$\mathcal{PSOPT}$ Optimal Control Solver
User Manual
Release 4 build 2015

Victor M. Becerra
Email: v.m.becerra@ieee.org
http://www.psopt.org

Disclaimer

This software is provided "as is" and is distributed free of charge. It comes with no warrantees of any kind. See the license terms for more details. The author does hope, however, that users will find this software useful for research and other purposes.

Licensing Agreement

The software package $\mathcal{PSOPT}$ is distributed under the GNU Lesser General Public License version 2.1. Users of the software must abide by the terms of the license.

Contents

Chapter 1

Introduction to $\mathcal{PSOPT}$

1.1 What is $\mathcal{PSOPT}$

$\mathcal{PSOPT}$ is an open source optimal control package written in C++ that uses direct collocation methods. These methods solve optimal control problems by approximating the time-dependent variables using global or local polynomials. This allows to discretize the differential equations and continuous constraints over a grid of nodes, and to compute any integrals associated with the problem using well known quadrature formulas. Nonlinear programming then is used to find local optimal solutions. $\mathcal{PSOPT}$ is able to deal with problems with the following characteristics:

- Single or multiphase problems

- Continuous time nonlinear dynamics

- General endpoint constraints

- Nonlinear path constraints (equalities or inequalities) on states and/or control variables

- Integral constraints

- Interior point constraints

- Bounds on controls and state variables

- General cost function with Lagrange and Mayer terms.

- Free or fixed initial and final conditions

- Linear or nonlinear linkages between phases

- Fixed or free initial time

- Fixed or free final time

7

- Optimisation of static parameters

- Parameter estimation problems with sampled measurements

- Differential equations with delayed variables.

The implementation has the following features:

- Automatic scaling

- Automatic first and second derivatives using the ADOL-C library

- Numerical differentiation by using sparse finite differences

- Automatic mesh refinement

- Automatic identification of the Jacobian and Hessian sparsity.

- DAE formulation, so that differential and algebraic constraints can be implemented in the same C++ function.

$\mathcal{PSOPT}$ has interfaces to the following NLP solvers:

- IPOPT: an open source C++ implementation of an interior point method for large scale problems.

- SNOPT: a well known and widely used proprietary large scale NLP solver.

1.1.1 Why use $\mathcal{PSOPT}$ and what alternatives exist

These are some reasons why users may wish to use $\mathcal{PSOPT}$:

- Users who for any reason do not have access to commercial optimal control solvers and wish to employ a free open source package for optimal control which does not need a proprietary software environment to run.

- Users who need to link an optimal control solver from stand alone applications written in C++ or other programming languages.

- Users who want to do research with the software, for instance by implementing their own problems, or by customising the code.

$\mathcal{PSOPT}$ does not require a commercial software environment to run on, or to be compiled. $\mathcal{PSOPT}$ is fully compatible with the `gcc` compiler, and has been developed under Linux, a free operating system. Note also that the default NLP solver (IPOPT) requires a sparse linear solver from a range of

options, some of which are available at no cost. The author has personally used the linear solver `ma27`.

$\mathcal{PSOPT}$ has also been ported to be compiled by Microsoft Visual Studio 2010 under Windows 7.

At the time of writing, there is at least one freely available tool implementing a pseudospectral optimal control method. This is GPOPS [34], which has been developed by Anil Rao (University of Florida) and co-workers. GPOPS uses a Gauss pseudospectral method and requires Matlab and SNOPT. GPOPS can be downloaded from:

http://sourceforge.net/projects/gpops/

There are commercial tools for solving large scale optimal control problems. Some modern commercial tools include:

- SOCS developed by J.T. Betts from Boeing, which is a well known tool that is able to solve very large optimal control problems and uses a direct transcription method. See:

  http://www.boeing.com/phantom/socs/

- GESOP developed by Astos Solutions GmbH, Germany, which uses various methods for solving complex optimal control problems and includes a graphical user interface. See:

  http://www.astos.de/products/gesop

- PROPT, developed by P. E. Rutquist and M. M. Edvall from Tomlab Optimization, which runs under Matlab and uses pseudospectral methods. See:

  http://www.tomdyn.com

- DIDO, developed by I.M. Ross from the Postgraduate Naval School in Monterey, California, is a package that runs under Matlab and uses pseudospectral methods. See:

  http://www.elissar.biz/DIDO.html

Other software tools implementing direct transcription methods for optimal control include:

- DIRCOL , authored by O. von Stryk, which is a Fortran 77 based tool that uses a direct collocation method. See:

- DYNOPT, authored by M. Fikar and M. Cizniar, which is a MATLAB based tool that uses orthogonal collocation on finite elements. See:

1.2 $\mathcal{PSOPT}$ user's group

A user's group has been created with the purpose of enabling users to share their experiences with using $\mathcal{PSOPT}$, and to keep a public record of exchanges with the author. It is also a way of being informed about the latest developments with $\mathcal{PSOPT}$ and to ask for help. Membership is free and open. The $\mathcal{PSOPT}$ user's group is located at:

1.2.1 About the author

Victor M. Becerra obtained his first degree in Electrical Engineering in 1990 from Simon Bolivar University, Caracas Venezuela. Between 1989 and 1991 he worked in power systems analysis in CVG Edelca, Caracas. He obtained his PhD for his work on the development of nonlinear optimal control methods from City University, London, in 1994. Between 1994 and 1999 he was a Research Fellow at the Control Engineering Research Centre at City University, London. He is currently a Reader in Cybernetics at the University of Reading, UK, where he has been an academic since 2000. At Reading, he lead the Cybernetics Intelligence Research Group between 2003 and 2009, and currently manages the MSc in Cybernetics, a one year postgraduate programme which he founded. He is a Senior Member of the IEEE and a Member of the Institute of Engineering and Technology. He has published over 100 research papers and one book. His web site is: http://www.reading.ac.uk/~shs99vmb.

1.2.2 Contacting the author

The author is open to discussing with users potential research collaboration leading to publications, academic exchanges, or joint projects. He can be contacted directly for help on the installation and use of the software. Consultancy and training on the use of $\mathcal{PSOPT}$ are also available. His email address is: `v.m.becerra@ieee.org`.

1.2.3 How you can help

You may help improve $\mathcal{PSOPT}$ in a number of ways.

- Sending bug reports.

- Sending corrections to the documentation.

- Discussing with the author ways to improve the computational aspects or capabilities of the software.

- Sending to the author proposed modifications to the source code, for consideration to be included in a future release of $\mathcal{PSOPT}$.

- Sending source code with new examples which may be included (with due acknowledgement) in future releases of $\mathcal{PSOPT}$.

- Porting the software to new architectures.

- If you have had a good experience with $\mathcal{PSOPT}$, tell your students or colleagues about it.

- Quoting the use of $\mathcal{PSOPT}$ in your scientific publications. This document may be referenced as follows:

    - Becerra, V.M. (2010). "Solving complex optimal control problems at no cost with PSOPT". *Proc. IEEE Multi-conference on Systems and Control*, Yokohama, Japan, September 7-10, 2010, pp. 1391-1396

    - Becerra, V.M. (2010). *PSOPT Optimal Control Solver User Manual. Release 3*. Available: `http://code.google.com/p/psopt/downloads/list`

- Developing interfaces to other NLP solvers.

1.3   What is new in Release 3

1. New interface to facilitate the definition of single or multi-phase parameter estimation problems involving sampled observations.

2. Support of newer versions of IPOPT and ADOL-C.

3. Additional auxiliary functions, including 2-D interpolation functions.

4. New function to generate multi-plots with GNUplot, this is multiple plots in a single window.

5. Additional examples

6. Miscellaneous improvements and bug fixing.

1.4 General problem formulation

$\mathcal{PSOPT}$ solves the following general optimal control problem with N_p phases:

Problem $\mathcal{P}_1$

Find the control trajectories, $u^{(i)}(t), t \in [t_0^{(i)}, t_f^{(i)}]$, state trajectories $x^{(i)}(t), t \in [t_0^{(i)}, t_f^{(i)}]$, static parameters $p^{(i)}$, and times $t_0^{(i)}, t_f^{(i)}, i = 1, \ldots, N_p$, to minimise the following performance index:

$$J = \sum_{i=1}^{N_p} \left[\varphi^{(i)}[x^{(i)}(t_f^{(i)}), p^{(i)}, t_f^{(i)}] + \int_{t_0^{(i)}}^{t_f^{(i)}} L^{(i)}[x^{(i)}(t), u^{(i)}(t), p^{(i)}, t]dt \right]$$

subject to the differential constraints:

$$\dot{x}^{(i)}(t) = f^{(i)}[x^{(i)}(t), u^{(i)}(t), p^{(i)}, t], \ t \in [t_0^{(i)}, t_f^{(i)}],$$

the path constraints

$$h_L^{(i)} \leq h^{(i)}[x^{(i)}(t), u^{(i)}(t), p^{(i)}, t] \leq h_U^{(i)}, \ t \in [t_0^{(i)}, t_f^{(i)}],$$

the event constraints:

$$e_L^{(i)} \leq e^{(i)}[x^{(i)}(t_0^{(i)}), u^{(i)}(t_0^{(i)}), x^{(i)}(t_f^{(i)}), u^{(i)}(t_f^{(i)}), p^{(i)}, t_0^{(i)}, t_f^{(i)}] \leq e_U^{(i)},$$

the phase linkage constraints:

$$\begin{aligned}
\Psi_l \leq \Psi[&x^{(1)}(t_0^{(1)}), u^{(1)}(t_0^{(1)}), \\
&x^{(1)}(t_f^{(1)}), u^{(1)}(t_f^{(1)}), p^{(1)}, t_0^{(1)}, t_f^{(1)}, \\
&x^{(2)}(t_0^{(2)}), u^{(2)}(t_0^{(2)}) \\
&, x^{(2)}(t_f^{(2)}), u^{(2)}(t_f^{(2)}), p^{(2)}, t_0^{(2)}, t_f^{(2)}, \\
&\vdots \\
&x^{(N_p)}(t_0^{(N_p)}), u^{(N_p)}(t_0^{(N_p)}), \\
&x^{(N_p)}(t_f^{(N_p)}), u^{(N_p)}(t_f^{(N_p)})), p^{(N_p)}, t_0^{(N_p)}, t_f^{(N_p)}] \leq \Psi_u
\end{aligned}$$

the bound constraints:

$$u_L^{(i)} \leq u^i(t) \leq u_U^{(i)}, \ t \in [t_0^{(i)}, t_f^{(i)}],$$

$$x_L^{(i)} \leq x^i(t) \leq x_U^{(i)}, \ t \in [t_0^{(i)}, t_f^{(i)}],$$

$$p_L^{(i)} \leq p^{(i)} \leq p_U^{(i)},$$

$$\underline{t}_0^{(i)} \leq t_0^{(i)} \leq \bar{t}_0^{(i)},$$

$$\underline{t}_f^{(i)} \leq t_f^{(i)} \leq \bar{t}_f^{(i)},$$

and the following constraints:

$$t_f^{(i)} - t_0^{(i)} \geq 0,$$

where $i = 1, \ldots, N_p$, and

$$
\begin{aligned}
&u^{(i)} : [t_0^{(i)}, t_f^{(i)}] \rightarrow \mathcal{R}^{n_u^{(i)}} \\
&x^{(i)} : [t_0^{(i)}, t_f^{(i)}] \rightarrow \mathcal{R}^{n_x^{(i)}} \\
&p^{(i)} \in \mathcal{R}^{n_p^{(i)}} \\
&\varphi^{(i)} : \mathcal{R}^{n_x^{(i)}} \times \mathcal{R}^{n_x^{(i)}} \times \mathcal{R}^{n_p^{(i)}} \times \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R} \\
&L^{(i)} : \mathcal{R}^{n_x^{(i)}} \times \mathcal{R}^{n_u^{(i)}} \times \mathcal{R}^{n_p^{(i)}} \times [t_0^{(i)}, t_f^{(i)}] \rightarrow \mathcal{R} \\
&f^{(i)} : \mathcal{R}^{n_x^{(i)}} \times \mathcal{R}^{n_u^{(i)}} \times \mathcal{R}^{n_p^{(i)}} \times [t_0^{(i)}, t_f^{(i)}] \rightarrow \mathcal{R}^{n_x^{(i)}} \\
&h^{(i)} : \mathcal{R}^{n_x^{(i)}} \times \mathcal{R}^{n_u^{(i)}} \times \mathcal{R}^{n_p^{(i)}} \times [t_0^{(i)}, t_f^{(i)}] \rightarrow \mathcal{R}^{n_h^{(i)}} \\
&e^{(i)} : \mathcal{R}^{n_x^{(i)}} \times \mathcal{R}^{n_u^{(i)}} \times \mathcal{R}^{n_x^{(i)}} \times \mathcal{R}^{n_u^{(i)}} \times \Re^{n_p^{(i)}} \times \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}^{n_e^{(i)}} \\
&\quad \Psi : U_\Psi \rightarrow \mathcal{R}^{n_\psi}
\end{aligned}
\tag{1.1}
$$

where U_ψ is the domain of function Ψ.

A multiphase problem like $\mathcal{P}_1$ is defined and discussed in the book by Betts [3].

1.5 Overview of the Legendre and Chebyshev pseudospectral methods

1.5.1 Introduction to pseudospectral optimal control

Pseudospectral methods were originally developed for the solution of partial differential equations and have become a widely applied computational tool in fluid dynamics [12, 11]. Moreover, over the last 15 years or so, pseudospectral techniques have emerged as important computational methods for solving optimal control problems [16, 17, 19, 36, 26]. While finite difference methods approximate the derivatives of a function using local information, pseudospectral methods are, in contrast, global in the sense that they use information over samples of the whole domain of the function to approximate its derivatives at selected points. Using these methods, the state and control functions are approximated as a weighted sum of smooth basis functions, which are often chosen to be Legendre or Chebyshev polynomials in the interval $[-1, 1]$, and collocation of the differential-algebraic

equations is performed at orthogonal collocation points, which are selected to yield interpolation of high accuracy. One of the main appeals of pseudospectral methods is their exponential (or spectral) rate of convergence, which is faster than any polynomial rate. Another advantage is that with relatively coarse grids it is possible to achieve good accuracy [42]. In cases where global collocation is unsuitable (for example, when the solution exhibits discontinuities), multi-domain pseudospectral techniques have been proposed, where the problem is divided into a number of subintervals and global collocation is performed along each subinterval [11].

Pseudospectral methods directly discretize the original optimal control problem to formulate a nonlinear programming problem, which is then solved numerically using a sparse nonlinear programming solver to find approximate local optimal solutions. Approximation theory and practice shows that pseudospectral methods are well suited for approximating smooth functions, integrations, and differentiations [10, 42], all of which are relevant to optimal control problems. For differentiation, the derivatives of the state functions at the discretization nodes are easily computed by multiplying a constant differentiation matrix by a matrix with the state values at the nodes. Thus, the differential equations of the optimal control problem are approximated by a set of algebraic equations. The integration in the cost functional of an optimal control problem is approximated by well known Gauss quadrature rules, consisting of a weighted sum of the function values at the discretization nodes. Moreover, as is the case with other direct methods for optimal control, it is easy to represent state and control dependent constraints.

The Legendre pseudospectral method for optimal control problems was originally proposed by Elnagar and co-workers in 1995 [16]. Since then, authors such as Ross, Fahroo and co-workers have analysed, extended and applied the method. For instance, convergence analysis is presented in [27], while an extension of the method to multi-phase problems is given in [36]. An application that has received publicity is the use of the Legendre pseudospectral method for generating real time trajectories for a NASA spacecraft maneouvre [26]. The Chebyshev pseudospectral method for optimal control problems was originally proposed in 1988 [44]. Fahroo and Ross proposed an alternative method for trajectory optimisation using Chebyshev polynomials [19].

Some details on approximating continuous functions using Legendre and Chebyshev polynomials are given below. Interested readers are referred to [10] for further details.

1.6   Pseudospectral approximations

1.6.1   Interpolation and the Lagrange polynomial

It is a well known fact in numerical analysis [9] that if $\tau_0, \tau_1, \ldots, \tau_N$ are $N+1$ distinct numbers and f is a function whose values are given at those numbers, then a unique polynomial $P(\tau)$ of degree at most N exists with

$$f(\tau_k) = P(\tau_k), \ \text{ for } k = 0, 1, \ldots, N$$

This polynomial is given by:

$$P(\tau) = \sum_{k=0}^{N} f(\tau_k)\mathcal{L}_k(\tau)$$

where

$$\mathcal{L}_k(\tau) = \prod_{i=0, i\neq k}^{N} \frac{\tau - \tau_i}{\tau_k - \tau_i} \tag{1.2}$$

$P(\tau)$ is known as the Lagrange interpolating polynomial and $\mathcal{L}_k(\tau)$ are known as Lagrange basis polynomials.

1.6.2   Polynomial expansions

Assume that $\{p_k\}_{k=0,1,\ldots}$ is a system of algebraic polynomials, with degree of $p_k = k$, that are mutually orthogonal over the interval $[-1, 1]$ with respect to a weight function w:

$$\int_{-1}^{1} p_k(\tau)p_m(\tau)w(\tau)\mathrm{d}\tau = 0, \ \text{ for } m \neq k$$

Define $L_w^2[-1, 1]$ as the space of functions where the norm:

$$||v||_w = \left(\int_{-1}^{1} |v(\tau)|^2 w(\tau)\mathrm{d}\tau \right)^{1/2}$$

is finite. A function $f \in L_w^2[-1, 1]$ in terms of the system $\{p_k\}$ can be represented as a series expansion:

$$f(\tau) = \sum_{k=0}^{\infty} \hat{f}_k p_k(\tau)$$

where the coefficients of the expansion are given by:

$$\hat{f}_k = \frac{1}{||p_k||^2} \int_{-1}^{1} f(\tau)p_k(\tau)w(\tau)\mathrm{d}\tau \tag{1.3}$$

The truncated expansion of f for a given N is:

$$\mathcal{P}_N f(\tau) = \sum_{k=0}^{N} \hat{f}_k p_k(\tau)$$

This type of expansion is at the heart of spectral and pseudospectral methods.

1.6.3 Legendre polynomials and numerical quadrature

A particular class of orthogonal polynomials are the Legendre polynomials, which are the eigenfunctions of a singular Sturm-Liouville problem [10]. Let $L_N(\tau)$ denote the Legendre polynomial of order N, which may be generated from:

$$L_N(\tau) = \frac{1}{2^N N!} \frac{d^N}{d\tau^N} (\tau^2 - 1)^N$$

Legendre polynomials are orthogonal over [-1,1] with the weight function $w = 1$. Examples of Legendre polynomials are:

$$L_0(\tau) = 1$$
$$L_1(\tau) = \tau$$
$$L_2(\tau) = \frac{1}{2}(3\tau^2 - 1)$$
$$L_3(\tau) = \frac{1}{2}(5\tau^3 - 3\tau)$$

Figure 1.1 illustrates the Legendre polynomials $L_N(\tau)$ for $N = 0, 1, 2, 4, 5, 10$.

Let τ_k, $k = 0, \ldots, N$ be the Lagrange-Gauss-Lobatto (LGL) nodes, which are defined as $\tau_0 = -1$, $\tau_N = 1$, and τ_k, being the roots of $\dot{L}_N(\tau)$ in the interval $[-1, 1]$ for $k = 1, 2, \ldots, N - 1$. There are no explicit formulas to compute the roots of $\dot{L}_N(\tau)$, but they can be computed using known numerical algorithms. For example, for $N = 20$, the LGL nodes $\tau_k, k = 0, \ldots, 20$ are shown in Figure 1.2.

Note that if $h(\tau)$ is a polynomial of degree $\leq 2N - 1$, its integral over $\tau \in [-1, 1]$ can be exactly computed as follows:

$$\int_{-1}^{1} h(\tau)d\tau = \sum_{k=0}^{N} h(\tau_k)w_k \tag{1.4}$$

where τ_k, $k = 0, \ldots, N$ are the LGL nodes and the weights w_k are given by:

$$w_k = \frac{2}{N(N + 1)} \frac{1}{[L_N(\tau_k)]^2}, \quad k = 0, \ldots, N. \tag{1.5}$$
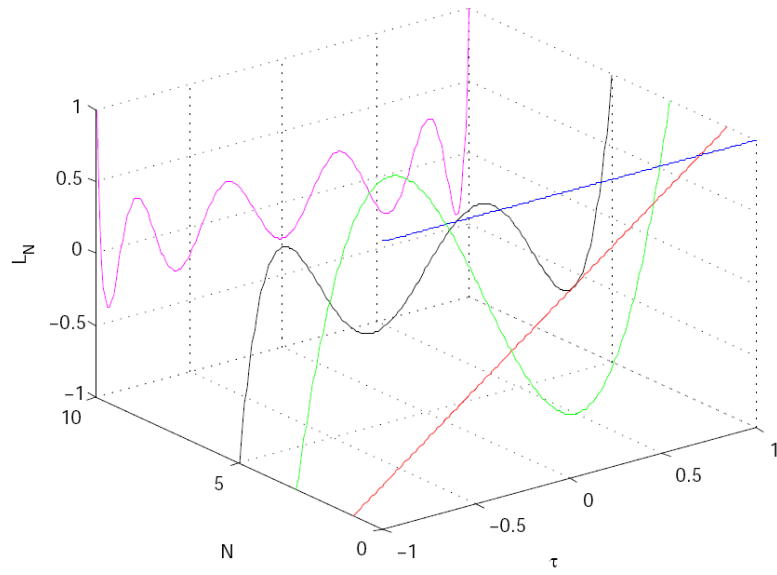
Figure 1.1: Illustration of the Legendre polynomials $L_N(\tau)$ for $N = 0, 1, 2, 4, 5, 10$.
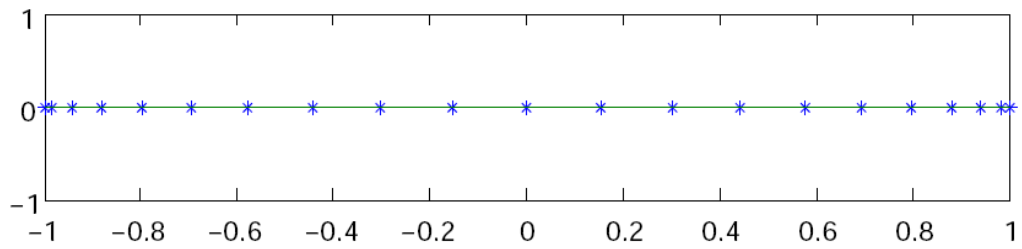
Figure 1.2: Illustration of the Legendre Gauss Lobatto (LGL) nodes for $N = 20$.

17

If $L(\tau)$ is a general smooth function, then for a suitable N, its integral over $\tau \in [-1, 1]$ can be approximated as follows:

$$\int_{-1}^{1} L(\tau)d\tau \approx \sum_{k=0}^{N} L(\tau_k)w_k \tag{1.6}$$

The LGL nodes are selected to yield highly accurate numerical integrals. For example, consider the definite integral

$$\int_{-1}^{1} e^t \cos(t)dt$$

The exact value of this integral to 7 decimal places is 1.9334214. For $N = 3$ we have $\tau = [-1, -0.4472, 0.4472, 1]$, $w = [0.1667, 0.8333, 0.8333, 0.1667]$, hence

$$\int_{-1}^{1} e^t \cos(t)dt \approx w^T h(\tau) = 1.9335$$

so that the error is $\mathcal{O}(10^{-5})$. On the other hand, if $N = 5$, then the approximate value is 1.9334215, so that the error is $\mathcal{O}(10^{-7})$.

1.6.4   Interpolation and Legendre polynomials

The Legendre-Gauss-Lobatto quadrature motivates the following expression to approximate the weights of the expansion (1.3):

$$\hat{f}_k \approx \tilde{f}_k = \frac{1}{\gamma_k} \sum_{j=0}^{N} f(\tau_j)L_k(\tau_j)w_j$$

where

$$\gamma_k = \sum_{j=0}^{N} L_k^2(\tau_j)w_j$$

It is simple to prove (see [24]) that with these weights, function $f : [-1, 1] \rightarrow \Re$ can be interpolated over the LGL nodes as a discrete expansion using Legendre polynomials:

$$I_N f(\tau) = \sum_{k=0}^{N} \tilde{f}_k L_k(\tau) \tag{1.7}$$

such that

$$I_N f(\tau_j) = f(\tau_j) \tag{1.8}$$

Because $I_N f(\tau)$ is an interpolant of $f(\tau)$ at the LGL nodes, and since the interpolating polynomial is unique, we may express $I_N f(\tau)$ as a Lagrange interpolating polynomial:

$$I_N f(\tau) = \sum_{k=0}^{N} f(\tau_k) \mathcal{L}_k(\tau) \tag{1.9}$$

so that the expressions (1.7) and (1.9) are mathematically equivalent. Expression (1.9) is computationally advantageous since, as discussed below, it allows to express the approximate values of the derivatives of the function f at the nodes as a matrix multiplication. It is possible to write the Lagrange basis polynomials $\mathcal{L}_k(\tau)$ as follows [24]:

$$\mathcal{L}_k(\tau) = \frac{1}{N(N+1)L_N(\tau_k)} \frac{(\tau^2 - 1)\dot{L}_N(\tau)}{\tau - \tau_k}$$

The use of polynomial interpolation to approximate a function using the LGL points is known in the literature as the *Legendre pseudospectral approximation method*. Denote $f^N(\tau) = I_N f(\tau)$. Then, we have:

$$f(\tau) \approx f^N(\tau) = \sum_{k=0}^{N} f(\tau_k) \mathcal{L}_k(\tau) \tag{1.10}$$

It should be noted that $\mathcal{L}_k(\tau_j) = 1$ if $k = j$ and $\mathcal{L}_k(\tau_j) = 0$, if $k \neq j$, so that:

$$f^N(\tau_k) = f(\tau_k) \tag{1.11}$$

Regarding the accuracy and error estimates of the Legendre pseudospectral approximation, it is well known that for smooth functions $f(\tau)$, the rate of convergence of $f^N(\tau)$ to $f(\tau)$ at the collocation points is faster than any power of $1/N$. The convergence of the pseudospectral approximations used by $\mathcal{PSOPT}$ has been analysed by Canuto *et al* [10].

Figure 1.3 shows the degree N interpolation of the function $f(\tau) = 1/(1 + \tau + 15\tau^2)$ in $(N+1)$ equispaced and LGL points for $N = 20$. With increasing N, the errors increase exponentially in the equispaced case (this is known as the Runge phenomenon) whereas in the LGL case they decrease exponentially.

1.6.5 Approximate differentiation

The derivatives of $f^N(\tau)$ in terms of $f(\tau)$ at the LGL points τ_k can be obtained by differentiating Eqn. (1.10). The result can be expressed as a matrix multiplication, such that:

$$\dot{f}(\tau_k) \approx \dot{f}^N(\tau_k) = \sum_{i=0}^{N} D_{ki} f(\tau_i)$$

19
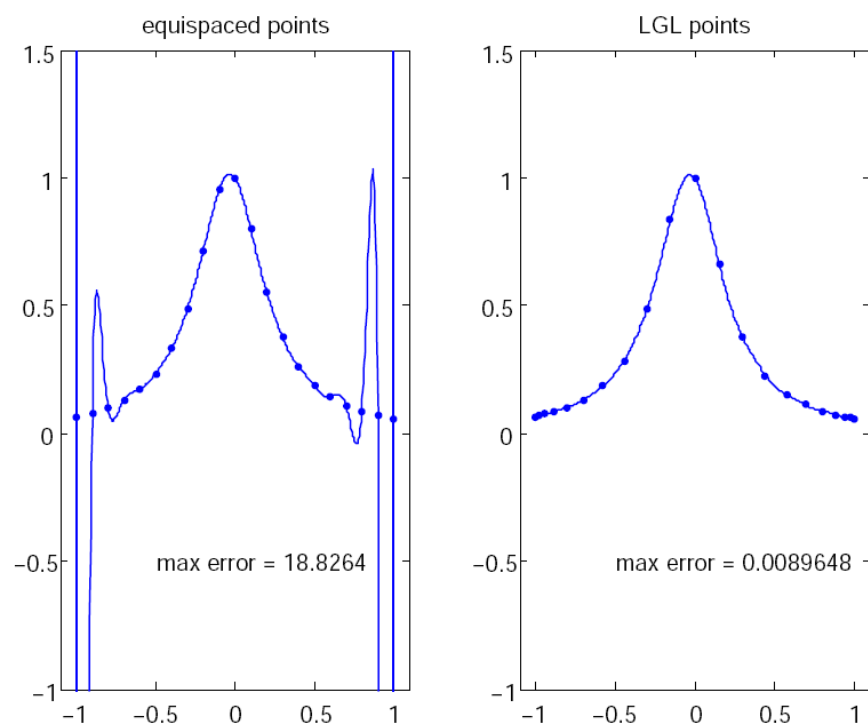
Figure 1.3: Illustration of polynomial interpolation over equispaced and LGL nodes

.

where

$$D_{ki} = \begin{cases} -\dfrac{L_N(\tau_k)}{L_N(\tau_i)} \dfrac{1}{\tau_k - \tau_i} & \text{if } k \neq i \\ N(N+1)/4 & \text{if } k = i = 0 \\ -N(N+1)/4 & \text{if } k = i = N \\ 0 & \text{otherwise} \end{cases} \tag{1.12}$$

which is known as the differentiation matrix.

For example, this is the Legendre differentiation matrix for $N = 5$.

$$D = \begin{bmatrix} 7.5000 & -10.1414 & 4.0362 & -2.2447 & 1.3499 & -0.5000 \\ 1.7864 & 0 & -2.5234 & 1.1528 & -0.6535 & 0.2378 \\ -0.4850 & 1.7213 & 0 & -1.7530 & 0.7864 & -0.2697 \\ 0.2697 & -0.7864 & 1.7530 & 0 & -1.7213 & 0.4850 \\ -0.2378 & 0.6535 & -1.1528 & 2.5234 & 0 & -1.7864 \\ 0.5000 & -1.3499 & 2.2447 & -4.0362 & 10.1414 & -7.5000 \end{bmatrix}$$

Figure 1.4 shows the Legendre differentiation of $f(t) = \sin(5t^2)$ for $N = 20$ and $N = 30$. Note the vertical scales in the error curves. Figure 1.5 shows the maximum error in the Legendre differentiation of $f(t) = \sin(5t^2)$ as a function of N. Notice that the error initially decreases very rapidly until such high precision is achieved (accuracy in the order of 10^{-12}) that round off errors due to the finite precision of the computer prevent any further reductions. This phenomenon is known as *spectral accuracy*.

1.6.6 Approximating a continuous function using Chebyshev polynomials

$\mathcal{PSOPT}$ also has facilities for pseudospectral function approximation using Chebyshev polynomials. Let $T_N(\tau)$ denote the Chebyshev polynomial of order N, which may be generated from:

$$T_N(\tau) = \cos(N \cos^{-1}(\tau)) \tag{1.13}$$

Let τ_k, $k = 0, \ldots, N$ be the Chebyshev-Gauss-Lobatto (CGL) nodes in the interval $[-1, 1]$, which are defined as $\tau_k = -\cos(\pi k/N)$ for $k = 0, 1, \ldots, N$.

Given any real-valued function $f(\tau) : [-1, 1] \to \Re$, it can be approximated by the Chebyshev pseudospectral method:

$$f(\tau) \approx f^N(\tau) = \sum_{k=0}^{N} f(\tau_k)\varphi_k(\tau) \tag{1.14}$$

where the Lagrange interpolating polynomial $\varphi_k(\tau)$ is defined by:

$$\varphi_k(\tau) = \frac{(-1)^{k+1}}{N^2 \bar{c}_k} \frac{(1 - \tau^2)\dot{T}_N(\tau)}{\tau - \tau_k} \tag{1.15}$$

21
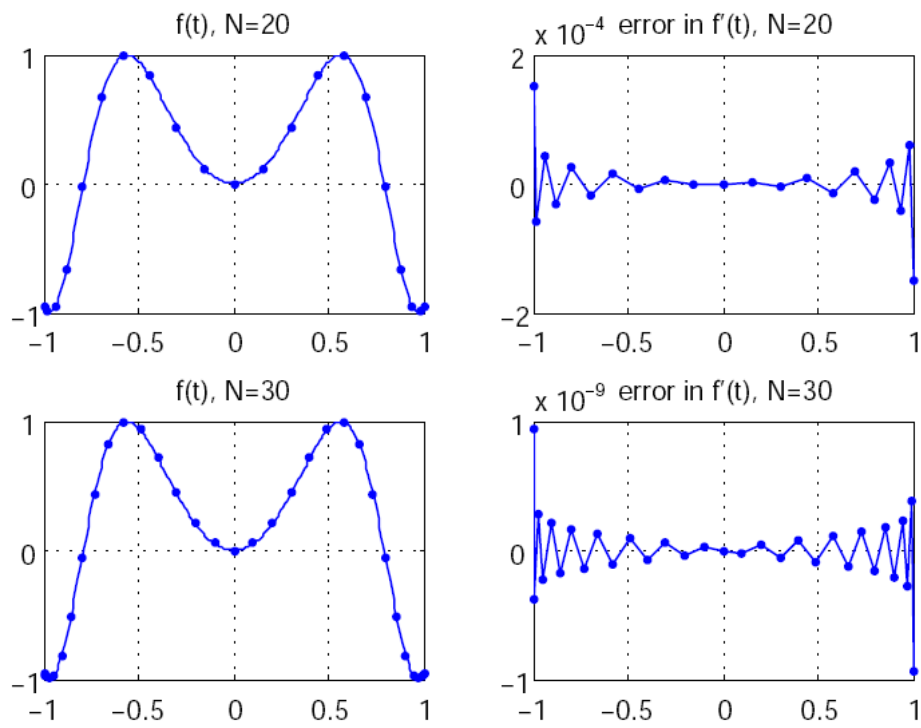
Figure 1.4: Legendre differentiation of $f(t) = \sin(5t^2)$ for $N = 20$ and $N = 30$.
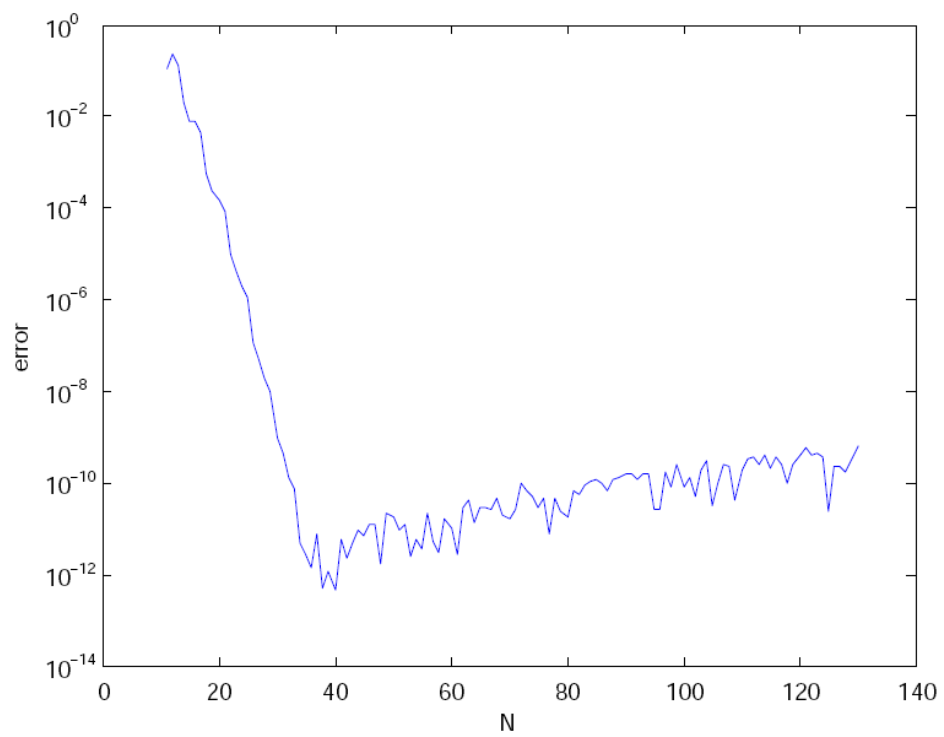
Figure 1.5: Maximum error in the Legendre differentiation of $f(t) = \sin(5t^2)$ as a function of N.

where
$$\bar{c}_k = \begin{cases} 2 & k = 0, N \\ 1 & 1 \leq k \leq N-1 \end{cases} \tag{1.16}$$

It should be noted that $\varphi_k(\tau_j) = 1$ if $k = j$ and $\varphi_k(\tau_j) = 0$, if $k \neq j$, so that:
$$f^N(\tau_k) = f(\tau_k) \tag{1.17}$$

1.6.7   Differentiation with Chebyshev polynomials

The derivatives of $f^N(\tau)$ in terms of $f(\tau)$ at the CGL points τ_k can be obtained by differentiating (1.14). The result can be expressed as a matrix multiplication, such that:

$$\dot{f}(\tau_k) \approx \dot{F}^N(\tau_k) = \sum_{i=0}^{N} D_{ki} f(\tau_i) \tag{1.18}$$

where
$$D_{ki} = \begin{cases} \dfrac{\bar{c}_k}{2\bar{c}_i} \dfrac{(-1)^{k+i}}{\sin[(k+i)\pi/2N]\sin[(k-i)\pi/2N]} & \text{if } k \neq i \\ \dfrac{\tau_k}{2\sin^2[k\pi/N]} & \text{if } i \leq k = i \leq N-1 \\ -\dfrac{2N^2+1}{6} & \text{if } k = i = 0 \\ \dfrac{2N^2+1}{6} & \text{if } k = i = N \end{cases} \tag{1.19}$$

which is known as the differentiation matrix.

1.6.8   Numerical quadrature with the Chebyshev-Gauss-Lobatto method

Note that if $h(\tau)$ is a polynomial of degree $\leq 2N - 1$, its weighted integral over $\tau \in [-1, 1]$ can be exactly computed as follows:

$$\int_{-1}^{1} g(\tau)h(\tau)d\tau = \sum_{k=0}^{N} h(\tau_k)w_k \tag{1.20}$$

where τ_k, $k = 0, \ldots, N$ are the CGL nodes, d the weights w_k are given by:
$$w_k = \begin{cases} \frac{\pi}{2N}, & k = 0, \ldots, N. \\ \frac{\pi}{N}, & k = 1, \ldots, N-1 \end{cases} \tag{1.21}$$

and $g(\tau)$ is a weighting function given by:
$$g(\tau) = \frac{1}{\sqrt{1-\tau^2}} \tag{1.22}$$

If $L(\tau)$ is a general smooth function, then for a suitable N, its weighted integral over $\tau \in [-1, 1]$ can be approximated as follows:

$$\int_{-1}^{1} g(\tau)L(\tau)d\tau \approx \sum_{k=0}^{N} L(\tau_k)w_k \tag{1.23}$$

1.6.9 Differentiation with reduced round-off errors

The following differentiation matrix, which offers reduced round-off errors [10], is employed optionally by $\mathcal{PSOPT}$. It can be used both with Legrendre and Chebyshed points.

$$
D_{jl} = \begin{cases} -\dfrac{\delta_l}{\delta_j} \dfrac{(-1)^{j+l}}{\tau_j - \tau_l} & j \neq l \\ \displaystyle\sum_{i=0, i \neq j}^{N} \dfrac{\delta_i}{\delta_j} \dfrac{(-1)^{i+j}}{\tau_j - \tau_i} & j = l \end{cases} \tag{1.24}
$$

1.7 The pseudospectral discretizations used in $\mathcal{PSOPT}$

To illustrate the pseudospectral discretizations employed in $\mathcal{PSOPT}$, consider the following single phase continuous optimal control problem:

Problem $\mathcal{P}_2$

Find the control trajectories, $u(t), t \in [t_0, t_f]$, state trajectories $x(t), t \in [t_0, t_f]$, static parameters p, and times t_0, t_f, to minimise the following performance index:

$$
J = \varphi[x(t_0), x(t_f), p, t_0, t_f] + \int_{t_0}^{t_f} L[x(t), u(t), p, t]dt
$$

subject to the differential constraints:

$$
\dot{x}(t) = f[x(t), u(t), p, t], \; t \in [t_0, t_f],
$$

the path constraints

$$
h_L \leq h[x(t), u(t), p, t] \leq h_U, \; t \in [t_0, t_f]
$$

the event constraints:

$$
e_L \leq e[x(t_0), u(t_0), x(t_f), u(t_f), p, t_0, t_f] \leq e_U,
$$

the bound constraints on states and controls:

$$
u_L \leq u(t) \leq u_U, \; t \in [t_0, t_f],
$$

$$
x_L \leq x(t) \leq x_U, \; t \in [t_0, t_f],
$$

and the constraints:

$$
\underline{t}_0 \leq t_0 \leq \bar{t}_0,
$$

$$
\underline{t}_f \leq t_f \leq \bar{t}_f,
$$

$$
t_f - t_0 \geq 0,
$$

where

$$u : [t_0, t_f] \to \mathcal{R}^{n_u}$$
$$x : [t_0, t_f] \to \mathcal{R}^{n_x}$$
$$p \in \mathcal{R}^{n_p}$$
$$\varphi : \mathcal{R}^{n_x} \times \mathcal{R}^{n_x} \times \mathcal{R}^{n_p} \times \mathcal{R} \times \mathcal{R} \to \mathcal{R}$$
$$L : \mathcal{R}^{n_x} \times \mathcal{R}^{n_u} \times \mathcal{R}^{n_p} \times [t_0, t_f] \to \mathcal{R}$$
$$f : \mathcal{R}^{n_x} \times \mathcal{R}^{n_u} \times \mathcal{R}^{n_p} \times [t_0, t_f] \to \mathcal{R}^{n_x}$$
$$h : \mathcal{R}^{n_x} \times \mathcal{R}^{n_u} \times \mathcal{R}^{n_p} \times [t_0, t_f] \to \mathcal{R}^{n_h}$$
$$e : \mathcal{R}^{n_x} \times \mathcal{R}^{n_u} \times \mathcal{R}^{n_x} \times \mathcal{R}^{n_u} \times \Re^{n_p} \times \mathcal{R} \times \mathcal{R} \to \mathcal{R}^{n_e}$$

(1.25)

By introducing the transformation:

$$\tau \leftarrow \frac{2}{t_f - t_0} t - \frac{t_f + t_0}{t_f - t_0},$$

it is possible to write problem $\mathcal{P}_3$ using a new independent variable τ in the interval $[-1, 1]$, as follows:

Problem $\mathcal{P}_3$

Find the control trajectories, $u(\tau), \tau \in [-1, 1]$, state trajectories $x(\tau), \tau \in [-1, 1]$, and times t_0, t_f, to minimise the following performance index:

$$J = \varphi[x(-1), x(1), p, t_0, t_f] + \frac{t_f - t_0}{2} \int_{-1}^{1} L[x(t), u(t), p, t] d\tau$$

subject to the differential constraints:

$$\dot{x}(\tau) = \frac{t_f - t_0}{2} f[x(\tau), u(\tau), p, \tau], \; \tau \in [-1, 1],$$

the path constraints

$$h_L \leq h[x(\tau), u(\tau), p, \tau] \leq h_U, \; \tau \in [-1, 1]$$

the event constraints:

$$e_L \leq e[x(-1), u(-1), x(1), u(1), p, t_0, t_f] \leq e_U,$$

the bound constraints on controls and states:

$$u_L \leq u(\tau) \leq u_U, \; \tau \in [-1, 1],$$

$$x_L \leq x(\tau) \leq x_U, \; \tau \in [-1, 1],$$

and the constraints:

$$\underline{t_0} \leq t_0 \leq \bar{t}_0,$$

26

$$\underline{t}_f \leq t_f \leq \bar{t}_f,$$

$$t_f - t_0 \geq 0,$$

The description below refers to the Legendre pseudospectral approximation method. The procedure employed with the Chebyshev approximation method is very similar. In the Legendre pseudospectral approximation of problem $\mathcal{P}_3$, the state $x(\tau)$, $\tau \in [-1, 1]$ is approximated by the N-order Lagrange polynomial $x^N(\tau)$ based on interpolation at the Legendre-Gauss-Lobatto (LGL) quadrature nodes, so that:

$$x(\tau) \approx x^N(\tau) = \sum_{k=0}^{N} x(\tau_k)\phi_k(\tau) \tag{1.26}$$

Moreover, the control $u(\tau)$, $\tau \in [-1, 1]$ is similarly approximated using an interpolating polynomial:

$$u(\tau) \approx u^N(\tau) = \sum_{k=0}^{N} u(\tau_k)\phi_k(\tau) \tag{1.27}$$

Note that, from (1.11), $x^N(\tau_k) = x(\tau_k)$ and $u^N(\tau_k) = u(\tau_k)$. The derivative of the state vector is approximated as follows:

$$\dot{x}(\tau_k) \approx \dot{x}^N(\tau_k) = \sum_{i=0}^{N} D_{ki}x^N(\tau_i), \; i = 0, \ldots N \tag{1.28}$$

where D is the $(N+1) \times (N+1)$ the differentiation matrix given by (??).

Define the following $n_u \times (N+1)$ matrix to store the trajectories of the controls at the LGL nodes:

$$U^N = \begin{bmatrix} u_1(\tau_0) & u_1(\tau_1) & \ldots & u_1(\tau_N) \\ u_2(\tau_0) & u_2(\tau_1) & \ldots & u_2(\tau_N) \\ \vdots & \vdots & \ddots & ,\vdots \\ u_{n_u}(\tau_0) & u_{n_u}(\tau_1) & \ldots & u_{n_u}(\tau_N) \end{bmatrix} \tag{1.29}$$

Define the following $n_x \times (N+1)$ matrices to store, respectively, the trajectories of the states and their derivatives at the LGL nodes:

$$X^N = \begin{bmatrix} x_1(\tau_0) & x_1(\tau_1) & \ldots & x_1(\tau_N) \\ x_2(\tau_0) & x_2(\tau_1) & \ldots & x_2(\tau_N) \\ \vdots & \vdots & \ddots & ,\vdots \\ x_{n_x}(\tau_0) & x_{n_x}(\tau_1) & \ldots & x_{n_x}(\tau_N) \end{bmatrix} \tag{1.30}$$

and

$$\dot{X}^N = \begin{bmatrix} \dot{x}_1(\tau_0) & \dot{x}_1(\tau_1) & \dots & \dot{x}_1(\tau_N) \\ \dot{x}_2(\tau_0) & \dot{x}_2(\tau_1) & \dots & \dot{x}_2(\tau_N) \\ \vdots & \vdots & \ddots & ,\vdots \\ \dot{x}_{n_x}(\tau_0) & \dot{x}_{n_x}(\tau_1) & \dots & \dot{x}_{n_x}(\tau_N) \end{bmatrix} \tag{1.31}$$

From (1.28), X^N and $\dot{X}_N$ are related as follows:

$$\dot{X}^N = X^N D^T \tag{1.32}$$

Now, form the following $n_x \times (N+1)$ matrix with the right hand side of the differential constraints evaluated at the LGL nodes:

$$F^N = \frac{t_0 - t_f}{2} \begin{bmatrix} f_1(x^N(\tau_0), u^N(\tau_0), p, \tau_0) & \dots & f_1(x^N(\tau_N), u^N(\tau_N), p, \tau_N) \\ f_2(x^N(\tau_0), u^N(\tau_0), p, \tau_0) & \dots & f_2(x^N(\tau_N), u^N(\tau_N), p, \tau_N) \\ \vdots & \ddots & \vdots \\ f_{n_x}(x^N(\tau_0), u^N(\tau_0), p, \tau_0) & \dots & f_{n_x}(x^N(\tau_N), u^N(\tau_N), p, \tau_N) \end{bmatrix} \tag{1.33}$$

Now, define the differential defects at the collocation points as the $n_x \times (N+1)$ matrix:

$$\zeta^N = \dot{X}^N - F^N = X^N D^T - F^N \tag{1.34}$$

Define the matrix of path constraint function values evaluated at the LGL nodes:

$$H^N = \begin{bmatrix} h_1(x^N(\tau_0), u^N(\tau_0), p, \tau_0) & \dots & h_1(x^N(\tau_N), p, u^N(\tau_N), \tau_N) \\ h_2(x^N(\tau_0), u^N(\tau_0), p, \tau_0) & \dots & h_2(x^N(\tau_N), u^N(\tau_N), p, \tau_N) \\ \vdots & \ddots & \vdots \\ h_{n_h}(x^N(\tau_0), u^N(\tau_0), p, \tau_0) & \dots & h_{n_h}(x^N(\tau_N), u^N(\tau_N), p, \tau_N) \end{bmatrix} \tag{1.35}$$

The objective function of $\mathcal{P}_3$ is approximated as follows:

$$J = \varphi[x(-1), x(1), p, t_0, t_f] + \frac{t_f - t_0}{2} \int_{-1}^{1} L[x(\tau), u(\tau), p, \tau] d\tau$$

$$\approx \varphi[x^N(-1), x^N(1), p, t_0, t_f] + \frac{t_f - t_0}{2} \sum_{k=0}^{N} L[x^N(\tau_k), u^N(\tau_k), p, \tau_k] w_k \tag{1.36}$$

where the weights w_k are defined in (1.5).

We are now ready to express problem $\mathcal{P}_3$ as a nonlinear programming problem, as follows.

Problem $\mathcal{P}_4$

$$\min_{y} F(y) \tag{1.37}$$

subject to:

$$\begin{aligned} G_l \leq &G(y) \leq G_u \\ y_l \leq &y \leq y_u \end{aligned} \tag{1.38}$$

The decision vector y, which has dimension $n_y = (n_u(N+1) + n_x(N+1) + n_p + 2)$, is constructed as follows:

$$y = \begin{bmatrix} \text{vec}(U^N) \\ \text{vec}(X^N) \\ p \\ t_0 \\ t_f \end{bmatrix} \tag{1.39}$$

The objective function is:

$$F(y) = \varphi[x^N(-1), x^N(1), p, t_0, t_f] + \frac{t_f - t_0}{2} \sum_{k=0}^{N} L[x^N(\tau_k), u^N(\tau_k), p, \tau_k] w_k \tag{1.40}$$

while the constraint function $G(y)$, which is of dimension $n_g = n_x(N+1) + n_h(N+1) + n_e + 1$, is given by:

$$G(y) = \begin{bmatrix} \text{vec}(\zeta^N) \\ \text{vec}(H^N) \\ e[x^N(-1), u^N(-1), x^N(1), u^N(1), p, t_0, t_f] \\ t_f - t_0 \end{bmatrix}, \tag{1.41}$$

The constraint bounds are given by:

$$G_l = \begin{bmatrix} \mathbf{0}_{n_x(N+1)} \\ \text{stack}(h_L, N+1) \\ e_L \\ (\underline{t}_0 - \bar{t}_f) \end{bmatrix}, \quad G_u \begin{bmatrix} \mathbf{0}_{n_x(N+1)} \\ \text{stack}(h_U, N+1) \\ e_U \\ 0 \end{bmatrix}, \tag{1.42}$$

and the bounds on the decision vector are given by:

$$y_l = \begin{bmatrix} \text{stack}(u_l, N+1) \\ \text{stack}(x_L, N+1) \\ p_L \\ \underline{t}_0 \\ \underline{t}_f \end{bmatrix}, \quad y_u \begin{bmatrix} \text{stack}(u_U, N+1) \\ \text{stack}(x_U, N+1) \\ p_U \\ \bar{t}_0 \\ \bar{t}_f \end{bmatrix}, \tag{1.43}$$

where $\text{vec}(A)$ forms a nm-column vector by vertically stacking the columns of the $n \times m$ matrix A, and $\text{stack}(x, n)$ creates a mn-column vector by stacking n copies of column m-vector x.

1.7.1 Costate estimates

Legendre approximation method

$\mathcal{PSOPT}$ implements the following approximation for the costates $\lambda(\tau) \in \Re^{n_x}, \tau \in [-1, 1]$ associated with $\mathcal{P}_3$ [18]:

$$\lambda(\tau) \approx \lambda^N(\tau) = \sum_{k=0}^{N} \lambda(\tau_k)\phi_k(\tau), \tau \in [-1, 1] \tag{1.44}$$

The costate values at the LGL nodes are given by:

$$\lambda(\tau_k) = \frac{\tilde{\lambda}_k}{w_k}, \ k = 0, \dots N \tag{1.45}$$

where w_k are the weights given by (1.5), and $\tilde{\lambda}_k \in \Re^{n_x}, k = 0, \dots, N$ are the KKTs multiplier associated with the collocation constraints $\text{vec}(\zeta^N) = 0$. The KKT multipliers can normally be obtained from the NLP solver, which allows $\mathcal{PSOPT}$ to return estimates of the costate trajectories at the LGL nodes.

It is known from the literature [18] that the costate estimates in the Legendre discretization method sometimes oscillate around the true values. To mitigate this, the estimates are smoothed by taking a weighted average for the estimats at k using the costate estimates at $k-1$, k and $k+1$ obtained from (1.44).

Chebyshev approximation method

$\mathcal{PSOPT}$ implements the following approximation for the costates $\lambda(\tau) \in \Re^{n_x}, \tau \in (-1, 1)$ associated with $\mathcal{P}_3$ at the CGL nodes [33]:

$$\lambda(\tau_k) = \frac{\tilde{\lambda}_k}{\sqrt{1 - \tau_k^2}w_k}, \ k = 0, \dots N - 1 \tag{1.46}$$

where w_k are the weights given by (1.21), and $\tilde{\lambda}_k \in \Re^{n_x}, k = 0, \dots, N$ are the KKTs multiplier associated with the collocation constraints $\text{vec}(\zeta^N) = 0$. Since (1.46) is singular for $\tau_0 = -1$ and $\tau_N = 1$, the estimates of the costates at $\tau = \pm 1$ are found using linear extrapolation. The costate estimates are also smoothed as described in 1.7.1

1.7.2 Discretizing a multiphase problem

It now becomes straightforward to describe the discretization used by $\mathcal{PSOPT}$ in the case of $\mathcal{P}_1$, a problem with multiple phases, to form a nonlinear program-

ming problem like $\mathcal{P}_4$. The decision variables of the NLP associated with $\mathcal{P}_1$ are given by:

$$
y = \begin{bmatrix}
\text{vec}(U^{N,(1)}) \\
\text{vec}(X^{N,(1)}) \\
p^{(1)} \\
t_0^{(1)} \\
t_f^{(1)} \\
\vdots \\
\text{vec}(U^{N,(N_p)}) \\
\text{vec}(X^{N,(N_p)}) \\
p^{(N_p)} \\
t_0^{(N_p)} \\
t_f^{(N_p)}
\end{bmatrix}
\tag{1.47}
$$

where N_p is the number of phases in the problem, and the superindex in parenthesis indicates the phase to which the variables belong. The constraint function $G(y)$ is given by:

$$
G(y) = \begin{bmatrix}
\text{vec}(\zeta^{N,(1)}) \\
\text{vec}(H^{N,(1)}) \\
e[x^{N,(1)}(-1), u^{N,(1)}(-1), x^{N,(1)}(1), u^{N,(1)}(1), p^{(1)}, t_0^{(1)}, t_f^{(1)}] \\
t_f^{(1)} - t_0^{(1)} \\
\vdots \\
\text{vec}(\zeta^{N,(N_p)}) \\
\text{vec}(H^{N,(N_p)}) \\
e[x^{N,(N_p)}(-1), u^{N,(N_p)}(-1), x^{N,(N_p)}(1), u^{N,(N_p)}(1), p^{(N_p)}, t_0^{(N_p)}, t_f^{(N_p)}] \\
t_f^{(N_p)} - t_0^{(N_p)} \\
\Psi
\end{bmatrix},
\tag{1.48}
$$

where Ψ corresponds to the linkage constraints associated with the problem, evaluated at y.

Based on the problem information, it is straightforward (but not shown here) to form the bounds on the decision variables y_l, y_u and the bounds on the constraints function G_l, G_u to complete the definition of the NLP problem associated with $\mathcal{P}_1$.

1.8   Parameter estimation problems

A parameter estimation problem arises when it is required to find values for parameters associated with a model of a system based on observations

from the actual system. These are also called *inverse problems*. The approach used in the $\mathcal{PSOPT}$ implementation uses the same techniques used for solving optimal control problems, with a special objective function used to measure the accuracy of the model for given parameter values.

1.8.1 Single phase case

For the sake of simplicity consider first a single phase problem defined over $t_0 \leq t \leq t_f$ with the dynamics given by a set of ODEs:

$$\dot{x} = f[x(t), u(t), p, t]$$

the path constraints

$$h_L \leq h[x(t), u(t), p, t] \leq h_U$$

the event constraints

$$e_L \leq e[x(t_0), u(t_0), x(t_f), u(t_f), p, t_0, t_f] \leq e_U$$

Consider the following model of the observations (or measurements) taken from the system:

$$y(\theta) = g[x(\theta), u(\theta), p, \theta]$$

where $g : \mathcal{R}^{n_x} \times \mathcal{R}^{n_u} \times \mathcal{R}^{n_p} \times \mathcal{R} \rightarrow \mathcal{R}^{n_o}$ is the observations function, and $y(\theta) \in \mathcal{R}^{n_o}$ is the estimated observation at sampling instant θ. Assume that $\{\tilde{y}\}_{k=1}^{n_s}$ is a sequence of n_s observations corresponding to the sampling instants $\{\theta_k\}_{k=1}^{n_s}$.

The objective is to choose the parameter vector $p \in \mathcal{R}^{n_p}$ to minimise the cost function:

$$J = \frac{1}{2} \sum_{k=1}^{n_s} \sum_{j=1}^{n_o} r_{j,k}^2$$

where the residual $r_{j,k} \in \mathcal{R}$ is given by:

$$r_{j,k} = w_{j,k}[g_j[x(\theta_k), u(\theta_k), p, \theta_k] - \tilde{y}_{j,k}]$$

where $w_{j,k} \in \mathcal{R}, j = 1, \ldots, n_o, k = 1, \ldots, n_s$ is a positive residual weight, g_j is the j-th element of the vector observations function g, and $\tilde{y}_{j,k}$ is the j-th element of the actual observation vector at time instant θ_k.

Note that in the parameter estimation case the times t_0 and t_f are assumed to be fixed. The sampling instants need not coincide with the collocation points, but they must obey the relationship:

$$t_0 \leq \theta_k \leq t_f, \ k = 1, \ldots, n_s$$

1.8.2 Multi-phase case

In the case of a problem with N_p phases, let $t_0^{(i)} \leq t \leq t_f^{(i)}$ be the intervals for each phase, with the dynamics given by a set of ODEs:

$$\dot{x}^{(i)} = f^{(i)}[x(t)^{(i)}, u^{(i)}(t), p^{(i)}, t]$$

the path constraints

$$h_L^{(i)} \leq h^{(i)}[x^{(i)}(t), u^{(i)}(t), p^{(i)}, t] \leq h_U^{(i)}$$

the event constraints

$$e_L^{(i)} \leq e^{(i)}[x^{(i)}(t_0), u^{(i)}(t_0), x^{(i)}(t_f), u^{(i)}(t_f), p^{(i)}, t_0^{(i)}, t_f^{(i)}] \leq e_U^{(i)}$$

Consider the following model of the observations (or measurements) taken from the system for each phase:

$$y^{(i)}(\theta_k^{(i)}) = g^{(i)}[x^{(i)}(\theta_k^{(i)}), u^{(i)}(\theta_k^{(i)}), p^{(i)}, \theta_k^{(i)}]$$

where $g^{(i)} : \mathcal{R}^{n_x^{(i)}} \times \mathcal{R}^{n_u^{(i)}} \times \mathcal{R}^{n_p^{(i)}} \times \mathcal{R} \to \mathcal{R}^{n_o^{(i)}}$ is the observations function for each phase, and $y^{(i)}(\theta_k^{(i)}) \in \mathcal{R}^{n_o^{(i)}}$ is the estimated observation at sampling instant $\theta_k^{(i)}$. Assume that $\{\tilde{y}^{(i)}\}_{k=1}^{n_s^{(i)}}$ is a sequence of $n_s^{(i)}$ observations corresponding to the sampling instants $\left\{\theta_k^{(i)}\right\}_{k=1}^{n_s^{(i)}}$.

The objective is to choose the set of parameter vectors $p^{(i)} \in \mathcal{R}^{n_p^{(i)}}, i \in [1, N_p]$ to minimise the cost function:

$$J = \frac{1}{2} \sum_{i=1}^{N_p} \sum_{k=1}^{n_s^{(i)}} \sum_{j=1}^{n_o^{(i)}} \left[r_{j,k}^{(i)}\right]^2$$

where the residual $r_{j,k}^{(i)} \in \mathcal{R}$ is given by:

$$r_{j,k}^{(i)} = w_{j,k}^{(i)}[g_j^{(i)}[x(\theta_k^{(i)}), u^{(i)}(\theta_k^{(i)}), p^{(i)}, \theta_k^{(i)}] - \tilde{y}_j^{(i)}]$$

where $w_{j,k}^{(i)} \in \mathcal{R}$ is a positive residual weight, $g_j^{(i)}$ is the j-th element of the vector observations function $g^{(i)}$, and $\tilde{y}_j^{(i)}$ is the j-th element of the actual observation vector at time instant $\theta_k^{(i)}$

Note that in the parameter estimation case the times $t_0^{(i)}$ and $t_f^{(i)}$ are assumed to be fixed. The sampling instants need not coincide with the collocation points, but they must obey the relationship:

$$t_0^{(i)} \leq \theta_k^{(i)} \leq t_f^{(i)}, \ k = 1, \ldots, n_s^{(i)}$$

1.8.3 Statistical measures on parameter estimates

$\mathcal{PSOPT}$ computes a residual matrix $[r_{j,k}^{(i)}]$ for each phase i and for the final value of the estimated parameters in all phases $\hat{p} \in \mathcal{R}^{n_p}$, where each element of the residual matrix is related to an individual measurement sample and observation within a phase. $\mathcal{PSOPT}$ also computes the covariance matrix $C \in \mathcal{R}^{n_p \times n_p}$ of the parameter estimates using the method described in [28], which uses a QR decomposition of the Jacobian matrix of the equality and active inequality constraints, together with the Jacobian matrix of the residual vector function (a stack of the elements of the residual matrices for all phases) with respect to all decision variables. In addition $\mathcal{PSOPT}$ computes 95% confidence intervals on the estimated parameters. The upper and lower limits of the confidence interval around the estimated value for parameter $\hat{p}_i$ are computed from [31]:

$$\delta_i = \pm t_{N_s - n_p}^{1 - (\alpha/2)} \sqrt{C_{ii}} \qquad (1.49)$$

where N_s is the total number of individual samples, n_p is the number of parameters, t is the inverse two tailed cumulative t-distribution with confidence level α, and $N - n_p$ degrees of freedom.

The residual matrix, the covariance matrix and the confidence intervals can be used to refine the parameter estimation problem. For instance, if the resulting confidence interval of one particular parameter is found to be small, the value of this parameter can be fixed by the user in a subsequent run, which may improve the estimates other parameters being estimated and reduces the possibility of having an overdetermined problem (i.e. a problem with too many parameters to be estimated). See [38] pages 210-211 for more details.

Notice, however, that the statistical analysis performed is based on a linearization of the model. As a result, the validity of the statistical analysis is dependent on the quality of the linearization and the curvature of the underlying functions being linearized, so care must be taken with the interpretation of results of the statistical analysis of the parameter estimates [38].

1.8.4 Remarks on parameter estimation

- In contrast to continuous optimal control problems, the kind of parameter estimation problem considered involves the evaluation of the objective at a finite number of sampling points. Internally, the values of the state and controls are interpolated over the collocation points to find estimated values at the sampling points. The type of interpolation employed depends on the collocation method specified by the user.

- Note that the sampling instants do not have to be sorted in ascending or descending order. Because of this, it is possible to accommodate problems with non-simultaneous observations of different variables by stacking the measured data and sampling instants for the different variables.

- It is possible to use an alternative objective function where the residuals are weighted with the covariance of the measurements simply by multiplying the observations function and the measurements vectors by the square root of the covariance matrix, see [4], page 221 for more details.

- When defining parameter estimation problems, the user needs to ensure that the underlying nonlinear programming problem has sufficient degrees of freedom. This is particularly important as it is common for parameter estimation problems not to involve any control variables. The number of degrees of freedom is the difference between the number of decision variables and the total number of equality and active inequality constraints. For example, in the case of a single phase problem having n_x differential states, n_u control variables (or algebraic states), n_h equality path constraints, and n_e equality event constraints, the number of relevant constraints is given by:

$$n_c = n_x(N+1) + n_h(N+1) + n_e + 1$$

where N is the degree of the polynomial approximation (in the case of a pseudospectral discretization). The number of decision variables is given by:

$$n_y = n_u(N+1) + n_x(N+1) + n_p + 2$$

The difference is:

$$n_y - n_c = (n_u - n_h)(N+1) + n_p + 1 - n_e$$

For the problem to be solvable it is important that $n_y - n_c \geq 0$, ideally $n_y - n_c \geq 1$. The total numbers of constraints and decision variables are always reported in the terminal window when a $\mathcal{PSOPT}$ problem is run. It should be noted that the nonlinear programming solver (IPOPT or SNOPT) may modify the numbers by eliminating redundant constraints or decision variables. These modifications are also visible when a problem is run.

1.9 Alternative local discretizations

Direct collocation methods that use local information to approximate the functions associated with an optimal control problem are well established [3].

Sometimes, it may be convenient for users to compare the performance and solutions obtained by means of the pseudospectral methods implemented in $\mathcal{PSOPT}$, with local discretization methods. Also, if a given problem cannot be solved by means of a pseudospectral discretization, the user has the option to try the local discretizations implemented in $\mathcal{PSOPT}$. The main impact of using a local discretization method as opposed to a pseudospectral discretization method, is that the resulting Jacobian and Hessian matrices needed by the NLP solver are more sparse with local methods, which facilitates the NLP solution. This becomes more noticeable as the number of grid points increases. The disadvantage of using a local method is that the spectral accuracy in the discretization of the differential constraints offered by pseudospectral methods is lost. Moreover, the accuracy of Gauss type integration employed in pseudospectral methods is also lost if pseudospectral grids are not used.

Note also that local mesh refinement methods are well established. These methods concentrate more grid points in areas of greater activity in the function, which helps improve the local accuracy of the solution. The trapezoidal method has an accuracy of $\mathcal{O}(h^2)$, while the Hermite-Simpson method has an accuracy of $\mathcal{O}(h^4)$, where h is the local interval between grid points. Both the trapezoidal and Hermite-Simpson discretization methods are widely used in computational optimal control, and have solve many challenging problems [3]. When the user selects the trapezoidal or Hermite-Simpson discretizations, and if the initial grid points are not provided, the grid is started with equal spacing between grid points. In these two cases any integrals associated with the problem are computed using the trapezoidal and Simpson quadrature method, respectively.

Additionally, an option is provided to use a differentiation matrix based on the central difference method (which has an accuracy of $\mathcal{O}(h^2)$) in conjunction with pseudospectral grids. The central differences option uses either the LGL or the Chebyshev points and Gauss-type quadrature.

The local discretizations implemented in $\mathcal{PSOPT}$ are described below. For simplicity, the phase index has been omitted and reference is made to single phase problems. However, the methods can also be used with multiphase problems.

1.9.1 Trapezoidal method

With the trapezoidal method [3], the defect constraints are computed as follows:

$$\zeta(\tau_k) = x(\tau_{k+1}) - x(\tau_k) - \frac{h_k}{2}(f_k + f_{k+1}), \tag{1.50}$$

where $\zeta(\tau_k) \in \Re^{n_x}$ is the vector of differential defect constraints at node τ_k, $k = 0, \ldots, N-1$, $h_k = \tau_{k+1} - \tau_k$, $f_k = f[(\tau_k), u(\tau_k), p, \tau_k]$, $f_{k+1} =$

$f[x(\tau_{k+1}), u(\tau_{k+1}), p, \tau_{k+1}]$. This gives rise to $n_x N$ differential defect constraints. In this case, the decision vector for single phase problems is given by equation (1.39), so that it is the same as the one used in the Legendre and Chebyshev methods.

1.9.2 Hermite-Simpson method

With the Hermite-Simpson method [3], the defect constraints are computed as follows:

$$\zeta(\tau_k) = x(\tau_{k+1}) - x(\tau_k) - \frac{h_k}{6}(f_k + 4\bar{f}_{k+1} + f_{k+1}), \qquad (1.51)$$

where

$$\bar{f}_{k+1} = f[\bar{x}_{k+1}, \bar{u}_{k+1}, p, \tau_k + \frac{h_k}{2}]$$

$$\bar{x}_{k+1} = \frac{1}{2}(x(\tau_k) + x(\tau_{k+1})) + \frac{h_k}{8}(f_k - f_{k+1})$$

where $\zeta(\tau_k) \in \Re^{n_x}$ is the vector of differential defect constraints at node τ_k, $k = 0, \ldots, N-1$, $h_k = \tau_{k+1} - \tau_k$, $f_k = f[(\tau_k), u(\tau_k), p, \tau_k]$, $f_{k+1} = f[x(\tau_{k+1}), u(\tau_{k+1}), p, \tau_{k+1}]$, and $\bar{u}_{k+1} = \bar{u}(\tau_{k+1})$ is a vector of midpoint controls (which are also decision variables). This gives rise to $n_x N$ differential defect constraints. In this case, the decision vector for single phase problems is given by

$$y = \begin{bmatrix} \text{vec}(U^N) \\ \text{vec}(X^N) \\ p \\ \text{vec}(\bar{U}^N) \\ t_0 \\ t_f \end{bmatrix} \qquad (1.52)$$

with

$$\bar{U}^N = \begin{bmatrix} \bar{u}_1(\tau_1) & \bar{u}_1(\tau_2) & \ldots & \bar{u}_1(\tau_N) \\ \bar{u}_2(\tau_1) & \bar{u}_2(\tau_2) & \ldots & \bar{u}_2(\tau_N) \\ \vdots & \vdots & \ddots & ,\vdots \\ \bar{u}_{n_u}(\tau_1) & \bar{u}_{n_u}(\tau_2) & \ldots & \bar{u}_{n_u}(\tau_N) \end{bmatrix} \qquad (1.53)$$

so that this decision vector is different from the one used in the Legendre and Chebyshev methods as it includes the midpoint controls.

1.9.3   Central difference method

This method computes the differential defect constraints using equation (1.34), but using a $(N+1) \times (N+1)$ differentiation matrix given by:

$$
\begin{aligned}
D_{0,0} &= -1/h_0 \\
D_{0,1} &= 1/h_0 \\
D_{i-1,i} &= 1/(h_i + h_{i-1}), & i = 2, \ldots N \\
D_{i-1,i-2} &= -1/(h_i + h_{i-1}) & i = 2, \ldots N \\
D_{N,N-1} &= -1/h_{N-1} \\
D_{N,N} &= 1/h_{N-1}
\end{aligned}
$$

where $h_k = \tau_{k+1} - \tau_k$. The method uses forward differences at τ_0, backward differences at τ_N, and central differences at τ_k, $k = 1, \ldots, N-1$. In this case, the decision vector for single phase problems is given by equation (1.39), so that it is the same as the one used in the Legendre and Chebyshev methods. Notice that this discretization has less accuracy at both ends of the interval. This is compensated by the use of pseudospectral grids, which concentrate more grid points at both ends of the interval.

1.9.4   Costate estimates with local discretizations

In the case of the trapezoidal and Hermite-Simpson discretizations, the costates at the discretization nodes are approximated according to the following equation:

$$
\lambda(t_{k+\frac{1}{2}}) \approx \frac{\tilde{\lambda}_k}{2h_k}, \ k = 0, \ldots N-1
$$

where $\lambda(t_{k+\frac{1}{2}})$ is the costate estimate at the midpoint in the interval between t_k and t_k+, $\tilde{\lambda}_k \in \Re^{n_x}$ is the vector of Lagrange multiplers obtained from the nonlinear programming solver corresponding to the differential defect constraints, and $h_k = t_{k+1} - t_k$.

In the case of the central-differences discretization, the costates are estimated as described in section 1.7.1.

1.10   External software libraries used by $\mathcal{PSOPT}$

$\mathcal{PSOPT}$ relies on various of libraries to perform a number of tasks.

1.10.1   BLAS and CLAPACK (or LAPACK)

These are standard and widely used linear algebra packages which can be downloaded in source and binary form from http://www.netlib.org. Some current Linux distrubutions, such as Ubuntu, make it very easy to install both BLAS and LAPACK libraries using a package manager.

1.10.2 DMatrix library

This is a C++ matrix library developed by the author which is included with the distrubution. DMatrix and SparseMatrix objects are used internally in the implementation, while DMatrix objects are used as part of the interface. Documentation on the use of this library is included with the distribution, under the directory `psopt-master /dmatrix/doc`, although the essential use of the DMatrix objects is quite intuitive and can be learned from the examples given in this document. The DMatrix library makes direct use of LAPACK, CXSparse and LUSOL functions.

1.10.3 CXSparse

This is an open source sparse matrix computation library written in C by T. Davis [14], which can be downloaded from
   http://www.cise.ufl.edu/research/sparse/CXSparse/

1.10.4 LUSOL

This is an open source sparse LU factorisation library written by M.A. Saunders and co-workers [22], which can be downloaded from
   http://www.stanford.edu/group/SOL/software/lusol/lusol.zip

1.10.5 IPOPT

IPOPT is an open source C++ package for large-scale nonlinear optimization, which uses an interior point method [45]. IPOPT is the default nonlinear programming algorithm used by $\mathcal{PSOPT}$. IPOPT can be downloaded from
   https://projects.coin-or.org/Ipopt

1.10.6 ADOL-C

ADOL-C is a library for the automatic differentiation of C++ code. It allows to compute automatically the gradients and sparse Jacobians required by $\mathcal{PSOPT}$. At the heart of the ADOL-C library is the "adouble" data type, which can be mostly treated as a C++ "double". A copy of ADOL-C is included with the distribution of $\mathcal{PSOPT}$. Some current Linux distrubutions, such as Ubuntu, make it very easy to install the ADOL-C library and headers using a package manager. It can also be downloaded from:
   https://projects.coin-or.org/ADOL-C

An important thing to keep in mind is that if an intermediate variable within a C++ function depends on one or more `adouble` variables, it should be declared as `adouble`. Conversely, if a C++ variable within a function does not depend on any `adouble` variables, it can be declared as the usual `double` type.

1.10.7 SNOPT (optional)

SNOPT [21] is a software package for solving large-scale optimization problems. SNOPT is a large scale extension of the sequential quadratic programming method. SNOPT is implemented in Fortran 77 and distributed as source code. Commercial and academic licenses of SNOPT can be purchased from

http://www.sbsi-sol-optimize.com/asp/sol_product_snopt.htm

1.10.8 GNUplot (optional)

Gnuplot is a portable command-line driven interactive data and function plotting utility which runs on many computer platforms. The software is freely distributed. The source code can be downloaded from the following page:

http://www.gnuplot.info

Some current Linux distrubutions, such as Ubuntu, make it very easy to install GNUplot using a package manager.

1.11 Supported platforms

$\mathcal{PSOPT}$ has been successfully compiled under the following operating systems and compilers:

- Ubuntu Linux version 11.04. It has been tested on Intel based PC's under the 64 bit variant of this operating system using the GCC compiler version that used in the above releases of Ubuntu.

- Windows 7 using the Microsft Visual Studio Professional 2010 compiler.

It is possible that PSOPT will also compile and work on earlier versions of the above operating systems and compilers, but time and resources do not allow for testing all possible combinations.

The specific versions of the software libraries employed by $\mathcal{PSOPT}$ are given below.

- IPOPT version 3.10.3

- ADOL-C version 2.4.1

- SNOPT version 7 (optional)

1.12   Repository and home page

The downloadable $\mathcal{PSOPT}$ distribution is held in the repository:
    http://code.google.com/p/psopt/downloads/list
    An web page is maintained to provide general information about $\mathcal{PSOPT}$:
http://www.psopt.org.

1.13   Release and build numbering

$\mathcal{PSOPT}$ release numbering is sequential increasing by 1 with every release.
The release number is complemented by the build number which is based
on the date of release as follows YYYY-MM-DD.

1.14   Installing and compiling $\mathcal{PSOPT}$

$\mathcal{PSOPT}$ Release 4 is distributed on a single archive called `psopt-master .zip`.

1.14.1   Ubuntu Linux 14.04

Install the following packages:

1. Compilers: `g++` and `gfortran`.

2. F2C library (packages `f2c`, `libf2c2-dev`, and `libf2c2`).

3. BLAS library (packages `libblas-dev`, `libblas3gf`, `libatlas-base-dev`,
   `libopenblas-base`, `libopenblas-dev`).

4. LAPACK library (packages `liblapack-dev` and `liblapack3gf`).

   The above packages can be installed using the following shell com-
   mand:

   ```
   $     sudo apt-get -y install g++ gfortran f2c libf2c2-dev
   libf2c2 libblas-dev libopenblas-base libopenblas-dev
   libblas3gf libatlas-base-dev liblapack-dev liblapack3gf
   ```

5. Download IPOPT from and extract the package into your Ubuntu
   home directory.
   http://www.coin-or.org/download/source/Ipopt/Ipopt-3.12.3.tgz
   The following shell commands will do this:

```
$ cd $HOME/Downloads
$ wget --continue http://www.coin-or.org/download
/source/Ipopt/Ipopt-3.12.3.tgz
$ cd $HOME
$ tar xzvf ./Downloads/Ipopt-3.12.3.tgz
```

Then install IPOPT following its installation instructions. Ensure that you pass `--enable-static` when calling the `configure` script. It is assumed that IPOPT is installed under `~/Ipopt-3.12.3` at the user's home directory. Note that the IPOPT installation will require downloading a sparse linear solver which should be placed in the appropriate folder under:

`~/Ipopt-3.12.3/ThirdParty`

If you choose to use the open source linear solver MUMPS to work with IPOPT, then you can use the following commands to download it, along with the library METIS (usef for graph colouring), you can run two scripts that come with IPOPT:

```
$ cd $HOME/Ipopt-3.12.3/ThirdParty/Metis
$ ./get.Metis
$ cd $HOME/Ipopt-3.12.3/ThirdParty/Mumps
$ ./get.Mumps
```

Once the third party components are in place, then at the very least, the installation commands for Ipopt are:

```
$ cd $HOME/Ipopt-3.12.3
$ ./configure --enable-static coin_skip_warn_cxxflags=yes
$ make -j
$ make install
```

6. If required, install SNOPT following its installation instructions. If SNOPT is required then it is assumed to be installed under `~/snopt7` at the user's home directory.

7. Download ADOL-C version 2.5.2 from

http://www.coin-or.org/download/source/ADOL-C/ADOL-C-2.5.2.tgz.

and extract it into the user's home directory. This can be done using the commands:

```
$ cd $HOME/Downloads
$ wget --continue www.coin-or.org/download/
source/ADOL-C/ADOL-C-2.5.2.tgz
$ cd $HOME
$ tar zxvf ./Downloads/ADOL-C-2.5.2.tgz
```

and extract it from your home directory as follows:

Download and extract ColPack from

http://cscapes.cs.purdue.edu/download/ColPack/ColPack-1.0.9.tar.gz.

as follows:

```
$ cd $HOME/ADOL-C-2.5.2
$ mkdir ./ThirdParty
$ cd ./ThirdParty
$ wget --continue http://cscapes.cs.purdue.edu/download/
ColPack/ColPack-1.0.9.tar.gz
$ tar zxvf ColPack-1.0.9.tar.gz
$ mv ColPack-1.0.9 ColPack
```

Then enter the following commands (the last two require the root password):

```
$ cd ColPack
$ ./configure
$ make
$ sudo make install
$ sudo cp /usr/local/lib/libCol* /usr/lib
```

Then configure and make ADOL-C as follows:

```
$ cd $HOME/ADOL-C-2.5.2
$ ./configure --enable-sparse
--with-colpack=$HOME/ADOL-C-2.5.2/ThirdParty/ColPack
$ make
$ make install
```

which installs ADOL-C into directory $\tilde{/}$`adolc_base`.

Then copy the installation as follows (this step requires the root password):

```
$ sudo cp $HOME/adolc_base/lib64/*.a /usr/lib
$ sudo cp -r $HOME/adolc_base/include/* /usr/include/
```

8. Install GNUplot (optional but very useful to visualise data) To generate plots as PDF files it may be necessary to compile your own GNUplot binaries by following the easy to follow instructions given in:

http://www.miscdebris.net/blog/2008/01/23/install-gnuplot-on-ubuntu-gutsy-gibbon/

The installation of GNUplot with PDF capabilities can be done through the following shell commands, but it is recommended to read the above page in case the installation of additional libraries is needed.

```
$ cd $HOME/Downloads
$ wget --continue http://www.pdflib.com/binaries/
PDFlib/705/PDFlib-Lite-7.0.5p3.tar.gz
$ tar zxvf PDFlib-Lite-7.0.5p3.tar.gz
$ cd PDFlib-Lite-7.0.5p3
$ ./configure
$ make; sudo make install
$ sudo ldconfig
$ cd $HOME/Downloads
$ wget --continue http://sourceforge.net/projects/gnuplot/
files/gnuplot/4.2.2/gnuplot-4.2.2.tar.gz/download
$ mv download gnuplot-4.2.2.tar.gz
$ tar zxvf gnuplot-4.2.2.tar.gz
$ sudo apt-get -y install libx11-dev libxt-dev
libgd2-xpm-dev libreadline6-dev
$ cd gnuplot-4.2.2
$ ./configure -with-readline=gnu -without-tutorial
$ make;sudo make install
```

All the examples provided have commands to generate PDF plots.

9. Download the $\mathcal{PSOPT}$ distribution from:

https://github.com/PSOPT/psopt/archive/master.zip

44

and extract the $\mathcal{PSOPT}$ archive on a suitable directory (the following commands assume that the archive will be extracted into the user home directory):

```
$ cd $HOME
$ wget --continue https://github.com/PSOPT/psopt
/archive/master.zip
$ unzip master.zip
$ mv master.zip $HOME/Downloads
```

This will create the following directory structure:

```
psopt-master
```
-	dmatrix	
-	-	src
-	-	examples
-	-	doc
-	-	include
-	PSOPT	
-	-	src
-	-	examples
-	-	doc

10. Download the SuiteSparse distribution from

    http://faculty.cse.tamu.edu/davis/SuiteSparse/SuiteSparse-4.4.3.tar.gz

    and copy and extract the archive into the `psopt-master` directory created in the previous step, as follows:

```
$ cd $HOME/psopt-master
$ wget --continue http://faculty.cse.tamu.edu/davis/
SuiteSparse/SuiteSparse-4.4.3.tar.gz
$ tar zxvf SuiteSparse-4.4.3.tar.gz
```

11. Download the LUSOL distribution from

    http://www.stanford.edu/group/SOL/software/lusol/lusol.zip

    and copy and extract the archive into the `psopt-master` directory created in the previous step.

```
$ wget --continue http://www.stanford.edu/group/SOL/
software/lusol/lusol.zip
$ unzip lusol.zip
```

12. Compile the $\mathcal{PSOPT}$ binaries as follows:

```
$ make all
```

This will build all needed libraries and will generate a number of exacutables within the various directories under the `psopt-master`/PSOPT/examples directory. It will also run one of the examples as a test. If you have GNUplot installed you will see a plot appearing on the screen.

The default build configuration is that the SNOPT libraries will not be linked. If the use of SNOPT is also required, then the compilation instruction should be as follows:

```
$ sh ./use_ipopt_and_snopt.sh
$ make all
```

It is assumed that SNOPT root directory is `~/snopt7` at the user's home directory. This will link both the IPOPT and SNOPT libraries into the generated executable. If after selecting the option to link the SNOPT libraries, the user wishes to return to the default build configuration, so that only the IPOPT libraries are linked, then the following script, and subsequent commands, may be run from the `psopt-master` directory:

```
$ sh ./use_ipopt_only.sh
$ make clean
$ make distclean
$ make all
```

13. After successful compilation, $\mathcal{PSOPT}$ is ready to be used. To see some of the examples running, move to the appropriate directory and run the executable file. For example, to run the "launch" example, do as follows:

```
$ cd PSOPT/examples/launch
$ ./launch
```

1.14.2 Microsoft Visual Studio (2010, 2008 or 2005) under Windows XP/Vista/7

1. Download IPOPT from

   http://www.coin-or.org/download/source/Ipopt/Ipopt-3.9.3.zip

   and extract it under the `C:\` folder. Install IPOPT following its installation instructions for Windows, which can be found at:

   `C:\Ipopt-3.9.3\Ipopt\MSVisualStudio\v8\README.TXT`

   Note that the IPOPT installation will require downloading additional third party code (LAPACK, F2C, BLAS, and a sparse linear solver) which should be placed in the appropriate folders under:

   `C:\Ipopt-3.9.3\ThirdParty`

   After building IPOPT, ensure that you have the following libraries:

   ```
   C:\Ipopt-3.9.3\Ipopt\MSVisualStudio\v8\libCoinLapack\Release\libCoinLapack.lib
   C:\Ipopt-3.9.3\Ipopt\MSVisualStudio\v8\libCoinBlas\Release\libCoinBlas.lib
   C:\Ipopt-3.9.3\Ipopt\MSVisualStudio\v8\libIpopt\Release\libIpopt.lib
   ```

   If you are using the HSL solver MA27, then you should also have the following library:

   ```
   C:\Ipopt-3.9.3\Ipopt\MSVisualStudio\v8\libCoinHSL\Release\libCoinHSL.lib
   ```

   The $\mathcal{PSOPT}$ Makefiles assume that IPOPT is installed under `C:\Ipopt-3.9.3`.

2. If SNOPT is required then it is assumed to be installed under `C:\snopt7`. You must build the SNOPT library using the Microsoft Visual C++ compiler. To do this, follow the instructions that can be found in file:

   `C:\snopt7\win32\README.SNOPTCLIB`

   After building the library, you should have the following file:

   `C:\snopt7\win32\snopt.lib`

3. Download the ADOL-C package for Windows from

   `http://www.coin-or.org/download/source/ADOL-C/ADOL-C-2.1.12.zip`.

   Extract the zip file at the root directory of the `C:` drive. The $\mathcal{PSOPT}$ Makefiles and source code assume that ADOL-C is installed at `C:\ADOL-C-2.1.12`.

   You should then follow the installation procedure for Windows given in the file `C:\ADOL-C-2.1.12\windows\README_VC++`. You should download the package ColPack (version 1.0.3) from

   http://www.cscapes.org/coloringpage/software.htm.

and extract the source code into the folder

`C:\ADOL-C-2.1.12\ThirdParty\ColPack.`

Select the "sparse" configuration to build ADOL-C from the Visual Studio Interface, After building ADOL-C from the Visual Studio Solution interface, please ensure that you have the file:

`C:\ADOL-C-2.1.12\windows\sparse\adolc.dll.`

You must also update your system's PATH environment variable to include the folder `C:\ADOL-C-2.1.12\windows\sparse`.

4. Download GNUplot (optional but very useful to visualise data). A binary distribution of GNUplot for Windows can be found at:

`http://www.tatsuromatsuoka.com/gnuplot/Eng/winbin/`

Extract the GNUplot zip archive into the `C:\` folder, and add the following directory to your Windows PATH environment variable

`C:\gp45-winbin\gnuplot\binary`

Important: Note also that the user needs to manually enter the command `quit` at the GNUplot prompt for each plot that is produced.

5. Extract the $\mathcal{PSOPT}$ archive on a suitable directory. This will create the following directory structure under the directory where the archive is extracted:

```
psopt-master
-               dmatrix
-       -               src
-       -               examples
-       -               doc
-       -               include
-               PSOPT
-       -               src
-       -               examples
-       -               doc
```

6. Download the CXSparse distribution from

http://www.cise.ufl.edu/research/sparse/CXSparse/CXSparse.tar.gz

and copy the archive into the `psopt-master` directory created in the previous step. Extract the `CXSparse.tar.gz` archive into the `psopt-master` directory using suitable software.

7. Download the LUSOL distribution from

http://www.stanford.edu/group/SOL/software/lusol/lusol.zip

and copy the archive into the `psopt-master` directory created in the previous step. Extract the `lusol.zip` archive into the `psopt-master` directory using suitable compression software.

8. The default configuration is to use Microsoft Visual Studio 2010 under 64 bit version of Microsoft Windows 7. To change the default configuration, it is necessary to edit the file

   `psopt-master \MicrosoftVisualStudio.inc`

   and to uncomment the lines as indicated in the comments:

```
# SUITABLE COMPILERS: MICROSOFT VISUAL STUDIO 2005 and 2008
#
# SUITABLE WINDOWS VERSIONS: XP, VISTA AND 7 (32 and 64 BIT)
#
#
#
# NOTE: THE DEFAULT CONFIGURATION IS:
# MICROSOFT VISUAL STUDIO 2010 UNDER A 64 BIT VERSION OF WINDOWS
#

# DEFAULT CONFIGURATION: MS VISUAL STUDIO 2010, 64 BIT WINDOWS

# IMPORTANT: COMMENT OUT THESE THREE LINES IF THIS VERSION IS NOT NOT USED

VCINCLUDE = "C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\include"

WINDOWS_H = "C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Include"

VCSDK  = "C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Include"


# UNCOMMENT THE FOLLOWING LINES TO USE MS VISUAL STUDIO 2008, 64 BIT WINDOWS

# IMPORTANT: COMMENT OUT THESE THREE LINES IF THIS VERSION IS NOT NOT USED

# VCINCLUDE = "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\include"

# WINDOWS_H = "C:\Program Files (x86)\Microsoft Visual Studio 9.0\SmartDevices\SDK\Smartphone2003\Include"

# VCSDK  = "C:\Program Files (x86)\Microsoft Visual Studio 9.0\SmartDevices\SDK\Smartphone2003\Include"


# UNCOMMENT THE THREE LINES BELOW TO USE MS VISUAL STUDIO 2008 ON A 32 BIT VERSION OF WINDOWS

# VCINCLUDE = "C:\Program Files\Microsoft Visual Studio 9.0\VC\include"

# WINDOWS_H = "C:\Program Files\Microsoft Visual Studio 9.0\SmartDevices\SDK\Smartphone2003\Include"

# VCSDK  = "C:\Program Files\Microsoft Visual Studio 9.0\SmartDevices\SDK\Smartphone2003\Include"


# UNCOMMENT THE THREE LINES BELOW TO USE MS VISUAL STUDIO 2005 UNDER A 32 BIT VERSION OF WINDOWS

# VCINCLUDE = "C:\Program Files\Microsoft Visual Studio 8\VC\include"

# VCSDK     = "C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\Include"

# WINDOWS_H = "C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\Include"


# UNCOMMENT THE THREE LINES BELOW TO USE MS VISUAL STUDIO 2005 UNDER A 64 BIT VERSION OF WINDOWS

# VCINCLUDE = "C:\Program Files (x86)\Microsoft Visual Studio 8\VC\include"

# VCSDK     = "C:\Program Files (x86)\Microsoft Visual Studio 8\VC\PlatformSDK\Include"

# WINDOWS_H = "C:\Program Files (x86)\Microsoft Visual Studio 8\VC\PlatformSDK\Include"
```

9. Users of Microsoft Visual Studio 2008 need to edit the file `Psopt3\Makefile.vc`, such that line 16 is uncommented:

```
cscript update_myblas_h.vbs
```

10. Open a Visual Studio Command Prompt and compile the $\mathcal{PSOPT}$ binaries as follows:

```
> cd psopt-master
> nmake -f Makefile.vc all
```

This will build all libraries that need to be built and will generate a number of exacutables within the various directories under the `psopt-master\PSOPT\examples` directory.

The default build configuration is that the SNOPT libraries will not be linked. If the use of SNOPT is required, then the compilation instruction should be as follows:

```
> use_ipopt_and_snopt.bat
> nmake -f Makefile.vc all
```

It is assumed that the SNOPT root directory is `C:\snopt7`. If after selecting the option to link the SNOPT libraries the user wishes to return to the default build configuration, so that only the IPOPT libraries are linked, then the following script, and subsequent commands, may be run from the `psopt-master` directory:

```
> use_ipopt_only.bat
> nmake -f Makefile.vc clean
> nmake -f Makefile.vc distclean
> nmake -f Makefile.vc all
```

11. After successful compilation, $\mathcal{PSOPT}$ is ready to be used. To see some of the examples running, move to the appropriate directory and run the executable file. For example, to run the "launch" example, do as follows:

```
> cd PSOPT\examples\launch
> launch.exe
```

1.15   Limitations and known issues

1. The discretization techniques used by $\mathcal{PSOPT}$ give approximate solutions for the state and control trajectories. The software is intended to be used for problems where the control variables are continuous (within a phase) and the state variables have continuous derivatives (within a phase). If within a phase the solution to the optimal control problem is of a different nature, the results may be incorrect or the optimization algorithm may fail to converge. Furthermore, $\mathcal{PSOPT}$ may not be suitable for solving problems involving differential-algebraic equations with index greater than one. Some of these issues can be avoided by reformulating the problem to have several phases.

2. The solution obtained by $\mathcal{PSOPT}$ corresponds to a local minimum of the discretized optimization problem. If the problem is suspected to have several local minima, then it may be worth trying various initial guesses.

3. The automatic scaling procedures work well for all the examples provided. However, note that the scaling of variables depends on the user provided bounds. If these bounds are not adequate for the problem, then the resulting scaling may be poor and this may lead to incorrect results or convergence problems. In some cases, users may need to provide the scaling factors manually to obtain satisfactory results.

4. The automatic mesh refinement procedures require an initial guess for the number of nodes in the global case (the number and/or initial distribution of nodes in the local case). If this initial guess is not adequate (e.g. the grid is too coarse or to dense), the mesh refinement procedure may fail to converge. In some cases, the user may need to manually tune some of the parameters of the mesh refinement procedure to achieve satisfactory results.

5. The efficiency with which the optimal control problem is solved depends in a good deal on the correct formulation of the problem. Unsuitable formulations may lead to trouble in finding a solution. Moreover, if the constraints are such that the problem is infeasible or if for any other reason the solution does not exist, then the nonlinear programming algorithm will fail.

6. The user supplied functions which define the cost function, DAE's, event and linkage constraints, are all assumed to be continuous and to have continuous first and second derivatives. Non-differentiable functions may cause covergence problems to the optimization algorithm. Moreover, it is known that discontinuities in the second derivatives may also cause convergence problems.

7. Only single phase problems are supported if the dynamics involve delays in the states or controls.

8. Note that the constraints associated with the problem are only enforced at the discretization nodes, not in the interval between the nodes.

9. When the problem requires a large number of nodes (say over 200) the nonlinear programming algorithm may have problems to converge if global collocation is being used. This may be due to numerical difficulties within the nonlinear programming solver as the Jacobian (and Hessian) matrices may not be sufficiently sparse. This occurs because the pseudospectral differentiation matrices are dense. When faced with this problem the user may wish to try the local collocation options availble within $\mathcal{PSOPT}$, or to split the problem into multiple segments to increase the sparsity of the derivatives. Note that the sparsity of the Jacobian and Hessian matrices is problem dependent.

10. The co-state approximations resulting from the Legendre pseudospectral method are not as accurate as those obtained by means of the Gauss pseudospectral method [2]. Moreover, the co-state approximations obtained by $\mathcal{PSOPT}$ using the Chebyshev pseudospectral methods are rather innacurate close to the edges of the time interval within each phase. Also the co-state approximation used in the the case of local discretizations (trapezoidal, Hermite-Simpson) converges at a lower rate (is less accurate) than the states or the controls.

11. Sometimes there are crashes when computing sparse derivatives with ADOL-C if the number of NLP variables is very large. This can be avoided by switching to numerical differentiation.

Chapter 2

Defining optimal control and estimation problems for $\mathcal{PSOPT}$

Defining an optimal control or parameter estimation problem involves specifying all the necessary values and functions that are needed to solve the problem. With $\mathcal{PSOPT}$, this is done by implementing C++ functions (e.g. the cost function), and assigning values to data structures which are described below. Once a $\mathcal{PSOPT}$ has obtained a solution, the relevant variables can be obtained by interrogating a data structure.

2.1 Interface data structures

The role of each structure used in the $\mathcal{PSOPT}$ interface is summarised below.

- *Problem data structure:* This structure is used to specify problem information, including the number of phases and pointers to the relevant functions, as well as phase related information such as number of states, controls, parameters, number of grid points, bounds on variables (e.g. state bounds), and functions (e.g. path function bounds).

- *Algorithm data structure:* This is used to control the solution algorithm and to pass parameters to the NLP solver.

- *Solution data structure:* This is used to store the resulting variables of a $\mathcal{PSOPT}$ run.

2.2 Required functions

Table 2.1 lists and describes the parameters used by the interface functions.

Parameter	Type	Role	Description
controls	adouble*	input	Array of intantaneous controls
derivatives	adouble*	output	Array of intantaneous state derivatives
e	adouble*	output	Array of event constraints
final_states	adouble*	input	Array of final states within a phase
initial_states	adouble*	input	Array of initial states within a phase
iphase	int	input	Phase index (starting from 1)
linkages	adouble*	output	Array of linkage constraints
parameters	adouble*	input	Array of static parameters within a phase
states	adouble*	input	Array of intantaneous states within a phase
time	adouble	input	Instant of time within a phase
t0	adouble	input	Initial phase time
tf	adouble	input	final phase time
xad	adouble*	input	vector of scaled decision variables
workspace	Workspace*	input	Pointer to workspace structure

Table 2.1: Description of parameters used by the $\mathcal{PSOPT}$ interface functions

2.2.1   endpoint_cost function

The purpose of this function is to specify the terminal costs $\phi_i[\cdot]$, $i = 1, \ldots, N$. The function prototype is as follows:

```
adouble endpoint_cost(adouble* initial_states,
                      adouble* final_states,
                      adouble* parameters,
                      adouble& t0, adouble& tf,
                      adouble* xad, int iphase,
                      Workspace* workspace)
```

The function should return the value of the end point cost, depending on the value of phase index iphase, which takes on values between 1 and problem.nphases.

Example of writing the endpoint cost function for a single phase problem with the following endpoint cost:

$$\varphi(x(t_f)) = x_1(t_f)^2 + x_2(t_f)^2 \tag{2.1}$$

```
adouble endpoint_cost(adouble* initial_states,
                      adouble* final_states,
                      adouble* parameters,
                      adouble& t0, adouble& tf,
                      adouble* xad, int iphase,
```

```
                     Workspace* workspace)
{
    adouble x1f = final_states[ CINDEX(1) ];
    adouble x2f = final_states[ CINDEX(2) ];


    return ( x1f*x1f + x2f*x2f);
}
```

2.2.2 `integrand_cost` function

The purpose of this function is to specify the integrand costs $L_i[\cdot]$ for each
phase as a function of the states, controls, static parameters and time. The
function prototype is as follows:

```
adouble integrand_cost(adouble* states,
                       adouble* controls,
                       adouble* parameters,
                       adouble& time,
                       adouble* xad,
                       int iphase,
                       Workspace* workspace)
```

The function should return the value of the integrand cost, depending on the
phase index `iphase`, which takes on values between 1 and `problem.nphases`.

Example of writing the integrand cost for a single phase problem with
L given by:

$$L(x(t), u(t), t) = x_1(t)^2 + x_2(t)^2 + 0.01u(t)^2 \qquad (2.2)$$

```
adouble integrand_cost(adouble* states,
                       adouble* controls,
                       adouble* parameters,
                       adouble& time,
                       adouble* xad,
                       int iphase,
                       Workspace* workspace)
{
    adouble x1 = states[   CINDEX(1) ];
    adouble x2 = states[   CINDEX(2) ];
    adouble  u = controls[ CINDEX(1) ];
```

```
    return ( x1*x1 + x2*x2 + 0.01*u*u );
}
```

If the problem does not involve any cost integrand, the user may simply not register any `cost_integrand` function, or register it as follows: `problem.cost_integrand=NULL`.

2.2.3 `dae` function

This function is used to speficy the time derivatives of the states $\dot{x}^{(i)} = f_i[\cdot]$ for each phase as a function of the states themselves, controls, static parameters, and time, as well as the algebraic functions related to the path constraints. Its prototype is as follows:

```
void dae(adouble* derivatives,
        adouble* path,
        adouble* states,
        adouble* controls,
        adouble* parameters,
        adouble& time,
        adouble* xad,
        int iphase,
        Workspace* workspace)
```

This is an example of writing the `dae` function for a single phase problem with the following state equations and path constraints:

$$
\dot{x}_1 = x_2
$$
$$
\dot{x}_2 = -3\exp(x_1) - 4x_2 + u \qquad (2.3)
$$
$$
0 \le x_1^2 + x_2^2 \le 1
$$

Note that the bounds are specified separately.

```
void dae(adouble* derivatives,
        adouble* path,
        adouble* states,
        adouble* controls,
        adouble* parameters,
        adouble& time,
        adouble* xad,
        int iphase,
```

```
        Workspace* workspace)
{
    adouble x1 = states[   CINDEX(1) ];
    adouble x2 = states[   CINDEX(2) ];
    adouble  u = controls[ CINDEX(1) ];

    derivatives[ CINDEX(1) ] = x2;
    derivatives[ CINDEX(2) ] = -3*exp(x1)-4*x2+u;
    path[        CINDEX(1) ] = x1*x1 + x2*x2
}
```

2.2.4   events function

This function is used to speficy the values of the event constraint functions $e_i[\cdot]$ for each phase. Its prototype is as follows:

```
void events(adouble* e,
            adouble* initial_states,
            adouble* final_states,
            adouble* parameters,
            adouble& t0,
            adouble& tf,
            int iphase,
            Workspace* workspace)
```

The following is an example of writing the **events** function for a single phase problem with the event constraints:

$$
\begin{aligned}
1 &= e_1(t_0) = x_1(t_0) \\
2 &= e_2(t_0) = x_2(t_0) \\
-0.1 &\leq e_3(t_0) = x_1(t_f)x_2(t_f) \leq 0.1
\end{aligned}
\tag{2.4}
$$

Note that the bounds are specified separately.

```
void events(adouble* e,
            adouble* initial_states,
            adouble* final_states,
            adouble* parameters,
            adouble& t0,
            adouble& tf,
            int iphase,
            Workspace* workspace)
{
    adouble x1i = initial_states[   CINDEX(1) ];
```

```
    adouble x2i = initial_states[   CINDEX(2) ];
    adouble x1f = final_states[     CINDEX(1) ];
    adouble x2f = final_states[     CINDEX(2) ];


    e[ CINDEX(1) ] = x1i;
    e[ CINDEX(2) ] = x2i;
    e[ CINDEX(3) ] = x1f*x2f;
}
```

2.2.5 `linkages` function

This function is used to speficy the values of the phase linkage constraint functions $\Psi[\cdot]$. Its prototype is as follows:

```
void linkages( adouble* linkages,
               adouble* xad,
               Workspace* workspace)
```

The following is an example of writing the `linkages` function for a two phase problem with two states in each phase and with the linkage constraints:

$$0 = \Psi_1 = x_1^{(0)}(t_f^{(1)}) - x_1^{(2)}(t_i^{(2)})$$
$$0 = \Psi_2 = x_2^{(1)}(t_f^{(1)}) - x_2^{(2)}(t_i^{(2)}) \tag{2.5}$$
$$0 = \Psi_3 = t_f^{(1)} - t_i^{(2)}$$

Note that the bounds are specified separately. These type of state and time continuity constraint can be entered automatically by using the `auto_link` function as shown below.

```
void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
    int index = 0;
    // Link phases 1 and 2
    auto_link( linkages, &index, xad, 1, 2 );
}
```

It is of course also possible to implement more general linkage constraints, as illustrated through the following example.

