xf(3));
ef2 = tan(dot(xf(4:6),xf(4:6)));
ef3 = xf(4)+xf(5)+xf(6);

events = [ei1;ei2;ef1;ef2;ef3];
```

Finally, it is noted that each event constraint need not be a function of either the initial or the terminal state, but can also be functions that contain *both* the initial and terminal state and/or the initial and terminal time. As an example of an event constraint that contains both the initial and terminal state, consider the following example.

Example of Event Constraint Containing Both Initial and Terminal State

Suppose we have a one-phase optimal control problem that contains only a single state. Furthermore, suppose that the problem contains a single event constraint on the *difference* between the terminal value of the state and the initial value of the state. Finally, suppose that the function that computes the values of these constraints is called "myeventfun.m". Then the event constraint is evaluated as

$$\phi = x(t_f) - x(t_0)$$

Then the syntax of the above event function is given as

```
function events = myeventfun(solevents);

t0 = solevents.initial.time;
x0 = solevents.initial.state;
tf = solevents.terminal.time;
xf = solevents.terminal.state;

events = xf-x0;
```

1.9.5 Syntax of Function Used to Evaluate Linkage Constraints

The syntax used to define the user defined vector of linkage constraints between two phases is given as follows:

function links=mylinkfun(sollink);

where *mylinkfun.m* is the name of the MATLAB function, *sollink* is the input to the function, and *links* is the output (i.e., the value of the linkage constraints). The input *sollink* is a structure while the output *links* is a *column vector* of length l, where l is the number of event constraints. The input structure *sollink* has the following fields:

- *sollink*.**left.phase**: the left phase of the pair of phases to be linked
- *sollink*.**right.phase**: the right phase of the pair of phases to be linked
- *sollink*.**left.state**: the state at the terminus of phase *sollink*.**left.phase**
- *sollink*.**right.state**: the state at the start of phase *sollink*.**right.phase**
- *sollink*.**left.parameter**: the static parameters in phase *sollink*.**left.phase**
- *sollink*.**right.state**: the static parameters in phase *sollink*.**right.phase**

The terms *left* and *right* are conventions adopted to help the user orient the phases on a page from left to right.

Example of Linkage Constraint

Suppose we have a multiple phase optimal control problem with a simple link between the phases, i.e. the state of the end of the phase is equal to the state at the beginning of the next phase.

$$\mathbf{P} = x^l(t_f) - x^r(t_0)$$

Then the syntax of the above linkage is given as

```
function links = mylinkagefun(sollink);

left_phase = sollink.left.phase;
right_phase = sollink.right.phase;
xf_left = sollink.left.state;
p_left  = sollink.left.parameter;
x0_left = sollink.right.phase;
p_left  = sollink.right.parameter;

links = xf_left - x0_right;
```

1.10 Specifying an Initial Guess of The Solution

The field **guess** of the user-defined structure *setup* contains the initial guess for the problem. The field **guess** is an array of structures of length P (where P is the number of phases in the problem). The p^{th} element of the array of structures **guess** contains the initial guess of the problem in phase $p \in [1, \ldots, P]$. The fields of each element of array of structures **guess** are given as follows:

- **guess(*p*).time**: a *column* vector of length s where s is the number of time points used in the guess
- **guess(*p*).state**: a matrix of size $s \times n$ where s is the number of time points and n is the number of states in the phase
- **guess(*p*).control**: a matrix of size $s \times m$ where s is the number of time points and m is the number of controls in the phase
- **guess(*p*).parameter**: a column vector of length q where q is the number of static parameters in the phase

It is noted that the element **guess(*p*).time** must be monotonic and in the same direction as that specified by the field **direction** of the structure *setup*. Schematically, in each phase of the problem the guess for the time, states, controls, and parameters is structured as follows:

$$\textbf{guess}(p)\textbf{.time} = \begin{bmatrix} t_0 \\ t_1 \\ t_2 \\ \cdots \\ t_s \end{bmatrix}$$

$$\textbf{guess}(p)\textbf{.state} = \begin{bmatrix} x_{10} & x_{20} & \cdots & x_{n0} \\ x_{11} & x_{21} & \cdots & x_{n1} \\ \vdots & \vdots & \vdots & \vdots \\ x_{1s} & x_{2s} & \cdots & x_{ns} \end{bmatrix}$$

$$\textbf{guess}(p)\textbf{.control} = \begin{bmatrix} u_{10} & x_{20} & \cdots & x_{m0} \\ u_{11} & u_{21} & \cdots & x_{m1} \\ \vdots & \vdots & \vdots & \vdots \\ u_{1s} & u_{2s} & \cdots & u_{ms} \end{bmatrix}$$

$$\textbf{guess}(p)\textbf{.parameter} = \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_q \end{bmatrix}$$

Example of Specifying an Initial Guess

Suppose we have a two-phase problem that has three states and two controls in phase 1 while it has two states and one control in phase 2. Furthermore, suppose that we choose five time points for the guess in phase 1 while we choose 3 time points for the guess in phase 2. A MATLAB code that would create such an initial guess is given below.

