A Beginner's Guide to DIDO

(Ver. 7.3)

A MATLAB© Application Package for Solving Optimal Control Problems [*]

I. Michael Ross

Prepared For
Elissar, LLC

Document # TR-711
Elissar, LLC
P.O. Box 1365
Monterey, CA 93942

[*]For additional information, see "DIDO User's Manual for Solving Hybrid Optimal Control Problems," to be released.

Acknowledgments

The first version of DIDO was in 1998. The first object-oriented version of DIDO was in 2001. Given its long history, I am indebted to a very large number of people who helped make DIDO such a successful code that this section is really not as exhaustive as it ought to be. As a result, I will restrict my thanks to the most recent people who are responsible for the success of DIDO.

First and foremost, I am particularly grateful to my friend and colleague, Fariba Fahroo, Professor of Applied Mathematics at the Naval Postgraduate School who introduced me to the wonderful world of pseudospectral (PS) methods in 1997. Our research collaboration continues to inspire me in many ways.

To Chris D'Souza and Jeremy Rea who were early advocates of PS methods. Chris showed me how to efficiently solve hybrid optimal control problems while Jeremy opened my eyes to looking for "hidden" singularities within DIDO constructs. This also led me to a deeper appreciation of mathematical optimal control theory.

I would also like to thank my many current and former students at the Naval Postgraduate School who continue to amaze me by solving complex optimal control problems. To Scott Josselyn and Pat Croley who showed me how to write efficient DIDO files and effective ways to scale badly scaled problems.

To Pooya Sekhavat, who can work wonders with DIDO. Despite that I knew how important scaling and balancing equations are to effective numerical computations, Pooya showed me that they are even more important than I could possibly imagine. The current manual pays homage to this extreme importance of this critical step in running DIDO efficiently.

To Qi Gong, who helped me better understand the various intricacies of PS methods and their relationship to control theory. His counter examples alone are a testament to his genius.

To Walter Murray, who's enormous time and patience helped me better understand and appreciate the power of nonlinear programming theory, particularly the intricacies involved in sequential-quadratic programming methods, interior point methods, warm-start techniques, and a whole host of interesting numerical mathematics.

To the AA 4850 class of 2001 who enthusiastically supported the "dense" version of DIDO and provided suggestions for modifications that have become ever so useful to this day.

THIS PAGE INTENTIONALLY LEFT ALMOST BLANK

Table of Contents

1 Introduction and Overview

DIDO[*] is a minimalist's approach to solving complex optimal control problems. That is, *only the problem formulation is required as an input to DIDO, in a manner that is nearly identical to writing it on a piece of paper*. This unique pencil-and-paper-like approach of coding DIDO files is made possible through the use of DIDO expressions. The DIDO expressions are constructed in a manner that makes optimal control programming intuitive. *The input to DIDO is as simple or complicated as only the problem formulation*.

The current version of DIDO (7.x) is backward compatible all the way back to DIDO 2001 (α) – the first, object-oriented, public version released in the Summer/Fall of 2001 [3]. We recommend new users use DIDO in the format described in this manual. DIDO users may continue to use DIDO in the pre-2005 format; however, this format will no longer be supported after 2011. We urge old users to shift to this format as the differences are not substantial. The current and forthcoming versions of DIDO are substantially more robust and faster than all prior versions.

DIDO is a complete package. No other third-party software is required other than MATLAB.

DIDO is capable of solving a broad class of *Smooth and Nonsmooth Hybrid Optimal Control* problems defined over a time[†] interval $[t_0, t_f]$ that may be fixed or free. DIDO can also solve *Dynamic Optimization* problems that may not be optimal control problems. In this context, an optimal control problem is a dynamic optimization problem parameterized by control variables[4, 5].

Although DIDO's capabilities are very broad, for the purposes of this introductory section, we describe a simpler problem to illustrate some of its main features.

Consider an optimal control problem with an interior point constraint; that is, a constraint at t_e where $t_0 < t_e < t_f$. With deceptive simplicity, the problem may be posed as follows: Determine the state-control function-pair, $[t_0, t_f] \mapsto \{\mathbf{x} \in \mathbb{R}^{N_x}, \mathbf{u} \in \mathbb{R}^{N_u}\}$, the event time t_e and parameters $\mathbf{p} \in \mathbb{R}^{N_p}$ that minimize the Bolza cost functional,

$$J[\mathbf{x}(\cdot), \mathbf{u}(\cdot); t_e; \mathbf{p}] = E(\gamma(\mathbf{x}(\cdot)); \Gamma; \mathbf{p}) + \int_{t_0}^{t_f} F(\mathbf{x}(t), \mathbf{u}(t); t; \mathbf{p})\, dt \tag{1.1}$$

subject to the constraints,

$$\dot{\mathbf{x}} \in \mathbb{F}(\mathbf{x}, \mathbf{u}, t; \mathbf{p}) \quad \mathbf{x} \in \mathbb{X}(t; \mathbf{p}), \quad \mathbf{u} \in \mathbb{U}(t, \mathbf{x}; \mathbf{p}), \quad \mathbf{p} \in \mathbb{P}, \quad \mathbf{x}(\Gamma) \in \mathbb{E} \tag{1.2}$$

where γ is a restriction operator over $\Gamma = \{t_0, t_e, t_f\}$ and $\mathbb{F}, \mathbb{X}, \mathbb{U}, \mathbb{P}, \mathbb{H}$ and $\mathbb{E}$ are constraint sets described by an intersection of functional constraints given in the form of inequalities and equalities. See Section 4 for an illustrative example.

[*]Contrary to popular belief, DIDO is not an acronym. It is named after Queen Dido of Carthage (circa 850BC) who was the first person to solve a dynamic optimization problem now referred to as an isoperimetric problem; see Refs.[1] and [2].

[†]The independent variable need not be time.

The generality of the problem posed allows for fairly complex interior point constraints, pre-defined segments, differentially-flat segments, transition conditions, "mid-flight" changes in dynamics, multi-dynamical systems, mid-flight changes in the cost function, switches, discrete events and a host of other possibilities. Design and "inverse design" problems are also included. Such complex problems arise naturally in many practical problems[6, 7, 8]; see Refs.[9, 10] and [11] for a perspective on solving such problems.

The basic idea behind the solution method is an adaptive spectral algorithm based on a pseudospectral approximation theory[12, 13, 14, 15, 16, 17, 18, 19]. The pseudospectral approach is significantly different from non-pseudospectral methods used to solve such problems and hence the code is a realization of a fundamentally different way of rapidly solving dynamic optimization problems. Currently, DIDO implements approximations of state and control functions in Sobolev-Hilbert spaces[17] and employs a spectral algorithm based on an active-set sequential quadratic programming method[20, 21, 22] tailored for pseudospectral methods.

How to use DIDO is best illustrated by first considering a "smooth" problem that we call the basic problem defined in Section 4. It is important to note that although DIDO does not require input files corresponding to the necessary conditions of optimality, *it is extremely important for the user to develop these conditions and use them as necessary conditions were meant to be used; namely, as extremality tests on optimality for candidate optimal solutions.* These tests, reviewed in Section 8, are also substantially useful for debugging complex codes based on practical problems. Failure to perform these tests may result in endless frustration.

The correct way to use DIDO is as a tool for generating candidate solutions to optimal control problems. Do not use DIDO as if it is some "direct" method. Both the Pontryagin and Bellman tests can be easily performed for the candidate optimal solution generated by DIDO. See Refs.[6], [10] and [23].

2 Installing and Using DIDO

You must have a legal license to use either the free or the professional version of DIDO. To obtain a legally licensed copy of DIDO email a request to contact@elissar.biz. Assuming you have a legally licensed copy of DIDO, please read the following items before installing and using the code:

2.1 Brief Notes

1. *You must delete all prior versions of DIDO to ensure a successful run.*
2. It is illegal for you to distribute or modify the DIDO code. Normally, you will get a p-code of the DIDO software. You may not reverse-engineer the p-code. You may not decompile, or de-code any parts of DIDO. Please read the README file that came with the DIDO package. Should you need additional utilities, please contact Elissar, LLC at contact@elissar.biz.

2.2 Please read the README file in the DIDO folder

2.3 Installing DIDO

Erase all copies and prior versions of DIDO If you do not do this, the code may not work properly.

To install DIDO, simply copy the folder, DIDO_7.x, to your directory of choice. Follow the instructions given in the README file.

2.4 Testing the Installation

At the MATLAB prompt, >>, type `TestDIDO`. If you get no error messages, DIDO is quite likely installed correctly.

You may perform a further test by typing (at the MATLAB prompt), `LanderProblem`. If this generates no error messages, DIDO is probably installed correctly.

2.5 Limitations on the Free Version of DIDO

The free version of DIDO is limited to fewer than:

- 4 state variables,
- 2 control variables,
- 30 node points, and
- 3 nonlinear path constraints.

Note also that you may not use the free version for profit or commercial venture.

3 What's New in DIDO 7.3

1. There is no need to specify a guess to initiate DIDO's algorithm. In its default mode, DIDO can run without a guess. The user may over ride DIDO's self-chosen starting point by specifying a potentially better starting point via the structure `algorithm` through its field, **guess**.

2. A user may opt for different algorithm modes via the structure `algorithm` through its field, **mode**. In its default mode DIDO will run under its nominal mode. The solution obtained under this mode will be quite accurate for most applications. For certain problems, the accuracy of the solution may not improve as the number of nodes are increased. In this case, the user may specify that DIDO run in its accurate mode. The run time in this mode will be higher, but can generate a more accurate solution.

Users who have been using DIDO since the year 2001 may also use DIDO 7.3 and all its new features; however, those DIDO users (i.e. pre-2005) who have not migrated to the simpler DIDO syntax may make avail of the new features of DIDO by telling DIDO through its problem structure that they are using the older format. Thus, all pre-2005 DIDO codes (including those compatible with DIDO PR1) will run under DIDO 7.3 provided that the user adds the following line before calling DIDO 7.3:

$$\texttt{problem}.\mathbf{format} = \ '2001'$$

where `problem` *is the name of your problem*. For more specific information on DIDO's backward compatibility, please see the folder OldLanderFiles in the ExampleProblems folder under the ForUser folder.