58

Consider a two phase problem with two states in each phase and with the nonlinear linkage constraints:

$$0 = \Psi_1 = x_1^{(1)}(t_f^{(1)}) - \sin[x_1^{(2)}(t_i^{(2)})]$$

$$0 = \Psi_2 = x_2^{(1)}(t_f^{(1)}) - \cos[x_2^{(2)}(t_i^{(2)})] \qquad (2.6)$$

$$0 = \Psi_3 = t_f^{(1)} - t_i^{(2)}$$

```
void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
    adouble xf_p0[  2  ];
    adouble xi_p1[  2  ];
    adouble tf_p0;
    adouble ti_p1;

    get_final_states(   xf_p0, xad, 1 );
    get_initial_states( xf_p1, xad, 2 );
    tf_p0 = get_final_time( xad, 1);
    ti_p1 = get_initial_time( xad, 2);

    linkages[CINDEX(1)]=xf_p0[CINDEX(1)]-sin(xi_p1[CINDEX(1)]);
    linkages[CINDEX(2)]=xf_p0[CINDEX(2)]-cos(xi_p1[CINDEX(2)]);
    linkages[CINDEX(3)]=tf_p0 - ti_p1;
}
```

2.2.6   Main function

Declaration of data structures

The `main()` function is used to declare and initialise the `problem`, `solution`, and `algorithm` data structures, to call the $\mathcal{PSOPT}$ algorithm, and to post-process the results. To declare the data structures the user may wish to use the following commands:

```
Alg  algorithm;
Sol  solution;
Prob problem;
```

Problem level information

The user should then define the problem name as follows:

```
problem.name = "My problem";
```

The number of phases and the number of linkage constraints should be declared afterwards. For example, for a single phase problem:

```
problem.nphases=1;
problem.nlinkages=0;
```

This declaration should be followed by the following function call, which initialises problem level structures.

```
psopt_level1_setup(problem);
```

Phase level information

After this, the user needs to specify phase level parameters using the following syntax:

```
problem.phases(iphase).nstates             = INTEGER;
problem.phases(iphase).ncontrols           = INTEGER;
problem.phases(iphase).nevents             = INTEGER;
problem.phases(iphase).npath               = INTEGER;
problem.phases(iphase).nodes               = INTEGER OR STRING;
```

The number of nodes in the grid can be speficied as a single value (e.g. 30), or a character string with the node sequence to be tried (if manual mesh refinement is being used): "[30, 50, 60]".

Once the phase level dimensions have been specified it is necessary to call the following function:

```
psopt_level2_setup(problem, algorithm);
```

Phase bounds information

The syntax to enter state bounds is as follows:

```
problem.phases(iphase).bounds.lower.states(j) = REAL;
problem.phases(iphase).bounds.upper.states(j) = REAL;
```

where `iphase` is a phase index between 1 and `problem.nphases`, and `j` is an index between 1 and `problem.phases(iphase).nstates`.

The syntax to enter control bounds is as follows:

```
problem.phases(iphase).bounds.lower.controls(j) = REAL;
problem.phases(iphase).bounds.upper.controls(j) = REAL;
```

where `j` is an index between 1 and `problem.phases(iphase).ncontrols`.

The syntax to enter event bounds is as follows:

```
problem.phases(iphase).bounds.lower.events(j) = REAL;
problem.phases(iphase).bounds.upper.events(j) = REAL;
```

where j is an index between 1 and `problem.phases(iphase).nevents`.

The bounds on the start time for each phase are entered using the following syntax:

```
problem.phases(iphase).bounds.lower.StartTime = REAL;
problem.phases(iphase).bounds.upper.StartTime = REAL;
```

The bounds on the end time for each phase are entered using the following syntax:

```
problem.phases(iphase).bounds.lower.EndTime = REAL;
problem.phases(iphase).bounds.upper.EndTime = REAL;
```

Linkage bounds information

The syntax to enter linkage bounds values is as follows:

```
problem.bounds.lower.linkage(j) = REAL;
problem.bounds.upper.linkage(j) = REAL;
```

j is an index between 1 and `problem.nlinkages`. The default value of the linkage bounds is zero.

Specifying the initial guess for each phase

The user may wish to specify an initial guess for the solution, rather than allowing $\mathcal{PSOPT}$ to determine the initial guess automatically. The user may specify any of the following for each phase: the control vector history, the state vector history and the time vector corresponding to the control and state histories, as well as a guess for the static parameter vector.

To specify the initial guesses for each phase, the user needs to create DMatrix objects to hold the initial guesses, and then assign the addresses of these objects to relevant pointers within the Guess structure.

The syntax to specify the initial guess in a particular phase is as follows:

```
problem.phases(iphase).guess.controls     = uGuess;
problem.phases(iphase).guess.states       = xGuess;
problem.phases(iphase).guess.parameters   = pGuess;
problem.phases(iphase).guess.time         = tGuess;
```

where uGuess, xGuess, pGuess and tGuess are DMatrix objects that contain the relevant guesses. Users may find it useful to employ the functions `zeros`, `ones`, and `linspace` to specify the initial guesses.

For example, the following code creates an object to store an initial control guess with 20 grid points, and assigns zeros to it.

```
DMatrix uGuess = zeros(1, 20);
```

The following code defines a linear history in the interval $[10, 15]$ for the first state, and a constant history for the second state, for a system with two states assuming 20 grid points:

```
DMatrix xGuess(2,20);
xGuess(1,colon()) = linspace( 10, 15, 20);
xGuess(2,colon()) = 10*ones( 1, 20);
```

The following code defines a time vector with equally spaced values in the interval $[0, 10]$ assuming 20 grid points:

```
DMatrix tGuess = linspace( 0, 10, 20);
```

Scaling information

The user may wish to supply the scaling factors rather than allowing $\mathcal{PSOPT}$ to compute them automatically.

Scaling factors for controls, states, event constraints, derivatives, path constraints, and time for each phase can be entered as follows:

```
problem.phases(iphase).scale.controls(j)      = REAL;
problem.phases(iphase).scale.states(j)        = REAL;
problem.phases(iphase).scale.events(j)        = REAL;
problem.phases(iphase).scale.defects(j)       = REAL;
problem.phases(iphase).scale.time             = REAL;
```

The scaling factor for the objective function is entered as follows:

```
problem.scale.objective                       = REAL;
```

Scaling factors for the linkage constraints are entered as follows:

```
problem.scale.linkage(j)                       = REAL;
```

Specifying algorithm options

Algorithm options and parameters can be specified as follows:

```
algorithm.nlp_method                      = STRING;
algorithm.scaling                         = STRING;
algorithm.defect_scaling                  = STRING;
algorithm.derivatives                     = STRING;
algorithm.collocation_method              = STRING;
algorithm.nlp_iter_max                    = INTEGER;
algorithm.nlp_tolerance                   = REAL;
algorithm.print_level                     = INTEGER;
algorithm.jac_sparsity_ratio              = REAL;
algorithm.hess_sparsity_ratio             = REAL;
algorithm.hessian                         = STRING;
algorithm.mesh_refinement                 = STRING;
algorithm.ode_tolerance                   = REAL;
algorithm.mr_max_iterations               = INTEGER;
algorithm.mr_min_extrapolation_points     = INTEGER;
algorithm.mr_initial_increment            = INTEGER;
algorithm.mr_kappa                        = REAL;
algorithm.mr_M1                           = INTEGER;
algorithm.switch_order                    = INTEGER;
algorithm.mr_max_increment_factor         = REAL;
```

Note that:

- `algorithm.nlp_method` takes the options "IPOPT" (default) or "SNOPT".

- `algorithm.scaling` takes the options "automatic" (default) or "user".

- `algorithm.defect_scaling` takes the options "state-based" (default) or "jacobian-based".

- `algorithm.derivatives` takes the options "automatic" (default) or "numerical".

- `algorithm.collocation_method` takes the options "Legendre" (default), "Chebyshev", "trapezoidal", or "Hermite-Simpson".

- `algorithm.diff_matrix` takes the options "standard" (default), "reduced-roundoff", or "central-differences".

- `algorithm.print_level` takes the values 1 (default), which causes $\mathcal{PSOPT}$ and the NLP solver to print information on the screen, or 0 to supress all output.

- `algorithm.nlp_tolerance` is a real positive number that is used as a tolerance to check convergence of the NLP solver (default 10^{-6}).

- `algorithm.jac_sparsity_ratio` is a real number in the interval (0,1] which indicates the maximum Jacobian density, which is ratio of nonzero elements to the total number of elements of the NLP constraint Jacobian matrix (default 0.5).

- `algorithm.hess_sparsity_ratio` is a real number in the interval (0,1] which indicates the maximum Hessian density, ratio of nonzero elements to the total number of elements of the NLP Hessian matrix (default 0.2).

- `algorithm.hessian` takes the options "reduced-memory" or "exact". The "exact" option is only used together with the IPOPT NLP solver.

- `algorithm.nsteps_error_integration` is an integer number that gives the number of integration steps to be taken within each interval when calculating the relative ODE error. The default value is 10.

- `algorithm.mesh_refinement` takes the values "manual" (default) or "automatic".

- `algorithm.ode_tolerance` is a small real value that is used as one of the stopping criteria for mesh refinement. If the maximum relative ODE error falls below this value, the mesh refinement iterations are terminated. The default value is 10^{-3}.

- `algorithm.mr_max_iterations` is a positive integer with the maximum number of mesh refinement iterations (default 7).

- `algorithm.mr_min_extrapolation_points` is the minimum number points to use to calculate the regression that is employed to extrapolate the number of nodes. This is only used if a global collocation method is employed (default 2).

- `algorithm.mr_initial_increment` is a positive integer with the initial increment in the number of nodes. This is only used if a global collocation method is employed (default 10).

- `algorithm.mr_kappa` is a positive real number used by the local mesh refinement algorithm (default 0.1).

- `algorithm.mr_M1` is a positive integer used by the local mesh refinement algorithm (default 5).

- `algorithm.switch_order` is a positive integer indicating the local mesh refinement iteration after which the order is switched from 2 (trapezoidal) to 4 (Hermite-Simpson). If the entered value is zero, then the order is not switched and the collocation method specified through

the option `algorithm.collocation_method` is used in all mesh refinement iterations. This option only applies if a local collocation method is specified (default 2).

- `algorithm.mr_max_increment_factor` is a positive real number in the range $(0, 1]$ used by the mesh refinement algorithms (default 0.4).

Calling $\mathcal{PSOPT}$

Once everything is ready, then the `psopt` algorithm can be called as follows:

```
psopt(problem, solution, algorithm);
```

Error checking

$\mathcal{PSOPT}$ will set `solution.error_flag` to "true" if an run time error is caught. This flag can be checked for errors so that appropriate action can be taken once $\mathcal{PSOPT}$ returns. A diagostic message will be printed on the screen. The diagnostic message can also be recovered from `solution.error_message`. Moreover, the error message is printed to file `error_message.txt`. $\mathcal{PSOPT}$ checks automatically many of the user supplied parameters and will return an error if an inconsisetency is found. The following example shows a call to $\mathcal{PSOPT}$, followed by error checking (in this case, the program exits with code 1 if the error flag is true).

```
psopt(problem, solution, algorithm);
if (solution.error_flag)
{
    exit(1);
}
```

Postprocessing the results

The `psopt()` function returns the results of the optimisation within the `solution` data structure. The results may then be post-processed.

For example, to save the time, control and state vectors of the first phase, the user may use the following commands:

```
DMatrix x = solution.get_states_in_phase(1);
DMatrix u = solution.get_controls_in_phase(1);
DMatrix t = solution.get_time_in_phase(1);

x.Save("x.dat");
u.Save("u.dat");
t.Save("t.dat");
```

Plotting with the GNUplot interface

If the software GNUplot is available in the system where $\mathcal{PSOPT}$ is being run, then the user may employ the `plot()`, `multiplot()`, `surf()`, `plot3()` and `polar()` functions, which constitute a simple interface to GNUplot implemented within the $\mathcal{PSOPT}$ library.

The prototype of the `plot()` function is as follows:

```
void plot(DMatrix& x,
          DMatrix& y,
          const string& title,
          char* xlabel,
          char* ylabel,
          char* legend=NULL,
          char* terminal=NULL,
          char* output=NULL);
```

where x is a column or row vector with n elements and y is a matrix with one either row or column dimension equal to n. `xlabel` is a string with the label for the x-axis, `ylabel` is a string with the label for y-axis, `legend` is a string with the legends for each curve that is plotted, separated by commas. `terminal` is a string with the GNUplot terminal to be used (see Table 2.2), and `output` is a string with the filename to be used for the output, if any.

The function is overloaded, such that the user may plot together curves generated from different x, y pairs, up to three pairs. The additional prototypes are as follows:

```
void plot(DMatrix& x1, DMatrix& y1,
          DMatrix& x2, DMatrix& y2,
          const string& title,
          char* xlabel,
          char* ylabel,
          char* legend=NULL,
          char* terminal=NULL,
          char* output=NULL);

void plot(DMatrix& x1, DMatrix& y1,
          DMatrix& x2, DMatrix& y2,
          DMatrix& x3, DMatrix& y3,
          const string& title,
          char* xlabel,
          char* ylabel,
          char* legend=NULL,
          char* terminal=NULL,
          char* output=NULL);
```

For example, if the user wishes to display a plot of the control trajectories of a system with two control variables which have been stored in DMatrix object "u", and assuming that the corresponding time vector has been stored in DMatrix object "t", then an example of the syntax to call the `plot()`function is:

```
plot(t,u,"Control variable","time (s)", "u", "u1 u2");
```

It is also possible to save plots to graphical files supported by GNU-plot. For example, to save the above plot to an encapsulated postscript file (instead of displaying it), the command is as follows:

```
plot(t,u,"Control variable", "time (s)", "u", "u1 u2",
          "postscript eps", "filename.eps");
```

The function `spplot()` allows to plot one or more curves together with one or more sets of isolated points (without joining the dots). This can be useful, for example, to compare how an estimated continuous variable compares with experimental data points. The prototype is as follows:

```
void spplot(DMatrix& x1, DMatrix& y1,
            DMatrix& x2, DMatrix& y2,
            const string& title,
            char* xlabel,
            char* ylabel,
            char* legend=NULL,
            char* terminal=NULL,
            char* output=NULL);
```

where `x1` is a column or row vector with n_1 elements and `y1` is a matrix with one either row or column dimension equal to n_1. The pair (`x1`, `y1`) is used to generate curve(s). `x2` is a column or row vector with n_2 elements and `y2` is a matrix with one either row or column dimension equal to n_2. The pair (`x2`, `y2`) is used to plot data points.

For example, if the user wishes to display a curve on the basis of the pair (`t` , `y1`) and on the same plot compare with experimental points stored in the pair (`te` , `ye`), then an example of the syntax to call the `spplot()`function is:

```
plot(t,y,te, ye, "Data fit for y","time (s)", "u", "y ye");
```

67

The `multiplot()` function allows the user to plot on a single window an array of sub-plots, having one curve per subplot. The function prototype is as follows:

```
void multiplot(DMatrix& x,
               DMatrix& y,
               const string& title,
               char* xlabel,
               char* ylabel,
               char* legend,
               int nrows=0,
               int ncols=0,
               char* terminal=NULL,
               char* output=NULL ) ;
```

where x is a column or row vector with n elements and y is a matrix with one either row or column dimension equal to n, `xlabel` should be a string with the common label for the x-axis of all subplots, `ylabel` should be a string with the labels for all y-axes of all subplots, separated by spaces, `nrows` is the number of rows of the array of subplots, `ncols` is the number of columns of the array of subplots. If `nrows` and `ncols` are not provided, then the array of subplots has a single column. Note that the product `nrows*ncols` should be equal to n, which is the number of curves to be plotted.

For example, if the user wishes to display an array of subplots of the state trajectories of a system with four state variables which have been stored in DMatrix object "y", and assuming that the corresponding time vector has been stored in DMatrix object "t", then an example of the syntax to call the `multiplot()` function is:

```
multiplot(t,y,"State variables","time (s)",
              "y1 y2 y3 y4", "y1 y2 y3 y4");
```

In the above case, a 4×1 array of sub-plots is produced. If a 2×2 array of sub-plots is required, then the following command can be used:

```
multiplot(t,y,"State variables","time (s)",
          "y1 y2 y3 y4", "y1 y2 y3 y4", 2, 2);
```

The function `surf()` plots the colored parametric surface defined by three matrix arguments. The prototype of the `surf()` function is as follows:

```
void surf(DMatrix& x,
```

```
               DMatrix& y,
               DMatrix& z,
               const string& title,
               char* xlabel,
               char* ylabel,
               char* zlabel,
               char* terminal=NULL,
               char* output=NULL,
               char* view=NULL);
```

Here `view` is a character string with two constants `<rot_x>,<rot_y>` (e.g. "50,60"), where `rot_x` is an angle in the interval $[0, 180]$ degrees, and `rot_y` is an angle in the interval $[0, 360]$ degrees. This is used to set the viewing angle of the surface plot.

For example, if the user wishes to display a surface plot of a $N \times M$ matrix Z with respect to the $1 \times N$ vector X and the $1 \times M$ vector Y, stored, respectively, in DMatrix objects "z", "x" and "y", then an example of the syntax to call the `surf()` function is:

```
surf(x, y,  z, "Title", "x-label", "y-label", "z-label");
```

The function `plot3()` plots a 3D parametric curve defined by three vector arguments. The prototype of the `plot3()` function is as follows:

```
void plot3(DMatrix& x,
           DMatrix& y,
           DMatrix& z,
           const string& title,
           char* xlabel,
           char* ylabel,
           char* zlabel,
           char* terminal=NULL,
           char* output=NULL
           char* view = NULL);
```

Here `view` is a character string with two constants `<rot_x>,<rot_y>` (e.g. "50,60"), where `rot_x` is an angle in the interval $[0, 180]$ degrees, and `rot_y` is an angle in the interval $[0, 360]$ degrees. This is used to set the viewing angle of the 3D plot.

For example, if the user wishes to display a 3D parametric curve of a $1 \times N$ vector Z with respect to the $1 \times N$ vector X and the $1 \times M$ vector Y, stored, respectively, in DMatrix objects "z", "x" and "y", then an example of the syntax to call the `plot3()`function is:

```
plot3(x, y,  z, "Title", "x-label", "y-label", "z-label");
```

The function `polar()` plots a polar curve defined by two vector arguments. The prototype of the `polar()` function is as follows:

```
void polar(DMatrix& theta,
           DMatrix& r,
           const string& title,
           char* legend=NULL,
           char* terminal=NULL,
           char* output=NULL);
```

For example, if the user wishes to display a polar plot using a of a $1 \times N$ vector θ (the angle values in radians), and a $1 \times N$ vector r (the corresponding values of the radius), stored, respectively, in DMatrix objects "theta" and "r", then an example of the syntax to call the `polar()`function is:

```
polar(theta, r,  "Title");
```

The `polar()` function is overloaded so that the user may plot together up to three different polar curves. The additional prototypes are given below. For two polar curves:

```
void polar(DMatrix& theta,
           DMatrix& r,
           DMatrix& theta2,
           DMatrix& r2,
           const string& title,
           char* legend=NULL,
           char* terminal=NULL,
           char* output=NULL);
```

For three polar curves.

```
void polar(DMatrix& theta,
           DMatrix& r,
           DMatrix& theta2,
```

Terminal	Description
postscript eps	Encapsulated postscript
pdf	Adobe portable document format (pdf)
Jpeg	jpg graphical format
Png	png graphical format
latex	LaTeX graphical code

Table 2.2: Some of the available GNUplot output graphical formats

```
DMatrix& r2,
DMatrix& theta3,
DMatrix& r3,
const string& title,
char* legend=NULL,
char* terminal=NULL,
char* output=NULL);
```

Some common GNUplot terminals (graphical formats) are given in Table 2.2. See the GNUplot documentation for further details on the keywords needed to specify different graphical formats.

```
http://www.gnuplot.info/documentation.html
```

2.3   Specifying a parameter estimation problem

To use the parameter estimation facilities implemented in $\mathcal{PSOPT}$ for problems where the observation function is defined, and where there is a set of observed data at given sampling points (see section 1.8). The user needs to specify, for each phase, the number of observed variables and the number of sampling points:

```
problem.phases(iphase).nobserved        = INTEGER;
problem.phases(iphase).nsamples         = INTEGER;
```

where `nobserved` is the number of simultaneous measurements taking place at each sampling node, and `nsamples` is the total number of sampling nodes. The above parameters should be entered before calling the function `psopt_level2_setup(problem, algorithm)`. After this, additional information may be entered:

```
problem.phases(iphase).observation_nodes      = DMatrix;
problem.phases(iphase).observations           = DMatrix;
problem.phases(iphase).residual_weights       = DMatrix;
```

where `observation_nodes` is a $1 \times$ `nsamples` matrix, `observations` is a `nobserved` $\times$ `nsamples` matrix, `residual_weights` is a $1 \times$ `nobserved \times nsamples` matrix. The `residual_weights` matrix is by default full of ones.

   If parameter estimation data for a particular phase is saved in a text file with the column format specified below, then an auxiliary function, which is described below, can be used to load the data:

```
 < Time > < Obs. # 1> < Weight # 1> ... <Obs. # n> < Weight # n>
```

where each column is separated by either tabs or spaces, the first column contains the time stamps of the samples, the second column contains the observations of the first variable, the third column contains the weights for each observation of the first variable, and so on. It is then possible to load observation nodes, observations, and residual weights and assign them to the appropriate fields of the `problem` structure by using the function `load_parameter_estimation_data`, whose prototype is given below.

```
void  load_parameter_estimation_data() (
        Prob& problem,
        int iphase,
        char* filename
);
```

   Note that the user should not register the `problem.end_point_cost` or the `problem.integrand_cost` functions, but the user needs to register `problem.observation_function`. The prototype of this function is as follows:

```
void  observation_function(
                        adouble* observations,
                        adouble* states,
                        adouble* controls,
                        adouble* parameters,
                        adouble& time_k,
                        int k,
                        adouble* xad,
                        int iphase,
                        Workspace* workspace  );
```

where on output the function should return the array of observed variables corresponding to sampling index k at sampling instant `time_k`. The rest of the interface is the same as for general optimal control problems.

2.4 Automatic scaling

If the user specifies the option `algorithm.scaling` as "automatic", then $\mathcal{PSOPT}$ will calculate scaling factors as follows.

1. Scaling factors for controls, states, static parameters, and time, are computed based on the user supplied bounds for these variables. For finite bounds, the variables are scaled such that their original value multiplied by the scaling factor results in a number within the range $[-1, 1]$. If any of the bounds is greater or equal than the constant `inf`, then the variable is scaled to lie within the intervals, $[-\inf, 1]$, $[1, \inf]$ or $[-\inf, \inf]$.The constant `inf` is defined in the include file `psopt.h` as 1×10^{19}.

2. Scaling factors for all constraints (except for the differential defect constraints) are computed as follows. The scaling factor for the i-th constraint is the reciprocal of the norm of the i-th row of the Jacobian of the constraints (Betts, 2001). If the computed norm is zero, then the scaling factor is set to 1.

3. The scaling factors of each differential defect constraint is by default equal to the scaling factor of the corresponding state by default (Betts, 2001). However, if `algorithm.defect_scaling` is set to "jacobian-based", then the scaling factors of the differential defect constraints are computed as is done for the other constraints.

4. The scaling factor for the objective function is the reciprocal of the norm of the gradient of the objective function evaluated at the initial guess. If the norm of the objective function at the initial guess is zero, then the scaling factor of the objective function is set to one.

2.5 Differentiation

Users are encouraged to use, whenever possible, the automatic differentiation facilities provided by the ADOL-C library. The use of automatic derivatives is the default behaviour, but it may be specified explicitly by setting the `derivatives` option to "automatic". $\mathcal{PSOPT}$ uses the ADOL-C drivers for sparsity determination, Jacobian and gradient evaluation. Automatic derivatives are more accurate than numerical derivatives as they are free of truncation errors. Moreover, $\mathcal{PSOPT}$ works faster when using automatic derivatives.

There may be cases, however, where it is preferrable or necessary to use numerical derivatives. If the user specifies the option `algorithm.derivatives` as "numerical", then $\mathcal{PSOPT}$ will calculate the derivatives required by the nonlinear programming algorithm as follows.

1. If IPOPT is being used for optimization, then the Jacobian of the constraints is computed by using sparse finite differences, such that groups of variables are perturbed simultaneously [13]. It is assumed that the Jacobian of the constraint function $G(y)$ is divided into constant and variable terms as follows:

$$\frac{\partial G(y)}{\partial y} = A + \frac{\partial g(y)}{\partial y} \qquad (2.7)$$

where matrices A and $\partial g(y)/\partial y$ do not have non-zero elements with the same indices. The constant part A of the constraint Jacobian is estimated first, and only the variable part of the jacobian $\partial g(y)/\partial y$ is estimated by sparse finite differences.

The gradient of the objective function is computed by perturbing one variable at a time. Normally the central difference formula is used, but if the perturbed variable is at (or very close to) one of its bounds, then the forward or backward difference formulas are employed.

2. If SNOPT is being used for optimization, then the numerical derivatives are computed by the SNOPT routines for that purpose. See the SNOPT documentation for further details.

2.6 Generation of initial guesses

If no guesses are supplied by the user, then $\mathcal{PSOPT}$ computes the initial guess for the unspecified decision variables as follows. Each variable is assumed to be constant and equal to the mean value of its bounds, provided none of the bounds is defined as `inf` or `-inf`. If only one of the bounds is `inf` or `-inf`, then the variable is initialized with the value of the other bound. If the upper and lower bounds are `inf` and `-inf`, respectively, then the variable is initialized at zero.

The variables that are initialized automatically for each phase include: the control variables, the state variables, the static parameters, the initial time, and the final time.

The user may also compute initial guesses for the state variables by propagating the differential equations associated with the problem. Two auxiliary functions are provided for this purpose. See section 2.10 for more details.

2.7 Evaluating the discretization error

$\mathcal{PSOPT}$ evaluates the discretization error using a method adopted from [3]. Define the error in the differential equation as a function of time:

$$\epsilon(t) = \dot{\tilde{x}}(t) - f[\tilde{x}(t), \tilde{u}(t), p, t]$$

where $\tilde{x}$ is an interpolated value of the state vector given the grid point values of the state vector, $\tilde{\dot{x}}$ is an estimate of the derivative of the state vector given the state vector interpolant, and $\tilde{u}$ is an interpolated value of the control vector given the grid points values of the control vector. The type of interpolation used depends on the collocation method employed. For Legendre and Chebyshev methods, the interpolation done by the Lagrange interpolant. For Trapezoidal and Hermite-Simpson methods and central difference methods, cubic spline interpolation is used. The absolute local error corresponding to state i on a particular interval $t \in [t_k, t_{k+1}]$, is defined as follows:

$$\eta_{i,k} = \int_{t_k}^{t_{k+1}} |\epsilon_i(t)| \mathrm{d}t$$

where the integral is computed using the composite Simpson method. The default number of integration steps for each interval is 10, but this can be changed by means of the input parameter `algorithm.nsteps_error_integration`. The relative local error is defined as:

$$\epsilon_k = \max_i \frac{\eta_{i,k}}{w_i + 1}$$

where

$$w_i = \max_{k=1}^{N} \left[|\tilde{x}_{i,k}|, |\tilde{\dot{x}}_{i,k}| \right]$$

After each $\mathcal{PSOPT}$ run, the sequence ϵ_k for each phase is available through the solution structure as follows:

```
epsilon = solution.get_relative_local_error_in_phase(iphase)
```

where `epsilon` is a DMatrix object. The error sequence can be analysed by the user to assess the quality of the discretization. This information may be useful to aid the mesh refinement process.

Additionally, the maximum value of the sequence ϵ_k for each phase is printed in the solution summary at the end of an execution.

2.8   Mesh refinement

2.8.1   Manual mesh refinement

Manual mesh refinement, which is the default option, is performed by interpolating a previous solution based on n_1 nodes, into a new mesh based on n_2 nodes, where $n_2 > n_1$, and using the interpolated solution as an initial guess for a new optimization. If global collocation is being used, $\mathcal{PSOPT}$ employs Lagrange polynomials to perform the interpolation associated with mesh refinement. If local collocation is being used, $\mathcal{PSOPT}$ employs cubic splines to perform the interpolation. The variables which are interpolated include the controls, states and Lagrange multipliers associated with the differential

defect constraints, which are related to the co-states. The other decision variables (start and final times, and static parameters) do not need to be interpolated.

To perform mesh refinement, the user must supply the desired sequence of grid points (or nodes) for each phase through a string constant with the values separated by commas which is assigned to

```
problem.phases(iphase).nodes
```

The number of mesh refinements to be carried out is one less than the amount of nodes to be tried. Note that the number of nodes to be tried must be the same in all phases (but not necessarily their value).

For example to try the sequence 10, 20 and 50 nodes in phase `iphase`, then the following command specifies that:

```
problem.phases(iphase).nodes = "[10, 20, 50]";
```

If the user wishes to try only a single grid size (with no mesh refinement), this is specified by providing a single value as follows:

```
problem.phases(iphase).nodes = INTEGER;
```

In problems with more than one phase, the length of the node sequence to be tried needs to be the same in each phase, but the actual grid sizes need not be the same betweeen phases.

2.8.2 Automatic mesh refinement with pseudospectral grids

If a global collocation method is being used and `algorithm.mesh_refinement` is set to `"automatic"`, then, mesh refinement is carried out as described below. $\mathcal{PSOPT}$ will compute the maximum discretization error $\epsilon^{(i,m)}$ for every phase i at every mesh refinement iteration m, as described in Section 2.7.

The method is based on a nonlinear least squares fit of the maximum discretization error for each phase with respect to the mesh size:

$$\hat{y}^i = \varphi_1 \theta_1 + \theta_2$$

where $\hat{y}^i$ is an estimate of $\log(\epsilon^{(i)})$, $\varphi_1 = \log(N)$, θ_1 and θ_2 are parameters which are estimated based on the mesh refinement history. This is equivalent to modelling the dependency of $\epsilon^{(i)}$ with respect to the number of nodes N_i as follows:

$$\epsilon^{(i)} = C \frac{1}{N_i^m}$$

where $m = -\theta_1$, $C = \exp(\theta_2)$. This dependency relates to the upper bound on the $\mathcal{L}_2$ norm of the interpolation error given in [10]. Given a desired tolerance $\epsilon_{\max}$, this approximation is applied when the discretization error has been reduced for at least two iterations to find an extrapolated number of nodes which reduces the discretization error by a factor of 0.25.

The user specifies an initial number of nodes for each phase, as follows:

```
problem.phases(iphase).nodes = INTEGER.
```

The user may also specify values for the following parameters which control the mesh refinement procedure. The default values are those shown:

Maximum discretization error, $\epsilon_{\max}$:

```
algorithm.ode_tolerance              = 1.e-3;
```

Maximum increment factor, F:

```
algorithm.mr_max_increment_factor    = 0.4;
```

Maximum number of mesh refinement interations, $m_{\max}$:

```
algorithm.mr_max_iterations = 7;
```

Minimum number of extrapolation points:

```
algorithm.mr_min_extrapolation_points = 3;
```

Initial increment for the number of nodes, ΔN_0:

```
algorithm.mr_initial_increment       = 10;
```

The mesh refinement algorithm is decribed below.

1. Set the iteration index $m = 1$.

2. If $m > m_{\max}$, terminate.

3. Solve the nonlinear programming problem for the current mesh, and find the maximum discretization error $\epsilon^{(i,m)}$ for each phase i.

4. If $\epsilon^{(i,m)} < \epsilon_{\max}$ for all phases, terminate.

5. The increment in the number of nodes in each phase i, denoted by ΔN_i, is computed as follows:

    (a) If $m < m_{\min}$ then $\Delta N_i = \Delta N_0$
    (b) if $\epsilon^{(i,m)}$ has increased in the last two iterations, then $\Delta N_i = 5$

(c) if $\epsilon^{(i,m)}$ has decreased in at least the last two iterations, compute the parameters θ_1 and θ_2 by solving a least squares problem based on the monotonic part of the mesh refinement history, then ΔN_i is computed as follows:

$$\Delta N_i = \max \left(\text{int} \left[\exp \left(\frac{y_d - \theta_2}{\theta_1} \right) \right] - N_i, \Delta N_{\max} \right)$$

where $y_d = \max(\log(0.25\epsilon^{(i,m)}), \log(0.99\epsilon_{\max}))$, and

$$\Delta N_{\max} = FN_i$$

where F is the maximum increment factor.

6. Increment the number of nodes in the mesh for each phase:

$$N_i \leftarrow N_i + \Delta N_i$$

7. Set $m \leftarrow m + 1$, and go back to step 2.

2.8.3 Automatic mesh refinement with local collocation

If a local collocation method (trapezoidal, Hermite-Simpson) is being used, and `algorithm.mesh_refinement` is set to `"automatic"`, then, mesh refinement is carried out as described below. $\mathcal{PSOPT}$ will compute the discretization error $\epsilon^{(i,m)}$ for every phase i at every mesh refinement iteration m, as described in Section 2.7. The method is based on the mesh refinement algorithm described by Betts [3]. If the current discretization method is trapezoidal, then the order $p = 2$, otherwise if the current method is Hermite-Simpson, then $p = 4$. Conversely, if p changes from 2 to 4, then the discretization method is changed from trapezoidal to Hermite-Simpson.

The user specifies an initial number of nodes for each phase, as follows:

```
problem.phases(iphase).nodes = INTEGER.
```

The user may also specify values for the following parameters which control the mesh refinement procedure. The default values are those shown:

Maximum discretization error, $\epsilon_{\max}$:

```
algorithm.ode_tolerance = 1.e-3;
```

Minimum increment factor, κ:

```
algorithm.mr_kappa = 0.1;
```

Maximum increment factor, ρ:

```
algorithm.mr_max_increment_factor    = 0.4;
```

78

Maximum number of mesh refinement interations, $m_{\max}$:

```
algorithm.mr_max_iterations = 7;
```

Maximum nodes to add within a single interval, M_1:

```
algorithm.mr_M1 = 5;
```

Define $M' = \min(M_1, \kappa N_i) + 1$, where N_i is the current number of nodes in phase i. The local mesh refinement algorithm is as follows.

1. Set the iteration index $m = 1$.

2. If $m > m_{\max}$, terminate.

3. Solve the nonlinear programming problem for the current mesh, and find the discretization error $\epsilon_k^{(i,m)}$ for each interval k and each phase i.

4. If $\max_k \epsilon_k^{(i,m)} < \epsilon_{\max}$ for all phases, terminate the mesh refinement iterations.

5. Select the primary order for the new mesh:

    (a) If $p < 4$ and $\epsilon_\alpha \le 2\bar{\epsilon}^{(i,m)}$, where $\bar{\epsilon}^{(i,m)}$ is the average discretization error in phase i.

    (b) Otherwise, if $p < 4$ and $i > 2$, then set $p = 4$.

6. Estimate the order reduction. The current and previous grid are compared and the order reduction r_k is computed for each interval in each phase. The order reduction is computed from:

$$r_k = \max[0, \min(\mathrm{nint}(\hat{r}_k), p)]$$

    where

$$\hat{r}_k = p + 1 - \frac{\theta_k/\eta_k}{1 + I_k}$$

    where $\mathrm{nint}()$ is the nearest integer function, I_k is the number of points being added to interval k, η_k is the estimated discretization error within interval k of the old grid, after the subdivision, and θ_k is the discretization error on the old grid before the subdivision.

7. Construct the new mesh.

    (a) Compute the interval α with maximum error within phase i:

$$\epsilon_\alpha^{(i)} = \max_k \epsilon_k^{(i,m)}$$

    (b) Terminate step 7 if

i. M' nodes have been added, and

ii. the error is below the tolerance in each phase: $\epsilon_\alpha^{(i)} < \epsilon_{\max}$ and $I_\alpha = 0$, or

iii. the predicted error is well below the tolerance $\epsilon_\alpha^{(i)} < \kappa \epsilon_{\max}$ and $0 \le I_\alpha < M_1$, or

iv. $\rho(N_i - 1)$ nodes have been added, or

v. M_1 nodes have been added to a single interval.

(c) Add one node to interval α, so that $I_\alpha \leftarrow I_\alpha + 1$.

(d) Update the predicted error for interval α using

$$\epsilon_\alpha \leftarrow \epsilon_\alpha \left(\frac{1}{1 + I_k} \right)^{p - r_k + 1}$$

(e) Return to step 7(a).

8. Set $m \leftarrow m + 1$ and go back to step 2.

2.8.4 LaTeX code generation

LaTeX code is generated automatically producing a table with a summary of information about the mesh refinement process. It may be useful to include this summary in publications that incorporate results generated with $\mathcal{PSOPT}$. A file named `mesh_statistics_$$$.tex` is automatically created, unless `algorithm.print_level` is set to zero, where `$$$` represents the characters of `problem.outfilename` which occur to the left of the file extension point ".".

To include the generated table in a LaTeX document, simply use the command:

`\input{mesh_statistics_$$$.tex}`

The generated table includes a caption associated with the problem name as set through `problem.name`, as well as a label which is generated by concatenating the string `"mesh_stats_"` with the characters of `problem.outfilename` which occur to the left of the file extension point ".". The caption and label can easily be changed to suit the user requirements by editing or renaming the generated file. A key to the abbreviations used in the file is also printed. The abbreviations for the discretization methods used are described in Table 2.3

2.9 Implementing multi-segment problems

Sometimes, it is useful for computational or other reasons to define a multi-segment problem. A multi-segment problem is an optimal control problem with multiple sequential phases that has the same dynamics and path

Abbreviation	Description
LGL-ST	LGL nodes with standard differentiation matrix given by equation (1.12)
LGL-RR	LGL nodes with reduced round-off differentiation matrix given by equation (1.24)
LGL-CD	LGL nodes with reduced central-differences differentiation matrix given by equation (1.24)
CGL-ST	CLG nodes with standard differentiation matrix given by equation (1.19)
CGL-RR	LGL nodes with reduced round-off differentiation matrix given by equation (1.24)
CGL-CD	LGL nodes with reduced central-differences differentiation matrix given by equation (1.24)
TRP	Trapezoidal discretization, see equation (1.50)
H-S	Hermite-Simpson discretization, see equation (1.51)

Table 2.3: Description of the abbreviations used for the discretization methods which are shown in the automatically generated LATEX table

constraints in each phase. The multi-segment facilities implemented in $\mathcal{PSOPT}$ allow the user to specify multi-segment problems in an easier way than defining a multi-phase problem. Special functions are called automatically to patch consecutive segments and ensure state and time continuity across the segment boundaries.

To specify a multi-segment problem the it is necessary to create a data structure of the type `MSdata` (in addition to the `problem`, `algorithm` and `solution` structures) and assign values to its elements as follows:

```
MSdata msdata;

msdata.nsegments           = INTEGER;
msdata.nstates             = INTEGER;
msdata.ncontrols           = INTEGER;
msdata.nparameters         = INTEGER;
msdata.npath               = INTEGER;
msdata.n_initial_events    = INTEGER;
msdata.n_final_events      = INTEGER;
msdata.nodes               = INTEGER OR STRING;
msdata.continuous_controls = BOOLEAN
```

If it is desired to enforce control continuity across the segment boundaries, then set `msdata.continuous_controls` to `true`. By default the controls are allowed to be discontinuous across the segment boundaries.

81

The number of nodes per segment can be speficied as follows (note that it is possible to create grids with segments that have different number of nodes):

- as a single value (e.g. 30), such that the same number of nodes is employed in each segment.

- If `algorithm.mesh_refinement` is set to `"manual"`, a character string can be entered with the node sequence to be tried per segment as part of a manual mesh refinement strategy (e.g. "[30, 50, 60]"). Here the number of values corresponds to the number of mesh refinement iterations to be performed. It is assumed that the same node sequence is tried for each segment. If `algorithm.mesh_refinement` is set to `"automatic"`, then only the first value of the specified sequence is used to start the automatic mesh refinement iterations.

- If `algorithm.mesh_refinement` is set to `"manual"`, a matrix can be entered, such that each row of the matrix corresponds to the node sequence to be tried in the corresponding segment (a character string can be used to enter the matrix, e.g. "[30, 50, 60; 10, 15, 20; 5, 10, 15]", noting the semicolons that separate the rows). Here the number of rows corresponds to the number of segments, and the number of columns corresponds to the number of manual mesh refinement iterations to be performed. If `algorithm.mesh_refinement` is set to `"automatic"`, then only the first value of the specified sequence for each segment is used to start the automatic mesh refinement iterations.

After this, the following function should be called:

```
multi_segment_setup(problem, algorithm, msdata);
```

The upper and lower bounds on the relevant event times of the problem (start time for each segment, and end time for the last segment) can be entered as follows:

```
problem.bounds.lower.times = DMATRIX OR STRING;
problem.bounds.upper.times = DMATRIX OR STRING;
```

where entered value specifies the time bounds in the following order:

$$[t_0^{(1)}, t_0^{(2)}, \ldots, t_0^{(N_p)}, t_f^{(N_p)}]$$

At this point, the bound information for segment 1 (phase 1) can be entered (bounds for states, controls, event constraints, path constraints, and parameters), as described in section 2.2.6. This should be followed by the bound information for the event constraints of the last phase or segment. After entering the bound information, the auxiliary function `auto_phase_bounds` should be called as follows:

```
    auto_phase_bounds(problem);
```

The initial guess for the solution can be specified by a call to the function `auto_phase_guess`. See section 2.10.

See section 3.29 for an example on the use of the multi-segment facilities available within $\mathcal{PSOPT}$.

2.10 Other auxiliary functions available to the user

$\mathcal{PSOPT}$ implements a number of auxiliary functions to help the user define optimal control problems. Most (but not all) of these functions are suitable for use with automatic differentiation. All the functions can also be used with numerical differentiation. See the examples section for further details on the use of these functions.

2.10.1   cross function

This function takes two arrays of adoubles x and y, each of dimension 3, and returns in array z (also of dimension 3) the result of the vector cross product of x and y. The prototype of the function is as follows:

```
void cross(adouble* x, adouble* y, adouble* z);
```

2.10.2   dot function

This function takes two arrays of adoubles x and y, each of dimension n, and returns the dot product of x and y. The prototype of the function is as follows:

```
adouble dot(adouble* x, adouble* y, int n);
```

2.10.3   get_delayed_state function

This function allows the user to implement DAE's with delayed states. Use only in single-phase problems. Its prototype is as follows:

```
void get_delayed_state(adouble* delayed_state,
                       int state_index,
                       int iphase,
                       adouble& time,
                       double delay,
                       adouble* xad);
```

The function parameters are as follows:

- **delayed_state**: on output, the variable pointed by this pointer contrains the value of the delayed state.

- **state_index**: is the index of the state vector whose delayed value is to be found (starting from 1).

- **iphase**: is the phase index (starting from 1).

- **time**: is the value of the current instant of time within the phase.

- **delay**: is the value of the delay.

- **xad**: is the vector of scaled decision variables.

2.10.4   get_delayed_control function

This function allows the user to implement DAE's with delayed controls. Use only in single-phase problems. Its prototype is as follows:

```
void get_delayed_control(adouble* delayed_control,
                         int control_index,
                         int iphase,
                         adouble& time,
                         double delay,
                         adouble* xad);
```

- **delayed_control**: on output, the variable pointed by this pointer contrains the value of the delayed state.

- **control_index**: is the index of the control vector whose delayed value is to be found (starting from 1).

- **iphase**: is the phase index (starting from 1).

- **time**: is the value of the current instant of time within the phase.

- **delay**: is the value of the delay.

- **xad**: is the vector of scaled decision variables.

2.10.5   get_interpolated_state function

This function allows the user to obtain interpolated values of the state at arbitrary values of time within a phase. Its prototype is as follows:

```
void get_interpolated_state(adouble* interp_state,
                            int state_index,
                            int iphase,
                            adouble& time,
                            adouble* xad);
```

- `interp_state`: on output, the variable pointed by this pointer contrains the value of the interpolated state.

- `state_index`: is the index of the state vector whose interpolated value is to be found (starting from 1).

- `iphase`: is the phase index (starting from 1).

- `time`: is the value of the current instant of time within the phase.

- `xad`: is the vector of scaled decision variables.

2.10.6   `get_interpolated_control` function

This function allows the user to obtain interpolated values of the control at arbitrary values of time within a phase. Its prototype is as follows:

```
void get_interpolated_control(adouble* interp_control,
                              int control_index,
                              int iphase,
                              adouble& time,
                              adouble* xad);
```

- `interp_control`: on output, the variable pointed by this pointer contains the value of the interpolated control.

- `control_index`: is the index of the control vector whose interpolated value is to be found (starting from 1).

- `iphase`: is the phase index (starting from 1).

- `time`: is the value of the current instant of time within the phase.

- `xad`: is the vector of scaled decision variables.

2.10.7   `get_control_derivative` function

This function allows the user to obtain the value of the derivative of a specified control variable at arbitrary values of time within a phase. Its prototype is as follows:

```
void get_control_derivative(adouble* control_derivative,
                            int control_index,
                            int iphase,
                            adouble& time,
                            adouble* xad)
```

- `control_derivative`: on output, the variable pointed by this pointer contains the value of the control derivative.

- `control_index`: is the index of the control vector whose interpolated value is to be found (starting from 1).

- `iphase`: is the phase index (starting from 1).

- `time`: is the value of the current instant of time within the phase.

- `xad`: is the vector of scaled decision variables.

2.10.8 `get_state_derivative` function

This function allows the user to obtain the value of the derivative of a specified state variable at arbitrary values of time within a phase. Its prototype is as follows:

```
void get_state_derivative(adouble* control_derivative,
                          int state_index,
                          int iphase,
                          adouble& time,
                          adouble* xad)
```

- `state_derivative`: on output, the variable pointed by this pointer contains the value of the state derivative.

- `state_index`: is the index of the state vector whose interpolated value is to be found (starting from 1).

- `iphase`: is the phase index (starting from 1).

- `time`: is the value of the current instant of time within the phase.

- `xad`: is the vector of scaled decision variables.

2.10.9 `get_initial_states` function

This function allows the user to obtain the values of the states at the initial time of a phase. Its prototype is as follows:

```
void get_initial_states(adouble* states, adouble* xad, int iphase);
```

- `states`: on output, this array contains the values of the initial states within the specified phase.

- `iphase`: is the phase index (starting from 1).

- `xad`: is the vector of scaled decision variables.

2.10.10   get_final_states **function**

This function allows the user to obtain the values of the states at the final time of a given phase. Its prototype is as follows:

```
void get_initial_states(adouble* states, adouble* xad, int iphase);
```

- states: on output, this array contains the values of the final states within the specified phase.

- iphase: is the phase index (starting from 1).

- xad: is the vector of scaled decision variables.

2.10.11   get_initial_controls **function**

This function allows the user to obtain the values of the controls at the initial time of a phase. Its prototype is as follows:

```
void get_initial_controls(adouble* controls, adouble* xad, int iphase);
```

- controls: on output, this array contains the values of the initial controls within the specified phase.

- iphase: is the phase index (starting from 1).

- xad: is the vector of scaled decision variables.

2.10.12   get_final_controls **function**

This function allows the user to obtain the values of the controls at the final time of a given phase. Its prototype is as follows:

```
void get_initial_controls(adouble* controls, adouble* xad, int iphase);
```

- controls: on output, this array contains the values of the final controls within the specified phase.

- iphase: is the phase index (starting from 1).

- xad: is the vector of scaled decision variables.

2.10.13 `get_initial_time` function

This function allows the user to obtain the value of the initial time of a given phase. Its prototype is as follows:

```
adouble get_initial_time(adouble* xad, int iphase);
```

- `iphase`: is the phase index (starting from 1).

- `xad`: is the vector of scaled decision variables.

- The function returns the value of the initial time within the specified phase as an `adouble` type.

2.10.14 `get_final_time` function

This function allows the user to obtain the value of the final time of a given phase. Its prototype is as follows:

```
adouble get_final_time(adouble* xad, int iphase);
```

- `iphase`: is the phase index (starting from 1).

- `xad`: is the vector of scaled decision variables.

- The function returns the value of the final time within a phase as an `adouble` type.

2.10.15 `auto_link` function

This function allows the user to automatically link two phases by generating suitable state and time continuity constraints. It is assumed that the number of states in the two phases being linked is the same. The function is intended to be called from within the user supplied `linkages` function. Each call to `auto_link` generates an additional number of linkage constraints given by the number of states being linked plus one.

The function prototype is as follows:

```
void auto_link(adouble* linkages,
               int* index,
               adouble* xad,
               int iphase_a,
               int iphase_b,
               Workspace* workspace);
```

- `linkages`: on output, this is the updated array of linkage constraint values.

- **index**: on input, the variable pointed to by this pointer contains the next value of the linkages array to be updated. On output, this value is updated to be used in the next call to the `auto_link` function. The first time the function is called, the value should be 0.

- **xad**: is the vector of scaled decision variables.

- **iphase_a**: is the phase index (starting from 1) of one phase to be linked.

- **iphase_b**: is the phase index (starting from 1) of the other phase to be linked.

2.10.16   `auto_link_2` function

This function works in a simular way as the auto link function, but it also forces the control variables to be continuous at the boundaries. It requires a match in the number of states and in the number of controls between the phases being linked. Each call to `auto_link_2` generates an additional number of linkage constraints given by the number of states plus the number of controls, plus one.

The function prototype is as follows:

```
void auto_link_2(adouble* linkages,
                 int* index,
                 adouble* xad,
                 int iphase_a,
                 int iphase_b,
                 Workspace* workspace);
```

- **linkages**: on output, this is the updated array of linkage constraint values.

- **index**: on input, the variable pointed to by this pointer contains the next value of the linkages array to be updated. On output, this value is updated to be used in the next call to the `auto_link_2` function. The first time the function is called, the value should be 0.

- **xad**: is the vector of scaled decision variables.

- **iphase_a**: is the phase index (starting from 1) of one phase to be linked.

- **iphase_b**: is the phase index (starting from 1) of the other phase to be linked.

2.10.17   `auto_phase_guess` **function**

This function allows the user to automatically specify the initial guess in a multi-segment problem. The function prototype is as follows:

```
void auto_phase_guess(Prob& problem,
                      DMatrix& controls,
                      DMatrix& states,
                      DMatrix& param,
                      DMatrix& time);
```

so that the controls, states, time and static parameters are specified as if the problem was single-phase.

2.10.18   `linear_interpolation` **function**

This function interpolates a point defined function using classical linear interpolation. The function is not suitable for automatic differentiation, so it should only be used with numerical differentiation. This is useful when the problem involves tabular data. The function prototype is as follows:

```
void linear_interpolation(adouble* y,
                          adouble& x,
                          DMatrix& pointx,
                          DMatrix& pointy,
                          int npoints);
```

- `y`: on output, the variable pointed to by this pointer contains the interpolated function value.

- `x`: is the value of the independent variable for which the interpolated function value is sought.

- `pointx`: is the DMatrix object of independent data points.

- `pointy`: is a DMatrix object of dependent data points.

- `npoints`: is the number of points in the data objects `pointx` and `pointy`.

2.10.19   `smoothed_linear_interpolation` **function**

This function interpolates a point defined function using a smoothed linear interpolation. The method used avoids joining sharp corners between adjacent linear segments. Instead, smooothed pulse functions are used to join the segments. The function is suitable for automatic differentiation. This is useful when the problem involves tabular data. The function prototype is as follows:

```
void smoothed_linear_interpolation(adouble* y,
                                   adouble& x,
                                   DMatrix& pointx,
                                   DMatrix& pointy,
                                   int npoints);
```

- y: on output, the variable pointed to by this pointer contains the interpolated function value.

- x: is the value of the independent variable for which the interpolated function value is sought.

- pointx: is the DMatrix object of independent data points.

- pointy: is a DMatrix object of dependent data points.

- npoints: is the number of points in the data objects pointx and pointy.

2.10.20   spline_interpolation **function**

This function interpolates a point defined function using cubic spline interpolation. The function is not suitable for automatic differentiation, so it should only be used with numerical differentiation. This is useful when the problem involves tabular data. The function prototype is as follows:

```
void spline_interpolation(     adouble* y,
                               adouble& x,
                               DMatrix& pointx,
                               DMatrix& pointy,
                               int npoints);
```

- y: on output, the variable pointed to by this pointer contains the interpolated function value.

- x: is the value of the independent variable for which the interpolated function value is sought.

- pointx: is the DMatrix object of independent data points.

- pointy: is a DMatrix object of dependent data points.

- npoints: is the number of points in the data objects pointx and pointy.

2.10.21 `bilinear_interpolation` function

The function interpolates functions of two variables on a regular grid using the classical bilinear interpolation method. This is useful when the problem involves tabular data. The function prototype is as follows.

```
void bilinear_interpolation(adouble* z,
                            adouble& x,
                            adouble& y,
                            DMatrix& X,
                            DMatrix& Y,
                            DMatrix& Z)
```

- `z`: on output the `adouble` variable pointed to by this pointer contains the interpolated function value.

- The `adouble` pair of variables (`x`, `y`) represents the point at which the interpolated value of the function is returned.

- `X`: is a vector (DMatrix object) of dimension `nxpoints` $\times$ 1.

- `Y`: is a vector (DMatrix object) of dimension `nypoints` $\times$ 1.

- `Z`: is a matrix (DMatrix object) of dimensions `nxpoints` $\times$ `nypoints`. Each element `Z(i,j)` corresponds to the pair (`X(i)`, `Y(j)`)

The function does not deal with sparse data. This function does not allow the use of automatic differentiation, so it should only be used with numerical differentiation.

2.10.22 `smooth_bilinear_interpolation` function

The function interpolates functions of two variables on a regular grid using the a smoothed bilinear interpolation method which allows the use of automatic differentiation. This is useful when the problem involves tabular data. The function prototype is as follows.

```
void smooth_bilinear_interpolation(adouble* z,
                                   adouble& x,
                                   adouble& y,
                                   DMatrix& X,
                                   DMatrix& Y,
                                   DMatrix& Z)
```

- `z`: on output the `adouble` variable pointed to by this pointer contains the interpolated function value.

- The `adouble` pair of variables (x, y) represents the point at which the interpolated value of the function is returned.

- X: is a vector (DMatrix object) of dimension `nxpoints` $\times$ 1.

- Y: is a vector (DMatrix object) of dimension `nypoints` $\times$ 1.

- Z: is a matrix (DMatrix object) of dimensions `nxpoints` $\times$ `nypoints`. Each element `Z(i,j)` corresponds to the pair (`X(i)`, `Y(j)`)

The function does not deal with sparse data.

2.10.23 `spline_2d_interpolation` function

The function interpolates functions of two variables on a regular grid using the a cubic spline interpolation method. This is useful when the problem involves tabular data. The function prototype is as follows.

```
void spline_2d_interpolation(adouble* z,
                             adouble& x,
                             adouble& y,
                             DMatrix& X,
                             DMatrix& Y,
                             DMatrix& Z,
                             Workspace* workspace)
```

- z: on output the `adouble` variable pointed to by this pointer contains the interpolated function value.

- The `adouble` pair of variables (x, y) represents the point at which the interpolated value of the function is returned.

- X: is a vector (DMatrix object) of dimension `nxpoints` $\times$ 1.

- Y: is a vector (DMatrix object) of dimension `nypoints` $\times$ 1.

- Z: is a matrix (DMatrix object) of dimensions `nxpoints` $\times$ `nypoints`. Each element `Z(i,j)` corresponds to the pair (`X(i)`, `Y(j)`)

The function does not deal with sparse data. This function does not allow the use of automatic differentiation, so it should only be used with numerical differentiation.

2.10.24 `smooth_heaviside` function

This function implements a smooth version of the Heaviside function $H(x)$, defined as $H(x) = 1, x > 0$, $H(x) = 0$ otherwise. The approximation is implemented as follows:

$$H(x) \approx 0.5(1 + \tanh(x/a)) \tag{2.8}$$

where $a > 0$ is a small real number. The function prototype is as follows:

```
adouble smooth_heaviside(adouble x, double a);
```

2.10.25 `smooth_sign` function

This function implements a smooth version of the function $\text{sign}(x)$, defined as $\text{sign}(x) = 1, x > 0$, $\text{sign}(x) = -1, x < 0$, and $\text{sign}(0) = 0$. The approximation is implemented as follows:

$$\text{sign}(x) \approx \tanh(x/a) \tag{2.9}$$

where $a > 0$ is a small real number. The function prototype is as follows:

```
adouble smooth_sign(adouble x, double a);
```

See the examples section for further details on usage of this function.

2.10.26 `smooth_fabs` function

This function implements a smooth version of the absolute value function $|x|$. The approximation is implemented as follows:

$$|x| \approx \sqrt{x^2 + a^2} \tag{2.10}$$

where $a > 0$ is a small real number. The function prototype is as follows:

```
adouble smooth_fabs(adouble x, double a);
```

2.10.27 `integrate` function

The `integrate` function computes the numerical quadrature Q of a scalar function g over the a single phase as a function of states, controls, static parameters and time.

$$Q = \int_{t_0^{(i)}}^{t_f^{(i)}} g[x^{(i)}(t), u^{(i)}(t), p^{(i)}, t] dt \tag{2.11}$$

The integration is done using the Gauss-Lobatto method. This is useful, for example, to incorporate constraints involving integrals over a phase, which can be included as additional event constraints:

$$Q_l \leq \int_{t_0^{(i)}}^{t_f^{(i)}} g[x^{(i)}(t), u^{(i)}(t), p^{(i)}, t]dt \leq Q_u \qquad (2.12)$$

Function `integrate` has the following prototype:

```
adouble integrate(
  adouble (*integrand)(adouble*,
                       adouble*,
                       adouble*,
                       adouble&,
                       adouble*,int
                       Workspace* workspace),
  adouble* xad,
  int iphase,
  Workspace* workspace )
```

- `integrand`: this is a pointer to the function to be integrated.

- `xad`: is the vector of scaled decision variables.

- `iphase`: is the phase index (starting from 1).

- The function returns the value of the integral as an `adouble` type.

The user needs to implement separately the `integrand` function, which must have the prototype:

```
adouble integrand( adouble* states,
                   adouble* controls,
                   adouble* parameters,
                   adouble& time,
                   adouble* xad,
                   int iphase,
                   Workspace* workspace)
```

- `states`: this is an array of instantaneous states.

- `controls`: is an array of instantaneous controls.

- `parameters`: is an array of static parameter values.

- `time`: is the value of the current instant of time within the phase.

- `xad`: is the vector of scaled decision variables.

- **iphase**: is the phase index (starting from 1).

- the function must return the value of the integrand function given the supplied parameters as an `adouble` type.

2.10.28   `product_ad` functions

There are two versions of this function. The first version has the prototype:

```
void product_ad(const DMatrix& A,
                const adouble* x,
                int nx,
                adouble* y);
```

This function multiplies a constant matrix stored in DMatrix object `A` by `adouble` vector stored in array `x`, which has length `nx`, and returns the result in `adouble` array `y`.

The second version has the prototype:

```
void product_ad(adouble* Apr,
                adouble* Bpr,
                int na,
                int ma,
                int nb,
                int mb,
                adouble* ABpr);
```

This function multiplies the (`na` × `nb`) matrix stored column by column in `adouble` array `Apr`, by the (`nb` × `mb`) matrix stored column by column in `adouble` array `Bpr`. The result is stored (column by column) in `adouble` array `ABpr`.

2.10.29   `sum_ad` function

This function adds a matrix or vector stored columnwise in `adouble` array `a`, to a matrix or vector of the same dimensions stored columnwise in adouble array `b`. Both arrays are assumed to have a total of `n` elements. The result is returned in `adouble` array `c`. The function prototype is as follows.

```
void sum_ad(const adouble* a,
            const adouble*b,
            int n,
            adouble* c);
```

2.10.30 `subtract_ad` function

This function subtracts a matrix or vector stored columnwise in `adouble` array `a`, to a matrix or vector of the same dimensions stored columnwise in adouble array `b`. Both arrays are assumed to have a total of `n` elements. The result is returned in `adouble` array `c`. The function prototype is as follows.

```
void subtract_ad(const adouble* a,
          const adouble*b,
          int n,
          adouble* c);
```

2.10.31 `inverse_ad` function

This function computes the inverse of an $n \times n$ square matrix stored columnwise in `adouble` array `a`. The result is returned in `adouble` array `ainv`, also using columnwise storage. The function prototype is as follows.

```
void inverse_ad(adouble* a,
            int n,
            adouble* ainv)
```

2.10.32 `rk4_propagate` function

This function may be used to generate an initial guess for the state trajectory by propagating the dynamics using 4th order Runge-Kutta integration. Note that no bounds are considered on states or controls and that any path constraints specified in function `dae()` are ignored. The user needs to specify a control trajectory and the corresponding time vector. The function prototype is as follows:

```
void rk4_propagate( void (*dae)(),
      DMatrix& control_trajectory,
      DMatrix& time_vector,
      DMatrix& initial_state,
      DMatrix& parameters,
      Prob & problem,
      int iphase,
      DMatrix& state_trajectory,
      Workspace* workspace);
```

- `dae` is a pointer to the problem's `dae` function;

- `control_trajectory` is a DMatrix object of dimensions

  `problem.phases(iphase).ncontrols` $\times$ `M`

  with the initial guess for the controls.

- `time_vector` is a DMatrix object of dimensions $1 \times$ M with the time instants that correspond to each element of `control_trajectory`.

- `initial_state` is a DMatrix object of dimensions

  `problem.phases(iphase).nstates` $\times 1$

  with the value of the initial state vector.

- `parameters` is a DMatrix object of dimensions

  `problem.phases(iphase).nparameters` $\times 1$

  with given values for the static parameters.

- `problem` is a `Prob` structure.

- `iphase` is the phase index.

- `state_trajectory` is a DMatrix object with dimensions

  `problem.phases(iphase).nstates` $\times$ M

  which on output contains the result of the of the propagation. The values of the states correspond to the time vector `time_vector`.

2.10.33   rkf_propagate **function**

This function may be used to generate an initial guess for the state trajectory by propagating the dynamics using the Runge-Kutta-Fehlberg method with variable step size and relative local truncation error within a given tolerance. Note that no bounds are considered on states or controls and that any path constraints specified in function `dae()` are ignored. The user needs to specify a control trajectory and the corresponding time vector, as well as minimum and maximum values for the integration step size, and a tolerance. Note that the function throws an error if the minimum step size is violated. The function prototype is as follows:

```
void rkf_propagate( void (*dae)(),
       DMatrix& control_trajectory,
       DMatrix& time_vector,
       DMatrix& initial_state,
       DMatrix& parameters,
       double tolerance,
       double hmin,
       double hmax,
       Prob & problem,
       int iphase,
       DMatrix& state_trajectory,
       DMatrix& new_time_vector,
```

```
      DMatrix& new_control_trajectory,
      Workspace* workspace);
```

- `dae` is a pointer to the problem's `dae` function;

- `control_trajectory` is a DMatrix object of dimensions

  `problem.phases(iphase).ncontrols` $\times$ `M`

  with the initial guess for the controls.

- `time_vector` is a DMatrix object of dimensions $1 \times$ `M` with the time instants that correspond to each element of `control_trajectory`.

- `initial_state` is a DMatrix object of dimensions

  `problem.phases(iphase).nstates` $\times$ `1`

  with the value of the initial state vector.

- `parameters` is a DMatrix object of dimensions

  `problem.phases(iphase).nparameters` $\times$ `1`

  with given values for the static parameters.

- `tolerance` is a positive value for the tolerance against which the maximum relative error in the state vector is compared.

- `hmin` is the minimum integration step size.

- `hmax` is the maximum integration step size.

- `problem` is a `Prob` structure.

- `iphase` is the phase index.

- `state_trajectory` is a DMatrix object with dimensions

  `problem.phases(iphase).nstates` $\times$ `M`

  which on output contains the result of the of the propagation. The values of the states correspond to the time vector `new_time_vector`.

- `new_time_vector` is a DMatrix object with dimensions $1 \times$ `N`

  which on output contains the time values of the propagation.

- `new_control_trajectory` is a DMatrix object with dimensions

  `problem.phases(iphase).nstates` $\times$ `N`

  which on output containts interpolated values of the control trajectory corresponding to the time vector `new_time_vector`. Linear interpolation is employed.

2.10.34 `resample_trajectory` function

This function resamples a trajectory given new values of the time vector using natural cubic spline interpolation.

```
void resample_trajectory(DMatrix& Y,
                         DMatrix& t,
                         DMatrix& Ydata,
                         DMatrix& tdata )
```

- `Y` is, on output, a DMatrix object with dimensions $n_y \times N$ with the interpolated values of the dependent variable.

- `t` is a DMatrix object of dimensions $1 \times N$ with the values of the independent variable at which the interpolated values are required. The elements of this vector should be monotonically increasing, i.e. `t(j+1) > t(j)`. The following restrictions should be satisfied: `t(1)` $\geq$ `tdata(1)`, and `t(N)` $\leq$ `tdata(M)`.

- `Ydata` is a DMatrix object of dimensions $n_y \times M$ with the data values of the dependent variable.

- `tdata` is a DMatrix object of dimensions $1 \times M$ with the data values of the independent variable. The elements of this vector should be monotonically increasing, i.e. `t(j+1) > t(j)`.

2.11 Pre-defined constants

The following constants are defined within the header file `psopt.h`:

- `pi`: defined as 3.141592653589793;

- `inf`: defined as 1×10^{19}.

2.12 Standard output

$\mathcal{PSOPT}$ will by default produce output information on the screen as it runs. $\mathcal{PSOPT}$ will produce a short file with a summary of information named with the string provided in `algorithm.outfilelname`. This file contains the problem name, the total CPU time spent, the NLP solver used, the optimal value of the objective function, the values of the endpoint cost function and cost integrals, the initial and final time, the maximum discretization error, and the output string from the NLP solver.

Additionally, every time a $\mathcal{PSOPT}$ excutable is run, it will produce a file named `psopt_solution_$$$.txt` ($$$ represents the characters of

`problem.outfilename` which occur to the left of the file extension point
".”). This file contains the problem name, time stamps, a summary of the
algorithm options used, and results obtained, the final grid points, the final
control variables, the final state variables, the final static parameter values.
The file also contains a summary of all constraints functions associated with
the NLP problem, including their final scaled value, bounds, and scaling
factor used; a summary of the final NLP decision variables, including their
final unscaled values, bounds and scaling factors used; and a summary of the
mesh refinement process. An indication is given at the end of a constraint
line, or decision variable line, if a scaled constraint function or scaled deci-
sion variable is within `algorithm.nlp_tolerance` of one of its bounds, or if
a scaled constraint function or scaled decision variable has violated one of its
bounds by more than `algorithm.nlp_tolerance`. For parameter estima-
tion problems this file also contains the covariance matrix of the parameter
vector, and the 95% confidence interval for each estimated parameter.

LATEX code to produce a table with a summary of the mesh refinement
process is also automatically generated as described in section 2.8.4.

If `algorithm.print_level` is set to zero, then no output is produced.

2.13   Implementing your own problem

A template C++ file named `user.cxx` is provided in the directory

$$\texttt{psopt-master/PSOPT/examples/user}$$

This file can be modified by the user to implement their own problem.
and an executable can then be built easily.

2.13.1   Building the user code from Linux

After modifying the `user.cxx` code, open a command prompt and cd to the
base directory of the $\mathcal{PSOPT}$ installation. Then simply type

```
$ cd PSOPT/examples/user
$ make user
```

If no compilation errors occur, an executable named `user` should be
created in the directory `psopt-master/PSOPT/examples/user`.

2.13.2   Building the user code from Microsoft Windows

After modifying the `user.cxx` code, open a command prompt and cd to the
base directory of the $\mathcal{PSOPT}$ installation. Then simply type

```
> cd PSOPT\examples\user
> nmake -f Makefile.vc user.exe
```

If no compilation errors occur, an executable named `user.exe` should be created in the directory `psopt-master \PSOPT\examples\user`.

Chapter 3

Examples of using $\mathcal{PSOPT}$

The following examples have been mostly selected from the literature such that their solution can be compared with published results by consulting the references provided.

3.1 Alp rider problem

Consider the following optimal control problem, which is known in the literature as the Alp rider problem [3]. It is known as Alp rider because the minimum of the objective function forces the states to ride the path constraint. Minimize the cost functional

$$J = \int_0^{20} (100(x_1^2 + x_2^2 + x_3^2 + x_4^2) + 0.01(u_1^2 + u_2^2))\mathrm{d}t \qquad (3.1)$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{x}_1 &= -10x_1 + u_1 + u_2 \\
\dot{x}_2 &= -2x_2 + u_1 + 2u_2 \\
\dot{x}_3 &= -3x_3 + 5x_4 + u_1 - u_2 \\
\dot{x}_4 &= 5x_3 - 3x_4 + u_1 + 3u_2
\end{aligned}
\qquad (3.2)
$$

the path constraint

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 - 3p(t,3,12) - 3p(t,6,10) - 3p(t,10,16) - 8p(t,15,4) - 0.01 \le 0 \qquad (3.3)$$

where the exponential peaks are $p(t, a, b) = e^{-b(t-a)^2}$, and the boundary conditions are given by:

$$
\begin{aligned}
x_1(0) &= 2 \\
x_2(0) &= 1 \\
x_3(0) &= 2 \\
x_4(0) &= 1 \\
x_1(20) &= 2 \\
x_2(20) &= 3 \\
x_3(20) &= 1 \\
x_4(20) &= -2
\end{aligned}
\tag{3.4}
$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
///////////////        alpine_rider.cxx            ///////////////////
//////////////////////////////////////////////////////////////////////
///////////////           PSOPT  Example           ///////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:        Alp rider problem            //////////////////
//////// Last modified: 09 February 2009            //////////////////
//////// Reference:     Betts (2001)                //////////////////
//////// (See PSOPT handbook for full reference)    //////////////////
//////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009  //////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which////////////
//////// is distributed under the terms of the GNU Lesser ////////////
//////// General Public License (LGPL)              //////////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
//////////////////  Define auxiliary function   ////////////////////
//////////////////////////////////////////////////////////////////////

adouble pk( adouble t, double a, double b)
{
   return exp(-b*(t-a)*(t-a));
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   return 0;
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
   adouble x1 = states[CINDEX(1)];
   adouble x2 = states[CINDEX(2)];
   adouble x3 = states[CINDEX(3)];
   adouble x4 = states[CINDEX(4)];
   adouble u1 = controls[CINDEX(1)];
   adouble u2 = controls[CINDEX(2)];

   adouble L;
```

```
    L = 100.0*( x1*x1 + x2*x2 + x3*x3 + x4*x4 ) + 0.01*( u1*u1 + u2*u2 );

    return  L;
}


//////////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ///////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
    adouble x1 = states[CINDEX(1)];
    adouble x2 = states[CINDEX(2)];
    adouble x3 = states[CINDEX(3)];
    adouble x4 = states[CINDEX(4)];
    adouble u1 = controls[CINDEX(1)];
    adouble u2 = controls[CINDEX(2)];
    adouble t  = time;

    derivatives[CINDEX(1)] = -10*x1 + u1 + u2;
    derivatives[CINDEX(2)] = -2*x2 + u1 + 2*u2;
    derivatives[CINDEX(3)] = -3*x3 + 5*x4 + u1 - u2;
    derivatives[CINDEX(4)] =  5*x3 - 3*x4 + u1 + 3*u2;

    path[0] = x1*x1 + x2*x2 + x3*x3 + x4*x4
              - 3*pk(t,3,12) - 3*pk(t,6,10) - 3*pk(t,10,6) - 8*pk(t,15,4)
              - 0.01;

}

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function /////////////////////////////
//////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{
   adouble x1i = initial_states[ CINDEX(1) ];
   adouble x2i = initial_states[ CINDEX(2) ];
   adouble x3i = initial_states[ CINDEX(3) ];
   adouble x4i = initial_states[ CINDEX(4) ];
   adouble x1f = final_states[ CINDEX(1) ];
   adouble x2f = final_states[ CINDEX(2) ];
   adouble x3f = final_states[ CINDEX(3) ];
   adouble x4f = final_states[ CINDEX(4) ];


   e[ CINDEX(1) ] = x1i;
   e[ CINDEX(2) ] = x2i;
   e[ CINDEX(3) ] = x3i;
   e[ CINDEX(4) ] = x4i;
   e[ CINDEX(5) ] = x1f;
   e[ CINDEX(6) ] = x2f;
   e[ CINDEX(7) ] = x3f;
   e[ CINDEX(8) ] = x4f;

}


//////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////////
//////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}



//////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////////
//////////////////////////////////////////////////////////////////////////

int main(void)
{
```

```
///////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures //////////////////////////////
///////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

///////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  //////////////////////////////
///////////////////////////////////////////////////////////////////////

    problem.name         = "Alp rider problem";
    problem.outfilename              = "alpine.txt";

///////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
///////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages              = 0;

    psopt_level1_setup(problem);

///////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
///////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 4;
    problem.phases(1).ncontrols  = 2;
    problem.phases(1).nevents    = 8;
    problem.phases(1).npath      = 1;
    problem.phases(1).nodes                  = "[120]";



    psopt_level2_setup(problem, algorithm);

///////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ////////////////////////
///////////////////////////////////////////////////////////////////////

    problem.phases(1).bounds.lower.states = "[-4.0, -4.0, -4.0, -4.0]";

    problem.phases(1).bounds.upper.states = "[ 4.0,  4.0,  4.0,  4.0]";

    problem.phases(1).bounds.lower.controls = "[ -500.0, -500 ]";

    problem.phases(1).bounds.upper.controls = "[  500.0,  500 ]";

    problem.phases(1).bounds.lower.events =  "[2.0, 1.0, 2.0, 1.0, 2.0, 3.0, 1.0, -2.0]";

    problem.phases(1).bounds.upper.events = problem.phases(1).bounds.lower.events;

    problem.phases(1).bounds.upper.path        = 100.0;

    problem.phases(1).bounds.lower.path        = 0.0;

    problem.phases(1).bounds.lower.StartTime    = 0.0;

    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 20.0;

    problem.phases(1).bounds.upper.EndTime      = 20.0;


///////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  ////////////////////////////
///////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae            = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

///////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ////////////////////////
///////////////////////////////////////////////////////////////////////
```

```
    int nnodes          = problem.phases(1).nodes(1);

    DMatrix x_guess               = zeros(4,nnodes);

    x_guess(1,colon())     = linspace(2,1,nnodes);
    x_guess(2,colon())     = linspace(2,3,nnodes);
    x_guess(3,colon())     = linspace(2,1,nnodes);
    x_guess(4,colon())     = linspace(1,-2,nnodes);

    problem.phases(1).guess.controls     = zeros(2,nnodes);
    problem.phases(1).guess.states       = x_guess;
    problem.phases(1).guess.time         = linspace(0.0,20.0,nnodes+1);


////////////////////////////////////////////////////////////////////////
/////////////////// Enter algorithm options ////////////////////////////
////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max          = 1000;
    algorithm.nlp_tolerance         = 1.e-6;
    algorithm.nlp_method            = "IPOPT";
    algorithm.scaling               = "automatic";
    algorithm.derivatives           = "automatic";
    algorithm.jac_sparsity_ratio    = 0.20;
    algorithm.collocation_method    = "Legendre";
    algorithm.diff_matrix           = "central-differences";
    algorithm.mesh_refinement       = "automatic";
    algorithm.mr_max_increment_factor = 0.3;
    algorithm.defect_scaling        = "jacobian-based";




////////////////////////////////////////////////////////////////////////
/////////////////// Now call PSOPT to solve the problem  ////////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////// Extract relevant variables from solution structure  ///////
////////////////////////////////////////////////////////////////////////


    DMatrix x = solution.get_states_in_phase(1);
    DMatrix u = solution.get_controls_in_phase(1);
    DMatrix t = solution.get_time_in_phase(1);


////////////////////////////////////////////////////////////////////////
/////////////  Save solution data to files if desired ///////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");


////////////////////////////////////////////////////////////////////////
/////////////  Plot some results if desired (requires gnuplot) //////////
////////////////////////////////////////////////////////////////////////

    plot(t,x(1,colon()),problem.name+": state", "time (s)", "state","x1");

    plot(t,x(2,colon()),problem.name+": state", "time (s)", "state","x2");

    plot(t,x(3,colon()),problem.name+": state", "time (s)", "state","x3");

    plot(t,x(4,colon()),problem.name+": state", "time (s)", "state","x4");

    plot(t,u(1,colon()),problem.name+": control","time (s)", "control", "u1");

    plot(t,u(2,colon()),problem.name+": control","time (s)", "control", "u2");

    plot(t,x(1,colon()),problem.name+": state x1", "time (s)", "state","x1",
                                   "pdf", "alpine_state1.pdf");

    plot(t,x(2,colon()),problem.name+": state x2", "time (s)", "state","x2",
                                   "pdf", "alpine_state2.pdf");
```

```
    plot(t,x(3,colon()),problem.name+": state x3", "time (s)", "state","x2",
                        "pdf", "alpine_state3.pdf");

    plot(t,x(4,colon()),problem.name+": state x4", "time (s)", "state","x2",
                        "pdf", "alpine_state4.pdf");

    plot(t,u(1,colon()),problem.name+": control u1","time (s)", "control", "u1",
                        "pdf", "alpine_control1.pdf");

    plot(t,u(2,colon()),problem.name+": control u1","time (s)", "control", "u2",
                        "pdf", "alpine_control2.pdf");

}

////////////////////////////////////////////////////////////////////////
////////////////////////     END OF FILE     ////////////////////////////
////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and shown in Figures 3.1-3.4 and Figures 3.5-3.6, which contain the elements of the state and the control, respectively. Table 3.1 shows the mesh refinement history for this problem.

```
PSOPT results summary
=====================


Problem:  Alp rider problem
CPU time (seconds): 1.017628e+02
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  2.028577e+03
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  2.028577e+03
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.000000e+01
Phase 1 maximum relative local error: 2.901823e-03
NLP solver reports:  The problem solved!
```

3.2 Brachistochrone problem

Consider the following optimal control problem. Minimize the cost functional

$$J = t_f \tag{3.5}$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{x} &= v\sin(\theta) \\
\dot{y} &= v\cos(\theta) \\
\dot{v} &= g\cos(\theta)
\end{aligned}
\tag{3.6}
$$

Table 3.1: Mesh refinement statistics: Alp rider problem

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	ϵ_{max}	CPU_a
1	LGL-CD	120	722	609	528	528	146	0	63360	2.210e-03	2.733e+00
2	LGL-CD	125	752	634	2001	2002	490	0	250250	3.410e-03	1.174e+01
3	LGL-CD	126	758	639	4455	4457	661	0	561582	3.714e-03	1.983e+01
4	LGL-CD	127	764	644	771	773	266	0	98171	2.998e-03	6.110e+00
5	LGL-CD	132	794	669	1331	1332	427	0	175824	3.395e-03	1.067e+01
6	LGL-CD	133	800	674	962	963	278	0	128079	3.351e-03	7.103e+00
7	LGL-CD	172	1034	869	788	789	264	0	135708	2.902e-03	8.099e+00
CPU_b	-	-	-	-	-	-	-	-	-	-	3.549e+01
-	-	-	-	-	10836	10844	2532	0	1412974	-	1.018e+02

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, ϵ_{max} = maximum relative ODE error, CPU_a = CPU time in seconds spent by NLP algorithm, CPU_b = additional CPU time in seconds spent by PSOPT

Figure 3.1: State $x_1(t)$ for the Alp rider problem

109

Figure 3.2: State $x_2(t)$ for the Alp rider problem

Figure 3.3: State $x_3(t)$ for the Alp rider problem

110

Figure 3.4: State $x_4(t)$ for the Alp rider problem

Figure 3.5: Control $u_1(t)$ for the Alp rider problem

111

Figure 3.6: Control $u_2(t)$ for the Alp rider problem

and the boundary conditions

$$
\begin{aligned}
x(0) &= 0 \\
y(0) &= 0 \\
v(0) &= 0 \\
x(t_f) &= 2 \\
y(t_f) &= 2
\end{aligned}
\tag{3.7}
$$

where $g = 9.8$. A version of this problem was originally formulated by Johann Bernoulli in 1696 and is referred to as the *Brachistochrone* problem. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
//////////////////          brac1.cxx        //////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////          PSOPT  Example        /////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:                Brachistochrone problem   //////////////
//////// Last modified:        04 January 2009           //////////////
//////// Reference:            Bryson and Ho (1975)       //////////////
//////// (See PSOPT handbook for full reference)          //////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009       //////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////
//////// is distributed under the terms of the GNU Lesser //////////////
//////// General Public License (LGPL)                    //////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////
```

```
adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return tf;
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    return  0.0;
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's //////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
    adouble xdot, ydot, vdot;

    adouble x = states[ CINDEX(1) ];
    adouble y = states[ CINDEX(2) ];
    adouble v = states[ CINDEX(3) ];

    adouble theta = controls[ CINDEX(1) ];

    xdot = v*sin(theta);
    ydot = v*cos(theta);
    vdot = 9.8*cos(theta);

    derivatives[ CINDEX(1) ] = xdot;
    derivatives[ CINDEX(2) ] = ydot;
    derivatives[ CINDEX(3) ] = vdot;
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
    adouble x0 = initial_states[ CINDEX(1) ];
    adouble y0 = initial_states[ CINDEX(2) ];
    adouble v0 = initial_states[ CINDEX(3) ];
    adouble xf = final_states[ CINDEX(1)];
    adouble yf = final_states[ CINDEX(2)];

    e[ CINDEX(1) ] = x0;
    e[ CINDEX(2) ] = y0;
    e[ CINDEX(3) ] = v0;
    e[ CINDEX(4) ] = xf;
    e[ CINDEX(5) ] = yf;
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////
//////////////////////////////////////////////////////////////////////


int main(void)
{

//////////////////////////////////////////////////////////////////////
```

```
//////////////////  Declare key structures ////////////////////////////////
///////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

///////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  //////////////////////////////////
///////////////////////////////////////////////////////////////////////////

    problem.name          = "Brachistochrone Problem";

    problem.outfilename              = "brac1.txt";

///////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
///////////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                = 0;

    psopt_level1_setup(problem);


///////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
///////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 3;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 5;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes                = "[40]";

    psopt_level2_setup(problem, algorithm);


///////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ////////////////////////
///////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states    = "[ 0;  0;  0]";
    problem.phases(1).bounds.upper.states    = "[20; 20; 20]";

    problem.phases(1).bounds.lower.controls  = 0.0;
    problem.phases(1).bounds.upper.controls  = 2*pi;

    problem.phases(1).bounds.lower.events    = "[0,  0,  0,  2,  2]";
    problem.phases(1).bounds.upper.events    = "[0,  0,  0,  2,  2]";


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 0.0;
    problem.phases(1).bounds.upper.EndTime      = 10.0;


///////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////////
///////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;


///////////////////////////////////////////////////////////////////////////
//////////////////  set scaling factors (optional) //////////////////////////
///////////////////////////////////////////////////////////////////////////

//    problem.phases(1).scale.controls   = 1.0*ones(1,1);
//    problem.phases(1).scale.states     = 1.0*ones(3,1);
//    problem.phases(1).scale.events     = 1.0*ones(5,1);
//    problem.phases(1).scale.defects    = 1.0*ones(3,1);
//    problem.phases(1).scale.time       = 1.0;

//    problem.scale.objective            = 1.0;
```

```
//////////////////////////////////////////////////////////////////////
/////////////////// Define & register initial guess /////////////////////
//////////////////////////////////////////////////////////////////////


    DMatrix x0(3,20);

    x0(1,colon()) = linspace(0.0,1.0, 20);
    x0(2,colon()) = linspace(0.0,1.0, 20);
    x0(3,colon()) = linspace(0.0,1.0, 20);

    problem.phases(1).guess.controls      = ones(1,20);
    problem.phases(1).guess.states        = x0;
    problem.phases(1).guess.time          = linspace(0.0, 2.0, 20);

//////////////////////////////////////////////////////////////////////
/////////////////// Enter algorithm options ////////////////////////////
//////////////////////////////////////////////////////////////////////


    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
    algorithm.nlp_iter_max            = 1000;
    algorithm.nlp_tolerance           = 1.e-6;
//    algorithm.hessian   = "exact";
    algorithm.collocation_method      = "Legendre";




//////////////////////////////////////////////////////////////////////
/////////////////// Now call PSOPT to solve the problem  /////////////////
//////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

    if (solution.error_flag) exit(0);

//////////////////////////////////////////////////////////////////////
///////////// Extract relevant variables from solution structure   //////////
//////////////////////////////////////////////////////////////////////

    DMatrix x  = solution.get_states_in_phase(1);
    DMatrix u  = solution.get_controls_in_phase(1);
    DMatrix t  = solution.get_time_in_phase(1);
    DMatrix H        = solution.get_dual_hamiltonian_in_phase(1);
    DMatrix lambda       = solution.get_dual_costates_in_phase(1);

//////////////////////////////////////////////////////////////////////
///////////// Save solution data to files if desired /////////////////////
//////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");
    lambda.Save("p.dat");


//////////////////////////////////////////////////////////////////////
///////////// Plot some results if desired (requires gnuplot) //////////////
//////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name + ": states", "time (s)", "states", "x y v");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u");

    plot(t,H,problem.name + ": Hamiltonian", "time (s)", "H", "H");

    plot(t,lambda,problem.name + ": costates", "time (s)", "lambda", "lambda_1 lambda_2 lambda_3");

    plot(t,x,problem.name + ": states", "time (s)", "states", "x y v",
                            "pdf", "brac1_states.pdf");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u",
                           "pdf", "brac1_control.pdf");

    plot(t,H,problem.name + ": Hamiltonian", "time (s)", "H", "H",
                            "pdf", "brac1_hamiltonian.pdf");
```

```
    plot(t,lambda,problem.name + ": costates", "time (s)", "lambda", "lambda_1 lambda_2 lambda_3",
                              "pdf", "brac1_costates.pdf");


}

//////////////////////////////////////////////////////////////////////
//////////////////////////    END OF FILE    ///////////////////////////
//////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and shown in Figures 3.7, 3.8, which contain the elements of the state, and the control respectively.

```
PSOPT results summary
=====================


Problem:  Brachistochrone Problem
CPU time (seconds): 8.172660e-01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  8.247591e-01
Phase 1 endpoint cost function value: 8.247591e-01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 8.247591e-01
Phase 1 maximum relative local error: 2.774975e-07
NLP solver reports:  The problem solved!
```

3.3   Breakwell problem

Consider the following optimal control problem, which is known in the literature as the Breakwell problem [8]. The problem benefits from having an analytical solution, which is reported (with some errors) in the book by Bryson and Ho (1975). Minimize the cost functional.

$$J = \int_0^{t_f} u(t)^2 \mathrm{d}t \tag{3.8}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= u \end{aligned} \tag{3.9}$$

Figure 3.7: States for brachistochrone problem

Figure 3.8: Control for brachistochrone problem

117

the state dependent constraint

$$x(t) \leq l \tag{3.10}$$

where $l = 0.1$, $t_f = 1$. and the boundary conditions

$$
\begin{aligned}
x(0) &= 0 \\
v(0) &= 1 \\
x(t_f) &= 0 \\
v(t_f) &= -1
\end{aligned}
\tag{3.11}
$$

The analytical solution of the problem (valid for $0 \leq l \leq 1/6$) is given by:

$$
u(t) = \begin{cases}
-\frac{2}{3l}(1 - \frac{t}{3l}), & 0 \leq t \leq 3l \\
0, & 3l \leq t \leq 1 - 3l \\
-\frac{2}{3l}(1 - \frac{1-t}{3l}), & 1 - 3l \leq t \leq 1
\end{cases}
\tag{3.12}
$$

$$
x(t) = \begin{cases}
l\left(1 - \left(1 - \frac{t}{3l}\right)^3\right), & 0 \leq t \leq 3l \\
l, & 3l \leq t \leq 1 - 3l \\
l\left(1 - \left(1 - \frac{1-t}{3l}\right)^3\right), & 1 - 3l \leq t \leq 1
\end{cases}
\tag{3.13}
$$

$$
v(t) = \begin{cases}
\left(1 - \frac{t}{3l}\right)^2, & 0 \leq t \leq 3l \\
0, & 3l \leq t \leq 1 - 3l \\
\left(1 - \frac{1-t}{3l}\right)^2, & 1 - 3l \leq t \leq 1
\end{cases}
\tag{3.14}
$$

$$
\lambda_x(t) = \begin{cases}
\frac{2}{9l^2}, & 0 \leq t \leq 3l \\
0, & 3l \leq t \leq 1 - 3l \\
-\frac{2}{9l^2}, & 1 - 3l \leq t \leq 1
\end{cases}
\tag{3.15}
$$

$$
\lambda_v(t) = \begin{cases}
\frac{2}{3l}(1 - \frac{t}{3l}), & 0 \leq t \leq 3l \\
0, & 3l \leq t \leq 1 - 3l \\
\frac{2}{3l}(1 - \frac{1-t}{3l}), & 1 - 3l \leq t \leq 1
\end{cases}
\tag{3.16}
$$

where $\lambda_x(t)$ and $\lambda_v(t)$ are the costates. The analytical optimal value of the objective function is $J = 4/(9l) = 4.4444444$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
////////////////          bryson_max_range.cxx         //////////////////
//////////////////////////////////////////////////////////////////////
////////////////             PSOPT  Example            //////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:         Bryson maximum range problem     //////////////
//////// Last modified: 05 January 2009                  //////////////
//////// Reference:     Bryson and Ho (1975)             //////////////
//////// (See PSOPT handbook for full reference)         //////////////
//////////////////////////////////////////////////////////////////////
////////       Copyright (c) Victor M. Becerra, 2009     //////////////
//////////////////////////////////////////////////////////////////////
```

```
//////// This is part of the PSOPT software library, which///////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                    ///////////////
////////////////////////////////////////////////////////////////////////

#include "psopt.h"

////////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   return (0);
}

////////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    adouble u = controls[0];
    return  0.5*u*u;
}


////////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ///////////////////////////////////
////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble xdot, vdot;

   double g = 1.0;
   double a = 0.5*g;

   adouble x = states[ CINDEX(1) ];
   adouble v = states[ CINDEX(2) ];

   adouble u = controls[ CINDEX(1) ];


   xdot = v;
   vdot = u;

   derivatives[ CINDEX(1) ] = xdot;
   derivatives[ CINDEX(2) ] = vdot;

}

////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function /////////////////////////
////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{
   adouble x0 = initial_states[ CINDEX(1) ];
   adouble v0 = initial_states[ CINDEX(2) ];
   adouble xf = final_states[ CINDEX(1) ];
   adouble vf = final_states[ CINDEX(2) ];

   e[ CINDEX(1) ] = x0;
   e[ CINDEX(2) ] = v0;
   e[ CINDEX(3) ] = xf;
   e[ CINDEX(4) ] = vf;

}


////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function /////////////////
////////////////////////////////////////////////////////////////////////
```

```
void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////
/////////////////// Define the main routine ////////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////
/////////////////// Declare key structures //////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
/////////////////// Register problem name ///////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name         = "Breakwell Problem";
    problem.outfilename            = "breakwell.txt";

//////////////////////////////////////////////////////////////////////
//////////// Define problem level constants & do level 1 setup /////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages              = 0;

    psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 2;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 4;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes            = "[200]";

    psopt_level2_setup(problem, algorithm);

//////////////////////////////////////////////////////////////////////
/////////////////// Declare DMatrix objects to store results //////////////
//////////////////////////////////////////////////////////////////////

    DMatrix x, u, t;
    DMatrix lambda, H;

//////////////////////////////////////////////////////////////////////
/////////////////// Enter problem bounds information ////////////////////
//////////////////////////////////////////////////////////////////////

    double xL = -2.0;
    double vL = -2.0;
    double xU = 0.1;
    double vU = 2.0;

    double uL = -10.0;
    double uU = 10.0;


    double x0 = 0.0;
    double v0 = 1.0;
    double xf = 0.0;
    double vf = -1.0;


    problem.phases(1).bounds.lower.states(1) = xL;
    problem.phases(1).bounds.lower.states(2) = vL;


    problem.phases(1).bounds.upper.states(1) = xU;
    problem.phases(1).bounds.upper.states(2) = vU;
```

```
    problem.phases(1).bounds.lower.controls(1) = uL;
    problem.phases(1).bounds.upper.controls(1) = uU;

    problem.phases(1).bounds.lower.events(1) = x0;
    problem.phases(1).bounds.lower.events(2) = v0;
    problem.phases(1).bounds.lower.events(3) = xf;
    problem.phases(1).bounds.lower.events(4) = vf;


    problem.phases(1).bounds.upper.events(1) = x0;
    problem.phases(1).bounds.upper.events(2) = v0;
    problem.phases(1).bounds.upper.events(3) = xf;
    problem.phases(1).bounds.upper.events(4) = vf;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.0;
    problem.phases(1).bounds.upper.EndTime      = 1.0;



////////////////////////////////////////////////////////////////////////
////////////////////  Register problem functions  ///////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae            = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
////////////////////  Define & register initial guess ///////////////////
////////////////////////////////////////////////////////////////////////

    int nnodes       = problem.phases(1).nodes(1);
    int ncontrols                        = problem.phases(1).ncontrols;
    int nstates                          = problem.phases(1).nstates;

    DMatrix x_guess    =  zeros(nstates,nnodes);

    x_guess(1,colon()) = x0*ones(1,nnodes);
    x_guess(2,colon()) = v0*ones(1,nnodes);


    problem.phases(1).guess.controls        = zeros(ncontrols,nnodes);
    problem.phases(1).guess.states          = x_guess;
    problem.phases(1).guess.time            = linspace(0.0,1.0,nnodes);



////////////////////////////////////////////////////////////////////////
////////////////////  Enter algorithm options  //////////////////////////
////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max              = 1000;
    algorithm.nlp_tolerance             = 1.e-6;
    algorithm.nlp_method                = "IPOPT";
    algorithm.scaling                   = "automatic";
    algorithm.derivatives               = "automatic";
//    algorithm.hessian                  = "exact";
//    algorithm.mesh_refinement          = "automatic";
    algorithm.collocation_method = "Hermite-Simpson";
//    algorithm.diff_matrix              = "central-differences";
//    algorithm.defect_scaling = "jacobian-based";
    algorithm.nlp_tolerance            = 1.e-6;



////////////////////////////////////////////////////////////////////////
////////////////////  Now call PSOPT to solve the problem   //////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////   Extract relevant variables from solution structure   ///////
////////////////////////////////////////////////////////////////////////
```

```
    DMatrix muE;

    x     = solution.get_states_in_phase(1);
    u     = solution.get_controls_in_phase(1);
    t     = solution.get_time_in_phase(1);
    lambda = solution.get_dual_costates_in_phase(1);
    H     = solution.get_dual_hamiltonian_in_phase(1);
    muE   = solution.get_dual_events_in_phase(1);


//////////////////////////////////////////////////////////////////////////
//////////// Compute the analytical solution, which is valid when the //////
//////////// state constraint x<=l, with 0<=l<=1/6. In this case l=0.1 /////
//////////// The analytical solution is given (with some errors) in    /////
//////////// the book by Bryson and Ho (1975), pages 121-122.          /////
//////////////////////////////////////////////////////////////////////////


    double l = 0.1;

    double t1 = 3*l;
    double t2 = 1.0-3*l;

    int nn = length(t);

    DMatrix ua(1,nn), xa(2,nn), pa(2,nn);

    for(int i=1;i <=nn;i++) {

        if (t(i)<t1) {
            ua(1,i) = -2.0/(3.0*l)*(1.0-t(i)/(3.0*l));
            xa(1,i) = l*( 1.0 - pow( (1.0-t(i)/(3.0*l)), 3.0) ) ;
            xa(2,i) = pow( 1.0 - t(i)/(3.0*l), 2.0);
            pa(1,i) = 2.0/(9.0*l*l);
            pa(2,i) = 2.0/(3.0*l)*(1.0-t(i)/(3*l));

        }
        else if (t(i)>=t1 && t(i)<t2) {

            ua(1,i) = 0.0;
            xa(1,i) = l;
            xa(2,i) = 0.0;
            pa(1,i) = 0.0;
            pa(2,i) = 0.0;

        }
        else if (t(i)>=t2) {
            ua(1,i) = -2.0/(3*l)*(1.0-(1.0-t(i))/(3.0*l));
            xa(1,i) = l*(1.0 - pow( (1.0-(1.0-t(i))/(3.0*l)), 3.0));
            xa(2,i) = -pow(1.0-(1.0-t(i))/(3.0*l), 2.0) ;
            pa(1,i) = -2.0/(9.0*l*l);
            pa(2,i) = 2.0/(3.0*l)*(1.0-(1.0-t(i))/(3*l));

        }


    }


//////////////////////////////////////////////////////////////////////////
//////////// Save solution data to files if desired ////////////////////////
//////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");
    lambda.Save("p.dat");
    H.Save("H.dat");

//////////////////////////////////////////////////////////////////////////
//////////// Plot some results if desired (requires gnuplot) ///////////////
//////////////////////////////////////////////////////////////////////////

    plot(t,x,t,xa,problem.name+": states", "time (s)", "states","x v xa va");

    plot(t,u,t,ua,problem.name+": controls","time (s)", "control", "u ua");

    plot(t,lambda,t,pa, problem.name+": costates","time (s)", "costates", "l_x l_v la_x la_v");

    plot(t,H,problem.name+": Hamiltonian","time (s)", "H", "H");
```

```
    plot(t,x,t,xa,problem.name+": states", "time (s)", "states","x v xa va", "pdf", "breakwell_states.pdf");

    plot(t,u,t,ua,problem.name+": control","time (s)", "control", "u ua", "pdf", "breakwell_control.pdf");

    plot(t,lambda,t,pa, problem.name+": costates","time (s)", "costates", "l_x l_v la_x la_v", "pdf", "breakwell_costates.pdf");

}
//////////////////////////////////////////////////////////////////////////
/////////////////////////     END OF FILE     ///////////////////////////
//////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the following box and shown in Figures 3.9 and 3.10, which contain the elements of the state and the control, respectively, and Figure 3.11 which shows the costates. The figures include curves with the analytical solution for each variable, which is very close to the computed solution.

```
PSOPT results summary
=====================


Problem:  Breakwell Problem
CPU time (seconds): 4.680000e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  4.444441e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  4.444441e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 8.294796e-06
NLP solver reports:  The problem solved!
```

3.4   Bryson-Denham problem

Consider the following optimal control problem, which is known in the literature as the Bryson-Denham problem [7]. Minimize the cost functional

$$J = x_3(t_f) \tag{3.17}$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= u \\
\dot{x}_3 &= \tfrac{1}{2}u^2
\end{aligned}
\tag{3.18}
$$

Figure 3.9: States for Breakwell problem

Figure 3.10: Control for Breakwell problem

124

Figure 3.11: Costates for Breakwell problem

the state bound

$$0 \leq x_1 \leq 1/9 \tag{3.19}$$

and the boundary conditions

$$
\begin{aligned}
x_1(0) &= 0 \\
x_2(0) &= 1 \\
x_3(0) &= 0 \\
x_1(t_f) &= 0 \\
x_2(t_f) &= -1
\end{aligned}
\tag{3.20}
$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
/////////////////          PSOPT  Example          //////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:                Bryson-Denham problem    //////////////////
//////// Last modified:        05 January 2009          //////////////////
//////// Reference:            GPOPS Handbook           //////////////////
//////// (See PSOPT handbook for full reference)        //////////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009     //////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////////
//////// is distributed under the terms of the GNU Lesser //////////////////
//////// General Public License (LGPL)                  //////////////////
//////////////////////////////////////////////////////////////////////


#include "psopt.h"


//////////////////////////////////////////////////////////////////////
//////////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////
```

```
adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    adouble x3f = final_states[ CINDEX(3) ];

    return x3f;
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    return  0.0;
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ///////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace )
{

   adouble x1 = states[CINDEX(1)];
   adouble x2 = states[CINDEX(2)];
   adouble x3 = states[CINDEX(3)];

   adouble u = controls[CINDEX(1)];

   derivatives[ CINDEX(1) ] = x2;
   derivatives[ CINDEX(2) ] = u;
   derivatives[ CINDEX(3) ] = u*u/2;
}



//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x10 = initial_states[ CINDEX(1) ];
   adouble x20 = initial_states[ CINDEX(2) ];
   adouble x30 = initial_states[ CINDEX(3) ];
   adouble x1f = final_states[ CINDEX(1) ];
   adouble x2f = final_states[ CINDEX(2) ];

   e[ CINDEX(1) ] = x10;
   e[ CINDEX(2) ] = x20;
   e[ CINDEX(3) ] = x30;
   e[ CINDEX(4) ] = x1f;
   e[ CINDEX(5) ] = x2f;
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////
//////////////////////////////////////////////////////////////////////


int main(void)
{
```

```
//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ///////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name        = "Bryson-Denham Problem";
    problem.outfilename                = "bryden.txt";

//////////////////////////////////////////////////////////////////////
////////////  Declare problem level constants & do level 1 setup ///////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                = 0;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
//////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates   = 3;
    problem.phases(1).ncontrols = 1;
    problem.phases(1).nevents   = 5;
    problem.phases(1).npath     = 0;
    problem.phases(1).nodes                = "[10, 50]";

    psopt_level2_setup(problem, algorithm);

//////////////////////////////////////////////////////////////////////
//////////////////  Declare DMatrix objects to store results //////////////
//////////////////////////////////////////////////////////////////////

    DMatrix x, u, t;
    DMatrix lambda, H;

//////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ///////////////////////
//////////////////////////////////////////////////////////////////////


   problem.phases(1).bounds.lower.states(1)  = 0.0;
   problem.phases(1).bounds.lower.states(2)  = -10.0;
   problem.phases(1).bounds.lower.states(3)  = -10.0;

   problem.phases(1).bounds.upper.states(1)   = 1.0/9.0;
   problem.phases(1).bounds.upper.states(2)  = 10.0;
   problem.phases(1).bounds.upper.states(3)  = 10.0;

   problem.phases(1).bounds.lower.controls(1) = -10.0;
   problem.phases(1).bounds.upper.controls(1)   =  10.0;

   problem.phases(1).bounds.lower.events(1)  = 0.0;
   problem.phases(1).bounds.lower.events(2)  = 1.0;
   problem.phases(1).bounds.lower.events(3)  = 0.0;
   problem.phases(1).bounds.lower.events(4)  = 0.0;
   problem.phases(1).bounds.lower.events(5)  = -1.0;

   problem.phases(1).bounds.upper.events(1)  = 0.0;
   problem.phases(1).bounds.upper.events(2)  = 1.0;
   problem.phases(1).bounds.upper.events(3)  = 0.0;
   problem.phases(1).bounds.upper.events(4)  = 0.0;
   problem.phases(1).bounds.upper.events(5)  = -1.0;


   problem.phases(1).bounds.lower.StartTime    = 0.0;
   problem.phases(1).bounds.upper.StartTime    = 0.0;
   problem.phases(1).bounds.lower.EndTime      = 0.0;
   problem.phases(1).bounds.upper.EndTime      = 50.0;