```
iphase = 1;
guess(iphase).time  = [0; 1; 3; 5; 7];
guess(iphase).state(:,1) = [1.27; 3.1; 5.8; 9.6; -13.7272];
guess(iphase).state(:,2) = [-4.2; -9.6; 8.5; 25.73; 100.00];
guess(iphase).state(:,3) = [18.727; 1.827; 25.272; -14.272; 26.84];
guess(iphase).control(:,1) = [8.4; -13.7; -26.5; 19; 87];
guess(iphase).control(:,2) = [-1.2; 5.8; -3.77; 14; 19.787];
guess(iphase).parameter = [];

iphase = 2;
guess(iphase).time = [7; 7.5; 8];
guess(iphase).state(:,1) = [0.5; 1.5; 8];
guess(iphase).state(:,2) = [-0.5; -2.5; 19];
guess(iphase).control(:,1) = [8.4; -13.7; -26.5; 19; 87];
guess(iphase).parameter = [];

setup.guess = guess;
```

It is noted again that, for the above example, auxiliary integer variables were used to minimize the cumbersomeness of coding and to minimize the chance of error.

1.11 Scaling of Optimal Control Problem

As with any numerical optimization procedure, a well-scaled optimal control problem is required. In general, it is recommended that the user scale the problem in accordance with any known large discrepancies either in the sizes of various quantities (i.e., state, control) or the sizes of the derivatives of such quantities. While it is beyond the scope of this user's manual to provide a general procedure for scaling, in an attempt to reduce the burden on the user an automatic scaling procedure has been developed. This procedure is based on the scaling algorithm developed in (**?**). In order to invoke the automatic scaling routine, the user must set the field **autoscale** in the user-defined structure *setup* to the string "on".

The automatic scaling procedure operates as follows. The bounds on the variables are used to scale all components of the state, control, parameters, and time to lie between -1 and 1. As a result, it is essential that the user provide *sensible* bounds on all quantities (e.g., do not provide unreasonably large bounds as this will result in a poorly scaled problem). Next, the constraints are scaled to make the row norms of the Jacobians of the respective functions approximately unity. The automatic scaling procedure is by no means foolproof, but it has been found in practice to work well on many problems that otherwise would require scaling by hand. The advice given here is to try the automatic scaling procedure, but not to use it for too long if it is proving to be unsuccessful.

1.12 Different Options for Specification of Derivatives

The user has six choices for the computation of the derivatives of the objective function gradient and the constraint Jacobian for use within the NLP solver. As stated above, the choices for **derivatives** are "numerical", "complex", "automatic", and "analytic" and correspond to the following differentiation methods:

- *setup*.**derivatives**="numerical": default finite-differencing algorithm within SNOPT is used.
- *setup*.**derivatives**="complex": the *built-in* complex-step differentiation method is used.
- *setup*.**derivatives**="automatic", the *built-in* automatic differentiator is used.
- *setup*.**derivatives**="analytic": analytic derivatives (supplied by the user) are used.

1.12.1 Complex-Step Differentiation

Of the differentiation methods given above, either the built-in automatic differentiator or the complex-step differentiator most preferred because these two methods provide highly accurate derivatives and are both included (i.e., the user does not have to obtain any third-party software). One drawback with complex-step differentiation, however, is that certain functions need to be handled with great care. In particular, the functions **min**, **max**, **abs**, and **dot** need to be redefined for use in complex-step differentiation (see Ref. **?** and the URL `http://mdolab.utias.utoronto.ca/resources/complex-step/complexify.f90` for details). Finally, the transpose operator must be replaced with a dot-transpose (i.e., a *real transpose*) because the standard transpose in MATLAB produces a complex conjugate transpose and it is necessary to maintain a real transpose when computing derivatives via complex-step differentiation.

1.12.2 Analytic Differentiation