4 The Basic Problem

Let $\mathbf{x} \in \mathbb{R}^{N_x}$ and $\mathbf{u} \in \mathbb{R}^{N_u}$ where $N_x, N_u \in \mathbb{N}$. Here and through out the rest of the manual, we will use the notation $N_{(\cdot)}$ to mean an element of the set, $\mathbb{N}$, of Natural numbers, $1, 2, 3 \ldots$. We define the basic problem as follows: Determine the state-control function-pair, $\{\mathbf{x}(\cdot), \mathbf{u}(\cdot)\}$, and possibly the clock times or event times[13], t_0 and t_f that minimize the Bolza ***cost functional***,

$$J[\mathbf{x}(\cdot), \mathbf{u}(\cdot), t_0, t_f] = E(\mathbf{x}(t_0), \mathbf{x}(t_f), t_0, t_f) + \int_{t_0}^{t_f} F(\mathbf{x}(t), \mathbf{u}(t), t)dt \tag{4.1}$$

subject to the ***dynamic constraints***,

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) \tag{4.2}$$

endpoint- or ***event constraints***,

$$\mathbf{e}^L \leq \mathbf{e}\big(\mathbf{x}(t_0), \mathbf{x}(t_f), t_0, t_f\big) \leq \mathbf{e}^U \tag{4.3}$$

mixed state-control ***path constraints***,

$$\mathbf{h}^L \leq \mathbf{h}(\mathbf{x}(t), \mathbf{u}(t), t) \leq \mathbf{h}^U \tag{4.4}$$

and ***box constraints*** on the states, controls and clock times,

$$\begin{aligned} \mathbf{x}^L \leq &\quad \mathbf{x}(t) &\leq \mathbf{x}^U \quad (4.5)\\ \mathbf{u}^L \leq &\quad \mathbf{u}(t) &\leq \mathbf{u}^U \quad (4.6)\\ t_0^L \leq &\quad t_0 &\leq t_0^U \quad (4.7)\\ t_f^L \leq &\quad t_f &\leq t_f^U \quad (4.8) \end{aligned}$$

By $\mathbf{v} \geq \mathbf{0}$ for any generic vector, $\mathbf{v}$, it is meant that all components of $\mathbf{v}$ are nonnegative (i.e. $\mathbf{v} \succeq 0$ in strict mathematical notation).

Note that, theoretically, the box constraints on the state and control variables may be listed as part of the state-control path constraints given by the $\mathbf{h}$ function, while the box constraints on the clock time may be listed as part of the endpoint constraints given by the $\mathbf{e}$ function; however, from a numerical perspective, it is far more efficient to separate the box constraints from the functional ones. Furthermore, from a theoretical point of view, the box constraints ensure that $\mathbf{x}(t) \in \mathbb{X}$ and $\mathbf{u}(t) \in \mathbb{U}$ where $\mathbb{X} \subseteq \mathbb{R}^{N_x}$ and $\mathbb{U} \subseteq \mathbb{R}^{N_u}$ are compact sets that may be construed as the closure of the state and control spaces respectively.

An equality constraint may be obtained by simply setting the lower bound equal to the upper bound. *If a problem does not have a finite bound, do not set to $-\infty$ for the lower bound and $+\infty$ for the upper bound using `Inf` in MATLAB; it is strongly advised that the user use some relatively large number instead of `Inf` for a trouble-free run.* See Section 10. It will be apparent later why we call the endpoint constraints, Eq.(4.3), events. In many cases, these conditions are split as

$$\mathbf{e}_0^L \le \mathbf{e}_0\big(\mathbf{x}(t_0), t_0\big) \le \mathbf{e}_0^U \tag{4.9}$$
$$\mathbf{e}_f^L \le \mathbf{e}_f\big(\mathbf{x}(t_f), t_f\big) \le \mathbf{e}_f^U \tag{4.10}$$

As far as the theory and the code goes, this split is irrelevant although it offers a notional advantage.

The functions, E and F are called the ***endpoint cost*** and ***running cost*** respectively. DIDO works best when the functions,

$$E: \quad \mathbb{R}^{N_x} \times \mathbb{R}^{N_x} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} \tag{4.11}$$
$$F: \quad \mathbb{R}^{N_x} \times \mathbb{R}^{N_u} \times \mathbb{R} \to \mathbb{R} \tag{4.12}$$
$$\mathbf{f}: \quad \mathbb{R}^{N_x} \times \mathbb{R}^{N_u} \times \mathbb{R} \to \mathbb{R}^{N_x} \tag{4.13}$$
$$\mathbf{e}: \quad \mathbb{R}^{N_x} \times \mathbb{R}^{N_x} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{N_e} \tag{4.14}$$
$$\mathbf{h}: \quad \mathbb{R}^{N_x} \times \mathbb{R}^{N_u} \times \mathbb{R} \to \mathbb{R}^{N_h} \tag{4.15}$$

called the ***problem data***, are twice continuously differentiable (C^2-smooth) with respect to their arguments. If this is not the case, DIDO may still run and provide a reasonable answer, but the user must always check the validity of the results.

The input to DIDO is as simple or complicated as is the problem formulation. For instance, if a problem has no path constraints, then there is no need to define a path function.

Faster run times may be possible when the problem posed has additional structure than the one formulated above. We highly recommend that even advanced users do not attempt alternative problem formulations before solving the problem in this *basic form*. Various other options are possible, some of which are discussed later.

For advanced problem formulation and solving it via DIDO, see Refs. [24] and [25].

5 DIDO Expressions for the Basic Problem

The input to DIDO is given by *only* two objects: `problem` and `algorithm`. Anything that is not part of the problem formulation is part of the algorithm; thus, a basic call to DIDO is as simple as the following one line command:

```
[cost, primal] = dido(problem, algorithm)
```

where `[cost, primal]` are typical DIDO outputs representing the candidate optimal values of the cost functional, J, and the candidate optimal solution,

$$\mathbf{x}(\cdot) = \texttt{primal}.\textbf{states} \tag{5.1}$$
$$\mathbf{u}(\cdot) = \texttt{primal}.\textbf{controls} \tag{5.2}$$

respectively.

5.1 State, Control and Other Primal Variables

Perhaps, the first thing for the user to understand clearly is that there is no "propagation" of the equations of motion. The computations are inherently parallel. *This notion is critically important to using DIDO as well as for writing an efficient code.* The states, controls and time are uniquely specified in terms of a DIDO Structure Array, `primal`,[‡] with fields defined by **states**, **controls** and **nodes**. The field **nodes** (and not, for instance, **time**) is used by DIDO to remind the user that candidate optimal values for `primal`.**states** and `primal`.**controls** are given by DIDO at certain discrete **optimal** points,[§]

$$\texttt{primal}.\textbf{nodes} = [t_0, t_1, \ldots, t_N]$$

where $N_n = N + 1$ is, currently, user-specified as

$$\texttt{algorithm}.\textbf{nodes} = N_n$$

Future versions of DIDO will automatically choose N_n as well. How to choose N_n will be apparent later; for most problems N_n ranges from about 15 to 150. Note that the optimal points (i.e. nodes) chosen by DIDO are not uniform: they are called the shifted Legendre-Gauss-Lobatto (LGL) points named after the respective three wise men. The shifted LGL points are completely transparent to the user; other types of "designer" nodes are also possible and discussed later. In any case, the values of state and control time histories at these nodes are described by,

[‡]The user may, of course, choose a variable name other than `primal` for the name of the primal variables, but the names of the **fields** must be exactly as defined in this manual.

[§]These optimal points are known as the Legendre-Gauss-Lobatto points and represent the most general distribution of optimal points for optimal control control problems; all other collection of points (e.g. Legendre-Gauss) are optimal only for a limited number of special problems [26].

$$\texttt{primal}.\textbf{states} = \begin{bmatrix} x_1(t_0) & \cdots & x_1(t_N) \\ \vdots & & \vdots \\ x_{N_x}(t_0) & \cdots & x_{N_x}(t_N) \end{bmatrix} \tag{5.3}$$

$$\texttt{primal}.\textbf{controls} = \begin{bmatrix} u_1(t_0) & \cdots & u_1(t_N) \\ \vdots & & \vdots \\ u_{N_u}(t_0) & \cdots & u_{N_u}(t_N) \end{bmatrix} \tag{5.4}$$

In other words, time runs horizontally (column-wise) and each column is the dimension of the relevant vector. Thus, `primal` is specified as:

- `primal`.**states** $= N_x \times N_n$ real matrix;
- `primal`.**controls** $= N_u \times N_n$ real matrix;
- `primal`.**nodes** $= 1 \times N_n$ real matrix;

5.2 User-Supplied M-files

As a result of the preceding formats, DIDO requires that the problem (defined in Section 4) now be re-formulated to a **vectorized form** for efficient coding in MATLAB. The user must supply all the functions defined in Section 4, as ***functions of matrices***. Thus the following **four** M-files must be provided by the user (see Section 4):

- function $[E, F]$ = `cost_fun`(`primal`)
- function *xdot* = `dynamics_fun`(`primal`)
- function e = `event_fun`(`primal`) [optional]
- function h = `path_fun`(`primal`) [optiona]

The names of all these function files can, of course, be chosen by the user. DIDO requires that these names be organized in a DIDO Structure Array with fields **cost**, **dynamics**, **events** and **path** each of which contain the strings of the names of these functions:

- `problem`.**cost** = ‘`cost_fun`’;
- `problem`.**dynamics** = ‘`dynamics_fun`’;
- `problem`.**events** =‘`event_fun`’;
- `problem`.**path** = ‘`path_fun`’;

You may want to use a descriptive name for problem as this is printed on the screen for easy reference.

5.2.1 M-file cost_fun

The input to `cost-fun` is the Structure Array `primal` as described above. The outputs are the Endpoint cost, E and the running cost, F. The endpoint cost is a numeric array (1×1 array) that can be obtained from `primal`.states(:, 1), `primal`.nodes(1), `primal`.states(:, end) and `primal`.nodes(end), since

- $\mathbf{x}_0 \equiv$ `primal`.states(:, 1),
- $\mathbf{x}_f \equiv$ `primal`.states(:, end),
- $t_0 \equiv$ `primal`.nodes(1), and
- $t_f \equiv$ `primal`.nodes(end).

The running cost is a $1 \times N$ numeric array (row vector) of the running cost evaluated at the discrete points, `primal`.nodes. (Recall time runs horizontally!) If the problem has no E-term or an F-term, it must be set to zero. **Do not** set either term to the empty matrix. If you do, you will get a MATLAB error message: `MATLAB segmentation violation detected !!!`. This is, in part, because in MATLAB, $\emptyset + x = \emptyset$, not, x.

5.2.2 M-file dynamics_fun

This M-file must provide the differential equation,

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) \tag{5.5}$$

Thus, given the Structure Array `primal`, DIDO expects an $N_x \times N_n$ numeric array, *xdot*, out of this M-file, containing the right-hand side of the differential equation. This can easily be evaluated from `primal`.states, `primal`.controls, and `primal`.nodes.