//////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions ////////////////////////////
//////////////////////////////////////////////////////////////////////
```

```
    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess //////////////////////
////////////////////////////////////////////////////////////////////////

    DMatrix x0(3,10);

    x0(1,colon()) = linspace(0.0, 0.0, 10);
    x0(2,colon()) = linspace(1.0,-1.0, 10);
    x0(3,colon()) = linspace(0.0, 0.0, 10);

    problem.phases(1).guess.controls        = zeros(1,10);
    problem.phases(1).guess.states          = x0;
    problem.phases(1).guess.time            = linspace(0.0, 0.5, 10);


////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  /////////////////////////////
////////////////////////////////////////////////////////////////////////

    algorithm.nlp_method               = "IPOPT";
    algorithm.scaling                  = "automatic";
    algorithm.derivatives              = "automatic";
    algorithm.nlp_iter_max             = 1000;
    algorithm.nlp_tolerance            = 1.e-6;

////////////////////////////////////////////////////////////////////////
//////////////////// Now call PSOPT to solve the problem  ////////////////
////////////////////////////////////////////////////////////////////////


    psopt(solution, problem, algorithm);


////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure  /////////
////////////////////////////////////////////////////////////////////////

    x      = solution.get_states_in_phase(1);
    u      = solution.get_controls_in_phase(1);
    t      = solution.get_time_in_phase(1);
    lambda = solution.get_dual_costates_in_phase(1);
    H      = solution.get_dual_hamiltonian_in_phase(1);

////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired //////////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");
    lambda.Save("lambda.dat");
    H.Save("H.dat");

////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) /////////////
////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name, "time (s)", "states", "x1 x2 x3");

    plot(t,u, problem.name ,"time (s)", "control", "u");

    plot(t,lambda, problem.name ,"time (s)", "lambda", "1 2 3");

    plot(t,x,problem.name, "time (s)", "states", "x1 x2 x3",
                            "pdf", "bryden_states.pdf");

    plot(t,u, problem.name ,"time (s)", "control", "u",
                            "pdf", "bryden_control.pdf");

    plot(t,lambda, problem.name ,"time (s)", "lambda", "1 2 3",
                            "pdf", "bryden_lambda.pdf");
```

```
}
/////////////////////////////////////////////////////////////////////////
///////////////////////////    END OF FILE     //////////////////////////////
/////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the following box and shown in Figures 3.12 and 3.13, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
======================


Problem:  Bryson-Denham Problem
CPU time (seconds): 1.448254e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  3.999539e+00
Phase 1 endpoint cost function value: 3.999539e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 6.474159e-01
Phase 1 maximum relative local error: 6.441690e-05
NLP solver reports:  The problem solved!
```

3.5   Bryson's maximum range problem

Consider the following optimal control problem, which is known in the literature as the Bryson's maximum range problem [7]. Minimize the cost functional

$$J = x(t_f) \tag{3.21}$$

subject to the dynamic constraints

$$\begin{aligned}
\dot{x} &= vu_1 \\
\dot{y} &= vu_2 \\
\dot{v} &= a - gu_2
\end{aligned} \tag{3.22}$$

the path constraint

$$u_1^2 + u_2^2 = 1 \tag{3.23}$$

129

Figure 3.12: States for Bryson Denham problem

Figure 3.13: Control for Bryson Denham problem

and the boundary conditions

$$
\begin{array}{rcl}
x(0) & = & 0 \\
y(0) & = & 0 \\
v(0) & = & 0 \\
y(t_f) & = & 0.1
\end{array}
\tag{3.24}
$$

where $t_f = 2$, $g = 1$ and $a = 0.5g$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
///////////////          bryson_max_range.cxx          ////////////////////
//////////////////////////////////////////////////////////////////////
///////////////              PSOPT  Example           ////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:         Bryson maximum range problem     ///////////////
//////// Last modified: 05 January 2009                  ///////////////
//////// Reference:     Bryson and Ho (1975)             ///////////////
//////// (See PSOPT handbook for full reference)         ///////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009      ///////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which///////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                   ///////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   adouble x = final_states[0];

   return (-x);
}

//////////////////////////////////////////////////////////////////////
//////////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    return  0.0;
}


//////////////////////////////////////////////////////////////////////
//////////////////////  Define the DAE's ///////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble xdot, ydot, vdot;

   double g = 1.0;
   double a = 0.5*g;

   adouble x = states[ CINDEX(1) ];
   adouble y = states[ CINDEX(2) ];
   adouble v = states[ CINDEX(3) ];

   adouble u1 = controls[ CINDEX(1) ];
   adouble u2 = controls[ CINDEX(2) ];
```

```
   xdot = v*u1;
   ydot = v*u2;
   vdot = a-g*u2;

   derivatives[ CINDEX(1) ] = xdot;
   derivatives[ CINDEX(2) ] = ydot;
   derivatives[ CINDEX(3) ] = vdot;

   path[ CINDEX(1) ] = (u1*u1) + (u2*u2);

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{
   adouble x0 = initial_states[ CINDEX(1) ];
   adouble y0 = initial_states[ CINDEX(2) ];
   adouble v0 = initial_states[ CINDEX(3) ];
   adouble xf = final_states[ CINDEX(1) ];
   adouble yf = final_states[ CINDEX(2) ];

   e[ CINDEX(1) ] = x0;
   e[ CINDEX(2) ] = y0;
   e[ CINDEX(3) ] = v0;
   e[ CINDEX(4) ] = yf;

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}



//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name          = "Bryson Maximum Range Problem";
    problem.outfilename                 = "brymr.txt";

//////////////////////////////////////////////////////////////////////
///////////  Define problem level constants & do level 1 setup /////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                  = 0;

    psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ///////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 3;
```

```
    problem.phases(1).ncontrols  = 2;
    problem.phases(1).nevents    = 4;
    problem.phases(1).npath      = 1;
    problem.phases(1).nodes                  = "[20]";

    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////
//////////////////  Declare DMatrix objects to store results /////////////
////////////////////////////////////////////////////////////////////////

    DMatrix x, u, t;
    DMatrix lambda, H;

////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information /////////////////////
////////////////////////////////////////////////////////////////////////

    double xL = -10.0;
    double yL = -10.0;
    double vL = -10.0;
    double xU = 10.0;
    double yU = 10.0;
    double vU = 10.0;

    double u1L = -10.0;
    double u2L = -10.0;
    double u1U = 10.0;
    double u2U = 10.0;

    double x0 = 0.0;
    double y0 = 0.0;
    double v0 = 0.0;
    double yf = 0.1;

    problem.phases(1).bounds.lower.states(1) = xL;
    problem.phases(1).bounds.lower.states(2) = yL;
    problem.phases(1).bounds.lower.states(3) = vL;


    problem.phases(1).bounds.upper.states(1) = xU;
    problem.phases(1).bounds.upper.states(2) = yU;
    problem.phases(1).bounds.upper.states(3) = vU;


    problem.phases(1).bounds.lower.controls(1) = u1L;
    problem.phases(1).bounds.lower.controls(2) = u2L;
    problem.phases(1).bounds.upper.controls(1) = u1U;
    problem.phases(1).bounds.upper.controls(2) = u2U;

    problem.phases(1).bounds.lower.events(1) = x0;
    problem.phases(1).bounds.lower.events(2) = y0;
    problem.phases(1).bounds.lower.events(3) = v0;
    problem.phases(1).bounds.lower.events(4) = yf;


    problem.phases(1).bounds.upper.events(1) = x0;
    problem.phases(1).bounds.upper.events(2) = y0;
    problem.phases(1).bounds.upper.events(3) = v0;
    problem.phases(1).bounds.upper.events(4) = yf;

    problem.phases(1).bounds.upper.path(1) = 1.0;
    problem.phases(1).bounds.lower.path(1) = 1.0;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 2.0;
    problem.phases(1).bounds.upper.EndTime      = 2.0;



////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost    = &endpoint_cost;
    problem.dae              = &dae;
```

```
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
/////////////////// Define & register initial guess ////////////////////
////////////////////////////////////////////////////////////////////////

    int nnodes     = problem.phases(1).nodes(1);
    int ncontrols                    = problem.phases(1).ncontrols;
    int nstates                      = problem.phases(1).nstates;

    DMatrix x_guess    = zeros(nstates,nnodes);

    x_guess(1,colon()) = x0*ones(1,nnodes);
    x_guess(2,colon()) = y0*ones(1,nnodes);
    x_guess(3,colon()) = v0*ones(1,nnodes);

    problem.phases(1).guess.controls      = zeros(ncontrols,nnodes);
    problem.phases(1).guess.states        = x_guess;
    problem.phases(1).guess.time          = linspace(0.0,2.0,nnodes);


////////////////////////////////////////////////////////////////////////
/////////////////// Enter algorithm options /////////////////////////////
////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max           = 1000;
    algorithm.nlp_tolerance          = 1.e-4;
    algorithm.nlp_method             = "IPOPT";
    algorithm.scaling                = "automatic";
    algorithm.derivatives            = "automatic";
    algorithm.mesh_refinement        = "automatic";
    algorithm.collocation_method = "trapezoidal";
//    algorithm.defect_scaling = "jacobian-based";
    algorithm.ode_tolerance          = 1.e-6;



////////////////////////////////////////////////////////////////////////
/////////////////// Now call PSOPT to solve the problem  /////////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
/////////// Extract relevant variables from solution structure  //////////
////////////////////////////////////////////////////////////////////////


    x      = solution.get_states_in_phase(1);
    u      = solution.get_controls_in_phase(1);
    t      = solution.get_time_in_phase(1);
    lambda = solution.get_dual_costates_in_phase(1);
    H      = solution.get_dual_hamiltonian_in_phase(1);


////////////////////////////////////////////////////////////////////////
/////////// Save solution data to files if desired ///////////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");
    lambda.Save("lambda.dat");
    H.Save("H.dat");

////////////////////////////////////////////////////////////////////////
/////////// Plot some results if desired (requires gnuplot) //////////////
////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name+": states", "time (s)", "states","x y v");

    plot(t,u,problem.name+": controls","time (s)", "controls", "u_1 u_2");

    plot(t,x,problem.name+": states", "time (s)", "states","x y v",
                         "pdf", "brymr_states.pdf");

    plot(t,u,problem.name+": controls","time (s)", "controls", "u_1 u_2",
                         "pdf", "brymr_controls.pdf");
}

////////////////////////////////////////////////////////////////////////
```

Figure 3.14: States for Bryson's maximum range problem

```
////////////////////////       END OF FILE      ///////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and shown in Figures 3.14 and 3.15, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================


Problem:  Bryson Maximum Range Problem
CPU time (seconds): 5.812903e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  -2.523051e-01
Phase 1 endpoint cost function value: -2.523051e-01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.000000e+00
Phase 1 maximum relative local error: 1.292787e-05
NLP solver reports:  The problem solved!
```

Figure 3.15: Controls for Bryson's maximum range problem

3.6 Catalytic cracking of gas oil

Consider the following optimization problem, which involves finding optimal static parameters subject to dynamic constraints [15]. Minimize

$$J = \sum_{i=1}^{21} (y_1(t_i) - y_{m,1}(i))^2 + (y_2(t_i) - y_{m,2}(i))^2 \qquad (3.25)$$

subject to the dynamic constraints

$$\begin{aligned} \dot{y}_1 &= -(\theta_1 + \theta_3)y_1^2 \\ \dot{y}_2 &= \theta_1 y_1^2 - \theta_2 y_2 \end{aligned} \qquad (3.26)$$

the parameter constraint

$$\begin{aligned} \theta_1 &\geq 0 \\ \theta_2 &\geq 0 \\ \theta_3 &\geq 0 \end{aligned} \qquad (3.27)$$

Note that, given the nature of the problem, the parameter estimation facilities of $\mathcal{PSOPT}$ are used in this example. In this case, the observations function is simple:

$$g(x(t), u(t), p, t) = [y_1 \ \ y_2]^T$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below. The code includes the values of the measurement vectors $y_{m,1}$, and $y_{m,2}$, as well as the vector of sampling instants $\theta_i, i = 1, \ldots, 21$.

```
//////////////////////////////////////////////////////////////////////
/////////////////        cracking.cxx      //////////////////////////
//////////////////////////////////////////////////////////////////////
///////////////          PSOPT Example           ////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:  Catalytic Cracking of Gas Oil        ///////////////
//////// Last modified: 15 January 2009               ///////////////
//////// Reference:    Users guide for DIRCOL     ////////////////
//////// (See PSOPT handbook for full reference)      ///////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009       ///////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                ///////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
/////////////////////  Define the observation function ///////////
//////////////////////////////////////////////////////////////////////



void  observation_function( adouble* observations,
                            adouble* states, adouble* controls,
                            adouble* parameters, adouble& time, int k,
                            adouble* xad, int iphase, Workspace* workspace)
{

     observations[ CINDEX(1) ] = states[ CINDEX(1) ];
     observations[ CINDEX(2) ] = states[ CINDEX(2) ];
}



//////////////////////////////////////////////////////////////////////
/////////////////////  Define the DAE's /////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

   adouble y1 = states[CINDEX(1)];
   adouble y2 = states[CINDEX(2)];

   adouble theta1 = parameters[ CINDEX(1) ];
   adouble theta2 = parameters[ CINDEX(2) ];
   adouble theta3 = parameters[ CINDEX(3) ];

   derivatives[CINDEX(1)] = -(theta1 + theta3)*y1*y1;
   derivatives[CINDEX(2)] =  theta1*y1*y1 - theta2*y2;

}

//////////////////////////////////////////////////////////////////////
/////////////////////  Define the events function ///////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   // No events

}


//////////////////////////////////////////////////////////////////////
/////////////////////  Define the phase linkages function ///////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}
```

137

```
//////////////////////////////////////////////////////////////////////
/////////////////// Define the main routine ///////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{

   DMatrix y1meas, y2meas, tmeas;
   // Measured values of y1
   y1meas =  "[1.0,0.8105,0.6208,0.5258,0.4345,0.3903,0.3342,0.3034, \
                 0.2735,0.2405,0.2283,0.2071,0.1669,0.153,0.1339,0.1265, \
                 0.12,0.099,0.087,0.077,0.069]";
   // Measured values of y2
   y2meas =  "[0.0,0.2,0.2886,0.301,0.3215,0.3123,0.2716,0.2551,0.2258, \
                 0.1959,0.1789,0.1457,0.1198,0.0909,0.0719,0.0561,0.046, \
               0.028,0.019,0.014,0.01]";
   // Sampling instants
   tmeas =    "[0.0,0.025,0.05,0.075,0.1,0.125,0.15,0.175,0.2,0.225,0.25, \
             0.3,0.35,0.4,0.45,0.5,0.55,0.65,0.75,0.85,0.95]";



//////////////////////////////////////////////////////////////////////
/////////////////// Declare key structures ////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
/////////////////// Register problem name  /////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name           = "Catalytic cracking of gas oil";
    problem.outfilename                 = "cracking.txt";

//////////////////////////////////////////////////////////////////////
///////////// Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                 = 0;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 2;
    problem.phases(1).ncontrols  = 0;
    problem.phases(1).nevents    = 0;
    problem.phases(1).npath      = 0;
    problem.phases(1).nparameters        = 3;
    problem.phases(1).nodes           = "[80]";
    problem.phases(1).nobserved   = 2;
    problem.phases(1).nsamples    = 21;

    psopt_level2_setup(problem, algorithm);

//////////////////////////////////////////////////////////////////////
///////////// Enter estimation information                    ///////////
//////////////////////////////////////////////////////////////////////


    problem.phases(1).observation_nodes     = tmeas;
    problem.phases(1).observations          = (y1meas && y2meas);
    problem.phases(1).residual_weights      = ones(2,21);


//////////////////////////////////////////////////////////////////////
/////////////////// Declare DMatrix objects to store results //////////////
//////////////////////////////////////////////////////////////////////

    DMatrix x, p, t;

//////////////////////////////////////////////////////////////////////
/////////////////// Enter problem bounds information //////////////////////
//////////////////////////////////////////////////////////////////////
```

```
    problem.phases(1).bounds.lower.states(1) =  0.0;
    problem.phases(1).bounds.lower.states(2) =  0.0;


    problem.phases(1).bounds.upper.states(1) =  2.0;
    problem.phases(1).bounds.upper.states(2) =  2.0;


    problem.phases(1).bounds.lower.parameters(1) = 0.0;
    problem.phases(1).bounds.lower.parameters(2) = 0.0;
    problem.phases(1).bounds.lower.parameters(3) = 0.0;
    problem.phases(1).bounds.upper.parameters(1) = 20.0;
    problem.phases(1).bounds.upper.parameters(2) = 20.0;
    problem.phases(1).bounds.upper.parameters(3) = 20.0;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 0.95;
    problem.phases(1).bounds.upper.EndTime      = 0.95;

////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////

    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;
    problem.observation_function = & observation_function;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess /////////////////////
////////////////////////////////////////////////////////////////////////

    DMatrix state_guess(2, 40);

    state_guess(1,colon()) = linspace(1.0,0.069, 40);
    state_guess(2,colon()) = linspace(0.30,0.01,  40);


    problem.phases(1).guess.states          = state_guess;
    problem.phases(1).guess.time            = linspace(0.0, 0.95, 40);
    problem.phases(1).guess.parameters      = zeros(3,1);


////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ////////////////////////////
////////////////////////////////////////////////////////////////////////

    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
//    algorithm.collocation_method       = "Hermite-Simpson";
    algorithm.nlp_iter_max            = 1000;
    algorithm.nlp_tolerance           = 1.e-4;
    algorithm.jac_sparsity_ratio      = 0.52;

////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   ////////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   ////////
////////////////////////////////////////////////////////////////////////

    x = solution.get_states_in_phase(1);
    t = solution.get_time_in_phase(1);
    p = solution.get_parameters_in_phase(1);


////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired //////////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    t.Save("t.dat");
    p.Print("Estimated parameters");
```

```
//////////////////////////////////////////////////////////////////////
//////////// Plot some results if desired (requires gnuplot) ///////////////
//////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name, "time (s)", "states", "y1 y2");
    plot(t,x,problem.name, "time (s)", "states", "y1 y2",
                       "pdf", "cracking_states.pdf");

}

//////////////////////////////////////////////////////////////////////
//////////////////////////       END OF FILE       ////////////////////////////
//////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and shown in Figure 3.16, which shows the states of the system. The optimal parameters found were:

$$\theta_1 = 11.40825702$$
$$\theta_2 = 8.123367918 \tag{3.28}$$
$$\theta_3 = 1.668727477$$

```
PSOPT results summary
=====================


Problem:  Catalytic cracking of gas oil
CPU time (seconds): 9.730000e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  6.267999e-04
Phase 1 endpoint cost function value: 6.267999e-04
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 9.500000e-01
Phase 1 maximum relative local error: 6.762859e-10
NLP solver reports:  The problem solved!
```

3.7   Catalyst mixing problem

Consider the following optimal control problem, which attempts to determine the optimal mixing policy of two catalysts along the length of a tubular plug flow reactor involving several reactions [39]. The catalyst mixing problem is a typical bang-singular-bang problem. Minimize the cost functional

$$J = -1 + x_1(t_f) + x_2(t_f) \tag{3.29}$$

Figure 3.16: States for catalytic cracking of gas oil problem

subject to the dynamic constraints

$$
\begin{array}{rcl}
\dot{x}_1 & = & u(10x_2 - x_1) \\
\dot{x}_2 & = & u(x_1 - 10x_2) - (1 - u)x_2
\end{array}
\tag{3.30}
$$

the boundary conditions

$$
\begin{array}{rcl}
x_1(0) & = & 1 \\
x_2(0) & = & 0 \\
x_1(t_f) & \leq & 0.95
\end{array}
\tag{3.31}
$$

and the box constraints:

$$
\begin{array}{rl}
0.9 & \leq x_1(t) \leq 1.0 \\
0 & \leq x_2(t) \leq 0.1 \\
0 & \leq u(t) \leq 1
\end{array}
\tag{3.32}
$$

where $t_f = 1$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////
///////////////////          catmix.cxx         ////////////////////
//////////////////////////////////////////////////////////////////
//////////////////          PSOPT  Example          ////////////////////
//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////
//////// Title:              Catalyst mixing problem   ////////////////
//////// Last modified:      26 January 2009           ////////////////
//////// Reference:          Vassiliadis (1999)        ////////////////
//////// (See PSOPT handbook for full reference)       ////////////////
//////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009     ////////////////
```

```
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                    ///////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    adouble x1f = final_states[ CINDEX(1) ];
    adouble x2f = final_states[ CINDEX(2) ];

    return -(1.0-x1f-x2f);
}

//////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters,  adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    return  0.0;
}

//////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
    adouble xdot, ydot, vdot;

    adouble x1 = states[ CINDEX(1) ];
    adouble x2 = states[ CINDEX(2) ];
    adouble u =  controls[ CINDEX(1) ];

    derivatives[ CINDEX(1) ] = u*(10*x2-x1);
    derivatives[ CINDEX(2) ] = u*(x1-10*x2) - (1-u)*x2;
}

//////////////////////////////////////////////////////////////////////
///////////////////  Define the events function /////////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
    adouble x10 = initial_states[ CINDEX(1)];
    adouble x20 = initial_states[ CINDEX(2) ];
    adouble x1f = final_states[   CINDEX(1) ];


    e[ CINDEX(1) ] = x10;
    e[ CINDEX(2) ] = x20;
    e[ CINDEX(3) ] = x1f;

}

//////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function /////////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ////////////////////////////////
//////////////////////////////////////////////////////////////////////
```

```
int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures /////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  /////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name        = "Catalyst mixing roblem";
    problem.outfilename             = "catmix.txt";

//////////////////////////////////////////////////////////////////////
///////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages              = 0;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup ////////////
//////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates   = 2;
    problem.phases(1).ncontrols = 1;
    problem.phases(1).nevents   = 3;
    problem.phases(1).npath     = 0;
    problem.phases(1).nodes             = 40;

    psopt_level2_setup(problem, algorithm);


//////////////////////////////////////////////////////////////////////
//////////////////  Declare DMatrix objects to store results /////////////
//////////////////////////////////////////////////////////////////////

    DMatrix x, u, t;
    DMatrix lambda, H;

//////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information /////////////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).bounds.lower.states(1) = 0.9;
    problem.phases(1).bounds.lower.states(2) = 0.0;

    problem.phases(1).bounds.upper.states(1) = 1.0;
    problem.phases(1).bounds.upper.states(2) = 0.1;

    problem.phases(1).bounds.lower.controls(1) = 0.0;
    problem.phases(1).bounds.upper.controls(1) = 1.0;

    problem.phases(1).bounds.lower.events(1) = 1.0;
    problem.phases(1).bounds.lower.events(2) = 0.0;
    problem.phases(1).bounds.lower.events(3) = 0.0;

    problem.phases(1).bounds.upper.events(1) = 1.0;
    problem.phases(1).bounds.upper.events(2) = 0.0;
    problem.phases(1).bounds.upper.events(3) = 0.95;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.0;
    problem.phases(1).bounds.upper.EndTime      = 1.0;


//////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
//////////////////////////////////////////////////////////////////////
```

```
    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;




////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ////////////////////
////////////////////////////////////////////////////////////////////


    DMatrix u0(1,40);
    DMatrix x0(3,40);



    DMatrix t0     =  linspace(0.0, 1.0, 40);

    x0(1,colon()) = ones(1,40) - 0.085*t0;
    x0(2,colon()) = 0.05*t0;
    u0            = ones(1,40)- t0;

    problem.phases(1).guess.controls       = u0;
    problem.phases(1).guess.states         = x0;
    problem.phases(1).guess.time           = t0;


////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ////////////////////////
////////////////////////////////////////////////////////////////////


    algorithm.nlp_method               = "IPOPT";
    algorithm.scaling                  = "automatic";
    algorithm.derivatives              = "automatic";
    algorithm.nlp_iter_max             = 1000;
    algorithm.nlp_tolerance            = 1.e-6;


////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////


    psopt(solution, problem, algorithm);


////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////

    x  = solution.get_states_in_phase(1);
    u  = solution.get_controls_in_phase(1);
    t  = solution.get_time_in_phase(1);
    lambda = solution.get_dual_costates_in_phase(1);
    H = solution.get_dual_hamiltonian_in_phase(1);

////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired /////////////////////
////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");
    lambda.Save("lambda.dat");
    H.Save("H.dat");

////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) //////////////
////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name + ": states", "time (s)", "states", "x1 x2");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u");

    plot(t,x,problem.name + ": states", "time (s)", "states", "x1 x2",
                         "pdf", "catmix_states.pdf");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u",
```

```
                        "pdf", "catmix_control.pdf");


}

/////////////////////////////////////////////////////////////////////////
///////////////////////        END OF FILE       ///////////////////////////
/////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.17 and 3.18, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================


Problem:  Catalyst mixing roblem
CPU time (seconds): 7.500000e-01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  -4.805320e-02
Phase 1 endpoint cost function value: -4.805320e-02
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 1.095216e-04
NLP solver reports:  The problem solved!
```

3.8   Coulomb friction

Consider the following optimal control problem, which consists of a system that exhibits Coulomb friction [30]. Minimize the cost:

$$J = t_f \tag{3.33}$$

subject to the dynamic constraints

$$\begin{aligned}
\ddot{q}_1 &= (-(k_1 - k_2)q_1 + k_2 q_2 - \mu\mathrm{sign}(\dot{q}_1) + u_1)/m_1 \\
\ddot{q}_2 &= (k_2 q_1 - k_2 q_2 - \mu\mathrm{sign}(\dot{q}_2) + u_2)/m_2
\end{aligned} \tag{3.34}$$

Figure 3.17: States for catalyist mixing problem

Figure 3.18: Control for catalyst mixing problem

and the boundary conditions

$$
\begin{aligned}
q_1(0) &= 0 \\
\dot{q}_1(0) &= -1 \\
q_2(0) &= 0 \\
\dot{q}_2(0) &= -2 \\
q_1(t_f) &= 1 \\
\dot{q}_1(t_f) &= 0 \\
q_2(t_f) &= 2 \\
\dot{q}_2(t_f) &= 0
\end{aligned}
\tag{3.35}
$$

where $k_1 = 0.95$, $k_2 = 0.85$, $\mu = 1.0$, $m_1 = 1.1$, $m_2 = 1.2$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
/////////////////          coulomb.cxx           /////////////////////
//////////////////////////////////////////////////////////////////////
///////////////          PSOPT  Example          /////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:               Coulumb friction problem  //////////////
//////// Last modified:        04 January 2009           //////////////
//////// Reference:   Driessen and Sadegh (2000)         //////////////
//////// (See PSOPT handbook for full reference)         //////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009      //////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////
//////// is distributed under the terms of the GNU Lesser //////////////
//////// General Public License (LGPL)                    //////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return tf;
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    return  0.0;
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's //////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble q1    = states[ CINDEX(1) ];
   adouble q1dot = states[ CINDEX(2) ];
   adouble q2    = states[ CINDEX(3) ];
   adouble q2dot = states[ CINDEX(4) ];

   adouble u1 = controls[ CINDEX(1) ];
   adouble u2 = controls[ CINDEX(2) ];
```

```
   double k1 = 0.95;
   double k2 = 0.85;
   double mu = 1.0;
   double m1 = 1.1;
   double m2 = 1.2;

   double epsilon = 0.01;

   derivatives[ CINDEX(1) ] = q1dot;
   derivatives[ CINDEX(2) ] = ( (-k1-k2)*q1+k2*q2-mu*smooth_sign(q1dot,epsilon)+u1 )/m1;
   derivatives[ CINDEX(3) ] = q2dot;
   derivatives[ CINDEX(4) ] = ( k2*q1-k2*q2-mu*smooth_sign(q2dot,epsilon)+u2 )/m2;
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ///////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{

   adouble q1_0    = initial_states[ CINDEX(1) ];
   adouble q1dot_0 = initial_states[ CINDEX(2) ];
   adouble q2_0    = initial_states[ CINDEX(3) ];
   adouble q2dot_0 = initial_states[ CINDEX(4) ];

   adouble q1_f    = final_states[ CINDEX(1) ];
   adouble q1dot_f = final_states[ CINDEX(2) ];
   adouble q2_f    = final_states[ CINDEX(3) ];
   adouble q2dot_f = final_states[ CINDEX(4) ];


   e[ CINDEX(1) ] = q1_0;
   e[ CINDEX(2) ] = q1dot_0;
   e[ CINDEX(3) ] = q2_0;
   e[ CINDEX(4) ] = q2dot_0;
   e[ CINDEX(5) ] = q1_f;
   e[ CINDEX(6) ] = q1dot_f;
   e[ CINDEX(7) ] = q2_f;
   e[ CINDEX(8) ] = q2dot_f;

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function //////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
   // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine //////////////////////////
//////////////////////////////////////////////////////////////////////


int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures //////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  //////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name         = "Coulomb friction problem";

    problem.outfilename                = "coulomb.txt";

//////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup /////////////
//////////////////////////////////////////////////////////////////////
```

```
    problem.nphases    = 1;
    problem.nlinkages                = 0;

    psopt_level1_setup(problem);


////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 4;
    problem.phases(1).ncontrols  = 2;
    problem.phases(1).nevents    = 8;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes                = 40;

    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information /////////////////////
////////////////////////////////////////////////////////////////////

    double q1_0 =            0.0;
    double dotq1_0 =       -1.0;
    double q2_0 =            0.0;
    double dotq2_0 =       -2.0;
    double q1_f =            1.0;
    double dotq1_f =        0.0;
    double q2_f =            2.0;
    double dotq2_f =        0.0;


    problem.phases(1).bounds.lower.states(1) = -2.0;
    problem.phases(1).bounds.lower.states(2) = -20.0;
    problem.phases(1).bounds.lower.states(3) = -2.0;
    problem.phases(1).bounds.lower.states(4) = -20.0;

    problem.phases(1).bounds.upper.states(1) =  2.0;
    problem.phases(1).bounds.upper.states(2) =  20.0;
    problem.phases(1).bounds.upper.states(3) =  2.0;
    problem.phases(1).bounds.upper.states(4) =  20.0;

    problem.phases(1).bounds.lower.controls(1) = -4.0;
    problem.phases(1).bounds.lower.controls(2) = -4.0;
    problem.phases(1).bounds.upper.controls(1) =  4.0;
    problem.phases(1).bounds.upper.controls(2) =  4.0;

    problem.phases(1).bounds.lower.events(1) = q1_0;
    problem.phases(1).bounds.lower.events(2) = dotq1_0;
    problem.phases(1).bounds.lower.events(3) = q2_0;
    problem.phases(1).bounds.lower.events(4) = dotq2_0;
    problem.phases(1).bounds.lower.events(5) = q1_f;
    problem.phases(1).bounds.lower.events(6) = dotq1_f;
    problem.phases(1).bounds.lower.events(7) = q2_f;
    problem.phases(1).bounds.lower.events(8) = dotq2_f;

    problem.phases(1).bounds.upper.events(1) = q1_0;
    problem.phases(1).bounds.upper.events(2) = dotq1_0;
    problem.phases(1).bounds.upper.events(3) = q2_0;
    problem.phases(1).bounds.upper.events(4) = dotq2_0;
    problem.phases(1).bounds.upper.events(5) = q1_f;
    problem.phases(1).bounds.upper.events(6) = dotq1_f;
    problem.phases(1).bounds.upper.events(7) = q2_f;
    problem.phases(1).bounds.upper.events(8) = dotq2_f;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.8;
    problem.phases(1).bounds.upper.EndTime      = 4.0;


////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  ////////////////////////
////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
```

```
    problem.linkages = &linkages;


//////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ////////////////////////
//////////////////////////////////////////////////////////////////////////


    DMatrix x0(2,40);

    x0(1,colon()) = linspace(q1_0,q1_f, 40);
    x0(2,colon()) = linspace(dotq1_0, dotq1_f, 40);
    x0(3,colon()) = linspace(q2_0, q2_f, 40);
    x0(4,colon()) = linspace(dotq2_0, dotq2_f, 40);

    problem.phases(1).guess.controls        = zeros(2,40);
    problem.phases(1).guess.states          = x0;
    problem.phases(1).guess.time            = linspace(0.0, 4.0,40);

//////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  ////////////////////////////
//////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
    algorithm.nlp_iter_max            = 1000;
    algorithm.nlp_tolerance           = 1.e-6;

//////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   //////////////////
//////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

    if (solution.error_flag) exit(0);

//////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
//////////////////////////////////////////////////////////////////////////
    DMatrix x, u, t;
    x  = solution.get_states_in_phase(1);
    u  = solution.get_controls_in_phase(1);
    t  = solution.get_time_in_phase(1);

//////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired /////////////////////////
//////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");


//////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ////////////////
//////////////////////////////////////////////////////////////////////////

    DMatrix q12 = x(1,colon()) && x(3,colon());

    plot(t,q12,problem.name + ": states q1 and q2",
                      "time (s)", "states", "q1 q2");

    plot(t,u,problem.name + ": controls", "time (s)", "control", "u1 u2");

    plot(t,q12,problem.name + ": states q1 and q2",
                      "time (s)", "states", "q1 q2",
                      "pdf", "coulomb_states.pdf");

    plot(t,u,problem.name + ": controls", "time (s)", "controls", "u1 u2",
                          "pdf", "coulomb_control.pdf");

}

//////////////////////////////////////////////////////////////////////////
//////////////////////       END OF FILE       ////////////////////////////
//////////////////////////////////////////////////////////////////////////
```

Figure 3.19: States for Coulomb friction problem

The output from $\mathcal{PSOPT}$ summarised in the box below and shown in Figures 3.19 and 3.20, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================


Problem:  Coulomb friction problem
CPU time (seconds): 1.900000e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  2.104798e+00
Phase 1 endpoint cost function value: 2.104798e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.104798e+00
Phase 1 maximum relative local error: 7.868328e-03
NLP solver reports:  The problem solved!
```

3.9   DAE index 3 parameter estimation problem

Consider the following parameter estimation problem, which involves a differential-algebraic equation of index 3 with four differential states and one algebraic

Figure 3.20: Controls for Coulomb friction problem

state [38].

The dynamics consists of the differential equations

$$
\begin{aligned}
\dot{x}_1(t) &= x_3(t) \\
\dot{x}_2(t) &= x_4(t) \\
\dot{x}_3(t) &= \lambda(t) x_1(t) \\
\dot{x}_4(t) &= \lambda(t) x_2(t)
\end{aligned}
\tag{3.36}
$$

and the algebraic equation

$$
0 = L^2 - x_1(t)^2 - x_2(t)^2 \tag{3.37}
$$

where $x_j(t), j = 1, \ldots, 4$ are the differential states, $\lambda(t)$ is an algebraic state (note that algebraic states are treated as control variables), and L is a parameter to be estimated.

The observations function is given by:

$$
\begin{aligned}
y_1 &= x_1 \\
y_2 &= x_2
\end{aligned}
\tag{3.38}
$$

And the following least squares objective is minimised:

$$
J = \sum_{k=1}^{n_s} \left[(y_1(t_k) - \hat{y}_1(t_k))^2 + (y_2(t_k) - \hat{y}_2(t_k))^2 \right] \tag{3.39}
$$

where $n_s = 20$, $t_1 = 0.5$ and $t_{20} = 10.0$.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
////////////////////////////////////////////////////////////////////
///////////////////        ident1.cxx        //////////////////////////
////////////////////////////////////////////////////////////////////
///////////////             PSOPT Example         /////////////////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
//////// Title:  DAE Index 3                          ///////////////
//////// Last modified:  07 June 2011                 ///////////////
//////// Reference:    Schittkowski (2002)            ///////////////
//////// (See PSOPT handbook for full reference)      ///////////////
////////////////////////////////////////////////////////////////////
////////       Copyright (c) Victor M. Becerra, 2011      ///////////////
////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                ///////////////
////////////////////////////////////////////////////////////////////

#include "psopt.h"

////////////////////////////////////////////////////////////////////
//////////////////  Define the observation function //////////
////////////////////////////////////////////////////////////////////



void  observation_function( adouble* observations,
                            adouble* states, adouble* controls,
                            adouble* parameters, adouble& time, int k,
                            adouble* xad, int iphase, Workspace* workspace)
{
      observations[ CINDEX(1) ] = states[ CINDEX(1) ];
      observations[ CINDEX(2) ] = states[ CINDEX(2) ];

}



////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ////////////////////////////////
////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

    // Variables
       adouble x1, x2, x3, x4, L, OMEGA, LAMBDA;
       adouble dx1, dx2, dx3, dx4;

    // Differential states
       x1 = states[CINDEX(1)];
       x2 = states[CINDEX(2)];
       x3 = states[CINDEX(3)];
       x4 = states[CINDEX(4)];

    // Algebraic variables
       LAMBDA = controls[CINDEX(1)];


    // Parameters
       L     = parameters[CINDEX(1)];
    // Differential equations

      dx1 = x3;

      dx2 = x4;

      dx3 = LAMBDA*x1;

      dx4 = LAMBDA*x2;

      derivatives[ CINDEX(1) ] = dx1;
      derivatives[ CINDEX(2) ] = dx2;
      derivatives[ CINDEX(3) ] = dx3;
      derivatives[ CINDEX(4) ] = dx4;


      // algebraic equation
```

```
      path[ CINDEX(1) ] = L*L - x1*x1 - x2*x2;

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
      // no events

      return;

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name            =        "DAE Index 3";
    problem.outfilename         =        "dae_i3.txt";

//////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases            = 1;
    problem.nlinkages                = 0;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////
/////////////   Define phase related information & do level 2 setup //////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates   = 4;
    problem.phases(1).ncontrols = 1;
    problem.phases(1).nevents   = 0;
    problem.phases(1).npath     = 1;
    problem.phases(1).nparameters     = 1;
    problem.phases(1).nodes        = "[10,20,30]";
    problem.phases(1).nobserved       = 2;
    problem.phases(1).nsamples        = 20;

    psopt_level2_setup(problem, algorithm);

//////////////////////////////////////////////////////////////////////
////////////   Load data for parameter estimation             ///////////
//////////////////////////////////////////////////////////////////////

  int iphase = 1;
  load_parameter_estimation_data(problem, iphase, "dae_i3.dat");
```

```
   problem.phases(1).observation_nodes.Print("observation nodes");
   problem.phases(1).observations.Print("observations");
   problem.phases(1).residual_weights.Print("weights");


//////////////////////////////////////////////////////////////////////
///////////////////  Declare DMatrix objects to store results //////////////
//////////////////////////////////////////////////////////////////////

   DMatrix x, u, p, t;

//////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
//////////////////////////////////////////////////////////////////////


   problem.phases(1).bounds.lower.states(1) = -2.0;
   problem.phases(1).bounds.lower.states(2) = -2.0;
   problem.phases(1).bounds.lower.states(3) = -2.0;
   problem.phases(1).bounds.lower.states(4) = -2.0;

   problem.phases(1).bounds.upper.states(1) = 2.0;
   problem.phases(1).bounds.upper.states(2) = 2.0;
   problem.phases(1).bounds.upper.states(3) = 2.0;
   problem.phases(1).bounds.upper.states(4) = 2.0;

   problem.phases(1).bounds.lower.controls(1) = -10.0;
   problem.phases(1).bounds.upper.controls(1) =  10.0;

   problem.phases(1).bounds.lower.parameters(1)  = 0.0;
   problem.phases(1).bounds.upper.parameters(1)  = 5.0;


   problem.phases(1).bounds.lower.path(1)  = 0.0;
   problem.phases(1).bounds.upper.path(1)  = 0.0;

   problem.phases(1).bounds.lower.StartTime    = 0.5;
   problem.phases(1).bounds.upper.StartTime    = 0.5;

   problem.phases(1).bounds.lower.EndTime     = 10.0;
   problem.phases(1).bounds.upper.EndTime     = 10.0;

//////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  //////////////////////////
//////////////////////////////////////////////////////////////////////

   problem.dae  = &dae;
   problem.events  = &events;
   problem.linkages = &linkages;
   problem.observation_function = & observation_function;

//////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess //////////////////////
//////////////////////////////////////////////////////////////////////

   int nnodes =     (int) problem.phases(1).nsamples;
   DMatrix state_guess(4, nnodes);
   DMatrix control_guess(1,nnodes);
   DMatrix param_guess(1,1);

   state_guess(1,colon()) = problem.phases(1).observations(1,colon());
   state_guess(2,colon()) = problem.phases(1).observations(2,colon());
   state_guess(3,colon()) = ones(1,nnodes);
   state_guess(4,colon()) = ones(1,nnodes);

   control_guess(1,colon()) = zeros(1,nnodes);

   param_guess = 0.5;


   problem.phases(1).guess.states        = state_guess;
   problem.phases(1).guess.time          = problem.phases(1).observation_nodes;
   problem.phases(1).guess.parameters    = param_guess;
   problem.phases(1).guess.controls      = control_guess;



//////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  /////////////////////////////
//////////////////////////////////////////////////////////////////////

   algorithm.nlp_method              = "IPOPT";
   algorithm.scaling                 = "automatic";
```

```
    algorithm.derivatives            = "automatic";
    algorithm.collocation_method     = "Legendre";
    algorithm.jac_sparsity_ratio     = 0.50;

////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   ///////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////   Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////

    x = solution.get_states_in_phase(1);
    u = solution.get_controls_in_phase(1);
    t = solution.get_time_in_phase(1);
    p = solution.get_parameters_in_phase(1);


////////////////////////////////////////////////////////////////////////
///////////   Save solution data to files if desired ///////////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");
    p.Print("Estimated parameter");


////////////////////////////////////////////////////////////////////////
///////////   Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////

    DMatrix tm;
    DMatrix ym;

    tm = problem.phases(1).observation_nodes;
    ym = problem.phases(1).observations;


    plot(t,x(1,colon()),tm,ym(1,colon()),problem.name, "time (s)", "state x1", "x1 yhat1");
    plot(t,x(2,colon()),tm,ym(2,colon()),problem.name, "time (s)", "state x2", "x2 yhat2");
    plot(t,u,problem.name, "time (s)", "algebraic state u", "u");

    plot(t,x(1,colon()),tm,ym(1,colon()),problem.name, "time (s)", "state x1", "x1 yhat1",
 "pdf", "x1.pdf");
    plot(t,x(2,colon()),tm,ym(2,colon()),problem.name, "time (s)", "state x2", "x2 yhat2",
 "pdf", "x2.pdf");
    plot(t,u,problem.name, "time (s)", "algebraic state lambda", "lambda", "pdf", "lambda.pdf");


}




////////////////////////////////////////////////////////////////////////
/////////////////////////     END OF FILE     ///////////////////////////
////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ summarised in the box below and shown in Figures 3.21 and 3.22, which compare the observations with the estimated outputs, and 3.23, which shows the algebraic state. The exact solution to the problem is $L = 1$ and $\lambda(t) = -1$. The numerical solution obtained is $L = 1.000000188$ and $\lambda(t) = -0.999868$. The 95% confidence interval for the estimated parameter is $[0.9095289, \ 1.090471]$

```
PSOPT results summary
```

156

Figure 3.21: State x_1 and observations

```
=====================

Problem:  DAE Index 3
CPU time (seconds): 1.046000e+01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  1.965352e+01
Phase 1 endpoint cost function value: 1.965352e+01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 5.000000e-01
Phase 1 final time: 1.000000e+01
Phase 1 maximum relative local error: 8.341290e-08
NLP solver reports:  The problem solved!
```

3.10 Delayed states problem 1

Consider the following optimal control problem, which consists of a linear system with delays in the state equations [30]. Minimize the cost functional:

$$J = x_3(t_f) \tag{3.40}$$

Figure 3.22: State x_2 and observations

Figure 3.23: Algebraic state $\lambda(t)$

158

subject to the dynamic constraints

$$\begin{aligned}
\dot{x}_1 &= x_2(t) \\
\dot{x}_2 &= -10x_1(t) - 5x_2(t) - 2x_1(t-\tau) - x_2(t-\tau) + u(t) \\
\dot{x}_3 &= 0.5(10x_1^2(t) + x_2^2(t) + u^2(t))
\end{aligned} \qquad (3.41)$$

and the boundary conditions

$$\begin{aligned}
x_1(0) &= 1 \\
x_2(0) &= 1 \\
x_3(0) &= 0
\end{aligned} \qquad (3.42)$$

where $t_f = 5$ and $\tau = 0.25$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
///////////////       delay1.cxx              ////////////////////////
//////////////////////////////////////////////////////////////////////
///////////////          PSOPT  Example       ////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:        Time delay problem         ////////////////////
//////// Last modified: 09 January 2009           ////////////////////
//////// Reference:    Luus (2002)                ////////////////////
//////// (See PSOPT handbook for full reference)  ////////////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009     //////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////
//////// is distributed under the terms of the GNU Lesser ////////////
//////// General Public License (LGPL)                    ////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    adouble x3f = final_states[CINDEX(3)];
    return x3f;
}


//////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    return  0.0;
}

//////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

    adouble x1delayed, x2delayed;
    double  tau = 0.25;
```

159

```
   adouble x1 = states[CINDEX(1)];
   adouble x2 = states[CINDEX(2)];
   adouble x3 = states[CINDEX(3)];

   get_delayed_state( &x1delayed, 1, iphase, time, tau, xad, workspace);
   get_delayed_state( &x2delayed, 2, iphase, time, tau, xad, workspace);

   adouble u = controls[CINDEX(1)];

   derivatives[CINDEX(1)] = x2;
   derivatives[CINDEX(2)] = -10*x1-5*x2-2*x1delayed-x2delayed+u;
// [uncomment the line below for approximate solution]
//   derivatives[CINDEX(2)] = (-12*x1+(2*tau-6)*x2+u)/(1-tau);
   derivatives[CINDEX(3)] = 0.5*(10*x1*x1+x2*x2+u*u);
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x10 = initial_states[CINDEX(1)];
   adouble x20 = initial_states[CINDEX(2)];
   adouble x30 = initial_states[CINDEX(3)];

   e[CINDEX(1)] = x10;
   e[CINDEX(2)] = x20;
   e[CINDEX(3)] = x30;

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name        = "Time delay problem 1";
    problem.outfilename                 = "delay1.txt";

//////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////


    problem.nphases    = 1;
    problem.nlinkages                  = 0;

    psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
//////////////////////////////////////////////////////////////////////
```

```
    problem.phases(1).nstates   = 3;
    problem.phases(1).ncontrols = 1;
    problem.phases(1).nevents   = 3;
    problem.phases(1).npath     = 0;

    problem.phases(1).nodes     = "[30]";

    psopt_level2_setup(problem, algorithm);


////////////////////////////////////////////////////////////////////////
//////////////////  Declare DMatrix objects to store results ////////////
////////////////////////////////////////////////////////////////////////

    DMatrix x, u, t;
    DMatrix lambda, H;

////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ////////////////////
////////////////////////////////////////////////////////////////////////

    double x1L = -100.0;
    double x2L = -100.0;
    double x3L = -100.0;
    double x1U = 100.0;
    double x2U = 100.0;
    double x3U = 100.0;

    double uL = -100.0;
    double uU =  100.0;

    problem.phases(1).bounds.lower.states(1) = x1L;
    problem.phases(1).bounds.lower.states(2) = x2L;
    problem.phases(1).bounds.lower.states(3) = x3L;

    problem.phases(1).bounds.upper.states(1) = x1U;
    problem.phases(1).bounds.upper.states(2) = x2U;
    problem.phases(1).bounds.upper.states(3) = x3U;

    problem.phases(1).bounds.lower.controls(1) = uL;
    problem.phases(1).bounds.upper.controls(1) = uU;

    problem.phases(1).bounds.lower.events(1) = 1.0;
    problem.phases(1).bounds.lower.events(2) = 1.0;
    problem.phases(1).bounds.lower.events(3) = 0.0;


    problem.phases(1).bounds.upper.events(1) = 1.0;
    problem.phases(1).bounds.upper.events(2) = 1.0;
    problem.phases(1).bounds.upper.events(3) = 0.0;



    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 5.0;
    problem.phases(1).bounds.upper.EndTime      = 5.0;


////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  ////////////////////////
////////////////////////////////////////////////////////////////////////

    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost   = &endpoint_cost;
    problem.dae             = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ////////////////////
////////////////////////////////////////////////////////////////////////


    DMatrix x0(3,60);

    x0(1,colon()) = linspace(1.0,1.0, 60);
    x0(2,colon()) = linspace(1.0,1.0, 60);
    x0(3,colon()) = linspace(0.0,0.0, 60);

    problem.phases(1).guess.controls        = zeros(1,60);
    problem.phases(1).guess.states          = x0;
```

161

```
    problem.phases(1).guess.time         = linspace(0.0,5.0, 60);

//////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  //////////////////////////
//////////////////////////////////////////////////////////////////////

    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
    algorithm.nlp_iter_max            = 1000;
    algorithm.nlp_tolerance           = 1.e-6;
    algorithm.collocation_method      = "Hermite-Simpson";
//    algorithm.mesh_refinement          = "automatic";


//////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem  ///////////////
//////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

//////////////////////////////////////////////////////////////////////
//////////// Extract relevant variables from solution structure  ///////
//////////////////////////////////////////////////////////////////////

    x = solution.get_states_in_phase(1);
    u = solution.get_controls_in_phase(1);
    t = solution.get_time_in_phase(1);

//////////////////////////////////////////////////////////////////////
//////////// Save solution data to files if desired ///////////////////
//////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

//////////////////////////////////////////////////////////////////////
//////////// Plot some results if desired (requires gnuplot) //////////
//////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name, "time (s)", "states", "x1 x2 x3");
    plot(t,u,problem.name, "time (s)", "control", "u");
    plot(t,x,problem.name, "time (s)", "states", "x1 x2 x3",
                     "pdf", "delay1_states.pdf");
    plot(t,u,problem.name, "time (s)", "control", "u",
                     "pdf", "delay1_controls.pdf");

}
```

The output from $\mathcal{PSOPT}$ summarised in the box below and shown in Figures 3.24 and 3.25, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================


Problem:  Time delay problem 1
CPU time (seconds): 1.300000e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  2.526875e+00
Phase 1 endpoint cost function value: 2.526875e+00
Phase 1 integrated part of the cost:  0.000000e+00
```

Figure 3.24: States for time delay problem 1

```
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.000000e+00
Phase 1 maximum relative local error: 1.848000e-02
NLP solver reports:  The problem solved!
```

3.11   Dynamic MPEC problem

Consider the following optimal control problem, which involves special handling of a system with a discontinuous right hand side [4]. Minimize the cost functional:

$$J = [y(2) - 5/3]^2 + \int_0^2 y^2(t)\mathrm{d}t \tag{3.43}$$

subject to

$$\dot{y} = 2 - \mathrm{sgn}(y) \tag{3.44}$$

and the boundary condition

$$y(0) = -1 \tag{3.45}$$

Note that there is no control variable, and the analytical solution of this problem satisfies $\dot{y}(t) = 3$, $0 \le t \le 1/3$, and $\dot{y}(t) = 1$, $1/3 \le t \le 2$.

In order to handle the discontinuous right hand side, the problem is converted into the following equivalent problem, which has three algebraic

163

Figure 3.25: Control for time delay problem 1

(control) variables. This type of problem is known in the literature as a dynamic MPEC problem.

$$J = [y(2) - 5/3]^2 + \int_0^2 \left(y^2(t) + \rho \left\{ p(t)[s(t) + 1] + q(t)[1 - s(t)] \right\} \right) dt \quad (3.46)$$

subject to

$$\begin{aligned} \dot{y} &= 2 - s(t) \\ 0 &= -y(t) - p(t) + q(t) \end{aligned} \quad (3.47)$$

the boundary condition

$$y(0) = -1 \quad (3.48)$$

and the bounds:

$$\begin{aligned} -1 &\leq s(t) \leq 1, \\ 0 &\leq p(t), \\ 0 &\leq q(t). \end{aligned} \quad (3.49)$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
///////////////        bryson_max_range.cxx        ////////////////////
//////////////////////////////////////////////////////////////////////
///////////////           PSOPT  Example           ////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:         Dynamic MPEC problem        ////////////////
//////// Last modified: 27 May 2011                 ////////////////
//////// Reference:     Betts (2010)                ////////////////
//////// (See PSOPT handbook for full reference)    ////////////////
//////////////////////////////////////////////////////////////////////
```

```
////////        Copyright (c) Victor M. Becerra, 2009        ///////////////
/////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                      ///////////////
/////////////////////////////////////////////////////////////////////////

#include "psopt.h"

/////////////////////////////////////////////////////////////////////////
////////////////////  Define the end point (Mayer) cost function //////////
/////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                       adouble* parameters,adouble& t0, adouble& tf,
                       adouble* xad, int iphase, Workspace* workspace)
{
   adouble y_f = final_states[ CINDEX(1) ];

   return pow( y_f - (5.0/3.0) , 2.0);
}

/////////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
/////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    adouble y = states[ CINDEX(1) ];
    adouble s = controls[ CINDEX(1) ];
    adouble p = controls[ CINDEX(2) ];
    adouble q = controls[ CINDEX(3) ];


    adouble retval;

    double rho = 1.e3;

    retval = y*y + rho*( p*(s+1.0)+ q*(1.0-s));

    return  retval;

}


/////////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's /////////////////////////////////////
/////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{


   adouble y = states[ CINDEX(1) ];
   adouble ydot;


   adouble s = controls[ CINDEX(1) ];
   adouble p = controls[ CINDEX(2) ];
   adouble q = controls[ CINDEX(3) ];

   ydot = 2.0 -s;

   derivatives[ CINDEX(1) ] = ydot;


   path[ CINDEX(1) ] = -y - p + q;

}

/////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ///////////////////////////
/////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{
   adouble y0 = initial_states[ CINDEX(1) ];
```

```
   e[ CINDEX(1) ] = y0;

}



//////////////////////////////////////////////////////////////////////
/////////////////// Define the phase linkages function ////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}



//////////////////////////////////////////////////////////////////////
/////////////////// Define the main routine ///////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////
/////////////////// Declare key structures /////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
/////////////////// Register problem name  /////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name          = "Dynamic MPEC problem";
    problem.outfilename                 = "mpec.txt";

//////////////////////////////////////////////////////////////////////
/////////////// Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                  = 0;

    psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////
/////////////    Define phase related information & do level 2 setup  ////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 1;
    problem.phases(1).ncontrols  = 3;
    problem.phases(1).nevents    = 1;
    problem.phases(1).npath      = 1;
    problem.phases(1).nodes             = "[20]";

    psopt_level2_setup(problem, algorithm);

//////////////////////////////////////////////////////////////////////
/////////////////// Declare DMatrix objects to store results //////////////
//////////////////////////////////////////////////////////////////////

    DMatrix y, controls, s, p, q, t;

//////////////////////////////////////////////////////////////////////
/////////////////// Enter problem bounds information ////////////////////
//////////////////////////////////////////////////////////////////////

    double y0 = -1.0;

    problem.phases(1).bounds.lower.states(1) = -10.0;


    problem.phases(1).bounds.upper.states(1) = 10.0;


    problem.phases(1).bounds.lower.controls(1) = -1.0;
```

```
    problem.phases(1).bounds.lower.controls(2) = 0.0;
    problem.phases(1).bounds.lower.controls(3) = 0.0;

    problem.phases(1).bounds.upper.controls(1) = 1.0;
    problem.phases(1).bounds.upper.controls(2) = inf;
    problem.phases(1).bounds.upper.controls(3) = inf;

    problem.phases(1).bounds.lower.events(1) = y0;
    problem.phases(1).bounds.upper.events(1) = y0;


    problem.phases(1).bounds.upper.path(1) = 0.0;
    problem.phases(1).bounds.lower.path(1) = 0.0;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 2.0;
    problem.phases(1).bounds.upper.EndTime      = 2.0;




////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae            = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess /////////////////////
////////////////////////////////////////////////////////////////////////

    int nnodes      = problem.phases(1).nodes(1);
    int ncontrols            = problem.phases(1).ncontrols;
    int nstates              = problem.phases(1).nstates;

    DMatrix x_guess    = zeros(nstates,nnodes);

    x_guess(1,colon()) = y0*ones(1,nnodes);

    problem.phases(1).guess.controls       = zeros(ncontrols,nnodes);
    problem.phases(1).guess.states         = x_guess;
    problem.phases(1).guess.time           = linspace(0.0,2.0,nnodes);


////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ////////////////////////////
////////////////////////////////////////////////////////////////////////


   algorithm.nlp_iter_max             = 1000;
   algorithm.nlp_tolerance            = 1.e-4;
   algorithm.nlp_method               = "IPOPT";
   algorithm.scaling                  = "automatic";
   algorithm.derivatives              = "automatic";
   algorithm.mesh_refinement          = "automatic";
   algorithm.collocation_method = "trapezoidal";
//   algorithm.defect_scaling = "jacobian-based";
   algorithm.ode_tolerance            = 1.e-6;



////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   ////////////////
////////////////////////////////////////////////////////////////////////

   psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
//////////  Extract relevant variables from solution structure  //////////
////////////////////////////////////////////////////////////////////////


   y       = solution.get_states_in_phase(1);
   controls = solution.get_controls_in_phase(1);
   t       = solution.get_time_in_phase(1);
   s       = controls(1,colon());
```

```
   p         = controls(2,colon());
   q         = controls(3,colon());


//////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
//////////////////////////////////////////////////////////////////////////

   y.Save("y.dat");
   controls.Save("controls.dat");
   t.Save("t.dat");

//////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
//////////////////////////////////////////////////////////////////////////

   plot(t,y,problem.name+": state", "time (s)", "state","y");

   plot(t,s,problem.name+": algebraic variable s","time (s)", "s", "s");

   plot(t,p,problem.name+": algebraic variable p","time (s)", "p", "p");

   plot(t,q,problem.name+": algebraic variable q","time (s)", "q", "q");


   plot(t,y,problem.name+": state", "time (s)", "state","y",
        "pdf", "y.pdf");

   plot(t,s,problem.name+": algebraic variable s","time (s)", "s", "s",
        "pdf", "s.pdf");

   plot(t,p,problem.name+": algebraic variable p","time (s)", "p", "p",
        "pdf", "p.pdf");

   plot(t,q,problem.name+": algebraic variable q","time (s)", "q", "q",
        "pdf", "q.pdf");



}
//////////////////////////////////////////////////////////////////////////
/////////////////////////    END OF FILE    ///////////////////////////////
//////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ summarised in the box below and shown in Figures 3.26, 3.27, 3.28, and 3.29.

```
PSOPT results summary
=====================


Problem:  Dynamic MPEC problem
CPU time (seconds): 3.250000e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  1.654401e+00
Phase 1 endpoint cost function value: 3.915611e-07
Phase 1 integrated part of the cost:  1.654401e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.000000e+00
Phase 1 maximum relative local error: 1.391591e-06
NLP solver reports:  The problem solved!
```

168

Figure 3.26: State y for dynamic MPEC problem

Figure 3.27: Algebraic variable s for dynamic MPEC problem

169

Figure 3.28: Algebraic variable p for dynamic MPEC problem

Figure 3.29: Algebraic variable q for dynamic MPEC problem

3.12 Goddard rocket maximum ascent problem

Consider the following optimal control problem, which is known in the literature as the Goddard rocket maximum ascent problem [6]. Find t_f and $T(t) \in [t_0, t_f]$ to minimize the cost functional

$$J = h(t_f) \tag{3.50}$$

subject to the dynamic constraints

$$
\begin{array}{rcl}
\dot{v} & = & \frac{1}{m}(T - D) - g \\
\dot{h} & = & v \\
\dot{m} & = & -\frac{T}{c}
\end{array}
\tag{3.51}
$$

the boundary conditions:

$$
\begin{array}{rcl}
h(0) & = & 0 \\
v(0) & = & 1 \\
m(0) & = & 1 \\
m(t_f) & = & 0.6
\end{array}
\tag{3.52}
$$

and the control bounds

$$0 \le T(t) \le 3.5 \tag{3.53}$$

where

$$
\begin{array}{rcl}
D & = & D_0 v^2 \exp(-\beta h) \\
g & = & 1/(h^2)
\end{array}
\ ,
\tag{3.54}
$$

$D_0 = 310$, $\beta = 500$, and $c = 0.5$, $0.1 \le t_f \le 1$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
///////////////////           goddard.cxx          /////////////////////
//////////////////////////////////////////////////////////////////////
///////////////////         PSOPT  Example         /////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:        Goddard rocket maximum ascent    ////////////////
//////// Last modified: 05 January 2009                 ////////////////
//////// Reference:     Bryson (1999)                   ////////////////
//////// (See PSOPT handbook for full reference)        ////////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009     ////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which /////////////
//////// is distributed under the terms of the GNU Lesser //////////////
//////// General Public License (LGPL)                  ////////////////
//////////////////////////////////////////////////////////////////////


#include "psopt.h"

//////////////////////////////////////////////////////////////////////
/////////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                 adouble* parameters,adouble& t0, adouble& tf,
                 adouble* xad, int iphase, Workspace* workspace)
{
```

```
   if (iphase==3) {
      adouble hf = final_states[1];
      return -hf;
   }
   else
      return 0.0;
}

//////////////////////////////////////////////////////////////////////
/////////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    return  0.0;
}

//////////////////////////////////////////////////////////////////////
/////////////////////  Define the DAE's /////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble vdot, hdot, mdot;

   adouble v = states[0];
   adouble h = states[1];
   adouble m = states[2];

   adouble T = controls[0];

   double c        = 1580.9425;
   double sigma    = 5.4915e-5;
   double g        = 32.174;
   double h0       = 23800.0;

   adouble D;

   D = sigma*v*v*exp(-h/h0);

   adouble mg = (1.0+v/c)*D;
   adouble gg = mg/m;

   vdot = 1.0/m*(T-D)-g;
   hdot = v;
   mdot = -T/c;

   derivatives[0] = vdot;
   derivatives[1] = hdot;
   derivatives[2] = mdot;

   if (iphase ==2 ) {

 //     path[0] = T-D-mg- mg*(c*c*(1.0+v/c)/(h0*g)-1.0-2.0*c/v)/(1.0+4.0*(c/v)+2.0*pow(c/v,2.0));

      path[0] = T-D-m*g- m*g*(c*c*(1.0+v/c)/(h0*g)-1.0-2.0*c/v)/(1.0+4.0*(c/v)+2.0*pow(c/v,2.0));
//      path[1] = m*g-(1.0+v/c)*sigma*v*v*exp(-h/h0);

   }
}

//////////////////////////////////////////////////////////////////////
/////////////////////  Define the events function /////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble vi = initial_states[0];
   adouble hi = initial_states[1];
   adouble mi = initial_states[2];
   adouble mf = final_states[2];

   if  (iphase == 1) {
     e[0] = vi;
     e[1] = hi;
     e[2] = mi;
```

```
    }

   if (iphase == 3) {
     e[0] = mf;
   }

}


//////////////////////////////////////////////////////////////////////
/////////////////////  Define the phase linkages function ////////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{

    double c        = 1580.9425;
    double sigma    = 5.4915e-5;
    double g        = 32.174;
    double h0       = 23800.0;

    int index = 0;


    adouble xi[3], v, h, m;
    get_initial_states(xi, xad, 2, workspace);

    v  =  xi[0];
    h  =  xi[1];
    m  =  xi[2];

    auto_link(linkages, &index, xad, 1, 2, workspace );
    auto_link(linkages, &index, xad, 2, 3, workspace );

//    linkages[8] = m*g-(1.0+v/c)*sigma*v*v*exp(-h/h0);

    get_final_states(xi, xad, 2, workspace);

    v  =  xi[0];
    h  =  xi[1];
    m  =  xi[2];


    linkages[8] = m*g-(1.0+v/c)*sigma*v*v*exp(-h/h0);
}


//////////////////////////////////////////////////////////////////////
/////////////////////  Define the main routine ////////////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{
//////////////////////////////////////////////////////////////////////
/////////////////////  Declare key structures ////////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
/////////////////////  Register problem name  ////////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name        = "Goddard Rocket Maximum Ascent";
    problem.outfilename                = "goddard.txt";

//////////////////////////////////////////////////////////////////////
////////////  Declare problem level constants & setup phases  //////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 3;
    problem.nlinkages                = 9;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////
/////////   Define phase related information &  setup PSOPT  ////////////////
//////////////////////////////////////////////////////////////////////

    int n1 = 20;
```

```
    int n2 = 10;
    int n3 = 30;


    problem.phases(1).nstates    = 3;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 3;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes                  = n1;

    problem.phases(2).nstates    = 3;
    problem.phases(2).ncontrols  = 1;
    problem.phases(2).nevents    = 0;
    problem.phases(2).npath      = 1;
    problem.phases(2).nodes                  = n2;

    problem.phases(3).nstates    = 3;
    problem.phases(3).ncontrols  = 1;
    problem.phases(3).nevents    = 1;
    problem.phases(3).npath      = 0;
    problem.phases(3).nodes                  = n3;

    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////
//////////////////  Declare DMatrix objects to store results ////////////
////////////////////////////////////////////////////////////////////////

    DMatrix x,u,t, x1, u1, t1, x2, u2, t2, x3, u3, t3;

////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ////////////////////
////////////////////////////////////////////////////////////////////////


    double v_L = 0;
    double h_L = 0.0;
    double m_L = 0.0;
    double v_U = 2000.0;
    double h_U = 30000.0;
    double m_U = 3.0;

    double T_L = 0.0;
    double T_U = 200.0;

    double v_i = 0.0;
    double h_i = 0.0;
    double m_i = 3.0;
    double m_f = 1.0;

    problem.phases(1).bounds.lower.states(1) = v_L;
    problem.phases(1).bounds.lower.states(2) = h_L;
    problem.phases(1).bounds.lower.states(3) = m_L;

    problem.phases(1).bounds.upper.states(1) = v_U;
    problem.phases(1).bounds.upper.states(2) = h_U;
    problem.phases(1).bounds.upper.states(3) = m_U;

    problem.phases(1).bounds.lower.controls(1) = T_L;
    problem.phases(1).bounds.upper.controls(1) = T_U;

    problem.phases(1).bounds.lower.events(1) = v_i;
    problem.phases(1).bounds.lower.events(2) = h_i;
    problem.phases(1).bounds.lower.events(3) = m_i;


    problem.phases(1).bounds.upper.events(1) = v_i;
    problem.phases(1).bounds.upper.events(2) = h_i;
    problem.phases(1).bounds.upper.events(3) = m_i;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 5.0;
    problem.phases(1).bounds.upper.EndTime      = 15.0;


    problem.phases(2).bounds.lower.states(1) = v_L;
    problem.phases(2).bounds.lower.states(2) = h_L;
    problem.phases(2).bounds.lower.states(3) = m_L;

    problem.phases(2).bounds.upper.states(1) = v_U;
```

```
    problem.phases(2).bounds.upper.states(2) = h_U;
    problem.phases(2).bounds.upper.states(3) = m_U;

    problem.phases(2).bounds.lower.controls(1) = T_L;
    problem.phases(2).bounds.upper.controls(1) = T_U;

    problem.phases(2).bounds.lower.path(1) = 0.0;
    problem.phases(2).bounds.upper.path(1) = 0.0;

//   problem.phases(2).bounds.lower.path(2) = 0.0;
//   problem.phases(2).bounds.upper.path(2) = 0.0;

    problem.phases(2).bounds.lower.StartTime    = 10.0;
    problem.phases(2).bounds.upper.StartTime    = 15.0;

    problem.phases(2).bounds.lower.EndTime      = 20.0;
    problem.phases(2).bounds.upper.EndTime      = 25.0;

    problem.phases(3).bounds.lower.states(1) = v_L;
    problem.phases(3).bounds.lower.states(2) = h_L;
    problem.phases(3).bounds.lower.states(3) = m_L;

    problem.phases(3).bounds.upper.states(1) = v_U;
    problem.phases(3).bounds.upper.states(2) = h_U;
    problem.phases(3).bounds.upper.states(3) = m_U;

    problem.phases(3).bounds.lower.controls(1) = T_L;
    problem.phases(3).bounds.upper.controls(1) = T_U;

  problem.phases(3).bounds.lower.events(1) = m_f;
  problem.phases(3).bounds.upper.events(1) = m_f;

    problem.phases(3).bounds.lower.StartTime    = 20.0;
    problem.phases(3).bounds.upper.StartTime    = 25.0;

    problem.phases(3).bounds.lower.EndTime      = 40.0;
    problem.phases(3).bounds.upper.EndTime      = 50.0;
////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess /////////////////////
////////////////////////////////////////////////////////////////////////


    DMatrix x0(3,n1);

    x0(1,colon()) = linspace(v_i,v_i, n1);
    x0(2,colon()) = linspace(h_i,h_i, n1);
    x0(3,colon()) = linspace(m_i,m_i, n1);

    problem.phases(1).guess.controls        = T_U*ones(1,n1);
    problem.phases(1).guess.states          = x0;
    problem.phases(1).guess.time            = linspace(0.0, 15.0, n1);

    x0.Resize(3,n2);

    x0(1,colon()) = linspace(v_i,v_i, n2);
    x0(2,colon()) = linspace(h_i,h_i, n2);
    x0(3,colon()) = linspace(m_i,m_i, n2);


    problem.phases(2).guess.controls        = zeros(1,n2);
    problem.phases(2).guess.states          = x0;
    problem.phases(2).guess.time            = linspace(15.0, 20.0, 20);

    x0.Resize(3,n3);

    x0(1,colon()) = linspace(v_i,v_i, n3);
    x0(2,colon()) = linspace(h_i,h_i, n3);
    x0(3,colon()) = linspace(m_i,m_i, n3);
```

```
    problem.phases(3).guess.controls      = zeros(1,n3);
    problem.phases(3).guess.states        = x0;
    problem.phases(3).guess.time          = linspace(20.0, 40.0, n3);


////////////////////////////////////////////////////////////////////////
//////////////////// Enter algorithm options ////////////////////////////
////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
    algorithm.nlp_iter_max            = 1000;
    algorithm.nlp_tolerance           = 1.e-6;
    algorithm.collocation_method      = "trapezoidal";
    algorithm.mesh_refinement         = "automatic";
    algorithm.mr_max_iterations       = 5;


////////////////////////////////////////////////////////////////////////
//////////////////// Now call PSOPT to solve the problem  ////////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////// Extract relevant variables from solution structure  ////////
////////////////////////////////////////////////////////////////////////

    x1 = solution.get_states_in_phase(1);
    u1 = solution.get_controls_in_phase(1);
    t1 = solution.get_time_in_phase(1);

    x2 = solution.get_states_in_phase(2);
    u2 = solution.get_controls_in_phase(2);
    t2 = solution.get_time_in_phase(2);

    x3 = solution.get_states_in_phase(3);
    u3 = solution.get_controls_in_phase(3);
    t3 = solution.get_time_in_phase(3);

    x = x1 || x2 || x3;

    u = u1 || u2 || u3;

    t = t1 || t2 || t3;


////////////////////////////////////////////////////////////////////////
///////////// Save solution data to files if desired /////////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

////////////////////////////////////////////////////////////////////////
///////////// Plot some results if desired (requires gnuplot) ////////////
////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name, "time (s)", "states", "v h m");

    plot(t,u,problem.name, "time (s)", "control", "T");

    plot(t,x,problem.name, "time (s)", "states", "v h m",
                        "pdf", "goddard_states.pdf");

    plot(t,u,problem.name, "time (s)", "control", "T",
                        "pdf", "goddard_control.pdf");

}

////////////////////////////////////////////////////////////////////////
/////////////////////////     END OF FILE     ////////////////////////////
////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown

in Figures 3.30 and 3.31, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Goddard Rocket Maximum Ascent
CPU time (seconds): 6.210000e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  -1.874466e+04
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.498379e+01
Phase 1 maximum relative local error: 1.917444e-02
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  0.000000e+00
Phase 2 initial time: 1.498379e+01
Phase 2 final time: 2.001830e+01
Phase 2 maximum relative local error: 7.284868e-08
Phase 3 endpoint cost function value: -1.874466e+04
Phase 3 integrated part of the cost:  0.000000e+00
Phase 3 initial time: 2.001830e+01
Phase 3 final time: 4.270924e+01
Phase 3 maximum relative local error: 1.373835e-02
NLP solver reports:  The problem solved!
```

3.13   Hang glider

This problem is about the range maximisation of a hang glider in the presence of a specified thermal draft [4]. Find t_f and $C_L(t), t \in [0, t_f]$, to minimise,

$$J = x(t_f) \tag{3.55}$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{x} &= v_x \\
\dot{y} &= v_y \\
\dot{v}_x &= \frac{1}{m}(-L\sin\eta - D\cos\eta) \\
\dot{v}_y &= \frac{1}{m}(L\cos\eta - D\sin\eta - W)
\end{aligned} \tag{3.56}
$$

Figure 3.30: States for Goddard rocket problem

Figure 3.31: Control for Goddard rocket problem

178

where

$$C_D = C_0 + kC_L^2$$

$$v_r = \sqrt{v_x^2 + v_y^2}$$

$$D = \frac{1}{2}C_D\rho Sv_r^2$$

$$L = \frac{1}{2}C_L\rho Sv_r^2$$

$$X = \left(\frac{x}{R} - 2.5\right)^2$$

$$u_a = u_M(1 - X)\exp(-X)$$

$$V_y = v_y - ua$$

$$\sin\eta = \frac{V_y}{v_r}$$

$$\cos\eta = \frac{v_x}{v_r}$$

$$W = mg$$

(3.57)

The control is bounded as follows:

$$0 \leq C_L \leq 1.4 \tag{3.58}$$

and the following boundary conditions:

$$\begin{aligned}
x(0) &= 0, & x(t_f) &= \text{free} \\
y(0) &= 1000, & y(t_f) &= 900 \\
v_x(0) &= 13.227567500, & v_x(t_f) &= 13.227567500 \\
v_y(0) &= -1.2875005200, & v_y(t_f) &= -1.2875005200
\end{aligned} \tag{3.59}$$

With the following parameter values:

$$\begin{aligned}
u_M &= 2.5, & m &= 100.0 \\
R &= 100.0, & S &= 14, \\
C_0 &= 0.034, & \rho &= 1.13 \\
k &= 0.069662, & g &= 9.80665
\end{aligned} \tag{3.60}$$

$\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
///////////////          glider.cxx              ////////////////////////
//////////////////////////////////////////////////////////////////////
///////////////          PSOPT example           ////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:  Hang glider problem             ////////////////
//////// Last modified: 11 July 2009             ///////////////
//////// Reference:     PROPT User Manual         ////////////////
////////                                          ///////////////
//////////////////////////////////////////////////////////////////////
```

```
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////
//////// is distributed under the terms of the GNU Lesser //////////////
//////// General Public License (LGPL)                     //////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"



//////////////////////////////////////////////////////////////////////
//////////////////// Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////


adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   adouble xf = final_states[CINDEX(1)];

   return -(xf);
}

//////////////////////////////////////////////////////////////////////
//////////////////// Define the integrand (Lagrange) cost function //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
   return 0.0;
}


//////////////////////////////////////////////////////////////////////
//////////////////// Define the DAE's //////////////////////////////////
//////////////////////////////////////////////////////////////////////


void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

    adouble CL  = controls[ CINDEX(1) ];

    adouble x  = states[ CINDEX(1) ];
    adouble y      = states[ CINDEX(2) ];
    adouble vx  = states[ CINDEX(3) ];
    adouble vy      = states[ CINDEX(4) ];

    double m  = 100.0,     g  = 9.80665;
    double uM = 2.5,       R  = 100.0;
    double C0 = 0.034,     k = 0.069662;
    double S  = 14.0,      rho = 1.13;

    adouble sin_eta, cos_eta, D, L, CD, Vy, ua, X, vr, W;


    CD = C0 + k*CL*CL;
    vr = sqrt(vx*vx + vy*vy);
    D = 0.5*CD*rho*S*vr*vr;
    L = 0.5*CL*rho*S*vr*vr;
    X  = pow(x/R - 2.5, 2.0);
    ua = uM*(1.0-X)*exp(-X);
    Vy = vy-ua;
    sin_eta = Vy/vr;
    cos_eta = vx/vr;
    W = m*g;


    derivatives[ CINDEX(1) ] =  vx;
    derivatives[ CINDEX(2) ] =  vy;
    derivatives[ CINDEX(3) ] =  1.0/m*(-L*sin_eta - D*cos_eta     );
    derivatives[ CINDEX(4) ] =  1.0/m*( L*cos_eta - D*sin_eta - W);



}

//////////////////////////////////////////////////////////////////////
```

```
//////////////////  Define the events function ////////////////////////////
//////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{

    adouble x_i  = initial_states[ CINDEX(1) ];
    adouble y_i  = initial_states[ CINDEX(2) ];
    adouble vx_i = initial_states[ CINDEX(3) ];
    adouble vy_i = initial_states[ CINDEX(4) ];

    adouble x_f  = final_states[ CINDEX(1) ];
    adouble y_f  = final_states[ CINDEX(2) ];
    adouble vx_f = final_states[ CINDEX(3) ];
    adouble vy_f = final_states[ CINDEX(4) ];

    e[ CINDEX(1) ]      = x_i;
    e[ CINDEX(2) ]      = y_i;
    e[ CINDEX(3) ]      = vx_i;
    e[ CINDEX(4) ]      = vy_i;

    e[ CINDEX(5) ]      = y_f;
    e[ CINDEX(6) ]   = vx_f;
    e[ CINDEX(7) ]   = vy_f;


}



//////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
//////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{

    // Single phase problem

}



//////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine //////////////////////////////
//////////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ///////////////////////////////
//////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ///////////////////////////////
//////////////////////////////////////////////////////////////////////////

    problem.name        = "Hang glider problem";
    problem.outfilename                = "hang_glider.txt";

//////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ///////////
//////////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages              = 0;

    psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  //////////
//////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates   = 4;
    problem.phases(1).ncontrols = 1;
```

```
    problem.phases(1).nevents    = 7;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes                    = "[30 40 50 80]";


    psopt_level2_setup(problem, algorithm);



////////////////////////////////////////////////////////////////////////
//////////////////// Enter problem bounds information ////////////////////
////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states = "[0.0      0.0    0.0  -4.0]";
    problem.phases(1).bounds.upper.states = "[1500.0  1100.0 15.0   4.0]";


    problem.phases(1).bounds.lower.controls(1) = 0.0;

    problem.phases(1).bounds.upper.controls(1) = 1.4;


    problem.phases(1).bounds.lower.events="[0.0,1000.0,13.2275675,-1.28750052,900.00,13.2275675,-1.28750052 ]";
    problem.phases(1).bounds.upper.events="[0.0,1000.0,13.2275675,-1.28750052,900.00,13.2275675,-1.28750052 ]";


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 0.1;
    problem.phases(1).bounds.upper.EndTime      = 200.0;



////////////////////////////////////////////////////////////////////////
//////////////////// Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae            = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
/////////////////// Define & register initial guess //////////////////////
////////////////////////////////////////////////////////////////////////

    int nnodes      = problem.phases(1).nodes(1);
    int ncontrols           = problem.phases(1).ncontrols;
    int nstates             = problem.phases(1).nstates;

    DMatrix state_guess    =  zeros(nstates,nnodes);
    DMatrix control_guess  =  1.0*ones(ncontrols,nnodes);
    DMatrix time_guess     =  linspace(0.0,105.0,nnodes);


    state_guess(1,colon()) = linspace(0.0, 1250, nnodes);
    state_guess(2,colon()) = linspace(1000.0, 900.0, nnodes);
    state_guess(3,colon()) = 13.23*ones(1,nnodes);
    state_guess(4,colon()) = -1.288*ones(1,nnodes);

    problem.phases(1).guess.states   = state_guess;
    problem.phases(1).guess.controls = control_guess;
    problem.phases(1).guess.time     = time_guess;



////////////////////////////////////////////////////////////////////////
/////////////////// Enter algorithm options ///////////////////////////////
////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max            = 1000;
    algorithm.nlp_tolerance           = 1.e-6;
    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";


////////////////////////////////////////////////////////////////////////
```

```
//////////////////  Now call PSOPT to solve the problem   //////////////////
/////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

/////////////////////////////////////////////////////////////////////////
//////////  Extract relevant variables from solution structure  //////////
/////////////////////////////////////////////////////////////////////////

    DMatrix states, CL, t, x, y, speeds;

    states     = solution.get_states_in_phase(1);
    CL         = solution.get_controls_in_phase(1);
    t          = solution.get_time_in_phase(1);



/////////////////////////////////////////////////////////////////////////
//////////  Save solution data to files if desired ///////////////////////
/////////////////////////////////////////////////////////////////////////

    states.Save("states.dat");
    CL.Save("cL.dat");
    t.Save("t.dat");


/////////////////////////////////////////////////////////////////////////
//////////  Plot some results if desired (requires gnuplot) ///////////////
/////////////////////////////////////////////////////////////////////////

  x  = states(1,colon());
  y  = states(2,colon());
  speeds = (states(3,colon()) && states(4,colon()));

  plot(x,y,problem.name+": trajectory","x [m]", "y [m]", "traj");

  plot(t,speeds,problem.name+": speeds","time (s)", "speeds [m/s]", "dxdt dydt");

  plot(t,CL,problem.name+": control","time (s)", "control", "CL");

  plot(x,y,problem.name+": trajectory","x [m]", "y [m]", "traj", "pdf", "traj.pdf");

  plot(t,speeds,problem.name+": speeds","time (s)", "speeds [m/s]", "dxdt dydt", "pdf", "velocities.pdf");

  plot(t,CL,problem.name+": control - lift coefficient","time (s)", "CL", "CL", "pdf", "control.pdf");


}
/////////////////////////////////////////////////////////////////////////
////////////////////////     END OF FILE     //////////////////////////////
/////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures .

3.14 Hanging chain problem

Consider the following optimal control problem, which includes an integral constraint. Minimize the cost functional

$$J = \int_0^{t_f} \left[x \sqrt{1 + (\dot{x})^2} \right] dt \tag{3.61}$$

Figure 3.32: $x - y$ trajectory for hang glider

Figure 3.33: Velocities for hang glider

184

Figure 3.34: Lift coefficient for hang glider problem

subject to the dynamic constraint

$$\dot{x} = u \qquad (3.62)$$

the integral constraint:

$$\int_0^{t_f} \left[\sqrt{1 + \left(\frac{dx}{dt}\right)^2} \right] dt = 4 \qquad (3.63)$$

the boundary conditions

$$\begin{aligned} x(0) &= 1 \\ x(t_f) &= 3 \end{aligned} \qquad (3.64)$$

and the bounds:

$$\begin{aligned} -20 &\le u(t) \le 20 \\ -10 &\le x(t) \le 10 \end{aligned} \qquad (3.65)$$

where $t_f = 1$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
/////////////////////        chain.cxx             //////////////////////
//////////////////////////////////////////////////////////////////////
/////////////////////        PSOPT  Example        //////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:           Hanging chain problem    //////////////////
//////// Last modified:   29 January 2009          //////////////////
//////// Reference:                                //////////////////
////////                                           //////////////////
//////////////////////////////////////////////////////////////////////
```

```
////////          Copyright (c) Victor M. Becerra, 2009          //////////////////
///////////////////////////////////////////////////////////////////////////////
//////////  This is part of the PSOPT software library, which ////////////////
////////////  is distributed under the terms of the GNU Lesser ////////////////
////////////  General Public License (LGPL)                    ////////////////
///////////////////////////////////////////////////////////////////////////////

#include "psopt.h"

///////////////////////////////////////////////////////////////////////////////
///////////////////////  Define the end point (Mayer) cost function //////////////
///////////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return 0.0;
}

///////////////////////////////////////////////////////////////////////////////
///////////////////////  Define the integrand (Lagrange) cost function  //////////
///////////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    adouble x  = states[   CINDEX(1) ];
    adouble dxdt  = controls[ CINDEX(1) ];

    adouble L = x*sqrt(1.0+ pow(dxdt,2.0));

    return  L;
}

///////////////////////////////////////////////////////////////////////////////
///////////////////////  Define the DAE's ////////////////////////////////////////
///////////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
    adouble xdot, ydot, vdot;

    adouble x = states[ CINDEX(1) ];

    adouble dxdt = controls[ CINDEX(1) ];

    derivatives[ CINDEX(1) ] = dxdt;

}


///////////////////////////////////////////////////////////////////////////////
///////////////////////  Define the integrand of the integral constraint ////////
///////////////////////////////////////////////////////////////////////////////

adouble integrand( adouble* states, adouble* controls, adouble* parameters,
                   adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    adouble G;
    adouble dxdt = controls[ CINDEX(1) ];

    G =    sqrt( 1.0 + pow(dxdt,2.0));

    return G;

}


///////////////////////////////////////////////////////////////////////////////
///////////////////////  Define the events function /////////////////////////////
///////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
    adouble x0 = initial_states[ CINDEX(1) ];
    adouble xf = final_states[   CINDEX(1) ];
    adouble Q;
```

```
   // Compute the integral to be constrained
   Q = integrate( integrand, xad, iphase, workspace );

   e[ CINDEX(1) ] = x0;
   e[ CINDEX(2) ] = xf;
   e[ CINDEX(3) ] = Q;

}



//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function //////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}



//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ////////////////////////////
//////////////////////////////////////////////////////////////////////


int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ///////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ///////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name        = "Hanging chain problem";

    problem.outfilename              = "chain.txt";

//////////////////////////////////////////////////////////////////////
////////////   Define problem level constants & do level 1 setup /////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages               = 0;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
//////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates   = 1;
    problem.phases(1).ncontrols = 1;
    problem.phases(1).nevents   = 3;
    problem.phases(1).npath     = 0;
    problem.phases(1).nodes          = "[20, 50]";

    psopt_level2_setup(problem, algorithm);



//////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information //////////////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).bounds.lower.states(1) =  -10.0;
    problem.phases(1).bounds.upper.states(1) =   10.0;


    problem.phases(1).bounds.lower.controls(1) = -20.0;
    problem.phases(1).bounds.upper.controls(1) =  20.0;
```

```
   problem.phases(1).bounds.lower.events(1) =  1.0;
   problem.phases(1).bounds.lower.events(2) =  3.0;
   problem.phases(1).bounds.lower.events(3) =  4.0;


   problem.phases(1).bounds.upper.events(1) = 1.0;
   problem.phases(1).bounds.upper.events(2) = 3.0;
   problem.phases(1).bounds.upper.events(3) = 4.0;



   problem.phases(1).bounds.lower.StartTime    = 0.0;
   problem.phases(1).bounds.upper.StartTime    = 0.0;

   problem.phases(1).bounds.lower.EndTime      = 1.0;
   problem.phases(1).bounds.upper.EndTime      = 1.0;


////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////


   problem.integrand_cost  = &integrand_cost;
   problem.endpoint_cost  = &endpoint_cost;
   problem.dae  = &dae;
   problem.events  = &events;
   problem.linkages = &linkages;




////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess /////////////////////
////////////////////////////////////////////////////////////////////////

   problem.phases(1).guess.controls        = 2.0*ones(1,30);
   problem.phases(1).guess.states          = linspace(1.0,3.0, 30);
   problem.phases(1).guess.time            = linspace(0.0,1.0, 30);

////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ////////////////////////////
////////////////////////////////////////////////////////////////////////


   algorithm.nlp_method              = "IPOPT";
   algorithm.scaling                 = "automatic";
   algorithm.derivatives             = "automatic";
   algorithm.nlp_iter_max            = 1000;
   algorithm.nlp_tolerance           = 1.e-6;



////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   ////////////////
////////////////////////////////////////////////////////////////////////

   psopt(solution, problem, algorithm);

   if (solution.error_flag) exit(0);

////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   ////////
////////////////////////////////////////////////////////////////////////

   DMatrix x, u, t;
   x  = solution.get_states_in_phase(1);
   u  = solution.get_controls_in_phase(1);
   t          = solution.get_time_in_phase(1);

////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired //////////////////////
////////////////////////////////////////////////////////////////////////

   x.Save("x.dat");
   u.Save("u.dat");
   t.Save("t.dat");


////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) /////////////
////////////////////////////////////////////////////////////////////////
```

```
    plot(t,x,problem.name + ": state", "time (s)", "x", "x");

    plot(t,u,problem.name + ": control", "time (s)", "u", "u");

    plot(t,x,problem.name + ": state", "time (s)", "x", "x",
                       "pdf", "chain_state.pdf");

    plot(t,u,problem.name + ": control", "time (s)", "u", "u",
                       "pdf", "chain_control.pdf");



}
////////////////////////////////////////////////////////////////////////
//////////////////////////     END OF FILE     ////////////////////////////
////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the text box below and in Figure 3.35, which illustrates the shape of the hanging chain.