Analytic differentiation has the advantage that it is the fasted and most accurate of the four methods, however, it is by far the most complex for the user to compute, code, and verify. The derivatives for the objective function gradient and the constraint Jacobian are computed from the user defined analytic derivatives. These derivatives are supplied as an additional output of the user functions for the cost, dae functions, event constraints, and linkage constraints (if applicable). The user defined derivatives can be checked relative to a finite-difference approximation by setting the flag *setup*.**checkDerivatives** equal to one. Upon execution, the derivatives will be computed at the user supplied initial guess using a finite-difference approximation and compared to the analytic derivatives with the results printed to the screen. It is recommended that the user run the derivative checking algorithm a least one time to verify that the derivatives are correct, however, it should be noted that the algorithm is not guaranteed to find any incorrect derivatives. The user must take special care to ensure that the analytic derivatives are coded correctly in order to take advantage of the speed and accuracy of analytic differentiation.

Syntax of Function Used to Evaluate Cost with Derivatives

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

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

See Section 1.9.1 for the definition of the regular inputs/outputs. The additional outputs of *mycostfun* are as follows:

- *DerivMayer*: a *row* vector of size $1 \times (2n+2+q)$
- *DerivLagrange*: a *matrix* of size $N \times (n+m+q+1)$

where n is the number of states, m is the number of controls, q is the number of parameters, and N is the number of LG points in the phase. The row vector *DerivMayer* defines the partial derivatives of the Mayer cost with respect to the initial state, initial time, final state, final time, and finally the parameters:

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

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

$$\textbf{DerivLagrange} = \left[\frac{\partial\mathcal{L}}{\partial\mathbf{x}}, \frac{\partial\mathcal{L}}{\partial\mathbf{u}}, \frac{\partial\mathcal{L}}{\partial p}, \frac{\partial\mathcal{L}}{\partial t}\right]$$

It is important to provide all the derivatives in the correct order *even if* they are zero.

Example of a Cost Functional with Derivatives

Suppose we have a two-phase optimal control problem that uses a cost functional named "mycostfun.m". Suppose further that the dimension of the state in each phase is 2 while the dimension of the control in each phase is 2. Also, suppose that the endpoint and integrand cost in phase 1 are given, respectively, as

$$\begin{aligned}\Phi^{(1)}(\mathbf{x}^{(1)}(t_0), t_0^{(1)}, \mathbf{x}^{(1)}(t_f), t_f^{(1)}) &= \mathbf{x}^T(t_f)\mathbf{S}\mathbf{x}(t_f) \\ \mathcal{L}^{(1)}(\mathbf{x}^{(1)}(t), \mathbf{u}^{(1)}(t), t) &= \mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{u}^T\mathbf{R}\mathbf{u}\end{aligned}$$

while the endpoint and integrand in phase 2 are given, respectively, as

$$\begin{aligned}\Phi^{(2)}(\mathbf{x}^{(2)}(t_0^{(2)}), t_0^{(2)}, \mathbf{x}^{(2)}(t_f^{(2)}), t_f^{(2)}) &= \mathbf{x}^T(t_f)\mathbf{x}(t_f) \\ \mathcal{L}^{(2)}(\mathbf{x}^{(2)}(t), \mathbf{u}^{(2)}(t), t) &= \mathbf{u}^T\mathbf{R}\mathbf{u}\end{aligned}$$

Then the syntax of the above cost functional is given as follows:

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

Q = [5 0; 0 2];
R = [1 0; 0 3];
S = [1 5; 5 1];
t0 = solcost.initial.time;
x0 = solcost.initial.state;
tf = solcost.terminal.time;
xf = solcost.terminal.state;
t  = solcost.time;
```

```
x   = solcost.state;
u   = solcost.control;
p   = solcost.parameter;

if iphase==1,
  Mayer   = dot(xf',S*xf');
  Lagrange = dot(x,x*Q',2)+dot(u,u*R',2); % Note transposes
  DerivMayer = [zeros(1,length(x0)), zeros(1,length(t0)), ...
                xf'*S, zeros(1,length(tf), zeros(1,length(p))];
  DerivLagrange = [x*Q', u*R', ...
                   zeros(length(t),length(p)), zeros(size(t))];
elseif iphase==2,
  Mayer   = dot(xf,xf);
  Lagrange = dot(u,u*R',2); % Note transposes
  DerivMayer = [zeros(1,length(x0)), zeros(1,length(t0)), ...
                xf', zeros(1,length(tf), zeros(1,length(p))];
  DerivLagrange = [zeros(size(x)), u*R', ...
                   zeros(length(t),length(p)), zeros(size(t))];
end;
```