5.2.3 M-file event_fun

This M-file provides the event function $\mathbf{e}$ an $N_e \times 1$ numeric array that describes the events: the boundary conditions for the basic problem. The output ($N_e \times 1$ numeric array) can be provided from `primal`.states(:, 1), `primal`.states(:, end), `primal`.nodes(1) and `primal`.nodes(end), since $\mathbf{x}_0 \equiv$ `primal`.states(:, 1), $\mathbf{x}_f \equiv$ `primal`.states(:, end), $t_0 \equiv$ `primal`.nodes(1), and $t_f \equiv$ `primal`.nodes(end).

5.2.4 M-file path_fun

This M-file provides the path constraints $\mathbf{h}$ at the nodes. That is, $\mathbf{h}$ is an $N_h \times N_n$ numeric array that is obtained from `primal`.states, `primal`.controls and `primal`.nodes.

If a problem does not have any path constraints, it is not necessary to define this function, and hence the corresponding field in the structure `problem`.

6 Calling DIDO

DIDO is typically called from a problem file. The problem file is a script file that calls DIDO using one line command:

```
[cost, primal, dual] = dido(problem, algorithm)
```

where the output variable `dual` is optional but highly recommended as this is one of the many strengths and utilities of DIDO. How to use the dual variables is described in Section 8.

6.1 Typical DIDO Inputs

The `problem` input to DIDO essentially defines the problem, namely the problem functions and the problem bounds. The problem functions (see Section 4) consists of the cost function, the dynamics function, the event function and the path function:

- `problem`.**cost** = '`cost_fun`';
- `problem`.**dynamics** = '`dynamics_fun`';
- `problem`.**events** ='`event_fun`';
- `problem`.**path** = '`path_fun`';

The problem bounds are given by $\mathbf{e}^L, \mathbf{e}^U, \mathbf{h}^L$ etc. (see Section 4) and are passed to DIDO via the problem structure using the field **bounds**,

`problem`.**bounds**

6.1.1 Specification of Problem Bounds

All bounds on the problem are specified via a MATLAB structure array `bounds` with the fields **lower** and **upper** nested by the appropriate variables; for example, states and controls are nested by **lower.states**, **upper.states**, **lower.controls** and **upper.controls**,

- $\mathbf{x}^L \equiv$ bounds.**lower.states** = $[N_x$ by 1] matrix;
- $\mathbf{u}^L \equiv$ bounds.**lower.controls** = $[N_u$ by 1] matrix;
- $\mathbf{x}^U \equiv$ bounds.**upper.states** = $[N_x$ by 1] matrix;
- $\mathbf{u}^U \equiv$ bounds.**upper.controls** = $[N_u$ by 1] matrix;

Similarly, if event and path constraints are specified in the problem then, the bounds corresponding to the *event* constraints ($[\mathbf{e}^L, \mathbf{e}^U]$) and state-control *path* constraints ($[\mathbf{h}^L, \mathbf{h}^U]$) must be specified by,

- bounds.**lower.events** = $\mathbf{e}^L$ [N_e by 1] matrix
- bounds.**lower.path** = $\mathbf{h}^L$ [N_h by 1] matrix;
- bounds.**upper.events** = $\mathbf{e}^U$ [N_e by 1] matrix;
- bounds.**upper.path** = $\mathbf{h}^U$ [N_h by 1] matrix;

The bounds on the clock times,

$$t_0^L \le t_0 \le t_0^U \tag{6.1}$$
$$t_f^L \le t_f \le t_f^U \tag{6.2}$$

are given by,

- bounds.**lower.time** = $[t_0^L, t_f^L]$;
- bounds.**upper.time** = $[t_0^U, t_f^U]$;

6.1.2 Specification of the Algorithm

At a minimum the user must specify the desired level of accuracy of the solution via the DIDO expression,

`algorithm`.**nodes**

Typical values for `algorithm`.**nodes** range from 15 to 150. Lower and higher values may be chosen but are frequently unnecessary. *Accurate solutions to even highly complicated problem can be obtained with node values as low as* 60.

6.2 More Optional DIDO Inputs

A number of additional inputs to DIDO may be specified for various purposes: for a faster run time, for a more accurate solution etc. Some of these optional input parameters are described here; for more information, see [25].

6.2.1 User-Specified Starting Point [Optional]

All algorithms require a starting point. In its default mode, DIDO initiates the spectral algorithm autonomously by choosing an arbitrary starting point based on some cognitive principles. The user may over ride this self-chosen starting point by specifying a potentially better starting point via the structure `algorithm` through its field, **guess**, as in,

`algorithm`.**guess**

The guess variable, `guess`, is defined by the three *fields*, **states**, **controls** and **time**:

- `guess`.**states** = [N_x by N_2] matrix; ($N_2 \ge 2$)

- guess.**controls** = $[N_u$ by $N_2]$ matrix; ($N_2 \geq 2$)
- guess.**time** = [1 by N_2] matrix; ($N_2 \geq 2$)

The number of points used in the guess, N_2, need not be equal to N_n, the number of nodes requested in the output (via `algorithm`.**nodes**), but $N_2 \geq 2$.

DIDO can exploit good guesses for the problem solution. See Sec 10 for further details.

6.2.2 Choosing a Mode for the DIDO Algorithm [Optional]

In its default mode, DIDO, will run in its "nominal" mode. A user may specify alternative modes via the DIDO structure `algorithm` through its field, **mode**, as in,

`algorithm`.**mode**

The input, `algorithm`.**mode**, is a character string that may be specified as 'nominal' (default) or 'accurate'.

For most applications, the results from DIDO under the default mode are quite accurate. The accurate mode is recommended only for certain pathological problems. Note also that the Bellman procedure [23] generates quite accurate results without an increase in the number of nodes.

7 Illustrative Example

One particular formulation of Bernoulli's classic Brachistochrone problem [5] can be summarized as,

$$\begin{aligned} \mathbf{x}^T = \quad & [x, y, v] \in \mathbb{X} & = \left\{ \mathbf{x} : \mathbf{x}^L \leq \mathbf{x} \leq \mathbf{x}^U \right\} \\ \mathbf{u} = \quad & [\theta] \in \mathbb{U} & = \left\{ \theta : \theta^L \leq \theta \leq \theta^U \right\} \end{aligned}$$

$$(Brac:1) \left\{ \begin{array}{lrl} \text{Minimize} & J[\mathbf{x}(\cdot), \mathbf{u}(\cdot), t_f] = & t_f \\ \text{Subject to} & \dot{x} = & v \sin\theta \\ & \dot{y} = & v \cos\theta \\ & \dot{v} = & g \cos\theta \\ & (x_0, y_0, v_0) = & (0, 0, 0) \\ & (x_f, y_f) = & (x^f, y^f) \\ & t_0 = & 0 \\ & t_f \leq & t^U \end{array} \right.$$

where g is a constant, equal to 9.8 m/s^2 for Earth. Note that, for computational purposes, we set $t_f \leq t^U$ where t^U is some reasonably large (but not too large) number. Similarly $\mathbb{X}$ and $\mathbb{U}$ must be large, but not too large spaces. Suppose $x^f = y^f = 10\ m$. Then, this problem is reasonably scaled in metric units and can be directly handled by DIDO without too much fuss; hence, we refer to this as the *Good Brachistochrone Problem*. A Bad Brachistochrone Problem is discussed in Section 11 and *all users are strongly advised to read this section before using DIDO to solve their own problem.*

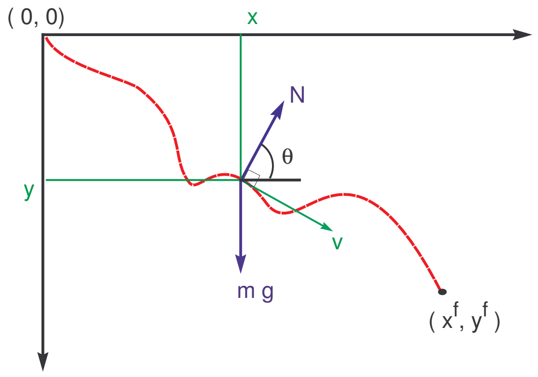

Figure 1: Schematic for the Brachistochrone problem.

7.1 Setting up the DIDO Files

The "Brac:1" formulation of the Brachistochrone problem has no path constraints thus, we only need to specify:

- a cost function file,
- a dynamics function file,
- an event or endpoint function file, and
- a main or problem file that defines the problem.

Recall (see Section 5)

$$\begin{aligned}\mathbf{x}(\cdot) &= \texttt{primal}.\textbf{states}\\ \mathbf{u}(\cdot) &= \texttt{primal}.\textbf{controls}\\ t &= \texttt{primal}.\textbf{nodes}\end{aligned}$$

Thus, we have,

$$x(\cdot) = \texttt{primal}.\textbf{states}(1,:) \tag{7.1}$$
$$y(\cdot) = \texttt{primal}.\textbf{states}(2,:) \tag{7.2}$$
$$v(\cdot) = \texttt{primal}.\textbf{states}(3,:) \tag{7.3}$$
$$\theta(\cdot) = \texttt{primal}.\textbf{controls} \tag{7.4}$$

7.1.1 The Cost Function File

```
function [EndpointCost, RunningCost] = Brac1Cost(primal)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Endpoint Cost for the Brac:1 Formulation of the Brachistochrone Prob
% Template for A Beginner's Guide to DIDO
% I. Michael Ross
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tf = primal.nodes(end);

EndpointCost = tf;
RunningCost  = 0;

% That's it!
% Remember to fill the first output first!
```

7.1.2 The Dynamics Function File

```
function XDOT = Brac1Dynamics(primal)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Dynamics for the Brac:1 Formulation of the Brachistochrone Prob
% Template for A Beginner's Guide to DIDO
% I. Michael Ross
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

x = primal.states(1,:);
y = primal.states(2,:);
```

```
v = primal.states(3,:);

theta  = primal.controls;

%========================================================
% Equations of Motion:
%========================================================
xdot = v.*sin(theta);
ydot = v.*cos(theta);
vdot = 9.8*cos(theta);
%=======================================================
XDOT = [xdot; ydot; vdot];
%=======================================================
```

7.1.3 The Events Function File

```
function endpointFunction = Brac1Events(primal)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Endpoint function for the Brac: 1 Problem
% Template for a A Beginner's Guide to DIDO
% I. Michael Ross
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

x0 = primal.states(1,1);        xf = primal.states(1,end);
y0 = primal.states(2,1);        yf = primal.states(2,end);
v0 = primal.states(3,1);        vf = primal.states(3,end);


% preallocate the endpointFunction evaluation for good MATLAB computing

endpointFunction = zeros(5,1); % t0 is specified in the problem file

%============================================================
endpointFunction(1) = x0;
endpointFunction(2) = y0;
endpointFunction(3) = v0;

%------------------------------------------------------------
endpointFunction(4) = xf;
endpointFunction(5) = yf;
%------------------------------------------------------------
```

7.1.4 The Problem File

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Problem (script) file for the Brachistochrone Problem Formulation, Brac: 1
% Template for A Beginner's Guide to DIDO
% I. Michael Ross
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
clear all;              % always a good idea to begin with this!