```
PSOPT results summary
=====================


Problem:  Hanging chain problem
CPU time (seconds): 9.700000e-01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  5.068480e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.068480e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 2.393446e-06
NLP solver reports:  The problem solved!
```

3.15   Heat difussion problem

This example can be viewed as a simplified model for the heating of a probe in a kiln [3]. The dynamics are a spatially discretized form of a partial differential equation, which is obtained by using the method of the lines. The problem is formulated on the basis of the state vector $\mathbf{x} = [x_1, \ldots, x_M]^T$ and the control vector $\mathbf{u} = [v_1, v_2, v_3]^T$, as follows

$$\min_{\mathbf{u}(t)} \ J = \frac{1}{2} \int_0^T \left\{ (x_N(t) - x_d(t))^2 + \gamma v_1(t)^2 \right\} \mathrm{d}t$$

189

Figure 3.35: State for hanging chain problem

subject to the differential constraints

$$\dot{x}_1 = \frac{1}{(a_1 + a_2 x_1)} \left[q_1 + \frac{1}{\delta^2}(a_3 + a_4 x_1)(x_2 - 2x_1 + v_2) + a_4 \left(\frac{x_2 - x_1}{2\delta} \right)^2 \right]$$

$$\dot{x}_i = \frac{1}{(a_1 + a_2 x_i)} \left[q_i + \frac{1}{\delta^2}(a_3 + a_4 x_i)(x_{i+1} - 2x_i + x_{i-1}) + a_4 \left(\frac{x_{i+1} - x_{i-1}}{2\delta} \right)^2 \right]$$

for $i = 2, \ldots, M - 1$

$$\dot{x}_M = \frac{1}{(a_1 + a_2 x_M)} \left[q_M + \frac{1}{\delta^2}(a_3 + a_4 x_M)(v_3 - 2x_N + x_{M-1}) + a_4 \left(\frac{v_3 - x_{M-1}}{2\delta} \right)^2 \right]$$

the path constraints

$$0 = g(x_1 - v_1) - \frac{1}{2\delta}(a_3 + a_4 x_1)(x_2 - v_2)$$

$$0 = \frac{1}{2\delta}(a_3 + a + 4x_M)(v_3 - x_{M-1})$$

the control bounds

$$u_L \leq v_1 \leq u_U$$

and the initial conditions for the states:

$$x_i(0) = 2 + \cos(\pi z_i)$$

where

$$z_i = \frac{i-1}{M-1}, \ i = 1, \ldots, M$$

$$x_d(t) = 2 - e^{\rho t}$$

$$q(z,t) = \left[\rho(a_1 + 2a_2) + \pi^2(a_3 + 2a_4)\right] e^{\rho t} \cos(\pi z)$$
$$- a_4 \pi^2 e^{2\pi t} + (2a_4\pi^2 + \rho a_2)e^{2\rho t} \cos^2(\pi z)$$

$$q_i \equiv q(z_i, t), \ i = 1, \ldots, M$$

with the parameter values $a_1 = 4$, $a_2 = 1$, $a_3 = 4$, $a_4 = -1$, $u_U = 0.1$, $\rho = -1$, $T = 0.5$, $\gamma = 10^{-3}$, $g = 1$, $u_L = -\infty$.

A spatial discretization given by $M = 10$ was used. The problem was solved initially by using first 50 nodes, then the mesh was refined to 60 nodes, and an interpolation of the previous solution was employed as an initial guess for the new solution.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
///////////////            heat.cxx                 //////////////////////
//////////////////////////////////////////////////////////////////////
///////////////            PSOPT Template           /////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:  Heat difussion process             ////////////////
//////// Last modified: 09 July 2009                ////////////////
//////// Reference:    Betts (2001)                 ////////////////
////////                                            ////////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009          ////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser  ////////////////
//////// General Public License (LGPL)                      ////////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"


//////////////////////////////////////////////////////////////////////
/////////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////


#define N_DISCRETIZATION  10


adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                       adouble* parameters,adouble& t0, adouble& tf,
                       adouble* xad, int iphase, Workspace* workspace)
{
   return 0.0;
}

//////////////////////////////////////////////////////////////////////
/////////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{

    int N = N_DISCRETIZATION;

    double gamma  = 1.0e-3;

    double rho  = -1.0;

    adouble yd = 2.0-exp(rho*time);
```

```
    adouble yN = states[ CINDEX(N) ];

    adouble v1 = controls[ CINDEX(1) ];

    return   0.5*( pow(yN-yd,2.0) + gamma*pow(v1,2.0) );
}


////////////////////////////////////////////////////////////////////
/////////////////////  Define the DAE's /////////////////////////////
////////////////////////////////////////////////////////////////////


void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

   adouble u;

   double a1  = 4.0;
   double a2  = 1.0;
   double a3  = 4.0;
   double a4  = -1.0;
   double rho = -1.0;
   double T   = 0.5;
   double g   = 1.0;

   int N  = N_DISCRETIZATION;

   int i;

   double delta   = 1.0/(N-1);


   adouble* y  = states;

   adouble v1  = controls[ CINDEX(1) ];
   adouble v2  = controls[ CINDEX(2) ];
   adouble v3  = controls[ CINDEX(3) ];

   adouble y1  = y[CINDEX(1)];
   adouble y2  = y[CINDEX(2)];
   adouble yN  = y[CINDEX(N)];
   adouble yNm1 = y[CINDEX(N-1)];

   adouble x1  =   0;
   adouble xN  =   1;

   adouble q1 =  (rho*(a1+2*a2) + pi*pi*(a3+2*a4))*exp(rho*time)*cos(pi*x1)
                     - a4*pi*pi*exp(2*rho*time) + (2*a4*pi*pi+rho*a2)*exp(2*rho*time)*pow(cos(pi*x1),2.0);
   adouble qN =  (rho*(a1+2*a2) + pi*pi*(a3+2*a4))*exp(rho*time)*cos(pi*xN)
                     - a4*pi*pi*exp(2*rho*time) + (2*a4*pi*pi+rho*a2)*exp(2*rho*time)*pow(cos(pi*xN),2.0);

   derivatives[ CINDEX(1) ] = 1.0/(a1+a2*y1)*(q1 +
                                 (1.0/pow(delta,2.0))*(a3+a4*y1)*(y2-2*y1+v2)+a4*pow( (y2-v2)/(2*delta), 2.0) );

   for(i=2;i<=(N-1);i++) {
       adouble yi   = y[CINDEX(i)];
       adouble yim1 = y[CINDEX(i-1)];
       adouble yip1 = y[CINDEX(i+1)];
       adouble xi =   ( (double) (i-1) ) /( (double) (N-1) );
       adouble qi = (rho*(a1+2*a2) + pi*pi*(a3+2*a4))*exp(rho*time)*cos(pi*xi)
                     - a4*pi*pi*exp(2*rho*time) + (2*a4*pi*pi+rho*a2)*exp(2*rho*time)*pow(cos(pi*xi),2.0);

       derivatives[CINDEX(i)]=1.0/(a1+a2*yi)*(qi + (1.0/pow(delta,2.0))*
                                 (a3+a4*yi)*(yip1-2*yi+yim1)+a4*pow( (yip1-yim1)/(2*delta), 2.0) );
   }

   derivatives[ CINDEX(N) ] = 1.0/(a1+a2*yN)*(qN + (1.0/pow(delta,2.0))*
                                 (a3+a4*yN)*(v3-2*yN+yNm1)+a4*pow( (v3-yNm1)/(2*delta), 2.0) );

   path[CINDEX(1)] =  g*(y1-v1) - (1.0/(2*delta))*(a3+a4*y1)*(y2-v2);

   path[CINDEX(2)] = (1.0/(2*delta))*(a3+a4*yN)*(v3-yNm1);

}

////////////////////////////////////////////////////////////////////
/////////////////////  Define the events function ///////////////////
////////////////////////////////////////////////////////////////////
```

```
void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{

   int i;

   int N = N_DISCRETIZATION;

   for(i=1;i<= N; i++) {
         e[CINDEX(i)] = initial_states[CINDEX(i)];
   }

}



//////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function //////////////////
//////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{

// Single phase problem

}



//////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine //////////////////////////////
//////////////////////////////////////////////////////////////////////////

int main(void)
{

   int N = N_DISCRETIZATION;

//////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures //////////////////////////////
//////////////////////////////////////////////////////////////////////////

   Alg  algorithm;
   Sol  solution;
   Prob problem;

//////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  //////////////////////////////
//////////////////////////////////////////////////////////////////////////

   problem.name        = "Heat diffusion process";
   problem.outfilename              = "heat.txt";

//////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////////

   problem.nphases    = 1;
   problem.nlinkages               = 0;

   psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
//////////////////////////////////////////////////////////////////////////

   problem.phases(1).nstates   = N;
   problem.phases(1).ncontrols = 3;
   problem.phases(1).nevents   = N;
   problem.phases(1).npath     = 2;
   problem.phases(1).nodes                  = "[20 50 60]";


   psopt_level2_setup(problem, algorithm);



//////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ////////////////////
//////////////////////////////////////////////////////////////////////////
```

```
    int i, j;


    problem.phases(1).bounds.lower.states = zeros(N,1);
    problem.phases(1).bounds.upper.states = 3.0*ones(N,1);


    problem.phases(1).bounds.lower.controls(1) = 0.0;
    problem.phases(1).bounds.lower.controls(2) = 0.0;
    problem.phases(1).bounds.lower.controls(3) = 0.0;


    problem.phases(1).bounds.upper.controls(1) = 0.1;
    problem.phases(1).bounds.upper.controls(2) = 3.0;
    problem.phases(1).bounds.upper.controls(3) = 3.0;

    for(i = 1; i<= N; i++ ) {
        double xi =    ( (double) (i-1) ) /( (double) (N-1) );
     double yI = 2.0+cos(pi*xi);

     problem.phases(1).bounds.lower.events(i) = yI;
     problem.phases(1).bounds.upper.events(i) = yI;

    }


    problem.phases(1).bounds.upper.path(1) = 0.0;
    problem.phases(1).bounds.upper.path(2) = 0.0;
    problem.phases(1).bounds.lower.path(1) = 0.0;
    problem.phases(1).bounds.lower.path(2) = 0.0;



    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 0.5;
    problem.phases(1).bounds.upper.EndTime      = 0.5;


////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae              = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess /////////////////////
////////////////////////////////////////////////////////////////////////

    int nnodes      = problem.phases(1).nodes(1);
    int ncontrols                         = problem.phases(1).ncontrols;
    int nstates                           = problem.phases(1).nstates;

    DMatrix state_guess    = zeros(nstates,nnodes);
    DMatrix control_guess  = zeros(ncontrols,nnodes);
    DMatrix param_guess    = zeros(0,0);
    DMatrix time_guess     = linspace(0,0.5,nnodes);

    for(i = 1; i<= N; i++ ) {
        double xi =    ( (double) (i-1) ) /( (double) (N-1) );
     double yI = 2.0+cos(pi*xi);
     state_guess(i,colon()) = yI*ones(1,nnodes);
    }

    control_guess(1,colon()) = linspace(0.1,0.1, nnodes);
    control_guess(2,colon()) = state_guess(1,colon());
    control_guess(3,colon()) = state_guess(N,colon());


    auto_phase_guess(problem, control_guess, state_guess, param_guess, time_guess);


////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ////////////////////////////
////////////////////////////////////////////////////////////////////////
```

```
    algorithm.nlp_iter_max            = 1000;
    algorithm.nlp_tolerance           = 1.e-6;
    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
    algorithm.defect_scaling          = "jacobian-based";

////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////


    DMatrix y, u, t;

    y       = solution.get_states_in_phase(1);
    u       = solution.get_controls_in_phase(1);
    t       = solution.get_time_in_phase(1);


////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////

    y.Save("y.dat");
    u.Save("u.dat");
    t.Save("t.dat");

    DMatrix x(N);

    for(i = 1; i<= N; i++ ) {
        x(i) =    ( (double) (i-1) ) /( (double) (N-1) );
    }

////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////


    plot(t,u(1,colon()),problem.name+": control","time (s)", "control", "u");

    plot(t,u(1,colon()),problem.name+": control","time (s)", "control", "v1",
"pdf", "heat_control.pdf");

    plot(t,y(1,colon()),problem.name+": state","time (s)", "state", "x1");

    plot(t,y(1,colon()),problem.name+": state","time (s)", "state", "x1",
"pdf", "heat_state1.pdf");

    plot(t,y(N,colon()),problem.name+": state","time (s)", "state", "xN");

    plot(t,y(N,colon()),problem.name+": state","time (s)", "state", "xN",
"pdf", "heat_stateN.pdf");

    surf(x, t,  y, problem.name, "x", "t", "h");

    surf(x, t,  y, problem.name, "z", "t", "x", "pdf", "heat_surf.pdf");

}
////////////////////////////////////////////////////////////////////////
/////////////////////////        END OF FILE     ////////////////////////////
////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized the box below. Figure 3.36 shows the control variable v_1 as a function of time. Figure 3.37 shows the resulting temperature distribution.

PSOPT results summary

```
=====================

Problem:  Heat diffusion process
CPU time (seconds): 1.542000e+01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  4.370350e-05
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  4.370350e-05
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.000000e-01
Phase 1 maximum relative local error: 5.418554e-07
NLP solver reports:  The problem solved!
```


Figure 3.36: Optimal control distribution for the heat diffusion process

3.16 Hypersensitive problem

Consider the following optimal control problem, which is known in the literature as the hypesensitive optimal control problem [35]. Minimize the cost

Heat diffusion process

Figure 3.37: Optimal temperature distribution for the heat diffusion process

functional

$$J = \frac{1}{2} \int_0^{t_f} [x^2 + u^2] dt \tag{3.66}$$

subject to the dynamic constraint

$$\dot{x} = -x^3 + u \tag{3.67}$$

and the boundary conditions

$$\begin{array}{rcl} x(0) & = & 1.5 \\ x(t_f) & = & 1 \end{array} \tag{3.68}$$

where $t_f = 50$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
/////////////////        hypersensitive.cxx    //////////////////////
//////////////////////////////////////////////////////////////////////
/////////////////        PSOPT  Example        //////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:        Hypersensitive problem     //////////////////
//////// Last modified: 05 January 2009            //////////////////
//////// Reference:     Rao and Mease (2000)       //////////////////
//////// (See PSOPT handbook for full reference)   //////////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009     //////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////
```

```
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
/////////////////////////////////////////////////////////////////////////

#include "psopt.h"

/////////////////////////////////////////////////////////////////////////
//////////////////// Define the end point (Mayer) cost function //////////
/////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   return 0.0;
}

/////////////////////////////////////////////////////////////////////////
//////////////////// Define the integrand (Lagrange) cost function //////
/////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
     adouble x = states[0];
     adouble u = controls[0];

     adouble L = 0.5*(  x*x + u*u  );

     return L;
}


/////////////////////////////////////////////////////////////////////////
//////////////////// Define the DAE's ////////////////////////////////////
/////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
     adouble x = states[0];
     adouble u = controls[0];

     adouble xdot = -pow(x,3) + u;

     derivatives[0] = xdot;
}


/////////////////////////////////////////////////////////////////////////
//////////////////// Define the events function //////////////////////////
/////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
     adouble xi = initial_states[0];
     adouble xf = final_states[0];

     e[0] = xi;
     e[1] = xf;

}

/////////////////////////////////////////////////////////////////////////
//////////////////// Define the phase linkages function //////////////////
/////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


/////////////////////////////////////////////////////////////////////////
//////////////////// Define the main routine /////////////////////////////
/////////////////////////////////////////////////////////////////////////

int main(void)
{
```

```
/////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures /////////////////////////////
/////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

/////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  /////////////////////////////
/////////////////////////////////////////////////////////////////////

    problem.name          = "Hypersensitive problem";
    problem.outfilename                = "hyper.txt";

/////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
/////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                = 0;

    psopt_level1_setup(problem);


/////////////////////////////////////////////////////////////////////
/////////   Define phase related information &  do level 2 setup ////////////
/////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates    = 1;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 2;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes                = "[25, 50]";


    psopt_level2_setup(problem, algorithm);

/////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information /////////////////////
/////////////////////////////////////////////////////////////////////

    double xi = 1.5;
    double xf = 1.0;

    problem.phases(1).bounds.lower.states(1) = -50.0;
    problem.phases(1).bounds.upper.states(1) = 50.0;


    problem.phases(1).bounds.lower.controls(1) = -50.0;
    problem.phases(1).bounds.upper.controls(1) =  50.0;


    problem.phases(1).bounds.lower.events(1) = xi;
    problem.phases(1).bounds.upper.events(1) = xi;

    problem.phases(1).bounds.lower.events(2) = xf;
    problem.phases(1).bounds.upper.events(2) = xf;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 50.0;
    problem.phases(1).bounds.upper.EndTime      = 50.0;

/////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
/////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;



/////////////////////////////////////////////////////////////////////
```

```
//////////////////  Define & register initial guess /////////////////////////
/////////////////////////////////////////////////////////////////////////////


    problem.phases(1).guess.controls      = zeros(1,60);
    problem.phases(1).guess.states        = linspace(1,1,60);
    problem.phases(1).guess.time          = linspace(0.0,50.0,60);




/////////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  /////////////////////////////////
/////////////////////////////////////////////////////////////////////////////

    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;


//    algorithm.ps_method  = "none";
//    algorithm.refinement_strategy = "TH";
//    algorithm.mr_tolerance = 1.e-8;
//    algorithm.mr_max_iterations = 8;




/////////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   /////////////////////
/////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

/////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   ////////////
/////////////////////////////////////////////////////////////////////////////


    DMatrix x      = solution.get_states_in_phase(1);
    DMatrix u      = solution.get_controls_in_phase(1);
    DMatrix t      = solution.get_time_in_phase(1);

/////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ///////////////////////////
/////////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

/////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) //////////////////
/////////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name + ": state", "time (s)", "state", "x");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u");

    plot(t,x,problem.name + ": state", "time (s)", "state", "x", "pdf", "hyper_state.pdf");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u", "pdf", "hyper_control.pdf");


}

/////////////////////////////////////////////////////////////////////////////
/////////////////////////      END OF FILE     ////////////////////////////////
/////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized the box below and shown in the following plots that contain the elements of the state and the control, respectively.

200

Figure 3.38: State for hypersensitive problem

```
PSOPT results summary
=====================


Problem:  Hypersensitive problem
CPU time (seconds): 6.900000e-01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  1.330826e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  1.330826e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.000000e+01
Phase 1 maximum relative local error: 5.728419e-04
NLP solver reports:  The problem solved!
```

3.17   Interior point constraint problem

Consider the following optimal control problem, which involves a scalar system with an interior point constraint on the state [25]. Minimize the cost functional

$$J = \int_0^1 [x^2 + u^2] dt \qquad (3.69)$$

201

Figure 3.39: Control for hypersensitive problem

subject to the dynamic constraint

$$\dot{x} = u, \tag{3.70}$$

the boundary conditions

$$\begin{array}{rcl} x(0) & = & 1, \\ x(1) & = & 0.75, \end{array} \tag{3.71}$$

and the interior point constraint:

$$x(0.75) = 0.9. \tag{3.72}$$

The problem is divided into two phases and the interior point constraint is accommodated as an event constraint at the end of the first phase. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
//////////////////         interior_point.cxx     //////////////////////
//////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example         //////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title: Problem with interior point constraint   ///////////////
//////// Last modified: 05 January 2009                  ///////////////
//////// Reference:    Miser Manual                      ///////////////
//////// (See PSOPT handbook for full reference)         ///////////////
//////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009       ///////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser  ///////////////
```

```
//////// General Public License (LGPL)                        //////////////
/////////////////////////////////////////////////////////////////////

#include "psopt.h"

/////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
/////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   return 0.0;
}

/////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  ////////
/////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
     adouble x = states[0];
     adouble u = controls[0];

     adouble L =   x*x + u*u;

     return L;
}


/////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's /////////////////////////////////
/////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
    adouble x = states[0];
    adouble u = controls[0];

    adouble xdot =  u;

    derivatives[0] = xdot;
}


/////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ///////////////////////////
/////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
    adouble xi = initial_states[0];
    adouble xf = final_states[0];
    adouble x075;

    if (iphase == 1) {
        xi = initial_states[0];
        x075 = final_states[0];
        e[0] = xi;
        e[1] = x075;
    }


    if (iphase == 2) {
        xf = final_states[0];
        e[0] = xf;
    }

}

/////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
/////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
   int index = 0;
```

```
   auto_link(linkages, &index, xad, 1, 2, workspace );

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ////////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name         = "Problem with interior point constraint";
    problem.outfilename       = "ipc.txt";

//////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 2;
    problem.nlinkages               = 2;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////
/////////   Define phase related information &  do level 2 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 1;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 2;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes         = 20;

    problem.phases(2).nstates    = 1;
    problem.phases(2).ncontrols  = 1;
    problem.phases(2).nevents    = 1;
    problem.phases(2).npath      = 0;
    problem.phases(2).nodes         = 20;

    psopt_level2_setup(problem, algorithm);


//////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ////////////////////
//////////////////////////////////////////////////////////////////////

    double xi   = 1;
    double x075 = 0.9;
    double xf   = 0.75;

    /////////////  Phase 1 bounds /////////////////////

    problem.phases(1).bounds.lower.states(1) = -1.0;
    problem.phases(1).bounds.upper.states(1) =  1.0;


    problem.phases(1).bounds.lower.controls(1) = -1.0;
    problem.phases(1).bounds.upper.controls(1) =  1.0;


    problem.phases(1).bounds.lower.events(1) = xi;
    problem.phases(1).bounds.upper.events(1) = xi;

    problem.phases(1).bounds.lower.events(2) = x075;
    problem.phases(1).bounds.upper.events(2) = x075;

    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;
```

```
    problem.phases(1).bounds.lower.EndTime      = 0.75;
    problem.phases(1).bounds.upper.EndTime      = 0.75;

    /////////////// Phase 2 bounds //////////////////

    problem.phases(2).bounds.lower.states(1) = -1.0;
    problem.phases(2).bounds.upper.states(1) =  1.0;

    problem.phases(2).bounds.lower.controls(1) = -1.0;
    problem.phases(2).bounds.upper.controls(1) =  1.0;


    problem.phases(2).bounds.lower.events(1) = xf;
    problem.phases(2).bounds.upper.events(1) = xf;


    problem.phases(2).bounds.lower.StartTime    = 0.75;
    problem.phases(2).bounds.upper.StartTime    = 0.75;

    problem.phases(2).bounds.lower.EndTime      = 1.0;
    problem.phases(2).bounds.upper.EndTime      = 1.0;

////////////////////////////////////////////////////////////////////
////////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;



////////////////////////////////////////////////////////////////////
////////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////


    /////////////// Phase 1 initial guess ///////////////////////

    problem.phases(1).guess.controls        = zeros(1,20);
    problem.phases(1).guess.states          = linspace(xi,x075, 20);
    problem.phases(1).guess.time            = linspace(0.0, 0.75 , 20);

    /////////////// Phase 2 initial guess ///////////////////////


    problem.phases(2).guess.controls        = zeros(1,20);
    problem.phases(2).guess.states          = linspace(x075,xf, 20);
    problem.phases(2).guess.time            = linspace(0.75, 1.0 , 20);


////////////////////////////////////////////////////////////////////
////////////////////  Enter algorithm options ///////////////////////////
////////////////////////////////////////////////////////////////////

    algorithm.nlp_method                = "IPOPT";
    algorithm.scaling                   = "automatic";
    algorithm.derivatives               = "automatic";
    algorithm.nlp_iter_max              = 1000;
    algorithm.nlp_tolerance             = 1.e-6;

////////////////////////////////////////////////////////////////////
////////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////
//////////   Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////

    DMatrix x, u, t;


    x=solution.get_states_in_phase(1)  || solution.get_states_in_phase(2);
    u=solution.get_controls_in_phase(1)|| solution.get_controls_in_phase(2);
    t=solution.get_time_in_phase(1)   || solution.get_time_in_phase(2);
```

```
//////////////////////////////////////////////////////////////////////
//////////  Save solution data to files if desired ////////////////////
//////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

//////////////////////////////////////////////////////////////////////
//////////  Plot some results if desired (requires gnuplot) ///////////
//////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name + ": state", "time (s)", "state", "x");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u");

    plot(t,x,problem.name + ": state", "time (s)", "state", "x",
                        "pdf", "ipc_state.pdf");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u", "pdf",
                        "ipc_control.pdf");

}

//////////////////////////////////////////////////////////////////////
/////////////////////////     END OF FILE     /////////////////////////
//////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized the box below and shown in the following plots that contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================


Problem:   Problem with interior point constraint
CPU time (seconds): 1.400000e-01
NLP solver used:   IPOPT
Optimal (unscaled) cost function value:   9.205314e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:   6.607877e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 7.500000e-01
Phase 1 maximum relative local error: 3.477757e-08
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:   2.597438e-01
Phase 2 initial time: 7.500000e-01
Phase 2 final time: 1.000000e+00
Phase 2 maximum relative local error: 1.882112e-08
NLP solver reports:   The problem solved!
```

206

Figure 3.40: State for interior point constraint problem

Figure 3.41: Control for interior point constraint problem

3.18 Isoperimetric constraint problem

Consider the following optimal control problem, which includes an integral constraint. Minimize the cost functional

$$J = \int_0^{t_f} x^2(t)dt \qquad (3.73)$$

subject to the dynamic constraint

$$\dot{x} = -\sin(x) + u \qquad (3.74)$$

the integral constraint:

$$\int_0^{t_f} u^2(t)dt = 10 \qquad (3.75)$$

the boundary conditions

$$\begin{aligned} x(0) &= 1 \\ x(t_f) &= 0 \end{aligned} \qquad (3.76)$$

and the bounds:

$$\begin{aligned} -4 \le u(t) \le 4 \\ -10 \le x(t) \le 10 \end{aligned} \qquad (3.77)$$

where $t_f = 1$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
/////////////////////        isoperimetric.cxx        //////////////////////
//////////////////////////////////////////////////////////////////////
///////////////////           PSOPT  Example          ////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////////  Title:               Isoperimetric problem    ////////////////
//////////  Last modified:        29 January 2009         ////////////////
//////////  Reference:                                    ////////////////
//////////                                                ////////////////
//////////////////////////////////////////////////////////////////////
//////////      Copyright (c) Victor M. Becerra, 2009     ////////////////
//////////////////////////////////////////////////////////////////////
//////////  This is part of the PSOPT software library, which ////////////////
//////////  is distributed under the terms of the GNU Lesser ////////////////
//////////  General Public License (LGPL)                 ////////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
////////////////////  Define the end point (Mayer) cost function ////////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                  adouble* parameters,adouble& t0, adouble& tf,
                  adouble* xad, int iphase, Workspace* workspace)
{
    return 0.0;
}

//////////////////////////////////////////////////////////////////////
////////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                  adouble* parameters, adouble& time, adouble* xad,
                  int iphase, Workspace* workspace)
{
```

```
    adouble x = states[0];

    adouble L = x;

    return  L;
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble xdot, ydot, vdot;

   adouble x = states[ CINDEX(1) ];

   adouble u = controls[ CINDEX(1) ];

   derivatives[0] = -sin(x) + u;

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand of the integral constraint ///////
//////////////////////////////////////////////////////////////////////

adouble integrand( adouble* states, adouble* controls, adouble* parameters,
                   adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
   adouble g;
   adouble u = controls[ CINDEX(1) ];

   g =  u*u ;

   return g;

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function /////////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x0 = initial_states[ CINDEX(1) ];
   adouble xf = final_states[   CINDEX(1) ];
   adouble Q;

   // Compute the integral to be constrained
   Q = integrate( integrand, xad, iphase, workspace );

   e[ CINDEX(1) ] = x0;
   e[ CINDEX(2) ] = xf;
   e[ CINDEX(3) ] = Q;

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////////
//////////////////////////////////////////////////////////////////////


int main(void)
{
```

```
//////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures /////////////////////////////
//////////////////////////////////////////////////////////////////////

   Alg  algorithm;
   Sol  solution;
   Prob problem;

//////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  /////////////////////////////
//////////////////////////////////////////////////////////////////////

   problem.name          = "Isoperimetric constraint problem";

   problem.outfilename              = "isoperimetric.txt";

//////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

   problem.nphases     = 1;
   problem.nlinkages                = 0;

   psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////
/////////  Define phase related information & do level 2 setup /////////////
//////////////////////////////////////////////////////////////////////


   problem.phases(1).nstates    = 1;
   problem.phases(1).ncontrols  = 1;
   problem.phases(1).nevents    = 3;
   problem.phases(1).npath      = 0;
   problem.phases(1).nodes = 50;

   psopt_level2_setup(problem, algorithm);



//////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information /////////////////////
//////////////////////////////////////////////////////////////////////

   problem.phases(1).bounds.lower.states(1) = -10;
   problem.phases(1).bounds.upper.states(1) = 10;

   problem.phases(1).bounds.lower.controls(1) = -4.0;
   problem.phases(1).bounds.upper.controls(1) =  4.0;

   problem.phases(1).bounds.lower.events(1) =  1.0;
   problem.phases(1).bounds.lower.events(2) =  0.0;
   problem.phases(1).bounds.lower.events(3) =  10.0;

   problem.phases(1).bounds.upper.events(1) = 1.0;
   problem.phases(1).bounds.upper.events(2) = 0.0;
   problem.phases(1).bounds.upper.events(3) = 10.0;

   problem.phases(1).bounds.lower.StartTime    = 0.0;
   problem.phases(1).bounds.upper.StartTime    = 0.0;

   problem.phases(1).bounds.lower.EndTime      = 1.0;
   problem.phases(1).bounds.upper.EndTime      = 1.0;


//////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  /////////////////////////
//////////////////////////////////////////////////////////////////////


   problem.integrand_cost  = &integrand_cost;
   problem.endpoint_cost  = &endpoint_cost;
   problem.dae  = &dae;
   problem.events  = &events;
   problem.linkages = &linkages;



//////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ////////////////////
```

```
///////////////////////////////////////////////////////////////////////

    problem.phases(1).guess.controls      = zeros(1,30);
    problem.phases(1).guess.states        = zeros(1,30);
    problem.phases(1).guess.time          = linspace(0.0,1.0,30);

///////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ////////////////////////////
///////////////////////////////////////////////////////////////////////


    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
    algorithm.nlp_iter_max            = 1000;
    algorithm.nlp_tolerance           = 1.e-6;

///////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   ///////////////
///////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

    if (solution.error_flag) exit(0);

///////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   ///////////
///////////////////////////////////////////////////////////////////////

    DMatrix x, u, t;
    x  = solution.get_states_in_phase(1);
    u  = solution.get_controls_in_phase(1);
    t          = solution.get_time_in_phase(1);

///////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
///////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");


///////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ////////////////
///////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name + ": state", "time (s)", "x", "x");

    plot(t,u,problem.name + ": control", "time (s)", "u", "u");

    plot(t,x,problem.name + ": state", "time (s)", "x", "x",
                         "pdf", "isop_state.pdf");

    plot(t,u,problem.name + ": control", "time (s)", "u", "u",
                         "pdf", "isop_control.pdf");


}

///////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ////////////////////////////
///////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the text box below and in Figures 3.42 and 3.43, which show the optimal state and control, respectively.

```
PSOPT results summary
=====================
```

Figure 3.42: State for isoperimetric constraint problem

```
Problem:  Isoperimetric constraint problem
CPU time (seconds): 1.333198e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  -3.755058e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  -3.755058e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 3.106347e-05
NLP solver reports:  The problem solved!
```

3.19 Lambert's problem

This example demonstrates the use of the $\mathcal{PSOPT}$ for a classical orbit determination problem, namely the determination of an orbit from two position vectors and time (Lambert's problem) [43]. The problem is formulated as follows. Find $\mathbf{r}(t) \in [0, t_f]$ and $\mathbf{v}(t) \in [0, t_f]$ to minimise:

$$J = 0 \tag{3.78}$$

Figure 3.43: Control for isoperimetric constraint problem

subject to

$$\dot{\mathbf{r}} = \mathbf{v}$$
$$\dot{\mathbf{v}} = -\mu \frac{\mathbf{r}}{||\mathbf{r}||^3} \tag{3.79}$$

with the boundary conditions:

$$\mathbf{r(0)} = [15945.34\text{E}3, 0.0, 0.0]^T$$
$$\mathbf{r}(t_f) = [12214.83899\text{E}3, 10249.46731\text{E}3, 0.0]^T \tag{3.80}$$

where $\mathbf{r} = [x, y, z]^T$ (m) is a cartesian position vector, and $\mathbf{v} = [v_x, v_z, v_z]^T$ is the corresponding velocity vector, $\mu = GM_e$, G (m^3/(kg s^2)) is the universal gravitational constant and M_e (kg) is the mass of Earth.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
///////////////          lambert.cxx                ////////////////////
//////////////////////////////////////////////////////////////////////
///////////////           PSOPT  Example            ////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title: Lambert problem                          ////////////////
//////// Last modified: 27 December 2009                 ////////////////
//////// Reference:     Vallado (2001), page 470         ////////////////
//////// (See PSOPT handbook for full reference)         ////////////////
//////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009       ////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                   ////////////////
//////////////////////////////////////////////////////////////////////
```

```
#include "psopt.h"


//////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   return 0.0;
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    return  0.0;
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's //////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

   // Define constants:

   double G =   6.6720e-11; // Universal gravity constant [m^3/(kg s^2)]
   double Me = 5.9742e24;   // Mass of earth;[kg]

   double mu   = G*Me;        // [m^3/sec^2]

   ADMatrix r(3), v(3);

   // Extract individual variables

   r(1) = states[ CINDEX(1) ];
   r(2) = states[ CINDEX(2) ];
   r(3) = states[ CINDEX(3) ];

   v(1) = states[ CINDEX(4) ];
   v(2) = states[ CINDEX(5) ];
   v(3) = states[ CINDEX(6) ];


   ADMatrix rdd(3);

   adouble rr = sqrt( r(1)*r(1)+r(2)*r(2)+r(3)*r(3) );

   adouble r3 = pow(rr,3.0);

   rdd(1) = -mu*r(1)/r3;
   rdd(2) = -mu*r(2)/r3;
   rdd(3) = -mu*r(3)/r3;

   derivatives[ CINDEX(1) ] = v(1);
   derivatives[ CINDEX(2) ] = v(2);
   derivatives[ CINDEX(3) ] = v(3);
   derivatives[ CINDEX(4) ] = rdd(1);
   derivatives[ CINDEX(5) ] = rdd(2);
   derivatives[ CINDEX(6) ] = rdd(3);

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{
```

```
   adouble ri1 = initial_states[ CINDEX(1) ];
   adouble ri2 = initial_states[ CINDEX(2) ];
   adouble ri3 = initial_states[ CINDEX(3) ];



   adouble rf1 = final_states[ CINDEX(1) ];
   adouble rf2 = final_states[ CINDEX(2) ];
   adouble rf3 = final_states[ CINDEX(3) ];

   e[ CINDEX(1) ] = ri1;
   e[ CINDEX(2) ] = ri2;
   e[ CINDEX(3) ] = ri3;

   e[ CINDEX(4) ] = rf1;
   e[ CINDEX(5) ] = rf2;
   e[ CINDEX(6) ] = rf3;

}



//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function //////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
 //  auto_link_multiple(linkages, xad, N_PHASES);
}



//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine /////////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures /////////////////////////////
//////////////////////////////////////////////////////////////////////

   Alg  algorithm;
   Sol  solution;
   Prob problem;

   MSdata msdata;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  /////////////////////////////
//////////////////////////////////////////////////////////////////////

   problem.name         = "Lambert problem";
   problem.outfilename              = "lambert.txt";

//////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

   problem.nphases    = 1;
   problem.nlinkages              = 0;

   psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////
/////////   Define phase related information &  do level 2 setup ////////////
//////////////////////////////////////////////////////////////////////

   problem.phases(1).nstates   = 6;
   problem.phases(1).ncontrols = 0;
   problem.phases(1).nevents   = 6;
   problem.phases(1).nparameters            = 6;
   problem.phases(1).npath     = 0;
   problem.phases(1).nodes              = "[100]";

   psopt_level2_setup(problem, algorithm);


//////////////////////////////////////////////////////////////////////
```

```
//////////////////  Enter problem bounds information ////////////////////////
///////////////////////////////////////////////////////////////////////////

    double r1i = 15945.34e3; // m
    double r2i = 0.0;
    double r3i = 0.0;

    double r1f = 12214.83899e3; //m
    double r2f = 10249.46731e3; //m
    double r3f = 0.0;

    double TF = 76.0*60.0; // seconds

    problem.phases(1).bounds.lower.states(1) = -10*max(r1i,r1f);
    problem.phases(1).bounds.lower.states(2) = -10*max(r2i,r2f);
    problem.phases(1).bounds.lower.states(3) = -10*max(r3i,r3f);
    problem.phases(1).bounds.upper.states(1) = 10*max(r1i,r1f);
    problem.phases(1).bounds.upper.states(2) = 10*max(r2i,r2f);
    problem.phases(1).bounds.upper.states(3) = 10*max(r3i,r3f);

    problem.phases(1).bounds.lower.states(4) = -10*max(r1i,r1f)/TF;
    problem.phases(1).bounds.lower.states(5) = -10*max(r1i,r1f)/TF;;
    problem.phases(1).bounds.lower.states(6) = -10*max(r1i,r1f)/TF;;
    problem.phases(1).bounds.upper.states(4) = 10*max(r1i,r1f)/TF;
    problem.phases(1).bounds.upper.states(5) = 10*max(r2i,r2f)/TF;
    problem.phases(1).bounds.upper.states(6) = 10*max(r3i,r3f)/TF;


    problem.phases(1).bounds.lower.events(1) = r1i;
    problem.phases(1).bounds.upper.events(1) = r1i;

    problem.phases(1).bounds.lower.events(2) = r2i;
    problem.phases(1).bounds.upper.events(2) = r2i;

    problem.phases(1).bounds.lower.events(3) = r3i;
    problem.phases(1).bounds.upper.events(3) = r3i;

    problem.phases(1).bounds.lower.events(4) = r1f;
    problem.phases(1).bounds.upper.events(4) = r1f;

    problem.phases(1).bounds.lower.events(5) = r2f;
    problem.phases(1).bounds.upper.events(5) = r2f;

    problem.phases(1).bounds.lower.events(6) = r3f;
    problem.phases(1).bounds.upper.events(6) = r3f;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = TF;
    problem.phases(1).bounds.upper.EndTime      = TF;


///////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem names and units    ////////////////////////
///////////////////////////////////////////////////////////////////////////


    problem.phases(1).name.states(1) = "x position";
    problem.phases(1).name.states(2) = "y position";
    problem.phases(1).name.states(3) = "z position";
    problem.phases(1).name.states(4) = "x velocity";
    problem.phases(1).name.states(5) = "y velocity";
    problem.phases(1).name.states(6) = "z velocity";

    problem.phases(1).units.states(1) = "m";
    problem.phases(1).units.states(2) = "m";
    problem.phases(1).units.states(3) = "m";
    problem.phases(1).units.states(4) = "m";
    problem.phases(1).units.states(5) = "m/s";
    problem.phases(1).units.states(6) = "m/s";




///////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  ////////////////////////////
///////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost   = &endpoint_cost;
```

```
    problem.dae          = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ////////////////////
////////////////////////////////////////////////////////////////////

    int nnodes     = 20;
    int ncontrols                  = problem.phases(1).ncontrols;
    int nstates                    = problem.phases(1).nstates;


    DMatrix x_guess    = zeros(nstates,nnodes);
    DMatrix time_guess = linspace(0.0,TF,nnodes);


    x_guess(1,colon()) = linspace(r1i,r1f, nnodes);
    x_guess(2,colon()) = linspace(r2i,r2f,nnodes);
    x_guess(3,colon()) = linspace(r3i,r3f,nnodes);
    x_guess(4,colon()) = linspace(r1i,r1f,nnodes)/TF;
    x_guess(5,colon()) = linspace(r2i,r2f,nnodes)/TF;
    x_guess(6,colon()) = linspace(r3i,r3f, nnodes)/TF;


    problem.phases(1).guess.states       = x_guess;
    problem.phases(1).guess.time         = time_guess;


////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ////////////////////////
////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max             = 1000;
    algorithm.nlp_tolerance            = 1.e-6;
    algorithm.nlp_method               = "IPOPT";
    algorithm.scaling                  = "automatic";
    algorithm.derivatives              = "automatic";
    algorithm.defect_scaling           = "jacobian-based";
    algorithm.collocation_method       = "Hermite-Simpson";




////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////

    DMatrix x, u, t, xi, ui, ti;

    x      = solution.get_states_in_phase(1);
    u      = solution.get_controls_in_phase(1);
    t      = solution.get_time_in_phase(1);


////////////////////////////////////////////////////////////////////
///////////   Save solution data to files if desired ////////////////////
////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

////////////////////////////////////////////////////////////////////
///////////   Plot some results if desired (requires gnuplot) //////////////
////////////////////////////////////////////////////////////////////

    DMatrix r1 = x(1,colon());
    DMatrix r2 = x(2,colon());
    DMatrix r3 = x(3,colon());

    DMatrix v1 = x(4,colon());
    DMatrix v2 = x(5,colon());
```

```
    DMatrix v3 = x(6,colon());

    DMatrix vi(3), vf(3);

    vi(1) = v1(1);
    vi(2) = v2(1);
    vi(3) = v3(1);

    vf(1) = v1("end");
    vf(2) = v2("end");
    vf(3) = v3("end");

    vi.Print("Initial velocity vector [m/s]");

    vf.Print("Final velocity vector [m/s]");


    plot(t,r1,problem.name+": x", "time (s)", "x","x");

    plot(t,r2,problem.name+": y", "time (s)", "x","y");

    plot(t,r3,problem.name+": z", "time (s)", "z","z");

    plot(r1,r2,problem.name+": x-y", "x (m)", "y (m)","y");

    plot(r1,r2,problem.name+": x-y trajectory", "x (m)", "y (m)","y", "pdf", "lambert_xy.pdf");

    plot3(r1,r2,r3,problem.name+": trajectory","x (m)", "y (m)", "z (m)");



}

//////////////////////////////////////////////////////////////////////
/////////////////////        END OF FILE        ///////////////////////
//////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the text box below and in Figure 3.44, which show the trajectory from $\mathbf{r}(0)$ to $\mathbf{r}(t_f)$, respectively.

```
PSOPT results summary
=====================


Problem:  Lambert problem
CPU time (seconds): 2.420000e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  0.000000e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 4.560000e+03
Phase 1 maximum relative local error: 3.440783e-05
NLP solver reports:  The problem solved!
```

The resulting initial and final velocity vectors are:

$$
\begin{aligned}
\mathbf{v}(0) &= [2058.902605, 2915.961924, -6.878790137\mathrm{E}-13]^T \\
\mathbf{v}(t_f) &= [-3451.55505, 910.3192974, -6.878787164\mathrm{E}-13]^T
\end{aligned}
\tag{3.81}
$$

Figure 3.44: Trajectory between the initial and final positions for Lambert's problem

3.20 Lee-Ramirez bioreactor

Consider the following optimal control problem, which is known in the literature as the Lee-Ramirez bioreactor [30, 37]. Find t_f and $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = -x_1(t_f)x_4(t_f) + \int_0^{t_f} \rho[\dot{u}_1(t)^2 + \dot{u}_2(t)^2]dt \qquad (3.82)$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{x}_1 &= u_1 + u_2; \\
\dot{x}_2 &= g_1 x_2 - \frac{u_1 + u_2}{x_1} x_2; \\
\dot{x}_3 &= 100\frac{u_1}{x_1} - \frac{u_1 + u_2}{x_1} x_3 - (g_1/0.51)x_2; \\
\dot{x}_4 &= R_{fp}x_2 - \frac{u_1 + u_2}{x_1} x_4; \\
\dot{x}_5 &= 4\frac{u_2}{x_1} - \frac{u_1 + u_2}{x_1} x_5; \\
\dot{x}_6 &= -k_1 x_6; \\
\dot{x}_7 &= k_2(1 - x_7).
\end{aligned}
\qquad (3.83)
$$

where $t_f = 10$, $\rho = 1/N$, and N is the number of discretization nodes,

$$k_1 = 0.09x_5/(0.034 + x_5);$$
$$k_2 = k_1;$$
$$g_1 = (x_3/(14.35 + x_3(1.0 + x_3/111.5)))(x_6 + 0.22x_7/(0.22 + x_5));$$
$$R_{fp} = (0.233x_3/(14.35 + x_3(1.0 + x_3/111.5)))((0.0005 + x_5)/(0.022 + x_5));$$
$$(3.84)$$

the initial conditions:

$$
\begin{aligned}
x_1(0) &= 1 \\
x_2(0) &= 0.1 \\
x_3(0) &= 40 \\
x_4(0) &= 0 \\
x_5(0) &= 0 \\
x_6(0) &= 1.0 \\
x_7(0) &= 0
\end{aligned}
\qquad (3.85)
$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
////////////////            bioreactor.cxx          ////////////////////
//////////////////////////////////////////////////////////////////////
////////////////            PSOPT Template          ////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:  Lee-Ramirez bioreactor             ////////////////
//////// Last modified: 09 July 2009                ////////////////
//////// Reference:     PROPT User Manual            ////////////////
////////                                             ////////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009  ////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)               ////////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"


//////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////




adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   adouble x1tf = final_states[CINDEX(1)];
   adouble x4tf = final_states[CINDEX(4)];

   return -(x1tf*x4tf);
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
```

```
    adouble u2 = controls[ CINDEX(2) ];

    adouble dot_u1, dot_u2;

    double Q = 0;

    get_control_derivative( &dot_u1, 1, iphase, time, xad, workspace);

    get_control_derivative( &dot_u2, 2, iphase, time, xad, workspace);

    double rho = 0.1/get_number_of_nodes(*workspace->problem, iphase);

    return   (Q*u2 + rho*(dot_u1*dot_u1 + dot_u2*dot_u2));

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's //////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

    adouble u1 = controls[ CINDEX(1) ];
    adouble u2 = controls[ CINDEX(2) ];

    adouble x1 = states[ CINDEX(1) ];
    adouble x2 = states[ CINDEX(2) ];
    adouble x3 = states[ CINDEX(3) ];
    adouble x4 = states[ CINDEX(4) ];
    adouble x5 = states[ CINDEX(5) ];
    adouble x6 = states[ CINDEX(6) ];
    adouble x7 = states[ CINDEX(7) ];


    adouble k1 = 0.09*x5/(0.034 + x5);

    adouble k2 = k1;

    adouble g1  = (x3/(14.35 + x3*(1.0+x3/111.5)))*(x6 + 0.22*x7/(0.22+x5));

    adouble Rfp = (0.233*x3/(14.35 + x3*(1.0+x3/111.5)))*((0.0005+x5)/(0.022+x5));

    derivatives[ CINDEX(1) ] =   u1 + u2;
    derivatives[ CINDEX(2) ] =   g1*x2 - (u1+u2)/x1*x2;
    derivatives[ CINDEX(3) ] =   100*u1/x1 - (u1+u2)/x1*x3 - g1/0.51*x2;
    derivatives[ CINDEX(4) ] =   Rfp*x2 - (u1+u2)/x1*x4;
    derivatives[ CINDEX(5) ] =   4*u2/x1 - (u1+u2)/x1*x5;
    derivatives[ CINDEX(6) ] =   -k1*x6;
    derivatives[ CINDEX(7) ] =   k2*(1-x7);


}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{

   int i;

   for(i=1;i<= 7; i++) {
        e[CINDEX(i)] = initial_states[CINDEX(i)];
   }



}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////
//////////////////////////////////////////////////////////////////////
```

```
void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
// Single phase problem

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ////////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name           = "Lee-Ramirez bioreactor";
    problem.outfilename                = "bioreactor.txt";

//////////////////////////////////////////////////////////////////////
///////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages               = 0;

    psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 7;
    problem.phases(1).ncontrols  = 2;
    problem.phases(1).nevents    = 7;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes                  = "[20 35 50]";


    psopt_level2_setup(problem, algorithm);



//////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ////////////////////
//////////////////////////////////////////////////////////////////////

    int i;

    problem.phases(1).bounds.lower.states = -10.0*ones(7,1);
    problem.phases(1).bounds.upper.states = "[10.0 10.0 50.0 10.0 10.0 10.0 10.0]";


    problem.phases(1).bounds.lower.controls(1) = 0.0;
    problem.phases(1).bounds.lower.controls(2) = 0.0;



    problem.phases(1).bounds.upper.controls(1) = 1.0;
    problem.phases(1).bounds.upper.controls(2) = 1.0;



    problem.phases(1).bounds.lower.events = "[1.0 0.1 40.0 0.0 0.0 1.0 0.0]";
    problem.phases(1).bounds.upper.events = "[1.0 0.1 40.0 0.0 0.0 1.0 0.0]";


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;
```

```
    problem.phases(1).bounds.lower.EndTime    = 10.0;
    problem.phases(1).bounds.upper.EndTime    = 10.0;


//////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////////
//////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae            = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

//////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess /////////////////////////
//////////////////////////////////////////////////////////////////////////

    int nnodes      = problem.phases(1).nodes(1);
    int ncontrols                        = problem.phases(1).ncontrols;
    int nstates                          = problem.phases(1).nstates;

    DMatrix state_guess    =  zeros(nstates,nnodes);
    DMatrix control_guess  =  zeros(ncontrols,nnodes);
    DMatrix param_guess    =  zeros(0,0);
    DMatrix time_guess     =  linspace(0,10.0,nnodes);


    state_guess(1,colon()) = 1.0*ones(1,nnodes);
    state_guess(2,colon()) = 0.1*ones(1,nnodes);
    state_guess(3,colon()) = 40.0*ones(1,nnodes);
    state_guess(4,colon()) = 0.0*ones(1,nnodes);
    state_guess(5,colon()) = 0.0*ones(1,nnodes);
    state_guess(6,colon()) = 1.0*ones(1,nnodes);
    state_guess(7,colon()) = 0.0*ones(1,nnodes);


    control_guess(1,colon()) = time_guess/30.0;
    control_guess(2,colon()) = zeros(1,nnodes);


    auto_phase_guess(problem, control_guess, state_guess, param_guess, time_guess);


//////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  /////////////////////////////
//////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max           = 1000;
    algorithm.nlp_tolerance          = 1.e-5;
    algorithm.nlp_method             = "IPOPT";
    algorithm.scaling                = "automatic";
    algorithm.derivatives            = "automatic";
    algorithm.defect_scaling         = "jacobian-based";
    algorithm.diff_matrix            = "central-differences";


//////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   /////////////////
//////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

//////////////////////////////////////////////////////////////////////////
//////////////  Extract relevant variables from solution structure   //////////
//////////////////////////////////////////////////////////////////////////


    DMatrix x, u, t;

    x      = solution.get_states_in_phase(1);
    u      = solution.get_controls_in_phase(1);
    t      = solution.get_time_in_phase(1);


//////////////////////////////////////////////////////////////////////////
//////////////  Save solution data to files if desired /////////////////////
//////////////////////////////////////////////////////////////////////////
```

```
   x.Save("x.dat");
   u.Save("u.dat");
   t.Save("t.dat");


//////////////////////////////////////////////////////////////////////////
//////////  Plot some results if desired (requires gnuplot) //////////////
//////////////////////////////////////////////////////////////////////////

   plot(t,u,problem.name+": control","time (s)", "control 1", "u1 u2");

   x(3,colon()) = x(3,colon())/10.0;

   plot(t,x,problem.name+": state","time (s)", "state", "x1 x2 x3/10 x4 x5 x6 x7");


   plot(t,u,problem.name+": control","time (s)", "control 1", "u1 u2",
     "pdf", "bioreactor_controls.pdf");

   x(3,colon()) = x(3,colon())/10.0;

   plot(t,x,problem.name+": states","time (s)", "states", "x1 x2 x3/10 x4 x5 x6 x7",
     "pdf", "bioreactor_states.pdf");




}
//////////////////////////////////////////////////////////////////////////
///////////////////////////      END OF FILE      /////////////////////////////
//////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.45 and 3.46, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================


Problem:  Lee-Ramirez bioreactor
CPU time (seconds): 4.216499e+01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  -6.043913e+00
Phase 1 endpoint cost function value: -6.044801e+00
Phase 1 integrated part of the cost:  8.886589e-04
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+01
Phase 1 maximum relative local error: 7.864620e-03
NLP solver reports:  The problem solved!
```

Figure 3.45: States for the Lee-Ramirez bioreactor problem

Figure 3.46: Control for the Lee-Ramirez bioreactor problem

3.21 Li's parameter estimation problem

This is a parameter estimation problem with two parameters and three observed variables, which is presented b Li *et. al* [29].

The dynamic equations are given by:

$$\frac{dx}{dt} = M(t,p)x + f(t), \ t \in [0, \pi] \tag{3.86}$$

with boundary condition:

$$x(0) + x(\pi) = (1 + e^{\pi}) [1, \ 1, \ 1]^T$$

where

$$M(t,p) = \begin{bmatrix} p_2 - p_1 \cos(p_2 t) & 0 & p_2 + p_1 \sin(p_2 t) \\ 0 & p_1 & 0 \\ -p_2 + p_1 \sin(p_2 t) & 0 & p_2 + p_1 \cos(p_2 t) \end{bmatrix} \tag{3.87}$$

and

$$f(t) = \begin{bmatrix} -1 + 19(\cos(t) - \sin(t)) \\ -18 \\ 1 - 19(\cos(t) + \sin(t)) \end{bmatrix} \tag{3.88}$$

and the observation functions are:

$$\begin{align} g_1 &= x_1 \\ g_2 &= x_2 \\ g_3 &= x_3 \end{align} \tag{3.89}$$

The trajectories of the dynamic system are characterised by rapidly varying fast and slow components if the difference between the two parameters p_1 and p_2 is large, which may cause numerical problems to some ODE solvers.

The estimation data set is generated by adding Gaussian noise with standard deviation 1 around the solution $[x_1(t), x_2(t), x_3(t)]^T = [e^t, \ e^t, \ e^t]^T$, with $N = 33$ equidistant samples within the interval $t = [0, \pi]$. The true values of the parameters are $p_1 = 19$ and $p_2 = 1$. The weights of the three observations are the same and equal to one.

The solution is found using Legendre discretisation with 40 grid points. The code that solves the problem is shown below. The estimated parameter values and their 95% confidence limits for $n_s = 129$ samples are shown in Table 3.21. Figure 3.47 shows the observations as well as the estimated values of variable x_1.

```
//////////////////////////////////////////////////////////////////////
//////////////////        param2.cxx         //////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////        PSOPT  Example        ///////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
```

```
//////// Title:  Parameter estimation for ODE with 2 parameters //////////
//////// Last modified:  31 Jan 2014                   ///////////////
//////// Reference:     Li et al (2005)                ///////////////
//////// (See PSOPT handbook for full reference)       ///////////////
////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2014        ///////////////
////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                  ///////////////
////////////////////////////////////////////////////////////////////

#include "psopt.h"

////////////////////////////////////////////////////////////////////
////////////////////  Define the observation function //////////
////////////////////////////////////////////////////////////////////


void  observation_function( adouble* observations,
                            adouble* states, adouble* controls,
                            adouble* parameters, adouble& time, int k,
                            adouble* xad, int iphase, Workspace* workspace)
{
int i;
for (i=0; i<3; i++) {
          observations[ i ] = states[ i ];
}
}



////////////////////////////////////////////////////////////////////
////////////////////  Define the DAE's /////////////////////////////////
////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

   // Variables
      adouble x1, x2, x3, p1, p2, t;

   // Differential states
      x1 = states[CINDEX(1)];
      x2 = states[CINDEX(2)];
      x3 = states[CINDEX(3)];

    // Parameters
      p1 = parameters[CINDEX(1)];
      p2 = parameters[CINDEX(2)];

      t= time;

   ADMatrix M(3,3), f(3,1), dxdt(3,1), x(3,1), y(3,1);

   x(1,1) = x1; x(2,1) = x2; x(3,1) = x3;

   M(1,1) = p2-p1*cos(p2*t);  M(1,2) = 0.0; M(1,3) = p2 + p1*sin(p2*t);
   M(2,1) = 0.0;                  M(2,2) = p1;  M(2,3) = 0.0;
   M(3,1) = -p2+p1*sin(p2*t); M(3,2) = 0.0; M(3,3) = p2 + p1*cos(p2*t);

   f(1,1) = exp(t)*(-1.0 + 19.0*( cos(t) - sin(t) ) );
   f(2,1) = exp(t)*(-18.0);
   f(3,1) = exp(t)*(1.0 - 19.0*( cos(t) + sin(t) ) );

      // Differential equations


          product_ad(M, x, &y);

   sum_ad(y, f, &dxdt );

          derivatives[CINDEX(1)] = dxdt(1,1);

          derivatives[CINDEX(2)] = dxdt(2,1);

   derivatives[CINDEX(3)] = dxdt(3,1);

}
```

```
//////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ///////////////////////////
//////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   int i;
   double rhs = 1.0 + exp(pi);

   for (i=0; i<3; i++) {

      e[i]  = initial_states[i] + final_states[i] - rhs;

   }

}


//////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
//////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine //////////////////////////////
//////////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ///////////////////////////////
//////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ///////////////////////////////
//////////////////////////////////////////////////////////////////////////

    problem.name          =       "Parameter estimation for ODE with two parameters";
    problem.outfilename        =      "param2.txt";

//////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////////

    problem.nphases           = 1;
    problem.nlinkages               = 0;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup ////////////
//////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 3;
    problem.phases(1).ncontrols  = 0;
    problem.phases(1).nevents    = 3;
    problem.phases(1).npath      = 0;
    problem.phases(1).nparameters             = 2;
    problem.phases(1).nodes            = 40;
    problem.phases(1).nobserved              = 3;
    problem.phases(1).nsamples               = 129;

    psopt_level2_setup(problem, algorithm);

//////////////////////////////////////////////////////////////////////////
////////////  Load data for parameter estimation              ////////////
//////////////////////////////////////////////////////////////////////////

    int iphase = 1;
    load_parameter_estimation_data(problem, iphase, "param2.dat");
```

```
   problem.phases(1).observation_nodes.Print("observation nodes");
   problem.phases(1).observations.Print("observations");
   problem.phases(1).residual_weights.Print("weights");


////////////////////////////////////////////////////////////////////////
/////////////////// Declare DMatrix objects to store results /////////////
////////////////////////////////////////////////////////////////////////

   DMatrix x, p, t;

////////////////////////////////////////////////////////////////////////
/////////////////// Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////

   problem.phases(1).bounds.lower.states(1) = 0.0;
   problem.phases(1).bounds.lower.states(2) = 0.0;
   problem.phases(1).bounds.lower.states(3) = 0.0;


   problem.phases(1).bounds.upper.states(1) = 30.0;
   problem.phases(1).bounds.upper.states(2) = 30.0;
   problem.phases(1).bounds.upper.states(3) = 30.0;


   problem.phases(1).bounds.lower.events(1) = 0.0;
   problem.phases(1).bounds.lower.events(2) = 0.0;
   problem.phases(1).bounds.lower.events(3) = 0.0;


   problem.phases(1).bounds.upper.events(1) = 0.0;
   problem.phases(1).bounds.upper.events(2) = 0.0;
   problem.phases(1).bounds.upper.events(3) = 0.0;


   problem.phases(1).bounds.lower.parameters(1)  = 0.0;
   problem.phases(1).bounds.lower.parameters(2)  = 0.0;

   problem.phases(1).bounds.upper.parameters(1)  = 30.0;
   problem.phases(1).bounds.upper.parameters(2)  = 30.0;

   problem.phases(1).bounds.lower.StartTime    = 0.0;
   problem.phases(1).bounds.upper.StartTime    = 0.0;

   problem.phases(1).bounds.lower.EndTime      = pi;
   problem.phases(1).bounds.upper.EndTime      = pi;

////////////////////////////////////////////////////////////////////////
/////////////////// Register problem functions  //////////////////////////
////////////////////////////////////////////////////////////////////////

   problem.dae  = &dae;
   problem.events  = &events;
   problem.linkages = &linkages;
   problem.observation_function = & observation_function;

////////////////////////////////////////////////////////////////////////
/////////////////// Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////

   int nnodes =      (int) problem.phases(1).nsamples;

   DMatrix state_guess(3, nnodes);
   DMatrix param_guess(2,1);


   state_guess(1,colon()) = linspace(1.0, exp(pi), nnodes );
   state_guess(2,colon()) = linspace(1.0, exp(pi), nnodes );
   state_guess(3,colon()) = linspace(1.0, exp(pi), nnodes );

   param_guess(1) = 19.0*1.5;
   param_guess(2) = 1.0*1.5;

   problem.phases(1).guess.states         = state_guess;
   problem.phases(1).guess.time           = linspace(0.0, pi, nnodes);
   problem.phases(1).guess.parameters     = param_guess;



////////////////////////////////////////////////////////////////////////
/////////////////// Enter algorithm options  /////////////////////////////
```

229

Table 3.2: Estimated parameter values and 95 percent statistical confidence limits on estimated parameters

Parameter	Low Confidence Limit	Value	High Confidence Limit
p_1	1.907055e+01	1.907712e+01	1.908369e+01
p_2	9.984900e-01	9.984990e-01	9.985080e-01

```
//////////////////////////////////////////////////////////////////////

    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
    algorithm.collocation_method      = "Legendre";
//  algorithm.mesh_refinement         = "automatic";
//  algorithm.defect_scaling          = "jacobian-based";
//  algorithm.nlp_iter_max            = 1000;
    algorithm.ode_tolerance           = 1.e-4;

//////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem  //////////////
//////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

//////////////////////////////////////////////////////////////////////
//////////// Extract relevant variables from solution structure //////////
//////////////////////////////////////////////////////////////////////

    x = solution.get_states_in_phase(1);
    t = solution.get_time_in_phase(1);
    p = solution.get_parameters_in_phase(1);


//////////////////////////////////////////////////////////////////////
//////////// Save solution data to files if desired ////////////////////
//////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    t.Save("t.dat");
    p.Print("Estimated parameters");


//////////////////////////////////////////////////////////////////////
//////////// Plot some results if desired (requires gnuplot) ///////////////
//////////////////////////////////////////////////////////////////////

    DMatrix tm;
    DMatrix ym;

    tm = problem.phases(1).observation_nodes;
    ym = problem.phases(1).observations;

    spplot(t,x(1,colon()),tm,ym(1,colon()),problem.name, "time (s)", "state x1", "x1 y1");
    spplot(t,x(2,colon()),tm,ym(2,colon()),problem.name, "time (s)", "state x2", "x2 y2");
    spplot(t,x(3,colon()),tm,ym(3,colon()),problem.name, "time (s)", "state x3", "x3 y3");


    spplot(t,x(1,colon()),tm,ym(1,colon()),problem.name, "time (s)", "state x1", "x1 y1", "pdf", "x1.pdf");
    spplot(t,x(2,colon()),tm,ym(2,colon()),problem.name, "time (s)", "state x2", "x2 y2", "pdf", "x2.pdf");
    spplot(t,x(3,colon()),tm,ym(3,colon()),problem.name, "time (s)", "state x3", "x3 y3", "pdf", "x3.pdf");


}




//////////////////////////////////////////////////////////////////////
//////////////////////      END OF FILE      ///////////////////////////
//////////////////////////////////////////////////////////////////////
```

Figure 3.47: Observations and estimated state $x_1(t)$

3.22 Linear tangent steering problem

Consider the following optimal control problem, which is known in the literature as the linear tangent steering problem [3]. Find t_f and $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = t_f \tag{3.90}$$

subject to the dynamic constraints

$$
\begin{array}{rcl}
\dot{x}_1 & = & x_2 \\
\dot{x}_2 & = & a\cos(u) \\
\dot{x}_3 & = & x_4 \\
\dot{x}_4 & = & a\sin(u)
\end{array}
\tag{3.91}
$$

the boundary conditions:

$$
\begin{array}{rcl}
x_1(0) & = & 0 \\
x_2(0) & = & 0 \\
x_3(0) & = & 0 \\
x_4(0) & = & 0 \\
x_2(t_f) & = & 45.0 \\
x_3(t_f) & = & 5.0 \\
x_4(t_f) & = & 0.0
\end{array}
\tag{3.92}
$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
///////////////////////////////////////////////////////////////////
/////////////////        lts.cxx        //////////////////////////
///////////////////////////////////////////////////////////////////
///////////////           PSOPT  Example        //////////////////
///////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////
//////// Title: Linear tangent steering problem      //////////////
//////// Last modified:       16 February 2009       //////////////
//////// Reference:           Betts (2001)           //////////////
//////// (See PSOPT handbook for full reference)     //////////////
///////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009   //////////////
///////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////
//////// is distributed under the terms of the GNU Lesser /////////
//////// General Public License (LGPL)               //////////////
///////////////////////////////////////////////////////////////////

#include "psopt.h"

///////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
///////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return tf;
}

///////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
///////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    return  0.0;
}

///////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ///////////////////////////////
///////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

   adouble x1 = states[ CINDEX(1) ];
   adouble x2 = states[ CINDEX(2) ];
   adouble x3 = states[ CINDEX(3) ];
   adouble x4 = states[ CINDEX(4) ];

   adouble u = controls[ CINDEX(1) ];

   double a = 100.0;

   derivatives[ CINDEX(1) ] = x2;
   derivatives[ CINDEX(2) ] = a*cos(u);
   derivatives[ CINDEX(3) ] = x4;
   derivatives[ CINDEX(4) ] = a*sin(u);
}

///////////////////////////////////////////////////////////////////
//////////////////  Define the events function /////////////////////
///////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x10 = initial_states[ CINDEX(1) ];
   adouble x20 = initial_states[ CINDEX(2) ];
   adouble x30 = initial_states[ CINDEX(3) ];
   adouble x40 = initial_states[ CINDEX(4) ];
   adouble x2f = final_states[   CINDEX(2) ];
   adouble x3f = final_states[   CINDEX(3) ];
   adouble x4f = final_states[   CINDEX(4) ];
```

232

```
   e[ CINDEX(1) ] = x10;
   e[ CINDEX(2) ] = x20;
   e[ CINDEX(3) ] = x30;
   e[ CINDEX(4) ] = x40;
   e[ CINDEX(5) ] = x2f;
   e[ CINDEX(6) ] = x3f;
   e[ CINDEX(7) ] = x4f;
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////
////////////////////  Define the main routine ///////////////////////////
//////////////////////////////////////////////////////////////////////


int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ///////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ///////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name         = "Linear Tangent Steering Problem";

    problem.outfilename              = "lts.txt";

//////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages               = 0;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////
/////////////   Define phase related information & do level 2 setup /////////////
//////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates    = 4;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 7;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes                   = "[10, 30]";

    psopt_level2_setup(problem, algorithm);


//////////////////////////////////////////////////////////////////////
//////////////////  Declare DMatrix objects to store results //////////////
//////////////////////////////////////////////////////////////////////

    DMatrix x, u, t;
    DMatrix lambda, H;

//////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information //////////////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).bounds.lower.states(1) = -100.0;
    problem.phases(1).bounds.lower.states(2) = -100.0;
    problem.phases(1).bounds.lower.states(3) = -100.0;
```

```
    problem.phases(1).bounds.lower.states(4) = -100.0;

    problem.phases(1).bounds.upper.states(1) =  100.0;
    problem.phases(1).bounds.upper.states(2) =  100.0;
    problem.phases(1).bounds.upper.states(3) =  100.0;
    problem.phases(1).bounds.upper.states(4) =  100.0;

    problem.phases(1).bounds.lower.controls(1) = -pi/2.0;
    problem.phases(1).bounds.upper.controls(1) =  pi/2.0;

    problem.phases(1).bounds.lower.events(1) = 0.0;
    problem.phases(1).bounds.lower.events(2) = 0.0;
    problem.phases(1).bounds.lower.events(3) = 0.0;
    problem.phases(1).bounds.lower.events(4) = 0.0;
    problem.phases(1).bounds.lower.events(5) = 45.0;
    problem.phases(1).bounds.lower.events(6) = 5.0;
    problem.phases(1).bounds.lower.events(7) = 0.0;

    problem.phases(1).bounds.upper.events(1) = 0.0;
    problem.phases(1).bounds.upper.events(2) = 0.0;
    problem.phases(1).bounds.upper.events(3) = 0.0;
    problem.phases(1).bounds.upper.events(4) = 0.0;
    problem.phases(1).bounds.upper.events(5) = 45.0;
    problem.phases(1).bounds.upper.events(6) = 5.0;
    problem.phases(1).bounds.upper.events(7) = 0.0;

    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 0.0;
    problem.phases(1).bounds.upper.EndTime      = 1.0;


////////////////////////////////////////////////////////////////////
////////////////// Register problem functions  //////////////////////////
////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost   = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;


////////////////////////////////////////////////////////////////////
////////////////// Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////

    DMatrix x0(3,10);

    x0(1,colon()) = linspace(0.0, 12.0, 10);
    x0(2,colon()) = linspace(0.0, 45.0, 10);
    x0(3,colon()) = linspace(0.0,  5.0, 10);
    x0(3,colon()) = linspace(0.0,  0.0, 10);

    problem.phases(1).guess.controls       = ones(1,10);
    problem.phases(1).guess.states         = x0;
    problem.phases(1).guess.time           = linspace(0.0,1.0, 10);

////////////////////////////////////////////////////////////////////
////////////////// Enter algorithm options  ///////////////////////////
////////////////////////////////////////////////////////////////////


    algorithm.nlp_method               = "IPOPT";
    algorithm.scaling                  = "automatic";
    algorithm.derivatives              = "automatic";
    algorithm.nlp_iter_max             = 1000;
    algorithm.nlp_tolerance            = 1.e-6;

////////////////////////////////////////////////////////////////////
////////////////// Now call PSOPT to solve the problem  ///////////////
////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

    if (solution.error_flag) exit(0);

////////////////////////////////////////////////////////////////////
///////////// Extract relevant variables from solution structure  //////////
////////////////////////////////////////////////////////////////////
```
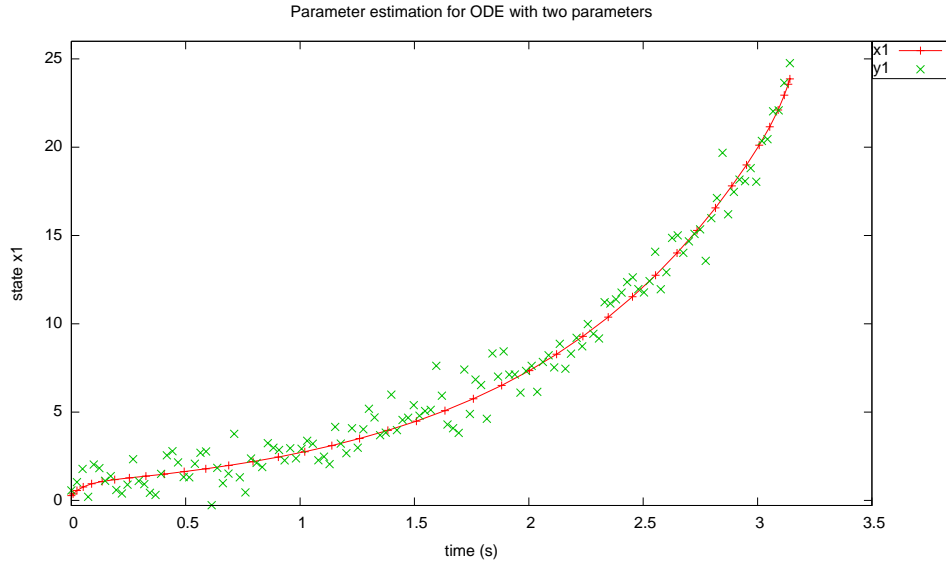
```
    x  = solution.get_states_in_phase(1);
    u  = solution.get_controls_in_phase(1);
    t  = solution.get_time_in_phase(1);


///////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////
///////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");


///////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////
///////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name + ": states", "time (s)", "states", "x1 x2 x3 x4");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u");

    plot(t,x,problem.name + ": states", "time (s)", "states", "x1 x2 x3 x4",
                           "pdf", "lts_states.pdf");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u",
                          "pdf", "lts_control.pdf");




}

///////////////////////////////////////////////////////////////////////
//////////////////////////    END OF FILE    ///////////////////////////
///////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.48 and 3.49, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================


Problem:  Linear Tangent Steering Problem
CPU time (seconds): 4.300000e-01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  5.545709e-01
Phase 1 endpoint cost function value: 5.545709e-01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.545709e-01
Phase 1 maximum relative local error: 1.331388e-07
NLP solver reports:  The problem solved!
```

Figure 3.48: States for the linear tangent steering problem

Figure 3.49: Control for the linear tangent steering problem

3.23 Low thrust orbit transfer

The goal of this problem is to compute an optimal low thrust policy for an spacecraft to go from a standard space shuttle park orbit to a specified final orbit, while maximising the final weight of the spacecraft. The problem is described in detail by Betts [3]. The problem is formulated as follows. Find $\mathbf{u}(t) = [u_r(t), u_\theta(t), u_h(t)]^T, t \in [0, t_f]$, the unknown throtle parameter τ, and the final time t_f, such that the following objective function is minimised:

$$J = -w(t_f) \tag{3.93}$$

subject to the dynamic constraints:

$$\dot{\mathbf{y}} = \mathbf{A}(\mathbf{y})\Delta + \mathbf{b}$$
$$\dot{w} = -T[1 + 0.01\tau]/I_{sp} \tag{3.94}$$

the path constraint:

$$||u(t)||^2 = 1 \tag{3.95}$$

and the parameter bounds:

$$\tau_L \leq \tau \leq 0 \tag{3.96}$$

where $\mathbf{y} = [p, f, g, h, k, L, w]^T$ is the vector of modified equinoctial elements, $w(t)$ is the weight of the spacecraft, I_{sp} is the specific impulse of the engine, expressions for $\mathbf{A}(\mathbf{y})$ and $\mathbf{b}$ are given in [3], the disturbing acceleration Δ is given by:

$$\Delta = \Delta_g + \Delta_T \tag{3.97}$$

where Δ_g is the gravitational disturbing acceleration due to the oblatness of Earth (given in [3]), and Δ_T is the thurst acceleration, given by:

$$\Delta_T = \frac{g_0 T[1 + 0.01\tau]}{w}\mathbf{u}$$

where T is the maximum thrust, and g_0 is the mass to weight conversion factor.

The boundary conditions of the problem are given by:

$$p(t_f) = 40007346.015232 \text{ ft}$$

$$\sqrt{f(t_f)^2 + g(t_f)^2} = 0.73550320568829$$

$$\sqrt{h(t_f)^2 + k(t_f)^2} = 0.61761258786099$$

$$f(t_f)h(t_f) + g(t_f)k(t_f) = 0$$

$$g(t_f)h(t_f) - k(t_f)f(t_f) = 0$$

$$p(0) = 21837080.052835 \text{ft}$$

$$f(0) = 0$$

$$g(0) = 0$$

$$h(0) = 0$$

$$h(0) = 0$$

$$k(0) = 0$$

$$L(0) = \pi \text{ (rad)}$$

$$w(0) = 1 \text{ (lb)}$$

(3.98)

and the values of the parameters are: $g_0 = 32.174$ (ft/sec^2), $I_{sp} = 450$ (sec), $T = 4.446618 \times 10^{-3}$ (lb), $\mu = 1.407645794 \times 10^{16}$ (ft^3/sec^2), $R_e = 20925662.73$ (ft), $J_2 = 1082.639 \times 10^{-6}$, $J_3 = -2.565 \times 10^{-6}$, $J_4 = -1.608 \times 10^{-6}$, $\tau_L = -50$.

An initial guess was computed by forward propagation from the initial conditions, assuming that the direction of the thrust vector is parallel to the cartesian velocity vector, such that the initial control input was computed as follows:

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

(3.99)

where $\mathbf{Q}_r$ is a matrix whose columns are the directions of the rotating radial frame:

$$\mathbf{Q}_r = \begin{bmatrix} \mathbf{i}_r & \mathbf{i}_\theta & \mathbf{i}_h \end{bmatrix} = \begin{bmatrix} \frac{\mathbf{r}}{||\mathbf{r}||} & \frac{(\mathbf{r} \times \mathbf{v}) \times \mathbf{r}}{||\mathbf{r} \times \mathbf{v}||||\mathbf{r}||} & \frac{(\mathbf{r} \times \mathbf{v})}{||\mathbf{r} \times \mathbf{v}||} \end{bmatrix}$$

(3.100)

The problem was solved using local collocation (trapezoidal followed by Hermite-Simpson) with automatic mesh refinement. The $\mathcal{PSOPT}$ code that solves the problem is shown below.

```
//////////////////////////////////////////////////////////////////
///////////////          low_thrust.cxx            /////////////////
//////////////////////////////////////////////////////////////////
///////////////           PSOPT  Example           /////////////////
//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////
//////// Title: Low thrust orbit transfer problem    ///////////////
//////// Last modified: 16 February 2009             ///////////////
//////// Reference:    Betts  (2001)                 ///////////////
```

```
////////  (See PSOPT handbook for full reference)          ////////////////
//////////////////////////////////////////////////////////////////////////
////////       Copyright (c) Victor M. Becerra, 2009       ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                     ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"


//////////////////////////////////////////////////////////////////////////
//////////////////////  Define auxiliary functions           //////////
//////////////////////////////////////////////////////////////////////////

adouble legendre_polynomial( adouble x, int n)
{
// This function computes the value of the legendre polynomials
// for a given value of the argument x and for n=0...5 only

  adouble retval=0.0;

  switch(n) {
     case 0:
         retval=1.0; break;
     case 1:
         retval= x; break;
     case 2:
         retval= 0.5*(3.0*pow(x,2)-1.0); break;
     case 3:
         retval= 0.5*(5.0*pow(x,3)- 3*x); break;
     case 4:
         retval= (1.0/8.0)*(35.0*pow(x,4) - 30.0*pow(x,2) + 3.0); break;
     case 5:
         retval= (1.0/8.0)*(63.0*pow(x,5) - 70.0*pow(x,3) + 15.0*x); break;
     default:
         error_message("legendre_polynomial(x,n) is limited to n=0...5");
  }

  return retval;

}

adouble legendre_polynomial_derivative( adouble x, int n)
{
// This function computes the value of the legendre polynomial derivatives
// for a given value of the argument x and for n=0...5 only.

  adouble retval=0.0;

  switch(n) {
     case 0:
         retval=0.0; break;
     case 1:
         retval= 1.0; break;
     case 2:
         retval= 0.5*(2.0*3.0*x); break;
     case 3:
         retval= 0.5*(3.0*5.0*pow(x,2)-3.0); break;
     case 4:
         retval= (1.0/8.0)*(4.0*35.0*pow(x,3) - 2.0*30.0*x ); break;
     case 5:
         retval= (1.0/8.0)*(5.0*63.0*pow(x,4) - 3.0*70.0*pow(x,2) + 15.0); break;
     default:
         error_message("legendre_polynomial_derivative(x,n) is limited to n=0...5");
  }

  return retval;

}


void compute_cartesian_trajectory(const DMatrix& x, DMatrix& xyz )
{

  int npoints = x.GetNoCols();
  xyz.Resize(3,npoints);

  for(int i=1; i<=npoints;i++) {
```

```
   double p = x(1,i);
   double f = x(2,i);
   double g = x(3,i);
   double h = x(4,i);
   double k = x(5,i);
   double L = x(6,i);

   double q      =  1.0 + f*cos(L) + g*sin(L);
   double r      =  p/q;
   double alpha2 = h*h - k*k;
   double X      = sqrt( h*h + k*k );
   double s2     = 1 + X*X;

   double r1 = r/s2*( cos(L) + alpha2*cos(L) + 2*h*k*sin(L));
   double r2 = r/s2*( sin(L) - alpha2*sin(L) + 2*h*k*cos(L));
   double r3 = 2*r/s2*( h*sin(L) - k*cos(L) );

   xyz(1,i) = r1;
   xyz(2,i) = r2;
   xyz(3,i) = r3;

   }

}

//////////////////////////////////////////////////////////////////////
/////////////////// Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{

   if (iphase == 1) {
   adouble w = final_states[CINDEX(7)];
   return (-w);
   }
   else {
   return (0);
   }
}

//////////////////////////////////////////////////////////////////////
/////////////////// Define the integrand (Lagrange) cost function //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                      adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    return  0.0;
}


//////////////////////////////////////////////////////////////////////
/////////////////// Define the DAE's /////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
        adouble* controls, adouble* parameters, adouble& time,
        adouble* xad, int iphase, Workspace* workspace)
{
   // Local integers
   int i, j;
   // Define constants:
   double Isp  = 450.0;              // [sec]
   double mu   = 1.407645794e16;     // [f2^2/sec^2]
   double g0   = 32.174;             // [ft/sec^2]
   double T    = 4.446618e-3;        // [lb]
   double Re   = 20925662.73;        // [ft]
   double J[5];
   J[2] =  1082.639e-6;
   J[3] = -2.565e-6;
   J[4] = -1.608e-6;

   // Extract individual variables

   adouble p = states[ CINDEX(1) ];
   adouble f = states[ CINDEX(2) ];
   adouble g = states[ CINDEX(3) ];
   adouble h = states[ CINDEX(4) ];
   adouble k = states[ CINDEX(5) ];
   adouble L = states[ CINDEX(6) ];
```

240

```
        adouble w = states[ CINDEX(7) ];

        adouble* u  = controls;

        adouble tau  = parameters[ CINDEX(1) ];

        // Define some dependent variables

        adouble q      =  1.0 + f*cos(L) + g*sin(L);
        adouble r      =  p/q;
        adouble alpha2 = h*h - k*k;
        adouble X      = sqrt( h*h + k*k );
        adouble s2     = 1 + X*X;

        // r and v

        adouble r1 = r/s2*( cos(L) + alpha2*cos(L) + 2*h*k*sin(L));
        adouble r2 = r/s2*( sin(L) - alpha2*sin(L) + 2*h*k*cos(L));
        adouble r3 = 2*r/s2*( h*sin(L) - k*cos(L) );

        adouble rvec[3];

        rvec[ CINDEX(1) ] = r1; rvec[ CINDEX(2)] = r2; rvec[ CINDEX(3) ] = r3;

        adouble v1 = -(1.0/s2)*sqrt(mu/p)*(  sin(L) + alpha2*sin(L) - 2*h*k*cos(L) + g - 2*f*h*k + alpha2*g);
        adouble v2 = -(1.0/s2)*sqrt(mu/p)*( -cos(L) + alpha2*cos(L) + 2*h*k*sin(L) - f + 2*g*h*k + alpha2*f);
        adouble v3 =  (2.0/s2)*sqrt(mu/p)*(h*cos(L) +      k*sin(L) + f*h + g*k);

        adouble vvec[3];

        vvec[ CINDEX(1) ] = v1; vvec[ CINDEX(2)] = v2; vvec[ CINDEX(3) ] = v3;

        // compute Qr

        adouble ir[3], ith[3], ih[3];
        adouble rv[3];
        adouble rvr[3];

        cross( rvec, vvec, rv );

        cross( rv, rvec, rvr );

        adouble norm_r = sqrt( dot(rvec, rvec, 3) );

        adouble norm_rv = sqrt( dot(rv, rv, 3) );

        for (i=0; i<3; i++) {

           ir[i]  = rvec[i]/norm_r;

           ith[i] = rvr[i]/( norm_rv*norm_r );

           ih[i]  = rv[i]/norm_rv;

        }

        adouble Qr1[3], Qr2[3], Qr3[3];

        for(i=0; i< 3; i++)
        {
             // Columns of matrix Qr
             Qr1[i] = ir[i];
             Qr2[i] = ith[i];
             Qr3[i] = ih[i];
        }

        // Compute in

        adouble en[3];
        en[ CINDEX(1) ] = 0.0; en[ CINDEX(2) ]= 0.0; en[ CINDEX(3) ] = 1.0;

        adouble enir = dot(en,ir,3);

        adouble in[3];

        for(i=0;i<3;i++) {
           in[i] = en[i] - enir*ir[i];
        }

        adouble norm_in = sqrt( dot( in, in, 3 ) );

        for(i=0;i<3;i++) {
           in[i] = in[i]/norm_in;
```

```
    }

    // Geocentric latitude angle:

    adouble sin_phi =  rvec[ CINDEX(3) ]/ sqrt( dot(rvec,rvec,3) ) ;
    adouble cos_phi =  sqrt(1.0- pow(sin_phi,2.0));

    adouble deltagn = 0.0;
    adouble deltagr = 0.0;
    for (j=2; j<=4;j++) {
      adouble Pdash_j = legendre_polynomial_derivative( sin_phi, j );
      adouble P_j     = legendre_polynomial( sin_phi, j );
      deltagn += -mu*cos_phi/(r*r)*pow(Re/r,j)*Pdash_j*J[j];
      deltagr += -mu/(r*r)* (j+1)*pow( Re/r,j)*P_j*J[j];
    }

    // Compute vector delta_g

    adouble delta_g[3];
    for (i=0;i<3;i++) {
        delta_g[i] = deltagn*in[i] - deltagr*ir[i];
    }

    // Compute vector DELTA_g

    adouble DELTA_g[3];

    DELTA_g[ CINDEX(1) ] = dot(Qr1, delta_g,3);
    DELTA_g[ CINDEX(2) ] = dot(Qr2, delta_g,3);
    DELTA_g[ CINDEX(3) ] = dot(Qr3, delta_g,3);

    // Compute DELTA_T

    adouble DELTA_T[3];


    for(i=0;i<3;i++) {
        DELTA_T[i] =  g0*T*(1.0+0.01*tau)*u[i]/w;
    }

    // Compute DELTA

    adouble DELTA[3];

    for(i=0;i<3;i++) {
        DELTA[i] =  DELTA_g[i] + DELTA_T[i];
    }

    adouble delta1= DELTA[ CINDEX(1) ];
    adouble delta2= DELTA[ CINDEX(2) ];
    adouble delta3= DELTA[ CINDEX(3) ];


 // derivatives

    adouble pdot =                                    2*p/q*sqrt(p/mu)              * delta2;
    adouble fdot =  sqrt(p/mu)*sin(L) * delta1 + sqrt(p/mu)*(1.0/q)*((q+1.0)*cos(L)+f) * delta2
                  -  sqrt(p/mu)*(g/q)*(h*sin(L)-k*cos(L))  * delta3;
    adouble gdot = -sqrt(p/mu)*cos(L) * delta1 + sqrt(p/mu)*(1.0/q)*((q+1.0)*sin(L)+g) * delta2
                  +  sqrt(p/mu)*(f/q)*(h*sin(L)-k*cos(L))  * delta3;
    adouble hdot =  sqrt(p/mu)*s2*cos(L)/(2.0*q)            * delta3;
    adouble kdot =  sqrt(p/mu)*s2*sin(L)/(2.0*q)            * delta3;
    adouble Ldot =  sqrt(p/mu)*(1.0/q)*(h*sin(L)-k*cos(L))* delta3     + sqrt(mu*p)*pow( (q/p),2.);

    adouble wdot = -T*(1.0+0.01*tau)/Isp;

    derivatives[ CINDEX(1) ] = pdot;
    derivatives[ CINDEX(2) ] = fdot;
    derivatives[ CINDEX(3) ] = gdot;
    derivatives[ CINDEX(4) ] = hdot;
    derivatives[ CINDEX(5) ] = kdot;
    derivatives[ CINDEX(6) ] = Ldot;
    derivatives[ CINDEX(7) ] = wdot;

    path[ CINDEX(1) ] = pow( u[CINDEX(1)] , 2)  + pow( u[CINDEX(2)], 2) + pow( u[CINDEX(3)], 2);

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function /////////////////////////
//////////////////////////////////////////////////////////////////////
```

```
void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{

   int offset;

   adouble pti = initial_states[ CINDEX(1) ];
   adouble fti = initial_states[ CINDEX(2) ];
   adouble gti = initial_states[ CINDEX(3) ];
   adouble hti = initial_states[ CINDEX(4) ];
   adouble kti = initial_states[ CINDEX(5) ];
   adouble Lti = initial_states[ CINDEX(6) ];
   adouble wti = initial_states[ CINDEX(7) ];


   adouble ptf = final_states[ CINDEX(1) ];
   adouble ftf = final_states[ CINDEX(2) ];
   adouble gtf = final_states[ CINDEX(3) ];
   adouble htf = final_states[ CINDEX(4) ];
   adouble ktf = final_states[ CINDEX(5) ];
   adouble Ltf = final_states[ CINDEX(6) ];


   if (iphase==1) {
    e[ CINDEX(1) ]  = pti;
    e[ CINDEX(2) ]  = fti;
    e[ CINDEX(3) ]  = gti;
    e[ CINDEX(4) ]  = hti;
    e[ CINDEX(5) ]  = kti;
    e[ CINDEX(6) ]  = Lti;
    e[ CINDEX(7) ]  = wti;
   }

   if (1 == 1) offset = 7;
   else offset = 0;

   if (iphase == 1 ) {
    e[ offset + CINDEX(1) ]  = ptf;
    e[ offset + CINDEX(2) ]  = sqrt( ftf*ftf + gtf*gtf );
    e[ offset + CINDEX(3) ]  = sqrt( htf*htf + ktf*ktf );
    e[ offset + CINDEX(4) ] = ftf*htf + gtf*ktf;
    e[ offset + CINDEX(5) ] = gtf*htf - ktf*ftf;
   }

}



//////////////////////////////////////////////////////////////////////
/////////////////// Define the phase linkages function /////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
 //  auto_link_multiple(linkages, xad, 1);
}



//////////////////////////////////////////////////////////////////////
/////////////////// Define the main routine ////////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////
/////////////////// Declare key structures /////////////////////////////
//////////////////////////////////////////////////////////////////////

   Alg  algorithm;
   Sol  solution;
   Prob problem;

   MSdata msdata;

//////////////////////////////////////////////////////////////////////
/////////////////// Register problem name //////////////////////////////
//////////////////////////////////////////////////////////////////////
```

```
    problem.name        = "Low thrust transfer problem";
    problem.outfilename              = "lowthrust.txt";

////////////////////////////////////////////////////////////////////
///////////// Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                = 0;

    psopt_level1_setup(problem);

////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 7;
    problem.phases(1).ncontrols  = 3;
    problem.phases(1).nparameters            = 1;
    problem.phases(1).nevents            = 12;
    problem.phases(1).npath      = 1;
    problem.phases(1).nodes                  = 80;

    psopt_level2_setup(problem, algorithm);


////////////////////////////////////////////////////////////////////
////////////////////  Enter problem bounds information ////////////////////
////////////////////////////////////////////////////////////////////

    double tauL = -50.0;
    double tauU =   0.0;


    double pti = 21837080.052835;
    double fti = 0.0;
    double gti = 0.0;
    double hti = -0.25396764647494;
    double kti = 0.0;
    double Lti = pi;
    double wti = 1.0;

    double wtf_guess;

    double  SISP = 450.0;
    double  DELTAV = 22741.1460;
    double  CM2W = 32.174;

    wtf_guess  = wti*exp(-DELTAV/(CM2W*SISP));

    double ptf            = 40007346.015232;
    double event_final_9  = 0.73550320568829;
    double event_final_10  = 0.61761258786099;
    double event_final_11 = 0.0;
    double event_final_12_upper = 0.0;
    double event_final_12_lower = -10.0;


    problem.phases(1).bounds.lower.parameters(1) =  tauL;
    problem.phases(1).bounds.upper.parameters(1) =  tauU;

    problem.phases(1).bounds.lower.states(1) = 10.e6;
    problem.phases(1).bounds.lower.states(2) = -0.20;
    problem.phases(1).bounds.lower.states(3) = -0.10;
    problem.phases(1).bounds.lower.states(4) = -1.0;
    problem.phases(1).bounds.lower.states(5) = -0.20;
    problem.phases(1).bounds.lower.states(6) = pi;
    problem.phases(1).bounds.lower.states(7) = 0.0;

    problem.phases(1).bounds.upper.states(1) = 60.e6;
    problem.phases(1).bounds.upper.states(2) = 0.20;
    problem.phases(1).bounds.upper.states(3) = 1.0;
    problem.phases(1).bounds.upper.states(4) = 1.0;
    problem.phases(1).bounds.upper.states(5) = 0.20;
    problem.phases(1).bounds.upper.states(6) = 20*pi;
    problem.phases(1).bounds.upper.states(7) = 2.0;


    problem.phases(1).bounds.lower.controls(1) = -1.0;
    problem.phases(1).bounds.lower.controls(2) = -1.0;
    problem.phases(1).bounds.lower.controls(3) = -1.0;
    problem.phases(1).bounds.upper.controls(1) = 1.0;
```

```
    problem.phases(1).bounds.upper.controls(2) = 1.0;
    problem.phases(1).bounds.upper.controls(3) = 1.0;


    problem.phases(1).bounds.lower.events(1)  = pti;
    problem.phases(1).bounds.lower.events(2)  = fti;
    problem.phases(1).bounds.lower.events(3)  = gti;
    problem.phases(1).bounds.lower.events(4)  = hti;
    problem.phases(1).bounds.lower.events(5)  = kti;
    problem.phases(1).bounds.lower.events(6)  = Lti;
    problem.phases(1).bounds.lower.events(7)  = wti;


    problem.phases(1).bounds.lower.events(8) = ptf;
    problem.phases(1).bounds.lower.events(9) = event_final_9;
    problem.phases(1).bounds.lower.events(10) = event_final_10;
    problem.phases(1).bounds.lower.events(11) = event_final_11;
    problem.phases(1).bounds.lower.events(12) = event_final_12_lower;


    problem.phases(1).bounds.upper.events(1)  = pti;
    problem.phases(1).bounds.upper.events(2)  = fti;
    problem.phases(1).bounds.upper.events(3)  = gti;
    problem.phases(1).bounds.upper.events(4)  = hti;
    problem.phases(1).bounds.upper.events(5)  = kti;
    problem.phases(1).bounds.upper.events(6)  = Lti;
    problem.phases(1).bounds.upper.events(7)  = wti;
    problem.phases(1).bounds.upper.events(8)  = ptf;
    problem.phases(1).bounds.upper.events(9)  = event_final_9;
    problem.phases(1).bounds.upper.events(10) = event_final_10;
    problem.phases(1).bounds.upper.events(11) = event_final_11;
    problem.phases(1).bounds.upper.events(12) = event_final_12_upper;

    double EQ_TOL = 0.001;

    problem.phases(1).bounds.upper.path(1) = 1.0+EQ_TOL;
    problem.phases(1).bounds.lower.path(1) = 1.0-EQ_TOL;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 50000.0;
    problem.phases(1).bounds.upper.EndTime      = 100000.0;


////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost   = &endpoint_cost;
    problem.dae             = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess /////////////////////
////////////////////////////////////////////////////////////////////////

    int nnodes      = 141;
    int ncontrols                       = problem.phases(1).ncontrols;
    int nstates                         = problem.phases(1).nstates;

    DMatrix u_guess     = zeros(ncontrols,nnodes);
    DMatrix x_guess     = zeros(nstates,nnodes);
    DMatrix time_guess  = linspace(0.0,86810.0,nnodes);

    DMatrix param_guess = -25.0*ones(1,1);

    u_guess.Load("U0.dat");
    x_guess.Load("X0.dat");
    time_guess.Load("T0.dat");

    auto_phase_guess(problem, u_guess, x_guess, param_guess, time_guess);


////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ////////////////////////////
////////////////////////////////////////////////////////////////////////
```

```
    algorithm.nlp_iter_max          = 1000;
    algorithm.nlp_tolerance         = 1.e-6;
    algorithm.nlp_method            = "IPOPT";
    algorithm.scaling               = "automatic";
    algorithm.derivatives           = "automatic";
    algorithm.defect_scaling        = "jacobian-based";
    algorithm.jac_sparsity_ratio    =  0.11; // 0.05;
    algorithm.collocation_method    = "trapezoidal";
    algorithm.mesh_refinement       = "automatic";
    algorithm.mr_max_increment_factor = 0.2;




////////////////////////////////////////////////////////////////////////
/////////////////// Now call PSOPT to solve the problem  ////////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure  ////////
////////////////////////////////////////////////////////////////////////


    DMatrix x, u, t;

    x       = solution.get_states_in_phase(1);
    u       = solution.get_controls_in_phase(1);
    t       = solution.get_time_in_phase(1);


    t = t/3600.0;


    DMatrix tau = solution.get_parameters_in_phase(1);

    tau.Print("tau");

////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired /////////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ////////////
////////////////////////////////////////////////////////////////////////

    DMatrix x1 = x(1,colon())/1.e6;
    DMatrix x2 = x(2,colon());
    DMatrix x3 = x(3,colon());
    DMatrix x4 = x(4,colon());
    DMatrix x5 = x(5,colon());
    DMatrix x6 = x(6,colon());
    DMatrix x7 = x(7,colon());
    DMatrix u1 = u(1,colon());
    DMatrix u2 = u(2,colon());
    DMatrix u3 = u(3,colon());

    plot(t,x1,problem.name+": states", "time (h)", "p (1000000 ft)","p (1000000 ft)");

    plot(t,x2,problem.name+": states", "time (h)", "f","f");

    plot(t,x3,problem.name+": states", "time (h)", "g","g");

    plot(t,x4,problem.name+": states", "time (h)", "h","h");

    plot(t,x5,problem.name+": states", "time (h)", "k","k");

    plot(t,x6,problem.name+": states", "time (h)", "L (rev)","L (rev)");

    plot(t,x7,problem.name+": states", "time (h)", "w (lb)","w (lb)");

    plot(t,u1,problem.name+": controls","time (h)", "ur", "ur");

    plot(t,u2,problem.name+": controls","time (h)", "ut", "ut");

    plot(t,u3,problem.name+": controls","time (h)", "uh", "uh");
```

```
   plot(t,x1,problem.name+": states", "time (h)", "p (1000000 ft)","p (1000000 ft)",
       "pdf","lowthr_x1.pdf");

   plot(t,x2,problem.name+": states", "time (h)", "f","f",
 "pdf","lowthr_x2.pdf");

   plot(t,x3,problem.name+": states", "time (h)", "g","g",
 "pdf","lowthr_x3.pdf");

   plot(t,x4,problem.name+": states", "time (h)", "h","h",
 "pdf","lowthr_x4.pdf");

   plot(t,x5,problem.name+": states", "time (h)", "k","k",
           "pdf","lowthr_x5.pdf");

   plot(t,x6,problem.name+": states", "time (h)", "L (rev)","L (rev)",
 "pdf","lowthr_x6.pdf");

   plot(t,x7,problem.name+": states", "time (h)", "w (lb)","w (lb)",
 "pdf","lowthr_x7.pdf");

   plot(t,u1,problem.name+": controls","time (h)", "ur", "ur",
 "pdf","lowthr_u1.pdf");

   plot(t,u2,problem.name+": controls","time (h)", "ut", "ut",
 "pdf","lowthr_u2.pdf");

   plot(t,u3,problem.name+": controls","time (h)", "uh", "uh",
 "pdf","lowthr_u3.pdf");

   DMatrix r;

   compute_cartesian_trajectory(x,r);

   double ft2km = 0.0003048;

   r = r*ft2km;

   DMatrix rnew, tnew;

   tnew = linspace(0.0, t("end"), 1000);

   resample_trajectory( rnew, tnew, r, t );

   plot3(rnew(1,colon()), rnew(2,colon()), rnew(3,colon()),
       "Low thrust transfer trajectory", "x (km)", "y (km)", "z (km)",
       NULL, NULL, "30,97");

   plot3(rnew(1,colon()), rnew(2,colon()), rnew(3,colon()),
      "Low thrust transfer trajectory", "x (km)", "y (km)", "z (km)",
       "pdf", "trajectory.pdf", "30,97");

}

//////////////////////////////////////////////////////////////////////////
//////////////////////       END OF FILE    ////////////////////////////
//////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.50, to 3.55 and 3.56 to 3.58, which contain the modified equinoctial elements and the controls, respectively.

```
PSOPT results summary
=====================


Problem:  Low thrust transfer problem
CPU time (seconds): 9.104000e+01
```

Table 3.3: Mesh refinement statistics: Low thrust transfer problem

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	ϵ_{max}	CPU_a
1	TRP	80	803	653	1968	362	181	0	57558	2.208e-03	5.560e+00
2	TRP	98	983	797	118	119	110	0	23205	2.263e-03	4.060e+00
3	H-S	108	1404	984	145	146	141	0	47012	1.180e-03	7.650e+00
4	H-S	116	1508	1056	209	210	185	0	72660	3.514e-04	1.119e+01
CPU_b	-	-	-	-	-	-	-	-	-	-	8.250e+00
-	-	-	-	-	2440	837	617	0	200435	-	3.671e+01

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, ϵ_{max} = maximum relative ODE error, CPU_a = CPU time in seconds spent by NLP algorithm, CPU_b = additional CPU time in seconds spent by PSOPT
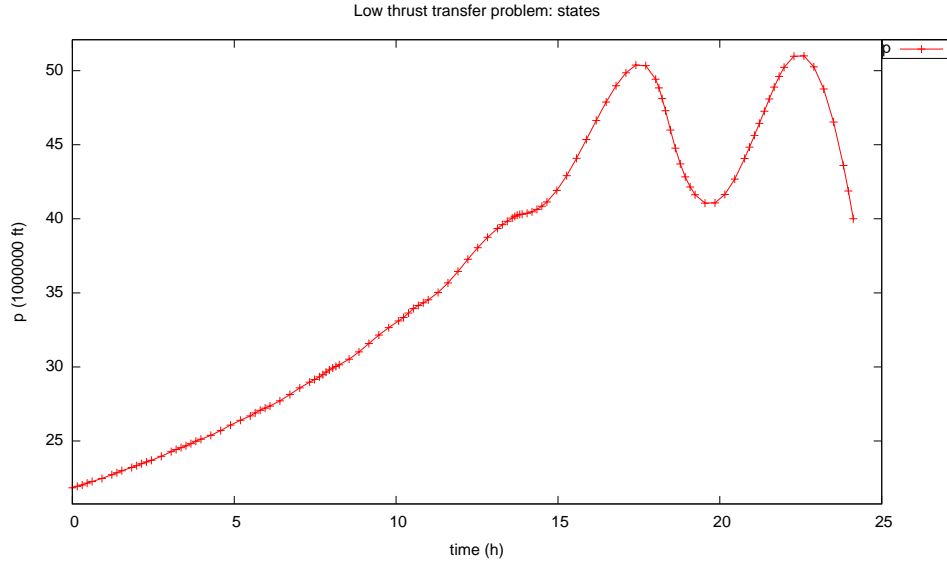
Figure 3.50: Modified equinoctial element p

```
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  -2.203404e-01
Phase 1 endpoint cost function value: -2.203404e-01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 8.684274e+04
Phase 1 maximum relative local error: 3.513482e-04
NLP solver reports:  The problem solved!
```
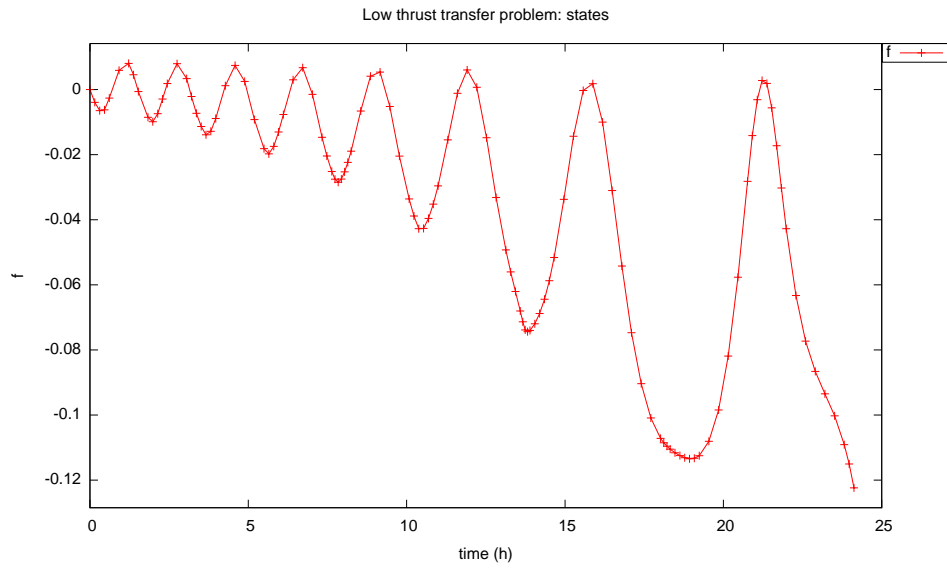
Figure 3.51: Modified equinoctial element f
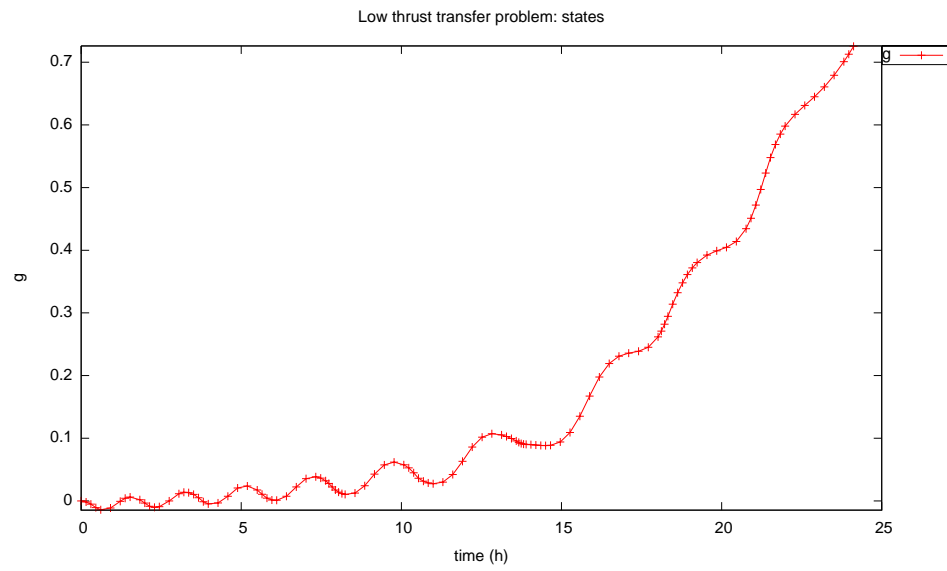
Figure 3.52: Modified equinoctial element g

249
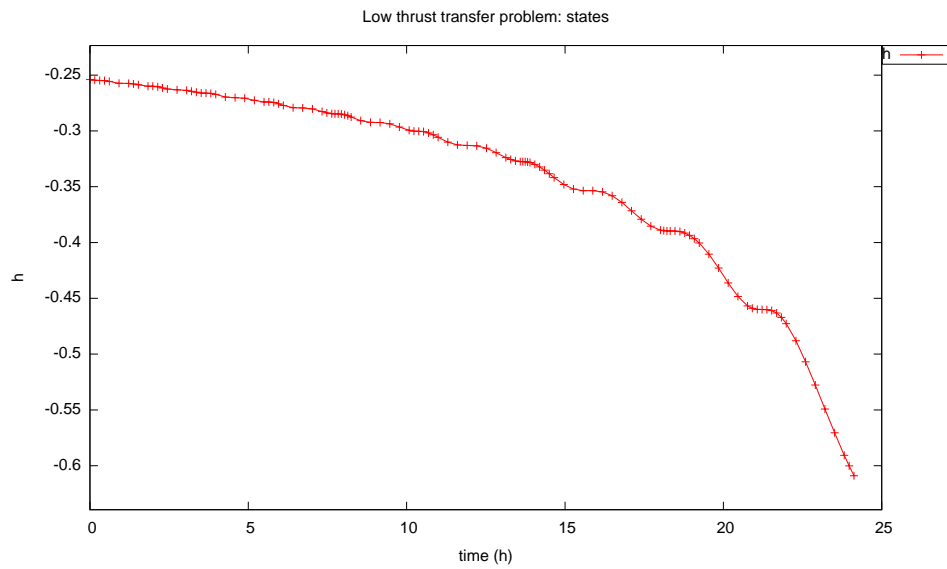
Figure 3.53: Modified equinoctial element h
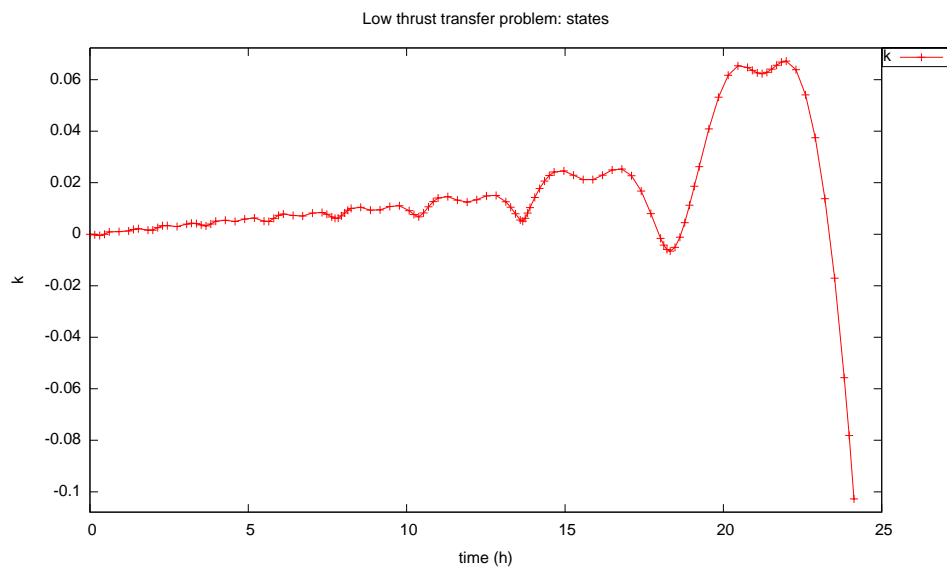
Figure 3.54: Modified equinoctial element k
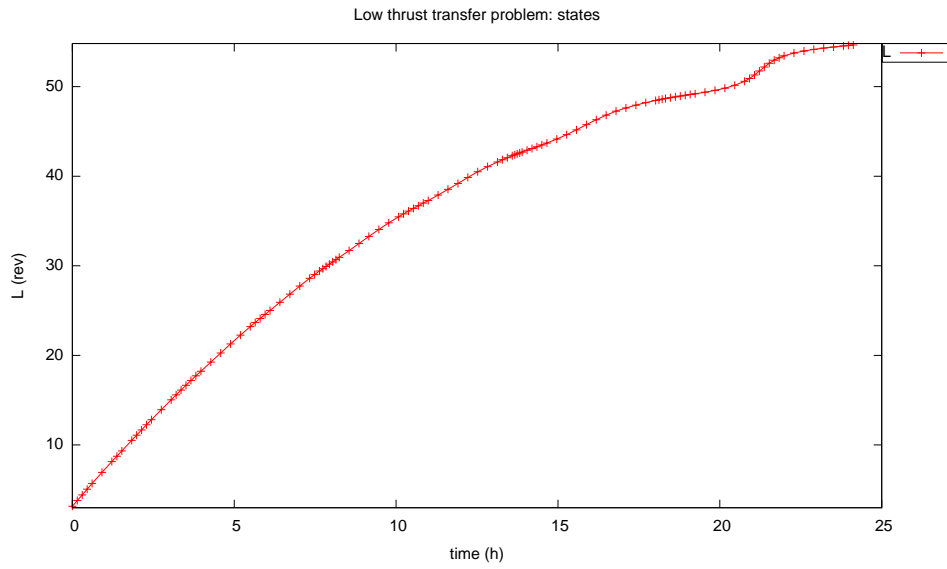
Figure 3.55: Modified equinoctial element L
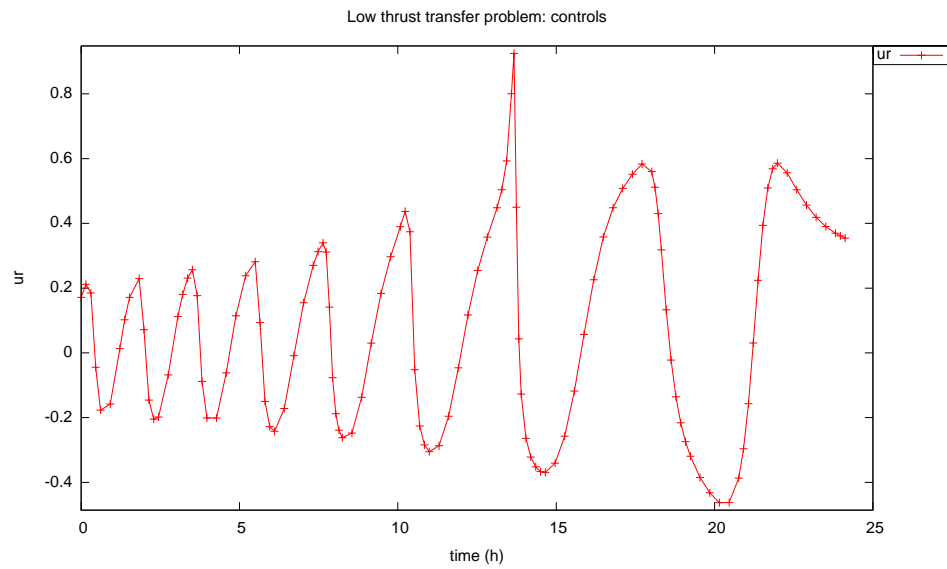
Figure 3.56: Radial component of the thrust direction vector, u_r

251
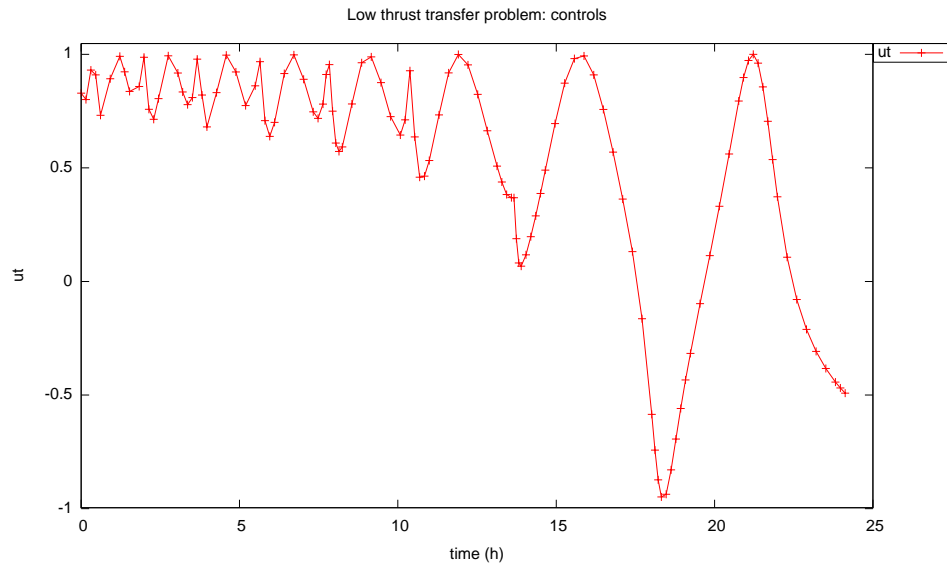
Figure 3.57: Tangential component of the thrust direction vector, u_t
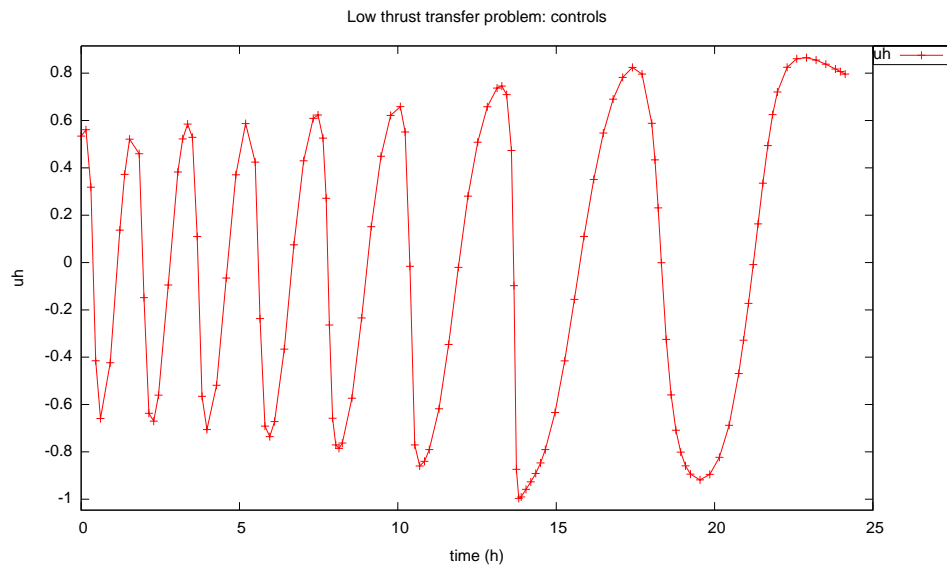
Figure 3.58: Normal component of the thrust direction vector, u_h

3.24 Manutec R3 robot

The DLR model 2 of the Manutec r3 robot, reported and validated by Otter and co-workers [32, 20], describes the motion of three links of the robot as a function of the control input signals of the robot drive:

$$\mathbf{M}(\mathbf{q}(t))\ddot{\mathbf{q}}(t) = \mathbf{V}(\mathbf{q}(t), \dot{\mathbf{q}}(t)) + \mathbf{G}(\mathbf{q}(t)) + \mathbf{D}\mathbf{u}(t)$$

where $\mathbf{q} = [q_1(t), q_2(t), q_3(t)]^T$ is the vector of relative angles between the links, the normalized torque controls are $\mathbf{u}(t) = [u_1(t), u_2(t), u_3(t)]^T$, $\mathbf{D}$ is a diagonal matrix with constant values, $\mathbf{M}(\mathbf{q})$ is a symmetric inertia matrix, $\mathbf{V}(\mathbf{q}(t), \dot{\mathbf{q}}(t))$ are the torques caused by coriolis and centrifugal forces, $\mathbf{G}(\mathbf{q}(t))$ are gravitational torques. The model is described in detail in [32] and is fully included in the code for this example[1].

The example reported here consists of a minimum energy point to point trajectory, so that the objective is to find t_f and $\mathbf{u}(t) = [u_1(t), u_2(t), u_3(t)]^T$, $t \in [0, t_f]$ to minimise:

$$J = \int_0^{t_f} \mathbf{u}(t)^T \mathbf{u}(t) \mathrm{d}t \tag{3.101}$$

The boundary conditions associated with the problem are:

$$\begin{aligned}
\mathbf{q}(0) &= \begin{bmatrix} 0 & -1.5 & 0 \end{bmatrix}^T \\
\dot{\mathbf{q}}(0) &= \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T \\
\mathbf{q}(t_f) &= \begin{bmatrix} 1.0 & -1.95 & 1.0 \end{bmatrix}^T \\
\dot{\mathbf{q}}(t_f) &= \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T
\end{aligned} \tag{3.102}$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
///////////////          manutec.cxx              /////////////////////
//////////////////////////////////////////////////////////////////////
///////////////          PSOPT example            /////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:        Minimum time control of a Manutec R3 robot //////
//////// Last modified: 25 January 2010                 ///////////////
//////// Reference: Chettibi et al (2007)               ///////////////
////////                                                ///////////////
//////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009      ///////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////
//////// is distributed under the terms of the GNU Lesser ////////////
//////// General Public License (LGPL)                  ///////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"
```