It is noted in the above function call that the third argument in the command **dot** takes the dot product across the *rows*, thereby producing a *column vector*.

Syntax of Function Used to Evaluate Differential-Algebraic Equations with Derivatives

The calling syntax used evaluate the derivatives of the right-hand side of a user-defined vector of differential equations is given as follows:

function [dae,Derivdae]=mydaefun(soldae);

See Section 1.9.3 for the definition of the regular inputs/outputs. The additional output of *myodefun* is as follows:

- *Derivdae*: a *matrix* of size $N(n+c) \times (n+m+q+1)$

where n is the number of states, m is the number of controls, q is the number of parameters, c is the number of path constraints, and N is the number of LG points in the phase. The matrix *Derivdae* defines the partial derivatives of the differential equations and path constraints with respect to the state, control, parameters, and time at each of the N LG points:

$$\textbf{Derivdae} = \begin{bmatrix} \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}}, & \frac{\partial \mathbf{f}_1}{\partial \mathbf{u}}, & \frac{\partial \mathbf{f}_1}{\partial p}, & \frac{\partial \mathbf{f}_1}{\partial t} \\ \vdots, & \vdots, & \vdots, & \vdots \\ \frac{\partial \mathbf{f}_n}{\partial \mathbf{x}}, & \frac{\partial \mathbf{f}_n}{\partial \mathbf{u}}, & \frac{\partial \mathbf{f}_n}{\partial p}, & \frac{\partial \mathbf{f}_n}{\partial t} \\ \frac{\partial \mathbf{C}_1}{\partial \mathbf{x}}, & \frac{\partial \mathbf{C}_1}{\partial \mathbf{u}}, & \frac{\partial \mathbf{C}_1}{\partial p}, & \frac{\partial \mathbf{C}_1}{\partial t} \\ \vdots, & \vdots, & \vdots, & \vdots \\ \frac{\partial \mathbf{C}_r}{\partial \mathbf{x}}, & \frac{\partial \mathbf{C}_r}{\partial \mathbf{u}}, & \frac{\partial \mathbf{C}_r}{\partial p}, & \frac{\partial \mathbf{C}_r}{\partial t} \end{bmatrix}$$

where $\mathbf{f}_i$, $(i = 1, \ldots, n)$ is the right-hand side of the i^{th} differential equation, and $\mathbf{C}_j$, $(j = 1, \ldots, r)$ is the j^{th} path constraint. It is important to provide all the derivatives in the correct order *even if* they are zero.