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%===================
% Problem variables:
%-------------------
% states = (x, y, v)
% controls = theta
%===================

%---------------------------------------
% bounds the state and control variables
%---------------------------------------

xL = 0; xU = 20;
yL = 0; yU = 20;
vL = 0; vU = 20;

bounds.lower.states = [xL; yL; vL];
bounds.upper.states = [xU; yU; vU];

bounds.lower.controls = [0];
bounds.upper.controls = [pi];


%------------------
% bound the horizon
%------------------
t0  = 0;
tfMax   = 10;   % swag for max tf

bounds.lower.time   = [t0; t0];
bounds.upper.time   = [t0; tfMax];          % Fixed time at t0 and a possibly free time at tf


%-------------------------------------------
% Set up the bounds on the endpoint function
%-------------------------------------------
% See events file for definition of events function

bounds.lower.events = [0; 0; 0; 10; 10];
bounds.upper.events = bounds.lower.events;  % equality event function bounds


%============================================
% Define the problem using DIDO expresssions:
%============================================
Brac_1.cost         = 'Brac1Cost';
```

```
Brac_1.dynamics     = 'Brac1Dynamics';
Brac_1.events       = 'Brac1Events';
%Path file not required for this problem formulation;

Brac_1.bounds       = bounds;
%=====================================================

algorithm.nodes     = [16];                % represents some measure of desired solution accuracy

% Call dido
tStart= cputime;    % start CPU clock
[cost, primal, dual] = dido(Brac_1, algorithm);
runTime = cputime-tStart
% Ta da!
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          OUTPUT            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

x = primal.states(1,:);
y = primal.states(2,:);
v = primal.states(3,:);
t = primal.nodes;

%============================================
figure; plot(t, x, '-o', t, y, '-x', t, v, '-.');
title('Brachistochrone States: Brac 1')
xlabel('time');
ylabel('states');
legend('x', 'y', 'v');

%---------------------------------------------
figure; plot(t, dual.Hamiltonian);
title('Brachistochrone Hamiltonian Evolution');
legend('H');
xlabel('time');
ylabel('Hamiltonian Value');

%---------------------------------------------
figure; plot(t, dual.dynamics);
title('Brachistochrone Costates: Brac 1')
xlabel('time');
ylabel('costates');
legend('\lambda_x', '\lambda_y', '\lambda_v');
%=============================================
```

7.2 Sample Output

The files for the Brachistochrone problem are available (electronically) in the folder ForUser.

A sample plot from a DIDO run are shown in Figures 2 and 3. Note that

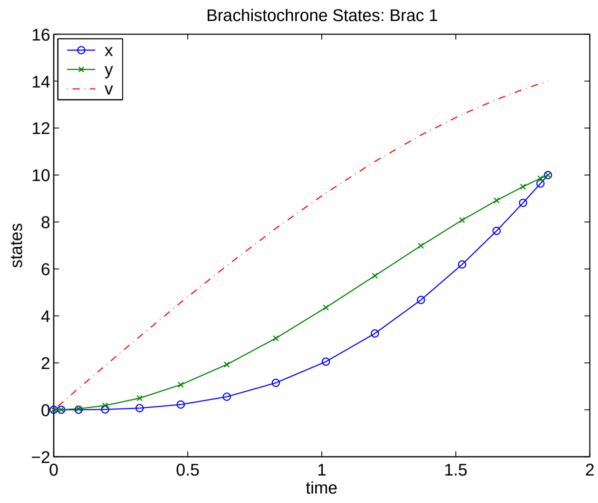

Figure 2: State trajectory from a DIDO run.

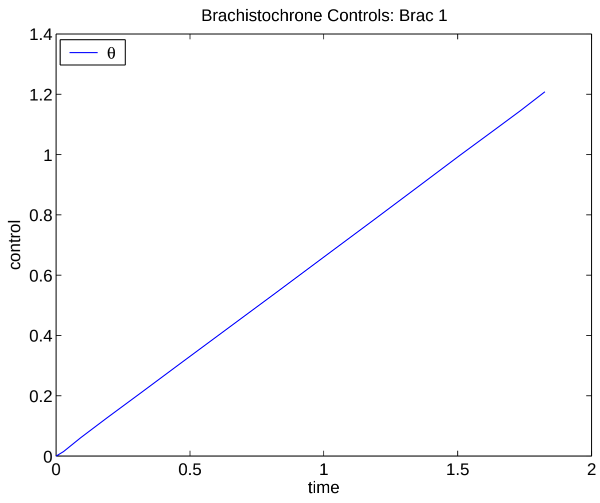

Figure 3: Control trajectory from a DIDO run.

the control trajectory appears to be linear. It can be proven [5] that $t \mapsto \theta$ is indeed a straight line. Thus, DIDO has indeed found the exact solution.

8 Verification and Validation of The Solution

Once DIDO finds a candidate optimal or near-optimal solution, the result can be quickly and easily validated for

- ODE Feasibility
- Pontryagin Extremality

Although DIDO does not require an ODE solver or the development of necessary conditions as an input file, the results generated by DIDO are best analyzed in the true sense of the meaning of necessary conditions. In this context, this section is most useful in debugging concept errors, coding errors and a host of other "weird" errors.

8.1 Verifying Feasibility of The Solution

Since optimality implies feasibility a simple feasibility test can be easily performed by propagating ("pushing forward") $\mathbf{x}^*(t_0)$ (the DIDO output of the initial state) through the differential equation,

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}^*(t), t)$$

where $[t_0^*, t_f^*] \mapsto \mathbf{u}^*$ is the control output from DIDO. For most problems, one can simply use a standard Runge-Kutta method for the propagator; for example, `ode45`. Remember to use the *SAME mathematical models* for the propagator as that used in DIDO. Various measures of error norms can now be used to validate the feasibility of the solution. For example, suppose that the output from `ode45` is given by, $[t_0^*, t_f^*] \mapsto \mathbf{x}_{ode}^*$. Then, a simple method is to eyeball or check to see if $t \mapsto \mathbf{x}_{ode}^*$ passes through $t \mapsto \mathbf{x}^*$. This is the concept of checking for point-wise errors all along the trajectory including the hit or miss from the target, $\mathbf{x}(t_f)$.

A time-function $t \mapsto \mathbf{u}^*$ can be generated by interpolating the values of $\mathbf{u}(t_i)$ at the node points. *DO NOT use Lagrange interpolation for this.* Often a simple linear interpolation suffices; however, it is quite possible that you may get better results using cubic and spline interpolations.

If you have done everything correctly, this should indicate small errors. Remember, though, that this is a negative test: if you don't get small errors, something is wrong (perhaps in your M-files, implementation etc.). On the other hand, an absence of errors does not indicate optimality, but feasibility of the solution. To investigate optimality, you need to develop and check the necessary conditions.

8.2 Verifying Optimality, Extremality

Assuming a successful DIDO run of the problem defined in Sec. 4, DIDO also generates (automatically!) all the dual functions and variables associated with the candidate optimal solution. Thus, one may use necessary conditions as necessary (and not sufficient!) conditions: as a test on the optimality of a candidate feasible solution.

Assuming feasibility, then, according to the Minimum Principle, given a candidate optimal solution, $[t_0^*, t_f^*] \mapsto (\mathbf{x}^*, \mathbf{u}^*)$, (i.e. a DIDO output) there exists a collection of dual functions and variables such that certain statements are true. Depending upon the problem, many, if not all of these statements can be examined as a means to test the optimality of the DIDO run.

From the Minimum Principle, ***if*** $[t_0^*, t_f^*] \mapsto \mathbf{u}^*$ is an ***optimal control trajectory***, then for each $t \in [t_0^*, t_f^*]$, $\mathbf{u}^*$ must satisfy the ***Hamiltonian Minimization Condition***[5, 4],

$$(\text{HMC}) \quad \begin{cases} \underset{\mathbf{u}}{\text{Minimize}} & H(\boldsymbol{\lambda}, \mathbf{x}, \mathbf{u}, t) \\ \text{Subject to} & \mathbf{u} \in \mathbb{U} \end{cases}$$

where H is the ***control Hamiltonian*** (or the Pontryagin Hamiltonian), $\boldsymbol{\lambda}$ is a covector, and $\mathbb{U}$ is the ***control space***. For the problem defined in Sec. 4, the control space is both state and time dependent, given by,

$$\mathbb{U}(\mathbf{x}, t) := \left\{\mathbf{u} : \ \mathbf{h}^L \le \mathbf{h}(\mathbf{x}, \mathbf{u}, t) \le \mathbf{h}^U\right\} \cap \left\{\mathbf{u} : \ \mathbf{u}^L \le \mathbf{u} \le \mathbf{u}^U\right\}$$

Thus, Problem HMC is a nonlinear programming problem (NLP) given by,

$$(\text{HMC}) \quad \begin{cases} \underset{\mathbf{u}}{\text{Minimize}} & H(\boldsymbol{\lambda}, \mathbf{x}, \mathbf{u}, t) \\ \text{Subject to} & \mathbf{h}^L \le \mathbf{h}(\mathbf{x}, \mathbf{u}, t) \le \mathbf{h}^U \\ & \mathbf{u}^L \le \mathbf{u} \le \mathbf{u}^U \end{cases}$$

The Karush-Kuhn-Tucker (KKT) conditions for an NLP are the ***gradient normality condition*** and the ***complementarity conditions***. The gradient normality condition for Problem HMC is given by,

$$\frac{\partial \overline{H}(\boldsymbol{\mu}, \boldsymbol{\lambda}, \mathbf{x}, \mathbf{u}, t)}{\partial \mathbf{u}} = \mathbf{0} \tag{8.1}$$

where $\overline{H}$ is the ***Lagrangian of the Hamiltonian***,

$$\overline{H}(\boldsymbol{\mu}, \boldsymbol{\lambda}, \mathbf{x}, \mathbf{u}, t) := H(\boldsymbol{\lambda}, \mathbf{x}, \mathbf{u}, t) + \boldsymbol{\mu}_h^T \mathbf{h}(\mathbf{x}, \mathbf{u}, t) + \boldsymbol{\mu}_x^T \mathbf{x} + \boldsymbol{\mu}_u^T \mathbf{u} \tag{8.2}$$

$t \mapsto \boldsymbol{\mu} = (\boldsymbol{\mu}_h, \boldsymbol{\mu}_x, \boldsymbol{\mu}_u)$ are the covector functions associated with the path constraints, state-variable box constraints and control-variable box constraints respectively. From the definition of the control Hamiltonian,

$$H(\boldsymbol{\lambda}, \mathbf{x}, \mathbf{u}, t) := F(\mathbf{x}, \mathbf{u}, t) + \boldsymbol{\lambda}^T \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \tag{8.3}$$

an expression for the left hand side of Eq. (8.1) may be obtained quite easily. Substituting the relevant outputs from DIDO into this expression, one can determine if Eq. (8.1) is satisfied to a reasonable level of numerical precision.

Similarity the complementarity conditions may also be checked. These are given by,

$$\mu_{h,i} \begin{cases} \leq 0 & & h_i(\mathbf{x},\mathbf{u},t) = h_i^L \\ = 0 & if & h_i^L < h_i(\mathbf{x},\mathbf{u},t) < h_i^U \\ \geq 0 & & h_i(\mathbf{x},\mathbf{u},t) = h_i^U \\ unrestricted & & h_i^L = h_i^U \end{cases} \tag{8.4}$$

$$\mu_{x,i} \begin{cases} \leq 0 & & x_i = x_i^L \\ = 0 & if & x_i^L < x_i < x_i^U \\ \geq 0 & & x_i = x_i^U \\ unrestricted & & x_i^L = x_i^U \end{cases} \tag{8.5}$$

and

$$\mu_{u,i} \begin{cases} \leq 0 & & u_i = u_i^L \\ = 0 & if & u_i^L < u_i < u_i^U \\ \geq 0 & & u_i = u_i^U \\ unrestricted & & u_i^L = u_i^U \end{cases} \tag{8.6}$$

If Eqs. (8.1), (8.4)-(8.6) are satisfied, then the DIDO control trajectory, $t \mapsto \mathbf{u}^*$, is a KKT point for Problem HMC for each $t \in [t_0^*, t_f^*]$, and $t \mapsto \mathbf{u}^*$ is called an ***extremal control***. If it can be shown (by further analysis) that the KKT point is a (global) minimizer for Problem HMC then the control trajectory, $t \mapsto \mathbf{u}^*$, obtained from DIDO is called a ***Pontryagin extremal control***.

Arguably, the most important feature of DIDO is its ability to get accurate values of the covector functions without solving for the associated necessary conditions (the two-point-boundary-value problem). The Covector Mapping Theorem[13, 9], implemented in DIDO, provides all the covector functions by way of the structure array **dual**.

8.3 Getting Dual Variables

In order to obtain the dual variables, the output from DIDO must be written as,