---

[1]Dr. Martin Otter from DLR, Germany, has kindly authorised the author to publish a translated form of subroutine R3M2SI as part of the $\mathcal{PSOPT}$ distribution.

```
// Set option to:  1 for minimum time problem
//                 2 for minimum time with regularization
//                 3 for minimum energy problem
#define OBJ_OPTION 3


typedef struct {
 double   g; // Gravity constant
 double   pl; // Point mass of load;
 double   Ia1; // Moment of inertia of arm 1, element (3,3)
 DMatrix* Fc; // Voltage-force constant of motor
 DMatrix* r; // Gear ratio of motor
 DMatrix* Im; // Moment of inertia of motor
 DMatrix* m; // Mass of arm 2 and 3
 DMatrix* L;    // Length of arm 2 and 3 (inc. tool)
 DMatrix* com; // Center of mass coordinates of arm 2 and 3;
 DMatrix* Ia; // Moment of inertia arm 2 and 3
} CONSTANTS_;

CONSTANTS_ CONSTANTS;

typedef struct {

adouble*   qdd_ad;
adouble*   F_ad;
adouble*   qd_ad;
adouble*   q_ad;

adouble* kw;
adouble* beta;
adouble* afor;
adouble* wabs;
adouble* zeta;
adouble* cosp;
adouble* ator;
adouble* sinp;
adouble* rhicm;
adouble* genvd ;
adouble* mrel12 ;
adouble* mrel22;
adouble* rhrel2;
adouble* workm3;
adouble* workm6;
adouble* works1;
adouble* works2;
adouble* workv3;
adouble* workv6;
adouble* cmhges;
adouble* ihhges;
adouble* inertc;
adouble* inertg;
adouble* inerth;
adouble* workam;
adouble* workas;
adouble* wworkm;
adouble* wworkv;

} VARS_;

VARS_ VARS;

void r3m2si(double *ml, adouble *aq, adouble *aqd,
adouble *fgen, adouble *aqdd, VARS_ *vars);

//////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////


adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   adouble retval;

   if (OBJ_OPTION==1 || OBJ_OPTION==2 ) retval = tf;
   else retval = 0.0;

   return retval;
}
```

254

```
///////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
///////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{

    adouble retval;

    adouble* u = controls;

    adouble* qdot = &states[3];

    double rho;


    if (OBJ_OPTION==1)  rho = 0.0;
    else if (OBJ_OPTION==2) rho = 1.e-5;
    else if (OBJ_OPTION==3) rho = 1.0;


    retval = rho*dot( u, u, 3 );


    return (retval);
}


///////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ///////////////////////////////////
///////////////////////////////////////////////////////////////////////



void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

 int i;
 double g    =  CONSTANTS.g;
 DMatrix& Fc = *CONSTANTS.Fc;
 DMatrix& r  = *CONSTANTS.r;
 DMatrix& Im = *CONSTANTS.Im;
 DMatrix& m  = *CONSTANTS.m;
 double pl   =  CONSTANTS.pl;
 DMatrix& L  = *CONSTANTS.L;
 DMatrix& com= *CONSTANTS.com;
 double   Ia1=  CONSTANTS.Ia1;
 DMatrix& Ia = *CONSTANTS.Ia;


adouble* F = VARS.F_ad;
adouble* qdd = VARS.qdd_ad;
adouble* qd = VARS.qd_ad;
adouble* q = VARS.q_ad;
double ml = pl;

for(i=0;i<3;i++) {
    q[i]  = states[i];
    qd[i] = states[3+i];
    F[i] = Fc(i+1)*controls[i];
}

r3m2si(&ml, q, qd, F, qdd, &VARS);

derivatives[CINDEX(1)] =  qd[CINDEX(1)];
derivatives[CINDEX(2)] =  qd[CINDEX(2)];
derivatives[CINDEX(3)] =  qd[CINDEX(3)];

derivatives[CINDEX(4)]  =  qdd[CINDEX(1)];
derivatives[CINDEX(5)]  =  qdd[CINDEX(2)];
derivatives[CINDEX(6)]  =  qdd[CINDEX(3)];


}


///////////////////////////////////////////////////////////////////////
//////////////////  Define the events function /////////////////////////
///////////////////////////////////////////////////////////////////////
```

```
void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{


    int i;

    for (i=0; i< 6; i++ ) {
       e[ i ]    =  initial_states[ i ];
    }

    for (i=0; i< 6; i++ ) {
       e[6 + i ] =  final_states[ i ];
    }


}



///////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function //////////////////
///////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{

    // Single phase problem

}



///////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine /////////////////////////////
///////////////////////////////////////////////////////////////////////

int main(void)
{
 double g = 9.81;                                 // Gravity constant [m/s2]
 double pl= 0.0;                                  // Point mass of load; [kg]
 double   Ia1 = 1.16;                             // Moment of inertia of arm 1, element (3,3) [kgm^2]
 DMatrix Fc(3,1); Fc = "[-126.0;252.0; 72.0]";    // Voltage-force constant of motor [N*m/V]
 DMatrix r(3,1); r = "[-105; 210; 60]";           // Gear ratio of motor
 DMatrix Im(3,1); Im = "[1.3e-3; 1.3e-3; 1.3e-3]";    // Moment of inertia of motor [kg*m^2]
 DMatrix m(2,1);  m = "[56.5; 60.3]";             // Mass of arm 2 and 3 [kg]
 DMatrix L(2,1);  L = "[0.5; 0.98]";              // Length of arm 2 and 3 (inc. tool) [m]
 DMatrix  com(2,2);                               // Center of mass coordinates of arm 2 and 3; [m]
 DMatrix Ia(4,2);                                 // Moment of inertia arm 2 and 3 [kg*m^2]

 com(1,1)=0.172; com(1,2)=0.028; com(2,1)=0.205; com(2,2)=0.202;

 Ia(1,1)=2.58; Ia(1,2)=11.0;
 Ia(2,1)=2.73; Ia(2,2)=8.0;
 Ia(3,1)=0.64; Ia(3,2)=0.80;
 Ia(4,1)=-0.46; Ia(4,2)=0.50;

 Fc.Print("Fc");
 r.Print("r");
 Im.Print("Im");
 m.Print("m");
 L.Print("L");
 com.Print("com");
 Ia.Print("Ia");


 CONSTANTS.g    = g;
 CONSTANTS.pl   = pl;
 CONSTANTS.Ia1 = Ia1;
 CONSTANTS.Fc = &Fc;
 CONSTANTS.r  = &r;
 CONSTANTS.Im = &Im;
 CONSTANTS.m  = &m;
 CONSTANTS.L  = &L;
 CONSTANTS.com= &com;
 CONSTANTS.Ia = &Ia;


 VARS.qdd_ad = new adouble[3];
 VARS.qd_ad  = new adouble[3];
```

```
    VARS.q_ad    = new adouble[3];
    VARS.F_ad    = new adouble[3];



    VARS.kw = new adouble[9];
    VARS.beta = new adouble[18];
    VARS.afor = new adouble[9];
    VARS.wabs = new adouble[9];
    VARS.zeta = new adouble[18];
    VARS.cosp = new adouble[3];
    VARS.ator = new adouble[9];
    VARS.sinp = new adouble[3];
    VARS.rhicm = new adouble[9];
    VARS.genvd = new adouble[3];
    VARS.mrel12 = new adouble[18];
    VARS.mrel22 = new adouble[3];
    VARS.rhrel2 = new adouble[3];
    VARS.workm3 = new adouble[9];
    VARS.workm6 = new adouble[180];
    VARS.works1 = new adouble[12];
    VARS.works2 = new adouble[6];
    VARS.workv3 = new adouble[39];
    VARS.workv6 = new adouble[24];
    VARS.cmhges = new adouble[3];
    VARS.ihhges = new adouble[9];
    VARS.inertc = new adouble[27];
    VARS.inertg = new adouble[108];
    VARS.inerth = new adouble[27];
    VARS.workam = new adouble[144];
    VARS.workas = new adouble[40];
    VARS.wworkm = new adouble[54];
    VARS.wworkv = new adouble[9];


///////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures /////////////////////////////
///////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

///////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  //////////////////////////////
///////////////////////////////////////////////////////////////////////

    problem.name          = "Manutec R3 robot problem";
    problem.outfilename                  = "manutec.txt";

///////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
///////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                = 0;

    psopt_level1_setup(problem);

///////////////////////////////////////////////////////////////////////
/////////    Define phase related information & do level 2 setup  ////////////
///////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 6;
    problem.phases(1).ncontrols  = 3;
    problem.phases(1).nevents    = 12;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes                    = "[20, 30, 40, 60, 80]";


    psopt_level2_setup(problem, algorithm);



///////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information //////////////////////
///////////////////////////////////////////////////////////////////////

    DMatrix qmax(1,3), qdmax(1,3), qddmax(1,3);

    // Joint position limits in rad

    qmax(1) = 2.97;    qmax(2) = 2.01; qmax(3) = 2.86;
```

```
    // Joint angular velocity limits in rad/s

    qdmax(1) = 3.0;   qdmax(2) = 1.5; qdmax(3) = 5.2;


    problem.phases(1).bounds.lower.states =  (-qmax || -qdmax);
    problem.phases(1).bounds.upper.states =  ( qmax ||  qdmax);

    DMatrix umax(1,3);

    // Control variable limits in V

    umax(1) = 7.5; umax(2) = 7.5; umax(3) = 7.5;


    problem.phases(1).bounds.lower.controls = -umax;

    problem.phases(1).bounds.upper.controls =  umax;


    DMatrix qi(1,3), qf(1,3), qdi(1,3), qdf(1,3);

    // Initial joint positions in rad
    qi(1) = 0.0;    qi(2) = -1.5; qi(3) = 0.0;

    // Final joint positions in rad
    qf(1) = 1.0; qf(2) = -1.95;  qf(3) = 1.0;

    // Initial joint velocities in rad/s

    qdi = zeros(1,3);

    // Final joint velocities in rad/s

    qdf = zeros(1,3);


    problem.phases(1).bounds.lower.events = (qi || qdi || qf || qdf);
    problem.phases(1).bounds.upper.events = (qi || qdi || qf || qdf);


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    if (OBJ_OPTION==1 || OBJ_OPTION==2) {
      problem.phases(1).bounds.lower.EndTime      = 0.0;
      problem.phases(1).bounds.upper.EndTime      = 1.0;
    }
    else {
      problem.phases(1).bounds.lower.EndTime      = 0.53;
      problem.phases(1).bounds.upper.EndTime      = 0.53;
    }



////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae             = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess /////////////////////
////////////////////////////////////////////////////////////////////////

    int nnodes      = problem.phases(1).nodes(1);
    int ncontrols                   = problem.phases(1).ncontrols;
    int nstates                     = problem.phases(1).nstates;

    DMatrix state_guess    = zeros(nstates,nnodes);
    DMatrix control_guess  = zeros(ncontrols,nnodes);
    DMatrix time_guess     = linspace(0.0,0.53,nnodes);


    state_guess(1,colon()) = linspace( qi(1), qf(1), nnodes );
```

```
    state_guess(2,colon()) = linspace( qi(2), qf(2), nnodes );
    state_guess(3,colon()) = linspace( qi(3), qf(3), nnodes );
    state_guess(4,colon()) = linspace( qi(4), qf(4), nnodes );
    state_guess(5,colon()) = linspace( qi(5), qf(5), nnodes );
    state_guess(6,colon()) = linspace( qi(6), qf(6), nnodes );


    problem.phases(1).guess.states   = state_guess;
    problem.phases(1).guess.controls = control_guess;
    problem.phases(1).guess.time     = time_guess;



////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ////////////////////////////
////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max           = 1000;
    algorithm.nlp_tolerance          = 1.e-6;
    algorithm.nlp_method             = "IPOPT";
    algorithm.scaling                = "automatic";
    algorithm.derivatives            = "automatic";
    algorithm.mesh_refinement        = "automatic";
    algorithm.ode_tolerance          = 1.e-5;
    algorithm.mr_max_iterations       = 5;


////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   ////////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   ///////
////////////////////////////////////////////////////////////////////////


    DMatrix x, u, t;

    x        = solution.get_states_in_phase(1);
    u        = solution.get_controls_in_phase(1);
    t        = solution.get_time_in_phase(1);



////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired /////////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

    DMatrix q = x( colon(1,3), colon() );
    DMatrix qd= x( colon(4,6), colon() );


////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ////////////
////////////////////////////////////////////////////////////////////////


    plot(t,q,problem.name+": positions","t", "q (rad)", "q1 q2 q3");

    plot(t,qd,problem.name+": velocities","t", "qdot (rad/s)", "qd1 qd2 qd3");

    plot(t,u(1,colon()),problem.name+": control u1","time (s)", "control", "u1");

    plot(t,u(2,colon()),problem.name+": control u2","time (s)", "control", "u2");

    plot(t,u(3,colon()),problem.name+": control u3","time (s)", "control", "u3");

    plot(t,q,problem.name+": positions","t", "q (rad)", "q1 q2 q3", "pdf",
                 "positions.pdf");

    plot(t,qd,problem.name+": velocities","t", "qdot (rad/s)", "qd1 qd2 qd3", "pdf",
                 "velocities.pdf");

    plot(t,u,problem.name+": controls","time (s)", "controls", "u1 u2 u3",
```

```
 "pdf", "controls.pdf");


}


 void r3m2si(double *ml, adouble *aq, adouble *aqd,
             adouble *fgen, adouble *aqdd, VARS_ *vars)
{

adouble mhges;

adouble* kw = vars->kw;
adouble* beta = vars->beta;
adouble* afor = vars->afor;
adouble* wabs = vars->wabs;
adouble* zeta = vars->zeta;
adouble* cosp = vars->cosp;
adouble* ator = vars->ator;
adouble* sinp = vars->sinp;
adouble* rhicm = vars->rhicm;
adouble* genvd = vars->genvd;
adouble* mrel12 = vars->mrel12;
adouble* mrel22 = vars->mrel22;
adouble* rhrel2 = vars->rhrel2;
adouble* workm3 = vars->workm3;
adouble* workm6 = vars->workm6;
adouble* works1 = vars->works1;
adouble* works2 = vars->works2;
adouble* workv3 = vars->workv3;
adouble* workv6 = vars->workv6;
adouble* cmhges = vars->cmhges;
adouble* ihhges = vars->ihhges;
adouble* inertc = vars->inertc;
adouble* inertg = vars->inertg;
adouble* inerth = vars->inerth;
adouble* workam = vars->workam;
adouble* workas = vars->workas;
adouble* wworkm = vars->wworkm;
adouble* wworkv = vars->wworkv;




/* Simulation model equations of the Manutec r3 robot (DFVLR model 2) */
/* This is a modified C++ tranlation of the original Fortran subroutine R3M2SI */
/* by Otter and co-workers, published here with kind persmission from Dr. Martin Otter */
/* DSL, Germany */

/* Procedure purpose: */
/*   This subroutine calculates the generalized accelerations (AQDD) for */
/*   the Manutec r3 robot of the DFVLR model 2. This model is based on */
/*   the following assumptions: */

/*   - The last 3 joints (joints 4,5,6) of the robot hand do not move. */
/*     This corresponds to the case that the brakes in these joints are */
/*     blocked (joints 4,5,6 are taken into account in model 1). */

/*   - The robot consists of a base body (the environment), three arms */
/*     and three rotors. The rotors represent the inertia effects of the */
/*     motors and of the wheels of the gear boxes. The rotors are */
/*     embedded in the preceeding arms, e.g. the rotor 2, which drives */
/*     arm2, is embedded in arm1. */

/*   - Arms and rotors are considered to be rigid bodies. */

/*   - Elasticity, backlash and friction in the gear boxes are neglected. */

/*   - The motors are modelled as ideal voltage-torque converters */
/*     without any dynamic effects. As input arguments of the subroutine */
/*     the torques at the gear output (FGEN in <Nm>) must be given. If the */
/*     voltage (U in <V>) at the input of the current regulator of the */
/*     motor is given, FGEN must be calculated (before calling this */
/*     subroutine) in the following way: */
/*         FGEN(1) = -126.0*U(1) */
/*         FGEN(2) =  252.0*U(2) */
/*         FGEN(3) =   72.0*U(3) */

/*   - At the robot's tip a load mass (ML) is attached. It can range */
/*     between 0 ... 15 kg. */

/*   Given the actual value ML of the load mass, the joint angles AQ(i), */
/*   the derivatives of the joint angles AQD(i), and the driving torques */
/*   in the joints FGEN(i), the second derivatives of the joint angles */
```

```
/*   AQDD(i) are calculated by this subroutine. */

/* Usage: */
/*   CALL  R3M2SI (ML, AQ, AQD, FGEN, AQDD) */

/*   ML   :  IN, DOUBLE, <kg>, 0<=ML<=15 */
/*           Load mass. */
/*   AQ   :  IN, DOUBLE(3), <rad>, -2.97 <= AQ(1) <= 2.97  (+/- 170 deg), */
/*                                 -2.01 <= AQ(2) <= 2.01  (+/- 115 deg), */
/*                                 -2.86 <= AQ(3) <= 2.86  (+/- 164 deg) */
/*           (The given limits are hardware constraints by limit switches). */
/*           Vector of generalized coordinates (relative angles between */
/*           two contigous robot arms). */
/*           AQ(1) is not used in this subroutine, because it is not */
/*           needed to calculate AQDD. */
/*   AQD  :  IN, DOUBLE(3), <rad/sec>, */
/*                                 -3.0 <= AQD(1) <= 3.0  (+/- 172 deg/sec), */
/*                                 -1.5 <= AQD(2) <= 1.5  (+/-  86 deg/sec), */
/*                                 -5.2 <= AQD(3) <= 5.2  (+/- 298 deg/sec) */
/*           Derivative of AQ. */
/*   FGEN :  IN, DOUBLE(3), <Nm>,    -945.0 <= FGEN(1) <=  945.0, */
/*                                  -1890.0 <= FGEN(2) <= 1890.0, */
/*                                   -540.0 <= FGEN(3) <=  540.0 */
/*           Torque at the gear output. */
/*   AQDD :  OUT, DOUBLE(3), <rad/sec**2> */
/*           Second derivative of AQ. */

/* Bibliography: */
/*   A detailed description of the model, together with the body-data */
/*   (mass, center of mass etc. of the arms and rotors) is given in */
/*     Otter M., Tuerk S., Mathematical Model of the Manutec r3 Robot, */
/*           (DFVLR Model No. 2), DFVLR - Oberpfaffenhofen, Institut fuer */
/*           Dynamik der Flugsysteme, D-8031 Wessling, West Germany, */
/*           corrected version april 1988. */

/*   This subroutine was generated by the program MYROBOT. See */
/*     Otter M., Schlegel S., Symbolic generation of efficient simulation */
/*           codes for robots. 2nd European Simulation Multiconference, */
/*           June 1-3, 1988, Nice. */

/*   The underlying multibody algorithm is a modified version of */
/*     Brandl H., Johanni R., Otter M., A very efficient algorithm for the */
/*           simulation of robots and similar multibody systems without */
/*           inversion of the mass matrix. IFAC/IFIP/IMACS International */
/*           Symposium on Theory of Robots, december 3-5, 1986, Vienna. */

/* Remarks: */
/*   - The limits given for the input variables are not checked in */
/*     this subroutine. The user is responsible for proper data. */

/*   - If a SINGLE PRECISION version of this subroutine is desired, just */
/*     change all strings from DOUBLE PRECISION to REAL in the declaration */
/*     part. */

/* Copyright: */
/*   1988  DFVLR - Institut fuer Dynamik der Flugsysteme */

/* Life cycle: */
/*   1988 APR  M. Otter, S. Tuerk (DFVLR)              : specified. */
/*   1988 APR  M. Otter (DFVLR)                        : generated. */
/*   1988 APR  M. Otter, C. Schlegel, S. Tuerk (DFVLR): tested. */
/* ------------------------------------------------------------------- */

/* Statistical information (MySymbol-Version 1.2) */
/* ============================================== */

/* 1. Number of Operations: */
/* ------------------------ */

/*                          + - | * | / | sin | cos | sqrt| */
/*                         -----|-----|-----|-----|-----|-----| */
/* Without simplifications  3180| 3702| 26|  3|  3|    0| */
/*                         -----|-----|-----|-----|-----|-----| */
/* Terms simplified          222| 247| 23|  3|  3|    0| */
/*                         -----|-----|-----|-----|-----|-----| */
/* Unnecessary statements      |     |     |     |     |     | */
/* removed                   140| 159| 17|  2|  2|    0| */


/* 2. Storage information: */
/* ----------------------- */

/*                  INTEGER |         | */
```

261

```
/*                          words |         |  */
/*                         --------|---------|  */
/*  available storage      100000 | 100.0 % |  */
/*                         --------|---------|  */
/*  array-elements           1412 |   1.4 % |  */
/*                         --------|---------|  */
/*  double-numbers            104 |   0.1 % |  */
/*                         --------|---------|  */
/*  statement buffer         2552 |   2.6 % |  */
/*  +++++++++++++++++++++++++++++++++++++++++  */
/*  used storage             4068 |   4.1 % |  */
/*                         --------|---------|  */
/*  free storage            95932 |  95.9 % |  */

/* ----------------------------------------------------------------------- */
/* cccccccccccccccccccccccc Procedural section ccccccccccccccccccccccccccccc */
    /* Parameter adjustments */
    --aqdd;
    --fgen;
    --aqd;
    --aq;

    /* Function Body */
    workv3[14] = *ml * .98;
    workv3[17] = workv3[14] + 12.1806;
    mhges = *ml + 60.3;
    cmhges[0] = 1.6884 / mhges;
    cmhges[2] = workv3[17] / mhges;
    workv3[20] = workv3[14] * .98;
    ihhges[0] = workv3[20] + 7.8704812;
    ihhges[4] = workv3[20] + 8.1077564;
    sinp[1] = sin(aq[2]);
    cosp[1] = cos(aq[2]);
    wworkv[0] = mhges * cmhges[0];
    wworkv[2] = mhges * cmhges[2];
    wworkv[3] = cmhges[0] * wworkv[0];
    wworkv[5] = cmhges[2] * wworkv[2];
    wworkv[6] = wworkv[3] + wworkv[5];
    wworkm[0] = wworkv[6] - wworkv[3];
    wworkm[8] = wworkv[6] - wworkv[5];
    wworkm[6] = -wworkv[0] * cmhges[2];
    inertc[18] = ihhges[0] - wworkm[0];
    inertc[22] = ihhges[4] - wworkv[6];
    inertc[24] = -.0110568 - wworkm[6];
    inertc[26] = .4372752 - wworkm[8];
    sinp[2] = sin(aq[3]);
    cosp[2] = cos(aq[3]);
/* ** The equations of motion have been generated with the algorithm */
/* ** of Brandl/Johanni/Otter (variant 6) */
/* ** Forward recursion ------------------------------------------------ */
/* -- Quantities of body 1 */
/* -- Quantities of body 2 */
    wabs[4] = sinp[1] * aqd[1];
    wabs[5] = cosp[1] * aqd[1];
    workv3[1] = aqd[1] * aqd[2];
    zeta[7] = cosp[1] * workv3[1];
    zeta[8] = -sinp[1] * workv3[1];
    kw[3] = aqd[2] * 4.9544125 - wabs[5] * 2.45219;
    kw[4] = wabs[4] * 6.7759085;
    kw[5] = aqd[2] * -2.45219 + wabs[5] * 2.311496;
/* -- Quantities of body 3 */
    workv3[19] = -sinp[2] * cmhges[2];
    workv3[20] = cosp[2] * cmhges[2];
    workm3[4] = cosp[2] * inertc[22];
    workm3[5] = sinp[2] * inertc[22];
    workm3[7] = -sinp[2] * inertc[26];
    workm3[8] = cosp[2] * inertc[26];
    inertc[21] = -inertc[24] * sinp[2];
    inertc[22] = workm3[4] * cosp[2] - workm3[7] * sinp[2];
    inertc[24] *= cosp[2];
    inertc[25] = workm3[4] * sinp[2] + workm3[7] * cosp[2];
    inertc[26] = workm3[5] * sinp[2] + workm3[8] * cosp[2];
    workv3[21] = mhges * cmhges[0];
    workv3[22] = mhges * workv3[19];
    workv3[23] = mhges * workv3[20];
    workv3[24] = cmhges[0] * workv3[21];
    workv3[25] = workv3[19] * workv3[22];
    workv3[26] = workv3[20] * workv3[23];
    workv3[27] = workv3[24] + workv3[25] + workv3[26];
    inerth[18] = workv3[27] - workv3[24];
    inerth[22] = workv3[27] - workv3[25];
    inerth[26] = workv3[27] - workv3[26];
    inerth[21] = -workv3[21] * workv3[19];
```

262

```
    inerth[24] = -workv3[21] * workv3[20];
    inerth[25] = -workv3[22] * workv3[20];
    inerth[18] += inertc[18];
    inerth[21] += inertc[21];
    inerth[22] += inertc[22];
    inerth[24] += inertc[24];
    inerth[25] += inertc[25];
    inerth[26] += inertc[26];
    wabs[6] = aqd[3] + aqd[2];
    workv3[1] = wabs[5] * aqd[3];
    workv3[2] = -wabs[4] * aqd[3];
    workv3[6] = wabs[4] * .5;
    workv3[7] = -aqd[2] * .5;
    workv3[9] = -wabs[5] * workv3[7];
    workv3[10] = wabs[5] * workv3[6];
    workv3[11] = aqd[2] * workv3[7] - wabs[4] * workv3[6];
    rhicm[6] = mhges * cmhges[0];
    rhicm[7] = mhges * workv3[19];
    rhicm[8] = mhges * workv3[20];
    kw[6] = inerth[18] * wabs[6] + inerth[21] * wabs[4] + inerth[24] * wabs[5]
    ;
    kw[7] = inerth[21] * wabs[6] + inerth[22] * wabs[4] + inerth[25] * wabs[5]
    ;
    kw[8] = inerth[24] * wabs[6] + inerth[25] * wabs[4] + inerth[26] * wabs[5]
    ;
    ator[3] = -wabs[4] * kw[5] + wabs[5] * kw[4];
    ator[4] = -wabs[5] * kw[3] + aqd[2] * kw[5];
    ator[5] = -aqd[2] * kw[4] + wabs[4] * kw[3];
    ator[6] = -wabs[4] * kw[8] + wabs[5] * kw[7];
    ator[7] = -wabs[5] * kw[6] + wabs[6] * kw[8];
    ator[8] = -wabs[6] * kw[7] + wabs[4] * kw[6];
    workv3[15] = wabs[4] * rhicm[8] - wabs[5] * rhicm[7];
    workv3[16] = wabs[5] * rhicm[6] - wabs[6] * rhicm[8];
    workv3[17] = wabs[6] * rhicm[7] - wabs[4] * rhicm[6];
    afor[6] = -wabs[4] * workv3[17] + wabs[5] * workv3[16];
    afor[7] = -wabs[5] * workv3[15] + wabs[6] * workv3[17];
/* ** Backward recursion --------------------------------------------- */
/*  -- Quantities of body 3 */
    mrel22[2] = inerth[18] + 4.68;
    workv6[0] = ator[6] - inerth[21] * workv3[1] - inerth[24] * workv3[2] +
    rhicm[8] * workv3[10] - rhicm[7] * workv3[11];
    workv6[1] = ator[7] - inerth[22] * workv3[1] - inerth[25] * workv3[2] -
    rhicm[8] * workv3[9] + rhicm[6] * workv3[11];
    workv6[2] = ator[8] - inerth[25] * workv3[1] - inerth[26] * workv3[2] +
    rhicm[7] * workv3[9] - rhicm[6] * workv3[10];
    workv6[3] = afor[6] - rhicm[8] * workv3[1] + rhicm[7] * workv3[2] - mhges
    * workv3[9];
    workv6[4] = afor[7] - rhicm[6] * workv3[2] - mhges * workv3[10];
    rhrel2[2] = fgen[3] + workv6[0];
    mrel12[12] = inerth[18] + .078 + rhicm[8] * .5;
    workv6[18] = workv6[0] - workv6[4] * .5;
    workv6[19] = workv6[1] + workv6[3] * .5;
    workm6[108] = inerth[18] - rhicm[8] * -.5;
    workm6[112] = -rhicm[8] + mhges * -.5;
    workm6[115] = inerth[22] + rhicm[8] * .5;
    workm6[117] = rhicm[8] + mhges * .5;
    inertg[36] = workm6[108] + 4.9544125 - workm6[112] * .5;
    inertg[43] = workm6[115] + 6.7759085 + workm6[117] * .5;
    inertg[48] = inerth[24] - 2.45219 - rhicm[6] * .5;
    inertg[49] = inerth[25] - rhicm[7] * .5;
    inertg[50] = inerth[26] + 2.311496;
    inertg[60] = -11.5825 - rhicm[8] - mhges * .5;
    inertg[62] = rhicm[6] + 9.718;
    inertg[67] = -9.718 - rhicm[6];
    beta[6] = ator[3] + workv6[18];
    beta[7] = ator[4] + workv6[19];
    beta[8] = ator[5] + workv6[2];
    works2[0] = mrel12[12] / mrel22[2];
    works2[1] = inerth[21] / mrel22[2];
    works2[2] = inerth[24] / mrel22[2];
    works2[4] = -rhicm[8] / mrel22[2];
    works2[5] = rhicm[7] / mrel22[2];
    inertg[36] -= mrel12[12] * works2[0];
    inertg[42] = inerth[21] - mrel12[12] * works2[1];
    inertg[43] -= inerth[21] * works2[1];
    inertg[48] -= mrel12[12] * works2[2];
    inertg[49] -= inerth[21] * works2[2];
    inertg[50] -= inerth[24] * works2[2];
    inertg[60] -= mrel12[12] * works2[4];
    inertg[61] = -inerth[21] * works2[4];
    inertg[62] -= inerth[24] * works2[4];
    inertg[66] = rhicm[7] - mrel12[12] * works2[5];
    inertg[67] -= inerth[21] * works2[5];
```

263

```
    inertg[68] = -inerth[24] * works2[5];
    beta[6] -= works2[0] * rhrel2[2];
    beta[7] -= works2[1] * rhrel2[2];
    beta[8] -= works2[2] * rhrel2[2];
/* -- Quantities of body 2 */
    mrel22[1] = inertg[36] + 57.33;
    workv6[0] = beta[6] - inertg[42] * zeta[7] - inertg[48] * zeta[8];
    workv6[1] = beta[7] - inertg[43] * zeta[7] - inertg[49] * zeta[8];
    workv6[2] = beta[8] - inertg[49] * zeta[7] - inertg[50] * zeta[8];
    rhrel2[1] = fgen[2] + workv6[0];
    works1[8] = sinp[1] * inertg[42] + cosp[1] * inertg[48];
    works1[11] = sinp[1] * inertg[60] + cosp[1] * inertg[66];
    workv6[14] = sinp[1] * workv6[1] + cosp[1] * workv6[2];
    workas[30] = cosp[1] + sinp[1];
    workas[31] = cosp[1] - sinp[1];
    workas[33] = workas[30] * workas[31];
    workas[32] = cosp[1] * sinp[1];
    workas[34] = workas[32] + workas[32];
    workas[0] = inertg[50] + inertg[43];
    workas[1] = inertg[50] - inertg[43];
    workas[4] = workas[0] / 2.;
    workas[5] = workas[1] / 2.;
    workas[8] = workas[33] * workas[5] + workas[34] * inertg[49];
    workam[115] = workas[4] + workas[8];
    workas[20] = inertg[68] + inertg[61];
    workas[21] = inertg[68] - inertg[61];
    workas[24] = workas[20] / 2.;
    workas[25] = workas[21] / 2.;
    workas[23] = -inertg[62] - inertg[67];
    workas[27] = workas[23] / 2.;
    workas[28] = workas[33] * workas[25] - workas[34] * workas[27];
    workam[133] = workas[24] + workas[28];
    inertg[14] = workam[115] + 1.16;
    works2[2] = works1[8] / mrel22[1];
    works2[5] = works1[11] / mrel22[1];
    inertg[14] -= works1[8] * works2[2];
    inertg[32] = workam[133] - works1[8] * works2[5];
    beta[2] = workv6[14] - works2[2] * rhrel2[1];
/* -- Quantities of body 1 */
    mrel22[0] = inertg[14] + 14.3325;
    workv6[2] = beta[2] - inertg[32] * 9.81;
    rhrel2[0] = fgen[1] + workv6[2];
/* ** Forward recursion -------------------------------------------- */
/* -- Quantities of body 1 */
    genvd[0] = rhrel2[0] / mrel22[0];
/* -- Quantities of body 2 */
    rhrel2[1] = rhrel2[1] - works1[8] * genvd[0] - works1[11] * 9.81;
    genvd[1] = rhrel2[1] / mrel22[1];
    workv6[7] = zeta[7] + sinp[1] * genvd[0];
    workv6[8] = zeta[8] + cosp[1] * genvd[0];
    workv6[10] = sinp[1] * 9.81;
    workv6[11] = cosp[1] * 9.81;
/* -- Quantities of body 3 */
    rhrel2[2] = rhrel2[2] - mrel12[12] * genvd[1] - inerth[21] * workv6[7] -
    inerth[24] * workv6[8] + rhicm[8] * workv6[10] - rhicm[7] *
    workv6[11];
    genvd[2] = rhrel2[2] / mrel22[2];
    aqdd[1] = genvd[0];
    aqdd[2] = genvd[1];
    aqdd[3] = genvd[2];
    return;
}


/////////////////////////////////////////////////////////////////////
/////////////////////////    END OF FILE    //////////////////////////
/////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.59, 3.60 and 3.61, which contain the elements of the position vector $\mathbf{q}(t)$, the velocity vector $\dot{\mathbf{q}}(t)$, and the controls $\mathbf{u}(t)$, respectively. The mesh refinement process is described in Table 3.4.
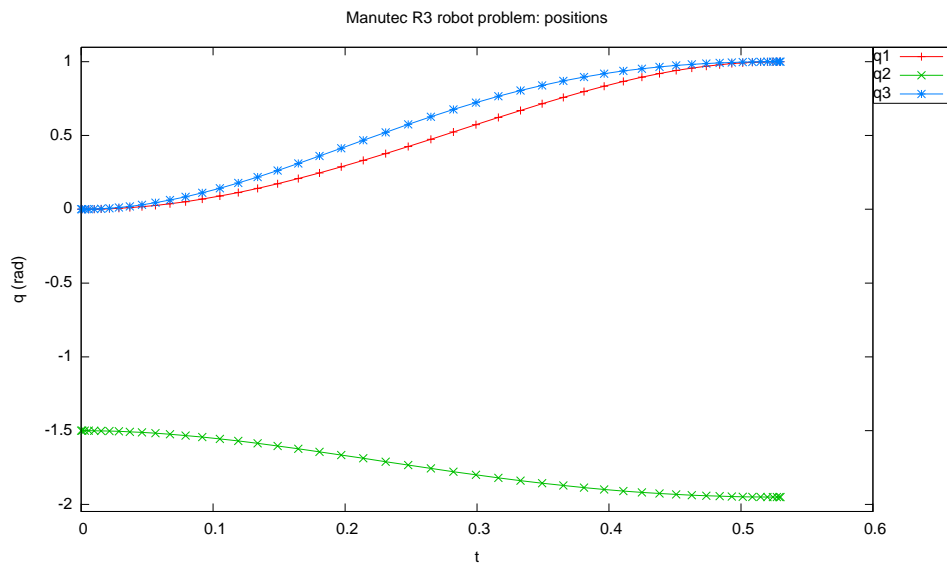
Figure 3.59: States q_1, q_2 and q_3 for the Manutec R3 robot minimum energy problem

```
PSOPT results summary
=====================


Problem:  Manutec R3 robot problem
CPU time (seconds): 4.705465e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  2.040419e+01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  2.040419e+01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.300000e-01
Phase 1 maximum relative local error: 7.660114e-06
NLP solver reports:  The problem solved!
```

3.25   Minimum swing control for a container crane

Consider the following optimal control problem [41], which seeks to minimise the load swing of a container crane, while the load is transferred from one
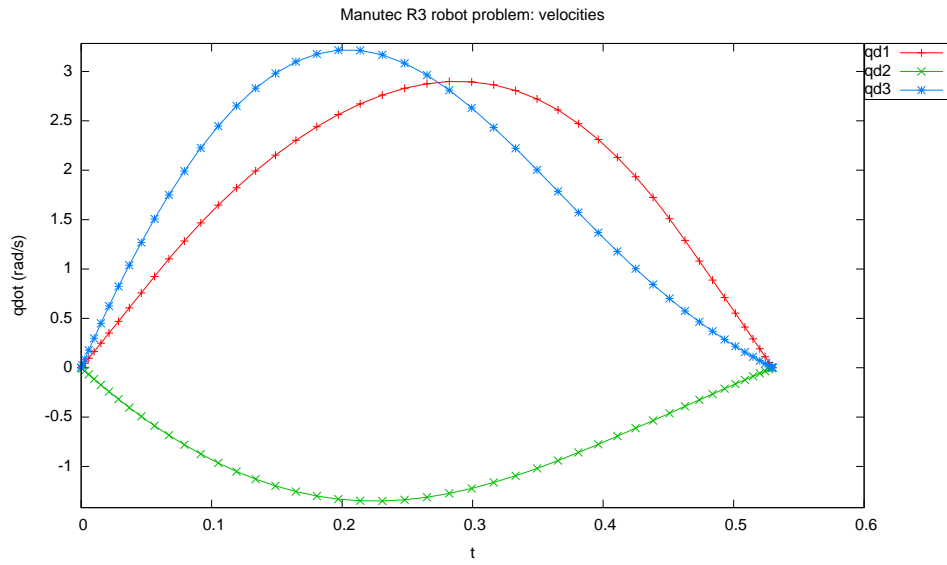
Figure 3.60: States $\dot{q}_1, \dot{q}_2$ and $\dot{q}_3$ for the Manutec R3 robot minimum energy problem
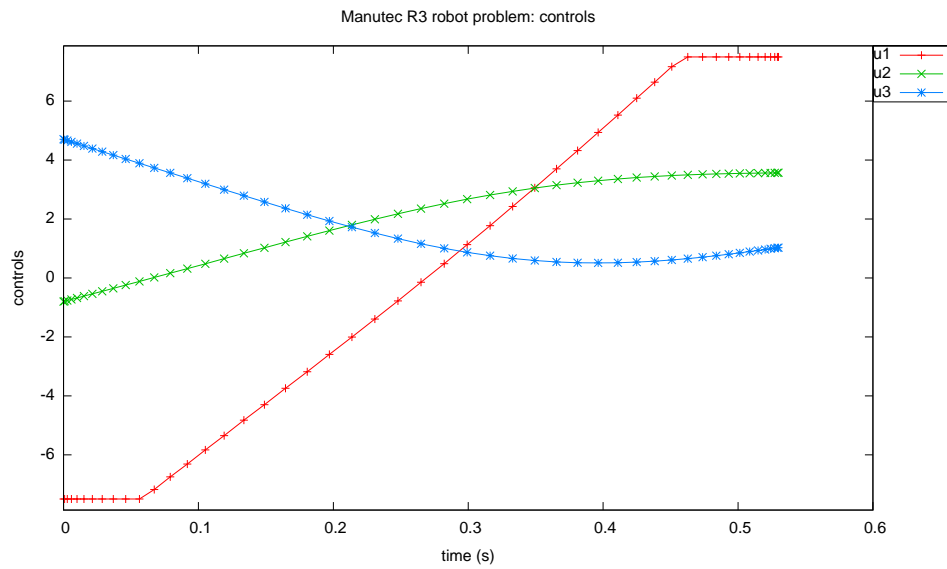
Figure 3.61: Controls u_1, u_2 and u_3 for the Manutec R3 robot minimum energy problem

Table 3.4: Mesh refinement statistics: Manutec R3 robot problem

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	$\epsilon_{\max}$	CPU$_a$
1	LGL-ST	20	182	133	43	43	35	0	860	4.676e-05	3.949e-01
2	LGL-ST	25	227	163	33	34	29	0	850	3.636e-05	4.298e-01
3	LGL-ST	35	317	223	37	38	30	0	1330	2.787e-05	7.294e-01
4	LGL-ST	49	443	307	18	19	17	0	931	7.660e-06	6.760e-01
CPU$_b$	-	-	-	-	-	-	-	-	-	-	2.475e+00
-	-	-	-	-	131	134	111	0	3971	-	4.705e+00

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, $\epsilon_{\max}$ = maximum relative ODE error, CPU$_a$ = CPU time in seconds spent by NLP algorithm, CPU$_b$ = additional CPU time in seconds spent by PSOPT

location to another. Find $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = 4.5 \int_0^{t_f} \left[x_3^2(t) + x_6^2(t) \right] dt \qquad (3.103)$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{x}_1 &= 9x_4 \\
\dot{x}_2 &= 9x_5 \\
\dot{x}_3 &= 9x_6 \\
\dot{x}_4 &= 9(u_1 + 17.2656x_3) \\
\dot{x}_5 &= 9u_2 \\
\dot{x}_6 &= -\frac{9}{x_2}\left[u_1 + 27.0756x_3 + 2x_5x_6 \right]
\end{aligned}
\qquad (3.104)
$$

the boundary conditions

$$
\begin{array}{llll}
x_1(0) &=& 0 & x_1(t_f) &=& 10 \\
x_2(0) &=& 22 & x_2(t_f) &=& 14 \\
x_3(0) &=& 0 & x_3(t_f) &=& 0 \\
x_4(0) &=& 0 & x_4(t_f) &=& 2.5 \\
x_5(0) &=& -1 & x_5(t_f) &=& 0 \\
x_6(0) &=& 0 & x_6(t_f) &=& 0
\end{array}
\qquad (3.105)
$$

and the bounds

$$
\begin{aligned}
-2.83374 &\le u_1(t) \le 2.83374, \\
-0.80865 &\le u_2(t) \le 0.71265, \\
-2.5 &\le x_4(t) \le 2.5, \\
-1 &\le x_5(t) \le 1.
\end{aligned}
\qquad (3.106)
$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
///////////////////////////////////////////////////////////////////////
//////////////////        crane.cxx          ////////////////////////
///////////////////////////////////////////////////////////////////////
//////////////////          PSOPT  Example          ////////////////////
///////////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////////
//////// Title: Minimum Swing Control for Container Crane ////////////////
//////// Last modified:         06 February 2009         ////////////////
//////// Reference:             Teo and Goh              ////////////////
//////// (See PSOPT handbook for full reference)         ////////////////
///////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009      ////////////////
///////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                   ////////////////
///////////////////////////////////////////////////////////////////////

#include "psopt.h"

///////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
///////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                       adouble* parameters,adouble& t0, adouble& tf,
                       adouble* xad, int iphase, Workspace* workspace)
{
    return 0;
}

///////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
///////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    adouble x3 = states[CINDEX(3)];
    adouble x6 = states[CINDEX(6)];
    return  4.5*( pow(x3,2) + pow(x6,2) );
}

///////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ////////////////////////////////////
///////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble xdot, ydot, vdot;

   adouble x1 = states[ CINDEX(1) ];
   adouble x2 = states[ CINDEX(2) ];
   adouble x3 = states[ CINDEX(3) ];
   adouble x4 = states[ CINDEX(4) ];
   adouble x5 = states[ CINDEX(5) ];
   adouble x6 = states[ CINDEX(6) ];

   adouble u1 = controls[ CINDEX(1) ];
   adouble u2 = controls[ CINDEX(2) ];


   derivatives[ CINDEX(1) ] = 9*x4;
   derivatives[ CINDEX(2) ] = 9*x5;
   derivatives[ CINDEX(3) ] = 9*x6;
   derivatives[ CINDEX(4) ] = 9*(u1 + 17.2656*x3);
   derivatives[ CINDEX(5) ] = 9*u2;
   derivatives[ CINDEX(6) ] = -(9/x2)*(u1 + 27.0756*x3 + 2*x5*x6);



}

///////////////////////////////////////////////////////////////////////
//////////////////  Define the events function //////////////////////////
///////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
```

```
              int iphase, Workspace* workspace)
{
   adouble x10 = initial_states[ CINDEX(1) ];
   adouble x20 = initial_states[ CINDEX(2) ];
   adouble x30 = initial_states[ CINDEX(3) ];
   adouble x40 = initial_states[ CINDEX(4) ];
   adouble x50 = initial_states[ CINDEX(5) ];
   adouble x60 = initial_states[ CINDEX(6) ];
   adouble x1f = final_states[ CINDEX(1) ];
   adouble x2f = final_states[ CINDEX(2) ];
   adouble x3f = final_states[ CINDEX(3) ];
   adouble x4f = final_states[ CINDEX(4) ];
   adouble x5f = final_states[ CINDEX(5) ];
   adouble x6f = final_states[ CINDEX(6) ];


   e[ CINDEX(1) ]  = x10;
   e[ CINDEX(2) ]  = x20;
   e[ CINDEX(3) ]  = x30;
   e[ CINDEX(4) ]  = x40;
   e[ CINDEX(5) ]  = x50;
   e[ CINDEX(6) ]  = x60;
   e[ CINDEX(7) ]  = x1f;
   e[ CINDEX(8) ]  = x2f;
   e[ CINDEX(9) ]  = x3f;
   e[ CINDEX(10)]  = x4f;
   e[ CINDEX(11)]  = x5f;
   e[ CINDEX(12)]  = x6f;


}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function //////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}



//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine /////////////////////////////
//////////////////////////////////////////////////////////////////////


int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures /////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  /////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name = "Minimum swing control for a container crane";

    problem.outfilename              = "crane.txt";

//////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                = 0;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
//////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates    = 6;
    problem.phases(1).ncontrols  = 2;
    problem.phases(1).nevents    = 12;
```

269

```
   problem.phases(1).npath      = 0;
   problem.phases(1).nodes    = "[40]";

   psopt_level2_setup(problem, algorithm);


////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information /////////////////////
////////////////////////////////////////////////////////////////////////

   problem.phases(1).bounds.lower.states(1) = -5.0;
   problem.phases(1).bounds.lower.states(2) = -5.0;
   problem.phases(1).bounds.lower.states(3) = -5.0;
   problem.phases(1).bounds.lower.states(4) = -2.5;
   problem.phases(1).bounds.lower.states(5) = -1.0;
   problem.phases(1).bounds.lower.states(6) = -5.0;

   problem.phases(1).bounds.upper.states(1) = 15.0;
   problem.phases(1).bounds.upper.states(2) = 25.0;
   problem.phases(1).bounds.upper.states(3) = 15.0;
   problem.phases(1).bounds.upper.states(4) = 2.5;
   problem.phases(1).bounds.upper.states(5) = 1.0;
   problem.phases(1).bounds.upper.states(6) = 15.0;

   problem.phases(1).bounds.lower.controls(1) = -2.83374;
   problem.phases(1).bounds.lower.controls(2) = -0.80865;
   problem.phases(1).bounds.upper.controls(1) = 2.83374;
   problem.phases(1).bounds.upper.controls(2) = 0.71265;

   // Initial states
   problem.phases(1).bounds.lower.events(1) = 0.0;
   problem.phases(1).bounds.lower.events(2) = 22.0;
   problem.phases(1).bounds.lower.events(3) = 0.0;
   problem.phases(1).bounds.lower.events(4) = 0.0;
   problem.phases(1).bounds.lower.events(5) = -1.0;
   problem.phases(1).bounds.lower.events(6) = 0.0;

   // Final states
   problem.phases(1).bounds.lower.events(7) = 10.0;
   problem.phases(1).bounds.lower.events(8) = 14.0;
   problem.phases(1).bounds.lower.events(9) = 0.0;
   problem.phases(1).bounds.lower.events(10)= 2.5;
   problem.phases(1).bounds.lower.events(11)= 0.0;
   problem.phases(1).bounds.lower.events(12)= 0.0;

   problem.phases(1).bounds.upper.events = problem.phases(1).bounds.lower.events;

   problem.phases(1).bounds.lower.StartTime    = 0.0;
   problem.phases(1).bounds.upper.StartTime    = 0.0;

   problem.phases(1).bounds.lower.EndTime      = 1.0;
   problem.phases(1).bounds.upper.EndTime      = 1.0;


////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  //////////////////////////
////////////////////////////////////////////////////////////////////////


   problem.integrand_cost  = &integrand_cost;
   problem.endpoint_cost = &endpoint_cost;
   problem.dae  = &dae;
   problem.events  = &events;
   problem.linkages = &linkages;



////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess //////////////////////
////////////////////////////////////////////////////////////////////////

   DMatrix x0(6,20);

   x0(1,colon()) = linspace(0.0,10.0, 20);
   x0(2,colon()) = linspace(22.0,14.0, 20);
   x0(3,colon()) = linspace(0.,0., 20);
   x0(4,colon()) = linspace(0.,2.5, 20);
   x0(5,colon()) = linspace(-1.0,0., 20);
   x0(6,colon()) = linspace(0.,0., 20);

   problem.phases(1).guess.controls        = zeros(2, 20);
   problem.phases(1).guess.states          = x0;
   problem.phases(1).guess.time            = linspace(0.0, 1.0, 20); ;
```

```
///////////////////////////////////////////////////////////////////////
////////////////// Enter algorithm options ////////////////////////////
///////////////////////////////////////////////////////////////////////


    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
    algorithm.nlp_iter_max            = 1000;
    algorithm.nlp_tolerance           = 1.e-6;
    algorithm.mesh_refinement         = "automatic";
    algorithm.defect_scaling          = "jacobian-based";

///////////////////////////////////////////////////////////////////////
//////////////////// Now call PSOPT to solve the problem ///////////////
///////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

///////////////////////////////////////////////////////////////////////
///////////// Extract relevant variables from solution structure ///////////
///////////////////////////////////////////////////////////////////////

    DMatrix x  = solution.get_states_in_phase(1);
    DMatrix u  = solution.get_controls_in_phase(1);
    DMatrix t  = solution.get_time_in_phase(1);


///////////////////////////////////////////////////////////////////////
///////////// Save solution data to files if desired ///////////////////////
///////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

    DMatrix x13 = x(colon(1,3), colon() );
    DMatrix x46 = x(colon(4,6), colon() );


///////////////////////////////////////////////////////////////////////
///////////// Plot some results if desired (requires gnuplot) ///////////////
///////////////////////////////////////////////////////////////////////

    plot(t,x13,problem.name + ": states x1, x2 and x3", "time (s)", "states", "x1 x2 x3");

    plot(t,x46,problem.name + ": states x4, x5 and x6", "time (s)", "states", "x4 x5 x6");

    plot(t,u,problem.name + ": controls", "time", "controls", "u1 u2");


    plot(t,x13,problem.name + ": states x1, x2 and x3", "time (s)", "states", "x1 x2 x3",
                        "pdf", "crane_states13.pdf");

    plot(t,x46,problem.name + ": states x4, x5 and x6", "time (s)", "states", "x4 x5 x6",
                        "pdf", "crane_states46.pdf");

    plot(t,u,problem.name + ": controls", "time", "controls", "u1 u2",
                     "pdf", "crane_controls.pdf");


}

///////////////////////////////////////////////////////////////////////
//////////////////////      END OF FILE      ///////////////////////////
///////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.62, 3.63 and 3.64, which contain the elements of the state x_1 to x_3, x_4 to x_6, and the controls, respectively.
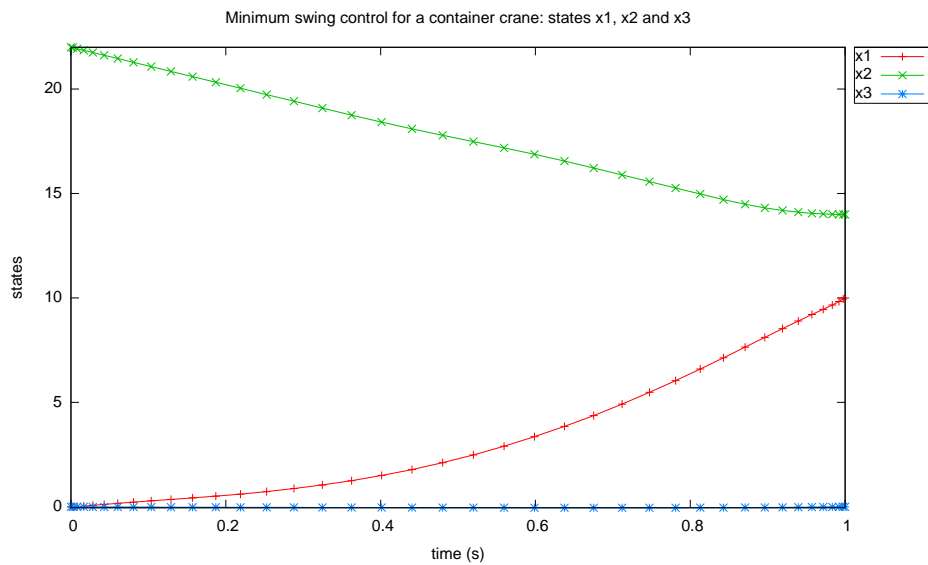
Figure 3.62: States x_1, x_2 and x_3 for minimum swing crane control problem

```
PSOPT results summary
=====================


Problem:  Minimum swing control for a container crane
CPU time (seconds): 4.190000e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  5.152166e-03
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.152166e-03
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 8.917468e-04
NLP solver reports:  The problem solved!
```

3.26 Minimum time to climb for a supersonic aircraft

Consider the following optimal control problem, which finds the minimum time to climb to a given altitude for a supersonic aircraft [4]. Minimize the
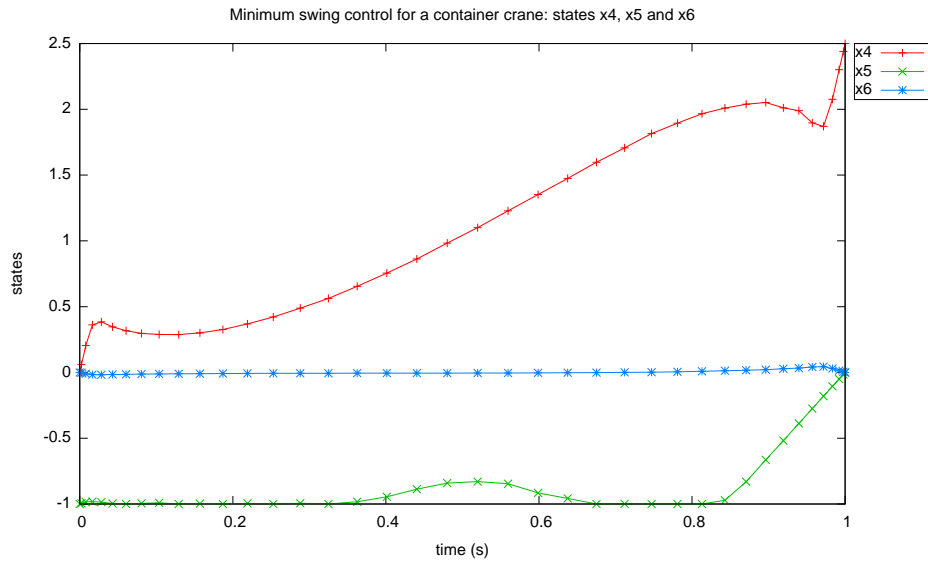
Figure 3.63: States x_4, x_5 and x_6 for minimum swing crane control problem
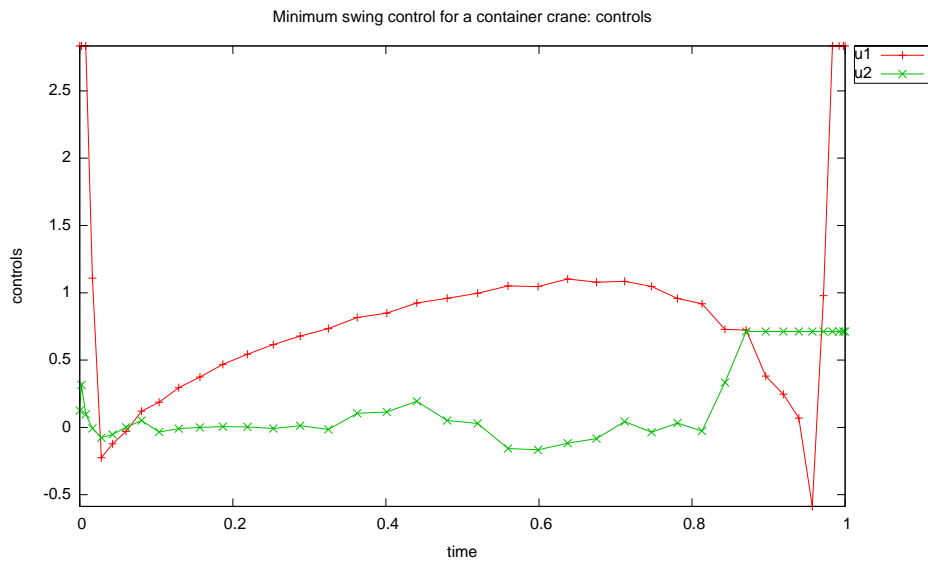
Figure 3.64: Controls for minimum swing crane control problem

cost functional

$$J = t_f \tag{3.107}$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{h} &= v \sin \gamma \\
\dot{v} &= \frac{1}{m} \left[T(M,h) \cos \alpha - D \right] - \frac{\mu}{(R_e+h)^2} \sin \gamma \\
\dot{\gamma} &= \frac{1}{mv} \left[T(M,h) \sin \alpha + L \right] + \cos \gamma \left[\frac{v}{(R_e+h)} - \frac{\mu}{v(R_e+h)^2} \right] \\
\dot{w} &= \frac{-T(M,h)}{I_{sp}}
\end{aligned}
\tag{3.108}
$$

where h is the altitude (ft), v is the velocity (ft/s), γ is the flight path angle (rad), w is the weight (lb), L is the lift force, D is the drag force (lb), T is the thrust (lb), $M = v/c$ is the mach number, $m = w/g_0$ (slug) is the mass, $c(h)$ is the speed of sound (ft/s), R_e is the radious of Earth, and μ is the gravitational constant. The control input α is the angle of attack (rad).

The speed of sound is given by:

$$c = 20.0468\sqrt{\theta} \tag{3.109}$$

where $\theta = \theta(h)$ is the atmospheric temperature (K).

The aerodynamic forces are given by:

$$
\begin{aligned}
D &= \frac{1}{2} C_D S \rho v^2 \\
L &= \frac{1}{2} C_L S \rho v^2
\end{aligned}
\tag{3.110}
$$

where

$$
\begin{aligned}
C_L &= c_{L\alpha}(M)\alpha \\
C_D &= c_{D0}(M) + \eta(M)c_{L\alpha}(M)\alpha^2
\end{aligned}
\tag{3.111}
$$

where C_L and C_D are aerodynamic lift and drag coefficients, S is the aerodynamic reference area of the aircraft, and $\rho = \rho(h)$ is the air density.

The boundary conditions are given by:

$$
\begin{aligned}
h(0) &= 0 \text{ (ft)}, \\
h(t_f) &= 65600.0 \text{ (ft)} \\
v(0) &= 424.260 \text{ (ft/s)}, \\
v(t_f) &= 968.148 \text{ (ft/s)} \\
\gamma(0) &= \gamma(t_f) = 0 \text{ (rad)} \\
w(0) &= 42000.0 \text{ lb}
\end{aligned}
\tag{3.112}
$$

The parameter values are given by:

$$S = 530 \text{ (ft}^2\text{)},$$
$$I_{sp} = 1600.0 \text{ (sec)}$$
$$\mu = 0.14046539 \times 10^{17} \text{ (ft}^3/\text{s}^2\text{)}, \qquad (3.113)$$
$$g_0 = 32.174 \text{ (ft/s}^2\text{)}$$
$$R_e = 20902900 \text{ (ft)}$$

The variables $c_{L\alpha}(M)$, $c_{D0}(M)$, $\eta(M)$ are interpolated from 1-D tabular data which is given in the code and also in [4], using spline interpolation, while the thrust $T(M, h)$ is interpolated from 2-D tabular data given in the code and in [4], using 2D spline interpolation.

The air density ρ and the atmospheric temperature θ were calculated using the US Standard Atmosphere Model 1976[2], based on the standard temperature of 15 (deg C) at zero altitude and the standard air density of 1.22521 (slug/ft^3) at zero altitude.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
//////////////////          climb.cxx           //////////////////////
//////////////////////////////////////////////////////////////////////
/////////////////          PSOPT  Example          ///////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title: Minimum time to climb of a supersonic aircraft ///////////
//////// Last modified: 12 January 2009              ////////////////
//////// Reference:    GPOPS Manual   ///////////////
//////// (See PSOPT handbook for full reference)          ///////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009       ///////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                ////////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"


void atmosphere(adouble* alt,adouble* sigma,adouble* delta,adouble* theta);

//////////////////////////////////////////////////////////////////////
/////////  Declare an auxiliary structure to hold local constants ////////
//////////////////////////////////////////////////////////////////////

struct Constants {
  double g0;
  double S;
  double Re;
  double Isp;
  double mu;
  DMatrix* CLa_table;
  DMatrix* CD0_table;
  DMatrix* eta_table;
  DMatrix* T_table;
  DMatrix* M1;
  DMatrix* M2;
  DMatrix* h1;
  DMatrix* htab;
  DMatrix* ttab;
  DMatrix* ptab;
  DMatrix* gtab;
```

---

[2]see http://www.pdas.com/programs/atmos.f90

```
};

typedef struct Constants Constants_;

static Constants_ CONSTANTS;

void atmosphere_model(adouble* rho, adouble* M, adouble v, adouble h);


//////////////////////////////////////////////////////////////////////
/////////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
return tf;
}

//////////////////////////////////////////////////////////////////////
/////////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)

{
    return  0.0;
}




//////////////////////////////////////////////////////////////////////
/////////////////////  Define the DAE's /////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

  adouble alpha = controls[ CINDEX(1)]; // Angle of attack (rad)


  adouble h     =   states[ CINDEX(1) ]; // Altitude (ft)
  adouble v     =   states[ CINDEX(2) ]; // Velocity (ft/s)
  adouble gamma =   states[ CINDEX(3) ]; // Flight path angle (rad)
  adouble w     =   states[ CINDEX(4) ]; // weight (lb)


  double g0   = CONSTANTS.g0;
  double S    = CONSTANTS.S;
  double Re   = CONSTANTS.Re;
  double Isp  = CONSTANTS.Isp;
  double mu   = CONSTANTS.mu;

  DMatrix& M1          = *CONSTANTS.M1;
  DMatrix& M2          = *CONSTANTS.M2;
  DMatrix& h1          = *CONSTANTS.h1;
  DMatrix& CLa_table   = *CONSTANTS.CLa_table;
  DMatrix& CD0_table   = *CONSTANTS.CD0_table;
  DMatrix& eta_table   = *CONSTANTS.eta_table;
  DMatrix& T_table     = *CONSTANTS.T_table;

  int lM1 = length(M1);

  adouble rho;
  adouble   m = w/g0;
  adouble   M;

  atmosphere_model( &rho, &M, v, h);

  adouble CL_a, CD0, eta, T;


  spline_interpolation( &CL_a, M, M1, CLa_table,  lM1);
  spline_interpolation( &CD0,  M, M1, CD0_table, lM1);
  spline_interpolation( &eta,  M, M1, eta_table, lM1);
  spline_2d_interpolation(&T, M, h, M2, h1, T_table, workspace);

//  smooth_linear_interpolation( &CL_a, M, M1, CLa_table,  lM1);
//  smooth_linear_interpolation( &CD0,  M, M1, CD0_table, lM1);
```

276

```
//   smooth_linear_interpolation( &eta,  M, M1, eta_table, lM1);
//   smooth_bilinear_interpolation(&T, M, h, M2, h1, T_table);


//   linear_interpolation( &CL_a, M, M1, CLa_table,  lM1);
//   linear_interpolation( &CD0,  M, M1, CD0_table, lM1);
//   linear_interpolation( &eta,  M, M1, eta_table, lM1);
//   bilinear_interpolation(&T, M, h, M2, h1, T_table);


   adouble CL = CL_a*alpha;
   adouble CD = CD0 + eta*CL_a*alpha*alpha;

   adouble D = 0.5*CD*S*rho*v*v;
   adouble L = 0.5*CL*S*rho*v*v;


   adouble hdot      = v*sin(gamma);
   adouble vdot      = 1.0/m*(T*cos(alpha)-D) - mu/pow(Re+h,2.0)*sin(gamma);
   adouble gammadot = (1.0/(m*v))*(T*sin(alpha)+L) + cos(gamma)*(v/(Re+h)-mu/(v*pow(Re+h,2.0)));
   adouble wdot      = -T/Isp;


   derivatives[ CINDEX(1) ] = hdot;
   derivatives[ CINDEX(2) ] = vdot;
   derivatives[ CINDEX(3) ] = gammadot;
   derivatives[ CINDEX(4) ] = wdot;


}
//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ///////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{


     adouble h0     = initial_states[CINDEX(1)];
     adouble v0     = initial_states[CINDEX(2)];
     adouble gamma0 = initial_states[CINDEX(3)];
     adouble w0     = initial_states[CINDEX(4)];

     adouble hf     = final_states[CINDEX(1)];
     adouble vf     = final_states[CINDEX(2)];
     adouble gammaf = final_states[CINDEX(3)];

     e[ CINDEX(1) ] = h0;
     e[ CINDEX(2) ] = v0;
     e[ CINDEX(3) ] = gamma0;
     e[ CINDEX(4) ] = w0;
     e[ CINDEX(5) ] = hf;
     e[ CINDEX(6) ] = vf;
     e[ CINDEX(7) ] = gammaf;

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{

  // Single phase problem

}



//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////
//////////////////////////////////////////////////////////////////////

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

```
//////////////////  Declare key structures ///////////////////////////////
//////////////////////////////////////////////////////////////////////////

   Alg   algorithm;
   Sol   solution;
   Prob  problem;

//////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  /////////////////////////////
//////////////////////////////////////////////////////////////////////////

   problem.name = "Minimum time to climb for a supersonic aircraft";
   problem.outfilename = "climb.txt";

//////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////////

   problem.nphases     = 1;
   problem.nlinkages                = 0;

   psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
//////////////////////////////////////////////////////////////////////////

   problem.phases(1).nstates   = 4;
   problem.phases(1).ncontrols = 1;
   problem.phases(1).nevents   = 7;
   problem.phases(1).npath     = 0;

   problem.phases(1).nodes     = "[30,60]";


   psopt_level2_setup(problem, algorithm);

//////////////////////////////////////////////////////////////////////////
//////////////////  Declare DMatrix objects to store results //////////////
//////////////////////////////////////////////////////////////////////////

   DMatrix x, u, t, H;



//////////////////////////////////////////////////////////////////////////
//////////////////  Initialize CONSTANTS and ///////////////////////////
//////////////////  declare local variables  ///////////////////////////
//////////////////////////////////////////////////////////////////////////

  CONSTANTS.g0  = 32.174; // ft/s^2
  CONSTANTS.S   = 530.0; // ft^2
  CONSTANTS.Re  = 20902900.0; // ft
  CONSTANTS.Isp = 1600.00; //s
  CONSTANTS.mu  = 0.14076539E17; // ft^3/s^2


  DMatrix M1(1,9,
  0.E0, .4E0, .8E0, .9E0, 1.E0, 1.2E0, 1.4E0, 1.6E0, 1.8E0);

  DMatrix M2(1,10,
  0.E0, .2E0, .4E0, .6E0, .8E0, 1.E0, 1.2E0, 1.4E0, 1.6E0, 1.8E0);

  DMatrix h1(1,10,
  0.E0, 5E3, 10.E3, 15.E3, 20.E3, 25.E3, 30.E3, 40.E3, 50.E3, 70.E3);

  DMatrix CLa_table(1,9,
  3.44E0, 3.44E0, 3.44E0, 3.58E0, 4.44E0, 3.44E0, 3.01E0, 2.86E0, 2.44E0);

  DMatrix CD0_table(1,9,
  .013E0, .013E0, .013E0, .014E0, .031E0, 0.041E0, .039E0, .036E0, .035E0);

  DMatrix eta_table(1,9,
  .54E0, .54E0, .54E0, .75E0, .79E0, .78E0, .89E0, .93E0, .93E0);

  DMatrix T_table(10,10,
  24200., 24000.,  20300.,  17300.,14500.,12200.,10200.,5700.,3400.,100.,
  28000., 24600.,  21100.,  18100.,15200.,12800.,10700.,6500.,3900.,200.,
  28300., 25200.,  21900.,  18700.,15900.,13400.,11200.,7300.,4400.,400.,
  30800., 27200.,  23800.,  20500.,17300.,14700.,12300.,8100.,4900.,800.,
  34500., 30300.,  26600.,  23200.,19800.,16800.,14100.,9400.,5600.,1100.,
  37900., 34300.,  30400.,  26800.,23300.,19800.,16800.,11200.,6800.,1400.,
  36100., 38000.,  34900.,  31300.,27300.,23600.,20100.,13400.,8300.,1700.,
```

278

```
      34300., 36600.,  38500.,  36100.,31600.,28100.,24200.,16200.,10000.,2200.,
      32500., 35200.,  42100.,  38700.,35700.,32000.,28100.,19300.,11900.,2900.,
      30700., 33800.,  45700.,  41300.,39800.,34600.,31100.,21700.,13300.,3100. );

      DMatrix htab(1,8,
                     0.0, 11.0, 20.0, 32.0, 47.0, 51.0, 71.0, 84.852);
      DMatrix ttab(1,8,
              288.15, 216.65, 216.65, 228.65, 270.65, 270.65, 214.65, 186.946);
      DMatrix ptab(1,8,
                   1.0, 2.233611E-1, 5.403295E-2, 8.5666784E-3, 1.0945601E-3,
                                       6.6063531E-4, 3.9046834E-5, 3.68501E-6);
      DMatrix gtab(1,8, -6.5, 0.0, 1.0, 2.8, 0.0, -2.8, -2.0, 0.0);

      M1.Print("M1");

      M2.Print("M2");

      h1.Print("h1");

      CLa_table.Print("CLa_table");

      CD0_table.Print("CD0_table");

      eta_table.Print("eta_table");

      T_table.Print("T_table");


      CONSTANTS.M1          = &M1;
      CONSTANTS.M2          = &M2;
      CONSTANTS.h1          = &h1;
      CONSTANTS.CLa_table   = &CLa_table;
      CONSTANTS.CD0_table   = &CD0_table;
      CONSTANTS.eta_table   = &eta_table;
      CONSTANTS.T_table     = &T_table;
      CONSTANTS.htab        = &htab;
      CONSTANTS.ttab        = &ttab;
      CONSTANTS.ptab        = &ptab;
      CONSTANTS.gtab        = &gtab;

      double h0    = 0.0;
      double hf    = 65600.0;
      double v0    = 424.26;
      double vf    = 968.148;
      double gamma0 = 0.0;
      double gammaf = 0.0;
      double w0    = 42000.0;

      double hmin =  0;
      double hmax =  69000.0;
      double vmin =  1.0;
      double vmax =  2000.0;
      double gammamin = -89.0*pi/180.0; // -89.0*pi/180.0;
      double gammamax =  89.0*pi/180.0; //  89.0*pi/180.0;
      double wmin     = 0.0;
      double wmax     = 45000.0;
      double alphamin = -20.0*pi/180.0;
      double alphamax =  20.0*pi/180.0;

      double t0min = 0.0;
      double t0max = 0.0;
      double tfmin = 200.0;
      double tfmax = 500.0;


//////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information /////////////////////
//////////////////////////////////////////////////////////////////////////

   int iphase =  0;

   problem.phase[iphase].bounds.lower.StartTime    = t0min;
   problem.phase[iphase].bounds.upper.StartTime    = t0max;

   problem.phase[iphase].bounds.lower.EndTime      = tfmin;
   problem.phase[iphase].bounds.upper.EndTime      = tfmax;


   problem.phase[iphase].bounds.lower.states(1) = hmin;
   problem.phase[iphase].bounds.upper.states(1) = hmax;
   problem.phase[iphase].bounds.lower.states(2) = vmin;
   problem.phase[iphase].bounds.upper.states(2) = vmax;
   problem.phase[iphase].bounds.lower.states(3) = gammamin;
```

279

```
    problem.phase[iphase].bounds.upper.states(3) = gammamax;
    problem.phase[iphase].bounds.lower.states(4) = wmin;
    problem.phase[iphase].bounds.upper.states(4) = wmax;


    problem.phase[iphase].bounds.lower.controls(1) = alphamin;
    problem.phase[iphase].bounds.upper.controls(1) = alphamax;

    // The following bounds fix the initial and final state conditions

    problem.phase[iphase].bounds.lower.events(1) = h0;
    problem.phase[iphase].bounds.upper.events(1) = h0;
    problem.phase[iphase].bounds.lower.events(2) = v0;
    problem.phase[iphase].bounds.upper.events(2) = v0;
    problem.phase[iphase].bounds.lower.events(3) = gamma0;
    problem.phase[iphase].bounds.upper.events(3) = gamma0;
    problem.phase[iphase].bounds.lower.events(4) = w0;
    problem.phase[iphase].bounds.upper.events(4) = w0;
    problem.phase[iphase].bounds.lower.events(5) = hf;
    problem.phase[iphase].bounds.upper.events(5) = hf;
    problem.phase[iphase].bounds.lower.events(6) = vf;
    problem.phase[iphase].bounds.upper.events(6) = vf;
    problem.phase[iphase].bounds.lower.events(7) = gammaf;
    problem.phase[iphase].bounds.upper.events(7) = gammaf;


////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess  ////////////////////
////////////////////////////////////////////////////////////////////////

    int nnodes = problem.phases(1).nodes(1);


    DMatrix stateGuess(4,nnodes);

    stateGuess(1, colon()) = linspace(h0,hf,nnodes);
    stateGuess(2, colon()) = linspace(v0,vf,nnodes);
    stateGuess(3, colon()) = linspace(gamma0,gammaf,nnodes);
    stateGuess(4, colon()) = linspace(w0,0.8*w0,nnodes);



    problem.phases(1).guess.controls      = zeros(1,nnodes);
    problem.phases(1).guess.states        = stateGuess;
    problem.phases(1).guess.time          = linspace(t0min, tfmax, nnodes);


////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////

    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ////////////////////////////
////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method                = "IPOPT";
    algorithm.scaling                   = "automatic";
    algorithm.derivatives               = "numerical";
    algorithm.collocation_method         = "Legendre";
    algorithm.nlp_iter_max              = 1000;
    algorithm.nlp_tolerance             = 1.e-6;
 //   algorithm.mesh_refinement             = "automatic";
    algorithm.mr_max_iterations         = 4;
    algorithm.defect_scaling             = "jacobian-based";



////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   ////////////////
////////////////////////////////////////////////////////////////////////


    psopt(solution, problem, algorithm);
```

```
///////////////////////////////////////////////////////////////////////
//////////  Extract relevant variables from solution structure   //////////
///////////////////////////////////////////////////////////////////////

    x  = solution.get_states_in_phase(1);
    u  = solution.get_controls_in_phase(1);
    t  = solution.get_time_in_phase(1);
    H           = solution.get_dual_hamiltonian_in_phase(1);

    DMatrix h     = x(1,colon());
    DMatrix v     = x(2,colon());
    DMatrix gamma = x(3,colon());
    DMatrix w     = x(4,colon());

///////////////////////////////////////////////////////////////////////
//////////  Save solution data to files if desired ///////////////////////
///////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");


///////////////////////////////////////////////////////////////////////
//////////  Plot some results if desired (requires gnuplot) ////////////////
///////////////////////////////////////////////////////////////////////

    plot(t,h/1000.0,problem.name + ": altitude", "time (s)", "altitude (x1,000 ft)", "h");
    plot(t,v/100.0,problem.name + ": velocity", "time (s)", "velocity (x100 ft/s)", "v");
    plot(t,gamma*180/pi,problem.name + ": flight path angle", "time (s)", "gamma (deg)", "gamma");
    plot(t,w/10000.0,problem.name + ": weight", "time (s)", "w (x10,000 lb)", "w");
    plot(t,u*180/pi,problem.name + ": angle of attack", "time (s)", "alpha (deg)", "alpha");

    plot(t,h/1000.0,problem.name + ": altitude", "time (s)", "altitude (x1,000 ft", "h",
                            "pdf","climb_altitude.pdf");
    plot(t,v/100.0,problem.name + ": velocity", "time (s)", "velocity (x100 ft/s)", "v",
                            "pdf","climb_velocity.pdf");
    plot(t,gamma*180/pi,problem.name + ": flight path angle", "time (s)", "gamma (deg)", "gamma",
                            "pdf","climb_fpa.pdf");
    plot(t,w/10000.0,problem.name + ": weight", "time (s)", "w (x10,000 lb)", "w", "pdf",
                            "weight.pdf");
    plot(t,u*180/pi,problem.name + ": angle of attack", "time (s)", "alpha (deg)", "alpha",
                            "pdf", "alpha.pdf");


}


void atmosphere(adouble* alt,adouble* sigma,adouble* delta,adouble* theta)
// US Standard Atmosphere Model 1976
// Adopted from original Fortran 90 code by Ralph Carmichael
// Fortran code located at: http://www.pdas.com/programs/atmos.f90
{
/*!  -----------------------------------------------------------------------
! PURPOSE - Compute the properties of the 1976 standard atmosphere to 86 km.
! AUTHOR - Ralph Carmichael, Public Domain Aeronautical Software
! NOTE - If alt > 86, the values returned will not be correct, but they will
!   not be too far removed from the correct values for density.
!   The reference document does not use the terms pressure and temperature
!   above 86 km.
  IMPLICIT NONE
!=======================================================================
!     A R G U M E N T S                                                |
!=======================================================================
  alt        ! geometric altitude, km.
  sigma      ! density/sea-level standard density
  delta      ! pressure/sea-level standard pressure
  theta      ! temperature/sea-level standard temperature
*/

/*!=======================================================================
!     L O C A L   C O N S T A N T S                                    |
!=======================================================================
*/
  double REARTH = 6369.0;                   // radius of the Earth (km)
  double GMR = 34.163195;                   // hydrostatic constant
  int NTAB=8;        // number of entries in the defining tables
/*!=======================================================================
!     L O C A L   V A R I A B L E S                                    |
!=======================================================================
*/
```

```
  int i,j,k;                                           // counters
  adouble h;                                          // geopotential altitude (km)
  adouble tgrad, tbase;      // temperature gradient and base temp of this layer
  adouble tlocal;                                        // local temperature
  adouble deltah;                            // height above base of this layer
/*!=========================================================================
!    L O C A L   A R R A Y S   ( 1 9 7 6   S T D.   A T M O S P H E R E )   |
!=========================================================================
*/

  DMatrix& htab = *CONSTANTS.htab;
  DMatrix& ttab = *CONSTANTS.ttab;
  DMatrix& ptab = *CONSTANTS.ptab;
  DMatrix& gtab = *CONSTANTS.gtab;


//!---------------------------------------------------------------------------
  h=(*alt)*REARTH/((*alt)+REARTH);      //convert geometric to geopotential altitude

  i=1;
  j=NTAB;                                      // setting up for binary search
  while (j<=i+1) {
    k=(i+j)/2;                                       // integer division
    if (h < htab(k)) {
      j=k;
    } else {
      i=k;
    }
  }

  tgrad=gtab(i);                                  // i will be in 1...NTAB-1
  tbase=ttab(i);
  deltah=h-htab(i);
  tlocal=tbase+tgrad*deltah;
  *theta=tlocal/ttab(1);                                // temperature ratio

  if (tgrad == 0.0) {                             //  pressure ratio
    *delta=ptab(i)*exp(-GMR*deltah/tbase);
  } else {
    *delta=ptab(i)*pow(tbase/tlocal, GMR/tgrad);
  }

  *sigma=(*delta)/(*theta);                                  // density ratio
  return;

}


void atmosphere_model(adouble* rho, adouble* M, adouble v, adouble h)
{
   double feet2meter = 0.3048;
   double kgperm3_to_slug_per_feet3 = 0.062427960841/32.174049;
   adouble alt, sigma, delta, theta;
   alt = h.value()*feet2meter/1000.0;

   // Call the standard atmosphere model 1976

   atmosphere(&alt, &sigma, &delta, &theta);

   adouble rho1 = 1.22521 * sigma; // Multiply by standard density at zero altitude and 15 deg C.

   rho1 = rho1*kgperm3_to_slug_per_feet3;

   *rho = rho1;

   adouble T;
   adouble mach;

   double TempStandardSeaLevel = 288.15; // in K, or 15 deg C.

   T = theta*TempStandardSeaLevel;

   adouble a = 20.0468 * sqrt(T); // Speed of sound in m/s.

   a = a/feet2meter;  // Speed of sound in ft/s

   mach = v/a;

   *M = mach;


   return;
}
```

Iter	Method	Nodes	NV	NC	OE	CE	JE	HE	RHS	$\epsilon_{\max}$	CPU(sec)
1	LGL-ST	80	322	247	747	746	89	0	59680	1.752e-03	8.020e+00
2	LGL-ST	90	362	277	70	71	55	0	6390	1.706e-03	6.890e+00
3	LGL-ST	100	402	307	28	29	28	0	2900	7.940e-04	4.810e+00
-	-	-	-	-	845	846	172	0	68970	-	1.972e+01

Key: Iter=iteration number, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, $\epsilon_{\max}$ = maximum relative ODE error, CPU(sec) = CPU time in seconds spent by nonlinear programming algorithm

Table 3.5: Mesh refinement statistics: Minimum time to climb for a supersonic aircraft

```
/////////////////////////////////////////////////////////////////////
///////////////////////    END OF FILE    /////////////////////////////
/////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and Figures 3.65, to 3.69. The results can be compared with those presented in [4]. Table 3.1 shows the mesh refinement history for this problem.

```
PSOPT results summary
=====================


Problem:  Minimum time to climb for a supersonic aircraft
CPU time (seconds): 3.210000e+02
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  3.188067e+02
Phase 1 endpoint cost function value: 3.188067e+02
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 3.188067e+02
Phase 1 maximum relative local error: 2.066822e-03
NLP solver reports:  The problem solved!
```

3.27   Missile terminal burn maneouvre

This example illustrates the design of a missile trajectory to strike a specified target from given initial conditions in minimum time [40]. Figure 3.27 shows the variables associated with the dynamic model of the missile employed in this example, where γ is the flight path angle, α is the angle of attack, V is the missile speed, x is the longitudinal position, h is the altitude, D is the
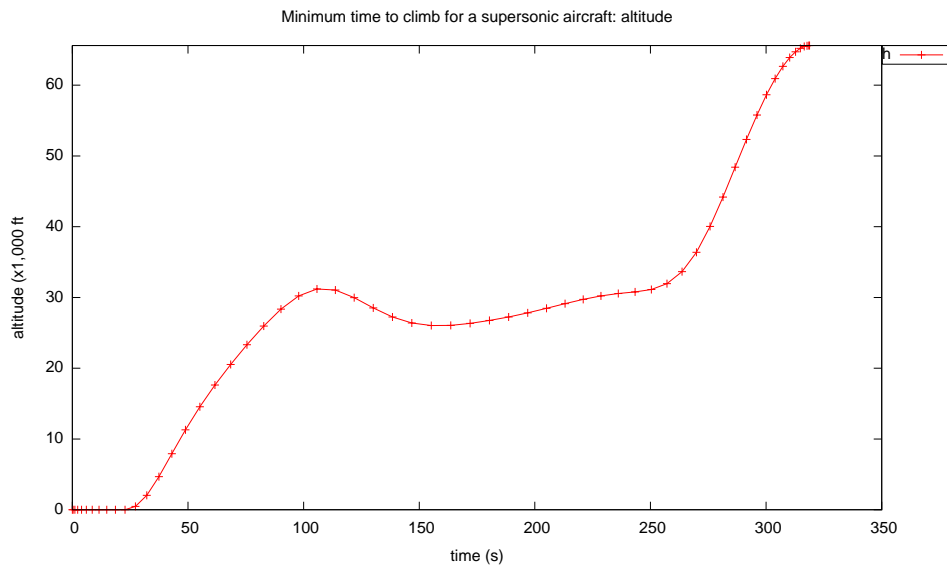
Figure 3.65: Altitude for minimum time to climb problem
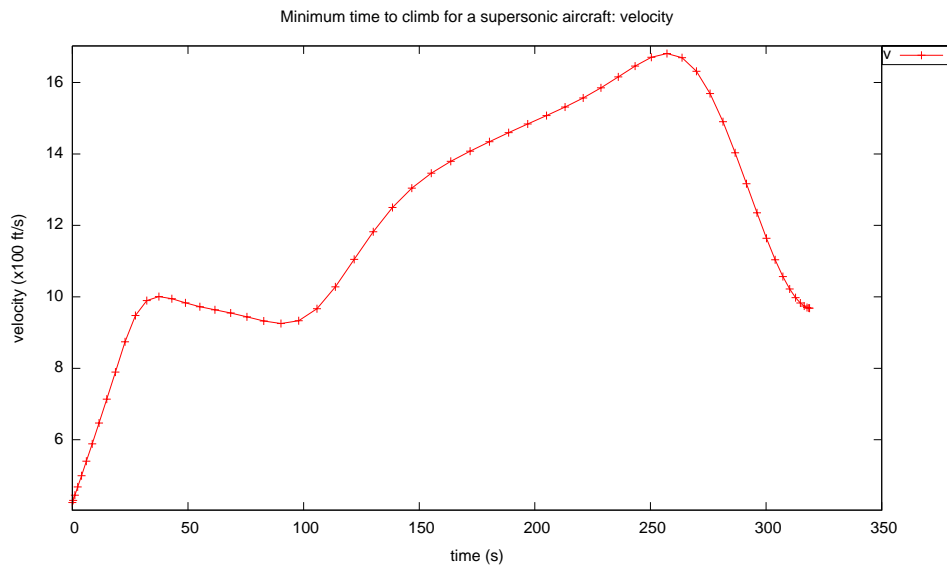
Figure 3.66: Velocity for minimum time to climb problem

284
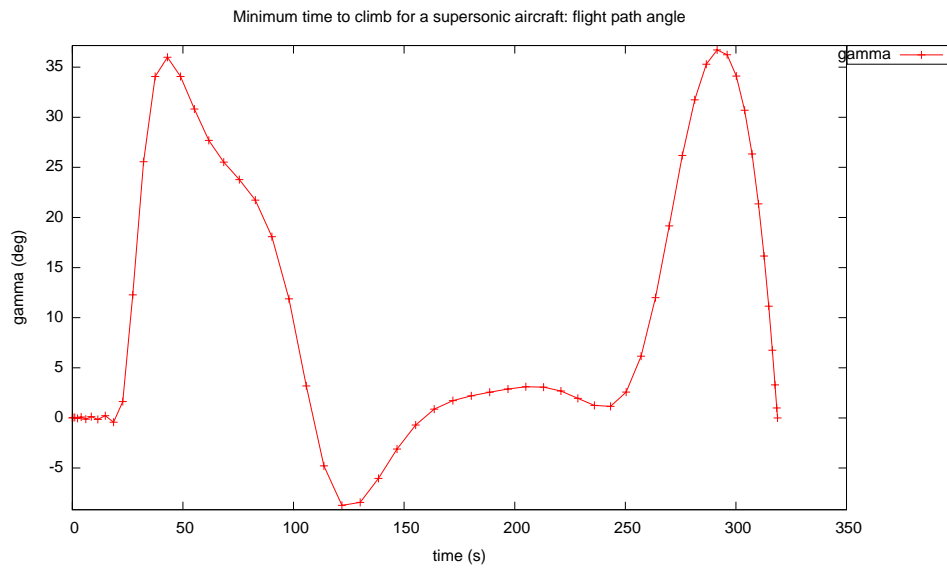
Figure 3.67: Flight path angle for minimum time to climb problem
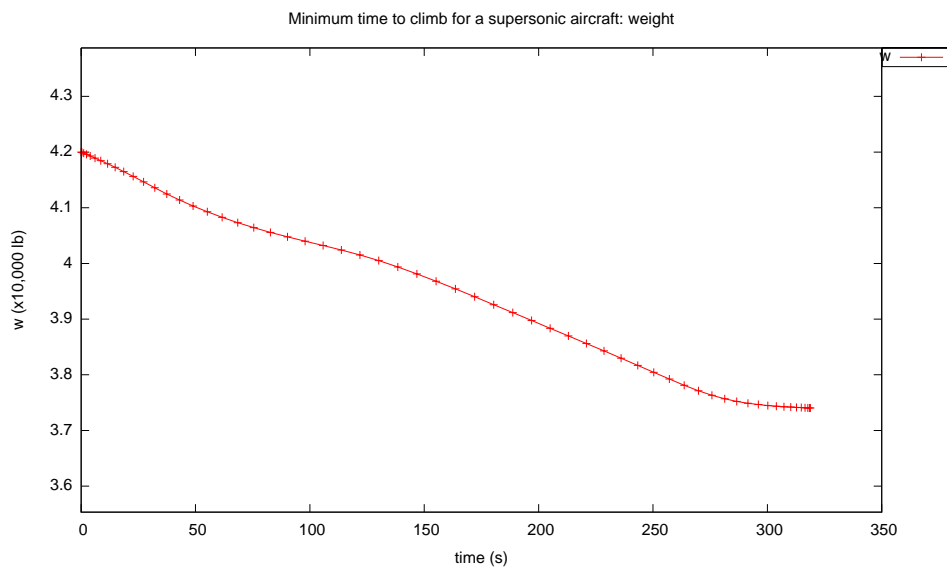
Figure 3.68: Weight for minimum time to climb problem
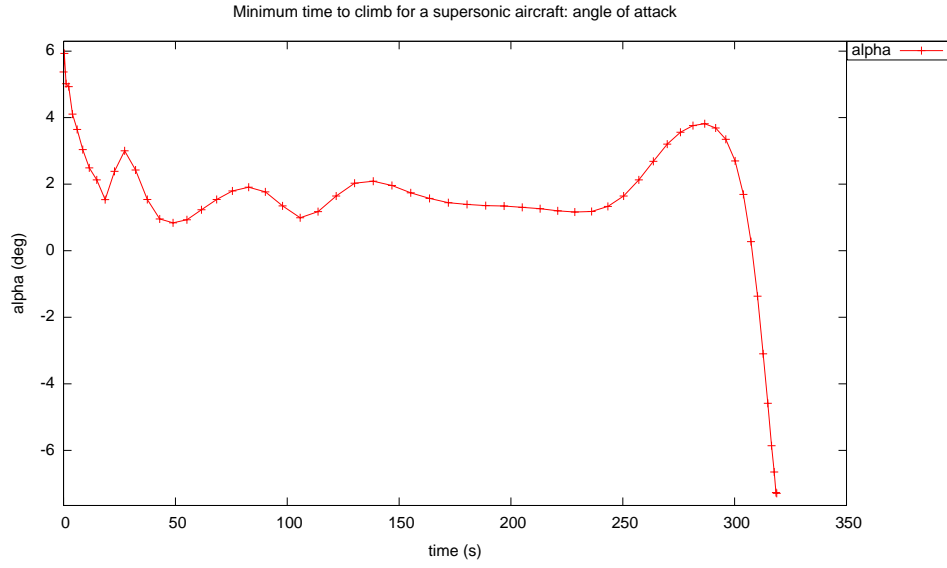
Figure 3.69: Angle of attack (α) for minimum time to climb problem

axial aerodynamic force, L is the normal aerodynamic force, and T is the thrust.

The equations of motion of the missile are given by:

$$\dot{\gamma} = \frac{T - D}{mg} \sin \alpha + \frac{L}{mV} \cos \alpha - \frac{g \cos \gamma}{V}$$
$$\dot{V} = \frac{T - D}{m} \cos \alpha - \frac{L}{m} \sin \alpha - g \cos \gamma$$
$$\dot{x} = V \cos \gamma$$
$$\dot{h} = V \sin \gamma$$

where

$$D = \frac{1}{2} C_d \rho V^2 Sref$$
$$C_d = A_1 \alpha^2 + A_2 \alpha + A_3$$
$$L = \frac{1}{2} C_l \rho V^2 Sref$$
$$C_l = B_1 \alpha + B_2$$
$$\rho = C_1 h^2 + C_2 h + C_3$$

where all the model parameters are given in Table 3.6. The initial conditions for the state variables are:
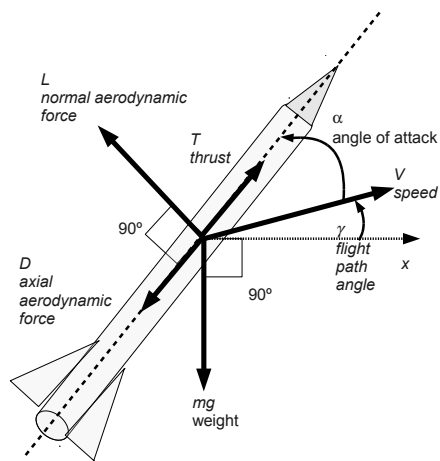
Figure 3.70: Ilustration of the variables associated with the missile model

Table 3.6: Parameters values of the missile model

Parameter	Value	Units
m	1005	kg
g	9.81	m/s^2
S_{ref}	0.3376	m^2
A_1	-1.9431	
A_2	-0.1499	
A_3	0.2359	
B_1	21.9	
B_2	0	
C_1	3.312×10^{-9}	kg/m^5
C_2	-1.142×10^{-4}	kg/m^4
C_3	1.224	kg/m^3

$$\gamma(0) = 0$$
$$V(0) = 272\text{m/s}$$
$$x(0) = 0\text{m}$$
$$h(0) = 30\text{m}$$

The terminal conditions on the states are:

$$\gamma(t_f) = -\pi/2$$
$$V(t_f) = 310\text{m/s}$$
$$x(t_f) = 10000\text{m}$$
$$h(t_f) = 0\text{m}$$

The problem constraints are given by:

$$200 \leq V \leq 310$$
$$1000 \leq T \leq 6000$$
$$-0.3 \leq \alpha \leq 0.3$$
$$-4 \leq \frac{L}{mg} \leq 4$$
$$h \geq 30 \text{ (for } x \leq 7500\text{m)}$$
$$h \geq 0 \text{ (for } x > 7500\text{m)}$$

Note that the path constraints on the altitude are non-smooth. Given that non-smoothness causes problems with nonlinear programming, the constraints on the altitude were approximated by a single smooth constraint:

$$\mathcal{H}_\epsilon(x - 7500))h(t) + [1 - \mathcal{H}_\epsilon(x - 7500)][h(t) - 30] \geq 0$$

where $\mathcal{H}_\epsilon(z)$ is a smooth version of the Heaviside function, which is computed as follows:

$$\mathcal{H}_\epsilon(z) = 0.5(1 + \tanh(z/\epsilon))$$

where $\epsilon > 0$ is a small number.