Example of a Differential-Algebraic Equation with Derivatives

Suppose we have a two-phase optimal control problem that uses a differential equation function called “mydaefun.m”. Suppose further that the dimension of the state in each phase is 2, the dimension of the control in each phase is 2. Furthermore, suppose that there are no path constraints in phase 1 and one path constraint in phase 2. Next, suppose that the differential equations in phase 1 are given as

$$\begin{aligned} \dot{x}_1 &= -x_1^2 - x_2^2 + u_1 u_2 \\ \dot{x}_2 &= -x_1 x_2 + 2(u_1 + u_2) \end{aligned}$$

Also, suppose that the differential equations in phase 2 are given as

$$\begin{aligned} \dot{x}_1 &= \sin(x_1^2 + x_2^2) + u_1 u_2^2 \\ \dot{x}_2 &= -\sin x_1 \cos x_2 + 2 u_1 u_2 \end{aligned}$$

Finally, suppose that the path constraint in phase 2 is given as

$$u_1^2 + u_2^2 = 1$$

Then a MATLAB code that will evaluate the above system of differential-algebraic equations is given as follows:

```
function [dae, Derivdae] = mydaefun(soldae);

iphase = soldae.phase;
t = soldae.time;
x = soldae.state;
u = soldae.control;
p = soldae.parameter;

if iphase==1,
  x1dot = -x(:,1).^2-x(:,2).^2 + u(:,1).*u(:,2);
  x2dot = -x(:,1).*x(:,2) + 2*(u(:,1)+u(:,2));
  path = [];
  df1_dx1 = -2*x(:,1);
  df1_dx2 = -2*x(:,2);
  df1_du1 = u(:,2);
  df1_du2 = u(:,1);
  df2_dx1 = -x(:,2);
  df2_dx2 = -x(:,1);
  df2_du1 = 2*ones(size(t));
  df2_du2 = 2*ones(size(t));
  dpath_dx1 = [];
  dpath_dx2 = [];
  dpath_du1 = [];
  dpath_du2 = [];
  dpath_dp = [];
  dpath_dt = [];
elseif iphase==2,
  x1dot = sin(x(:,1).^2 + x(:,2).^2) + u(:,1).*u(:,2).^2;
  x2dot = -sin(x(:,1)).*cos(x(:,2)) + 2*u(:,1).*u(:,2);
  path  = u(:,1).^2+u(:,2).^2;
  df1_dx1 = 2*x(:,1)*cos(x(:,1).^2 + x(:,2).^2);
  df1_dx2 = 2*x(:,2)*cos(x(:,1).^2 + x(:,2).^2);
  df1_du1 = u(:,2).^2;
  df1_du2 = 2*u(:,1).*u(:,2);
  df2_dx1 = -cos(x(:,1)).*cos(x(:,2));
  df2_dx2 = sin(x(:,1)).*sin(x(:,2));
  df2_du1 = 2*u(:,2);
  df2_du2 = 2*u(:,1);
  dpath_dx1 = zeros(size(x(:,1)));
  dpath_dx2 = zeros(size(x(:,2)));
  dpath_du1 = 2*u(:,1);
  dpath_du2 = 2*u(:,2);
  dpath_dp = zeros(length(t),length(p));
  dpath_dt = zeros(size(t));
end;
df1_dp = zeros(length(t),length(p));
df1_dt = zeros(size(t));
df2_dp = zeros(length(t),length(p));
df2_dt = zeros(size(t));

dae = [x1dot x2dot path];

Derivdae = [df1_dx1,   df1_dx2,   df1_du1,   df1_du2,   df1_dp,   df1_dt; ...
            df2_dx1,   df2_dx2,   df2_du1,   df2_du2,   df2_dp,   df2_dt; ...
          dpath_dx1, dpath_dx2, dpath_du1, dpath_du2, dpath_dp, dpath_dt];
```

Syntax of Function Used to Evaluate Event Constraints with Derivatives

The syntax used to evaluate the derivative of a user-defined vector of event constraints is given as follows:

function [events, Derivevents]=myeventfun(solevents);