```
[cost, primal, dual] = dido(problem, algorithm)
```

The structure dual is self-explanatory and is given by,

- $\boldsymbol{\lambda}$ = dual.**dynamics**;
- $\boldsymbol{\mu}_h$ = dual.**path**
- $\boldsymbol{\mu}_x$ = dual.**states**
- $\boldsymbol{\mu}_u$ = dual.**controls**

- $\boldsymbol{\nu}_e$ = dual.**events**
- H = dual.**Hamiltonian**
- $[\boldsymbol{\nu}_{t_0}, \boldsymbol{\nu}_{t_f}]$ = dual.**clock**

9 Illustrating V & V Techniques

Recall the Brachistochrone problem from Section 7:

$$\begin{aligned}\mathbf{x}^T = &\; [x, y, v] \in \mathbb{X} &= \left\{\mathbf{x} : \mathbf{x}^L \le \mathbf{x} \le \mathbf{x}^U\right\}\\ \mathbf{u} = &\; [\theta] \in \mathbb{U} &= \left\{\theta : \theta^L \le \theta \le \theta^U\right\}\end{aligned}$$

$$(Brac:1)\left\{\begin{array}{lrl} \textsf{Minimize} & J[\mathbf{x}(\cdot), \mathbf{u}(\cdot), t_f] = & t_f \\ \textsf{Subject to} & \dot{x} = & v \sin\theta \\ & \dot{y} = & v\cos\theta \\ & \dot{v} = & g\cos\theta \\ & (x_0, y_0, v_0) = & (0,0,0) \\ & (x_f, y_f) = & (x^f, y^f) \\ & t_0 = & 0 \\ & t_f \le & t^U \end{array}\right.$$

Although one may consider only the "primal" problem for using DIDO, this is not advisable, particularly for complex problems. This is because an examination of the necessary conditions will often reveal information about the problem and even debugging suggestions (!) that are simply not possible by consideration of the primal problem alone. Furthermore, an easy analysis of the problem by way of the dual variables is one of the most powerful features of DIDO. Ignoring this is like driving your car in first gear on a freeway. In recognizing an effective way to analyze and solve problems, consider the totality of necessary conditions. To this end, we first construct the Pontryagin Hamiltonian (Cf. §8),

$$H(\boldsymbol{\lambda}, \mathbf{x}, \mathbf{u}) := \lambda_x v \sin\theta + \lambda_y v \cos\theta + \lambda_v g\cos\theta$$

In the theoretical problem, $\mathbf{u} = [\theta]$ is unconstrained; however, in the computational problem, note that $\mathbf{u} \in \mathbb{U}$. Thus, the Hamiltonian minimization condition is given by,

$$(\textsf{HMC})\left\{\begin{array}{ll} \underset{\theta}{\textsf{Minimize}} & H(\boldsymbol{\lambda}, \mathbf{x}, \mathbf{u}) \\ \textsf{Subject to} & \theta^L \le \theta \le \theta^U \end{array}\right.$$

Thus, the *Lagrangian of the Hamiltonian* is given by

$$\overline{H}(\boldsymbol{\mu}, \boldsymbol{\lambda}, \mathbf{x}, \mathbf{u}, t) := H(\boldsymbol{\lambda}, \mathbf{x}, \mathbf{u}, t) + \boldsymbol{\mu}_\theta \theta \tag{9.1}$$

Therefore, the KKT conditions for Problem HMC are

$$\lambda_x v\cos\theta - \lambda_y v \sin\theta - \lambda_v g\sin\theta + \mu_\theta = 0$$

and

$$\mu_\theta \left\{\begin{array}{lcc} \le 0 & & \theta = \theta^L \\ = 0 & if & \theta^L < \theta < \theta^U \\ \ge 0 & & \theta = \theta^U \end{array}\right. \tag{9.2}$$

Since we desire to have $\theta^L < \theta < \theta^U$ so that the computational problem matches the desired problem, we need to make sure that θ^L and θ^U are sufficiently large; i.e. non-binding constraints. If θ^L and θ^U are set too right (i.e. binding constraints), then DIDO will solve the problem given to it, and not the problem "thought" by the user. Assuming a correctly posed problem, then, we must have $t \mapsto \mu_\theta = 0$. This can be checked via `dual`.**controls**. In this case, we have,

$$t \mapsto \lambda_x v \cos\theta - \lambda_y v \sin\theta - \lambda_v g \sin\theta = 0 \tag{9.3}$$

The Hamiltonian value condition and the Hamiltonian evolution equation provide the condition, $t \mapsto H(\boldsymbol{\lambda}, \mathbf{x}, \mathbf{u}) = -1$; that is,

$$t \mapsto \lambda_x v \sin\theta + \lambda_y v \cos\theta + \lambda_v g \cos\theta = -1 \tag{9.4}$$

An examination of the adjoint system, $\dot{\boldsymbol{\lambda}} = -\partial_{\mathbf{x}} H(\boldsymbol{\lambda}, \mathbf{x}, \mathbf{u})$, indicates that,

$$\lambda_x(t) = \text{constant} \quad \lambda_y(t) = \text{constant} \tag{9.5}$$

Thus, Eqs. 9.3–9.5 may be used to validate (or invalidate) a DIDO run.

DIDO Output

The costates are given by `dual`.**dynamics** and the Hamiltonian is given by `dual`.**Hamiltonian**. A plot of these variables are shown in Figures 4 and 5. It

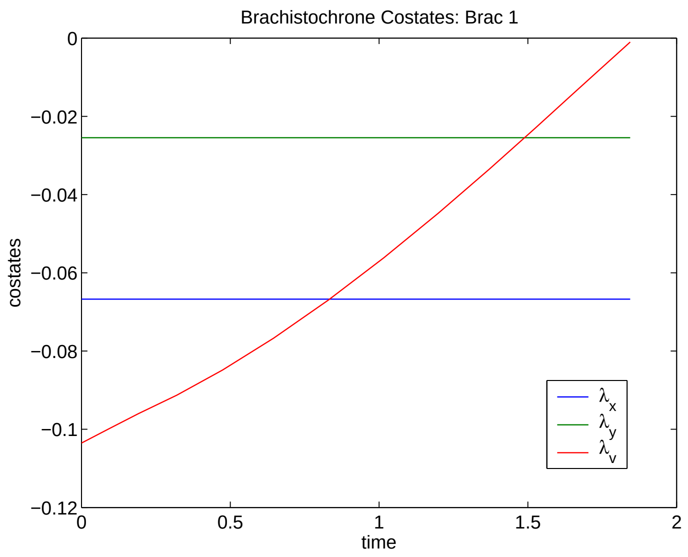

Figure 4: Costate trajectory from a DIDO run.

is quite apparent that $t \mapsto \lambda_x$ and $t \mapsto \lambda_y$ are indeed constants. Furthermore, not only is the Hamiltonian a constant at approximately 10^{-4} precision, it is also equal to -1 as required by theory. Thus, DIDO has indeed found the exact optimal solution at a very high precision.

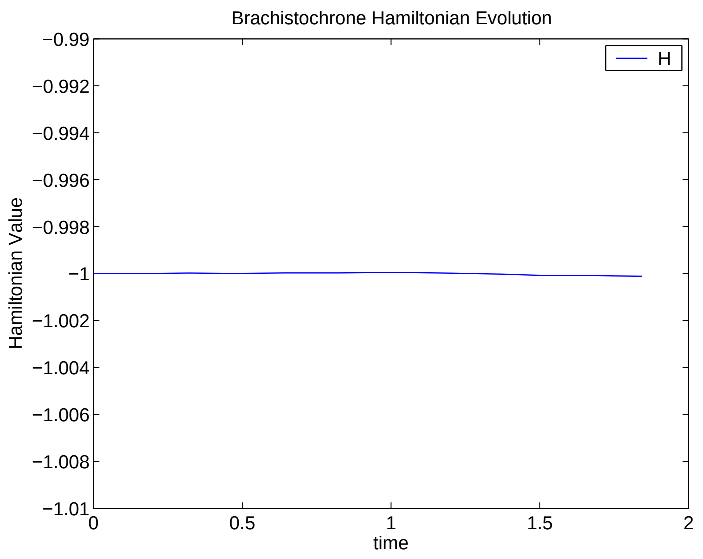

Figure 5: Hamiltonian evolution from a DIDO run; note the scale on the ordinate.

10 Procedure for a Trouble-Free DIDO Run

The purpose of this procedure section is to help a new user in avoiding some of the common pitfalls encountered by the novice. Even seasoned users may find this section helpful.

10.1 Writing the Problem

Many first-time users tend to code a problem without writing (on a piece of paper) a *mathematical formulation* of the problem. Avoid the temptation of arrogance: this leads to a vast number of easily avoidable problems. ***Majority of the mistakes can be traced to the decisions (or the lack of it) made in the first 10 minutes!***[¶] It is strongly recommended that you ***write the problem you are solving***! All it takes is a piece of paper and pencil. This exercise includes the simple effort of recognizing any potential numerical problems such as singularities or scaling issues (See §11). Remember, typing up a code is not writing the problem no matter how similar the code is to the write up. ***Scribbling is also not writing!*** To drive home this point further it might help to know a wonderful fact: *ten hours of coding can save you ten minutes of writing!* ***You will not be given any support with regards to your DIDO results (or the lack of it) if you have not written the problem.***

10.2 Solving the Problem

The Brachistochrone problem is a sufficiently simple problem (i.e. few variables and constraints) that it can be coded in one step. Most practical problems are not that simple. Many newcomers to optimization tend to code their "full problem" right away. This is inadvisable regardless of your experience. After resisting the temptation of arrogance in coding the full problem right away, define the "simplest problem" first. For example, in astronautical applications, a simpler problem can be defined where the only forces acting on the spacecraft are inverse-square gravity and thrust. A "higher-order model" may then be defined as one with J_2 perturbations. A still higher-order model can then include drag and so on. Similarly, depending on the problem, certain path constraints can be ignored at first (e.g. g-load or heating-rate constraints).

Thus, the recommended procedure is as follows: For reasonably complex problems define a "homotopy coding/debugging path" vis-a-vis modeling from a simple problem to the desired problem. In respecting the complexity of the problem at hand, set Problem $P = P_n$ and construct a sequence of problems, $P_n, P_{n-1} \ldots$ that represents in some fashion a family of successively simpler problems. Thus, for example, Problem P_{n-5} may have fewer constraints and/or a lower-fidelity dynamical model. Such problem formulations requires expertise in the particular discipline that the problem belongs to: aerospace, mechanical, economics etc. You are apparently the expert in this discipline. Solve problems starting from P_0 all the way up to $P_n = P$, the desired problem. This "homotopy

[¶]Augstine's Laws.

path" to problem solving will provide insights to the problem formulation as well as the solution in addition to validating the acceptability of the fidelity of a solution and/or the the problem formulation. Not only is this procedure more likely to be free of modeling errors, but it also forces you to write your code in a modular fashion so that a whole family of problems can be solved with relative ease. This recommend procedure has almost nothing to do with DIDO, rather, it is a basic "art" in trouble-free computer programming.