The problem is solved by using automatic mesh refinement starting with 50 nodes. The final solution, which is found after six mesh refinement iterations, has 85 nodes. Figure 3.71 shows the missile altitude as a function of the longitudinal position. Figures 3.72 and 3.73 show, respectively, the missile speed and angle of attack as functions of time. The output from $\mathcal{PSOPT}$ is summarised in the box below.

```
PSOPT results summary
=====================


Problem:  Missile problem
CPU time (seconds): 3.227818e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:   4.091754e+01
Phase 1 endpoint cost function value: 4.091754e+01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 4.091754e+01
Phase 1 maximum relative local error: 4.933590e-04
NLP solver reports:  The problem solved!
```

3.28 Moon lander problem

Consider the following optimal control problem, which is known in the literature as the moon lander problem [37]. Find t_f and $T(t) \in [0, t_f]$ to minimize the cost functional

$$J = \int_0^{t_f} T(t)dt \tag{3.114}$$

subject to the dynamic constraints

$$\begin{aligned}
\dot{h} &= v \\
\dot{v} &= -g + T/m \\
\dot{m} &= -T/E
\end{aligned} \tag{3.115}$$
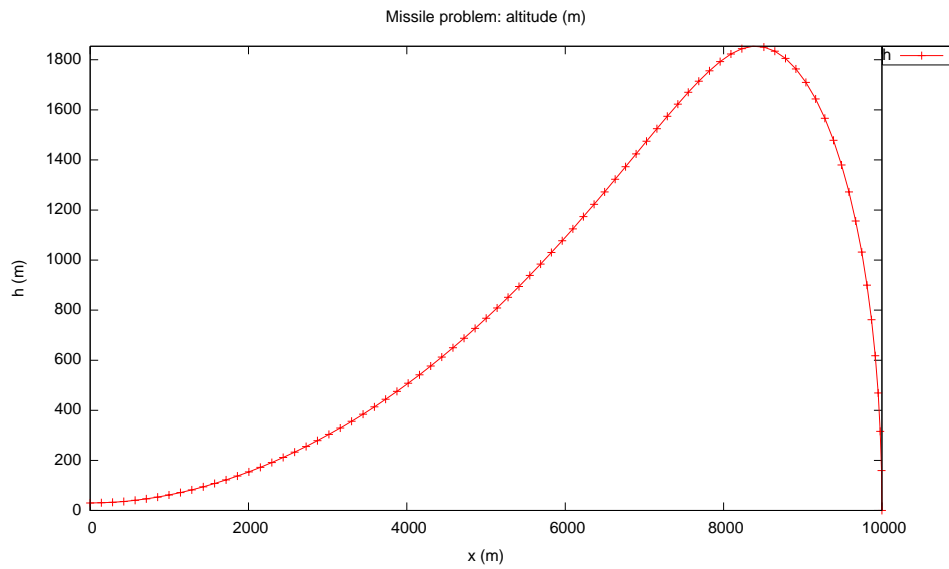
Figure 3.71: Missile altitude and a function of the longitudinal position
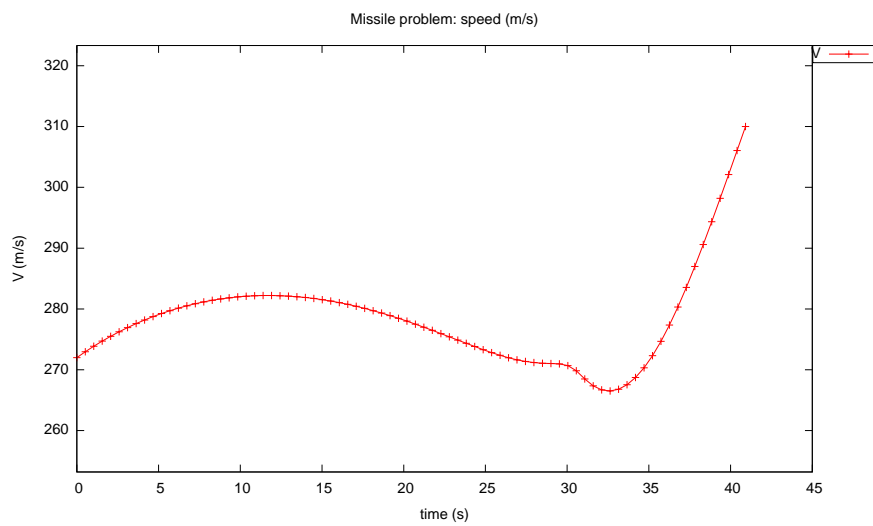
Figure 3.72: Missile speed as a function of time

290
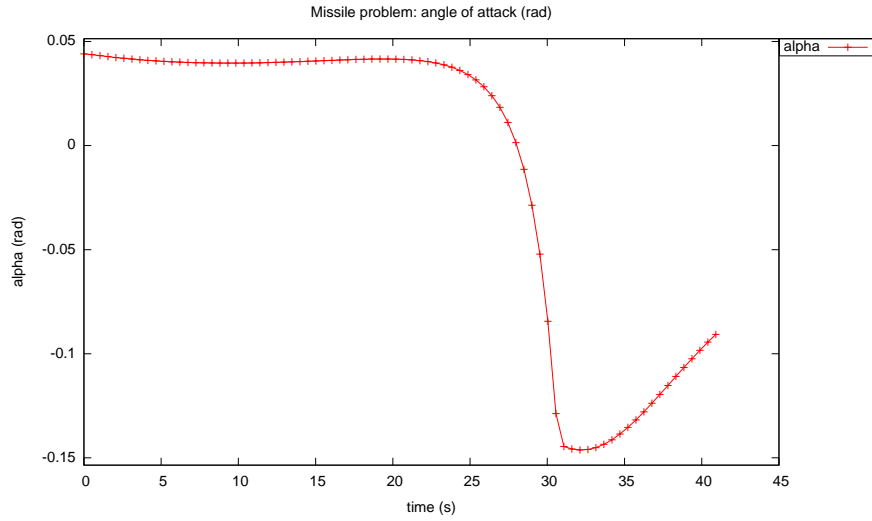
Figure 3.73: Missile angle of attack as a function of time

the boundary conditions:

$$
\begin{aligned}
h(0) &= 1 \\
v(0) &= -0.783 \\
m(0) &= 1 \\
h(t_f) &= 0.0 \\
v(t_f) &= 0.0
\end{aligned}
\tag{3.116}
$$

and the bounds

$$
\begin{aligned}
0 \le T(t) &\le 1.227 \\
-20 \le h(t) &\le 20 \\
-20 \le v(t) &\le 20 \\
0.01 \le m(t) &\le 1 \\
0 \le t_f &\le 1000
\end{aligned}
\tag{3.117}
$$

where $g = 1.0$, and $E = 2.349$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
/////////////////       moonlander.cxx      ////////////////////////
//////////////////////////////////////////////////////////////////////
////////////////         PSOPT  Example         ////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:      Moonlander problem          ////////////////
//////// Last modified: 05 January 2009          ////////////////
//////// Reference:    PROPT Handbook            ////////////////
//////// (See PSOPT handbook for full reference) ////////////////
//////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009    ////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////
```

```
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                    ///////////////
/////////////////////////////////////////////////////////////////////

#include "psopt.h"


/////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
/////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return 0.0;
}


/////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
/////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    adouble thrust = controls[0];
    return  thrust;
}


/////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's /////////////////////////////////
/////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble altitude_dot, speed_dot, mass_dot;

   adouble altitude  = states[0];
   adouble speed     = states[1];
   adouble mass      = states[2];

   adouble thrust = controls[0];

   double exhaust_velocity = 2.349;
   double gravity          = 1.0;

   altitude_dot =  speed;
   speed_dot    = -gravity + thrust/mass;
   mass_dot     = -thrust/exhaust_velocity;

   derivatives[0] = altitude_dot;
   derivatives[1] = speed_dot;
   derivatives[2] = mass_dot;
}


/////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ///////////////////////////
/////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble altitude_i = initial_states[0];
   adouble speed_i    = initial_states[1];
   adouble mass_i     = initial_states[2];
   adouble altitude_f = final_states[0];
   adouble speed_f    = final_states[1];

   e[0] = altitude_i;
   e[1] = speed_i;
   e[2] = mass_i;
   e[3] = altitude_f;
   e[4] = speed_f;

}


/////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////////
/////////////////////////////////////////////////////////////////////
```

```
void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


////////////////////////////////////////////////////////////////////////
//////////////////////  Define the main routine /////////////////////////
////////////////////////////////////////////////////////////////////////

int main(void)
{
////////////////////////////////////////////////////////////////////////
//////////////////////  Declare key structures //////////////////////////
////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////
//////////////////////  Register problem name  //////////////////////////
////////////////////////////////////////////////////////////////////////

    problem.name          = "Moon Lander Problem";
    problem.outfilename            = "moon.txt";

////////////////////////////////////////////////////////////////////////
/////////////  Define problem level constants & do level 1 setup /////////////
////////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages            = 0;

    psopt_level1_setup(problem);

////////////////////////////////////////////////////////////////////////
/////////    Define phase related information & do level 2 setup  ////////////
////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 3;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 5;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes                = 70;

    psopt_level2_setup(problem, algorithm);

    int nodes = 70;

////////////////////////////////////////////////////////////////////////
//////////////////////  Declare DMatrix objects to store results //////////////
////////////////////////////////////////////////////////////////////////

    DMatrix x, u, t;
    DMatrix lambda, H;

////////////////////////////////////////////////////////////////////////
//////////////////////  Enter problem bounds information /////////////////////
////////////////////////////////////////////////////////////////////////


    double altitudeL   = -20.0;
    double speedL  = -20.0;
    double massL  = 0.01;
    double altitudeU  = 20.0;
    double speedU  = 20.0;
    double massU  = 1.0;

    double thrustL       = 0.0;
    double thrustU       = 1.227;

    double altitude_i  = 1.0;
    double speed_i     = -0.783;
    double mass_i      = 1.0;

    double altitude_f  = 0.0;
    double speed_f     = 0.0;


    problem.phases(1).bounds.lower.states(1) = altitudeL;
    problem.phases(1).bounds.lower.states(2) = speedL;
    problem.phases(1).bounds.lower.states(3) = massL;
```

```
    problem.phases(1).bounds.upper.states(1) = altitudeU;
    problem.phases(1).bounds.upper.states(2) = speedU;
    problem.phases(1).bounds.upper.states(3) = massU;

    problem.phases(1).bounds.lower.controls(1) = thrustL;
    problem.phases(1).bounds.upper.controls(1) = thrustU;

    problem.phases(1).bounds.lower.events(1) = altitude_i;
    problem.phases(1).bounds.lower.events(2) = speed_i;
    problem.phases(1).bounds.lower.events(3) = mass_i;
    problem.phases(1).bounds.lower.events(4) = altitude_f;
    problem.phases(1).bounds.lower.events(5) = speed_f;

    problem.phases(1).bounds.upper.events =  problem.phases(1).bounds.lower.events;

    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 0.0;
    problem.phases(1).bounds.upper.EndTime      = 1000.0;


////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  //////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////

    DMatrix states_guess(3,nodes+1);

    states_guess(1,colon()) = linspace(altitude_i, altitude_f, nodes+1);
    states_guess(2,colon()) = linspace(speed_i, speed_f, nodes+1);
    states_guess(3,colon()) = linspace(mass_i, massL, nodes+1);


    problem.phases(1).guess.controls = 0.5*(thrustL+thrustU)*ones(1,nodes+1);
    problem.phases(1).guess.states   = states_guess;
    problem.phases(1).guess.time     = linspace(0.0, 1.5, nodes+1);

////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  /////////////////////////////
////////////////////////////////////////////////////////////////////////

    algorithm.nlp_iter_max   = 1000;
    algorithm.nlp_tolerance     = 1.e-6;
    algorithm.nlp_method    = "IPOPT";
    algorithm.scaling       = "automatic";
    algorithm.derivatives   = "automatic";

////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////


    x  = solution.get_states_in_phase(1);
    u  = solution.get_controls_in_phase(1);
    t  = solution.get_time_in_phase(1);
    lambda  = solution.get_dual_costates_in_phase(1);
    H  = solution.get_dual_hamiltonian_in_phase(1);

////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ///////////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
```

```
    t.Save("t.dat");
    lambda.Save("lambda.dat");
    H.Save("H.dat");

//////////////////////////////////////////////////////////////////////////
//////////// Plot some results if desired (requires gnuplot) //////////////
//////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name, "time (s)", "states", "altitude speed mass");

    plot(t,u,problem.name, "time (s)", "control", "thrust");

    plot(t,x,problem.name, "time (s)", "states", "altitude speed mass",
                           "pdf", "moon_states.pdf");

    plot(t,u,problem.name, "time (s)", "control", "thrust",
                           "pdf", "moon_control.pdf");


}

//////////////////////////////////////////////////////////////////////////
/////////////////////////   END OF FILE   /////////////////////////////////
//////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.74 and 3.75, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================


Problem:  Moon Lander Problem
CPU time (seconds): 5.160000e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  1.420377e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  1.420377e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.396970e+00
Phase 1 maximum relative local error: 5.392272e-05
NLP solver reports:  The problem solved!
```

3.29   Multi-segment problem

Consider the following optimal control problem, where the optimal control has a characteristic stepped shape [23]. Find $u(t) \in [0, 3]$ to minimize the cost functional

$$J = \int_0^3 x(t)dt \qquad (3.118)$$
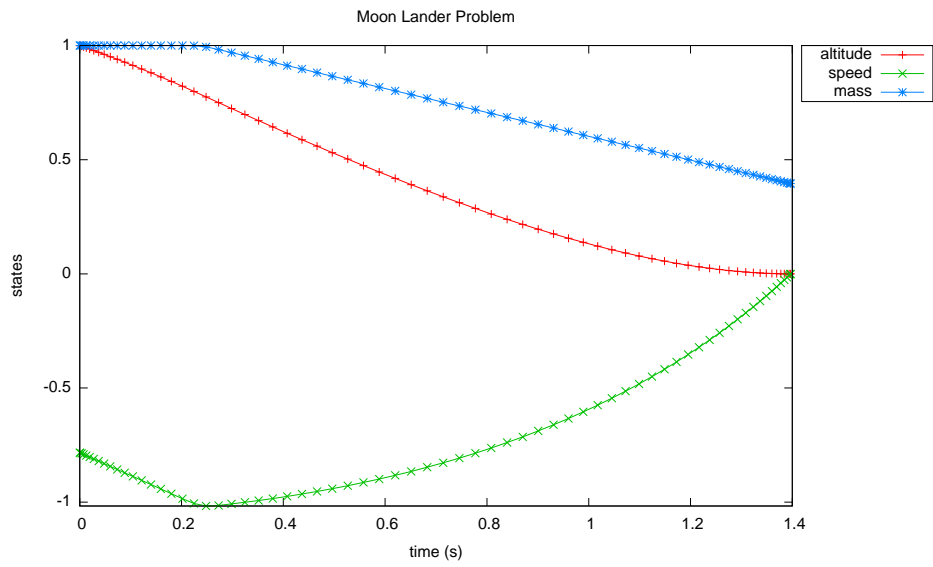
Figure 3.74: States for moon lander problem
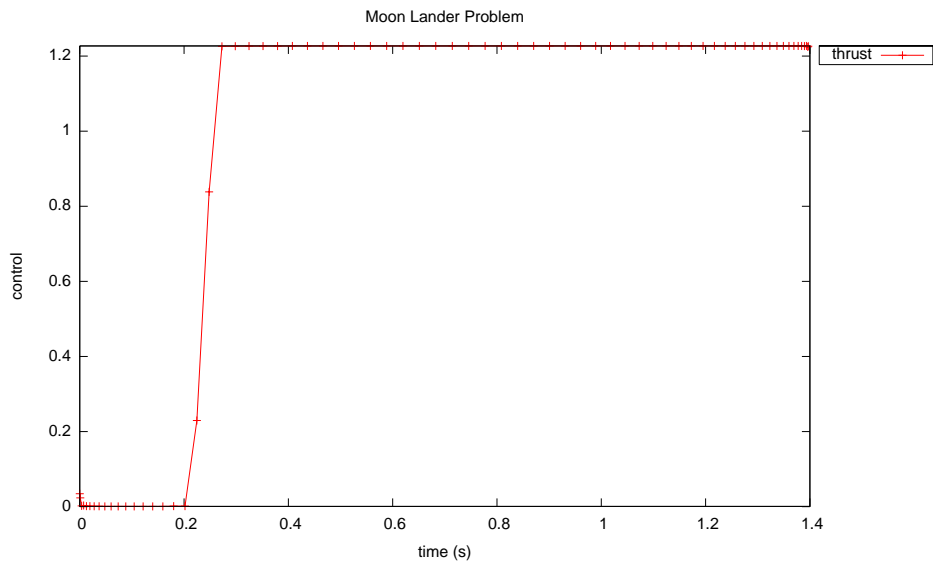
Figure 3.75: Control for moon lander problem

subject to the dynamic constraints

$$\dot{x} = u \qquad\qquad (3.119)$$

the boundary conditions:

$$
\begin{aligned}
x(0) &= 1 \\
x(3) &= 1
\end{aligned}
\qquad\qquad (3.120)
$$

and the bounds

$$
\begin{aligned}
-1 \le u(t) &\le 1 \\
x(t) &\ge 0
\end{aligned}
\qquad\qquad (3.121)
$$

The analytical optimal control is given by:

$$
u(t) = \left\{
\begin{array}{ll}
-1, & t \in [0,1) \\
0, & t \in [1,2] \\
1, & t \in (2,3]
\end{array}
\right.
\qquad\qquad (3.122)
$$

The problem has been solved using the multi-segment paradigm. Three segments are defined in the code, such that the initial time is fixed at $t_0^{(1)} = 0$, the final time is fixed at $t_f^{(3)} = 3$, and the intermediate junction times are $t_f^{(1)} = 1$, and $t_f^{(2)} = 2$.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
////////////////          steps.cxx               ////////////////////
//////////////////////////////////////////////////////////////////////
////////////////          PSOPT example           ////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:  Steps problem                    ////////////////
//////// Last modified: 12 July 2009              ////////////////
//////// Reference:     Gong, Farhoo, and Ross (2008)   ////////////////
////////                                          ////////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009     ////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)            ////////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"


//////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////


adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                    adouble* parameters,adouble& t0, adouble& tf,
                    adouble* xad, int iphase, Workspace* workspace)
{
    return 0.0;
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                    adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
```

```
   adouble x = states[ CINDEX(1) ];

   return (x);
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ///////////////////////////////////
//////////////////////////////////////////////////////////////////////


void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

    adouble u = controls[CINDEX(1)];

    derivatives[ CINDEX(1) ]  = u;

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function /////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{

    adouble x1_i  = initial_states[ CINDEX(1) ];

    adouble x1_f  = final_states[ CINDEX(1) ];


   if ( iphase==1 ) {
    e[ CINDEX(1) ] =  x1_i;
   }
   else if ( iphase==3 ) {
    e[ CINDEX(1) ] = x1_f;
    }

}



//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function /////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{

}



//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ////////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures /////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;
    MSdata msdata;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  /////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name          = "Steps problem";
    problem.outfilename                 = "steps.txt";

//////////////////////////////////////////////////////////////////////
```

```
//////////// Define problem level constants & do level 1 setup ////////////
/////////////////////////////////////////////////////////////////////////

    msdata.nsegments       = 3;
    msdata.nstates         = 1;
    msdata.ncontrols       = 1;
    msdata.nparameters     = 0;
    msdata.npath           = 0;
    msdata.ninitial_events = 1;
    msdata.nfinal_events   = 1;
    msdata.nodes           = 20; // nodes per segment

    multi_segment_setup(problem, algorithm, msdata );

/////////////////////////////////////////////////////////////////////////
//////////////////// Enter problem bounds information /////////////////////
/////////////////////////////////////////////////////////////////////////

    problem.phases(1).bounds.lower.controls(1) = -1.0;
    problem.phases(1).bounds.upper.controls(1) =  1.0;
    problem.phases(1).bounds.lower.states(1) = 0.0;
    problem.phases(1).bounds.upper.states(1) = 5.0;
    problem.phases(1).bounds.lower.events(1) = 1.0;
    problem.phases(3).bounds.lower.events(1) = 1.0;


    problem.phases(1).bounds.upper.events=problem.phases(1).bounds.lower.events;
    problem.phases(3).bounds.upper.events=problem.phases(3).bounds.lower.events;



    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(3).bounds.lower.EndTime      = 3.0;
    problem.phases(3).bounds.upper.EndTime      = 3.0;
//    problem.bounds.lower.times = "[0.0, 1.0, 2.0, 3.0]";
//    problem.bounds.upper.times = "[0.0, 1.0, 2.0, 3.0]";

    problem.bounds.lower.times = "[0.0, 1.0, 2.0, 3.0]";
    problem.bounds.upper.times = "[0.0, 1.0, 2.0, 3.0]";

    auto_phase_bounds(problem);



/////////////////////////////////////////////////////////////////////////
////////////////// Register problem functions  ///////////////////////////
/////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae            = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

/////////////////////////////////////////////////////////////////////////
////////////////// Define & register initial guess ///////////////////////
/////////////////////////////////////////////////////////////////////////

    int nnodes      = problem.phases(1).nodes(1);
    int ncontrols                      = problem.phases(1).ncontrols;
    int nstates                        = problem.phases(1).nstates;

    DMatrix state_guess    = zeros(nstates,nnodes);
    DMatrix control_guess  = zeros(ncontrols,nnodes);
    DMatrix time_guess     = linspace(0.0,3.0,nnodes);
    DMatrix param_guess;


    state_guess(1,colon()) = linspace(1.0, 1.0, nnodes);

    control_guess(1,colon()) = zeros(1,nnodes);


    auto_phase_guess(problem, control_guess, state_guess, param_guess, time_guess);

/////////////////////////////////////////////////////////////////////////
////////////////// Enter algorithm options  //////////////////////////////
```

```
////////////////////////////////////////////////////////////////////

   algorithm.nlp_iter_max              = 1000;
   algorithm.nlp_tolerance             = 1.e-6;
   algorithm.nlp_method                = "IPOPT";
   algorithm.scaling                   = "automatic";
   algorithm.derivatives               = "automatic";
   algorithm.hessian                   = "exact";
   algorithm.mesh_refinement           = "automatic";
   algorithm.ode_tolerance             = 1.e-5;


////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////

   psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////
//////////// Extract relevant variables from solution structure  //////////
////////////////////////////////////////////////////////////////////


   DMatrix x, u, t, xi, ui, ti;

   x       = solution.get_states_in_phase(1);
   u       = solution.get_controls_in_phase(1);
   t       = solution.get_time_in_phase(1);

   for(int i=2;i<=problem.nphases;i++) {
         xi      = solution.get_states_in_phase(i);
       ui      = solution.get_controls_in_phase(i);
       ti      = solution.get_time_in_phase(i);

x = x || xi;

       u = u || ui;

       t = t || ti;

   }


////////////////////////////////////////////////////////////////////
//////////// Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////

   x.Save("x.dat");
   u.Save("u.dat");
   t.Save("t.dat");

////////////////////////////////////////////////////////////////////
//////////// Plot some results if desired (requires gnuplot) ////////////////
////////////////////////////////////////////////////////////////////

   plot(t,x(1,colon()),problem.name+": state","time (s)", "x", "x");

   plot(t,u(1,colon()),problem.name+": control","time (s)", "u", "u");

   plot(t,x(1,colon()),problem.name+": state","time (s)", "x", "x",
                         "pdf", "steps_state.pdf");


   plot(t,u(1,colon()),problem.name+": control","time (s)", "u", "u",
                         "pdf", "steps_control.pdf");


}

////////////////////////////////////////////////////////////////////
/////////////////////////     END OF FILE    ////////////////////////////
////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.76 and 3.77, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Steps problem
CPU time (seconds): 2.100000e-01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  1.000010e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.000034e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 6.960805e-07
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  3.401583e-06
Phase 2 initial time: 1.000000e+00
Phase 2 final time: 2.000000e+00
Phase 2 maximum relative local error: 4.856293e-06
Phase 3 endpoint cost function value: 0.000000e+00
Phase 3 integrated part of the cost:  5.000034e-01
Phase 3 initial time: 2.000000e+00
Phase 3 final time: 3.000000e+00
Phase 3 maximum relative local error: 6.961472e-07
NLP solver reports:  The problem solved!
```

3.30   Notorious parameter estimation problem

Consider the following parameter estimation problem, which is known to be challenging to single-shooting methods because of internal instability of the differential equations [38]. Find $p \in \Re$ to minimize

$$J = \sum_{i=1}^{200} (y_1(t_i) - \tilde{y}_1(i))^2 + (y_2(t_i) - \tilde{y}_2(i))^2 \qquad (3.123)$$

subject to the dynamic constraints

$$\begin{array}{rcl} \dot{y}_1 & = & y_2 \\ \dot{y}_2 & = & \mu^2 y_1 - (\mu^2 + p^2)\sin(pt) \end{array} \qquad (3.124)$$

where $\mu = 60.0$, $y_1(0) = 0$, $y_2(0) = \pi$. The parameter estimation facilities of $\mathcal{PSOPT}$ are used in this example. In this case, the observations function is:

$$g(x(\theta_k), u(\theta_k), p, \theta_k) = [y_1(\theta_k) \ \ y_2(\theta_k)]^T$$
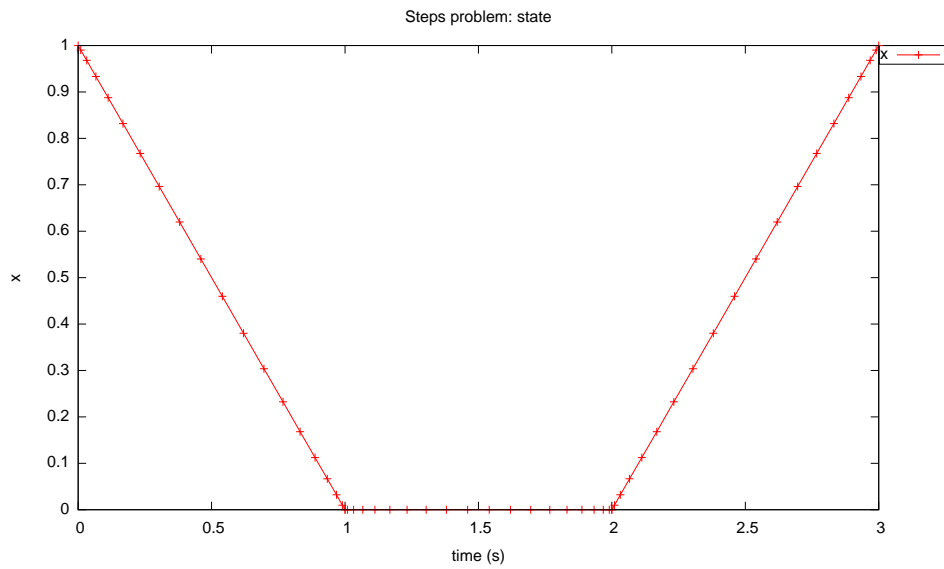
Figure 3.76: State trajectory for the multi-segment problem
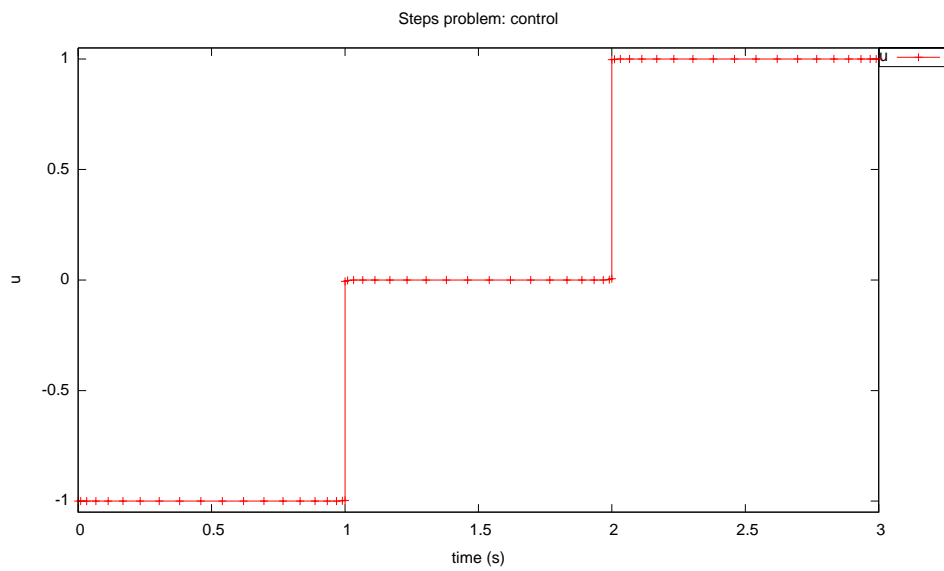
Figure 3.77: Control trajectory for the multi-segment problem

The $\mathcal{PSOPT}$ code that solves this problem is shown below. The code includes the generation of the measurement vectors $\tilde{y}_1$, and $\tilde{y}_2$ by adding Gaussian noise with standard deviation 0.05 to the exact solution of the problem with $p = \pi$, which is given by:

$$y_1(t) = \sin(\pi t)$$
$$y_2(t) = \pi \cos(\pi t)$$

The code also defines the vector of sampling instants $\theta_i, i = 1, \ldots, 200$ as a uniform random samples in the interval $[0, 1]$.

```
//////////////////////////////////////////////////////////////////////
//////////////////          notorious.cxx        //////////////////////
//////////////////////////////////////////////////////////////////////
//////////////           PSOPT  Example           /////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:  Bock's notorious parameter estimation problem //////////
//////// Last modified: 08 April 2011               ////////////////
//////// Reference:    Schittkowskit (2002)         ////////////////
//////// (See PSOPT handbook for full reference)        ////////////////
//////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009      ////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)              ////////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
//////////////////  Define the observation function //////////
//////////////////////////////////////////////////////////////////////


DMatrix tobs_global;
DMatrix u_global;


void  observation_function( adouble* observations,
                            adouble* states, adouble* controls,
                            adouble* parameters, adouble& time, int k,
                            adouble* xad, int iphase, Workspace* workspace)
{

     observations[ CINDEX(1) ] = states[ CINDEX(1) ];
     observations[ CINDEX(2) ] = states[ CINDEX(2) ];
}




//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ///////////////////////////////////
//////////////////////////////////////////////////////////////////////


void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

   adouble x1 = states[ CINDEX(1) ];
   adouble x2 = states[ CINDEX(2) ];


   adouble p  = parameters[ CINDEX(1) ];
   adouble t  = time;

   double mu = 60.0;

   derivatives[CINDEX(1)] =  x2;
```

```
   derivatives[CINDEX(2)] =  mu*mu*x1 - (mu*mu + p*p)*sin(p*t);


}
//////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////////
//////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{

   e[ 0 ] =  initial_states[0];
   e[ 1 ] =  initial_states[1];


}


//////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////////
//////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////////
//////////////////////////////////////////////////////////////////////////

int main(void)
{
   int nobs =200;

   // Generate true solution at sampling points and add noise

   double sigma = 0.05;

   DMatrix theta, y1m, y2m, ym;

   theta = randu(1,nobs);

   sort(theta);

   y1m = sin( pi* theta ) + sigma*randn(1,nobs);

   y2m = pi*cos( pi*theta ) + sigma*randn(1,nobs);


//////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////////
//////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  /////////////////////////////////
//////////////////////////////////////////////////////////////////////////

    problem.name          = "Bocks notorious parameter estimation problem";
    problem.outfilename                = "notorious.txt";

//////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                  = 0;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup ////////////
//////////////////////////////////////////////////////////////////////////
```

```
    problem.phases(1).nstates    = 2;
    problem.phases(1).ncontrols  = 0;
    problem.phases(1).nevents    = 2;
    problem.phases(1).npath      = 0;
    problem.phases(1).nparameters        = 1;
    problem.phases(1).nodes         = "[80]";
    problem.phases(1).nobserved              = 2;
    problem.phases(1).nsamples               = nobs;


    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////
///////////// Enter estimation information               ///////////
////////////////////////////////////////////////////////////////////


    problem.phases(1).observation_nodes    = (theta);
    problem.phases(1).observations         = (y1m && y2m);



////////////////////////////////////////////////////////////////////
/////////////////// Enter problem bounds information ////////////////////
////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states(1) = -10.0;
    problem.phases(1).bounds.lower.states(2) = -100.0;


    problem.phases(1).bounds.upper.states(1) =  10.0;
    problem.phases(1).bounds.upper.states(2) =  100.0;


    problem.phases(1).bounds.lower.parameters(1) =  -10.0;
    problem.phases(1).bounds.upper.parameters(1) =   10.0;

    problem.phases(1).bounds.lower.events(1) =  0.0;
    problem.phases(1).bounds.upper.events(1) =  0.0;

    problem.phases(1).bounds.lower.events(2) =  pi;
    problem.phases(1).bounds.upper.events(2) =  pi;



    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.0;
    problem.phases(1).bounds.upper.EndTime      = 1.0;

////////////////////////////////////////////////////////////////////
/////////////////// Register problem functions /////////////////////
////////////////////////////////////////////////////////////////////

    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;
    problem.observation_function = & observation_function;

////////////////////////////////////////////////////////////////////
/////////////////// Define & register initial guess ////////////////////
////////////////////////////////////////////////////////////////////

    int nnodes = problem.phases(1).nodes(1);

    DMatrix state_guess;
    state_guess = zeros(2,nnodes);

    problem.phases(1).guess.states          = state_guess;
    problem.phases(1).guess.time            = linspace(0.0, 1.0, nnodes);
    problem.phases(1).guess.parameters      = 0.0;


////////////////////////////////////////////////////////////////////
/////////////////// Enter algorithm options /////////////////////////
////////////////////////////////////////////////////////////////////

    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
```

```
    algorithm.collocation_method        = "trapezoidal";
    algorithm.nlp_iter_max              =  200;
    algorithm.nlp_tolerance             = 1.e-4;
//    algorithm.mesh_refinement            = "automatic";
//    algorithm.ode_tolerance              = 1.e-6;




////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   ///////////////////
////////////////////////////////////////////////////////////////////////

    int ntrials = 1;

    Sol best_solution;
    double best_cost = inf;

    for(int i=1; i<=ntrials;i++) {
        int j;
        double dpj, r;

        for(j=1;j<=problem.phases(1).nparameters;j++) {
           r = (randu(1,1))(1);
           dpj = problem.phases(1).bounds.upper.parameters(j) - problem.phases(1).bounds.lower.parameters(j);
           problem.phases(1).guess.parameters(j) = problem.phases(1).bounds.lower.parameters(j) + r*dpj;
        }

        psopt(solution, problem, algorithm);

        if (solution.get_cost() < best_cost ) {
              best_solution = solution;
              best_cost = solution.get_cost();
        }

    }

    solution= best_solution;

////////////////////////////////////////////////////////////////////////
///////////////////  Declare DMatrix objects to store results //////////////
////////////////////////////////////////////////////////////////////////

    DMatrix states, x1, x2, p, t;


////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure    //////////
////////////////////////////////////////////////////////////////////////

    states = solution.get_states_in_phase(1);
    t      = solution.get_time_in_phase(1);
    p      = solution.get_parameters_in_phase(1);
    x1     = states(1,colon());
    x2     = states(2,colon());


////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////

    states.Save("states.dat");
    t.Save("t.dat");
    p.Print("Estimated parameter");
    Abs(p-pi).Print("Parameter error");


////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) //////////////
////////////////////////////////////////////////////////////////////////

    plot(theta,ym(1,colon()),t,x1,problem.name, "time (s)", "observed x1", "x1m x1");
    plot(theta,ym(2,colon()),t,x2,problem.name, "time (s)", "observed x2", "x2m x2");



}

////////////////////////////////////////////////////////////////////////
/////////////////////       END OF FILE      /////////////////////////////
////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below. The optimal parameter found was $p = 3.141180$, which is an approximation of π with an error of the order of 10^{-4}. The 95% confidence interval of the estimated parameter is $[3.132363, 3.149998]$.

```
PSOPT results summary
=====================


Problem:  Bocks notorious parameter estimation problem
CPU time (seconds): 1.502405e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  8.588468e-01
Phase 1 endpoint cost function value: 8.588468e-01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 1.363735e-04
NLP solver reports:  The problem solved!
```

3.31   Predator-prey parameter estimation problem

This is a well knowon model that describes the behaviour of predator and prey species of an ecological system. The Letka-Volterra model system consist of two differential equations [38].

The dynamic equations are given by:

$$\begin{aligned} \dot{x}_1 &= -p_1 x_1 + p_2 x_1 x_2 \\ \dot{x}_2 &= p_3 x_2 - p_4 x_1 x_2 \end{aligned} \tag{3.125}$$

with boundary condition:

$$\begin{aligned} x_1(0) &= 0.4 \\ x_2(0) &= 1 \end{aligned}$$

The observation functions are:

$$\begin{aligned} g_1 &= x_1 \\ g_2 &= x_2 \end{aligned} \tag{3.126}$$

The measured data, with consists of $n_s = 10$ samples over the interval $t \in [0, 10]$, was constructed from simulations with parameter values $[p_1, p_2, p_3, p_4] = [1, 1, 1, 1]$ with added noise. The weights of both observations are the same and equal to one.

The solution is found using local discretisation (trapezoidal, Hermite-Simpson) and automatic mesh refinement, starting with 20 grid points with ODE tolerance 10^{-4}. The code that solves the problem is shown below. The estimated parameter values and their 95% confidence limits are shown in Table 3.31. Figure 3.78 shows the observations as well as the estimated values of variables x_1 and x_2. The mesh statistics can be seen in Table 3.8

```
//////////////////////////////////////////////////////////////////////
//////////////////         predator.cxx       //////////////////////////
//////////////////////////////////////////////////////////////////////
///////////////          PSOPT  Example           /////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:  Lotka-Volterra model parameter estimation ////  //////////
//////// Last modified:  03 Jan 2014                     //////////////
//////// Reference:    Schittkowski (2002)            ///////////////
//////// (See PSOPT handbook for full reference)         //////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2014      ///////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                ////////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
//////////////////  Define the observation function //////////
//////////////////////////////////////////////////////////////////////


void  observation_function( adouble* observations,
                            adouble* states, adouble* controls,
                            adouble* parameters, adouble& time, int k,
                            adouble* xad, int iphase, Workspace* workspace)
{
int i;
for (i=0; i<2; i++) {
            observations[ i ] = states[ i ];
}
}



//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's //////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
        adouble* controls, adouble* parameters, adouble& time,
        adouble* xad, int iphase, Workspace* workspace)
{

   // Variables
      adouble x1, x2, p1, p2, p3, p4;

   // Differential states
      x1 = states[CINDEX(1)];
      x2 = states[CINDEX(2)];


   // Parameters
      p1 = parameters[CINDEX(1)];
      p2 = parameters[CINDEX(2)];
   p3 = parameters[CINDEX(3)];
      p4 = parameters[CINDEX(4)];

      derivatives[CINDEX(1)] = -p1*x1 + p2*x1*x2;

      derivatives[CINDEX(2)] = p3*x2 - p4*x1*x2;

}

//////////////////////////////////////////////////////////////////////
```

```
/////////////////  Define the events function ////////////////////////////
///////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   int i;

   for (i=0; i<2; i++) {

      e[i]  = initial_states[i];

   }

}


///////////////////////////////////////////////////////////////////////////
/////////////////  Define the phase linkages function //////////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


///////////////////////////////////////////////////////////////////////////
/////////////////////  Define the main routine ////////////////////////////
///////////////////////////////////////////////////////////////////////////

int main(void)
{

///////////////////////////////////////////////////////////////////////////
/////////////////////  Declare key structures //////////////////////////////
///////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

///////////////////////////////////////////////////////////////////////////
/////////////////////  Register problem name  //////////////////////////////
///////////////////////////////////////////////////////////////////////////

    problem.name           =        "Predator-prey example";
    problem.outfilename         =      "predator.txt";

///////////////////////////////////////////////////////////////////////////
/////////////  Define problem level constants & do level 1 setup ////////////
///////////////////////////////////////////////////////////////////////////

    problem.nphases            = 1;
    problem.nlinkages                = 0;

    psopt_level1_setup(problem);


///////////////////////////////////////////////////////////////////////////
/////////////   Define phase related information & do level 2 setup //////////////
///////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 2;
    problem.phases(1).ncontrols  = 0;
    problem.phases(1).nevents    = 2;
    problem.phases(1).npath      = 0;
    problem.phases(1).nparameters        = 4;
    problem.phases(1).nodes          = 20;
    problem.phases(1).nobserved        = 2;
    problem.phases(1).nsamples         = 10;

    psopt_level2_setup(problem, algorithm);

///////////////////////////////////////////////////////////////////////////
/////////////  Load data for parameter estimation              ////////////
///////////////////////////////////////////////////////////////////////////

    int iphase = 1;
    load_parameter_estimation_data(problem, iphase, "predator.dat");

    problem.phases(1).observation_nodes.Print("observation nodes");
```

309

```
    problem.phases(1).observations.Print("observations");
    problem.phases(1).residual_weights.Print("weights");


////////////////////////////////////////////////////////////////////////
//////////////////  Declare DMatrix objects to store results //////////////
////////////////////////////////////////////////////////////////////////

    DMatrix x, p, t;

////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ///////////////////////
////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states(1) = 0.0;
    problem.phases(1).bounds.lower.states(2) = 0.0;


    problem.phases(1).bounds.upper.states(1) = 3.0;
    problem.phases(1).bounds.upper.states(2) = 3.0;

    problem.phases(1).bounds.lower.events(1) = 0.4;
    problem.phases(1).bounds.lower.events(2) = 1.0;


    problem.phases(1).bounds.upper.events(1) = 0.4;
    problem.phases(1).bounds.upper.events(2) = 1.0;


    problem.phases(1).bounds.lower.parameters(1)  = -1.0;
    problem.phases(1).bounds.lower.parameters(2)  = -1.0;
problem.phases(1).bounds.lower.parameters(3)  = -1.0;
    problem.phases(1).bounds.lower.parameters(4)  = -1.0;

    problem.phases(1).bounds.upper.parameters(1)  = 5.0;
    problem.phases(1).bounds.upper.parameters(2)  = 5.0;
problem.phases(1).bounds.upper.parameters(3)  = 5.0;
    problem.phases(1).bounds.upper.parameters(4)  = 5.0;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 10.0;
    problem.phases(1).bounds.upper.EndTime      = 10.0;

////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////

    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;
    problem.observation_function = & observation_function;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////

    int nnodes =     (int) problem.phases(1).nsamples;

    DMatrix state_guess(3, nnodes);
    DMatrix param_guess(2,1);


    state_guess(1,colon()) = linspace(0.4, 0.4, nnodes );
    state_guess(2,colon()) = linspace(1.0, 1.0, nnodes );

    param_guess(1) = 0.8;
    param_guess(2) = 0.8;
param_guess(3) = 1.5;
param_guess(4) = 1.5;

    problem.phases(1).guess.states        = state_guess;
    problem.phases(1).guess.time          = linspace(0.0, 10.0, nnodes);
    problem.phases(1).guess.parameters    = param_guess;



////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ///////////////////////////////
```

Table 3.7: Estimated parameter values and 95 percent statistical confidence limits on estimated parameters

Parameter	Low Confidence Limit	Value	High Confidence Limit
p_1	7.166429e-01	9.837490e-01	1.250855e+00
p_2	7.573469e-01	9.803930e-01	1.203439e+00
p_3	7.287846e-01	1.016900e+00	1.305015e+00
p_4	6.914964e-01	1.022702e+00	1.353909e+00

```
//////////////////////////////////////////////////////////////////////////

    algorithm.nlp_method             = "IPOPT";
    algorithm.scaling                = "automatic";
    algorithm.derivatives            = "automatic";
    algorithm.collocation_method     = "trapezoidal";
    algorithm.mesh_refinement        = "automatic";
     algorithm.ode_tolerance          = 1.e-4;

//////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   //////////////////
//////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

//////////////////////////////////////////////////////////////////////////
//////////// Extract relevant variables from solution structure   //////////
//////////////////////////////////////////////////////////////////////////

    x = solution.get_states_in_phase(1);
    t = solution.get_time_in_phase(1);
    p = solution.get_parameters_in_phase(1);


//////////////////////////////////////////////////////////////////////////
//////////// Save solution data to files if desired /////////////////////////
//////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    t.Save("t.dat");
    p.Print("Estimated parameters");


//////////////////////////////////////////////////////////////////////////
//////////// Plot some results if desired (requires gnuplot) ////////////////
//////////////////////////////////////////////////////////////////////////

    DMatrix tm;
    DMatrix ym;

    tm = problem.phases(1).observation_nodes;
    ym = problem.phases(1).observations;


    spplot(t,x,tm,ym,problem.name, "time (s)", "state x1", "x1 x2 y1 y2");

 spplot(t,x,tm,ym,problem.name, "time (s)", "state x1", "x1 x2 y1 y2", "pdf", "x1x2.pdf");



}




//////////////////////////////////////////////////////////////////////////
//////////////////////////     END OF FILE     /////////////////////////////
//////////////////////////////////////////////////////////////////////////
```

Figure 3.78: Observations y_1, y_2 and estimated states $x_1(t)$ and $x_2(t)$

Table 3.8: Mesh refinement statistics: Predator-prey example

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	ϵ_{max}	CPU_a
1	TRP	20	46	43	20	20	20	0	780	1.615e-02	4.000e-02
2	TRP	28	62	59	14	14	14	0	770	8.919e-03	4.000e-02
3	H-S	39	84	81	9	9	9	0	1035	1.670e-03	4.000e-02
4	H-S	54	114	111	11	11	11	0	1760	1.114e-04	5.000e-02
5	H-S	62	130	127	10	10	10	0	1840	3.985e-05	5.000e-02
CPU_b	-	-	-	-	-	-	-	-	-	-	4.500e-01
-	-	-	-	-	64	64	64	0	6185	-	6.700e-01

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, ϵ_{max} = maximum relative ODE error, CPU_a = CPU time in seconds spent by NLP algorithm, CPU_b = additional CPU time in seconds spent by PSOPT

3.32 Rayleigh problem with mixed state-control path constraints

Consider the following optimal control problem, which involves a path constraint in which the control and the state appear explicitly [4]. Find $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = \int_0^{t_f} \left[x_1(t)^2 + u(t)^2 \right] \mathrm{d}t \qquad (3.127)$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -x_1 + x_2(1.4 - px_2^2) + 4u\sin(\theta) \end{aligned} \qquad (3.128)$$

The path constraint:

$$u + \frac{x_1}{6} \leq 0 \qquad (3.129)$$

and the boundary conditions:

$$\begin{aligned} x_1(0) &= -5 \\ x_2(0) &= -5 \end{aligned} \qquad (3.130)$$

where $t_f = 4.5$, and $p = 0.14$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
/////////////////          PSOPT  Example          ///////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:                Rayleigh problem     ///////////////////
//////// Last modified:         11 July 2011         //////////////////
//////// Reference:             Betts (2010)          /////////////////
//////// (See PSOPT handbook for full reference)      /////////////////
//////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2011     ////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////
//////// is distributed under the terms of the GNU Lesser /////////////
//////// General Public License (LGPL)                     ////////////
//////////////////////////////////////////////////////////////////////


#include "psopt.h"

//////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   return 0.0;
}

//////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
```

```
                          int iphase, Workspace* workspace)
{
  adouble x1 = states[CINDEX(1)];
  adouble u = controls[CINDEX(1)];

  return  (u*u + x1*x1);
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ///////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

   adouble x1 = states[CINDEX(1)];
   adouble x2 = states[CINDEX(2)];

   adouble u = controls[CINDEX(1)];

   double p = 0.14;

   derivatives[ CINDEX(1) ] = x2;
   derivatives[ CINDEX(2) ] = -x1 + x2*(1.4-p*x2*x2) + 4.0*u;

   path[ CINDEX(1) ] = u + x1/6.0;
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x10 = initial_states[ CINDEX(1) ];
   adouble x20 = initial_states[ CINDEX(2) ];

   e[ CINDEX(1) ] = x10;
   e[ CINDEX(2) ] = x20;

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////
//////////////////////////////////////////////////////////////////////


int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////
//////////////////////////////////////////////////////////////////////

   Alg  algorithm;
   Sol  solution;
   Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////
//////////////////////////////////////////////////////////////////////

   problem.name        = "Rayleigh problem";
   problem.outfilename                = "rayleigh.txt";

//////////////////////////////////////////////////////////////////////
///////////  Declare problem level constants & do level 1 setup ///////////
```

```
///////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                = 0;

    psopt_level1_setup(problem);


///////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup ////////////
///////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates    = 2;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 2;
    problem.phases(1).npath      = 1;
    problem.phases(1).nodes                = "[80]";

    psopt_level2_setup(problem, algorithm);

///////////////////////////////////////////////////////////////////////
////////////////// Declare DMatrix objects to store results //////////////
///////////////////////////////////////////////////////////////////////

    DMatrix x, u, t;
    DMatrix lambda, H, mu;

///////////////////////////////////////////////////////////////////////
////////////////// Enter problem bounds information //////////////////////
///////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states(1)  = -10.0;
    problem.phases(1).bounds.lower.states(2)  = -10.0;

    problem.phases(1).bounds.upper.states(1)   = 10.0;
    problem.phases(1).bounds.upper.states(2)  = 10.0;

    problem.phases(1).bounds.lower.controls(1) = -10.0;
    problem.phases(1).bounds.upper.controls(1)   = 10.0;

    problem.phases(1).bounds.lower.events(1)  = -5.0;
    problem.phases(1).bounds.lower.events(2)  = -5.0;

    problem.phases(1).bounds.upper.events(1)  = -5.0;
    problem.phases(1).bounds.upper.events(2)  = -5.0;

    problem.phases(1).bounds.lower.path(1) = -100.0;
    problem.phases(1).bounds.upper.path(1) = 0.0;

    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;
    problem.phases(1).bounds.lower.EndTime      = 4.5;
    problem.phases(1).bounds.upper.EndTime      = 4.5;


///////////////////////////////////////////////////////////////////////
////////////////// Register problem functions  //////////////////////////
///////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

///////////////////////////////////////////////////////////////////////
////////////////// Define & register initial guess /////////////////////
///////////////////////////////////////////////////////////////////////

    DMatrix x0(2,30);

    x0(1,colon()) = -5.0*ones(1,30);
    x0(2,colon()) = -5.0*ones(1, 30);

    problem.phases(1).guess.controls      = zeros(1,30);
    problem.phases(1).guess.states        = x0;
    problem.phases(1).guess.time          = linspace(0.0, 4.5, 30);
```

```
///////////////////////////////////////////////////////////////////////
/////////////////// Enter algorithm options ////////////////////////////
///////////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////////
/////////////////// Enter algorithm options ////////////////////////////
///////////////////////////////////////////////////////////////////////

    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
    algorithm.collocation_method      = "Hermite-Simpson";
    algorithm.nlp_iter_max            = 1000;
    algorithm.nlp_tolerance           = 1.e-10;

///////////////////////////////////////////////////////////////////////
/////////////////// Now call PSOPT to solve the problem   ///////////////
///////////////////////////////////////////////////////////////////////


    psopt(solution, problem, algorithm);


///////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   ///////
///////////////////////////////////////////////////////////////////////

    x      = solution.get_states_in_phase(1);
    u      = solution.get_controls_in_phase(1);
    t      = solution.get_time_in_phase(1);
    lambda = solution.get_dual_costates_in_phase(1);
    mu     = solution.get_dual_path_in_phase(1);
    H      = solution.get_dual_hamiltonian_in_phase(1);

///////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired /////////////////////
///////////////////////////////////////////////////////////////////////


///////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ////////////
///////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name, "time (s)", "states", "x1 x2" );
    plot(t,u,problem.name ,"time (s)", "control", "u" );
    plot(t,lambda,problem.name ,"time (s)", "costates", "p1 p2");
    plot(t,mu, problem.name,"time (s)", "mu", "mu");

    plot(t,x,problem.name, "time (s)", "states", "x1 x2", "pdf", "rayleigh_states.pdf" );
    plot(t,u,problem.name ,"time (s)", "control", "u", "pdf", "rayleigh_control.pdf" );
    plot(t,lambda,problem.name ,"time (s)", "costates", "l1 l2", "pdf", "rayleigh_costates.pdf");
    plot(t,mu, problem.name,"time (s)", "mu", "mu", "pdf", "rayleigh_mu.pdf");

}

///////////////////////////////////////////////////////////////////////
/////////////////////////    END OF FILE    /////////////////////////////
///////////////////////////////////////////////////////////////////////
```
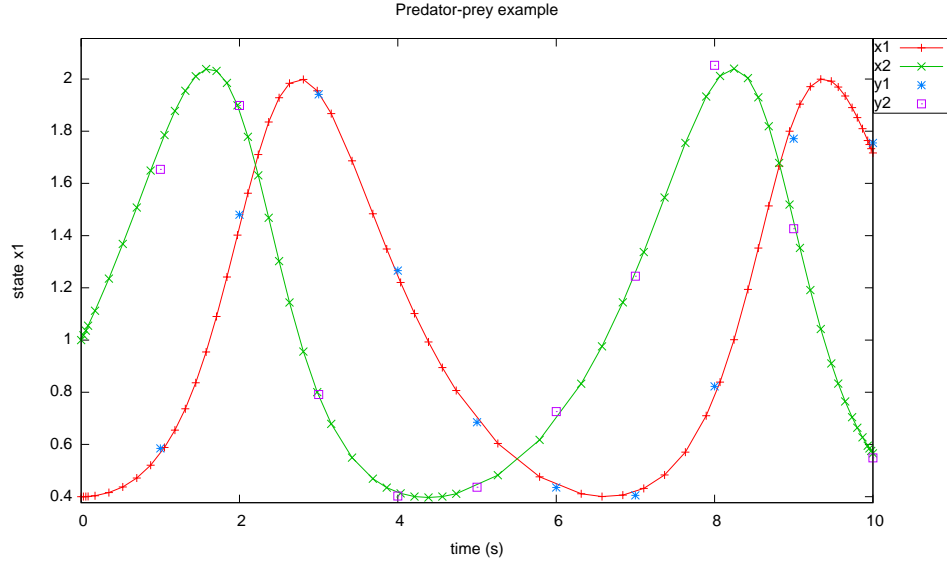
The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.79, 3.80, 3.82, 3.82, 3.82, which show, respectively, the trajectories of the states, control, costates and path constraint multiplier. The results are comparable to those presented by [4].

```
PSOPT results summary
=====================


Problem:  Rayleigh problem
```
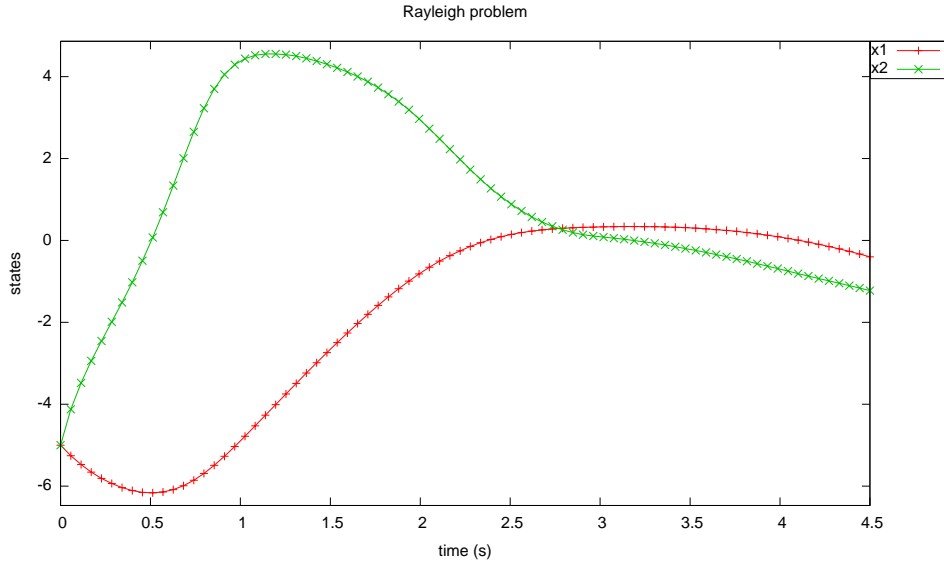
Figure 3.79: States for Rayleigh problem

```
CPU time (seconds): 6.600000e-01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  4.477625e+01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  4.477625e+01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 4.500000e+00
Phase 1 maximum relative local error: 2.329140e-03
NLP solver reports:  The problem solved!
```

3.33 Obstacle avoidance problem

Consider the following optimal control problem, which involves finding an optimal trajectory for a particle to travel from A to B while avoiding two forbidden regions [37]. Find $\theta(t) \in [0, t_f]$ to minimize the cost functional

$$J = \int_0^{t_f} \left[\dot{x}(t)^2 + \dot{y}(t)^2 \right] dt \tag{3.131}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x} &= V \cos(\theta) \\ \dot{y} &= V \sin(\theta) \end{aligned} \tag{3.132}$$

317
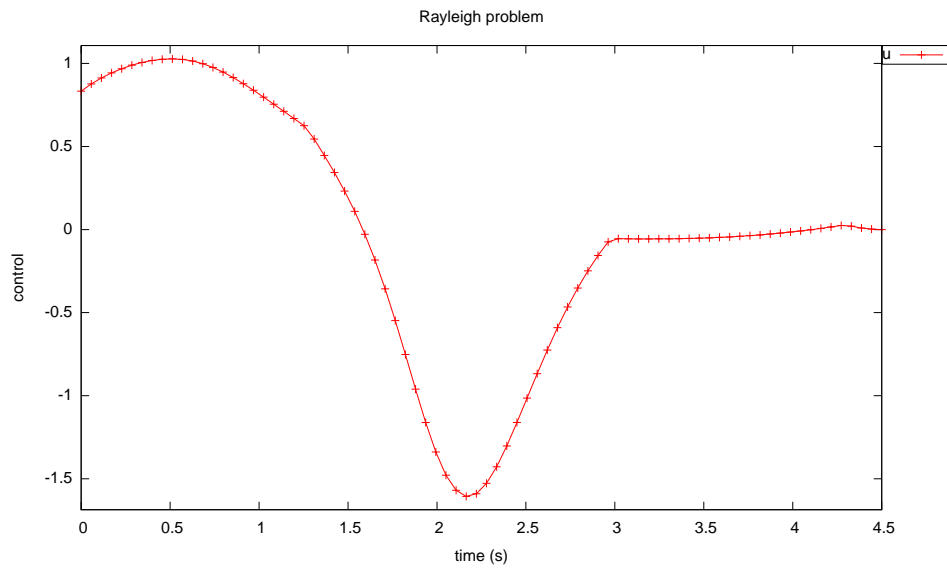
Figure 3.80: Optimal control for Rayleigh problem
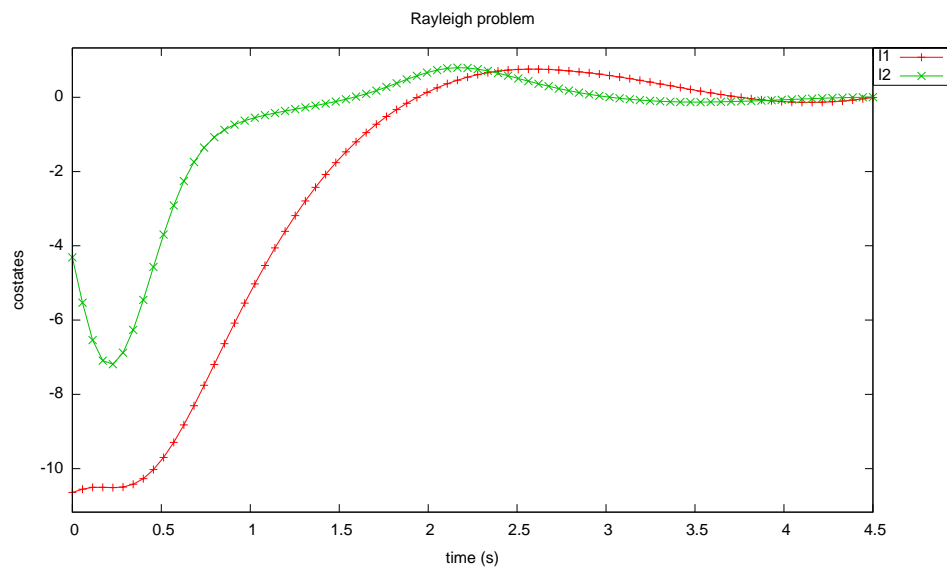
Figure 3.81: Costates for Rayleigh problem
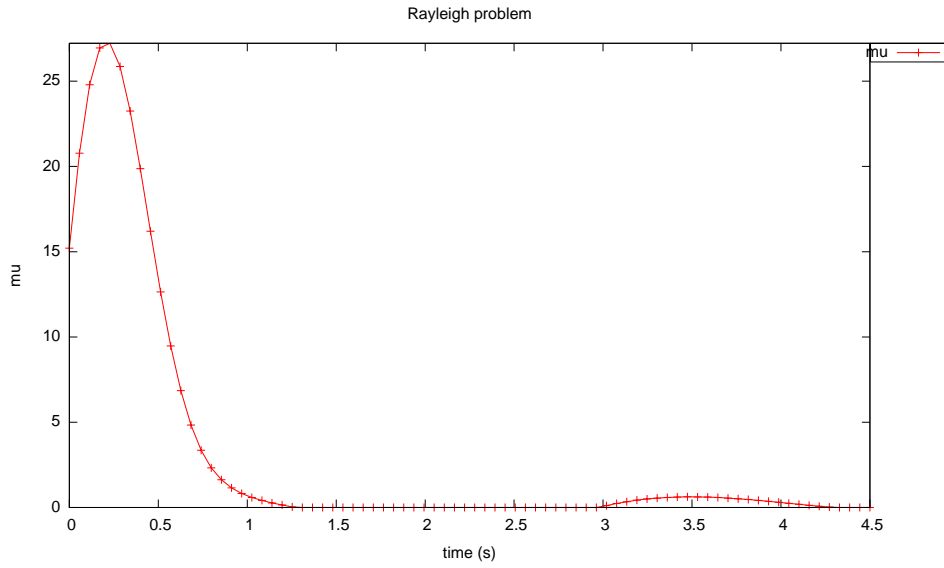
Figure 3.82: Path constraint multiplier for Rayleigh problem

The path constraints:

$$(x(t) - 0.4)^2 + (y(t) - 0.5)^2 \geq 0.1$$
$$(x(t) - 0.8)^2 + (y(t) - 1.5)^2 \geq 0.1, \qquad (3.133)$$

and the boundary conditions:

$$
\begin{aligned}
x(0) &= 0 \\
y(0) &= 0 \\
x(t_f) &= 1.2 \\
y(t_f) &= 1.6
\end{aligned}
\qquad (3.134)
$$

where $t_f = 1.0$, and $V = 2.138$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
///////////////        obstacle.cxx            ///////////////////////
//////////////////////////////////////////////////////////////////////
//////////////         PSOPT  Example          ///////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:         Obstacle avoidance problem     ////////////////
//////// Last modified: 05 January 2009                ////////////////
//////// Reference:  PROPT User's Guide                ////////////////
//////// (See PSOPT handbook for full reference)       ////////////////
//////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009     ////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                 ////////////////
//////////////////////////////////////////////////////////////////////
```

```
#include "psopt.h"

//////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase,Workspace* workspace)
{
   return 0;
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    double V = 2.138;
    adouble theta = controls[ CINDEX(1) ];

    adouble dxdt = V*cos(theta);
    adouble dydt = V*sin(theta);

//      adouble dxdt, dydt;

//      get_state_derivative( &dxdt, 1, iphase, time, xad);

//      get_state_derivative( &dydt, 2, iphase, time, xad);

    adouble L =  pow(dxdt,2.0) + pow(dydt,2.0);

    return  L;
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's //////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

   adouble x    = states[ CINDEX(1) ];
   adouble y    = states[ CINDEX(2) ];


   adouble theta = controls[ CINDEX(1) ];

   double V = 2.138;

   adouble dxdt = V*cos(theta);
   adouble dydt = V*sin(theta);


   derivatives[ CINDEX(1) ] = dxdt;
   derivatives[ CINDEX(2) ] = dydt;


   path[ CINDEX(1) ] = pow(x-0.4,2.0) + pow(y-0.5,2.0);
   path[ CINDEX(2) ] = pow(x-0.8,2.0) + pow(y-1.5,2.0);

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{
   adouble x0 = initial_states[ CINDEX(1) ];
   adouble y0 = initial_states[ CINDEX(2) ];
   adouble xf = final_states[   CINDEX(1) ];
   adouble yf = final_states[   CINDEX(2) ];
```

```
   e[ CINDEX(1) ] = x0;
   e[ CINDEX(2) ] = y0;
   e[ CINDEX(3) ] = xf;
   e[ CINDEX(4) ] = yf;

}



//////////////////////////////////////////////////////////////////////
/////////////////////  Define the phase linkages function ////////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}



//////////////////////////////////////////////////////////////////////
/////////////////////  Define the main routine ///////////////////////////////
//////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////
/////////////////////  Declare key structures ///////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
/////////////////////  Register problem name  ///////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name        = "Obstacle avoidance problem";
    problem.outfilename              = "obstacle.txt";

//////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup /////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                = 0;

    psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  /////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates   = 2;
    problem.phases(1).ncontrols = 1;
    problem.phases(1).nevents   = 4;
    problem.phases(1).npath                = 2;
    problem.phases(1).nodes                = "[20]";

    psopt_level2_setup(problem, algorithm);


//////////////////////////////////////////////////////////////////////
/////////////////////  Enter problem bounds information /////////////////////
//////////////////////////////////////////////////////////////////////

    double xL = 0.0;
    double yL = 0.0;
    double xU = 2.0;
    double yU = 2.0;

    double thetaL = -10.0;
    double thetaU = 10.0;

    double x0 = 0.0;
    double y0 = 0.0;
    double xf = 1.2;
    double yf = 1.6;


    problem.phases(1).bounds.lower.states(1) = xL;
```

```
    problem.phases(1).bounds.lower.states(2) = yL;

    problem.phases(1).bounds.upper.states(1) = xU;
    problem.phases(1).bounds.upper.states(2) = yU;


    problem.phases(1).bounds.lower.controls(1) = thetaL;
    problem.phases(1).bounds.upper.controls(1) = thetaU;

    problem.phases(1).bounds.lower.events(1) = x0;
    problem.phases(1).bounds.lower.events(2) = y0;
    problem.phases(1).bounds.lower.events(3) = xf;
    problem.phases(1).bounds.lower.events(4) = yf;


    problem.phases(1).bounds.upper.events(1) = x0;
    problem.phases(1).bounds.upper.events(2) = y0;
    problem.phases(1).bounds.upper.events(3) = xf;
    problem.phases(1).bounds.upper.events(4) = yf;

    problem.phases(1).bounds.lower.path(1) = 0.1;
    problem.phases(1).bounds.upper.path(1) = 100.0;

    problem.phases(1).bounds.lower.path(2) = 0.1;
    problem.phases(1).bounds.upper.path(2) = 100.0;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.0;
    problem.phases(1).bounds.upper.EndTime      = 1.0;



////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  ////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae             = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ////////////////////
////////////////////////////////////////////////////////////////////////

    int nnodes     = 30;
    int ncontrols                    = problem.phases(1).ncontrols;
    int nstates                      = problem.phases(1).nstates;

    DMatrix u_guess    =  zeros(ncontrols,nnodes);
    DMatrix x_guess    =  zeros(nstates,nnodes);
    DMatrix time_guess =  linspace(0.0,1.0,nnodes);


    x_guess(1,colon()) = linspace(x0,xf,nnodes);
    x_guess(2,colon()) = linspace(y0,yf,nnodes);

    u_guess(1,colon()) = zeros(1,nnodes);

    problem.phases(1).guess.controls       = u_guess;
    problem.phases(1).guess.states         = x_guess;
    problem.phases(1).guess.time           = time_guess;


////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ///////////////////////////
////////////////////////////////////////////////////////////////////////
    algorithm.nlp_iter_max            = 1000;
    algorithm.nlp_tolerance           = 1.e-4;
    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
    algorithm.collocation_method      = "trapezoidal";
    algorithm.mesh_refinement         = "automatic";
    algorithm.ode_tolerance           = 1.0e-2;


////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem  ////////////////
```

```
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////

    DMatrix states = solution.get_states_in_phase(1);
    DMatrix theta  = solution.get_controls_in_phase(1);
    DMatrix t      = solution.get_time_in_phase(1);
    DMatrix mu     = solution.get_dual_path_in_phase(1);
    DMatrix lambda = solution.get_dual_costates_in_phase(1);

    DMatrix x = states(1,colon());
    DMatrix y = states(2,colon());

////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("obstacle_x.dat");
    y.Save("obstacle_y.dat");
    theta.Save("obstacle_theta.dat");
    t.Save("obstacle_t.dat");


////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////

    DMatrix alpha = colon(0.0, pi/20, 2*pi);

    DMatrix xObs1 = sqrt(0.1)*cos(alpha) + 0.4;
    DMatrix yObs1 = sqrt(0.1)*sin(alpha) + 0.5;

    DMatrix xObs2 = sqrt(0.1)*cos(alpha) + 0.8;
    DMatrix yObs2 = sqrt(0.1)*sin(alpha) + 1.5;

    plot(x,y,xObs1,yObs1,xObs2,yObs2,problem.name+": x-y trajectory",
                                        "x", "y", "y obs1 obs2");

    plot(x,y,xObs1,yObs1,xObs2,yObs2,problem.name+": x-y trajectory",
                                        "x", "y", "y obs1 obs2",
                                        "pdf", "obstacle_xy.pdf");

    plot(t,theta, problem.name+": theta","t", "theta");

    plot(t,mu, problem.name+": path constraint multipliers","t", "mu_1 mu_2");

    plot(t,lambda, problem.name+": costates","t", "lambda_1 lambda_2");

}
////////////////////////////////////////////////////////////////////////
/////////////////////////      END OF FILE      ///////////////////////////
////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figure 3.83, which illustrates the optimal (x, y) trajectory of the particle.

```
PSOPT results summary
=====================


Problem:   Obstacle avoidance problem
CPU time (seconds): 1.377402e+00
NLP solver used:   IPOPT
```
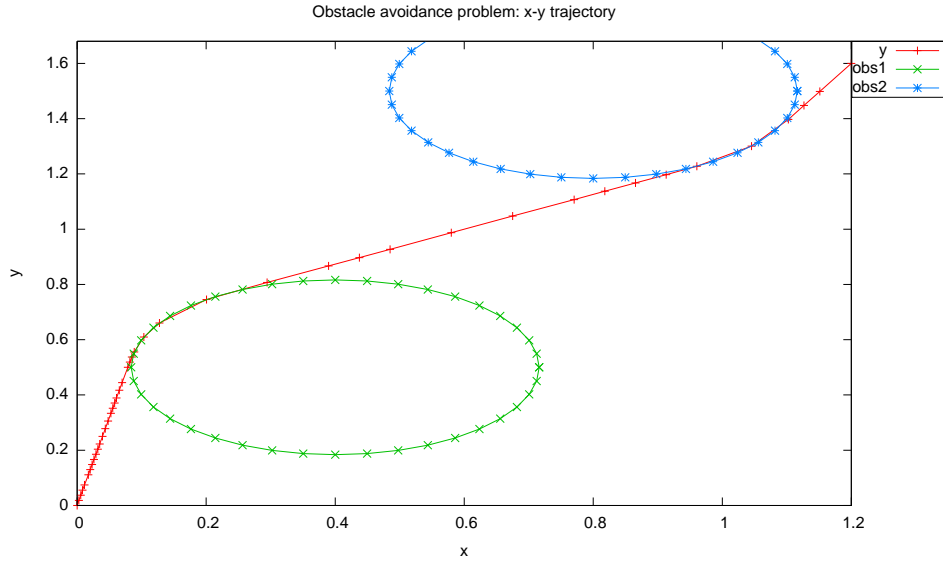
Figure 3.83: Optimal (x, y) trajectory for obstacle avoidance problem

```
Optimal (unscaled) cost function value:  4.571044e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  4.571044e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 6.468613e-03
NLP solver reports:  The problem solved!
```

3.34   Reorientation of an asymmetric rigid body

Consider the following optimal control problem, which consists of the reorientation of an asymmetric rigid body in minimum time [4]. Find t_f, $\hat{\mathbf{u}}(t) = [u_1(t), u_2(t), u_3(t), q_4(t)]^T$ to minimize the cost functional

$$J = t_f \tag{3.135}$$

subject to the dynamic constraints

$$\dot{q}_1 = \frac{1}{2}\left[\omega_1 q_4 - \omega_2 q_3 + \omega_3 q_2\right]$$

$$\dot{q}_2 = \frac{1}{2}\left[\omega_1 q_3 + \omega_2 q_4 - \omega_3 q_1\right]$$

$$\dot{q}_3 = \frac{1}{2}\left[-\omega_1 q_2 + \omega_2 q_1 + \omega_3 q_4\right]$$

$$\dot{\omega}_1 = \frac{u_1}{I_x} - \left[\frac{I_z - I_y}{I_x}\omega_2\omega_3\right] \qquad (3.136)$$

$$\dot{\omega}_2 = \frac{u_2}{I_y} - \left[\frac{I_x - I_z}{I_y}\omega_1\omega_3\right]$$

$$\dot{\omega}_3 = \frac{u_3}{I_z} - \left[\frac{I_y - I_x}{I_z}\omega_1\omega_2\right]$$

The path constraint:

$$0 = q_1^2 + q_2^2 + q_3^2 + q_4^2 - 1 \qquad (3.137)$$

the boundary conditions:

$$q_1(0) = 0,$$
$$q_2(0) = 0,$$
$$q_3(0) = 0,$$
$$q_4(0) = 1.0$$
$$q_1(t_f) = \sin\frac{\phi}{2},$$
$$q_2(t_f) = 0,$$
$$q_3(t_f) = 0,$$
$$q_4(t_f) = \cos\frac{\phi}{2} \qquad (3.138)$$
$$\omega_1(0) = 0,$$
$$\omega_2(0) = 0,$$
$$\omega_3(0) = 0,$$
$$\omega_1(t_f) = 0,$$
$$\omega_2(t_f) = 0,$$
$$\omega_3(t_f) = 0,$$

where $\phi = 150\,\text{deg}$ is the Euler axis rotation angle, $\mathbf{q} = [q_1, q_2, q_3, q_4]^T$ is the quarternion vector, $\omega = [\omega_1, \omega_2, \omega_3]^T$ is the angular velocity vector, and $\mathbf{u} = [u_1, u_2, u_3]^T$ is the control vector. Note that in the implementation, variable $q_4(t)$ is treated as an algebraic variable (i.e. as a control variable).

The variable bounds and other parameters are given in the code. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
///////////////////////////////////////////////////////////////////
////////////////            reorientation.cxx        ////////////////////
///////////////////////////////////////////////////////////////////
////////////////             PSOPT example           ////////////////////
///////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////
//////// Title: Reorientation of an asymmetric rigid body ////////////////
//////// Last modified: 03 July 2010                     ////////////////
//////// Reference:    Fleming et al                     ////////////////
////////                                                 ////////////////
///////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2010       ////////////////
///////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
///////////////////////////////////////////////////////////////////

#include "psopt.h"


///////////////////////////////////////////////////////////////////
//////////////////// Define the end point (Mayer) cost function //////////
///////////////////////////////////////////////////////////////////


adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   return tf;
}

///////////////////////////////////////////////////////////////////
//////////////////// Define the integrand (Lagrange) cost function  //////
///////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
   return 0.0;
}


///////////////////////////////////////////////////////////////////
//////////////////// Define the DAE's ////////////////////////////////////
///////////////////////////////////////////////////////////////////


void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

    adouble u1  = controls[ CINDEX(1) ];
    adouble u2          = controls[ CINDEX(2) ];
    adouble u3          = controls[ CINDEX(3) ];
    adouble q4          = controls[ CINDEX(4) ];

    adouble q1  = states[ CINDEX(1) ];
    adouble q2  = states[ CINDEX(2) ];
    adouble q3  = states[ CINDEX(3) ];
    adouble omega1  = states[ CINDEX(4) ];
    adouble omega2      = states[ CINDEX(5) ];
    adouble omega3      = states[ CINDEX(6) ];

    double Ix = 5621.0;
    double Iy = 4547.0;
    double Iz = 2364.0;

    adouble dq1     = 0.5*( omega1*q4 - omega2*q3 + omega3*q2 );
    adouble dq2     = 0.5*( omega1*q3 + omega2*q4 - omega3*q1 );
    adouble dq3     = 0.5*(-omega1*q2 + omega2*q1 + omega3*q4 );
    adouble domega1 = u1/Ix - ((Iz-Iy)/Ix)*omega2*omega3;
    adouble domega2 = u2/Iy - ((Ix-Iz)/Iy)*omega1*omega3;
    adouble domega3 = u3/Iz - ((Iy-Ix)/Iz)*omega1*omega2;
```

```
    derivatives[ CINDEX(1) ] =    dq1;
    derivatives[ CINDEX(2) ] =    dq2;
    derivatives[ CINDEX(3) ] =    dq3;
    derivatives[ CINDEX(4) ] =    domega1;
    derivatives[ CINDEX(5) ] =    domega2;
    derivatives[ CINDEX(6) ] =    domega3;


    path[ CINDEX(1) ] = q1*q1 + q2*q2 + q3*q3 + q4*q4 - 1.0;

}

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function //////////////////////////////
//////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{

    adouble q1i  = initial_states[ CINDEX(1) ];
    adouble q2i  = initial_states[ CINDEX(2) ];
    adouble q3i  = initial_states[ CINDEX(3) ];
    adouble omega1i  = initial_states[ CINDEX(4) ];
    adouble omega2i     = initial_states[ CINDEX(5) ];
    adouble omega3i     = initial_states[ CINDEX(6) ];

    adouble initial_controls[4], final_controls[4], q4i, q4f;

    get_initial_controls( initial_controls, xad, iphase, workspace );

    get_final_controls(   final_controls  , xad, iphase, workspace );

    q4i = initial_controls[ CINDEX(4) ];

    q4f = final_controls[   CINDEX(4) ];

    adouble q1f  = final_states[ CINDEX(1) ];
    adouble q2f  = final_states[ CINDEX(2) ];
    adouble q3f  = final_states[ CINDEX(3) ];
    adouble omega1f  = final_states[ CINDEX(4) ];
    adouble omega2f     = final_states[ CINDEX(5) ];
    adouble omega3f     = final_states[ CINDEX(6) ];


    e[ CINDEX(1)  ]  = q1i;
    e[ CINDEX(2)  ]      = q2i;
    e[ CINDEX(3)  ]      = q3i;
    e[ CINDEX(4)  ]     = q4i;
    e[ CINDEX(5)  ]    = omega1i;
    e[ CINDEX(6)  ]      = omega2i;
    e[ CINDEX(7)  ]   = omega3i;

    e[ CINDEX(8)  ]  = q1f;
    e[ CINDEX(9)  ]      = q2f;
    e[ CINDEX(10) ]      = q3f;
    e[ CINDEX(11) ]      = q4f;
    e[ CINDEX(12) ]    = omega1f;
    e[ CINDEX(13) ]      = omega2f;
    e[ CINDEX(14) ]   = omega3f;


}



//////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function //////////////////
//////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{

   // Single phase problem

}


//////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine //////////////////////////////
```

```
/////////////////////////////////////////////////////////////////////

int main(void)
{

/////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures  ///////////////////////////
/////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

/////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ///////////////////////////
/////////////////////////////////////////////////////////////////////

    problem.name          = "Reorientation of a rigid body";
    problem.outfilename                = "reorientation.txt";

/////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
/////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                 = 0;

    psopt_level1_setup(problem);

/////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
/////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 6;
    problem.phases(1).ncontrols  = 4;
    problem.phases(1).nevents    = 14;
    problem.phases(1).npath      = 1;
    problem.phases(1).nodes                = "[60]";


    psopt_level2_setup(problem, algorithm);



/////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ///////////////////////
/////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states = "[-1.0    -1.0    -1.0   -0.5    -0.5 -0.5]";
    problem.phases(1).bounds.upper.states = "[ 1.0     1.0     1.0    0.5     0.5   0.5]";


    problem.phases(1).bounds.lower.controls = "[-50.0  -50.0 -50.0  -1.0]";

    problem.phases(1).bounds.upper.controls = "[ 50.0   50.0  50.0   1.0]";


    problem.phases(1).bounds.lower.path(1) = 0.000;

    problem.phases(1).bounds.upper.path(1) =  0.000;

    DMatrix q0, qf, omega0, omegaf;

    double phi = 150.0*(pi/180.0);

    q0(1) = 0.0; q0(2) = 0.0; q0(3)=0.0; q0(4)=1.0;
    qf(1) = sin(phi/2.0); qf(2) = 0.0; qf(3) = 0.0; qf(4) = cos(phi/2.0);

    omega0 = zeros(1,3); omegaf = zeros(1,3);

    problem.phases(1).bounds.lower.events = q0 || omega0 || qf || omegaf;
    problem.phases(1).bounds.upper.events = problem.phases(1).bounds.lower.events;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 25.0;
    problem.phases(1).bounds.upper.EndTime      = 35.0;
```

```
/////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  //////////////////////
/////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae             = &dae;
    problem.events = &events;
    problem.linkages = &linkages;

/////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ///////////////////
/////////////////////////////////////////////////////////////////////

    int nnodes      = problem.phases(1).nodes(1);
    int ncontrols                       = problem.phases(1).ncontrols;
    int nstates                         = problem.phases(1).nstates;

    DMatrix state_guess    =  zeros(nstates,nnodes);
    DMatrix control_guess  =  50.0*ones(ncontrols,nnodes);
    DMatrix time_guess     =  linspace(0.0,40,nnodes);


    state_guess(1, colon() ) = linspace( q0(1), qf(1), nnodes );
    state_guess(2, colon() ) = linspace( q0(2), qf(2), nnodes );
    state_guess(3, colon() ) = linspace( q0(3), qf(3), nnodes );
    control_guess(4, colon() )=linspace( q0(4), qf(4), nnodes );
    state_guess(4, colon() ) = linspace( omega0(1), omegaf(1), nnodes );
    state_guess(5, colon() ) = linspace( omega0(2), omegaf(2), nnodes );
    state_guess(6, colon() ) = linspace( omega0(3), omegaf(3), nnodes );

    problem.phases(1).guess.states   = state_guess;
    problem.phases(1).guess.controls = control_guess;
    problem.phases(1).guess.time     = time_guess;



/////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  //////////////////////////
/////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max           = 1000;
    algorithm.nlp_tolerance          = 1.e-6;
    algorithm.nlp_method             = "IPOPT";
    algorithm.scaling                = "automatic";
    algorithm.derivatives            = "automatic";
    algorithm.collocation_method     = "trapezoidal";
    algorithm.mesh_refinement        = "automatic";
    algorithm.ode_tolerance          = 1.e-5;



/////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   ///////////////////
/////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

/////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
/////////////////////////////////////////////////////////////////////


    DMatrix states, controls, t, q1, q2, q3, q4, omega, u, q;

    states      = solution.get_states_in_phase(1);
    controls    = solution.get_controls_in_phase(1);
    t           = solution.get_time_in_phase(1);

    q1 = states(1,colon());
    q2 = states(2,colon());
    q3 = states(3,colon());
    q4 = controls(4, colon());

    q = q1 && q2 && q3 && q4;

    omega = states(colon(4,6), colon());

    u = controls(colon(1,3), colon());
```

```
//////////////////////////////////////////////////////////////////////////
//////////   Save solution data to files if desired //////////////////////
//////////////////////////////////////////////////////////////////////////

    states.Save("states.dat");
    controls.Save("controls.dat");
    t.Save("t.dat");


//////////////////////////////////////////////////////////////////////////
//////////   Plot some results if desired (requires gnuplot) //////////////
//////////////////////////////////////////////////////////////////////////


    multiplot(t,q,problem.name+": quarternion","time (s)", "q1 q2 q3 q4", "q1 q2 q3 q4", 2, 2);

    multiplot(t,u, problem.name+": controls", "time (s)", "u1 u2 u3", "u1 u2 u3");


    multiplot(t,q,problem.name+": quarternion","time (s)", "q1 q2 q3 q4", "q1 q2 q3 q4", 2, 2,
              "pdf", "reorientation_q.pdf");

    multiplot(t,u, problem.name+": controls", "time (s)", "u1 u2 u3", "u1 u2 u3", 3, 1,
              "pdf", "reorientation_u.pdf");


}

//////////////////////////////////////////////////////////////////////////
////////////////////////      END OF FILE    //////////////////////////////
//////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.84 to 3.85, which contain the elements of the quarternion vector $\mathbf{q}$, and the control vector $\mathbf{u} = [u_1, u_2, u_3]^T$, respectively.

```
PSOPT results summary
=====================


Problem:  Reorientation of a rigid body
CPU time (seconds): 9.865000e+01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  2.863042e+01
Phase 1 endpoint cost function value: 2.863042e+01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.863042e+01
Phase 1 maximum relative local error: 9.145819e-06
NLP solver reports:  The problem solved!
```

3.35   Shuttle re-entry problem

Consider the following optimal control problem, which is known in the literature as the shuttle re-entry problem [3]. Find t_f, $\alpha(t)$ and $\beta(t) \in [0, t_f]$

330
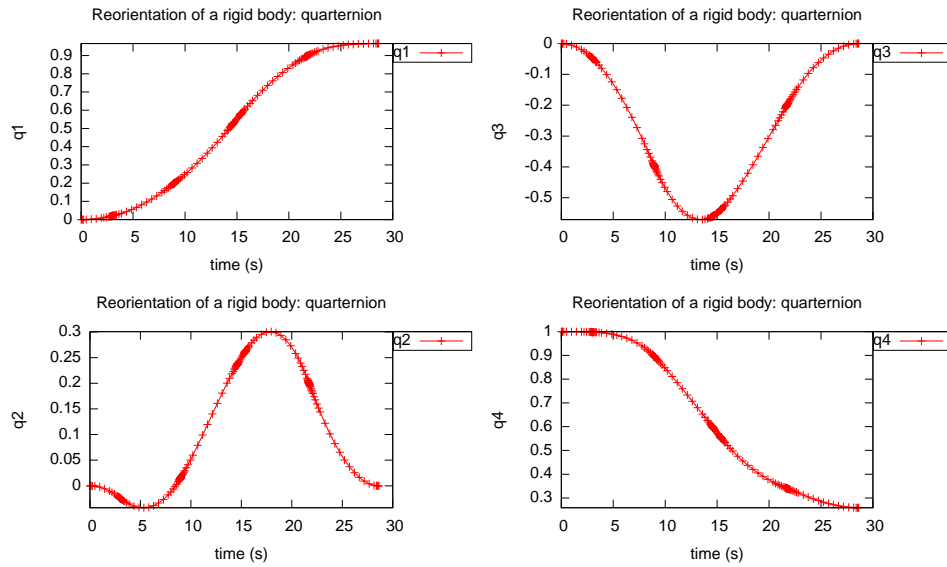
Figure 3.84: Quarternion vector elements for the reorientation problem
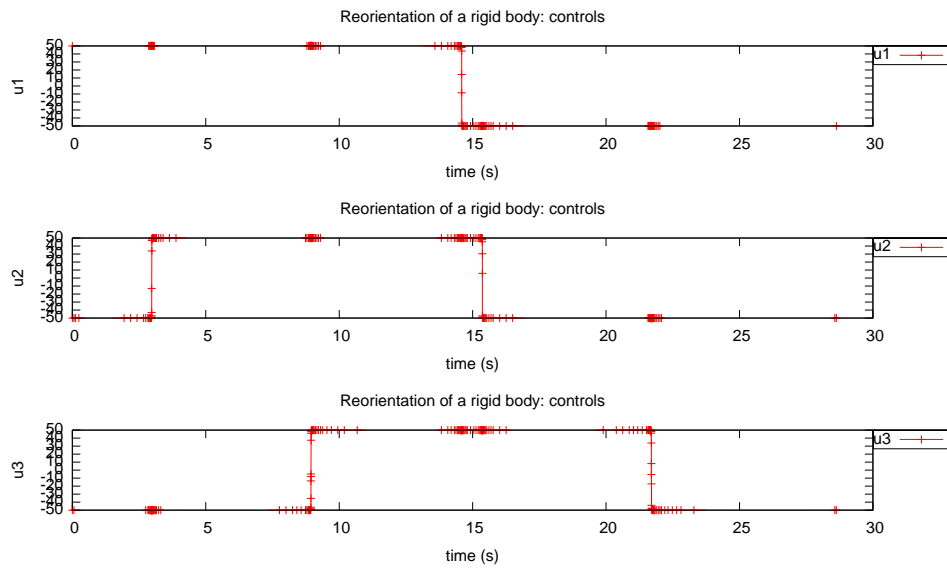
Figure 3.85: Control vector elements for the reorientation problem

331

to minimize the cost functional

$$J = -\frac{180}{\pi}\theta(t_f) \tag{3.139}$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{h} &= v\sin(\gamma) \\
\dot{\phi} &= \frac{v}{r}\cos(\gamma)\sin(\psi)/\cos(\theta) \\
\dot{m} &= \frac{v}{r}\cos(\gamma)\cos(\psi) \\
\dot{v} &= -\frac{D}{m} - g\sin(\gamma) \\
\dot{\gamma} &= \frac{L}{mv}\cos(\beta) + \cos(\gamma)(\frac{v}{r} - \frac{g}{v}) \\
\dot{\psi} &= \frac{1}{mv\cos(\gamma)}L\sin(\beta) + \frac{v}{r\cos(\theta)}\cos(\gamma)\sin(\psi)\sin(\theta)
\end{aligned}
\tag{3.140}
$$

the boundary conditions:

$$
\begin{aligned}
h(0) &= 260000.0 \\
\phi(0) &= -0.6572 \\
\theta(0) &= 0.0 \\
v(0) &= 25600.0 \\
\gamma(0) &= -0.0175 \\
h(t_f) &= 80000.0 \\
v(t_f) &= 2500.0 \\
\gamma(t_f) &= -0.0873
\end{aligned}
\tag{3.141}
$$

The variable bounds and other parameters are given in the code. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
//////////////////       shuttle_reentry1.cxx       /////////////////////
//////////////////////////////////////////////////////////////////////
////////////////              PSOPT  Example            /////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:          Shuttle reentry problem        //////////////
//////// Last modified: 05 January 2009                 //////////////
//////// Reference:      Betts (2001)                   //////////////
//////// (See PSOPT handbook for full reference)        //////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009     //////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////
//////// is distributed under the terms of the GNU Lesser  //////////////
//////// General Public License (LGPL)                  //////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
//////////////////    Define some local macros      /////////////////////
//////////////////////////////////////////////////////////////////////


#define H_INDX    1
#define PHI_INDX  2
#define THETA_INDX  3
#define V_INDX 4
#define GAMMA_INDX 5
#define PSI_INDX 6
```

```
#define ALPHA_INDX     1
#define BETA_INDX 2

#define DEG2RAD(x)  (3.141592653589793*(x)/180.0)


/////////////////////////////////////////////////////////////////////
/////////////////////  Define the end point (Mayer) cost function //////////
/////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    adouble theta = final_states[THETA_INDX-1];

    return (-theta*180/3.141592653589793);
}

/////////////////////////////////////////////////////////////////////
/////////////////////  Define the integrand (Lagrange) cost function  //////
/////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters,
                       adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    return  0.0;
}

/////////////////////////////////////////////////////////////////////
/////////////////////  Define the DAE's ///////////////////////////////////
/////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

adouble alpha, beta;


adouble alt    = states[H_INDX-1];
adouble lon    = states[PHI_INDX-1];
adouble lat    = states[THETA_INDX-1];
adouble vel    = states[V_INDX-1];
adouble gamma  = states[GAMMA_INDX-1];
adouble azi    = states[PSI_INDX-1];


alpha = controls[ALPHA_INDX-1];
beta  = controls[BETA_INDX-1];


double pi      = 3.141592653589793;
double cr2d    = 180.0/pi;
double weight  = 203000.0;
double cm2w    = 32.174;
double cea     = 20902900.0;
double mu      = 0.14076539e17;
double rho0    = 0.002378;
double href    = 23800.0;
double cl0     = -0.20704;
double cl1     = 0.029244;
double cd0     = 0.07854;
double cd1     = -6.1592e-3;
double cd2     = 6.21408e-4;
double sref    = 2690.0;


double mass    = weight/cm2w;
adouble sgamma  = sin(gamma);
adouble cgamma  = cos(gamma);
adouble sazi    = sin(azi);
adouble cazi    = cos(azi);
adouble sbeta   = sin(beta);
adouble cbeta   = cos(beta);
adouble slat    = sin(lat);
adouble clat    = cos(lat);

adouble alphad  = cr2d*alpha;
adouble radius  = cea+alt;
```

```
adouble grav     = mu/pow(radius,2);

adouble rhodns   = rho0*exp(-alt/href);
adouble dynp     = 0.5*(rhodns*pow(vel,2));
adouble subl     = cl0+cl1*alphad;
adouble subd     = cd0+((cd1+cd2*alphad)*alphad);
adouble drag     = (dynp*subd)*sref;
adouble lift     = (dynp*subl)*sref;
adouble vrelg    = (vel/radius)-(grav/vel);

adouble d_alt_dt   = vel*sgamma;
adouble d_lon_dt   = ((vel*cgamma)*sazi)/(radius*clat);
adouble d_lat_dt   = ((vel*cgamma)*cazi)/radius;
adouble d_vel_dt   = (-(drag/mass)-(grav*sgamma));
adouble d_gamma_dt = ((lift*cbeta)/(mass*vel))+(cgamma*vrelg);
adouble d_azi_dt   = ((lift*sbeta)/((mass*vel)*cgamma))
                             + ((vel*cgamma)*(sazi*slat)/(radius*clat));

derivatives[H_INDX-1]   = d_alt_dt      ;
derivatives[PHI_INDX-1] = d_lon_dt      ;
derivatives[THETA_INDX-1] = d_lat_dt      ;
derivatives[V_INDX-1]   = d_vel_dt      ;
derivatives[GAMMA_INDX-1] = d_gamma_dt    ;
derivatives[PSI_INDX-1] = d_azi_dt      ;

//    adouble alpha_hat = 180.0*alpha/pi;

//    adouble qa  = c0  + c1*alpha_hat + c2*pow(alpha_hat,2) + c3*pow(alpha_hat,3);

//    adouble qr = (17700.0*sqrt(rhodns))*pow(0.0001*v,3.07);

//    adouble q = qa*qr;

//    path[0] = q;


}

//////////////////////////////////////////////////////////////////////
///////////////////  Define the events function //////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble h0  = initial_states[H_INDX-1];
   adouble phi0  = initial_states[PHI_INDX-1];
   adouble theta0  = initial_states[THETA_INDX-1];
   adouble v0  = initial_states[V_INDX-1];
   adouble gamma0  = initial_states[GAMMA_INDX-1];
   adouble psi0        = initial_states[PSI_INDX-1];

   adouble hf          = final_states[H_INDX-1];
   adouble vf          = final_states[V_INDX-1];
   adouble gammaf      = final_states[GAMMA_INDX-1];

   e[H_INDX-1]  = h0;
   e[PHI_INDX-1]  = phi0;
   e[THETA_INDX-1] = theta0;
   e[V_INDX-1]  = v0;
   e[GAMMA_INDX-1] = gamma0;
   e[PSI_INDX-1]  = psi0;

   e[6]  = hf;
   e[7]  = vf;
   e[8]  = gammaf;
}




//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////
```

```
//////////////////////////////////////////////////////////////////////
int main(void)
{
//////////////////////////////////////////////////////////////////////
/////////////////  Declare key structures ///////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
/////////////////  Register problem name  ///////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name         = "Shuttle re-entry problem";
    problem.outfilename              = "shuttle.txt";

//////////////////////////////////////////////////////////////////////
///////////   Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages               = 0;

    psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 6;
    problem.phases(1).ncontrols  = 2;
    problem.phases(1).nevents    = 9;
    problem.phases(1).npath             = 0;
    problem.phases(1).nodes                 = "[60 80]";
    problem.phases(1).zero_cost_integrand       = true;

    psopt_level2_setup(problem, algorithm);


//////////////////////////////////////////////////////////////////////
/////////////////  Declare DMatrix objects to store results /////////////
//////////////////////////////////////////////////////////////////////

    DMatrix x, u, t;

//////////////////////////////////////////////////////////////////////
/////////////////  Enter problem bounds information ////////////////////
//////////////////////////////////////////////////////////////////////

    double hL     = 0.0;
    double hU     = 300000.0;
    double phiL   = DEG2RAD(-45.0);
    double phiU   = DEG2RAD( 45.0);
    double vL     = 1000.0  ;
    double vU     = 40000.0      ;
    double thetaL = DEG2RAD(-89.0) ;
    double thetaU = DEG2RAD( 89.0) ;
    double gammaL = DEG2RAD(-89.0)  ;
    double gammaU = DEG2RAD(89.0)  ;
    double psiL   = DEG2RAD(-180.0);
    double psiU   = DEG2RAD(180.0) ;

    double alphaL = DEG2RAD(-89.0) ;
    double alphaU = DEG2RAD( 89.0) ;
    double betaL  = DEG2RAD(-90.0) ;
    double betaU  = DEG2RAD( 1.0)  ;

    double qU     = 70.0;
    double qL     = -INF;


    double h0     =  260000.0     ;
    double v0     =  25600.0        ;
    double phi0   =  DEG2RAD(-0.5*75.3153) ;
    double gamma0 =  DEG2RAD(-1.0);
    double theta0 =  0.0           ;
    double psi0   =  DEG2RAD(90.0) ;

    double hf     = 80000.0     ;
    double vf     = 2500.0        ;
```

335

```
    double gammaf  = DEG2RAD(-5.0);


    double pi = 3.141592653589793;


    problem.phases(1).bounds.lower.states(H_INDX)   = hL;
    problem.phases(1).bounds.lower.states(PHI_INDX)  = phiL;
    problem.phases(1).bounds.lower.states(THETA_INDX) = thetaL;
    problem.phases(1).bounds.lower.states(V_INDX)   = vL;
    problem.phases(1).bounds.lower.states(GAMMA_INDX) = gammaL;
    problem.phases(1).bounds.lower.states(PSI_INDX)  = psiL;


    problem.phases(1).bounds.upper.states(H_INDX)   = hU;
    problem.phases(1).bounds.upper.states(PHI_INDX)  = phiU;
    problem.phases(1).bounds.upper.states(THETA_INDX) = thetaU;
    problem.phases(1).bounds.upper.states(V_INDX)   = vU;
    problem.phases(1).bounds.upper.states(GAMMA_INDX) = gammaU;
    problem.phases(1).bounds.upper.states(PSI_INDX)  = psiU;

    problem.phases(1).bounds.lower.controls(ALPHA_INDX) = alphaL;
    problem.phases(1).bounds.upper.controls(ALPHA_INDX) = alphaU;
    problem.phases(1).bounds.lower.controls(BETA_INDX) = betaL;
    problem.phases(1).bounds.upper.controls(BETA_INDX) = betaU;


    problem.phases(1).bounds.lower.events(H_INDX)   = h0;
    problem.phases(1).bounds.lower.events(PHI_INDX)  = phi0;
    problem.phases(1).bounds.lower.events(THETA_INDX) = theta0;
    problem.phases(1).bounds.lower.events(V_INDX)   = v0;
    problem.phases(1).bounds.lower.events(GAMMA_INDX) = gamma0;
    problem.phases(1).bounds.lower.events(PSI_INDX)  = psi0;
    problem.phases(1).bounds.lower.events(7)   = hf;
    problem.phases(1).bounds.lower.events(8)   = vf;
    problem.phases(1).bounds.lower.events(9)   = gammaf;


    problem.phases(1).bounds.upper.events(H_INDX)   = h0;
    problem.phases(1).bounds.upper.events(PHI_INDX)  = phi0;
    problem.phases(1).bounds.upper.events(THETA_INDX) = theta0;
    problem.phases(1).bounds.upper.events(V_INDX)   = v0;
    problem.phases(1).bounds.upper.events(GAMMA_INDX) = gamma0;
    problem.phases(1).bounds.upper.events(PSI_INDX)  = psi0;
    problem.phases(1).bounds.upper.events(7)   = hf;
    problem.phases(1).bounds.upper.events(8)   = vf;
    problem.phases(1).bounds.upper.events(9)   = gammaf;


    problem.phases(1).bounds.lower.path(1)              = qL;
    problem.phases(1).bounds.upper.path(1)  = qU;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime     = 100.0;
    problem.phases(1).bounds.upper.EndTime     = 4000.0;


////////////////////////////////////////////////////////////////////////
////////////////////  Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;


////////////////////////////////////////////////////////////////////////
////////////////////  Define & register initial guess /////////////////////////
////////////////////////////////////////////////////////////////////////

    int nodes = (int) problem.phases(1).nodes(1);

    DMatrix u_guess = ( zeros(1,nodes+1) && (DEG2RAD(1.0)*ones(1,nodes+1) ) );
    DMatrix x_guess = zeros(6,nodes+1);
    DMatrix time_guess;

    x_guess(H_INDX,colon())  = linspace(h0, hf, nodes+1);
```

```
    x_guess(PHI_INDX,colon())  = -0.5*DEG2RAD(90.0)*ones(1,nodes+1);
    x_guess(THETA_INDX,colon()) = DEG2RAD(-89.0)*ones(1,nodes+1);
    x_guess(V_INDX,colon())   = linspace(v0,vf, nodes+1);
    x_guess(GAMMA_INDX,colon()) = linspace(gamma0, gammaf, nodes+1);
    x_guess(PSI_INDX,colon())  = linspace(pi/2, -pi/2, nodes+1);

    time_guess = linspace(0.0, 1000.0, nodes+1);

    problem.phases(1).guess.controls = u_guess;
    problem.phases(1).guess.states   = x_guess;
    problem.phases(1).guess.time     = time_guess;



////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ////////////////////////////
////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method               = "IPOPT";
    algorithm.nlp_iter_max             = 1000;
    algorithm.nlp_tolerance            = 5e-6;
    algorithm.scaling                  = "automatic";
    algorithm.derivatives          = "automatic";
    algorithm.collocation_method       = "trapezoidal";
    algorithm.mesh_refinement          = "automatic";

////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   ////////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   ////////
////////////////////////////////////////////////////////////////////////

    x = solution.get_states_in_phase(1);
    u = solution.get_controls_in_phase(1);
    t = solution.get_time_in_phase(1);

////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired //////////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) //////////////
////////////////////////////////////////////////////////////////////////

    DMatrix h     = x(1,colon());
    DMatrix phi   = x(2,colon());
    DMatrix theta = x(3,colon());
    DMatrix v     = x(4,colon());
    DMatrix gamma = x(5,colon());
    DMatrix psi   = x(6,colon());
    DMatrix alpha = u(1,colon());
    DMatrix beta  = u(2,colon());


    plot(t,x(1,colon()),problem.name, "time (s)", "x1","altitude");

    plot(t,u(1,colon()),problem.name,"time (s)", "alpha");

    plot(t,u(2,colon()),problem.name,"time (s)", "beta");

    plot(t,h, problem.name+": altitude", "time (s)", "h (ft)",
            "altitude","pdf", "shutt_alt.pdf" );

    plot(t,phi, problem.name+": longitude", "time (s)", "phi (rad)",
            "longitude","pdf", "shutt_lon.pdf" );

    plot(t,theta, problem.name+": latitude", "time (s)", "theta (rad)",
            "latitude","pdf", "shutt_lat.pdf" );

    plot(t,v, problem.name+": velocity", "time (s)", "v (ft/s)",
            "velocity","pdf", "shutt_vel.pdf" );

    plot(t,gamma, problem.name+": flight path angle", "time (s)", "gamma (rad)",
```

```
          "fpa","pdf", "shutt_fpa.pdf" );

    plot(t,psi, problem.name+": azimuth", "time (s)", "psi (rad)",
          "azi","pdf", "shutt_azi.pdf" );

    plot(t,alpha, problem.name+": alpha", "time (s)", "alpha (rad)",
          "alpha","pdf", "shutt_alpha.pdf" );

    plot(t,beta, problem.name+": beta", "time (s)", "beta (rad)",
          "beta","pdf", "shutt_beta.pdf" );


}
////////////////////////////////////////////////////////////////////////
////////////////////////       END OF FILE     ///////////////////////////
////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.86 to 3.93, which contain the elements of the state and the control vectors.

```
PSOPT results summary
=====================


Problem:  Shuttle re-entry problem
CPU time (seconds): 1.718845e+01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  -3.414119e+01
Phase 1 endpoint cost function value: -3.414119e+01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.008584e+03
Phase 1 maximum relative local error: 5.997880e-04
NLP solver reports:  The problem solved!
```

3.36   Singular control problem

Consider the following optimal control problem, whose solution is known to have a singular arc [30, 37]. Find $u(t), t \in [0, 1]$ to minimize the cost functional

$$J = \int_0^1 [x_1^2 + x_2^2 + 0.0005(x_2 + 16x_5 - 8 - 0.1x_3u^2)^2]dt \qquad (3.142)$$

subject to the dynamic constraints

$$\begin{aligned}
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= -x_3u + 16t - 8 \qquad (3.143) \\
\dot{x}_3 &= u
\end{aligned}$$
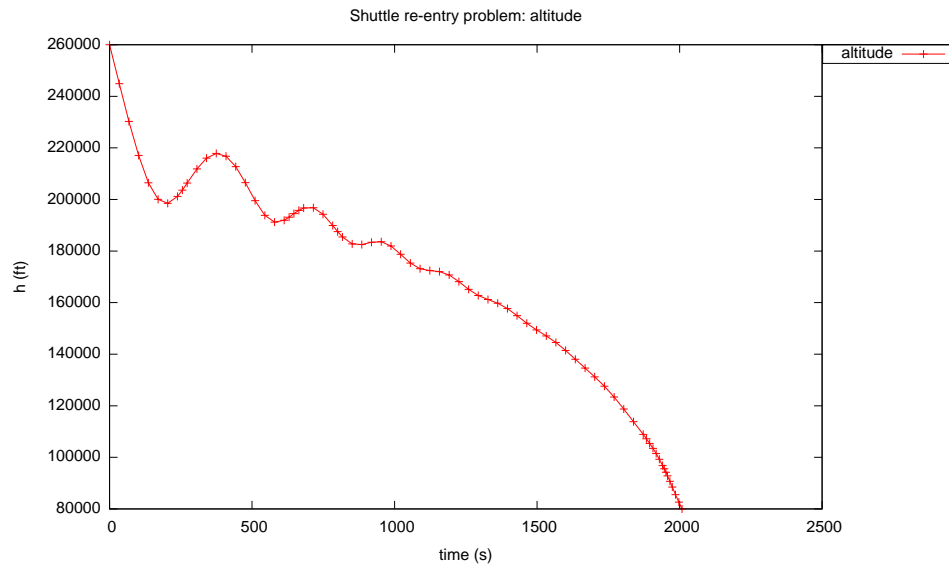
Figure 3.86: Altitude $h(t)$ for the shuttle re-entry problem
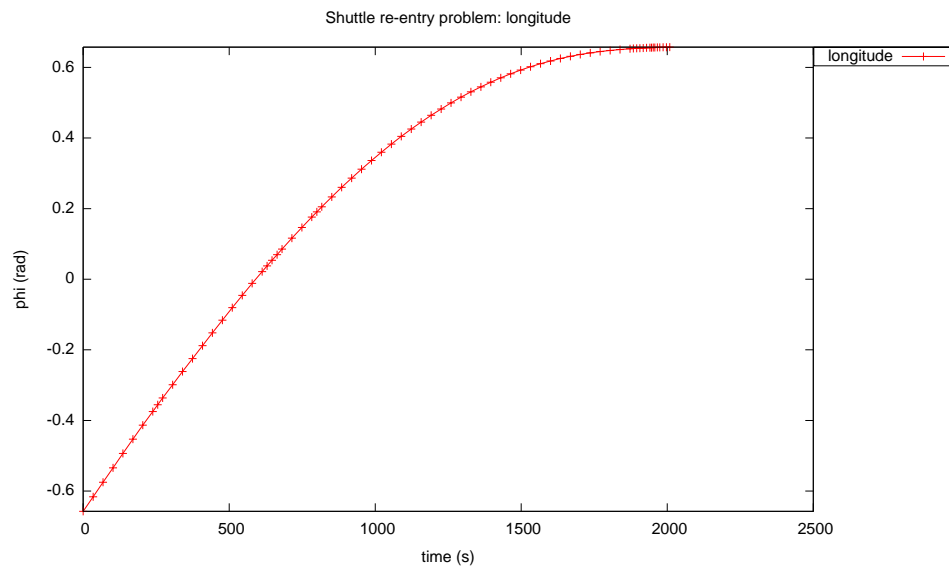
Figure 3.87: Longitude $\phi(t)$ for the shuttle re-entry problem