See Section 1.9.4 for the definition of the regular inputs/outputs. The additional output of *myeventfun* is as follows:

- *Derivevents*: a *matrix* of size $e \times (2n + 2 + q)$

where n is the number of states, q is the number of parameters, and e is the number of event constraints in the phase. The matrix *Derivevents* defines the partial derivatives of each event constraint with respect to the initial state, initial time, final state, final time, and parameters:

$$\textbf{Derivevents} = \begin{bmatrix} \dfrac{\partial \phi_1}{\partial \mathbf{x}(t_0)}, & \dfrac{\partial \phi_1}{\partial t_0}, & \dfrac{\partial \phi_1}{\partial \mathbf{x}(t_f)}, & \dfrac{\partial \phi_1}{\partial t_f}, & \dfrac{\partial \phi_1}{\partial p} \\ \vdots, & \vdots, & \vdots, & \vdots, & \vdots \\ \dfrac{\partial \phi_e}{\partial \mathbf{x}(t_0)}, & \dfrac{\partial \phi_e}{\partial t_0}, & \dfrac{\partial \phi_e}{\partial \mathbf{x}(t_f)}, & \dfrac{\partial \phi_e}{\partial t_f}, & \dfrac{\partial \phi_e}{\partial p} \end{bmatrix}$$

where ϕ_i, $(i = 1, \ldots, e)$ is the i^{th} event constraint. It is important to provide all the derivatives in the correct order *even if* they are zero.

Example of Event Constraints with Derivatives

Suppose we have a one-phase optimal control problem that has two initial event constraints and three terminal event constraints. Suppose further that the number of states in the phase is six and that the function that computes the values of these constraints is called "myeventfun.m". Finally, let the two initial event constraints be given as

$$\begin{aligned}\phi_{01} &= x_1(t_0)^2 + x_2(t_0)^2 + x_3(t_0)^2 \\ \phi_{02} &= x_4(t_0)^2 + x_5(t_0)^2 + x_6(t_0)^2\end{aligned}$$

while the three terminal event constraints are given as

$$\begin{aligned}\phi_{f1} &= \sin(x_1(t_f))\cos(x_2(t_f) + x_3(t_f)) \\ \phi_{f2} &= \tan(x_4^2(t_f) + x_5^2(t_f) + x_6^2(t_f)) \\ \phi_{f3} &= x_4(t_f) + x_5(t_f) + x_6(t_f)\end{aligned}$$

Then the syntax of the above event function is given as

```
function [events, Derivevents] = myeventfun(solevents);

iphase = solevents.phase;
t0 = solevents.initial.time;
x0 = solevents.initial.state;
tf = solevents.terminal.time;
xf = solevents.terminal.state;

ei1 = dot(x0(1:3),x0(1:3));
ei2 = dot(x0(4:6),x0(4:6));
ef1 = sin(xf(1))*cos(xf(2)+xf(3));
ef2 = tan(dot(xf(4:6),xf(4:6)));
ef3 = xf(4)+xf(5)+xf(6);

events = [ei1;ei2;ef1;ef2;ef3];

dei1_dx0 = [2*x0(1:3).' zeros(1,3)];
dei1_dt0 = 0;
dei1_dxf = zeros(1,6);
dei1_dtf = 0;
dei1_dp = [];
dei1_dt = 0;
dei2_dx0 = [zeros(1,3), 2*x0(4:6).'];
dei2_dt0 = 0;
dei2_dxf = zeros(1,6);
dei2_dtf = 0;
dei2_dp = [];
def1_dx0 = zeros(1,6);
def1_dt0 = 0;
def1_dxf = [cos(xf(1))*cos(xf(2)+xf(3)), -sin(xf(1))*sin(xf(2)+xf(3)), ...
           -sin(xf(1))*sin(xf(2)+xf(3)), zeros(1,3)];
def1_dtf = 0;
def1_dp = [];
def2_dx0 = zeros(1,6);
def2_dt0 = 0;
def2_dxf = [zeros(1,3), 2*xf(4:6).']/(cos(dot(xf(4:6),xf(4:6))))^2;
def2_dtf = 0;
def2_dp = [];
def3_dx0 = zeros(1,6);
def3_dt0 = 0;
def3_dxf = [zeros(1,3), ones(1,3)];
def3_dtf = 0;
```

```
def3_dp = [];

Derivevents = [dei1_dx0, dei1_dt0, dei1_dxf, dei1_dtf, dei1_dp, dei1_dt; ...
               dei2_dx0, dei2_dt0, dei1_dxf, dei2_dtf, dei2_dp, dei2_dt; ...
               def1_dx0, def1_dt0, def1_dxf, def1_dtf, def1_dp, def1_dt; ...
               def2_dx0, def2_dt0, def2_dxf, def2_dtf, def2_dp, def2_dt; ...
               def3_dx0, def3_dt0, def3_dxf, def3_dtf, def3_dp, def3_dt];
```