10.3 Scaling and Balancing the Equations

Optimization codes (or for that matter almost all numerical codes) behave in a fundamentally superior fashion (accuracy, speed etc.) when the variables are scaled properly. *We strongly suggest that you scale and balance your problem before you code it for use inside the DIDO loop.* Often, conventional units (e.g. Metric) are not suitable as units. In many problems, significantly superior units can be found for numerical computation. For example, for an orbit transfer problem, the initial radius of the orbit (or sometimes the radius of the planet) provides a natural scale dimension for distance (than, for example kilometers or miles). Thus, for example the semi-major axis for GEO is better coded as 6.61 DU (distance units in terms of radius of Earth) than 42164000 meters. The unit of time is then chosen as the inverse of the Schuler frequency. This naturally defines a unit for speed and so on. The canonical units for orbit transfer problems are discussed in almost all books on Astrodynamics. For other problems, astronautical or otherwise, such units may be naturally and wisely chosen by the modeler. *Sometimes canonical units do not provide good scaling properties.* Analyze the problem scales before coding.

While choosing scales, make sure the problem is well-balanced. Often a poorly scaled problem will have serious numerical difficulties that can quickly and easily be resolved by the very simple procedure of dividing the appropriate quantities by their scaling counterparts.

10.4 Avoiding Problem Singularities

Numerical problems arise when the problem model has potential singularities. In addition to the usual ones (like dividing by zero), problems will also arise in apparently benign functions such as $\sqrt{x}$. Note that this function is not differentiable at $x = 0$. In addition, if $x < 0$ during the course of an iteration, MATLAB will treat it to be complex and "expand the search space". To avoid numerical problems, repose the problem without using $\sqrt{x}$ (e.g. by squaring). If this is not possible, it is always wise to impose bounds on the problem to avoid these "singularities". For example, including a constraint such as $x \geq \epsilon$ will avoid many headaches later. Using ϵ instead of zero prevents a singularity in the computation of the Jacobian. Note however, this may lead to bad scaling!

Warning: Divide by Zero!

This message will appear on the screen when there is a division by zero. It might appear benign since DIDO might continue to run and even generate seemingly correct results. Correct results should not be assumed in the presence of such warning signs. To avoid these warnings, check the following:

- Check the lower and upper bounds on all variables and functions. If the evaluation of any variable or function at the bounds generates a division by zero, change the bounds. For example, you might set the lower bound on radius to be zero. Evaluating an inverse-square gravity field at the lower bound generates a division by zero.

Future versions of DIDO will have singularity-avoidance techniques. Stay tuned!

10.5 Detecting Infeasibilities

Suppose you want to solve a reasonably complex problem, say, Problem P. As indicated in [10], this problem formulation is itself is an iterative process. Two of the biggest mistakes many beginners make are

- Not recognizing that problem formulation is an iterative process, and
- Starting to solve Problem P in the first step!

The fist mistake is somewhat forgivable as it is a result of ignorance; the second mistake, however, is an act of arrogance: the presumption that the problem can be solved in one stroke. See Section 10.2 for ideas on how to solve complex problems. This tried and true approach made sense hundreds of years ago and makes more sense even today for the simple reason that DIDO can easily solve problems that were once considered difficult. As a result of this capability, the burden on the analyst is less about problem solving than verifying and validating computed solutions.

Now, suppose that DIDO returns an infeasible flag. This raises the following possibilities: Is a solution not being obtained because

1. Your computer code of Problem P_k is incorrect, or
2. The problem does not have a solution, or
3. DIDO is failing?

These questions lie at the intersection of problem fidelity, software practice and the seemingly abstract question of the existence of a solution. The most common error beginners make is in claiming that DIDO is not working for Problem P. Before jumping to this conclusion (particularly in foolishly attempting to solve Problem $P = P_n$ in the first step) begin by solving Problem P_0 and work your way up to Problem $P = P_n$. The simple purpose being that having successively solved say Problem P_3, it is far simpler to detect why a solution to Problem P_4 is not forthcoming.

An important caveat in the successive solutions to problems of increasing fidelity is that although we may have a feel for the existence of a solution to Problem $P = P_n$, this "feel" may not hold for Problem P_0 or may violate some mathematical hypothesis required for the existence of a solution; see [10] where the issue of Lipschitz continuity affects the existence of a solution to an apparently benign reduction in the number of variables. Thus, an appreciation for certain mathematical facts is far more important now than it was ever before. A certain collection of preliminary mathematics required is discussed in [10]. Thus, in examining any problem formulation, the first question that needs to be answered is: does a solution exist? The question of an existence of a solution, once largely a purview of theoreticians, is now a major issue in solving practical problems.

10.6 Speeding up DIDO

Although DIDO does not require a guess for initiating its algorithm, it has the potential of taking a user-specified guess and providing a quick solution. We assume that the problem modeler knows something about the problem. Most engineers can design a "good guess" for the control. A good guess does not mean a feasible solution. For complicated problems, we suggest you use common-engineering-sense and generate a "dynamically feasible guess" for the states by integrating the controls. That is, using "engineering intuition" guess $\mathbf{u}(t)$; then determine $\mathbf{x}(t)$ by a numerical integration (e.g. RK45) of $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}(t), t)$. It is not necessary for your guess (control or states) to be feasible from the point of view of satisfying the boundary conditions.

Remember also, that under mild assumptions, the DIDO's spectral algorithm is globally convergent. A good guess has the advantage of fewer iterations and thus a faster run time. *If you get an error message that the problem is infeasible, there is a strong probability that the problem is not modeled correctly (if the problem is indeed feasible) than a problem (issue) with the guess. Although it may appear to be wise to bootstrap your iterations by using a coarse grid (i.e. low N) first, remember not to use too coarse a grid, since the discrete problem might not be feasible.*

You can use the results of a DIDO run from a "preliminary" guess to kick-start a new run using `algorithm.guess` = `primal` (where `primal` is the output from a previous run).

10.7 Speed Bumps

What might appear to be minor computational burdens might actually end up taking significant computational time. Paying attention to some of these details can help make your code run faster but *you risk not understanding your own code when you try to debug it at some later time!*. To alleviate these issues, it may be wise to heavily comment your code — no matter how trivial your "trick".

11 Must Read Section