339
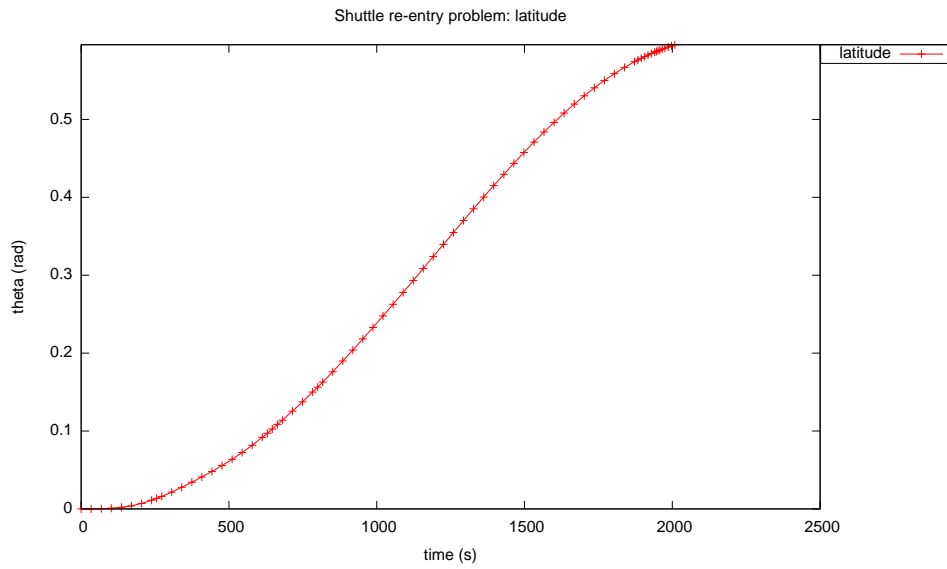
Figure 3.88: Latitude $\theta(t)$ for the shuttle re-entry problem
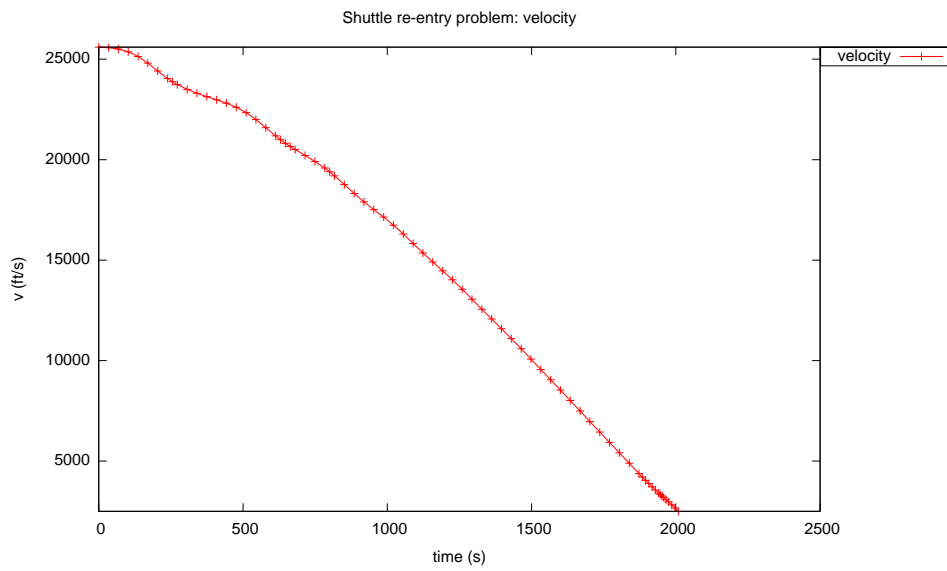
Figure 3.89: Velocity $v(t)$ for the shuttle re-entry problem
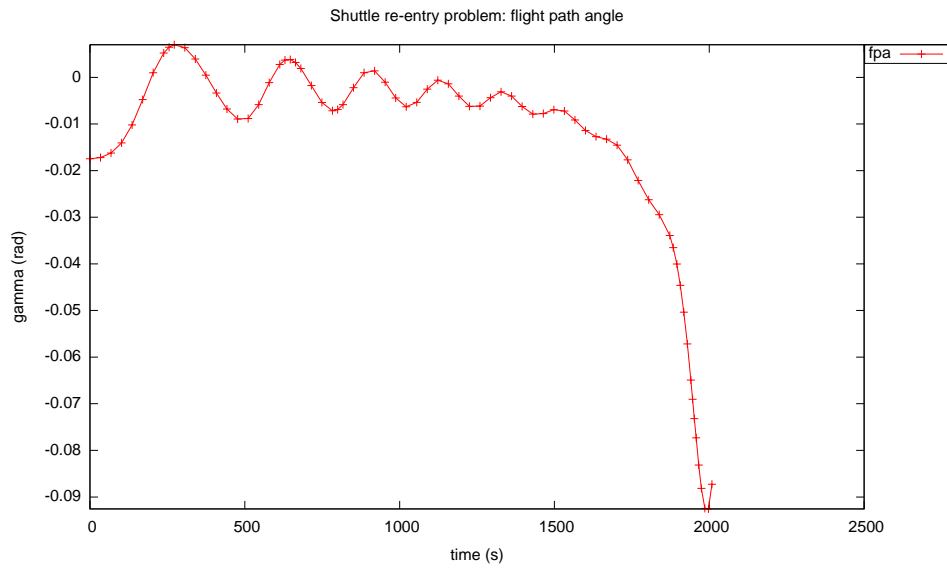
Figure 3.90: Flight path angle $\gamma(t)$ for the shuttle re-entry problem
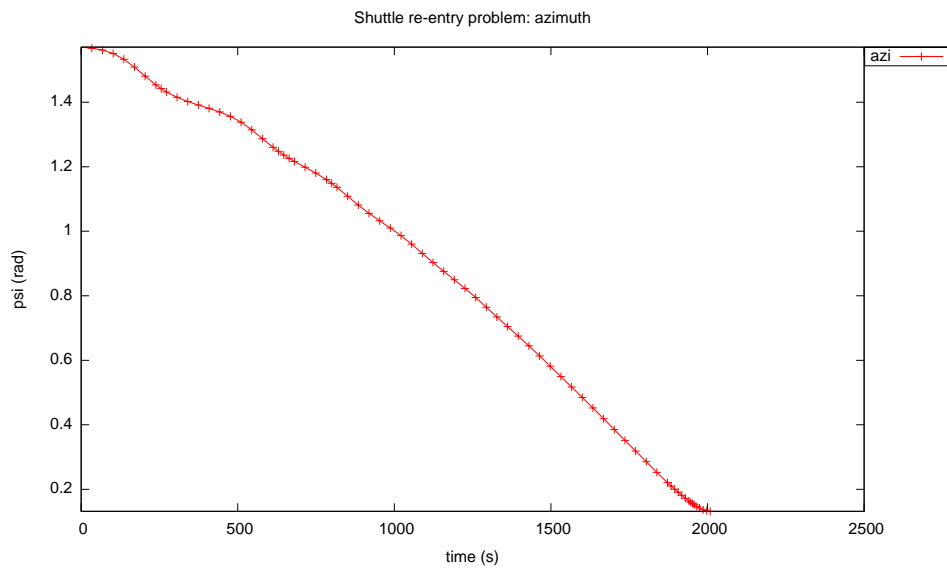
Figure 3.91: Azimuth $\psi(t)$ for the shuttle re-entry problem

341
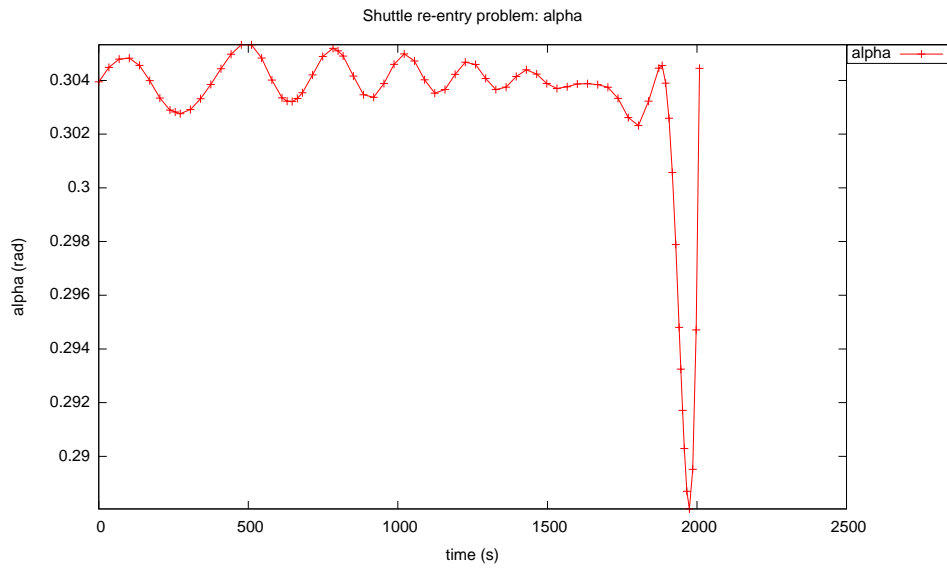
Figure 3.92: Angle of attack $\alpha(t)$ for the shuttle re-entry problem
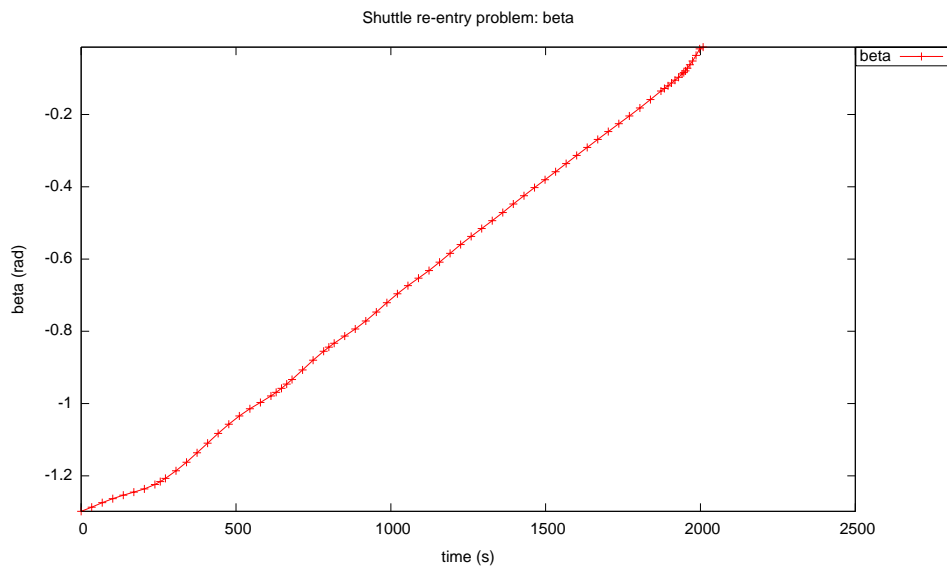
Figure 3.93: Bank angle $\beta(t)$ for the shuttle re-entry problem

342

the boundary conditions:

$$\begin{aligned} x_1(0) &= 0 \\ x_2(0) &= -1 \\ x_3(0) &= \sqrt{5} \end{aligned} \qquad (3.144)$$

and the control bounds

$$-4 \le u(t) \le 10 \qquad (3.145)$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
//////////////////          singular5.cxx          //////////////////
//////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example          ////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:                Singular problem       //////////////////
//////// Last modified:        09 January 2009        //////////////////
//////// Reference:            Luus (2002)            //////////////////
//////// (See PSOPT handbook for full reference)      //////////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009   //////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which /////////////
//////// is distributed under the terms of the GNU Lesser //////////////
//////// General Public License (LGPL)                //////////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"




//////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return 0.0;
}

//////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
   adouble x1 = states[ CINDEX(1) ];
   adouble x2 = states[ CINDEX(2) ];
   adouble x3 = states[ CINDEX(3) ];
   adouble u = controls[ CINDEX(1) ];
   adouble t = time;
   adouble L;

   L =  x1*x1 + x2*x2 + 0.0005*pow(x2+16*t-8.0-0.1*x3*u*u,2.0);

   return L;
}

//////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's /////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
```

```
    adouble x1 = states[ CINDEX(1) ];
    adouble x2 = states[ CINDEX(2) ];
    adouble x3 = states[ CINDEX(3) ];

    adouble u = controls[ CINDEX(1) ];
    adouble t = time;

    derivatives[ CINDEX(1) ] = x2;
    derivatives[ CINDEX(2) ] = -x3*u + 16*t - 8.0;
    derivatives[ CINDEX(3) ] =  u;

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
    adouble x10 = initial_states[ CINDEX(1) ];
    adouble x20 = initial_states[ CINDEX(2) ];
    adouble x30 = initial_states[ CINDEX(3) ];

    e[ CINDEX(1) ] = x10;
    e[ CINDEX(2) ] = x20;
    e[ CINDEX(3) ] = x30;


}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////
//////////////////////////////////////////////////////////////////////


int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;
    MSdata msdata;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name        = "Singular control 5";
    problem.outfilename                = "sing5.txt";

//////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases   = 1;
    problem.nlinkages                 = 0;

    psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 3;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 3;
    problem.phases(1).npath                  = 0;
```

```
    problem.phases(1).nodes                = "[80]";

    psopt_level2_setup(problem, algorithm);


////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ////////////////////
////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states(1) = -inf;
    problem.phases(1).bounds.lower.states(2) = -inf;
    problem.phases(1).bounds.lower.states(3) = -inf;


    problem.phases(1).bounds.upper.states(1) = inf;
    problem.phases(1).bounds.upper.states(2) = inf;
    problem.phases(1).bounds.upper.states(3) = inf;

    problem.phases(1).bounds.lower.controls(1) =  0.0;
    problem.phases(1).bounds.upper.controls(1) =  10.0;

    problem.phases(1).bounds.lower.events(1) = 0.0;
    problem.phases(1).bounds.lower.events(2) = -1.0;
    problem.phases(1).bounds.lower.events(3) = -sqrt(5.0);

    problem.phases(1).bounds.upper.events(1) = 0.0;
    problem.phases(1).bounds.upper.events(2) = -1.0;
    problem.phases(1).bounds.upper.events(3) = -sqrt(5.0);

    problem.bounds.lower.times = "[0.0,  1.0]";
    problem.bounds.upper.times = "[0.0,  1.0]";


////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost   = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess /////////////////////
////////////////////////////////////////////////////////////////////////

    DMatrix state_guess(3,20), control_guess(1,20), param_guess, time_guess;

    state_guess(1,colon())= linspace(0.0, 0.0, 20);
    state_guess(2,colon())= linspace(-1.0, -1.0, 20);
    state_guess(3,colon())= linspace(-sqrt(5.0),-sqrt(5.0), 20);

    control_guess = zeros(1,20);

    time_guess = linspace(0.0, 1.0, 20);

    auto_phase_guess(problem, control_guess, state_guess, param_guess, time_guess);

////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ////////////////////////////
////////////////////////////////////////////////////////////////////////

    algorithm.nlp_iter_max             =  1000;
    algorithm.nlp_tolerance            =  1.e-7;
    algorithm.nlp_method               = "IPOPT";
    algorithm.scaling                  = "automatic";
    algorithm.derivatives              = "automatic";
    algorithm.defect_scaling           = "jacobian-based";
//   algorithm.collocation_method      = "trapezoidal";
//   algorithm.mesh_refinement         = "automatic";
//   algorithm.ode_tolerance           = 5.e-3 ;
//   algorithm.mr_max_iterations       = 3;


////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   ////////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);
```

345

```
//////////////////////////////////////////////////////////////////////
//////////// Extract relevant variables from solution structure  //////////
//////////////////////////////////////////////////////////////////////


    DMatrix x, u, t, xi, ui, ti;

    x       = solution.get_states_in_phase(1);
    u       = solution.get_controls_in_phase(1);
    t       = solution.get_time_in_phase(1);



//////////////////////////////////////////////////////////////////////
//////////// Save solution data to files if desired ////////////////////////
//////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

//////////////////////////////////////////////////////////////////////
//////////// Plot some results if desired (requires gnuplot) ///////////////
//////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name,"time (s)", "x");
    plot(t,u,problem.name,"time (s)", "u");
    plot(t,x,problem.name,"time (s)", "x","x1 x2 x3",
                        "pdf","sing5_states.pdf");
    plot(t,u,problem.name,"time (s)", "u","u",
      "pdf","sing5_control.pdf");

}

//////////////////////////////////////////////////////////////////////
///////////////////////////    END OF FILE    /////////////////////////////
//////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.94 and 3.95, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================


Problem:  Singular control 5
CPU time (seconds): 9.600000e-01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  1.188646e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  1.188646e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 1.201301e-03
NLP solver reports:  The problem solved!
```
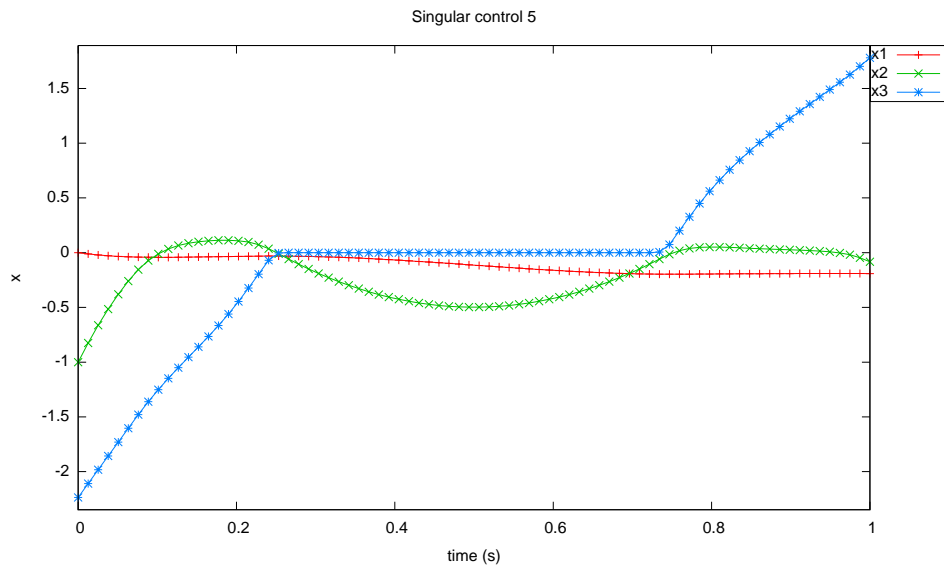
Figure 3.94: States for singular control problem
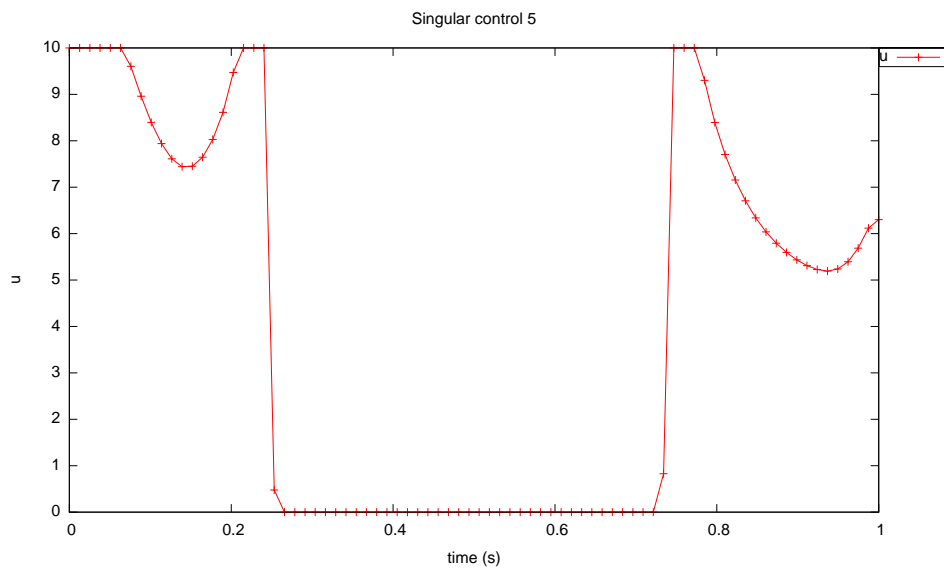
Figure 3.95: Control for singular control problem

347

3.37 Time varying state constraint problem

Consider the following optimal control problem, which involves a time varying state constraint [41]. Find $u(t) \in [0, 1]$ to minimize the cost functional

$$J = \int_0^1 [x_1^2(t) + x_2^2(t) + 0.005u^2(t)]dt \qquad (3.146)$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -x_2 + u \end{aligned} \qquad (3.147)$$

the boundary conditions:

$$\begin{aligned} x_1(0) &= 0 \\ x_2(0) &= -1 \end{aligned} \qquad (3.148)$$

and the path constraint

$$x_2 \leq 8(t - 0.5)^2 - 0.5 \qquad (3.149)$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
//////////////////          statecon1.cxx          ////////////////////
//////////////////////////////////////////////////////////////////////
////////////////            PSOPT  Example          ////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:  Time varying state constraint problem    ///////////////
//////// Last modified:        06 February 2009      ///////////////
//////// Reference:            Teo et. al (1991)      ///////////////
//////// (See PSOPT handbook for full reference)         ///////////////
//////////////////////////////////////////////////////////////////////
////////       Copyright (c) Victor M. Becerra, 2009       ///////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)               ///////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
//////////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    adouble retval = 0.0;

    return  retval;
}

//////////////////////////////////////////////////////////////////////
//////////////////////  Define the running (Lagrange) cost function  ////////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
```

```
    adouble x1 = states[0];
    adouble x2 = states[1];
    adouble u  = controls[0];

    return  ( x1*x1 + x2*x2 + 0.005*u*u );
}

//////////////////////////////////////////////////////////////////////
////////////////////  Define the DAE's /////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble xdot, ydot, vdot;

   adouble x1 = states[0];
   adouble x2 = states[1];
   adouble t  = time;

   adouble u = controls[0];


   derivatives[0] = x2;
   derivatives[1] = -x2 + u;

   path[0]     = -( 8.0*((t-0.5)*(t-0.5)) -0.5 - x2 );

}

//////////////////////////////////////////////////////////////////////
////////////////////  Define the events function ///////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x10 = initial_states[0];
   adouble x20 = initial_states[1];


   e[0] = x10;
   e[1] = x20;
}


//////////////////////////////////////////////////////////////////////
////////////////////  Define the phase linkages function ////////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////
////////////////////  Define the main routine //////////////////////////
//////////////////////////////////////////////////////////////////////


int main(void)
{

//////////////////////////////////////////////////////////////////////
////////////////////  Declare key structures //////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
////////////////////  Register problem name  //////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name      = "Time varying state constraint problem";
    problem.outfilename= "stc1.txt";

//////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
```

```
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages               = 0;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////
/////////   Define phase related information &  do level 2 setup ////////////
//////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates    = 2;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 2;
    problem.phases(1).npath      = 1;
    problem.phases(1).nodes                 = "[20 50]";

    psopt_level2_setup(problem, algorithm);


//////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information ////////////////////
//////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states(1) = -2.0;
    problem.phases(1).bounds.lower.states(2) = -2.0;


    problem.phases(1).bounds.upper.states(1) = 2.0;
    problem.phases(1).bounds.upper.states(2) = 2.0;


    problem.phases(1).bounds.lower.controls(1) = -20.0;
    problem.phases(1).bounds.upper.controls(1) =  20.0;

    problem.phases(1).bounds.lower.events(1) = 0.0;
    problem.phases(1).bounds.lower.events(2) = -1.0;

    problem.phases(1).bounds.upper.events(1) = 0.0;
    problem.phases(1).bounds.upper.events(2) = -1.0;

    problem.phases(1).bounds.upper.path(1) = 0.0;
    problem.phases(1).bounds.lower.path(1) = -100.0;



    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.0;
    problem.phases(1).bounds.upper.EndTime      = 1.0;


//////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  /////////////////////////
//////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

//////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).guess.controls       = zeros(2,20);
    problem.phases(1).guess.states         = zeros(2,20);
    problem.phases(1).guess.time           = linspace(0.0,1.0, 20);

//////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  ////////////////////////////
//////////////////////////////////////////////////////////////////////

    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
    algorithm.nlp_iter_max     = 1000;
    algorithm.nlp_tolerance  = 1.e-4;
```

```
    algorithm.nlp_method ="IPOPT";

//////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   ///////////////////
//////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);


//////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
//////////////////////////////////////////////////////////////////////

    DMatrix x = solution.get_states_in_phase(1);
    DMatrix u = solution.get_controls_in_phase(1);
    DMatrix t = solution.get_time_in_phase(1);

//////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired //////////////////////////
//////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

//////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) //////////////////
//////////////////////////////////////////////////////////////////////
    // Generate points to plot the constraint boundary

    DMatrix x2c =  8.0*((t-0.5)^2.0) -0.5*ones(1,length(t));

    plot(t,x,t,x2c,"states","time (s)", "x", "x1 x2 constraint");
    plot(t,u,"control","time (s)", "u");

    plot(t,x,t,x2c,"states","time (s)", "x", "x1 x2 constraint",
                 "pdf", "stc1_states.pdf");
    plot(t,u,"control","time (s)", "u", "u", "pdf","stc1_control.pdf");

}

//////////////////////////////////////////////////////////////////////
/////////////////////////      END OF FILE     ////////////////////////////////
//////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.96 and 3.97, which contain the elements of the states with the boundary of the constraint on x_2, and the control, respectively.

```
PSOPT results summary
=====================


Problem:  Time varying state constraint problem
CPU time (seconds): 2.580000e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  1.698182e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  1.698182e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 3.125120e-05
NLP solver reports:  The problem solved!
```

Figure 3.96: States for time-varying state constraint problem

3.38 Two burn orbit transfer

The goal of this problem is to compute a trajectory for an spacecraft to go from a standard space shuttle park orbit to a geosynchronous final orbit. It is assumed that the engines operate over two short periods during the mission, and it is desired to compute the timing and duration of the burn periods, as well as the instantaneous direction of the thrust during these two periods, to maximise the final weight of the spacecraft. The problem is described in detail by Betts [3]. The mission then involves four phases: coast, burn, coast and burn. The problem is formulated as follows. Find $\mathbf{u}(t) = [\theta(t), \phi(t)]^T, t \in [t_f^{(1)}, t_f^{(2)}]$ and $t \in [t_f^{(3)}, t_f^{(4)}]$, and the instants $t_f^{(1)}, t_f^{(2)}, t_f^{(3)}, t_f^{(4)}$ such that the following objective function is minimised:

$$J = -w(t_f) \tag{3.150}$$

subject to the dynamic constraints for phases 1 and 3:

$$\dot{\mathbf{y}} = \mathbf{A}(\mathbf{y})\Delta_g + \mathbf{b} \tag{3.151}$$

the following dynamic constraints for phases 2 and 4:

$$\dot{\mathbf{y}} = \mathbf{A}(\mathbf{y})\Delta + \mathbf{b}$$
$$\dot{w} = -T/I_{sp} \tag{3.152}$$
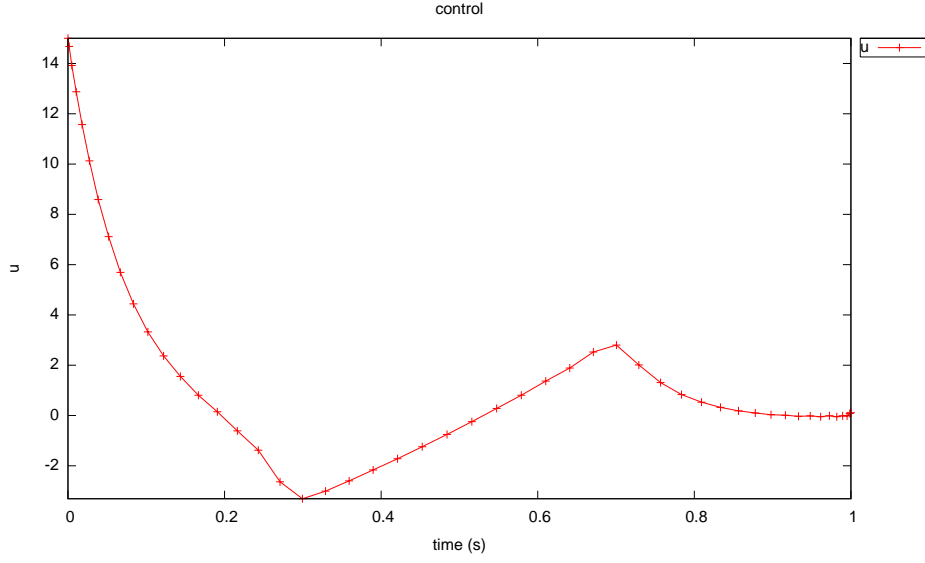
Figure 3.97: Control for time-varying state constraint problem

and the following linkages between phases

$$\mathbf{y}(t_f^{(1)}) = \mathbf{y}(t_0^{(2)})$$
$$\mathbf{y}(t_f^{(2)}) = \mathbf{y}(t_0^{(3)})$$
$$\mathbf{y}(t_f^{(3)}) = \mathbf{y}(t_0^{(4)})$$
$$t_f^{(1)} = t_0^{(2)} \tag{3.153}$$
$$t_f^{(2)} = t_0^{(3)}$$
$$t_f^{(3)} = t_0^{(4)}$$
$$w(t_f^{(2)}) = w(t_0^{(4)})$$

where $\mathbf{y} = [p, f, g, h, k, L, w]^T$ is the vector of modified equinoctial elements, w is the spacecraft weight, I_{sp} is the specific impulse of the engine, T is the maximum thrust, expressions for $\mathbf{A}(\mathbf{y})$ and $\mathbf{b}$ are given in [3]. the disturbing acceleration is $\Delta = \Delta_g + \Delta_T$, where Δ_g is the gravitational disturbing acceleration due to the oblatness of Earth (given in [3]), and Δ_T is the thurst acceleration, given by:

$$\Delta_T = \mathbf{Q}_r \mathbf{Q}_v \begin{bmatrix} T_a \cos\theta \cos\phi \\ T_a \cos\theta \sin\phi \\ T_a \sin\theta \end{bmatrix} \tag{3.154}$$

where $T_a(t) = g_0 T / w(t)$, g_0 is a constant, θ is the pitch angle and ϕ is the yaw angle of the thurst, matrix $\mathbf{Q}_v$ is given by:

$$\mathbf{Q}_v = \left[\frac{\mathbf{v}}{||\mathbf{v}||}, \frac{\mathbf{v} \times r}{||\mathbf{v} \times \mathbf{r}||}, \frac{\mathbf{v}}{||\mathbf{v}||} \times \frac{\mathbf{v} \times r}{||\mathbf{v} \times \mathbf{r}||} \right] \tag{3.155}$$

matrix $\mathbf{Q}_r$ is given by:

$$\mathbf{Q}_r = \begin{bmatrix} \mathbf{i}_r & \mathbf{i}_\theta & \mathbf{i}_h \end{bmatrix} = \left[\frac{\mathbf{r}}{||\mathbf{r}||} \quad \frac{(\mathbf{r}\times\mathbf{v})\times\mathbf{r}}{||\mathbf{r}\times\mathbf{v}||||\mathbf{r}||} \quad \frac{(\mathbf{r}\times\mathbf{v})}{||\mathbf{r}\times\mathbf{v}||} \right] \tag{3.156}$$

The boundary conditions of the problem are given by:

$$
\begin{aligned}
p(0) &= 218327080.052835 \\
f(0) &= 0 \\
g(0) &= 0 \\
h(0) &= 0 \\
h(0) &= 0 \\
k(0) &= 0 \\
L(0) &= \pi \ (\text{rad}) \\
w(0) &= 1 \ (\text{lb}) \\
p(t_f) &= 19323/\sigma + R_e \\
f(t_f) &= 0 \\
g(t_f) &= 0 \\
h(t_f) &= 0 \\
k(t_f) &= 0
\end{aligned}
\tag{3.157}
$$

and the values of the parameters are: $g_0 = 32.174$ (ft/sec^2), $I_{sp} = 300$ (sec), $T = 1.2$ (lb), $\mu = 1.407645794 \times 10^{16}$ (ft^3/sec^2), $R_e = 20925662.73$ (ft), $\sigma = 1.0/6076.1154855643$, $J_2 = 1082.639 \times 10^{-6}$, $J_3 = -2.565 \times 10^{-6}$, $J_4 = -1.608 \times 10^{-6}$.

An initial guess was computed by forward propagation from the initial conditions, assuming the following guesses for the controls and burn periods [3]:

$$
\begin{aligned}
\mathbf{u}(t) &= \begin{bmatrix} 0.148637 \times 10^{-2}, & -9.08446 \end{bmatrix}^T & t \in [2840, 21650] \\
\mathbf{u}(t) &= \begin{bmatrix} -0.136658 \times 10^{-2}, & 49.7892 \end{bmatrix} & t \in [21650, 21700]
\end{aligned}
\tag{3.158}
$$

The problem was solved using local collocation (trapezoidal followed by Hermite-Simpson) with automatic mesh refinement. The $\mathcal{PSOPT}$ code that solves the problem is shown below.

```
//////////////////////////////////////////////////////////////////////
///////////////        twoburn.cxx                ///////////////////
//////////////////////////////////////////////////////////////////////
///////////////              PSOPT Example        ///////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:   two burn orbit transfer problem      //////////////
//////// Last modified: 07 February 2010               //////////////
//////// Reference:     Betts (2001)            ///////////////
//////// (See PSOPT handbook for full reference)       //////////////
//////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2010     //////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////
//////// is distributed under the terms of the GNU Lesser //////////////
//////// General Public License (LGPL)                 //////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"


//////////////////////////////////////////////////////////////////////
//////////////////////  Define auxiliary functions          //////////
//////////////////////////////////////////////////////////////////////

adouble legendre_polynomial( adouble x, int n)
{
// This function computes the value of the legendre polynomials
// for a given value of the argument x and for n=0...5 only

  adouble retval=0.0;

  switch(n) {
     case 0:
         retval=1.0; break;
     case 1:
         retval= x; break;
     case 2:
         retval= 0.5*(3.0*pow(x,2)-1.0); break;
     case 3:
         retval= 0.5*(5.0*pow(x,3)- 3*x); break;
     case 4:
         retval= (1.0/8.0)*(35.0*pow(x,4) - 30.0*pow(x,2) + 3.0); break;
     case 5:
         retval= (1.0/8.0)*(63.0*pow(x,5) - 70.0*pow(x,3) + 15.0*x); break;
     default:
         error_message("legendre_polynomial(x,n) is limited to n=0...5");
  }

  return retval;

}

adouble legendre_polynomial_derivative( adouble x, int n)
{
// This function computes the value of the legendre polynomial derivatives
// for a given value of the argument x and for n=0...5 only.

  adouble retval=0.0;

  switch(n) {
     case 0:
         retval=0.0; break;
     case 1:
         retval= 1.0; break;
     case 2:
         retval= 0.5*(2.0*3.0*x); break;
     case 3:
         retval= 0.5*(3.0*5.0*pow(x,2)-3.0); break;
     case 4:
         retval= (1.0/8.0)*(4.0*35.0*pow(x,3) - 2.0*30.0*x ); break;
     case 5:
         retval= (1.0/8.0)*(5.0*63.0*pow(x,4) - 3.0*70.0*pow(x,2) + 15.0); break;
     default:
         error_message("legendre_polynomial_derivative(x,n) is limited to n=0...5");
  }

  return retval;

}
```

```
void compute_cartesian_trajectory(const DMatrix& x, DMatrix& xyz )
{

   int npoints = x.GetNoCols();
   xyz.Resize(3,npoints);

   for(int i=1; i<=npoints;i++) {


   double p = x(1,i);
   double f = x(2,i);
   double g = x(3,i);
   double h = x(4,i);
   double k = x(5,i);
   double L = x(6,i);

   double q      =  1.0 + f*cos(L) + g*sin(L);
   double r      =  p/q;
   double alpha2 = h*h - k*k;
   double X      = sqrt( h*h + k*k );
   double s2     = 1 + X*X;

   double r1 = r/s2*( cos(L) + alpha2*cos(L) + 2*h*k*sin(L));
   double r2 = r/s2*( sin(L) - alpha2*sin(L) + 2*h*k*cos(L));
   double r3 = 2*r/s2*( h*sin(L) - k*cos(L) );

   xyz(1,i) = r1;
   xyz(2,i) = r2;
   xyz(3,i) = r3;

   }

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{

   if (iphase == 4) {
   adouble w = final_states[CINDEX(7)];
   return (-w);
   }
   else {
   return (0);
   }
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    return  0.0;
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's //////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   // Local integers
   int i, j;
   // Define constants:
   double Isp  = 300.0;             // [sec]
   double mu   = 1.407645794e16;    // [f2^2/sec^2]
   double g0   = 32.174;            // [ft/sec^2]
   double Re   = 20925662.73;       // [ft]
   double T    = 1.2;               // [lb]
   double  CM2W = 32.174;

   double J[5];
```

```
J[2] =  1082.639e-6;
J[3] = -2.565e-6;
J[4] = -1.608e-6;

// Extract individual variables

adouble p = states[ CINDEX(1) ];
adouble f = states[ CINDEX(2) ];
adouble g = states[ CINDEX(3) ];
adouble h = states[ CINDEX(4) ];
adouble k = states[ CINDEX(5) ];
adouble L = states[ CINDEX(6) ];
adouble w;


adouble* u  = controls;

// Define some dependent variables

adouble q      =  1.0 + f*cos(L) + g*sin(L);
adouble r      =  p/q;
adouble alpha2 = h*h - k*k;
adouble X      = sqrt( h*h + k*k );
adouble s2     = 1 + X*X;

// r and v

adouble r1 = r/s2*( cos(L) + alpha2*cos(L) + 2*h*k*sin(L));
adouble r2 = r/s2*( sin(L) - alpha2*sin(L) + 2*h*k*cos(L));
adouble r3 = 2*r/s2*( h*sin(L) - k*cos(L) );

adouble rvec[3];

rvec[ CINDEX(1) ] = r1; rvec[ CINDEX(2)] = r2; rvec[ CINDEX(3) ] = r3;

adouble v1 = -(1.0/s2)*sqrt(mu/p)*(  sin(L) + alpha2*sin(L) - 2*h*k*cos(L) + g - 2*f*h*k + alpha2*g);
adouble v2 = -(1.0/s2)*sqrt(mu/p)*( -cos(L) + alpha2*cos(L) + 2*h*k*sin(L) - f + 2*g*h*k + alpha2*f);
adouble v3 =  (2.0/s2)*sqrt(mu/p)*(h*cos(L) +      k*sin(L) + f*h + g*k);

adouble vvec[3];

vvec[ CINDEX(1) ] = v1; vvec[ CINDEX(2)] = v2; vvec[ CINDEX(3) ] = v3;

// compute Qr

adouble ir[3], ith[3], ih[3];
adouble rv[3];
adouble rvr[3];

cross( rvec, vvec, rv );

cross( rv, rvec, rvr );

adouble norm_r = sqrt( dot(rvec, rvec, 3) );

adouble norm_rv = sqrt( dot(rv, rv, 3) );

for (i=0; i<3; i++) {

   ir[i]  = rvec[i]/norm_r;

   ith[i] = rvr[i]/( norm_rv*norm_r );

   ih[i]  = rv[i]/norm_rv;

}

adouble Qr1[3], Qr2[3], Qr3[3];

for(i=0; i< 3; i++)
{
    // Columns of matrix Qr
    Qr1[i] = ir[i];
    Qr2[i] = ith[i];
    Qr3[i] = ih[i];
}

adouble Qv1[3], Qv2[3], Qv3[3];

adouble norm_v = sqrt( dot(vvec, vvec,3) );

for(i=0;i<3;i++)
{
```

```
    Qv1[i] = vvec[i]/norm_v;
}

adouble vr[3];
cross( vvec, rvec, vr);

adouble norm_vr = sqrt( dot(vr, vr,3) );

for(i=0;i<3;i++)
{
    Qv2[i] = vr[i]/norm_vr;
}

cross( Qv1, Qv2, Qv3 );

// Compute in

adouble en[3];
en[ CINDEX(1) ] = 0.0; en[ CINDEX(2) ]= 0.0; en[ CINDEX(3) ] = 1.0;

adouble enir = dot(en,ir,3);

adouble in[3];

for(i=0;i<3;i++) {
    in[i] = en[i] - enir*ir[i];
}

adouble norm_in = sqrt( dot( in, in, 3 ) );

for(i=0;i<3;i++) {
    in[i] = in[i]/norm_in;
}

// Geocentric latitude angle:

adouble sin_phi =  rvec[ CINDEX(3) ]/ sqrt( dot(rvec,rvec,3) ) ;
adouble cos_phi =  sqrt(1.0- pow(sin_phi,2.0));

adouble deltagn = 0.0;
adouble deltagr = 0.0;
for (j=2; j<=4;j++) {
    adouble Pdash_j = legendre_polynomial_derivative( sin_phi, j );
    adouble P_j     = legendre_polynomial( sin_phi, j );
    deltagn += -mu*cos_phi/(r*r)*pow(Re/r,j)*Pdash_j*J[j];
    deltagr += -mu/(r*r)* (j+1)*pow( Re/r,j)*P_j*J[j];
}

// Compute vector delta_g

adouble delta_g[3];
for (i=0;i<3;i++) {
    delta_g[i] = deltagn*in[i] - deltagr*ir[i];
}

// Compute vector DELTA_g

adouble DELTA_g[3];

DELTA_g[ CINDEX(1) ] = dot(Qr1, delta_g,3);
DELTA_g[ CINDEX(2) ] = dot(Qr2, delta_g,3);
DELTA_g[ CINDEX(3) ] = dot(Qr3, delta_g,3);

// Compute DELTA_T

adouble DELTA_T[3];

if (iphase == 1 || iphase==3) {
    for(i=0;i<3;i++) {
        DELTA_T[i] =  0.0;
    }
}

else {
    adouble Qr[9], Qrt[9], Qv[9], Tvel[3];

    for(i=0;i<3;i++) {
Qr[i]   = Qr1[i];
Qr[3+i] = Qr2[i];
Qr[6+i] = Qr3[i];

Qv[i]   = Qv1[i];
Qv[3+i] = Qv2[i];
```

```
    Qv[6+i] = Qv3[i];
        }

        transpose_ad(Qr, 3, 3, Qrt );

        adouble theta = u[ CINDEX(1) ];
        adouble phi   = u[ CINDEX(2) ];

        adouble Tvec[3];

        w = states[CINDEX(7)];

        adouble mass = w/CM2W;

        adouble Tacc = T/mass;

        Tvec[ CINDEX(1) ] = Tacc*cos(theta)*cos(phi);
        Tvec[ CINDEX(2) ] = Tacc*cos(theta)*sin(phi);
        Tvec[ CINDEX(3) ] = Tacc*sin(theta);

        product_ad( Qv, Tvec, 3, 3, 3, 1, Tvel );

        product_ad(Qrt, Tvel, 3, 3, 3, 1, DELTA_T );


    }

    // Compute DELTA

    adouble DELTA[3];

    for(i=0;i<3;i++) {
        DELTA[i] =  DELTA_g[i] + DELTA_T[i];
    }

    adouble delta1= DELTA[ CINDEX(1) ];
    adouble delta2= DELTA[ CINDEX(2) ];
    adouble delta3= DELTA[ CINDEX(3) ];


    // derivatives

    adouble pdot =                                  2*p/q*sqrt(p/mu)              * delta2;
    adouble fdot =  sqrt(p/mu)*sin(L) * delta1 + sqrt(p/mu)*(1.0/q)*((q+1.0)*cos(L)+f) * delta2
                    -  sqrt(p/mu)*(g/q)*(h*sin(L)-k*cos(L))  * delta3;
    adouble gdot = -sqrt(p/mu)*cos(L) * delta1 + sqrt(p/mu)*(1.0/q)*((q+1.0)*sin(L)+g) * delta2
                    + sqrt(p/mu)*(f/q)*(h*sin(L)-k*cos(L))  * delta3;
    adouble hdot =  sqrt(p/mu)*s2*cos(L)/(2.0*q)            * delta3;
    adouble kdot =  sqrt(p/mu)*s2*sin(L)/(2.0*q)            * delta3;
    adouble Ldot =  sqrt(p/mu)*(1.0/q)*(h*sin(L)-k*cos(L))* delta3      + sqrt(mu*p)*pow( (q/p),2.);

    adouble wdot;

    if (iphase==2 || iphase==4) {
        wdot = -T/Isp;
    }
    else {
        wdot = 0.0;
    }

    derivatives[ CINDEX(1) ] = pdot;
    derivatives[ CINDEX(2) ] = fdot;
    derivatives[ CINDEX(3) ] = gdot;
    derivatives[ CINDEX(4) ] = hdot;
    derivatives[ CINDEX(5) ] = kdot;
    derivatives[ CINDEX(6) ] = Ldot;
    if (iphase==2 || iphase==4) {
     derivatives[ CINDEX(7) ] = wdot;
    }


}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{
```

```
    adouble pti = initial_states[ CINDEX(1) ];
    adouble fti = initial_states[ CINDEX(2) ];
    adouble gti = initial_states[ CINDEX(3) ];
    adouble hti = initial_states[ CINDEX(4) ];
    adouble kti = initial_states[ CINDEX(5) ];
    adouble Lti = initial_states[ CINDEX(6) ];
    adouble wti;
    if (iphase==2) {
        wti = initial_states[ CINDEX(7) ];
    }

    adouble ptf = final_states[ CINDEX(1) ];
    adouble ftf = final_states[ CINDEX(2) ];
    adouble gtf = final_states[ CINDEX(3) ];
    adouble htf = final_states[ CINDEX(4) ];
    adouble ktf = final_states[ CINDEX(5) ];
    adouble Ltf = final_states[ CINDEX(6) ];


    if (iphase==1) {
     e[ CINDEX(1) ]  = pti;
     e[ CINDEX(2) ]  = fti;
     e[ CINDEX(3) ]  = gti;
     e[ CINDEX(4) ]  = hti;
     e[ CINDEX(5) ]  = kti;
     e[ CINDEX(6) ]  = Lti;
    }

    if (iphase==2) {
      e[ CINDEX(1) ] = wti;
    }

    if (iphase == 4) {
     e[ CINDEX(1) ] = ptf;
     e[ CINDEX(2) ] = ftf;
     e[ CINDEX(3) ] = gtf;
     e[ CINDEX(4) ] = htf;
     e[ CINDEX(5) ] = ktf;
    }

}



//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
    // Numeber of linkages:
    // Boundary of phases 1,2: 6 state continuity + 1 time continuity
    // Boundary of phases 2,3: 6 state continuity + 1 time continuity
    // Boundary of phases 3,4: 6 state continuity + 1 time continuity
    // 1 extra linkage w(tf2) = w(ti4)
    // Total: 22 linkage constraints

    adouble xf[7], xi[7], t0a, t0b, tfa, tfb, wtf2, wti4;

    // Linking phases 1 and 2

    get_final_states(        xf, xad, 1, workspace );
    get_initial_states(      xi, xad, 2, workspace );
    tfa = get_final_time(        xad, 1, workspace );
    t0b = get_initial_time(      xad, 2, workspace );


    linkages[ CINDEX(1) ] = xf[ CINDEX(1) ] - xi[ CINDEX(1) ];
    linkages[ CINDEX(2) ] = xf[ CINDEX(2) ] - xi[ CINDEX(2) ];
    linkages[ CINDEX(3) ] = xf[ CINDEX(3) ] - xi[ CINDEX(3) ];
    linkages[ CINDEX(4) ] = xf[ CINDEX(4) ] - xi[ CINDEX(4) ];
    linkages[ CINDEX(5) ] = xf[ CINDEX(5) ] - xi[ CINDEX(5) ];
    linkages[ CINDEX(6) ] = xf[ CINDEX(6) ] - xi[ CINDEX(6) ];
    linkages[ CINDEX(7) ] = t0b - tfa;

    // Linking phases 2 and 3

    get_final_states(        xf, xad, 2, workspace );
    get_initial_states(      xi, xad, 3, workspace );
    tfa = get_final_time(        xad, 2, workspace );
    t0b = get_initial_time(      xad, 3, workspace );
```

```
    wtf2 = xf[ CINDEX(7) ];

    linkages[ CINDEX(8) ]  = xf[ CINDEX(1) ] - xi[ CINDEX(1) ];
    linkages[ CINDEX(9) ]  = xf[ CINDEX(2) ] - xi[ CINDEX(2) ];
    linkages[ CINDEX(10) ] = xf[ CINDEX(3) ] - xi[ CINDEX(3) ];
    linkages[ CINDEX(11) ] = xf[ CINDEX(4) ] - xi[ CINDEX(4) ];
    linkages[ CINDEX(12) ] = xf[ CINDEX(5) ] - xi[ CINDEX(5) ];
    linkages[ CINDEX(13) ] = xf[ CINDEX(6) ] - xi[ CINDEX(6) ];
    linkages[ CINDEX(14) ] = t0b - tfa;

    // Linking phases 3 and 4

    get_final_states(    xf, xad, 3, workspace );
    get_initial_states(  xi, xad, 4, workspace );
    tfa = get_final_time(    xad, 3, workspace );
    t0b = get_initial_time(  xad, 4, workspace );

    wti4 = xi[ CINDEX(7) ];


    linkages[ CINDEX(15) ] = xf[ CINDEX(1) ] - xi[ CINDEX(1) ];
    linkages[ CINDEX(16) ] = xf[ CINDEX(2) ] - xi[ CINDEX(2) ];
    linkages[ CINDEX(17) ] = xf[ CINDEX(3) ] - xi[ CINDEX(3) ];
    linkages[ CINDEX(18) ] = xf[ CINDEX(4) ] - xi[ CINDEX(4) ];
    linkages[ CINDEX(19) ] = xf[ CINDEX(5) ] - xi[ CINDEX(5) ];
    linkages[ CINDEX(20) ] = xf[ CINDEX(6) ] - xi[ CINDEX(6) ];
    linkages[ CINDEX(21) ] = t0b - tfa;

    // Linking the weight at the end of phase 2 with the weight at the beginning of phase 4

    linkages[ CINDEX(22) ] = wtf2 - wti4;

}


////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ////////////////////////////
////////////////////////////////////////////////////////////////////////

int main(void)
{

////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////
////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

    MSdata msdata;

////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////
////////////////////////////////////////////////////////////////////////

    problem.name          = "Two burn transfer problem";
    problem.outfilename             = "twoburn.txt";

////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////

    problem.nphases   = 4;
    problem.nlinkages               = 22;

    psopt_level1_setup(problem);

////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates   = 6;
    problem.phases(1).ncontrols = 0;
    problem.phases(1).nparameters           = 0;
    problem.phases(1).nevents           = 6;
    problem.phases(1).npath     = 0;
    problem.phases(1).nodes                 = 10;

    problem.phases(2).nstates   = 7;
    problem.phases(2).ncontrols = 2;
    problem.phases(2).nparameters           = 0;
```

```
    problem.phases(2).nevents             = 1;
    problem.phases(2).npath      = 0;
    problem.phases(2).nodes                 = 10;

    problem.phases(3).nstates    = 6;
    problem.phases(3).ncontrols  = 0;
    problem.phases(3).nparameters          = 0;
    problem.phases(3).nevents              = 0;
    problem.phases(3).npath      = 0;
    problem.phases(3).nodes                 = 10;

    problem.phases(4).nstates    = 7;
    problem.phases(4).ncontrols  = 2;
    problem.phases(4).nparameters          = 0;
    problem.phases(4).nevents              = 5;
    problem.phases(4).npath      = 0;
    problem.phases(4).nodes                 = 10;

    psopt_level2_setup(problem, algorithm);


////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ////////////////////
////////////////////////////////////////////////////////////////////////


    double pti = 21837080.052835;
    double fti = 0.0;
    double gti = 0.0;
    double hti = -0.25396764647494;
    double kti = 0.0;
    double Lti = pi;
    double wti = 1.0;

    double  SISP = 300.0;
    double  DELTAV2 = 8000;
    double  DELTAV4 = 3000;
    double  CM2W = 32.174;

    double sigma = 1.0/6076.1154855643;

    double Re = 20925662.73;

    double wtf2  = wti*exp(-DELTAV2/(CM2W*SISP));
    double wtf4  = wtf2*exp(-DELTAV4/(CM2W*SISP));

    double ptf = 19323/sigma + Re;
    double ftf = 0.0;
    double gtf = 0.0;
    double htf = 0.0;
    double ktf = 0.0;

    double D2R = pi/180.0;


  // BOUNDS FOR PHASE 1

  problem.phases(1).bounds.lower.states(1) = 10.e6;
  problem.phases(1).bounds.lower.states(2) = -1;
  problem.phases(1).bounds.lower.states(3) = -1;
  problem.phases(1).bounds.lower.states(4) = -1;
  problem.phases(1).bounds.lower.states(5) = -1;
  problem.phases(1).bounds.lower.states(6) = pi;


  problem.phases(1).bounds.upper.states(1) = 2e8;
  problem.phases(1).bounds.upper.states(2) = 1;
  problem.phases(1).bounds.upper.states(3) = 1;
  problem.phases(1).bounds.upper.states(4) = 1;
  problem.phases(1).bounds.upper.states(5) = 1;
  problem.phases(1).bounds.upper.states(6) = 30*pi;


  problem.phases(1).bounds.lower.events(1)  = pti;
  problem.phases(1).bounds.lower.events(2)  = fti;
  problem.phases(1).bounds.lower.events(3)  = gti;
  problem.phases(1).bounds.lower.events(4)  = hti;
  problem.phases(1).bounds.lower.events(5)  = kti;
  problem.phases(1).bounds.lower.events(6)  = Lti;
```

```
problem.phases(1).bounds.upper.events(1)  = pti;
problem.phases(1).bounds.upper.events(2)  = fti;
problem.phases(1).bounds.upper.events(3)  = gti;
problem.phases(1).bounds.upper.events(4)  = hti;
problem.phases(1).bounds.upper.events(5)  = kti;
problem.phases(1).bounds.upper.events(6)  = Lti;


problem.phases(1).bounds.lower.StartTime    = 0.0;
problem.phases(1).bounds.upper.StartTime    = 0.0;

problem.phases(1).bounds.lower.EndTime      = 2000.0;
problem.phases(1).bounds.upper.EndTime      = 3000.0;

// BOUNDS FOR PHASE 2

problem.phases(2).bounds.lower.states(1) = 10.e6;
problem.phases(2).bounds.lower.states(2) = -1;
problem.phases(2).bounds.lower.states(3) = -1;
problem.phases(2).bounds.lower.states(4) = -1;
problem.phases(2).bounds.lower.states(5) = -1;
problem.phases(2).bounds.lower.states(6) = pi;
problem.phases(2).bounds.lower.states(7) = 0.0;

problem.phases(2).bounds.upper.states(1) = 2.e8;
problem.phases(2).bounds.upper.states(2) = 1;
problem.phases(2).bounds.upper.states(3) = 1;
problem.phases(2).bounds.upper.states(4) = 1;
problem.phases(2).bounds.upper.states(5) = 1;
problem.phases(2).bounds.upper.states(6) = 30*pi;
problem.phases(2).bounds.upper.states(7) = 2.0;


problem.phases(2).bounds.lower.controls(1) = -pi;
problem.phases(2).bounds.lower.controls(2) = -pi;
problem.phases(2).bounds.upper.controls(1) = pi;
problem.phases(2).bounds.upper.controls(2) = pi;

problem.phases(2).bounds.lower.events(1)  = wti;
problem.phases(2).bounds.upper.events(1)  = wti;


problem.phases(2).bounds.lower.StartTime    = 2000;
problem.phases(2).bounds.upper.StartTime    = 3000;

problem.phases(2).bounds.lower.EndTime      = 2100;
problem.phases(2).bounds.upper.EndTime      = 3100;


// BOUNDS FOR PHASE 3

problem.phases(3).bounds.lower.states(1) = 10.e6;
problem.phases(3).bounds.lower.states(2) = -1;
problem.phases(3).bounds.lower.states(3) = -1;
problem.phases(3).bounds.lower.states(4) = -1;
problem.phases(3).bounds.lower.states(5) = -1;
problem.phases(3).bounds.lower.states(6) = pi;


problem.phases(3).bounds.upper.states(1) = 2.e8;
problem.phases(3).bounds.upper.states(2) = 1.0;
problem.phases(3).bounds.upper.states(3) = 1.0;
problem.phases(3).bounds.upper.states(4) = 1.0;
problem.phases(3).bounds.upper.states(5) = 1.0;
problem.phases(3).bounds.upper.states(6) = 30*pi;


problem.phases(3).bounds.lower.StartTime    = 2100;
problem.phases(3).bounds.upper.StartTime    = 3100;

problem.phases(3).bounds.lower.EndTime      = 21600;
problem.phases(3).bounds.upper.EndTime      = 21800;


// BOUNDS FOR PHASE 4

problem.phases(4).bounds.lower.states(1) = 10.e6;
problem.phases(4).bounds.lower.states(2) = -1;
problem.phases(4).bounds.lower.states(3) = -1;
problem.phases(4).bounds.lower.states(4) = -1;
```

```
    problem.phases(4).bounds.lower.states(5) = -1;
    problem.phases(4).bounds.lower.states(6) = pi;
    problem.phases(4).bounds.lower.states(7) = 0.0;

    problem.phases(4).bounds.upper.states(1) = 2.e8;
    problem.phases(4).bounds.upper.states(2) = 1;
    problem.phases(4).bounds.upper.states(3) = 1;
    problem.phases(4).bounds.upper.states(4) = 1;
    problem.phases(4).bounds.upper.states(5) = 1;
    problem.phases(4).bounds.upper.states(6) = 30*pi;
    problem.phases(4).bounds.upper.states(7) = 2.0;


    problem.phases(4).bounds.lower.controls(1) = -pi;
    problem.phases(4).bounds.lower.controls(2) = -pi;
    problem.phases(4).bounds.upper.controls(1) = pi;
    problem.phases(4).bounds.upper.controls(2) = pi;


    problem.phases(4).bounds.lower.events(1)  = ptf;
    problem.phases(4).bounds.lower.events(2)  = ftf;
    problem.phases(4).bounds.lower.events(3)  = gtf;
    problem.phases(4).bounds.lower.events(4)  = htf;
    problem.phases(4).bounds.lower.events(5)  = ktf;

    problem.phases(4).bounds.upper.events(1)  = ptf;
    problem.phases(4).bounds.upper.events(2)  = ftf;
    problem.phases(4).bounds.upper.events(3)  = gtf;
    problem.phases(4).bounds.upper.events(4)  = htf;
    problem.phases(4).bounds.upper.events(5)  = ktf;

    problem.phases(4).bounds.lower.StartTime    = 21600;
    problem.phases(4).bounds.upper.StartTime    = 21800;

    problem.phases(4).bounds.lower.EndTime      = 21650;
    problem.phases(4).bounds.upper.EndTime      = 21900;


////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  /////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae            = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess /////////////////////
////////////////////////////////////////////////////////////////////////

    int nnodes;
    int ncontrols;
    int nstates;
    int iphase;
    DMatrix x_guess, u_guess, time_guess, param_guess, xini, xfinal;

    // Phase 1

    nnodes      = problem.phases(1).nodes(1);
    nstates                     = problem.phases(1).nstates;
    iphase = 1;

    x_guess    =  zeros(nstates,nnodes);
    time_guess =  linspace(0.0,2690,nnodes);

    xini.Resize(6,1);
    xini(1)= pti; xini(2)=fti;xini(3)=gti;xini(4)=hti;xini(5)=kti;xini(6)=Lti;


    rk4_propagate( dae, u_guess, time_guess, xini, param_guess, problem, iphase, x_guess, NULL);

    tra(x_guess).Print("x_guess(iphase=1)");

    xfinal = x_guess(colon(),nnodes);

    problem.phases(1).guess.states = x_guess;
    problem.phases(1).guess.time   = time_guess;
```

```
// Phase 2

nnodes     = problem.phases(2).nodes(1);
nstates                  = problem.phases(2).nstates;
ncontrols                = problem.phases(2).ncontrols;
iphase = 2;

u_guess    = zeros(ncontrols,nnodes);
x_guess    = zeros(nstates,nnodes);
time_guess = linspace(2690,2840,nnodes);

xini.Resize(7,1);
xini(1)= xfinal(1); xini(2)=xfinal(2);xini(3)=xfinal(3);xini(4)=xfinal(4);xini(5)=xfinal(5);xini(6)=xfinal(6);
xini(7)= wti;

u_guess(1,colon()) = 0.148637e-2*D2R*ones(1,nnodes);
u_guess(2,colon()) = -9.08446*D2R*ones(1,nnodes);

rk4_propagate( dae, u_guess, time_guess, xini, param_guess, problem, iphase, x_guess, NULL);

tra(x_guess).Print("x_guess(iphase=2)");


xfinal = x_guess(colon(),nnodes);
double wtf2__ = xfinal(7);

problem.phases(2).guess.states   = x_guess;
problem.phases(2).guess.controls = u_guess;
problem.phases(2).guess.time     = time_guess;

// Phase 3

nnodes     = problem.phases(3).nodes(1);
nstates                  = problem.phases(3).nstates;
iphase = 3;


x_guess    = zeros(nstates,nnodes);
time_guess = linspace(2840,21650,nnodes);


xini.Resize(6,1);
xini(1)= xfinal(1); xini(2)=xfinal(2);xini(3)=xfinal(3);xini(4)=xfinal(4);xini(5)=xfinal(5);xini(6)=xfinal(6);

rk4_propagate( dae, u_guess, time_guess, xini, param_guess, problem, iphase, x_guess, NULL);

tra(x_guess).Print("x_guess(iphase=3)");

xfinal = x_guess(colon(),nnodes);

problem.phases(3).guess.states = x_guess;
problem.phases(3).guess.time   = time_guess;

// Phase 4

nnodes     = problem.phases(4).nodes(1);
nstates                  = problem.phases(4).nstates;
ncontrols                = problem.phases(4).ncontrols;
iphase = 4;

u_guess    = zeros(ncontrols,nnodes);
x_guess    = zeros(nstates,nnodes);
time_guess = linspace(21650,21700,nnodes);

u_guess(1,colon()) = -0.136658e-2*D2R*ones(1,nnodes);
u_guess(2,colon()) = 49.7892*D2R*ones(1,nnodes);

xini.Resize(7,1);
xini(1)= xfinal(1); xini(2)=xfinal(2);xini(3)=xfinal(3);xini(4)=xfinal(4);xini(5)=xfinal(5);xini(6)=xfinal(6);
xini(7)= wtf2__;

rk4_propagate( dae, u_guess, time_guess, xini, param_guess, problem, iphase, x_guess, NULL);

tra(x_guess).Print("x_guess(iphase=4)");


problem.phases(4).guess.states   = x_guess;
problem.phases(4).guess.controls = u_guess;
problem.phases(4).guess.time     = time_guess;
```

```
//////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ///////////////////////////
//////////////////////////////////////////////////////////////////////

    algorithm.nlp_iter_max             = 1000;
    algorithm.nlp_tolerance            = 1.e-6;
    algorithm.nlp_method               = "IPOPT";
    algorithm.scaling                  = "automatic";
    algorithm.derivatives              = "automatic";
    algorithm.defect_scaling           = "jacobian-based";
    algorithm.jac_sparsity_ratio       =  0.11;
    algorithm.collocation_method       = "trapezoidal";
//    algorithm.diff_matrix              = "central-differences";
    algorithm.mesh_refinement          = "automatic";
    algorithm.mr_max_iterations        = 5;
    algorithm.ode_tolerance            = 1.0e-6;




//////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem  //////////////////
//////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

//////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
//////////////////////////////////////////////////////////////////////


    DMatrix x, t, w2, xi, w4,  ti;

    x      = solution.get_states_in_phase(1);
    t      = solution.get_time_in_phase(1);

    w2     = solution.get_states_in_phase(2);
    w2     = w2(7,colon());

    w4     = solution.get_states_in_phase(4);
    w4     = w4(7,colon());


    for(int i=2;i<=problem.nphases;i++) {
            xi      = solution.get_states_in_phase(i);
          ti        = solution.get_time_in_phase(i);

xi = xi(colon(1,6),colon());

x = x || xi;

        t = t || ti;

    }

    DMatrix u_phase2      = solution.get_controls_in_phase(2);
    DMatrix u_phase4      = solution.get_controls_in_phase(4);
    DMatrix t2            = solution.get_time_in_phase(2);
    DMatrix t4            = solution.get_time_in_phase(4);



//////////////////////////////////////////////////////////////////////
///////////   Save solution data to files if desired ////////////////////////
//////////////////////////////////////////////////////////////////////

//    x.Save("x.dat");
//    u.Save("u.dat");
//    t.Save("t.dat");

//////////////////////////////////////////////////////////////////////
///////////   Plot some results if desired (requires gnuplot) ///////////////
//////////////////////////////////////////////////////////////////////

    double R2D = 180/pi;

    DMatrix x1 = x(1,colon())/1.e6;
    DMatrix x2 = x(2,colon());
    DMatrix x3 = x(3,colon());
    DMatrix x4 = x(4,colon());
```

```
    DMatrix x5 = x(5,colon());
    DMatrix x6 = x(6,colon());
    DMatrix x7 = x(7,colon());
    DMatrix theta_phase2;
    DMatrix theta_phase4;
    DMatrix phi_phase2;
    DMatrix phi_phase4;
    DMatrix r;

    theta_phase2 = u_phase2(1,colon())*R2D;
    phi_phase2   = u_phase2(2,colon())*R2D;
    theta_phase4 = u_phase4(1,colon())*R2D;
    phi_phase4   = u_phase4(2,colon())*R2D;

    compute_cartesian_trajectory(x,r);

    r.Save("r.dat");

    double ft2km = 0.0003048;

    r = r*ft2km;


    plot(t2,theta_phase2,problem.name+": thrust theta phase 2","time (s)", "theta (deg)", "theta");

    plot(t2,phi_phase2,problem.name+": thrust phi phase 2","time (s)", "phi (deg)", "phi");

    plot(t4,theta_phase4,problem.name+": thrust theta phase 4","time (s)", "theta (deg)", "theta");

    plot(t4,phi_phase4,problem.name+": thrust phi phase 4","time (s)", "phi (deg)", "phi");

    plot(t2,theta_phase2,problem.name+":  thrust pitch angle phase 2","time (s)", "theta (deg)", "theta",
"pdf", "theta2.pdf");

    plot(t2,phi_phase2,problem.name+": thrust angle phase 2","time (s)", "phi (deg)", "phi",
"pdf", "phi2.pdf");

    plot(t4,theta_phase4,problem.name+": thrust pitch angle phase 4","time (s)", "theta (deg)", "theta",
"pdf", "theta4.pdf");

    plot(t4,phi_phase4,problem.name+": thrust yaw angle phase 4","time (s)", "phi (deg)", "phi",
      "pdf", "phi4.pdf");

    plot3(r(1,colon()), r(2,colon()), r(3,colon()), "Two burn trasnfer trajectory", "x (km)", "y (km)", "z (km)",
  NULL, NULL, "30,110");

    plot3(r(1,colon()), r(2,colon()), r(3,colon()), "Two burn transfer trajectory", "x (km)", "y (km)", "z (km)",
  "pdf", "trajectory.pdf", "30,110");

    plot(r(1,colon()), r(2,colon()),  "Two burn trajectory - projection on the equatorial plane",
  "x (km)", "y (km)");



}
/////////////////////////////////////////////////////////////////////////
/////////////////////////     END OF FILE     /////////////////////////////
/////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below. The controls during the burn periods are shown Figures 3.98 to 3.101, which show the control variables during phases 2 and 4, and Figure 3.102, which shows the trajectory in cartesian co-ordinates.

```
PSOPT results summary
=====================


Problem:   Two burn transfer problem
```

Iter	Method	Nodes	NV	NC	OE	CE	JE	HE	RHS	ϵ_{max}	CPU_a
1	TRAPZ	40	308	298	637	636	20	0	48336	4.942e-02	2.400e-01
2	TRAPZ	56	428	402	25	26	23	0	2808	4.129e-03	3.600e-01
3	H-S	76	650	532	38	39	37	0	8580	1.568e-04	1.000e+00
4	H-S	104	888	714	46	47	41	0	14288	2.222e-05	1.590e+00
5	H-S	133	1132	902	66	67	57	0	26197	8.212e-06	2.880e+00
CPU_b	-	-	-	-	-	-	-	-	-	-	3.786e+01
-	-	-	-	-	812	815	178	0	100209	-	4.393e+01

Key: Iter=iteration number, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, ϵ_{max} = maximum relative ODE error, CPU_a = CPU time in seconds spent by NLP algorithm, CPU_b = additional CPU time in seconds spent by PSOPT

Table 3.9: Mesh refinement statistics: Two burn transfer problem

```
CPU time (seconds): 1.260316e+01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  -2.367249e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.609964e+03
Phase 1 maximum relative local error: 1.451279e-06
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  0.000000e+00
Phase 2 initial time: 2.609964e+03
Phase 2 final time: 2.751426e+03
Phase 2 maximum relative local error: 9.391974e-07
Phase 3 endpoint cost function value: 0.000000e+00
Phase 3 integrated part of the cost:  0.000000e+00
Phase 3 initial time: 2.751426e+03
Phase 3 final time: 2.163415e+04
Phase 3 maximum relative local error: 1.158679e-05
Phase 4 endpoint cost function value: -2.367249e-01
Phase 4 integrated part of the cost:  0.000000e+00
Phase 4 initial time: 2.163415e+04
Phase 4 final time: 2.168351e+04
Phase 4 maximum relative local error: 5.589031e-06
NLP solver reports:  The problem solved!
```
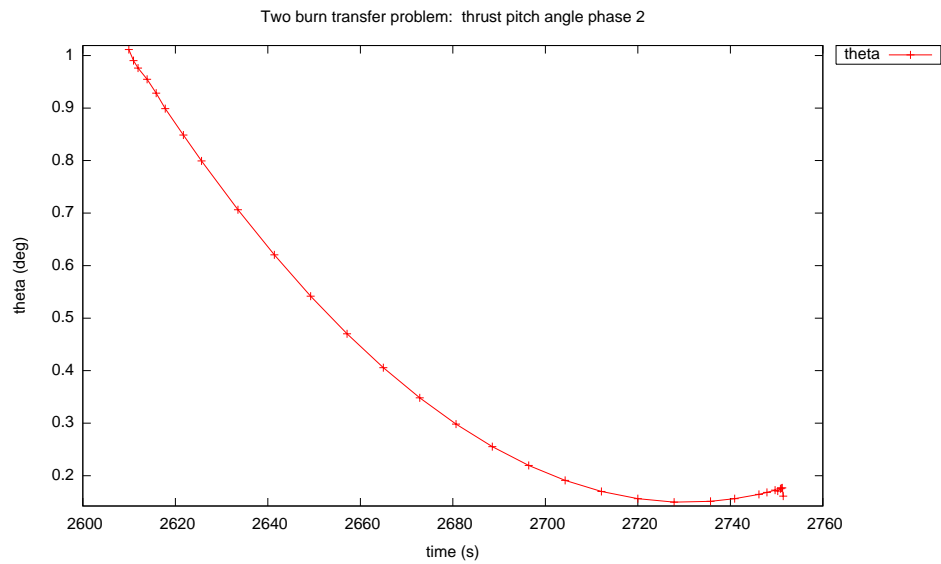
Figure 3.98: Pitch angle during phase 2
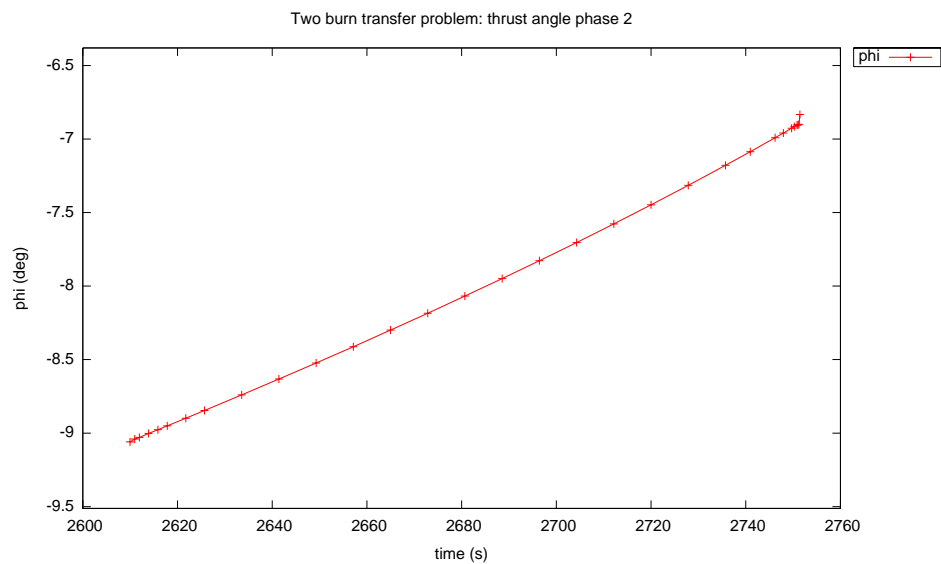
Figure 3.99: Yaw angle during phase 2

369
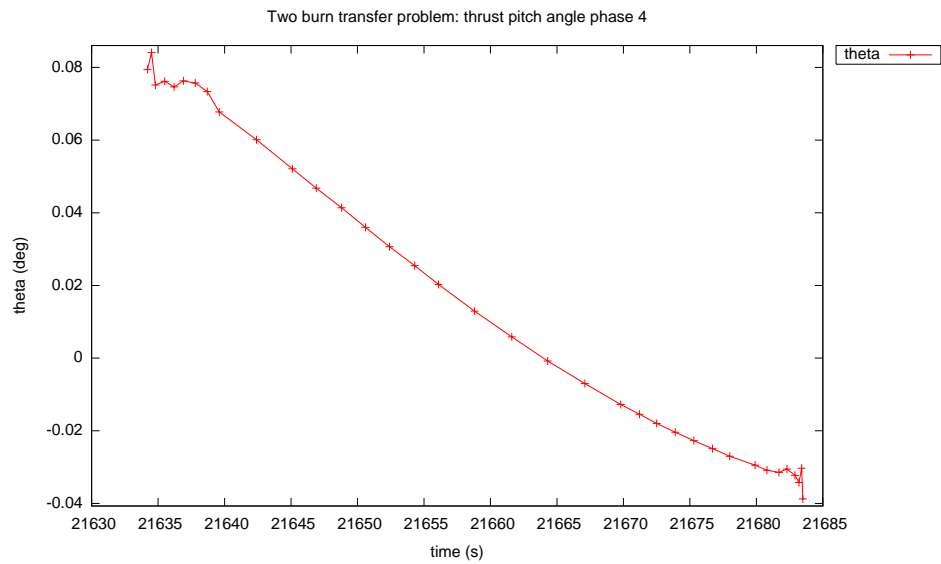
Figure 3.100: Pitch angle during phase 4
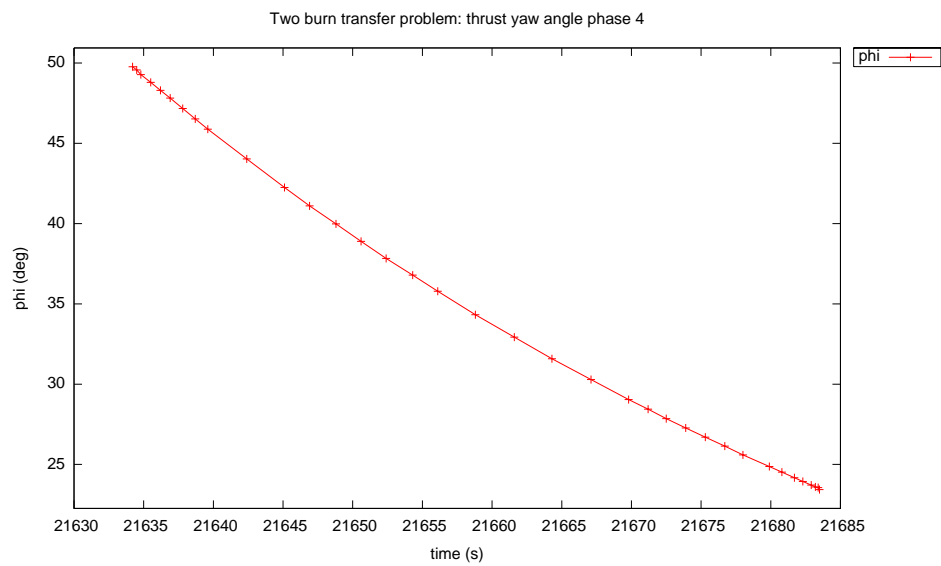
Figure 3.101: Yaw angle during phase 4
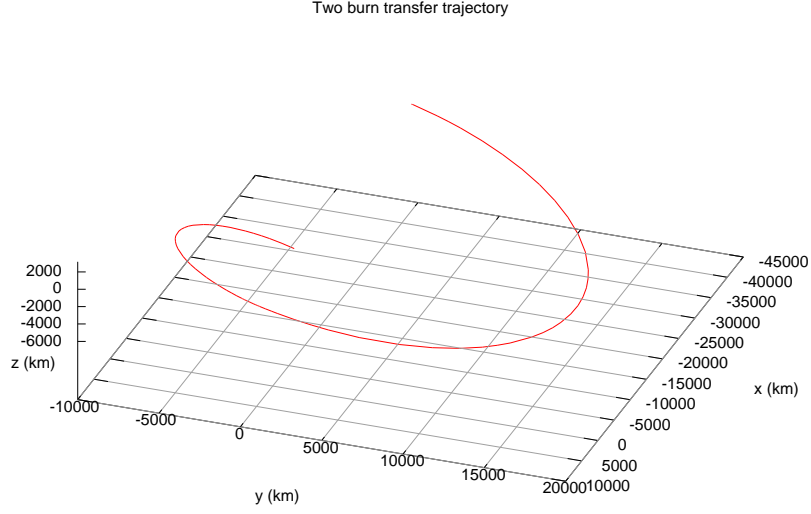
Figure 3.102: Two burn transfer trajectory

3.39 Two link robotic arm

Consider the following optimal control problem [30]. Find t_f, and $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = t_f \tag{3.159}$$

subject to the dynamic constraints

$$\dot{x}_1 = \frac{\sin(x_3)(\frac{9.0}{4.0}\cos(x_3)x_1^2 + 2*x_2^2) + \frac{4.0}{3.0}(u_1 - u_2) - \frac{3.0}{2.0}\cos(x_3)u_2}{\frac{31.0}{36.0} + \frac{9.0}{4.0\sin^2(x_3)}}$$

$$\dot{x}_2 = \frac{-(\sin(x_3)*(\frac{7.0}{2.0}*x_1^2 + \frac{9.0}{4.0}\cos(x_3)x_2^2) - \frac{7.0}{3.0}u_2 + \frac{3.0}{2.0}\cos(x_3)(u_1 - u_2))}{\frac{31.0}{36.0} + \frac{9.0}{4.0\sin^2(x_3)}} \tag{3.160}$$

$$\dot{x}_3 = x_2 - x_1$$
$$\dot{x}_4 = x_1$$

the boundary conditions:

$$\begin{array}{rclrcl}
x_1(0) & = & 0 & x_1(t_f) & = & 0 \\
x_2(0) & = & 0 & x_2(t_f) & = & 0 \\
x_3(0) & = & 0.5 & x_3(t_f) & = & 0.5 \\
x_4(0) & = & 0.0 & x_4(t_f) & = & 0.522
\end{array} \tag{3.161}$$

The control bounds:

$$-1 \leq u_1(t) \leq 1$$
$$-1 \leq u_2(t) \leq 1 \tag{3.162}$$

371

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
////////////////////////////////////////////////////////////////////
//////////////////        twolinkarm.cxx         /////////////////////
////////////////////////////////////////////////////////////////////
////////////////////           PSOPT  Example         ////////////////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
//////// Title:                Two link arm problem     ///////////////
//////// Last modified:        04 January 2009          ///////////////
//////// Reference:            PROPT users guide         ///////////////
//////// (See PSOPT handbook for full reference)        ///////////////
////////////////////////////////////////////////////////////////////
////////         Copyright (c) Victor M. Becerra, 2009       ///////////////
////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                  ///////////////
////////////////////////////////////////////////////////////////////

#include "psopt.h"

////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return tf;
}


////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    return  0.0;
}

////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's /////////////////////////////////
////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble xdot, ydot, vdot;

   adouble x1 = states[ CINDEX(1) ];
   adouble x2 = states[ CINDEX(2) ];
   adouble x3 = states[ CINDEX(3) ];
   adouble x4 = states[ CINDEX(4) ];

   adouble u1 = controls[ CINDEX(1) ];
   adouble u2 = controls[ CINDEX(2) ];

   adouble num1 =  sin(x3)*( (9.0/4.0)*cos(x3)*x1*x1+2*x2*x2 )
                   + (4.0/3.0)*(u1-u2)-(3.0/2.0)*cos(x3)*u2;

   adouble num2 =  -(sin(x3)*((7.0/2.0)*x1*x1+(9.0/4.0)*cos(x3)*x2*x2)
                   -(7.0/3.0)*u2+(3.0/2.0)*cos(x3)*(u1-u2));

   adouble den  =  31.0/36.0 + 9.0/4.0*pow(sin(x3),2);

   derivatives[ CINDEX(1) ] = num1/den;
   derivatives[ CINDEX(2) ] = num2/den;
   derivatives[ CINDEX(3) ] = x2 - x1;
   derivatives[ CINDEX(4) ] = x1;


}

////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////
////////////////////////////////////////////////////////////////////
```

```
void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x10 = initial_states[ CINDEX(1) ];
   adouble x20 = initial_states[ CINDEX(2) ];
   adouble x30 = initial_states[ CINDEX(3) ];
   adouble x40 = initial_states[ CINDEX(4) ];
   adouble x1f = final_states[ CINDEX(1) ];
   adouble x2f = final_states[ CINDEX(2) ];
   adouble x3f = final_states[ CINDEX(3) ];
   adouble x4f = final_states[ CINDEX(4) ];

   e[ CINDEX(1) ] = x10;
   e[ CINDEX(2) ] = x20;
   e[ CINDEX(3) ] = x30;
   e[ CINDEX(4) ] = x40;
   e[ CINDEX(5) ] = x1f;
   e[ CINDEX(6) ] = x2f;
   e[ CINDEX(7) ] = x3f;
   e[ CINDEX(8) ] = x4f;

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function //////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine //////////////////////////
//////////////////////////////////////////////////////////////////////


int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////
//////////////////////////////////////////////////////////////////////

    problem.name         = "Two link robotic arm";

    problem.outfilename           = "twolink.txt";

//////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                = 0;

    psopt_level1_setup(problem);


//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates   = 4;
    problem.phases(1).ncontrols = 2;
    problem.phases(1).nevents   = 8;
    problem.phases(1).npath     = 0;
    problem.phases(1).nodes                = 40;

    psopt_level2_setup(problem, algorithm);


//////////////////////////////////////////////////////////////////////
```

```
//////////////////  Declare DMatrix objects to store results /////////////////
/////////////////////////////////////////////////////////////////////

    DMatrix x, u, t;
    DMatrix lambda, H;

/////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ////////////////////
/////////////////////////////////////////////////////////////////////

    problem.phases(1).bounds.lower.states(1) = -2.0;
    problem.phases(1).bounds.lower.states(2) = -2.0;
    problem.phases(1).bounds.lower.states(3) = -2.0;
    problem.phases(1).bounds.lower.states(4) = -2.0;

    problem.phases(1).bounds.upper.states(1) = 2.0;
    problem.phases(1).bounds.upper.states(2) = 2.0;
    problem.phases(1).bounds.upper.states(3) = 2.0;
    problem.phases(1).bounds.upper.states(4) = 2.0;

    problem.phases(1).bounds.lower.controls(1) = -1.0;
    problem.phases(1).bounds.lower.controls(2) = -1.0;

    problem.phases(1).bounds.upper.controls(1) = 1.0;
    problem.phases(1).bounds.upper.controls(2) = 1.0;

    problem.phases(1).bounds.lower.events(1) = 0.0;
    problem.phases(1).bounds.lower.events(2) = 0.0;
    problem.phases(1).bounds.lower.events(3) = 0.5;
    problem.phases(1).bounds.lower.events(4) = 0.0;
    problem.phases(1).bounds.lower.events(5) = 0.0;
    problem.phases(1).bounds.lower.events(6) = 0.0;
    problem.phases(1).bounds.lower.events(7) = 0.5;
    problem.phases(1).bounds.lower.events(8) = 0.522;

    problem.phases(1).bounds.upper.events = problem.phases(1).bounds.lower.events;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.0;
    problem.phases(1).bounds.upper.EndTime      = 10.0;


/////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  ////////////////////////
/////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;



/////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ////////////////////
/////////////////////////////////////////////////////////////////////


    DMatrix x0(4,40);

    x0(1,colon()) = linspace(0.0,0.0, 40);
    x0(2,colon()) = linspace(0.0,0.0, 40);
    x0(3,colon()) = linspace(0.5,0.5, 40);
    x0(4,colon()) = linspace(0.522,0.522, 40);

    problem.phases(1).guess.controls       = zeros(1,40);
    problem.phases(1).guess.states         = x0;
    problem.phases(1).guess.time           = linspace(0.0, 3.0, 40);

/////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  /////////////////////////
/////////////////////////////////////////////////////////////////////


    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
```

```
    algorithm.nlp_iter_max              = 1000;
    algorithm.nlp_tolerance             = 1.e-6;


////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   ////////////////////
////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////

    x  = solution.get_states_in_phase(1);
    u  = solution.get_controls_in_phase(1);
    t  = solution.get_time_in_phase(1);

////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired /////////////////////////
////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");


////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ////////////////
////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name + ": states", "time (s)", "states", "x1 x2 x3 x4");

    plot(t,u,problem.name + ": controls", "time (s)", "controls", "u1 u2");


    plot(t,x,problem.name + ": states", "time (s)", "states", "x1 x2 x3 x4",
                          "pdf", "twolinkarm_states.pdf");

    plot(t,u,problem.name + ": controls", "time (s)", "controls", "u1 u2",
                          "pdf", "twolinkarm_controls.pdf");

}

////////////////////////////////////////////////////////////////////
////////////////////////       END OF FILE      ////////////////////////////
////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown
in Figures 3.103 and 3.104, which contain the elements of the state and the
control, respectively.

```
PSOPT results summary
=====================


Problem:  Two link robotic arm
CPU time (seconds): 1.080000e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  2.988662e+00
Phase 1 endpoint cost function value: 2.988662e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.988662e+00
```

Figure 3.103: States for two-link robotic arm problem

```
Phase 1 maximum relative local error: 3.815633e-04
NLP solver reports:  The problem solved!
```

3.40   Two-phase path tracking robot

Consider the following two-phase optimal control problem, which consists of a robot following a specified path [39, 37]. Find $u(t) \in [0, 2]$ to minimize the cost functional

$$
J = \int_0^2 [100(x_1 - x_{1,ref})^2 + 100(x_2 - x_{2,ref})^2 \\
+ 500(x_3 - x_{3,ref})^2 + 500(x_4 - x_{4,ref})^2] dt
\tag{3.163}
$$

subject to the dynamic constraints

$$
\begin{aligned}
\dot{x}_1 &= x_3 \\
\dot{x}_2 &= x_4 \\
\dot{x}_3 &= u_1 \\
\dot{x}_4 &= u_2
\end{aligned}
\tag{3.164}
$$
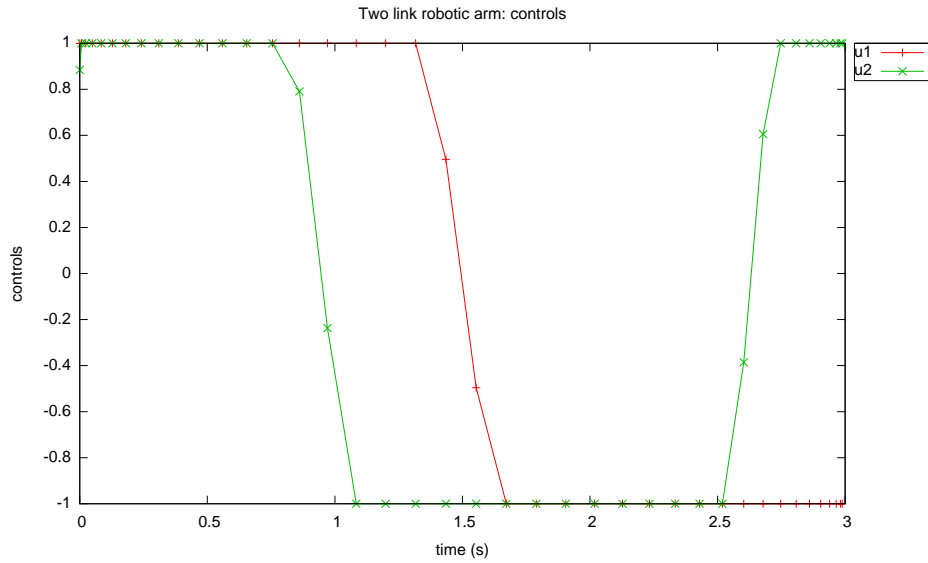
Figure 3.104: Controls for two link robotic arm problem

the boundary conditions:

$$
\begin{array}{llllll}
x_1(0) & = & 0 & x_1(2) & = & 0.5 \\
x_2(0) & = & 0 & x_2(2) & = & 0.5 \\
x_3(0) & = & 0.5 & x_3(2) & = & 0 \\
x_4(0) & = & 0.0 & x_4(2) & = & 0.5
\end{array}
\tag{3.165}
$$

where the reference signals are given by:

$$
\begin{aligned}
x_{1,ref} &= \frac{t}{2}\,(0 \le t < 1),\, \frac{1}{2}\,(1 \le t \le 2) \\
x_{2,ref} &= 0\,(0 \le t < 1),\, \frac{t-1}{2}\,(1 \le t \le 2) \\
x_{3,ref} &= \frac{1}{2}\,(0 \le t < 1),\, 0\,(1 \le t \le 2) \\
x_{4,ref} &= 0\,(0 \le t < 1),\, \frac{1}{2}\,(1 \le t \le 2)
\end{aligned}
\tag{3.166}
$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below. The first phase covers the period $t \in [0,1]$, while the second phase covers the period $t \in [1,2]$.

```
//////////////////////////////////////////////////////////////////////
///////////////////     twophase_robot.cxx      ////////////////////////
//////////////////////////////////////////////////////////////////////
////////////////////     PSOPT  Example        ////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:    Two phase path tracking robot        ////////////////
//////// Last modified:        09 January 2009          ////////////////
```

377

```
//////// Reference:            PROPT Users Guide       ///////////////
//////// (See PSOPT handbook for full reference)        ///////////////
//////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ///////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                    ///////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
//////////////////// Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   return 0.0;
}

//////////////////////////////////////////////////////////////////////
//////////////////// Define the integrand (Lagrange) cost function //////
//////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    adouble retval;

    adouble x1, x2, x3, x4;

    adouble x1ref, x2ref, x3ref, x4ref;

    double w1, w2, w3, w4;

    w1 = 100.0;
    w2 = 100.0;
    w3 = 500.0;
    w4 = 500.0;


   x1 = states[0];
   x2 = states[1];
   x3 = states[2];
   x4 = states[3];

   if (iphase==1) {
    x1ref = time/2;

    x2ref = 0.0;

    x3ref = 0.5;

    x4ref = 0.0;

   }

   if (iphase==2) {

    x1ref =  0.5;

    x2ref =  (time-1.0)/2.0;

    x3ref =  0.0;

    x4ref =  0.5;
   }

   retval = w1*pow(x1-x1ref,2)+w2*pow(x2-x2ref,2)+w3*pow(x3-x3ref,2)+w4*pow(x4-x4ref,2);

   return  retval;
}


//////////////////////////////////////////////////////////////////////
//////////////////// Define the DAE's /////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
```

```
          adouble* controls, adouble* parameters, adouble& time,
          adouble* xad, int iphase, Workspace* workspace)
{


   adouble  x3, x4, u1, u2;

   x3 = states[2];
   x4 = states[3];

   u1 = controls[0];
   u2 = controls[1];

   derivatives[0] = x3;
   derivatives[1] = x4;
   derivatives[2] = u1;
   derivatives[3] = u2;

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function //////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x1i, x2i, x3i, x4i;
   adouble x1f, x2f, x3f, x4f;

   if (iphase == 1)
   {

      x1i = initial_states[0];
      x2i = initial_states[1];
      x3i = initial_states[2];
      x4i = initial_states[3];

      e[0] = x1i;
      e[1] = x2i;
      e[2] = x3i;
      e[3] = x4i;

   }

   else if (iphase == 2)
   {

      x1f = final_states[0];
      x2f = final_states[1];
      x3f = final_states[2];
      x4f = final_states[3];

      e[0] = x1f;
      e[1] = x2f;
      e[2] = x3f;
      e[3] = x4f;

   }


}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function //////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
   int index = 0;

   auto_link( linkages, &index, xad, 1, 2, workspace );

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine //////////////////////////////
//////////////////////////////////////////////////////////////////////
```

379

```
int main(void)
{

////////////////////////////////////////////////////////////////////
////////////////////  Declare key structures ////////////////////////////
////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////
////////////////////  Register problem name  /////////////////////////////
////////////////////////////////////////////////////////////////////

    problem.name          = "Two phase path tracking robot";
    problem.outfilename                = "twophro.txt";

////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////

    problem.nphases    = 2;
    problem.nlinkages               = 5;

    psopt_level1_setup(problem);

////////////////////////////////////////////////////////////////////
/////////   Define phase related information &  do level 2 setup  ///////////
////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 4;
    problem.phases(1).ncontrols  = 2;
    problem.phases(1).nevents    = 4;
    problem.phases(1).npath      = 0;

    problem.phases(2).nstates    = 4;
    problem.phases(2).ncontrols  = 2;
    problem.phases(2).nevents    = 4;
    problem.phases(2).npath      = 0;

    problem.phases(1).nodes         = 30;
    problem.phases(2).nodes         = 30;

    psopt_level2_setup(problem, algorithm);


////////////////////////////////////////////////////////////////////
////////////////////  Enter problem bounds information ////////////////////
////////////////////////////////////////////////////////////////////

    double x1i = 0.0;
    double x2i = 0.0;
    double x3i = 0.5;
    double x4i = 0.0;

    double x1f = 0.5;
    double x2f = 0.5;
    double x3f = 0.0;
    double x4f = 0.5;

    // Phase 0 bounds

    problem.phases(1).bounds.lower.states(1) = -10.0;
    problem.phases(1).bounds.lower.states(2) = -10.0;
    problem.phases(1).bounds.lower.states(3) = -10.0;
    problem.phases(1).bounds.lower.states(4) = -10.0;

    problem.phases(1).bounds.upper.states(1) = 10.0;
    problem.phases(1).bounds.upper.states(2) = 10.0;
    problem.phases(1).bounds.upper.states(3) = 10.0;
    problem.phases(1).bounds.upper.states(4) = 10.0;


    problem.phases(1).bounds.lower.controls(1) = -10.0;
    problem.phases(1).bounds.upper.controls(1) =  10.0;
    problem.phases(1).bounds.lower.controls(2) = -10.0;
    problem.phases(1).bounds.upper.controls(2) =  10.0;

    problem.phases(1).bounds.lower.events(1) = x1i;
    problem.phases(1).bounds.lower.events(2) = x2i;
    problem.phases(1).bounds.lower.events(3) = x3i;
    problem.phases(1).bounds.lower.events(4) = x4i;
```

```
    problem.phases(1).bounds.upper.events(1) = x1i;
    problem.phases(1).bounds.upper.events(2) = x2i;
    problem.phases(1).bounds.upper.events(3) = x3i;
    problem.phases(1).bounds.upper.events(4) = x4i;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.0;
    problem.phases(1).bounds.upper.EndTime      = 1.0;


    // Phase 1 bounds


    problem.phases(2).bounds.lower.states(1) = -10.0;
    problem.phases(2).bounds.lower.states(2) = -10.0;
    problem.phases(2).bounds.lower.states(3) = -10.0;
    problem.phases(2).bounds.lower.states(4) = -10.0;

    problem.phases(2).bounds.upper.states(1) = 10.0;
    problem.phases(2).bounds.upper.states(2) = 10.0;
    problem.phases(2).bounds.upper.states(3) = 10.0;
    problem.phases(2).bounds.upper.states(4) = 10.0;

    problem.phases(2).bounds.lower.controls(1) = -10.0;
    problem.phases(2).bounds.upper.controls(1) =  10.0;
    problem.phases(2).bounds.lower.controls(2) = -10.0;
    problem.phases(2).bounds.upper.controls(2) =  10.0;

    problem.phases(2).bounds.lower.events(1) = x1f;
    problem.phases(2).bounds.lower.events(2) = x2f;
    problem.phases(2).bounds.lower.events(3) = x3f;
    problem.phases(2).bounds.lower.events(4) = x4f;


    problem.phases(2).bounds.upper.events(1) = x1f;
    problem.phases(2).bounds.upper.events(2) = x2f;
    problem.phases(2).bounds.upper.events(3) = x3f;
    problem.phases(2).bounds.upper.events(4) = x4f;

    problem.phases(2).bounds.lower.StartTime    = 1.0;
    problem.phases(2).bounds.upper.StartTime    = 1.0;

    problem.phases(2).bounds.lower.EndTime      = 2.0;
    problem.phases(2).bounds.upper.EndTime      = 2.0;


////////////////////////////////////////////////////////////////////////
////////////////////  Register problem functions  ///////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;


////////////////////////////////////////////////////////////////////////
////////////////////  Define & register initial guess ///////////////////
////////////////////////////////////////////////////////////////////////
    int iphase;

    DMatrix u0(2,30);
    DMatrix x0(4,30);

    DMatrix time_guess0    = linspace(0.0, 1.0 , 30);
    DMatrix time_guess1    = linspace(1.0, 2.0 , 30);

    iphase = 1;

    u0 = zeros(2,30+1);

    x0(1,colon()) = linspace(x1i,(x1i+x1f)/2, 30);
    x0(2,colon()) = linspace(x2i,(x2i+x2f)/2, 30);
    x0(3,colon()) = linspace(x3i,(x3i+x3f)/2, 30);
    x0(4,colon()) = linspace(x4i,(x4i+x4f)/2, 30);
```

```
    problem.phases(iphase).guess.controls = u0;
    problem.phases(iphase).guess.states   = x0;
    problem.phases(iphase).guess.time     = time_guess0;

    iphase = 2;

    u0 = zeros(2,30+1);

    x0(1,colon()) = linspace((x1i+x1f)/2, x1f, 30);
    x0(2,colon()) = linspace((x2i+x2f)/2, x2f, 30);
    x0(3,colon()) = linspace((x3i+x3f)/2, x3f, 30);
    x0(4,colon()) = linspace((x4i+x4f)/2, x4f, 30);

    problem.phases(iphase).guess.controls = u0;
    problem.phases(iphase).guess.states   = x0;
    problem.phases(iphase).guess.time     = time_guess1;

////////////////////////////////////////////////////////////////////////
/////////////////// Enter algorithm options  ////////////////////////////
////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method              = "IPOPT";
    algorithm.scaling                 = "automatic";
    algorithm.derivatives             = "automatic";
    algorithm.hessian                     = "exact";
    algorithm.nlp_iter_max            = 1000;
    algorithm.nlp_tolerance           = 1.e-6;



////////////////////////////////////////////////////////////////////////
////////////////////  Now call PSOPT to solve the problem   //////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   ////////
////////////////////////////////////////////////////////////////////////

    DMatrix xphase1 = solution.get_states_in_phase(1);
    DMatrix uphase1 = solution.get_controls_in_phase(1);
    DMatrix tphase1 = solution.get_time_in_phase(1);

    DMatrix xphase2 = solution.get_states_in_phase(2);
    DMatrix uphase2 = solution.get_controls_in_phase(2);
    DMatrix tphase2 = solution.get_time_in_phase(2);

    DMatrix x = (xphase1 || xphase2);
    DMatrix u = (uphase1 || uphase2);
    DMatrix t = (tphase1 || tphase2);

////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ///////////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) /////////////
////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name, "time (s)", "states");

    plot(t,u,problem.name, "time (s)", "control");

    plot(t,x,problem.name+": states", "time (s)", "states", "x1 x2 x3 x4",
                        "pdf", "twophro_states.pdf");

    plot(t,u,problem.name+": controls", "time (s)", "controls", "u1 u2",
                        "pdf", "twophro_controls.pdf" );

}

////////////////////////////////////////////////////////////////////////
////////////////////////     END OF FILE    //////////////////////////////
////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.105 and 3.106, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Two phase path tracking robot
CPU time (seconds): 9.900000e-01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  1.042533e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.212664e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 3.444867e-04
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  5.212664e-01
Phase 2 initial time: 1.000000e+00
Phase 2 final time: 2.000000e+00
Phase 2 maximum relative local error: 3.444878e-04
NLP solver reports:  The problem solved!
```