Syntax of Function Used to Evaluate Linkage Constraints with Derivatives

The syntax used to define the user defined vector of linkage constraints between two phases is given as follows:

function [links,Derivlinks]=mylinkfun(sollink);

See Section 1.9.5 for the definition of the regular inputs/outputs. The additional output of *mylinkfun* is as follows:

- *Derivlinks*: a *matrix* of size $l \times (n^l + q^l + n^r + q^r)$

where l is the number of linkages in the constraint, n^l is the number of states in the left phase, q^l is the number of parameters in the left phase, n^r is the number of states in the right phase, and q^r is the number of parameters in the right phase. The matrix *Derivlinks* defines the partial derivatives of each linkage with respect to the left state, left parameters, right state, and right parameters:

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

where $\mathbf{P}_i$, $(i = 1, \ldots, l)$ is the i^{th} linkage constraint. It is important to provide all the derivatives in the correct order *even if* they are zero.

Example of Linkage Constraint with Derivatives

Suppose we have a multiple phase optimal control problem with a simple link between the phases, i.e. the state of the end of the phase is equal to the state at the beginning of the next phase.

$$\mathbf{P} = x^l(t_f) - x^r(t_0)$$

Then the syntax of the above linkage is given as

```
function [links, Derivlinks] = mylinkagefun(sollink,left_phase,right_phase);

xf_left = sollink.left.state;
p_left  = sollink.left.parameter;
x0_right = sollink.right.state;
p_right  = sollink.right.parameter;

links = xf_left - x0_right;

nlink = length(xf_left); %number of linkages
Derivlinks = [ eye(nlink), zeros(nlink,length(p_left)), ...
              -eye(nlink), zeros(nlink,length(p_right))];
```

1.13 Output of Execution

Upon execution of the software, new fields are created in the output structure *output*. In particular, upon completion of the execution, the following new fields are created (in addition to the fields that were created prior to running the software on the problem):

- **solution:** an array of structures of length P (where P is the number of phases) containing the solution in each phase

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

- **solution(*p*).time:** a column vector of size $M \times 1$ containing the time at each point along the trajectory (where $M = N + 2$ is the number of time points and N is the number of LG points)
- **solution(*p*).state:** a matrix of size $M \times n$ such that the rows contain the state at the time points along the trajectory
- **solution(*p*).control:** a matrix of size $M \times m$ such that the rows contain the state at the time points along the trajectory
- **solution(*p*).parameter:** a column vector of length q containing the static parameters
- **solution(*p*).costate:** a matrix of size $M \times n$ such that the rows contain the costate at each time point along the trajectory
- **solution(*p*).pathmult:** a structure containing the Lagrange multipliers of the path constraints
- **solution(*p*).Hamiltonian:** a column vector of size $M \times 1$ that contains the Hamiltonian at each time point along the trajectory
- **solution(*p*).Mayer_cost:** The Mayer part of the cost along the trajectory
- **solution(*p*).Lagrange_cost:** The Lagrange (integrated) cost along the trajectory