A well-scaled Brachistochrone problem is one of the easiest problem to solve. If a problem is very badly scaled, this can easily break DIDO (and almost all numerical algorithms to date). *As scaling and balancing are the most important computational steps* a user must perform in order to have a well-oiled, stable code, particularly if CPU run times are critical, a *Bad Brachistochrone Problem* is considered to illustrate the main points.

11.1 The Bad Brachistochrone Problem

Reconsider the "Brac 1" formulation of Bernoulli's Brachistochrone problem,

$$\mathbf{x}^T = [x, y, v] \quad \mathbf{u} = [\theta]$$

$$(Brac:1)\left\{\begin{array}{lrl} \text{Minimize} & J[\mathbf{x}(\cdot), \mathbf{u}(\cdot), t_f] = & t_f \\ \text{Subject to} & \dot{x} = & v\sin\theta \\ & \dot{y} = & v\cos\theta \\ & \dot{v} = & g\cos\theta \\ & (x_0, y_0, v_0) = & (0,0,0) \\ & (x_f, y_f) = & (x^f, y^f) \\ & t_0 = & 0 \\ & t_f \leq & t^U \end{array}\right.$$

where g is a constant, equal to 9.8 m/s^2 for Earth. It is apparent by inspection that if x^f and y^f are 1 or 10 m, then the problem is well-scaled in metric units. Now suppose that

$$x^f = 1\ km = 1000m \quad \text{and} \quad y^f = 1m$$

Theoretically, this makes no difference, but computationally, the problem becomes difficult to manage due to poor scaling. It turns out that even when the problem is well-scaled in metric units, better scaling may be achieved by way of *designer units*[23]; hence, we consider the Brachistochrone problem for arbitrary values of the problem constants, g, x^f and y^f.

11.2 Scaling and Balancing the Problem

Let

$$\overline{\mathbf{x}}^T = [x/X, y/Y, v/V] \quad \overline{\mathbf{u}} = [\theta/\Theta]$$

where X, Y, V and Θ are arbitrary numbers or **designer units**. Similarly let

$$\bar{t} = t/T$$

where T is a designer unit of time. Then, $t = \bar{t}T$, and

$$x = \overline{x}X, \quad y = \overline{y}Y, \quad v = \overline{v}V, \quad \theta = \overline{\theta}\Theta$$

Denoting $d\bar{x}/d\bar{t}$ as $\dot{\bar{x}}$ etc., **note carefully** how Brac:1 is re-written:

$$(Brac:1)\left\{\begin{array}{lrl} \textsf{Minimize} & J[\overline{\mathbf{x}}(\cdot),\overline{\mathbf{u}}(\cdot),\bar{t}_f] = & T\bar{t}_f \\ \textsf{Subject to} & \dot{\bar{x}}\dfrac{X}{T} = & \bar{v}V\sin(\bar{\theta}\Theta) \\ & \dot{\bar{y}}\dfrac{Y}{T} = & \bar{v}V\cos(\bar{\theta}\Theta) \\ & \dot{\bar{v}}\dfrac{V}{T} = & g\cos(\bar{\theta}\Theta) \\ & (\bar{t}_0,\bar{x}_0,\bar{y}_0,\bar{v}_0) = & (0,0,0,0) \\ & (X\bar{x}_f, Y\bar{y}_f) = & (x^f, y^f) \end{array}\right.$$

That is, the variables are scaled but the cost function and constraints are not necessarily scaled. Typically, constraint scaling is not necessary, but in problems that are truly badly scaled, constraint scaling may be necessary. Finally, for DIDO, the Brachistochrone dynamics in the scaled variables are not in the standard format because the left hand side of the dynamics must be $\dot{\mathbf{x}}$. Rewriting the dynamics in this form has the natural effect of scaling the constraint; for example, the $\dot{x}$ equation gets scaled by T/X. Collecting all these ideas together, we have a potentially well-scaled problem given by,

$$(\overline{Brac}:1)\left\{\begin{array}{lrl} \textsf{Minimize} & \overline{J}[\overline{\mathbf{x}}(\cdot),\overline{\mathbf{u}}(\cdot),\bar{t}_f] & =\bar{t}_f \\ \textsf{Subject to} & \dot{\bar{x}} & = \dfrac{TV}{X}\bar{v}\sin(\bar{\theta}\Theta) \\ & \dot{\bar{y}} & = \dfrac{TV}{Y}\bar{v}\cos(\bar{\theta}\Theta) \\ & \dot{\bar{v}} & = \dfrac{Tg}{V}\cos(\bar{\theta}\Theta) \\ & (\bar{x}_0,\bar{y}_0,\bar{v}_0) & = (0,0,0) \\ & (\bar{x}_f,\bar{y}_f) & = \left(\dfrac{x^f}{X},\dfrac{y^f}{Y}\right) \\ & \bar{t}_0 & = 0 \\ & \bar{t}_f & \le \dfrac{t^U}{T} \end{array}\right.$$

Note that the cost function in this formulation ($\overline{J}$, not J) is scaled by T. As will be apparent shortly, this implies that the covectors also get scaled. Furthermore, the cost function may also be scaled by any value (and not necessarily, T)

because minimizing a function or some multiple of the function generates the same value for the independent variable (but not necessarily the cost value). In any event, the problem in its metric units is easily recovered by setting all values of the designer units to be one.

To incorporate Brac:1 within DIDO, we now need to add **non-binding** box constraints,

$$\overline{\mathbf{x}}^L \leq \overline{\mathbf{x}}(t) \leq \mathbf{x}^U \qquad \overline{\mathbf{u}}^L \leq \overline{\mathbf{u}}(t) \leq \overline{\mathbf{u}}^U$$

Failure to make them non-binding (because the "true" problem does not have a box constraints) is one of the most common causes of coding errors. Setting $\overline{\mathbf{x}}^L$ and $\overline{\mathbf{x}}^U$ to be very large may reduce DIDO's CPU performance (i.e. run slow). Experimenting with the bounds on the box constraints to determine good, reasonable values for the box constraints is well worth the effort if the CPU run time is critical for the application.

11.3 How to Choose Good Designer Units

As the input to DIDO is the scaled problem, its output is also "scaled" accordingly. *Failure in recognizing this point is also a common error.* In performing V & V, it is extremely important that DIDO users develop the necessary conditions in the scaled or designer units.

The control Hamiltonian for the scaled problem is given by (see Eq. 8.3),

$$H(\overline{\boldsymbol{\lambda}}, \overline{\mathbf{x}}, \overline{\mathbf{u}}) := \overline{\lambda}_x \frac{TV}{X}\overline{v}\sin(\overline{\theta}\Theta) + \overline{\lambda}_y \frac{TV}{Y}\overline{v}\cos(\overline{\theta}\Theta) + \overline{\lambda}_v \frac{Tg}{V}\cos(\overline{\theta}\Theta)$$

From the Hamiltonian value condition and the Hamiltonian evolution equation, we have

$$H(\overline{\boldsymbol{\lambda}}(t), \overline{\mathbf{x}}(t), \overline{\mathbf{u}}(t)) = -1$$

This implies,

$$\overline{\lambda}_x \frac{TV}{X}\overline{v}\sin(\overline{\theta}\Theta) + \overline{\lambda}_y \frac{TV}{Y}\overline{v}\cos(\overline{\theta}\Theta) + \overline{\lambda}_v \frac{Tg}{V}\cos(\overline{\theta}\Theta) = -1 \qquad (11.1)$$

Now, in the "good" Brachistochrone problem (Cf. §7), we had,

$$x^f = 10\ m \quad \text{and} \quad y^f = 10\ m$$

and all units were "1". An inspection of Figure 2 (see page 18) shows that v and hence, $\overline{v}$, varies from 0 to about 15. From Equation 11.1, this implies that we should expect the costates to vary from 0 to about 0.1. This is precisely the result in Figure 4. It is clear that these set of units (or metric units) would generate a badly scaled problem if we have,

$$x^f = 1\ km = 1000\ m \quad \text{and} \quad y^f = 1\ m$$

A fundamental rule for balancing equations is to choose designer units in such a manner that the states and costates are roughly the same order of magnitude.[‖] As

[‖]I am indebted to Steve Paris for making this point and to Pooya Sekhavat for generating some of the balancing rules discussed in this section.

noted in [5], the covectors do have physical meaning with **co-units** given by,

$$\boldsymbol{\lambda} \text{ UNITS} = \frac{\text{Cost UNITS}}{\text{State UNITS}} \tag{11.2}$$

Consequently, the units for the adjoint covectors are automatically set when one chooses the units for the states; that is, we have,

$$\overline{\lambda}_x = \lambda_x \frac{X}{T} \qquad \overline{\lambda}_y = \lambda_y \frac{Y}{T} \qquad \overline{\lambda}_v = \lambda_v \frac{V}{T} \tag{11.3}$$

As an example, if we choose

$$X = 10 = Y = V$$

in the good Brachistochrone problem with all other units as before (i.e. 1, or metric), then the state variables will vary from approximately 0 to 1 (because of the "physics" of the problem and its data; see Figure 2). Then, according to Equation 11.3, the costates must vary from approximately 0 to 1 as well (Cf. Figure 4). This analysis is borne out in Figure 6. Because the good

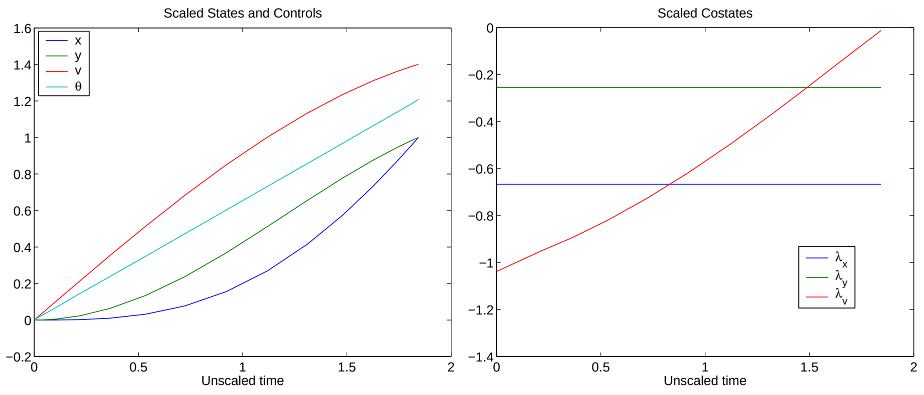

Figure 6: Scaled state and costate trajectory from a DIDO run; compare Figures 2, 3 and 4.

Brachistochrone problem is already well scaled in metric units DIDO works just fine in either units. Note also that the designer units used above are not the usual "canonical" units typically used in the literature. If we were to choose canonical units, we must set,

$$X = Y, \quad V = \frac{X}{T}, \quad T = \frac{V}{g} = \frac{X}{gT}$$

In this case, all units get set automatically if any one of them is chosen to be a specific number. For example, choosing $X = 1m$ we get,

$$X = Y = 1\ m, \quad T = \sqrt{\frac{1}{9.81}} \approx 0.32s, \quad V = \frac{X}{T} \approx 3.1m/s$$

which has the effect of expanding the time scale by about a third, and hence shrinking the velocity scale by about a third. A better choice of canonical units might be to choose $X = x^f = 10\,m$; in this case, the time unit is nearly 1 and the velocity unit is nearly 10 and all the variables range from approximately 0 to 1 and the results will be nearly the same as that illustrated in Figure 6.

It is clear that metric units or canonical units would generate a badly scaled problem if we have,

$$x^f = 1\ km = 1000\ m \quad \text{and} \quad y^f = 1\ m$$

Based on the preceding discussions, one might hastily (and falsely) conclude that we wish to choose $X = 1000$ and $Y = 1$ so that $\overline{x}$ and $\overline{y}$ range from 0 to 1. There are many reasons why this is false, one of which is the presumption that $t \mapsto y$ does not exceed y^f. Note also, that this would imply that $\overline{\lambda}_x$ is multiplied by 1000 (everything else remaining the same). Thus, a few trial runs of DIDO might be necessary to determine a good scaling procedure (or alternatively, deep physical insight!). Based on the physics of the good Brachistochrone problem, we might guess that the solution to the bad Brachistochrone problem would generate a "wild" variation in $t \mapsto y$. Thus, a more informative choice for the units might be

$$X = 100, \quad Y = 20, \quad V = 10, \quad \Theta = 1, \quad T = 10$$

Results from a DIDO run with these units are shown in Figure 7. This result

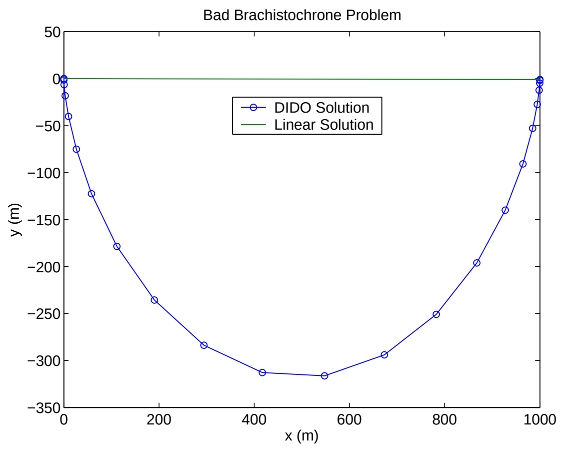

Figure 7: DIDO's Brachistochrone solution vs linear solution; compare the scales with Figures 2 and 3.

further amplifies the rationale behind the unit selection. Of course, the proposed units are not the only "good" units. Systematic experimentation with units can substantially improve DIDO's performance, particularly with respect to run time. It is worth observing that the cost (travel time) for the linear solution is approximately 452 seconds while the optimal cost found by DIDO is approximately 25 seconds. That DIDO's solution is at least an extremal is illustrated in in Figure 8.

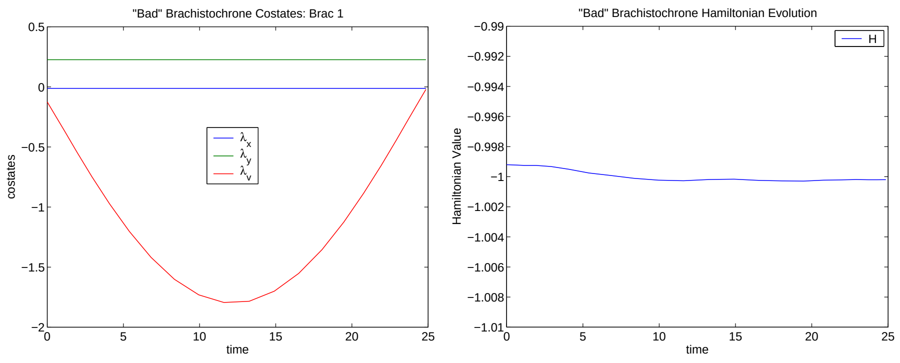

Figure 8: DIDO's Costates and Hamiltonian for the "bad" Brachistochrone problem; compare with Figure 4.

Note that the preceding plots are in metric units. What DIDO actually sees and solves for are the variables in the designer units. These DIDO-computed variables are plotted in Figures 9. Note that the costate λ_v in designer units

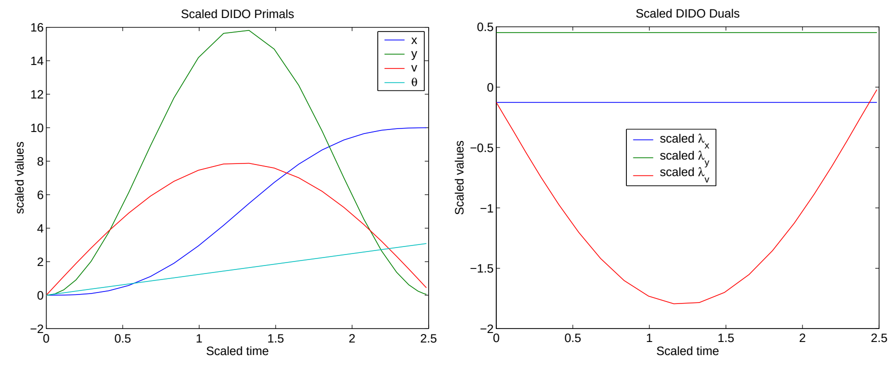

Figure 9: Variables in designer units solved by DIDO.

is indeed exactly equal to its value in metric units. This follows from Equation 11.3 and our choice of $V = T = 10$. Furthermore because $X = 100$, the value of λ_x in designer units (= -0.127) is ten times larger (in absolute value) than it metric value of -0.0127 s/m. Likewise, because $Y = 20$, the value of λ_y (= 0.4510) in designer units is two times larger (in absolute value) than its metric value of 0.2255 s/m.

The files for generating all these plots and analysis is in the folder BrachistochroneProblems in the ForUser folder.

References

[1] Nahin, P. J., *When Least is Best*, Princeton University Press, Princeton, N.J., 2004.

[2] Young, L. C., *Lectures on the Calculus of Variations and Optimal Control Theory*, Saunders, Philadelphia, PA, 1969.

[3] Ross, I. M. and Fahroo, F., "User's Manual for DIDO 2001: A MATLAB Application Package for Dynamic Optimization," *NPS Technical Report AA-01-003*, Department of Aeronautics and Astronautics, Naval Postgraduate School, Monterey, CA, October 2001.

[4] Vinter, R., *Optimal Control*, Birkhäuser, Boston, MA 2000.

[5] Ross, I. M., *Control and Optimization: An Introduction to its Principles and Applications, Electronic Edition*, Naval Postgraduate School, Monterey, CA, October, 2006.

[6] Josselyn, S. and Ross, I. M., "A Rapid Verification Method for the Trajectory Optimization of Reentry Vehicles," *Journal of Guidance, Control, and Dynamics*, Vol. 26, No. 3, 2003.

[7] Lu, P., Sun, H. and Tsai, B., "Closed-Loop Endoatmospheric Ascent Guidance," *Journal of Guidance, Control and Dynamics*, Vol.26, No. 2, 2003.

[8] Stevens, R. and Ross, I. M., "Preliminary Design of Earth-Mars Cyclers Using Solar Sails," *Journal of Spacecraft and Rockets*, Vol. 42, No. 1, Jan-Feb 2005, pp. 132-137.

[9] Ross, I. M., and Fahroo, F., "A Perspective on Methods for Trajectory Optimization," *Proceedings of the AIAA/AAS Astrodynamics Conference*, Monterey, CA, August 2002. Invited Paper No. AIAA 2002-4727.

[10] Ross, I. M., and Gong, Q., *Emerging Principles in Fast Trajectory Optimization*, NPS Technical Report, GNC # 07-2, Monterey, CA 2007.

[11] Ross, I. M., "A Roadmap for Optimal Control: The Right Way to Commute," *Annals of the New York Academy of Sciences*, Vol. 1065, New York, N.Y., January 2006.

[12] Fahroo, F., and Ross, I. M., "Costate Estimation by a Legendre Pseudospectral Method, *Journal of Guidance, Control and Dynamics*, Vol. 24, No. 2, pp. 270-277, 2001.

[13] Ross, I. M., and Fahroo, F., "Legendre Pseudospectral Approximations of Optimal Control Problems," *Lecture Notes in Control and Information Sciences*, Vol.295, Springer-Verlag, New York, 2003, pp. 327-342.

[14] Ross, I. M. and Fahroo, F., "A Unified Framework for Real-Time Optimal Control," *Proceedings of the IEEE Conference on Decision and Control*, Maui, December, 2003.

[15] Ross, I. M. and Fahroo, F., "Pseudospectral Knotting Methods for Solving Optimal Control Problems," *Journal of Guidance, Control and Dynamics*, Vol. 27, No. 3, pp.397-405, 2004.

[16] Gong, G., Kang W., and Ross, I. M., "A Pseudospectral Method for the Optimal Control of Constrained Feedback Linearizable Systems," *IEEE Transactions on Automatic Control*, Vol. 51, No. 7, July 2006, pp. 1115-1129.

[17] Gong, Q., Ross, I. M., Kang, W., and Fahroo, F., "Connections Between the Covector Mapping Theorem and Convergence of Pseudospectral Methods for Optimal Control, to appear in *Computational Optimization and Applications*, 2007.

[18] Gong, Q., Ross, I. M., Kang, W., and Fahroo, F., "On the Pseudospectral Covector Mapping Theorem for Nonlinear Optimal Control, *45th IEEE Conference on Decision and Control*, pp. 2679-2686, San Diago, CA, Dec. 2006.

[19] Ross, I. M. and Fahroo, F., "Issues in the Real-Time Computation of Optimal Control," *Mathematical and Computer Modelling*, An International Journal, Vol. 43, Issues 9-10, May 2006, pp.1172-1188. (Special Issue: Optimization and Control for Military Applications).

[20] Eldersveld, S. K., "Large-Scale Sequential Quadratic Programming Algorithms," PhD thesis, Department of Operations Research, Stanford University, Stanford, CA, 1991.

[21] Gill, P.E., Murray, W., and Saunders, M.A., "SNOPT: An SQP Algorithm for Large-Scale Constrainted Optimization," *SIAM Review*, Vol. 47, No. 1, 2005, pp. 99-131.

[22] Fletcher, R., Leyffer, S. and Toint, P. L., "On the Global Convergence of a FilterSQP Algorithm, *SIAM Journal of Optimization*, Vol. 13, 2002, pp. 4459.

[23] Ross, I. M., Gong, Q. and Sekhavat, P., "Low-Thrust, High-Accuracy Trajectory Optimization," *Journal of Guidance Control and Dynamics*, Vol. 30, No. 4, 2007, pp. 921-933

[24] Ross, I. M. and D'Souza, C. D., "Hybrid Optimal Control Framework for Mission Planning," *Journal of Guidance, Control and Dynamics*, Vol. 28, No. 4, July-August 2005, pp. 686-697.

[25] Ross, I. M., "DIDO User's Manual for Solving Hybrid Optimal Control Problems," (in preparation), Elissar, Monterey, CA 93940.

[26] Fahroo, F. and Ross, I. M., "On Discrete-Time Optimality Conditions for Pseudospectral Methods," *Proceedings of the AIAA/AAS Astrodynamics Conference*, Keystone, CO, August 2006. AIAA-2006-6304