3.41   Two-phase Schwartz problem

Consider the following two-phase optimal control problem [37]. Find $u(t) \in [0, 2.9]$ to minimize the cost functional

$$J = 5(x_1(t_f)^2 + x_2(t_f)^2) \tag{3.167}$$

subject to the dynamic constraints

$$\begin{align} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= u - 0.1(1 + 2x_1^2)x_2 \end{align} \tag{3.168}$$

the boundary conditions:

$$\begin{align} x_1(0) &= 1 \\ x_2(0) &= 1 \end{align} \tag{3.169}$$
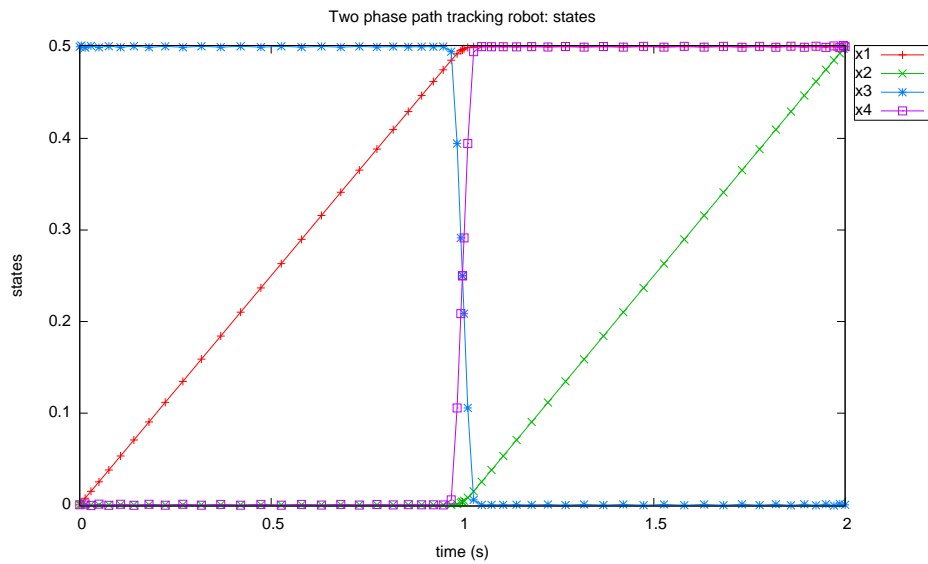
Figure 3.105: States for two-phase path tracking robot problem
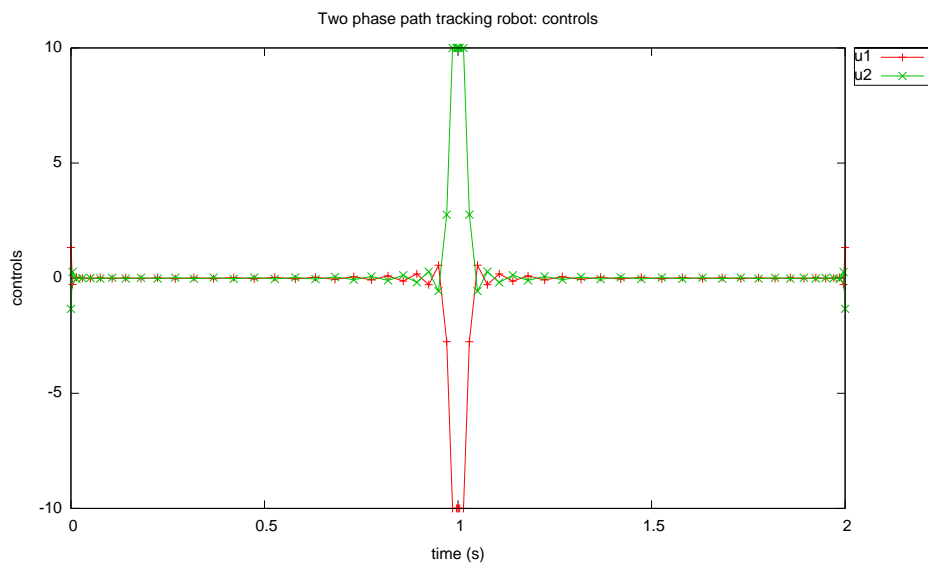
Figure 3.106: Control for two phase path tracking robot problem

and the constraints for $t < 1$:

$$1 - 9(x_1 - 1)^2 - \left(\frac{x_2 - 0.4}{0.3}\right)^2 \leq 0$$
$$-0.8 \leq x_2$$
$$-1 \leq u \leq 1$$

(3.170)

The $\mathcal{PSOPT}$ code that solves this problem is shown below. The problem has been divided into two phases. The first phase covers the period $t \in [0, 1]$, while the second phase covers the period $t \in [1, 2.9]$.

```
//////////////////////////////////////////////////////////////////////
//////////////////        twophase_schwartz.cxx        //////////////////
//////////////////////////////////////////////////////////////////////
///////////////             PSOPT  Example               //////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:       Two phase Schwartz problem         //////////////
//////// Last modified:        09 January 2009           //////////////
//////// Reference:        PROPT Users Guide             //////////////
//////// (See PSOPT handbook for full reference)         //////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009      //////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////
//////// is distributed under the terms of the GNU Lesser //////////////
//////// General Public License (LGPL)                   //////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   adouble retval;
   adouble x1f, x2f;

   // Phase 1 cost


   if ( iphase ==  1)
   {
    retval  = 0.0;

   }

   // Phase 2 cost


   if( iphase == 2 ) {

    x1f = final_states[0];
    x2f = final_states[1];

    retval  = 5*(x1f*x1f + x2f*x2f);
   }

   return retval;

}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////
```

```
adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    return  0.0;
}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

   adouble x1 = states[0];
   adouble x2 = states[1];

   adouble u = controls[0];

   derivatives[0] = x2;
   derivatives[1] = u-0.1*(1+2*x1*x1)*x2;

    if(iphase==1) {

        path[0] =  1.0 - 9.0*((x1-1.0)*(x1-1.0))-((x2-0.4)*(x2-0.4))/(0.3*0.3);

    }

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ///////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x1i = initial_states[0];
   adouble x2i = initial_states[1];


   if (iphase==1) {
    e[0] = x1i;
    e[1] = x2i;
    }


}


//////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
//////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
    int index = 0;
    auto_link( linkages, &index, xad, 1, 2, workspace );

}

//////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ////////////////////////////
//////////////////////////////////////////////////////////////////////


int main(void)
{

//////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////
//////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////
//////////////////////////////////////////////////////////////////////
```

```
    problem.name        = "Two phase Schwartz problem";
    problem.outfilename             = "twophsc.txt";

//////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////////

    problem.nphases    = 2;
    problem.nlinkages               = 3;

    psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////////
/////////   Define phase related information &  do level 2 setup ////////////
//////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates   = 2;
    problem.phases(1).ncontrols = 1;
    problem.phases(1).nevents   = 2;
    problem.phases(1).npath     = 1;

    problem.phases(2).nstates   = 2;
    problem.phases(2).ncontrols = 1;
    problem.phases(2).nevents   = 0;
    problem.phases(2).npath     = 0;

    problem.phases(1).nodes     = 40;
    problem.phases(2).nodes     = 40;

    psopt_level2_setup(problem, algorithm);


//////////////////////////////////////////////////////////////////////////
///////////////////// Enter problem bounds information /////////////////////
//////////////////////////////////////////////////////////////////////////

    double x1L = -20.0;
    double x2L_phase1 = -0.8;
    double x2L_phase2 = -10.0;
    double uL  = -1.0;
    double x1U = 10.0;
    double x2U = 10.0;
    double uU  =  1.0;
    double hL  = -100.0;
    double hU  = 0.0;

    // Phase 0 bounds

    problem.phases(1).bounds.lower.states(1) = x1L;
    problem.phases(1).bounds.lower.states(2) = x2L_phase1;

    problem.phases(1).bounds.upper.states(1) = x1U;
    problem.phases(1).bounds.upper.states(2) = x2U;

    problem.phases(1).bounds.lower.controls(1) = uL;
    problem.phases(1).bounds.upper.controls(1) = uU;

    problem.phases(1).bounds.lower.events(1) = 1.0;
    problem.phases(1).bounds.lower.events(2) = 1.0;


    problem.phases(1).bounds.upper.events(1) = 1.0;
    problem.phases(1).bounds.upper.events(2) = 1.0;

    problem.phases(1).bounds.lower.path(1) = hL;
    problem.phases(1).bounds.upper.path(1) = hU;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.0;
    problem.phases(1).bounds.upper.EndTime      = 1.0;

    // Phase 1 bounds

    problem.phases(2).bounds.lower.states(1) = x1L;
    problem.phases(2).bounds.lower.states(2) = x2L_phase2;

    problem.phases(2).bounds.upper.states(1) = x1U;
    problem.phases(2).bounds.upper.states(2) = x2U;
```

```
    problem.phases(2).bounds.lower.controls(1) = -50.0;
    problem.phases(2).bounds.upper.controls(1) =  50.0;

    problem.phases(2).bounds.lower.StartTime    = 1.0;
    problem.phases(2).bounds.upper.StartTime    = 1.0;

    problem.phases(2).bounds.lower.EndTime      = 2.9;
    problem.phases(2).bounds.upper.EndTime      = 2.9;


////////////////////////////////////////////////////////////////////////
////////////////////  Register problem functions  ///////////////////////
////////////////////////////////////////////////////////////////////////

    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost = &endpoint_cost;
    problem.dae   = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
////////////////////  Define & register initial guess ///////////////////
////////////////////////////////////////////////////////////////////////

    int iphase;


    DMatrix u0(1,40);
    DMatrix x0(2,40);


    DMatrix time_guess0    = linspace(0.0, 1.0 , 40);
    DMatrix time_guess1    = linspace(1.0, 2.9 , 40);


    iphase = 1;


    u0 = zeros(1,40);
    x0(1,colon()) = linspace(1.0,1.0, 40);
    x0(2,colon()) = linspace(1.0,1.0, 40);

    problem.phases(iphase).guess.controls = u0;
    problem.phases(iphase).guess.states   = x0;
    problem.phases(iphase).guess.time     = time_guess0;

    iphase = 2;

    u0 = zeros(1,40);
    x0(1,colon()) = linspace(1.0,1.0, 40);
    x0(2,colon()) = linspace(1.0,1.0, 40);

    problem.phases(iphase).guess.controls = u0;
    problem.phases(iphase).guess.states   = x0;
    problem.phases(iphase).guess.time     = time_guess1;

////////////////////////////////////////////////////////////////////////
////////////////////  Enter algorithm options  //////////////////////////
////////////////////////////////////////////////////////////////////////

    algorithm.nlp_method             = "IPOPT";
    algorithm.scaling                = "automatic";
    algorithm.derivatives            = "automatic";
    algorithm.nlp_iter_max           = 1000;
    algorithm.nlp_tolerance          = 1.e-6;



////////////////////////////////////////////////////////////////////////
////////////////////  Now call PSOPT to solve the problem   //////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
//////////////   Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////

    DMatrix xphase1 = solution.get_states_in_phase(1);
    DMatrix uphase1 = solution.get_controls_in_phase(1);
    DMatrix tphase1 = solution.get_time_in_phase(1);

    DMatrix xphase2 = solution.get_states_in_phase(2);
```

```
    DMatrix uphase2 = solution.get_controls_in_phase(2);
    DMatrix tphase2 = solution.get_time_in_phase(2);

    DMatrix x = (xphase1 || xphase2);
    DMatrix u = (uphase1 || uphase2);
    DMatrix t = (tphase1 || tphase2);

////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired /////////////////////
////////////////////////////////////////////////////////////////////////


    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ////////////
////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name, "time (s)", "states", "x1 x2");

    plot(t,u,problem.name, "time (s)", "control", "u");

    plot(t,x,problem.name, "time (s)", "states", "x1 x2",
                        "pdf", "twophsc_states.pdf");

    plot(t,u,problem.name, "time (s)", "control", "u",
                        "pdf", "twophsc_control.pdf");


}

////////////////////////////////////////////////////////////////////////
/////////////////////////    END OF FILE    /////////////////////////////
////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.107 and 3.108, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================


Problem:  Two phase Schwartz problem
CPU time (seconds): 8.300000e-01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  4.634554e-15
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 5.734313e-06
Phase 2 endpoint cost function value: 4.634554e-15
Phase 2 integrated part of the cost:  0.000000e+00
Phase 2 initial time: 1.000000e+00
Phase 2 final time: 2.900000e+00
Phase 2 maximum relative local error: 9.827652e-05
```

389
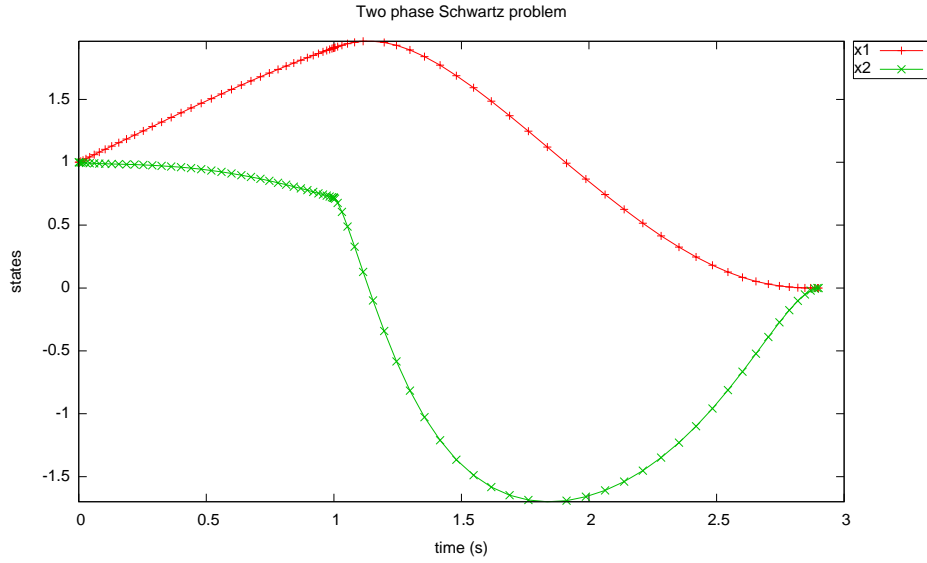
Figure 3.107: States for two-phase Schwartz problem

```
NLP solver reports:  The problem solved!
```

3.42 Vehicle launch problem

This problem consists of the launch of a space vehicle. See [34, 2] for a full description of the problem. Only a brief description is given here. The flight of the vehicle can be divided into four phases, with dry masses ejected from the vehicle at the end of phases 1, 2 and 3. The final times of phases 1, 2 and 3 are fixed, while the final time of phase 4 is free. The optimal control problem is to find the control, $\mathbf{u}$, that minimizes the cost function

$$J = -m^{(4)}(t_f) \tag{3.171}$$

In other words, it is desired to maximise the vehicle mass at the end of phase 4. The dynamics are given by:

$$
\begin{aligned}
\dot{\mathbf{r}} &= \mathbf{v} \\
\dot{\mathbf{v}} &= -\frac{\mu}{\|\mathbf{r}\|^3}\mathbf{r} + \frac{T}{m}\mathbf{u} + \frac{\mathbf{D}}{m} \\
\dot{m} &= -\frac{T}{g_0 I_{sp}}
\end{aligned}
\tag{3.172}
$$
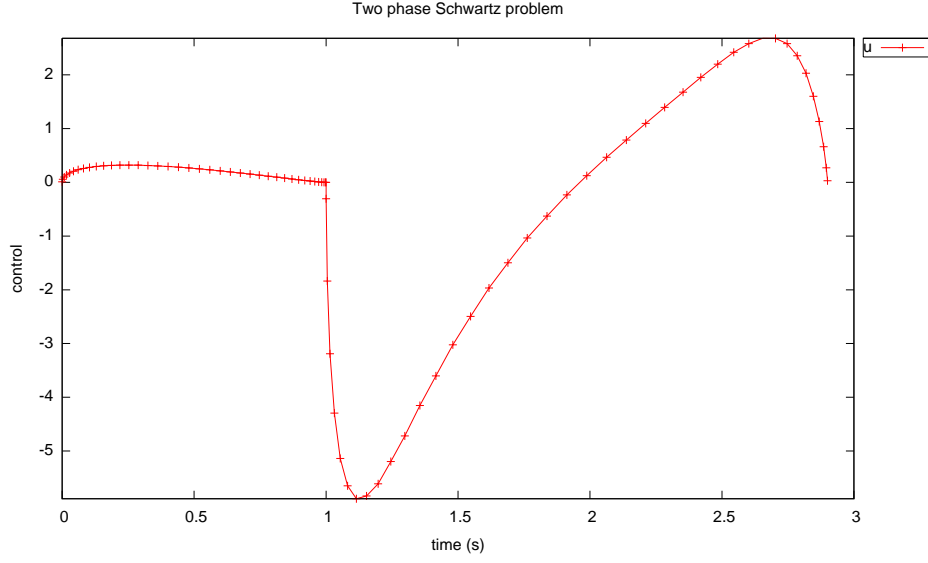
Figure 3.108: Control for two-phase Schwartz problem

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

$$\mathbf{D} = -\frac{1}{2}C_D A_{ref}\rho\|\mathbf{v}_{rel}\|\mathbf{v}_{rel} \tag{3.173}$$

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

$$\mathbf{v}_{rel} = \mathbf{v} - \boldsymbol{\omega} \times \mathbf{r} \tag{3.174}$$

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

$$\rho = \rho_0\exp[-h/h_0] \tag{3.175}$$

where ρ_0 is the atmospheric density at sea level, $h = \|\mathbf{r}\| - R_e$ is the altitude, R_e is the equatorial radius of the Earth, and h_0 is the density scale height. The numerical values for these constants can be found in the code.

The vehicle starts on the ground at rest (relative to the Earth) at time

t_0, so that the initial conditions are

$$\begin{aligned}
\mathbf{r}(t_0) &= \mathbf{r}_0 = \begin{bmatrix} 5605.2 & 0 & 3043.4 \end{bmatrix}^T \quad \text{km} \\
\mathbf{v}(t_0) &= \mathbf{v}_0 = \begin{bmatrix} 0 & 0.4076 & 0 \end{bmatrix}^T \quad \text{km/s} \\
m(t_0) &= m_0 = 301454 \quad \text{kg}
\end{aligned} \tag{3.176}$$

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

$$\begin{aligned}
a_f &= 24361.14 \text{ km}, \\
e_f &= 0.7308, \\
i_f &= 28.5 \text{ deg}, \\
\Omega_f &= 269.8 \text{ deg}, \\
\omega_f &= 130.5 \text{ deg}
\end{aligned} \tag{3.177}$$

There is also a path constraint associated with this problem:

$$||\mathbf{u}||^2 = 1 \tag{3.178}$$

The following linkage constraints force the position and velocity to be continuous and also account for discontinuity in the mass state due to the ejections at the end of phases 1, 2 and 3:

$$\begin{aligned}
\mathbf{r}^{(p)}(t_f) - \mathbf{r}^{(p+1)}(t_0) &= \mathbf{0}, \\
\mathbf{v}^{(p)}(t_f) - \mathbf{v}^{(p+1)}(t_0) &= \mathbf{0}, \qquad (p = 1, \ldots, 3) \\
m^{(p)}(t_f) - m_{dry}^{(p)} - m^{(p+1)}(t_0) &= 0
\end{aligned} \tag{3.179}$$

where the superscript (p) represents the phase number.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
//////////////////          launch.cxx          ////////////////////////
//////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example         ///////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:        Multiphase vehicle launch        ////////////////
//////// Last modified: 05 January 2009                 ////////////////
//////// Reference:     GPOPS Manual                    ////////////////
//////// (See PSOPT handbook for full reference)        ////////////////
//////////////////////////////////////////////////////////////////////
////////       Copyright (c) Victor M. Becerra, 2009        ////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which //////////////
//////// is distributed under the terms of the GNU Lesser //////////////
//////// General Public License (LGPL)                  ////////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"


//////////////////////////////////////////////////////////////////////
//////////////////// Declare auxiliary functions ////////////////////////
//////////////////////////////////////////////////////////////////////

void oe2rv(DMatrix& oe, double mu, DMatrix* ri, DMatrix* vi);

void rv2oe(adouble* rv, adouble* vv, double mu, adouble* oe);
```

```
////////////////////////////////////////////////////////////////////
/////////  Declare an auxiliary structure to hold local constants  //////
////////////////////////////////////////////////////////////////////

struct Constants {
  DMatrix* omega_matrix;
  double mu;
  double cd;
  double sa;
  double rho0;
  double H;
  double Re;
  double g0;
  double thrust_srb;
  double thrust_first;
  double thrust_second;
  double ISP_srb;
  double ISP_first;
  double ISP_second;
};

typedef struct Constants Constants_;

static Constants_ CONSTANTS;




////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    adouble retval;
    adouble mass_tf = final_states[6];

    if (iphase < 4)
      retval = 0.0;

    if (iphase== 4)
      retval = -mass_tf;

    return retval;
}

////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)

{
    return  0.0;
}



////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's //////////////////////////////////
////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

    int j;

    adouble* x = states;
    adouble* u = controls;

    adouble  r[3]; r[0]=x[0]; r[1]=x[1]; r[2]=x[2];

    adouble  v[3]; v[0]=x[3]; v[1]=x[4]; v[2]=x[5];

    adouble  m = x[6];

    double T_first, T_second, T_srb, T_tot, m1dot, m2dot, mdot;
```

393

```
    adouble rad = sqrt( dot( r, r, 3)  );

    DMatrix& omega_matrix = *CONSTANTS.omega_matrix;

    adouble vrel[3];
    for (j=0;j<3;j++)
       vrel[j] =  v[j] - omega_matrix(j+1,1)*r[0] -omega_matrix(j+1,2)*r[1] - omega_matrix(j+1,3)*r[2];


    adouble speedrel = sqrt( dot(vrel,vrel,3) );
    adouble altitude = rad-CONSTANTS.Re;

    adouble rho = CONSTANTS.rho0*exp(-altitude/CONSTANTS.H);
    double   a1 = CONSTANTS.rho0*CONSTANTS.sa*CONSTANTS.cd;
    adouble  a2 = a1*exp(-altitude/CONSTANTS.H);
    adouble  bc = (rho/(2*m))*CONSTANTS.sa*CONSTANTS.cd;



    adouble bcspeed     = bc*speedrel;

    adouble Drag[3];
    for(j=0;j<3;j++) Drag[j] =  - (vrel[j]*bcspeed);



    adouble muoverradcubed = (CONSTANTS.mu)/(pow(rad,3));
    adouble grav[3];
    for(j=0;j<3;j++) grav[j] = -muoverradcubed*r[j];


if (iphase==1) {
  T_srb = 6*CONSTANTS.thrust_srb;
  T_first = CONSTANTS.thrust_first;
  T_tot = T_srb+T_first;
  m1dot = -T_srb/(CONSTANTS.g0*CONSTANTS.ISP_srb);
  m2dot = -T_first/(CONSTANTS.g0*CONSTANTS.ISP_first);
  mdot = m1dot+m2dot;
}
else if (iphase==2) {
  T_srb = 3*CONSTANTS.thrust_srb;
  T_first = CONSTANTS.thrust_first;
  T_tot = T_srb+T_first;
  m1dot = -T_srb/(CONSTANTS.g0*CONSTANTS.ISP_srb);
  m2dot = -T_first/(CONSTANTS.g0*CONSTANTS.ISP_first);
  mdot = m1dot+m2dot;
}
else if (iphase==3) {
  T_first = CONSTANTS.thrust_first;
  T_tot = T_first;
  mdot = -T_first/(CONSTANTS.g0*CONSTANTS.ISP_first);
}
else if (iphase==4) {
  T_second = CONSTANTS.thrust_second;
  T_tot = T_second;
  mdot = -T_second/(CONSTANTS.g0*CONSTANTS.ISP_second);
}


adouble Toverm = T_tot/m;

adouble thrust[3];

for(j=0;j<3;j++) thrust[j] = Toverm*u[j];

adouble rdot[3];
for(j=0;j<3;j++) rdot[j] = v[j];

adouble vdot[3];
for(j=0;j<3;j++)  vdot[j] = thrust[j]+Drag[j]+grav[j];


derivatives[0] = rdot[0];
derivatives[1] = rdot[1];
derivatives[2] = rdot[2];
derivatives[3] = vdot[0];
derivatives[4] = vdot[1];
derivatives[5] = vdot[2];
derivatives[6] = mdot;

path[0] = dot( controls, controls, 3);
```

```
}
////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function //////////////////////////
////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{

    adouble rv[3]; rv[0]=final_states[0]; rv[1]=final_states[1]; rv[2]=final_states[2];
    adouble vv[3]; vv[0]=final_states[3]; vv[1]=final_states[4]; vv[2]=final_states[5];

    adouble oe[6];

    int j;

    if(iphase==1) {
        // These events are related to the initial state conditions in phase 1
        for(j=0;j<7;j++) e[j] = initial_states[j];
    }

    if (iphase==4) {
      // These events are related to the final states in phase 4
      rv2oe( rv, vv, CONSTANTS.mu, oe );
      for(j=0;j<5;j++) e[j]=oe[j];

    }



}


////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function //////////////////
////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{

    double m_tot_first   = 104380.0;
    double m_prop_first  = 95550.0;
    double m_dry_first   = m_tot_first-m_prop_first;
    double m_tot_srb     = 19290.0;
    double m_prop_srb    = 17010.0;
    double m_dry_srb     = m_tot_srb-m_prop_srb;

    int index=0;

    auto_link(linkages, &index, xad, 1, 2, workspace );
    linkages[index-2]-= 6*m_dry_srb;
    auto_link(linkages, &index, xad, 2, 3, workspace );
    linkages[index-2]-= 3*m_dry_srb;
    auto_link(linkages, &index, xad, 3, 4, workspace );
    linkages[index-2]-= m_dry_first;

}



////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine /////////////////////////////
////////////////////////////////////////////////////////////////////////

int main(void)
{
////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures //////////////////////////////
////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  //////////////////////////////
////////////////////////////////////////////////////////////////////////

    problem.name         = "Multiphase vehicle launch";
    problem.outfilename                   = "launch.txt";
```

```
//////////////////////////////////////////////////////////////////////
///////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

    problem.nphases    = 4;
    problem.nlinkages              = 24;

    psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
//////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates   = 7;
    problem.phases(1).ncontrols = 3;
    problem.phases(1).nevents   = 7;
    problem.phases(1).npath     = 1;

    problem.phases(2).nstates   = 7;
    problem.phases(2).ncontrols = 3;
    problem.phases(2).nevents   = 0;
    problem.phases(2).npath     = 1;


    problem.phases(3).nstates   = 7;
    problem.phases(3).ncontrols = 3;
    problem.phases(3).nevents   = 0;
    problem.phases(3).npath     = 1;

    problem.phases(4).nstates   = 7;
    problem.phases(4).ncontrols = 3;
    problem.phases(4).nevents   = 5;
    problem.phases(4).npath     = 1;

    problem.phases(1).nodes     = "[5, 15]";
    problem.phases(2).nodes     = "[5, 15]";
    problem.phases(3).nodes     = "[5, 15]";
    problem.phases(4).nodes     = "[5, 15]";




    psopt_level2_setup(problem, algorithm);

//////////////////////////////////////////////////////////////////////
//////////////////  Declare DMatrix objects to store results //////////////
//////////////////////////////////////////////////////////////////////

    DMatrix x, u, t, H;

//////////////////////////////////////////////////////////////////////
//////////////////  Initialize CONSTANTS and /////////////////////////////
//////////////////  declare local variables  /////////////////////////////
//////////////////////////////////////////////////////////////////////


    double omega            = 7.29211585e-5; // Earth rotation rate (rad/s)
    DMatrix omega_matrix(3,3);

    omega_matrix(1,1) = 0.0;   omega_matrix(1,2) = -omega;   omega_matrix(1,3)=0.0;
    omega_matrix(2,1) = omega; omega_matrix(2,2) = 0.0;      omega_matrix(2,3)=0.0;
    omega_matrix(3,1) = 0.0;   omega_matrix(3,2) = 0.0;      omega_matrix(3,3)=0.0;

    CONSTANTS.omega_matrix = &omega_matrix; // Rotation rate matrix (rad/s)
    CONSTANTS.mu = 3.986012e14;       // Gravitational parameter (m^3/s^2)
    CONSTANTS.cd = 0.5;               // Drag coefficient
    CONSTANTS.sa = 4*pi;              // Surface area (m^2)
    CONSTANTS.rho0 = 1.225;           // sea level gravity (kg/m^3)
    CONSTANTS.H = 7200.0;             // Density scale height (m)
    CONSTANTS.Re = 6378145.0;         // Radius of earth (m)
    CONSTANTS.g0 = 9.80665;           // sea level gravity (m/s^2)

    double lat0 = 28.5*pi/180.0;              // Geocentric Latitude of Cape Canaveral
    double x0 = CONSTANTS.Re*cos(lat0);       // x component of initial position
    double z0 = CONSTANTS.Re*sin(lat0);       // z component of initial position
    double y0 = 0;
    DMatrix r0(3,1,  x0, y0, z0);
    DMatrix v0 = omega_matrix*r0;

    double bt_srb = 75.2;
    double bt_first = 261.0;
    double bt_second = 700.0;
```

```
    double t0 = 0;
    double t1 = 75.2;
    double t2 = 150.4;
    double t3 = 261.0;
    double t4 = 961.0;

    double m_tot_srb      = 19290.0;
    double m_prop_srb     = 17010.0;
    double m_dry_srb      = m_tot_srb-m_prop_srb;
    double m_tot_first    = 104380.0;
    double m_prop_first   = 95550.0;
    double m_dry_first    = m_tot_first-m_prop_first;
    double m_tot_second   = 19300.0;
    double m_prop_second  = 16820.0;
    double m_dry_second   = m_tot_second-m_prop_second;
    double m_payload      = 4164.0;
    double thrust_srb     = 628500.0;
    double thrust_first   = 1083100.0;
    double thrust_second  = 110094.0;
    double mdot_srb       = m_prop_srb/bt_srb;
    double ISP_srb        = thrust_srb/(CONSTANTS.g0*mdot_srb);
    double mdot_first     = m_prop_first/bt_first;
    double ISP_first      = thrust_first/(CONSTANTS.g0*mdot_first);
    double mdot_second    = m_prop_second/bt_second;
    double ISP_second     = thrust_second/(CONSTANTS.g0*mdot_second);

    double af = 24361140.0;
    double ef = 0.7308;
    double incf = 28.5*pi/180.0;
    double Omf = 269.8*pi/180.0;
    double omf = 130.5*pi/180.0;
    double nuguess = 0;
    double cosincf = cos(incf);
    double cosOmf = cos(Omf);
    double cosomf = cos(omf);
    DMatrix oe(6,1, af, ef, incf, Omf, omf, nuguess);

    DMatrix rout(3,1);
    DMatrix vout(3,1);

    oe2rv(oe,CONSTANTS.mu, &rout, &vout);

    rout.Transpose();
    vout.Transpose();

    double m10 = m_payload+m_tot_second+m_tot_first+9*m_tot_srb;
    double m1f = m10-(6*mdot_srb+mdot_first)*t1;
    double m20 = m1f-6*m_dry_srb;
    double m2f = m20-(3*mdot_srb+mdot_first)*(t2-t1);
    double m30 = m2f-3*m_dry_srb;
    double m3f = m30-mdot_first*(t3-t2);
    double m40 = m3f-m_dry_first;
    double m4f = m_payload;

    CONSTANTS.thrust_srb    = thrust_srb;
    CONSTANTS.thrust_first  = thrust_first;
    CONSTANTS.thrust_second = thrust_second;
    CONSTANTS.ISP_srb       = ISP_srb;
    CONSTANTS.ISP_first     = ISP_first;
    CONSTANTS.ISP_second    = ISP_second;


    double rmin = -2*CONSTANTS.Re;
    double rmax = -rmin;
    double vmin = -10000.0;
    double vmax = -vmin;

////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information /////////////////////
////////////////////////////////////////////////////////////////////////


    int iphase;

    problem.bounds.lower.times = "[0, 75.2, 150.4, 261.0, 261.0]";
    problem.bounds.upper.times = "[0, 75.2, 150.4  261.0, 961.0]";

//    double t0 = 0;
//    double t1 = 75.2;
//    double t2 = 150.4;
//    double t3 = 261.0;
//    double t4 = 961.0;
```

```
// Phase 1 bounds

iphase =  1;


problem.phases(iphase).bounds.lower.states(1) = rmin;
problem.phases(iphase).bounds.upper.states(1) = rmax;
problem.phases(iphase).bounds.lower.states(2) = rmin;
problem.phases(iphase).bounds.upper.states(2) = rmax;
problem.phases(iphase).bounds.lower.states(3) = rmin;
problem.phases(iphase).bounds.upper.states(3) = rmax;
problem.phases(iphase).bounds.lower.states(4) = vmin;
problem.phases(iphase).bounds.upper.states(4) = vmax;
problem.phases(iphase).bounds.lower.states(5) = vmin;
problem.phases(iphase).bounds.upper.states(5) = vmax;
problem.phases(iphase).bounds.lower.states(6) = vmin;
problem.phases(iphase).bounds.upper.states(6) = vmax;
problem.phases(iphase).bounds.lower.states(7) = m1f;
problem.phases(iphase).bounds.upper.states(7) = m10;


problem.phases(iphase).bounds.lower.controls(1) = -1.0;
problem.phases(iphase).bounds.upper.controls(1) =  1.0;
problem.phases(iphase).bounds.lower.controls(2) = -1.0;
problem.phases(iphase).bounds.upper.controls(2) =  1.0;
problem.phases(iphase).bounds.lower.controls(3) = -1.0;
problem.phases(iphase).bounds.upper.controls(3) =  1.0;

problem.phases(iphase).bounds.lower.path(1)     = 1.0;
problem.phases(iphase).bounds.upper.path(1)     = 1.0;


// The following bounds fix the initial state conditions in phase 0.

problem.phases(iphase).bounds.lower.events(1) = r0(1);
problem.phases(iphase).bounds.upper.events(1) = r0(1);
problem.phases(iphase).bounds.lower.events(2) = r0(2);
problem.phases(iphase).bounds.upper.events(2) = r0(2);
problem.phases(iphase).bounds.lower.events(3) = r0(3);
problem.phases(iphase).bounds.upper.events(3) = r0(3);
problem.phases(iphase).bounds.lower.events(4) = v0(1);
problem.phases(iphase).bounds.upper.events(4) = v0(1);
problem.phases(iphase).bounds.lower.events(5) = v0(2);
problem.phases(iphase).bounds.upper.events(5) = v0(2);
problem.phases(iphase).bounds.lower.events(6) = v0(3);
problem.phases(iphase).bounds.upper.events(6) = v0(3);
problem.phases(iphase).bounds.lower.events(7) = m10;
problem.phases(iphase).bounds.upper.events(7) = m10;

// Phase 2 bounds

iphase =  2;


problem.phases(iphase).bounds.lower.states(1) = rmin;
problem.phases(iphase).bounds.upper.states(1) = rmax;
problem.phases(iphase).bounds.lower.states(2) = rmin;
problem.phases(iphase).bounds.upper.states(2) = rmax;
problem.phases(iphase).bounds.lower.states(3) = rmin;
problem.phases(iphase).bounds.upper.states(3) = rmax;
problem.phases(iphase).bounds.lower.states(4) = vmin;
problem.phases(iphase).bounds.upper.states(4) = vmax;
problem.phases(iphase).bounds.lower.states(5) = vmin;
problem.phases(iphase).bounds.upper.states(5) = vmax;
problem.phases(iphase).bounds.lower.states(6) = vmin;
problem.phases(iphase).bounds.upper.states(6) = vmax;
problem.phases(iphase).bounds.lower.states(7) = m2f;
problem.phases(iphase).bounds.upper.states(7) = m20;

problem.phases(iphase).bounds.lower.controls(1) = -1.0;
problem.phases(iphase).bounds.upper.controls(1) =  1.0;
problem.phases(iphase).bounds.lower.controls(2) = -1.0;
problem.phases(iphase).bounds.upper.controls(2) =  1.0;
problem.phases(iphase).bounds.lower.controls(3) = -1.0;
problem.phases(iphase).bounds.upper.controls(3) =  1.0;

problem.phases(iphase).bounds.lower.path(1)     = 1.0;
problem.phases(iphase).bounds.upper.path(1)     = 1.0;
```

```
    // Phase 3 bounds

    iphase =  3;



    problem.phases(iphase).bounds.lower.states(1) = rmin;
    problem.phases(iphase).bounds.upper.states(1) = rmax;
    problem.phases(iphase).bounds.lower.states(2) = rmin;
    problem.phases(iphase).bounds.upper.states(2) = rmax;
    problem.phases(iphase).bounds.lower.states(3) = rmin;
    problem.phases(iphase).bounds.upper.states(3) = rmax;
    problem.phases(iphase).bounds.lower.states(4) = vmin;
    problem.phases(iphase).bounds.upper.states(4) = vmax;
    problem.phases(iphase).bounds.lower.states(5) = vmin;
    problem.phases(iphase).bounds.upper.states(5) = vmax;
    problem.phases(iphase).bounds.lower.states(6) = vmin;
    problem.phases(iphase).bounds.upper.states(6) = vmax;
    problem.phases(iphase).bounds.lower.states(7) = m3f;
    problem.phases(iphase).bounds.upper.states(7) = m30;

    problem.phases(iphase).bounds.lower.controls(1) = -1.0;
    problem.phases(iphase).bounds.upper.controls(1) =  1.0;
    problem.phases(iphase).bounds.lower.controls(2) = -1.0;
    problem.phases(iphase).bounds.upper.controls(2) =  1.0;
    problem.phases(iphase).bounds.lower.controls(3) = -1.0;
    problem.phases(iphase).bounds.upper.controls(3) =  1.0;

    problem.phases(iphase).bounds.lower.path(1)     = 1.0;
    problem.phases(iphase).bounds.upper.path(1)     = 1.0;

    // Phase 4 bounds

    iphase =  4;



    problem.phases(iphase).bounds.lower.states(1) = rmin;
    problem.phases(iphase).bounds.upper.states(1) = rmax;
    problem.phases(iphase).bounds.lower.states(2) = rmin;
    problem.phases(iphase).bounds.upper.states(2) = rmax;
    problem.phases(iphase).bounds.lower.states(3) = rmin;
    problem.phases(iphase).bounds.upper.states(3) = rmax;
    problem.phases(iphase).bounds.lower.states(4) = vmin;
    problem.phases(iphase).bounds.upper.states(4) = vmax;
    problem.phases(iphase).bounds.lower.states(5) = vmin;
    problem.phases(iphase).bounds.upper.states(5) = vmax;
    problem.phases(iphase).bounds.lower.states(6) = vmin;
    problem.phases(iphase).bounds.upper.states(6) = vmax;
    problem.phases(iphase).bounds.lower.states(7) = m4f;
    problem.phases(iphase).bounds.upper.states(7) = m40;

    problem.phases(iphase).bounds.lower.controls(1) = -1.0;
    problem.phases(iphase).bounds.upper.controls(1) =  1.0;
    problem.phases(iphase).bounds.lower.controls(2) = -1.0;
    problem.phases(iphase).bounds.upper.controls(2) =  1.0;
    problem.phases(iphase).bounds.lower.controls(3) = -1.0;
    problem.phases(iphase).bounds.upper.controls(3) =  1.0;

    problem.phases(iphase).bounds.lower.path(1)     = 1.0;
    problem.phases(iphase).bounds.upper.path(1)     = 1.0;

    problem.phases(iphase).bounds.lower.events(1)       = af;
    problem.phases(iphase).bounds.lower.events(2)       = ef;
    problem.phases(iphase).bounds.lower.events(3)       = incf;
    problem.phases(iphase).bounds.lower.events(4)       = Omf;
    problem.phases(iphase).bounds.lower.events(5)       = omf;
    problem.phases(iphase).bounds.upper.events(1)       = af;
    problem.phases(iphase).bounds.upper.events(2)       = ef;
    problem.phases(iphase).bounds.upper.events(3)       = incf;
    problem.phases(iphase).bounds.upper.events(4)       = Omf;
    problem.phases(iphase).bounds.upper.events(5)       = omf;

////////////////////////////////////////////////////////////////////
////////////////////  Define & register initial guess ////////////////////////
////////////////////////////////////////////////////////////////////

    iphase = 1;

    problem.phases(iphase).guess.states = zeros(7,5);
```

```matlab
    problem.phases(iphase).guess.states(1, colon()) = linspace( r0(1), r0(1), 5);
    problem.phases(iphase).guess.states(2, colon()) = linspace( r0(2), r0(2), 5);
    problem.phases(iphase).guess.states(3, colon()) = linspace( r0(3), r0(3), 5);
    problem.phases(iphase).guess.states(4, colon()) = linspace( v0(1), v0(1), 5);
    problem.phases(iphase).guess.states(5, colon()) = linspace( v0(2), v0(2), 5);
    problem.phases(iphase).guess.states(6, colon()) = linspace( v0(3), v0(3), 5);
    problem.phases(iphase).guess.states(7, colon()) = linspace( m10  , m1f  , 5);

    problem.phases(iphase).guess.controls = zeros(3,5);

    problem.phases(iphase).guess.controls(1,colon()) = ones( 1, 5);
    problem.phases(iphase).guess.controls(2,colon()) = zeros(1, 5);
    problem.phases(iphase).guess.controls(3,colon()) = zeros(1, 5);


    problem.phases(iphase).guess.time = linspace(t0,t1, 5);


    iphase = 2;

    problem.phases(iphase).guess.states = zeros(7,5);

    problem.phases(iphase).guess.states(1, colon()) = linspace( r0(1), r0(1), 5);
    problem.phases(iphase).guess.states(2, colon()) = linspace( r0(2), r0(2), 5);
    problem.phases(iphase).guess.states(3, colon()) = linspace( r0(3), r0(3), 5);
    problem.phases(iphase).guess.states(4, colon()) = linspace( v0(1), v0(1), 5);
    problem.phases(iphase).guess.states(5, colon()) = linspace( v0(2), v0(2), 5);
    problem.phases(iphase).guess.states(6, colon()) = linspace( v0(3), v0(3), 5);
    problem.phases(iphase).guess.states(7, colon()) = linspace( m20  , m2f  , 5);

    problem.phases(iphase).guess.controls = zeros(3,5);

    problem.phases(iphase).guess.controls(1,colon()) = ones( 1, 5);
    problem.phases(iphase).guess.controls(2,colon()) = zeros(1, 5);
    problem.phases(iphase).guess.controls(3,colon()) = zeros(1, 5);


    problem.phases(iphase).guess.time = linspace(t1,t2, 5);

    iphase = 3;

    problem.phases(iphase).guess.states = zeros(7,5);

    problem.phases(iphase).guess.states(1, colon()) = linspace( rout(1), rout(1), 5);
    problem.phases(iphase).guess.states(2, colon()) = linspace( rout(2), rout(2), 5);
    problem.phases(iphase).guess.states(3, colon()) = linspace( rout(3), rout(3), 5);
    problem.phases(iphase).guess.states(4, colon()) = linspace( vout(1), vout(1), 5);
    problem.phases(iphase).guess.states(5, colon()) = linspace( vout(2), vout(2), 5);
    problem.phases(iphase).guess.states(6, colon()) = linspace( vout(3), vout(3), 5);
    problem.phases(iphase).guess.states(7, colon()) = linspace( m30  , m3f  , 5);

    problem.phases(iphase).guess.controls = zeros(3,5);

    problem.phases(iphase).guess.controls(1,colon()) = zeros( 1, 5);
    problem.phases(iphase).guess.controls(2,colon()) = zeros( 1, 5);
    problem.phases(iphase).guess.controls(3,colon()) = ones(  1, 5);


    problem.phases(iphase).guess.time = linspace(t2,t3, 5);

    iphase = 4;

    problem.phases(iphase).guess.states = zeros(7,5);

    problem.phases(iphase).guess.states(1, colon()) = linspace( rout(1), rout(1), 5);
    problem.phases(iphase).guess.states(2, colon()) = linspace( rout(2), rout(2), 5);
    problem.phases(iphase).guess.states(3, colon()) = linspace( rout(3), rout(3), 5);
    problem.phases(iphase).guess.states(4, colon()) = linspace( vout(1), vout(1), 5);
    problem.phases(iphase).guess.states(5, colon()) = linspace( vout(2), vout(2), 5);
    problem.phases(iphase).guess.states(6, colon()) = linspace( vout(3), vout(3), 5);
    problem.phases(iphase).guess.states(7, colon()) = linspace( m40  , m4f  , 5);

    problem.phases(iphase).guess.controls = zeros(3,5);

    problem.phases(iphase).guess.controls(1,colon()) = zeros( 1, 5);
    problem.phases(iphase).guess.controls(2,colon()) = zeros( 1, 5);
    problem.phases(iphase).guess.controls(3,colon()) = ones(  1, 5);


    problem.phases(iphase).guess.time = linspace(t3,t4, 5);

//////////////////////////////////////////////////////////////////////
```

```
////////////////// Register problem functions /////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
////////////////// Enter algorithm options ///////////////////////////
////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method             = "IPOPT";
    algorithm.scaling                = "automatic";
    algorithm.derivatives            = "automatic";
    algorithm.nlp_iter_max           = 500;
//   algorithm.mesh_refinement                = "automatic";
//   algorithm.collocation_method = "trapezoidal";
    algorithm.ode_tolerance = 1.e-5;

////////////////////////////////////////////////////////////////////////
////////////////// Now call PSOPT to solve the problem ////////////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
//////////// Extract relevant variables from solution structure ///////////
////////////////////////////////////////////////////////////////////////

    x = (solution.get_states_in_phase(1) || solution.get_states_in_phase(2) ||
         solution.get_states_in_phase(3) || solution.get_states_in_phase(4) );
    u = (solution.get_controls_in_phase(1) || solution.get_controls_in_phase(2) ||
         solution.get_controls_in_phase(3) || solution.get_controls_in_phase(4) );
    t = (solution.get_time_in_phase(1) || solution.get_time_in_phase(2) ||
         solution.get_time_in_phase(3) || solution.get_time_in_phase(4) );


////////////////////////////////////////////////////////////////////////
//////////// Save solution data to files if desired ///////////////////////
////////////////////////////////////////////////////////////////////////

    x.Save("x.dat");
    u.Save("u.dat");
    t.Save("t.dat");

////////////////////////////////////////////////////////////////////////
//////////// Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////


    DMatrix r = x(colon(1,3),colon());

    DMatrix v = x(colon(4,6),colon());

    DMatrix altitude = Sqrt(sum(elemProduct(r,r)))/1000.0;

    DMatrix speed = Sqrt(sum(elemProduct(v,v)));

    plot(t,altitude,problem.name, "time (s)", "position (km)");


    plot(t,speed,problem.name, "time (s)", "speed (m/s)");

    plot(t,u,problem.name,"time (s)", "u");


    plot(t,altitude,problem.name, "time (s)", "position (km)", "alt",
                           "pdf", "launch_position.pdf");

    plot(t,speed,problem.name, "time (s)", "speed (m/s)", "speed",
                         "pdf", "launch_speed.pdf");

    plot(t,u,problem.name,"time (s)", "u (dimensionless)", "u1 u2 u3",
                         "pdf", "launch_control.pdf");


}
```

```
////////////////////////////////////////////////////////////////////////
//////////////////// Define auxiliary functions //////////////////////////
////////////////////////////////////////////////////////////////////////


void rv2oe(adouble* rv, adouble* vv, double mu, adouble* oe)
{
        int j;

        adouble K[3]; K[0] = 0.0; K[1]=0.0; K[2]=1.0;

        adouble hv[3];
        cross(rv,vv, hv);

        adouble nv[3];
        cross(K, hv, nv);

        adouble n  = sqrt(  dot(nv,nv,3) );

        adouble h2 = dot(hv,hv,3);

        adouble v2 = dot(vv,vv,3);

        adouble r         = sqrt(dot(rv,rv,3));

        adouble ev[3];
        for(j=0;j<3;j++)  ev[j] = 1/mu *( (v2-mu/r)*rv[j] - dot(rv,vv,3)*vv[j] );

        adouble p         = h2/mu;

adouble e  = sqrt(dot(ev,ev,3)); // eccentricity
adouble a  = p/(1-e*e);    // semimajor axis
adouble i  = acos(hv[2]/sqrt(h2)); // inclination



#define USE_SMOOTH_HEAVISIDE
        double a_eps = 0.1;

#ifndef USE_SMOOTH_HEAVISIDE
  adouble Om = acos(nv[0]/n); // RAAN
  if ( nv[1] < -DMatrix::GetEPS() ){ // fix quadrant
  Om = 2*pi-Om;
  }
#endif

#ifdef USE_SMOOTH_HEAVISIDE

        adouble Om =  smooth_heaviside( (nv[1]+DMatrix::GetEPS()), a_eps )*acos(nv[0]/n)
                     +smooth_heaviside( -(nv[1]+DMatrix::GetEPS()), a_eps )*(2*pi-acos(nv[0]/n));
#endif

#ifndef USE_SMOOTH_HEAVISIDE
  adouble om = acos(dot(nv,ev,3)/n/e); // arg of periapsis
  if ( ev[2] < 0 ) { // fix quadrant
  om = 2*pi-om;
  }
#endif

#ifdef USE_SMOOTH_HEAVISIDE
        adouble om =  smooth_heaviside( (ev[2]), a_eps )*acos(dot(nv,ev,3)/n/e)
                     +smooth_heaviside( -(ev[2]), a_eps )*(2*pi-acos(dot(nv,ev,3)/n/e));
#endif

#ifndef USE_SMOOTH_HEAVISIDE
  adouble nu = acos(dot(ev,rv,3)/e/r); // true anomaly
  if ( dot(rv,vv,3) < 0 ) { // fix quadrant
  nu = 2*pi-nu;
  }
#endif

#ifdef USE_SMOOTH_HEAVISIDE
        adouble nu =  smooth_heaviside( dot(rv,vv,3), a_eps )*acos(dot(ev,rv,3)/e/r)
                     +smooth_heaviside( -dot(rv,vv,3), a_eps )*(2*pi-acos(dot(ev,rv,3)/e/r));
#endif

        oe[0] = a;
        oe[1] = e;
        oe[2] = i;
        oe[3] = Om;
        oe[4] = om;
        oe[5] = nu;
```

```
        return;

}


void oe2rv(DMatrix& oe, double mu, DMatrix* ri, DMatrix* vi)
{
double a=oe(1), e=oe(2), i=oe(3), Om=oe(4), om=oe(5), nu=oe(6);
double p = a*(1-e*e);
double r = p/(1+e*cos(nu));
DMatrix rv(3,1);
        rv(1) = r*cos(nu);
        rv(2) = r*sin(nu);
        rv(3) = 0.0;

DMatrix vv(3,1);

        vv(1) = -sin(nu);
        vv(2) = e+cos(nu);
        vv(3) = 0.0;
        vv    *= sqrt(mu/p);

double c0 = cos(Om),  s0 = sin(Om);
double co = cos(om),  so = sin(om);
double ci = cos(i),   si = sin(i);

DMatrix R(3,3);
        R(1,1)=  c0*co-s0*so*ci; R(1,2)=  -c0*so-s0*co*ci; R(1,3)=  s0*si;
R(2,1)=  s0*co+c0*so*ci; R(2,2)=  -s0*so+c0*co*ci; R(2,3)=-c0*si;
R(3,1)=   so*si;         R(3,2)=   co*si;          R(3,3)=  ci;

*ri = R*rv;
*vi = R*vv;

        return;
}
////////////////////////////////////////////////////////////////////////
//////////////////////////    END OF FILE    ////////////////////////////
////////////////////////////////////////////////////////////////////////

=
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.109, 3.110 and 3.111, which contain the trajectories of the altitude, speed and the elements of the control vector, respectively.

```
PSOPT results summary
=====================


Problem:  Multiphase vehicle launch
CPU time (seconds): 2.089997e+00
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  -7.529661e+03
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 7.520000e+01
Phase 1 maximum relative local error: 1.179806e-06
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  0.000000e+00
Phase 2 initial time: 7.520000e+01
```
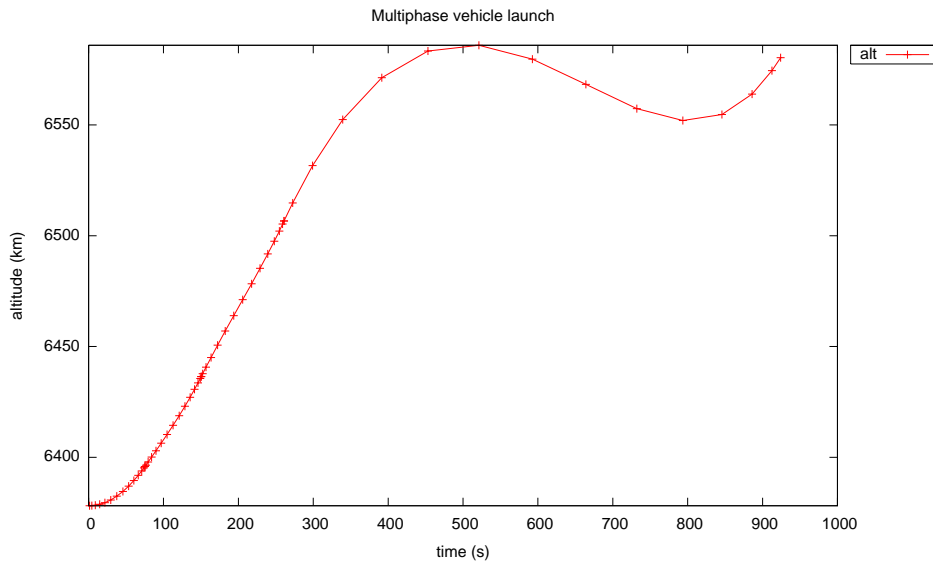
Figure 3.109: Altitude for the vehicle launch problem

```
Phase 2 final time: 1.504000e+02
Phase 2 maximum relative local error: 7.293732e-07
Phase 3 endpoint cost function value: 0.000000e+00
Phase 3 integrated part of the cost:  0.000000e+00
Phase 3 initial time: 1.504000e+02
Phase 3 final time: 2.610000e+02
Phase 3 maximum relative local error: 4.236462e-07
Phase 4 endpoint cost function value: -7.529661e+03
Phase 4 integrated part of the cost:  0.000000e+00
Phase 4 initial time: 2.610000e+02
Phase 4 final time: 9.241413e+02
Phase 4 maximum relative local error: 1.805398e-06
NLP solver reports:  The problem solved!
```

3.43 Zero propellant maneouvre of the International Space Station

This problem illustrates the use of $\mathcal{PSOPT}$ for solving an optimal control problem associated with the design of a zero propellant maneouvre for the international space station by means of control moment gyroscopes (CMGs). The example is based on the results presented in the thesis by Bhatt [5]
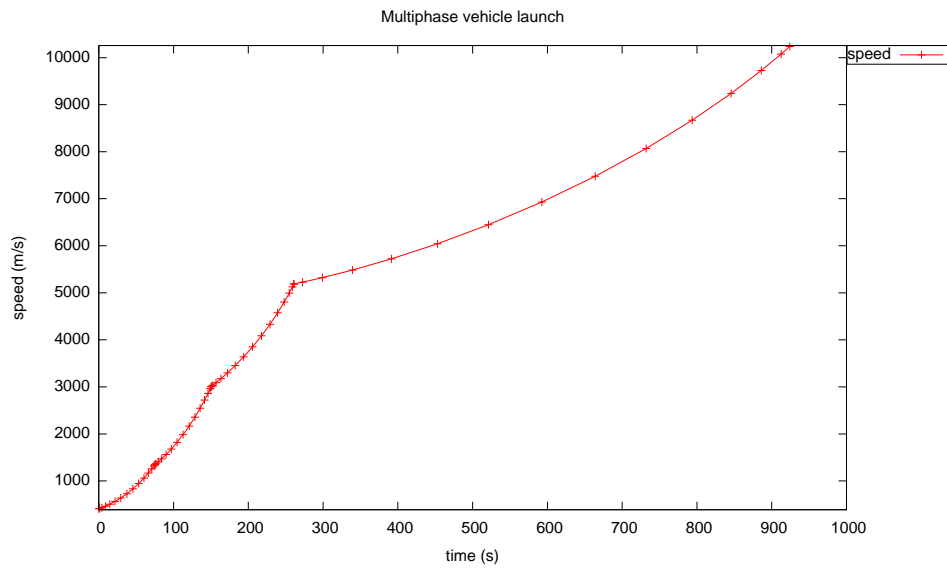
Figure 3.110: Speed for the vehicle launch problem
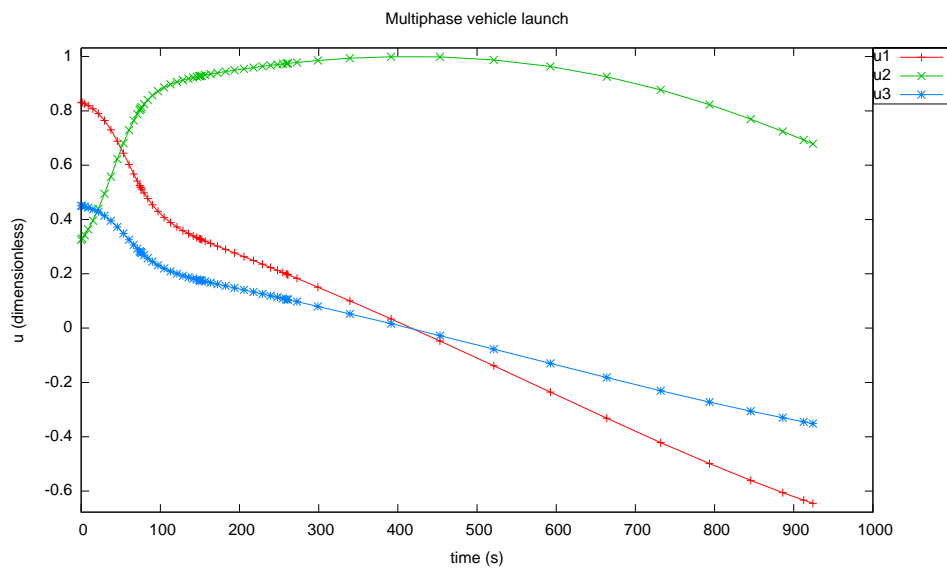
Figure 3.111: Controls for the vehicle launch problem

405

and also reported by Bedrossian and co-workers [1]. The original 90 and 180 degree maneouvres were computed using DIDO, and they were actually implemented on the International Space Station on 5 November 2006 and 2 January 2007, respectively, resulting in savings for NASA of around US\$1.5m in propellant costs. The dynamic model employed here does not account for atmospheric drag as the atmosphere model used in the original study is not available. Otherwise, the equations and parameters are the same as those reported by Bhatt in his thesis. The effects of atmospheric drag are, however, small, and the results obtained are comparable with those given in Bhatt's thesis. The implemented case corresponds with a maneovure lasting 7200 seconds and using 3 CMG's.

The problem is formulated as follows. Find $\mathbf{q_c}(t) = [q_{c,1}(t)\, q_{c,2}(t)\, q_{c,3}(t)\, q_{c,4}]^T$, $t \in [t_0, t_f]$ and the scalar parameter γ to minimise,

$$J = 0.1\gamma + \int_{t_0}^{t_f} ||u(t)||^2 \mathrm{d}t \tag{3.180}$$

subject to the dynamical equations:

$$\dot{\mathbf{q}}(t) = \frac{1}{2}\mathbf{T}(\mathbf{q})(\omega(t) - \omega_o(\mathbf{q}))$$
$$\dot{\omega}(t) = \mathbf{J}^{-1}\left(\tau_d(\mathbf{q}) - \omega(t) \times (\mathbf{J}\omega(t)) - \mathbf{u}(t)\right) \tag{3.181}$$
$$\dot{\mathbf{h}}(t) = \mathbf{u}(t) - \omega(t) \times \mathbf{h}(t)$$

the path constraints:

$$||\mathbf{q}(t)||_2^2 = 1$$
$$||\mathbf{q}_c(t)||_2^2 = 1$$
$$||\mathbf{h}(t)||_2^2 \leq \gamma \tag{3.182}$$
$$||\dot{\mathbf{h}}(t)||_2^2 = \dot{h}_{\max}^2$$

the parameter bounds

$$0 \leq \gamma \leq h_{\max}^2 \tag{3.183}$$

and the boundary conditions:

$$\mathbf{q}(t_0) = \bar{\mathbf{q}}_0 \quad \omega(t_0) = \omega_o(\bar{\mathbf{q}}_0) \quad \mathbf{h}(t_0) = \bar{\mathbf{h}}_0$$
$$\mathbf{q}(t_f) = \bar{\mathbf{q}_f} \quad \omega(t_f) = \omega_o(\bar{\mathbf{q}}_f) \quad \mathbf{h}(t_f) = \bar{\mathbf{h}}_f \tag{3.184}$$

where $\mathbf{J}$ is a $3{\times}3$ inertia matrix, $\mathbf{q} = [q_1, q_2, q_3, q_4]^T$ is the quarternion vector, ω is the spacecraft angular rate relative to an inertial reference frame and expressed in the body frame, $\mathbf{h}$ is the momentum, $\mathbf{T}(\mathbf{q})$ is given by:

$$\mathbf{T}(\mathbf{q}) = \begin{bmatrix} -q_2 & -q_3 & -q_4 \\ q_1 & -q_4 & q_3 \\ q_4 & q_1 & -q_2 \\ -q_3 & q_2 & q_1 \end{bmatrix} \tag{3.185}$$

406

u is the control force, which is given by:

$$\mathbf{u}(t) = \mathbf{J}\left(K_P\tilde{\varepsilon}(q,q_c) + K_D\tilde{\omega}(\omega,q_c)\right) \qquad (3.186)$$

where

$$\tilde{\varepsilon}(\mathbf{q},\mathbf{q}_c) = 2\mathbf{T}(\mathbf{q}_c)^T\mathbf{q}$$
$$\tilde{\omega}(\omega,\omega_c) = \omega - \omega_c \qquad (3.187)$$

ω_o is given by:

$$\omega_o(\mathbf{q}) = n\mathbf{C}_2(\mathbf{q}) \qquad (3.188)$$

where n is the orbital rotation rate, $\mathbf{C}_j$ is the j column of the rotation matrix:

$$\mathbf{C}(\mathbf{q}) = \begin{bmatrix} 1-2(q_3^2+q_4^2) & 2(q_2q_3+q_1q_4) & 2(q_2q_4-q_1q_3) \\ 2(q_2q_3-q_1q_4) & 1-2(q_2^2+q_4^2) & 2(q_3q_4+q_1q_2) \\ 2(q_2q_4+q_1q_3) & 2(q_3q_4-q_1q_2) & 1-2(q_2^2+q_3^2) \end{bmatrix} \qquad (3.189)$$

τ_d is the disturbance torque, which in this case only incorporates the gravity gradient torque τ_{gg} (the disturbance torque also incorporates the atmospheric drag torque in the original study):

$$\tau_d = \tau_{gg} = 3n^2\mathbf{C}_3(\mathbf{q}) \times (\mathbf{J}\mathbf{C}_3(\mathbf{q})) \qquad (3.190)$$

The constant parameter values used were: $n = 1.1461 \times 10^{-3}$ rad/s, $h_{\max} = 3 \times 3600.0$ ft-lbf-sec, $\dot{h}_{\max} = 200.0$ ft-lbf, $t_0 = 0$ s, $t_f = 7200$ s, and

$$\mathbf{J} = \begin{bmatrix} 18836544.0 & 3666370.0 & 2965301.0 \\ 3666370.0 & 27984088.0 & -1129004.0 \\ 2965301.0 & -1129004.0 & 39442649.0 \end{bmatrix} \text{slug} - \text{ft}^2 \qquad (3.191)$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////
////////////////            zpm.cxx               /////////////////////
//////////////////////////////////////////////////////////////////////
////////////////          PSOPT Example           /////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////// Title:  Zero propellant maneouvre problem  ////////////////
//////// Last modified: 09 November 2009            ////////////////
//////// Reference:     S.A. Bhatt (2007), Masters Thesis,/////////////
////////               Rice University, Houston TX    ////////////////
//////////////////////////////////////////////////////////////////////
////////      Copyright (c) Victor M. Becerra, 2009     ////////////////
//////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser /////////////
//////// General Public License (LGPL)                ////////////////
//////////////////////////////////////////////////////////////////////

#include "psopt.h"


// Set CASE below to 1: 6000 s maneouvre, or to 2: 7200 maneouvre
#define CASE 2
```

```
struct Constants {
DMatrix J;
double n;
double Kp;
double Kd;
double hmax;
};

typedef struct Constants Constants_;

static Constants_ CONSTANTS;


void Tfun( adouble T[][3], adouble *q )
{

adouble q1 = q[CINDEX(1)];
adouble q2 = q[CINDEX(2)];
adouble q3 = q[CINDEX(3)];
adouble q4 = q[CINDEX(4)];

T[CINDEX(1)][CINDEX(1)] = -q2 ; T[CINDEX(1)][CINDEX(2)] = -q3; T[CINDEX(1)][CINDEX(3)] = -q4;

T[CINDEX(2)][CINDEX(1)] =  q1 ; T[CINDEX(2)][CINDEX(2)] = -q4; T[CINDEX(2)][CINDEX(3)] =  q3;

T[CINDEX(3)][CINDEX(1)] =  q4 ; T[CINDEX(3)][CINDEX(2)] =  q1; T[CINDEX(3)][CINDEX(3)] = -q2;

T[CINDEX(4)][CINDEX(1)] = -q3;  T[CINDEX(4)][CINDEX(2)] =  q2; T[CINDEX(4)][CINDEX(3)] =  q1;


}


void compute_omega0(adouble* omega0, adouble* q)
{
// This function computes the angular speed in the rotating LVLH reference frame
int i;
double n = CONSTANTS.n;
adouble C2[3];

        adouble q1 = q[CINDEX(1)];
        adouble q2 = q[CINDEX(2)];
        adouble q3 = q[CINDEX(3)];
        adouble q4 = q[CINDEX(4)];

C2[ CINDEX(1) ] = 2*(q2*q3 + q1*q4);
C2[ CINDEX(2) ] = 1.0-2.0*(q2*q2+q4*q4);
C2[ CINDEX(3) ] = 2*(q3*q4-q1*q2);

for (i=0;i<3;i++)   omega0[i] = -n*C2[i];

}

void compute_control_torque(adouble* u, adouble* q, adouble* qc, adouble* omega )
{
// This function computes the control torque
//

double Kp = CONSTANTS.Kp; // Proportional gain
double Kd = CONSTANTS.Kd; // Derivative gain
double n  = CONSTANTS.n;  // Orbital rotation rate [rad/s]


int i, j;

DMatrix& J = CONSTANTS.J;


adouble T[4][3];

Tfun( T, q );

adouble Tc[4][3];

Tfun( Tc, qc );

adouble epsilon_tilde[3];

for(i=0;i<3;i++) {
epsilon_tilde[i] = 0.0;
for(j=0;j<4;j++) {
epsilon_tilde[i] += 2*Tc[j][i]*q[j];
```

```
}
}

adouble omega_c[3];

compute_omega0( omega_c, qc );

adouble omega_tilde[3];

for(i=0;i<3;i++) {
omega_tilde[i] = omega[i]-omega_c[i];
}

adouble uaux[3];

for(i=0;i<3;i++) {
uaux[i]= Kp*epsilon_tilde[i]+Kd*omega_tilde[i];
}

product_ad( J, uaux, 3, u );



}

void quarternion2Euler( DMatrix& phi, DMatrix& theta, DMatrix& psi, DMatrix& q)
{
// This function finds the Euler angles given the quarternion vector
//
long N = q.GetNoCols();
DMatrix q0 = q(1,colon());
  DMatrix q1 = q(2,colon());
  DMatrix q2 = q(3,colon());
DMatrix q3 = q(4,colon());

phi.Resize(1,N);
theta.Resize(1,N);
psi.Resize(1,N);

    for(int i=1;i<=N;i++) {
phi(i)=atan2( 2*(q0(i)*q1(i) + q2(i)*q3(i)), 1.0-2.0*(q1(i)*q1(i)+q2(i)*q2(i)) );
                theta(i)=asin( 2*(q0(i)*q2(i)-q3(i)*q1(i)) );
                psi(i) = atan2( 2*(q0(i)*q3(i)+q1(i)*q2(i)), 1.0-2.0*(q2(i)*q2(i)+q3(i)*q3(i)) );
}
}



void compute_aerodynamic_torque(adouble* tau_aero, adouble& time )
{
// This function approximates the aerodynamic torque by using the model and
// parameters given in the following reference:
// A. Chun Lee (2003) "Robust Momemtum Manager Controller for Space Station Applications".
// Master of Arts Thesis, Rice University.
//

double alpha1[3] = {1.0, 4.0, 1.0};
        double alpha2[3] = {1.0, 2.0, 1.0};
        double alpha3[3] = {0.5, 0.5, 0.5};
adouble t = time;

double phi1 = 0.0;
double phi2 = 0.0;

double n = CONSTANTS.n;

  for(int i=0;i<3;i++) {
                // Aerodynamic torque in [lb-ft]
tau_aero[i] = alpha1[i] + alpha2[i]*sin( n*t + phi1 ) + alpha3[i]*sin( 2*n*t + phi2);
}
}


////////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
////////////////////////////////////////////////////////////////////////


adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                        adouble* parameters,adouble& t0, adouble& tf,
                        adouble* xad, int iphase, Workspace* workspace)
{
```

```
double end_point_weight = 0.1;

        adouble gamma      = parameters[ CINDEX(1) ];

return (end_point_weight*gamma);

}

///////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
///////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                       adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
double running_cost_weight = 1.0;

adouble q[4]; // quarternion vector
adouble u[3]; // control torque

q[CINDEX(1)] = states[ CINDEX(1) ];
q[CINDEX(2)] = states[ CINDEX(2) ];
q[CINDEX(3)] = states[ CINDEX(3) ];
q[CINDEX(4)] = states[ CINDEX(4) ];

adouble omega[3]; // angular rate vector

omega[CINDEX(1)] = states[ CINDEX(5) ];
omega[CINDEX(2)] = states[ CINDEX(6) ];
omega[CINDEX(3)] = states[ CINDEX(7) ];


adouble qc[4]; // control vector

qc[CINDEX(1)] = controls[ CINDEX(1) ];
qc[CINDEX(2)] = controls[ CINDEX(2) ];
qc[CINDEX(3)] = controls[ CINDEX(3) ];
qc[CINDEX(4)] = controls[ CINDEX(4) ];

compute_control_torque(u,q,qc,omega);

return running_cost_weight*dot(u,u,3);

}

///////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ///////////////////////////////////
///////////////////////////////////////////////////////////////////////
void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

int i,j;


double n = CONSTANTS.n; // Orbital rotation rate [rad/s]

adouble q[4]; // quarternion vector

q[CINDEX(1)] = states[ CINDEX(1) ];
q[CINDEX(2)] = states[ CINDEX(2) ];
q[CINDEX(3)] = states[ CINDEX(3) ];
q[CINDEX(4)] = states[ CINDEX(4) ];

adouble omega[3]; // angular rate vector

omega[CINDEX(1)] = states[ CINDEX(5) ];
omega[CINDEX(2)] = states[ CINDEX(6) ];
omega[CINDEX(3)] = states[ CINDEX(7) ];

adouble h[3]; // momentum vector

h[CINDEX(1)] = states[ CINDEX(8) ];
h[CINDEX(2)] = states[ CINDEX(9) ];
h[CINDEX(3)] = states[ CINDEX(10) ];

adouble qc[4]; // control vector

qc[CINDEX(1)] = controls[ CINDEX(1) ];
qc[CINDEX(2)] = controls[ CINDEX(2) ];
qc[CINDEX(3)] = controls[ CINDEX(3) ];
```

```
qc[CINDEX(4)] = controls[ CINDEX(4) ];

adouble C2[3], C3[3];

adouble u[3];

adouble gamma;

gamma = parameters[ CINDEX(1) ];

// Inertia matrix in slug-ft^2

DMatrix& J = CONSTANTS.J;

DMatrix Jinv; Jinv = inv(J);

adouble q1 = q[CINDEX(1)];
adouble q2 = q[CINDEX(2)];
adouble q3 = q[CINDEX(3)];
adouble q4 = q[CINDEX(4)];

C3[ CINDEX(1) ] = 2*(q2*q4 - q1*q3);
C3[ CINDEX(2) ] = 2*(q3*q4 + q1*q2);
C3[ CINDEX(3) ] = 1.0-2.0*(q2*q2 + q3*q3);


adouble T[4][3];

Tfun( T, q );

adouble qdot[4];

adouble omega0[3];

compute_omega0( omega0, q);

// Quarternion attitude kinematics

for(j=0;j<4;j++) {

qdot[j]=0;

for(i=0;i<3;i++) {

    qdot[j] += 0.5*T[j][i]*(omega[i]-omega0[i]);

}

}


adouble Jomega[3];

product_ad( J, omega, 3, Jomega );

adouble omegaCrossJomega[3];

cross(omega,Jomega, omegaCrossJomega);

adouble F[3];

// Compute the torque disturbances:

adouble tau_grav[3], tau_aero[3];

adouble v1[3];

for(i=0;i<3;i++) {
  v1[i] = 3*pow(n,2)*C3[i];
}

adouble JC3[3];

product_ad( J, C3, 3, JC3 );

//gravity gradient torque

cross( v1, JC3, tau_grav );

//Aerodynamic torque
```

411

```
compute_aerodynamic_torque(tau_aero, time );

for(i=0;i<3;i++) {
  // Uncomment this section to ignore the aerodynamic disturbance torque
  tau_aero[i] = 0.0;
}
adouble tau_d[3];

for (i=0;i<3;i++) {

    tau_d[i] =  tau_grav[i] + tau_aero[i];

}

compute_control_torque(u, q, qc, omega );


for (i=0;i<3;i++) {
   F[i]      = tau_d[i] - omegaCrossJomega[i] - u[i];
}

adouble omega_dot[3];

// Rotational dynamics

product_ad( Jinv, F, 3, omega_dot );

adouble OmegaCrossH[3];

cross( omega, h , OmegaCrossH );

adouble hdot[3];


//Momemtum derivative

for(i=0; i<3; i++) {
   hdot[i] = u[i] - OmegaCrossH[i];
}


derivatives[CINDEX(1)] =       qdot[ CINDEX(1) ];
derivatives[CINDEX(2)] =       qdot[ CINDEX(2) ];
derivatives[CINDEX(3)] =       qdot[ CINDEX(3) ];
derivatives[CINDEX(4)] =       qdot[ CINDEX(4) ];
derivatives[CINDEX(5)] =       omega_dot[ CINDEX(1) ];
derivatives[CINDEX(6)] =       omega_dot[ CINDEX(2) ];
derivatives[CINDEX(7)] =       omega_dot[ CINDEX(3) ];
derivatives[CINDEX(8)] =       hdot[ CINDEX(1) ];
derivatives[CINDEX(9)] =       hdot[ CINDEX(2) ];
derivatives[CINDEX(10)]=       hdot[ CINDEX(3) ];


path[ CINDEX(1) ] =   dot( q,  q,  4);
path[ CINDEX(2) ] =   dot( qc, qc, 4);
path[ CINDEX(3) ] =   dot( h, h, 3 ) - gamma;   //  <= 0
path[ CINDEX(4) ] =   dot( hdot, hdot,3); // <= hdotmax^2,

}






//////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ///////////////////////////
//////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{

adouble q1_i       = initial_states[CINDEX(1)];
adouble q2_i       = initial_states[CINDEX(2)];
adouble q3_i       = initial_states[CINDEX(3)];
adouble q4_i       = initial_states[CINDEX(4)];
adouble omega1_i   = initial_states[CINDEX(5)];
adouble omega2_i   = initial_states[CINDEX(6)];
```

```
adouble omega3_i    = initial_states[CINDEX(7)];
adouble h1_i        = initial_states[CINDEX(8)];
adouble h2_i        = initial_states[CINDEX(9)];
adouble h3_i        = initial_states[CINDEX(10)];

adouble q1_f        = final_states[CINDEX(1)];
adouble q2_f        = final_states[CINDEX(2)];
adouble q3_f        = final_states[CINDEX(3)];
adouble q4_f        = final_states[CINDEX(4)];
adouble omega1_f    = final_states[CINDEX(5)];
adouble omega2_f    = final_states[CINDEX(6)];
adouble omega3_f    = final_states[CINDEX(7)];
adouble h1_f        = final_states[CINDEX(8)];
adouble h2_f        = final_states[CINDEX(9)];
adouble h3_f        = final_states[CINDEX(10)];

// Initial conditions

e[ CINDEX(1) ] = q1_i;
e[ CINDEX(2) ] = q2_i;
e[ CINDEX(3) ] = q3_i;
e[ CINDEX(4) ] = q4_i;
e[ CINDEX(5) ] = omega1_i;
e[ CINDEX(6) ] = omega2_i;
e[ CINDEX(7) ] = omega3_i;
e[ CINDEX(8) ] = h1_i;
e[ CINDEX(9) ] = h2_i;
e[ CINDEX(10)] = h3_i;

// Final conditions

 e[ CINDEX(11) ] = q1_f;
 e[ CINDEX(12) ] = q2_f;
 e[ CINDEX(13) ] = q3_f;
 e[ CINDEX(14) ] = q4_f;
 e[ CINDEX(15) ] = omega1_f;
 e[ CINDEX(16) ] = omega2_f;
 e[ CINDEX(17) ] = omega3_f;
 e[ CINDEX(18) ] = h1_f;
 e[ CINDEX(19) ] = h2_f;
 e[ CINDEX(20) ] = h3_f;


}



////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function //////////////////
////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
    // Single phase
}



////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ////////////////////////////////
////////////////////////////////////////////////////////////////////

int main(void)
{

////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ///////////////////////////////
////////////////////////////////////////////////////////////////////

   Alg  algorithm;
   Sol  solution;
   Prob problem;



   CONSTANTS.Kp = 0.000128; // Proportional gain
   CONSTANTS.Kd = 0.015846; // Derivative gain

   double hmax; // maximum momentum magnitude  in [ft-lbf-sec]


   if (CASE==1) {
CONSTANTS.n = 1.1461E-3; // Orbital rotation rate [rad/s]
```

```
hmax = 4*3600.0; // 4 CMG's
   }
   else if (CASE==2) {
CONSTANTS.n = 1.1475E-3;
hmax = 3*3600.0; // 3 CMG's
   }

   CONSTANTS.hmax = hmax;

   DMatrix& J = CONSTANTS.J;

   J.Resize(3,3);

   // Inertia matrix in slug-ft^2

   if (CASE==1) {
J(1,1) = 17834580.0 ; J(1,2)= 2787992.0;  J(1,3)= 2873636.0;
J(2,1) = 2787992.0  ; J(2,2)= 2773815.0; J(2,3)= -863810.0;
J(3,1) = 28736361.0 ; J(3,2)= -863810.0; J(3,3)=  38030467.0;
   }

   else if (CASE==2) {
J(1,1) = 18836544.0 ; J(1,2)= 3666370.0;  J(1,3)= 2965301.0;
J(2,1) = 3666370.0  ; J(2,2)= 27984088.0; J(2,3)= -1129004.0;
J(3,1) = 2965301.0  ; J(3,2)= -1129004.0; J(3,3)=  39442649.0;
   }


//////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  /////////////////////////////
//////////////////////////////////////////////////////////////////////

   problem.name           = "Zero Propellant Maneouvre of the ISS";
   problem.outfilename               = "zpm.txt";

//////////////////////////////////////////////////////////////////////
/////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////

   problem.nphases    = 1;
   problem.nlinkages               = 0;

   psopt_level1_setup(problem);

//////////////////////////////////////////////////////////////////////
/////////    Define phase related information & do level 2 setup  ////////////
//////////////////////////////////////////////////////////////////////


   problem.phases(1).nstates    = 10;
   problem.phases(1).ncontrols  = 4;
   problem.phases(1).nevents            = 20;
   problem.phases(1).npath      = 4;
   problem.phases(1).nodes                   = "[20, 30, 40, 50, 60]";


   problem.phases(1).nparameters             = 1;


   psopt_level2_setup(problem, algorithm);

//////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ////////////////////
//////////////////////////////////////////////////////////////////////

   // Control bounds

   problem.phases(1).bounds.lower.controls(1) =  -1.0;
   problem.phases(1).bounds.lower.controls(2) =  -1.0;
   problem.phases(1).bounds.lower.controls(3) =  -1.0;
   problem.phases(1).bounds.lower.controls(4) =  -1.0;


   problem.phases(1).bounds.upper.controls(1) =   1.0;
   problem.phases(1).bounds.upper.controls(2) =   1.0;
   problem.phases(1).bounds.upper.controls(3) =   1.0;
   problem.phases(1).bounds.upper.controls(4) =   1.0;

   // state bounds

   problem.phases(1).bounds.lower.states(1) =  -1.0;
   problem.phases(1).bounds.lower.states(2) =  -0.2;
```

```
    problem.phases(1).bounds.lower.states(3) =  -0.2;
    problem.phases(1).bounds.lower.states(4) =  -1.0;
    problem.phases(1).bounds.lower.states(5) =  -1.E-2;
    problem.phases(1).bounds.lower.states(6) =  -1.E-2;
    problem.phases(1).bounds.lower.states(7) =  -1.E-2;
    problem.phases(1).bounds.lower.states(8) =  -8000.0;
    problem.phases(1).bounds.lower.states(9) =  -8000.0;
    problem.phases(1).bounds.lower.states(10)=  -8000.0;

    problem.phases(1).bounds.upper.states(1) =  1.0;
    problem.phases(1).bounds.upper.states(2) =  0.2;
    problem.phases(1).bounds.upper.states(3) =  0.2;
    problem.phases(1).bounds.upper.states(4) =  1.0;
    problem.phases(1).bounds.upper.states(5) =  1.E-2;
    problem.phases(1).bounds.upper.states(6) =  1.E-2;
    problem.phases(1).bounds.upper.states(7) =  1.E-2;
    problem.phases(1).bounds.upper.states(8) =  8000.0;
    problem.phases(1).bounds.upper.states(9) =  8000.0;
    problem.phases(1).bounds.upper.states(10)=  8000.0;

  // Parameter bound


    problem.phases(1).bounds.lower.parameters(1) =  0.0;
    problem.phases(1).bounds.upper.parameters(1) =  hmax*hmax;


    // Event bounds

    // Initial / Final  condition values:
    DMatrix q_i(4,1),  q_f(4,1), omega_i(3,1), omega_f(3,1), h_i(3,1), h_f(3,1);

    adouble q_ad[4], omega_ad[3];

  // Initial conditions
    if (CASE==1) {
     q_i(1) =  0.98966;
     q_i(2) =  0.02690;
     q_i(3) = -0.08246;
     q_i(4) =  0.11425;

        q_ad[ CINDEX(1) ]=q_i(1);
        q_ad[ CINDEX(2) ]=q_i(2);
        q_ad[ CINDEX(3) ]=q_i(3);
        q_ad[ CINDEX(4) ]=q_i(4);
        compute_omega0( omega_ad, q_ad);
        omega_i(1) = omega_ad[ CINDEX(1) ].value();
        omega_i(2) = omega_ad[ CINDEX(2) ].value();
        omega_i(3) = omega_ad[ CINDEX(3) ].value();

 //    omega_i(1) = -2.5410E-4;
 //    omega_i(2) = -1.1145E-3;
 //    omega_i(3) =  8.2609E-5;

    h_i(1) =  -496.0;
    h_i(2) =  -175.0;
    h_i(3) = -3892.0;
    }

    else if (CASE==2) {
q_i(1) =  0.98996;
     q_i(2) =  0.02650;
     q_i(3) = -0.07891;
     q_i(4) =  0.11422;
        q_ad[ CINDEX(1) ]=q_i(1);
        q_ad[ CINDEX(2) ]=q_i(2);
        q_ad[ CINDEX(3) ]=q_i(3);
        q_ad[ CINDEX(4) ]=q_i(4);
        compute_omega0( omega_ad, q_ad);
        omega_i(1) = omega_ad[ CINDEX(1) ].value();
        omega_i(2) = omega_ad[ CINDEX(2) ].value();
        omega_i(3) = omega_ad[ CINDEX(3) ].value();

 //    omega_i(1) = -2.5470E-4;
 //    omega_i(2) = -1.1159E-3;
 //    omega_i(3) =  8.0882E-5;

    h_i(1) =  1000.0;
    h_i(2) =  -500.0;
    h_i(3) = -4200.0;
    }

    // Final conditions
```

```
    q_f(1) = 0.70531;
    q_f(2) = -0.06201;
    q_f(3) = -0.03518;
    q_f(4) = -0.70531;

    q_ad[ CINDEX(1) ]=q_f(1);
    q_ad[ CINDEX(2) ]=q_f(2);
    q_ad[ CINDEX(3) ]=q_f(3);
    q_ad[ CINDEX(4) ]=q_f(4);
    compute_omega0( omega_ad, q_ad);
    omega_f(1) = omega_ad[ CINDEX(1) ].value();
    omega_f(2) = omega_ad[ CINDEX(2) ].value();
    omega_f(3) = omega_ad[ CINDEX(3) ].value();

//    omega_f(1) = 1.1353E-3;
//    omega_f(2) = 3.0062E-6;
//    omega_f(3) = -1.5713E-4;

    h_f(1) = -9.0;
    h_f(2) = -3557.0;
    h_f(3) =  -135.0;

    double DQ = 0.0001;
    double DWF = 0.0;
    double DHF = 0.0;

    problem.phases(1).bounds.lower.events(1) = q_i(1)-DQ;
    problem.phases(1).bounds.lower.events(2) = q_i(2)-DQ;
    problem.phases(1).bounds.lower.events(3) = q_i(3)-DQ;
    problem.phases(1).bounds.lower.events(4) = q_i(4)-DQ;
    problem.phases(1).bounds.lower.events(5) = omega_i(1);
    problem.phases(1).bounds.lower.events(6) = omega_i(2);
    problem.phases(1).bounds.lower.events(7) = omega_i(3);
    problem.phases(1).bounds.lower.events(8) = h_i(1);
    problem.phases(1).bounds.lower.events(9) = h_i(2);
    problem.phases(1).bounds.lower.events(10)= h_i(3);
    problem.phases(1).bounds.lower.events(11) = q_f(1)-DQ;
    problem.phases(1).bounds.lower.events(12) = q_f(2)-DQ;
    problem.phases(1).bounds.lower.events(13) = q_f(3)-DQ;
    problem.phases(1).bounds.lower.events(14) = q_f(4)-DQ;
    problem.phases(1).bounds.lower.events(15) = omega_f(1)-DWF;
    problem.phases(1).bounds.lower.events(16) = omega_f(2)-DWF;
    problem.phases(1).bounds.lower.events(17) = omega_f(3)-DWF;
    problem.phases(1).bounds.lower.events(18) = h_f(1)-DHF;
    problem.phases(1).bounds.lower.events(19) = h_f(2)-DHF;
    problem.phases(1).bounds.lower.events(20)= h_f(3)-DHF;


    problem.phases(1).bounds.upper.events(1) = q_i(1)+DQ;
    problem.phases(1).bounds.upper.events(2) = q_i(2)+DQ;
    problem.phases(1).bounds.upper.events(3) = q_i(3)+DQ;
    problem.phases(1).bounds.upper.events(4) = q_i(4)+DQ;
    problem.phases(1).bounds.upper.events(5) = omega_i(1);
    problem.phases(1).bounds.upper.events(6) = omega_i(2);
    problem.phases(1).bounds.upper.events(7) = omega_i(3);
    problem.phases(1).bounds.upper.events(8) = h_i(1);
    problem.phases(1).bounds.upper.events(9) = h_i(2);
    problem.phases(1).bounds.upper.events(10)= h_i(3);
    problem.phases(1).bounds.upper.events(11) = q_f(1)+DQ;
    problem.phases(1).bounds.upper.events(12) = q_f(2)+DQ;
    problem.phases(1).bounds.upper.events(13) = q_f(3)+DQ;
    problem.phases(1).bounds.upper.events(14) = q_f(4)+DQ;
    problem.phases(1).bounds.upper.events(15) = omega_f(1)+DWF;
    problem.phases(1).bounds.upper.events(16) = omega_f(2)+DWF;
    problem.phases(1).bounds.upper.events(17) = omega_f(3)+DWF;
    problem.phases(1).bounds.upper.events(18) = h_f(1)+DHF;
    problem.phases(1).bounds.upper.events(19) = h_f(2)+DHF;
    problem.phases(1).bounds.upper.events(20)=  h_f(3)+DHF;



    // Path bounds

    double hdotmax = 200.0; // [ ft-lbf ]

    double EQ_TOL = 0.0002;

    problem.phases(1).bounds.lower.path(1) = 1.0-EQ_TOL;
    problem.phases(1).bounds.upper.path(1) = 1.0+EQ_TOL;

    problem.phases(1).bounds.lower.path(2) = 1.0-EQ_TOL;
    problem.phases(1).bounds.upper.path(2) = 1.0+EQ_TOL;
```

```
    problem.phases(1).bounds.lower.path(3) = -hmax*hmax;
    problem.phases(1).bounds.upper.path(3) =  0.0;


    problem.phases(1).bounds.lower.path(4) = 0.0;
    problem.phases(1).bounds.upper.path(4) = hdotmax*hdotmax;

    // Time bounds

    double TFINAL;

    if (CASE==1) {
TFINAL = 6000.0;
    }
    else {
TFINAL = 7200.0;
    }

    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = TFINAL;
    problem.phases(1).bounds.upper.EndTime      = TFINAL;




////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  ////////////////////////
////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae            = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ////////////////////
////////////////////////////////////////////////////////////////////////

    DMatrix time_guess = linspace(0.0, TFINAL, 50 );
    DMatrix state_guess = zeros(10,50);
    DMatrix control_guess = zeros(4,50);
    DMatrix parameter_guess = hmax*hmax*ones(1,1);

    control_guess(1, colon() ) = linspace( q_i(1), q_i(1), 50 );
    control_guess(2, colon() ) = linspace( q_i(2), q_i(2), 50 );
    control_guess(3, colon() ) = linspace( q_i(3), q_i(3), 50 );
    control_guess(4, colon() ) = linspace( q_i(4), q_i(4), 50 );

    state_guess(1, colon() ) = linspace( q_i(1), q_i(1), 50);
    state_guess(2, colon() ) = linspace( q_i(2), q_i(2), 50);
    state_guess(3, colon() ) = linspace( q_i(3), q_i(3), 50);
    state_guess(4, colon() ) = linspace( q_i(4), q_i(4), 50);

    state_guess(5, colon() ) = linspace( omega_i(1), omega_f(1), 50);
    state_guess(6, colon() ) = linspace( omega_i(2), omega_f(2), 50);
    state_guess(7, colon() ) = linspace( omega_i(3), omega_f(3), 50);

    state_guess(8, colon() ) = linspace( h_i(1), h_f(1), 50);
    state_guess(9, colon() ) = linspace( h_i(2), h_f(2), 50);
    state_guess(10, colon()) = linspace( h_i(3), h_f(3), 50);



    problem.phases(1).guess.controls = control_guess;
    problem.phases(1).guess.states   = state_guess;
    problem.phases(1).guess.time     = time_guess;
    problem.phases(1).guess.parameters=parameter_guess;




////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  ///////////////////////////
////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max              = 1000;
    algorithm.nlp_tolerance             = 1.e-5;
    algorithm.nlp_method                = "IPOPT";
```

```
    algorithm.scaling                = "automatic";
    algorithm.derivatives            = "automatic";
    algorithm.defect_scaling         = "jacobian-based";
    algorithm.jac_sparsity_ratio     = 0.104;




////////////////////////////////////////////////////////////////////////
/////////////////////  Now call PSOPT to solve the problem   //////////////////////
////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////


    DMatrix states, controls, t;

    states     = solution.get_states_in_phase(1);
    controls   = solution.get_controls_in_phase(1);
    t          = solution.get_time_in_phase(1);


////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////

    states.Save("states.dat");
    controls.Save("controls.dat");
    t.Save("t.dat");

    DMatrix omega, h, q, phi, theta, psi, qc, euler_angles;

    q      = states( colon(1,4), colon() );
    omega  = states( colon(5,7), colon() );
    h      = states( colon(8,10),colon() );
    qc     = controls;

    quarternion2Euler(phi, theta, psi,  q);

    euler_angles = phi && theta && psi;

    adouble  qc_ad[4], u_ad[3];

    DMatrix u(3,length(t));
    DMatrix hnorm(1,length(t));
    DMatrix hi;
    DMatrix hm = hmax*ones(1,length(t));

    int i,j;

    for (i=1; i<= length(t); i++ ) {
for(j=1;j<=3;j++) {
   omega_ad[j-1] = omega(j,i);
        }
for(j=1;j<=4;j++) {
   q_ad[j-1] = q(j,i);
        qc_ad[j-1]= qc(j,i);
        }

compute_control_torque(u_ad, q_ad, qc_ad, omega_ad );

for(j=1;j<=3;j++) {
   u(j,i) = u_ad[j-1].value();
        }

       hi = h(colon(),i);

hnorm(1,i) = enorm(hi);

    }

    omega = omega*(180.0/pi)*1000; // convert to mdeg/s

    phi = phi*180.0/pi; theta=theta*180.0/pi; psi=psi*180.0/pi;


    u.Save("u.dat");
```

```
    euler_angles.Save("euler_angles.dat");


/////////////////////////////////////////////////////////////////////
//////////  Plot some results if desired (requires gnuplot) //////////////
/////////////////////////////////////////////////////////////////////

    plot(t,q,problem.name+" quarternion elements: q",  "time (s)", "q", "q");

    plot(t,qc,problem.name+" Control variables: qc",  "time (s)", "qc", "qc");

    plot(t,phi,problem.name+" Euler angles: phi",      "time (s)", "angles (deg)", "phi");

    plot(t,theta,problem.name+" Euler angles: theta", "time (s)", "angles (deg)", "theta");

    plot(t,psi,problem.name+" Euler angle: psi",        "time (s)", "psi (deg)", "psi");

    plot(t,omega(1,colon()),problem.name+": omega 1","time (s)", "omega1", "omega1");

    plot(t,omega(2,colon()),problem.name+": omega 2","time (s)", "omega2", "omega2");

    plot(t,omega(3,colon()),problem.name+": omega 3","time (s)", "omega3", "omega3");

    plot(t,h(1,colon()),problem.name+": momentum 1","time (s)", "h1", "h1");

    plot(t,h(2,colon()),problem.name+": momentum 2","time (s)", "h2", "h2");

    plot(t,h(3,colon()),problem.name+": momentum 3","time (s)", "h3", "h3");

    plot(t,u(1,colon()),problem.name+": control torque 1","time (s)", "u1", "u1");

    plot(t,u(2,colon()),problem.name+": control torque 2","time (s)", "u2", "u2");

    plot(t,u(3,colon()),problem.name+": control torque 3","time (s)", "u3", "u3");

    plot(t,hnorm,t,hm,problem.name+": momentum norm", "time (s)", "h", "h hmax");



    plot(t,phi,problem.name+" Euler angles: phi",      "time (s)", "angles (deg)", "phi",
        "pdf", "zpm_phi.pdf" );

    plot(t,theta,problem.name+" Euler angles: theta", "time (s)", "angles (deg)", "theta",
        "pdf", "zpm_theta.pdf");

    plot(t,psi,problem.name+" Euler angle: psi",        "time (s)", "psi (deg)", "psi",
        "pdf", "zpm_psi.pdf");

    plot(t,omega(1,colon()),problem.name+": omega 1","time (s)", "omega1", "omega1",
        "pdf", "zpm_omega1.pdf");

    plot(t,omega(2,colon()),problem.name+": omega 2","time (s)", "omega2", "omega2",
        "pdf", "zpm_omega2.pdf");

    plot(t,omega(3,colon()),problem.name+": omega 3","time (s)", "omega3", "omega3",
        "pdf", "zpm_omega3.pdf");

    plot(t,h(1,colon()),problem.name+": momentum 1","time (s)", "h1", "h1",
            "pdf", "zpm_h1.pdf");

    plot(t,h(2,colon()),problem.name+": momentum 2","time (s)", "h2", "h2",
        "pdf", "zpm_h2.pdf");

    plot(t,h(3,colon()),problem.name+": momentum 3","time (s)", "h3", "h3",
        "pdf", "zpm_h3.pdf");

    plot(t,u(1,colon()),problem.name+": control torque 1","time (s)", "u1", "u1",
        "pdf", "zpm_u1.pdf");

    plot(t,u(2,colon()),problem.name+": control torque 2","time (s)", "u2", "u2",
        "pdf", "zpm_u2.pdf");

    plot(t,u(3,colon()),problem.name+": control torque 3","time (s)", "u3", "u3",
        "pdf", "zpm_u3.pdf");

    plot(t,hnorm,t,hm,problem.name+": momentum norm", "time (s)", "h (ft-lbf-sec)", "h hmax",
        "pdf", "zpm_hnorm.pdf");

    plot(t,u,problem.name+": control torques","time (s)", "u (ft-lbf)", "u1 u2 u3",
        "pdf", "zpm_controls.pdf");
```

```
}

//////////////////////////////////////////////////////////////////
/////////////////////////    END OF FILE    ////////////////////////////
//////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.112 to 3.124..

```
PSOPT results summary
=====================


Problem:  Zero Propellant Maneouvre of the ISS
CPU time (seconds): 8.252000e+01
NLP solver used:  IPOPT
Optimal (unscaled) cost function value:  6.680109e+06
Phase 1 endpoint cost function value: 5.239657e+06
Phase 1 integrated part of the cost:  1.440452e+06
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 7.200000e+03
Phase 1 maximum relative local error: 1.003102e-03
NLP solver reports:  The problem solved!
```
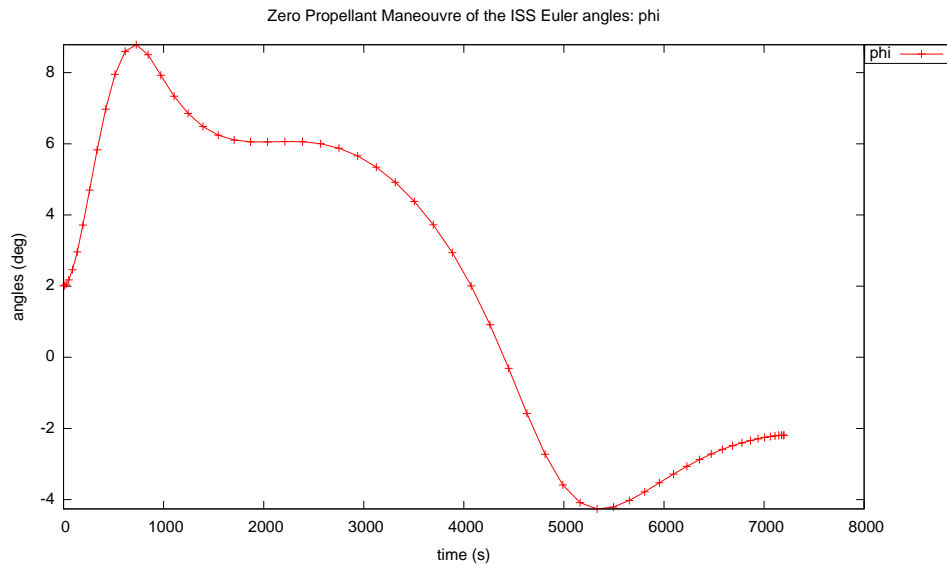
Figure 3.112: Euler angle ϕ (roll)
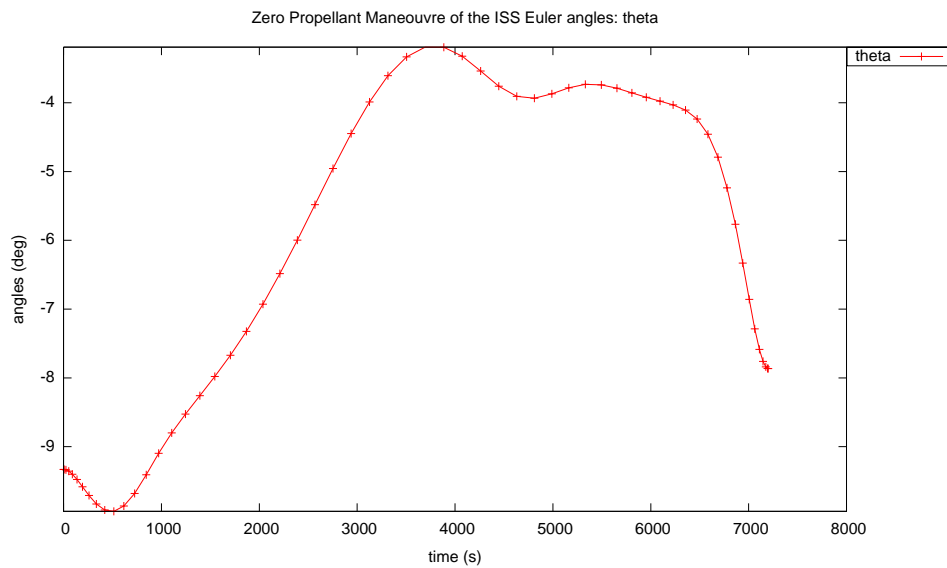
Figure 3.113: Euler angle θ (pitch)
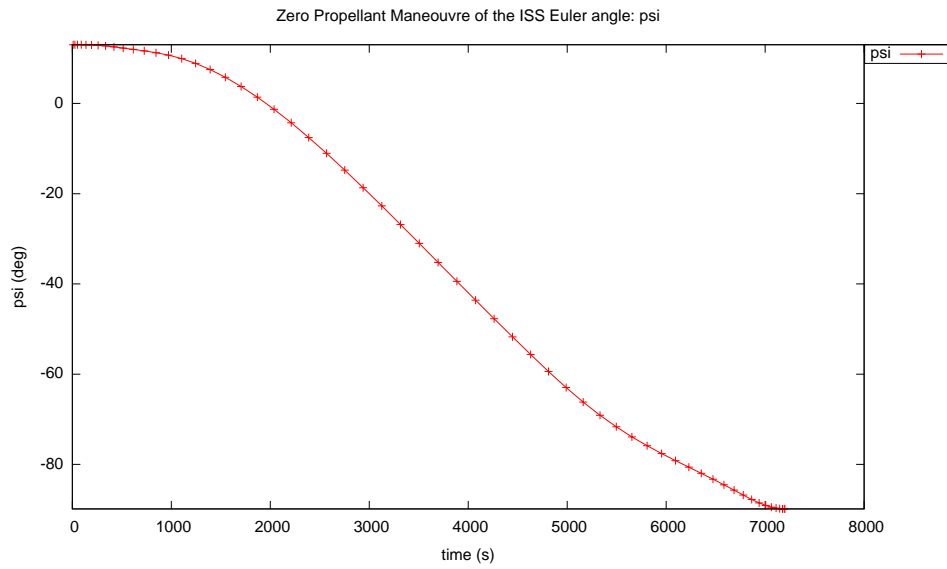
Figure 3.114: Euler angle ψ (yaw)
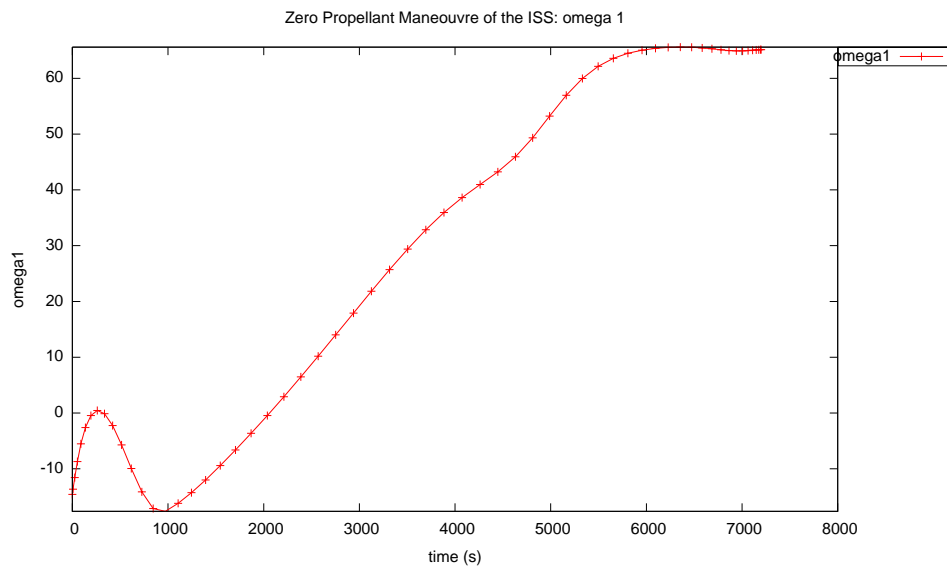
Figure 3.115: Angular speed ω_1 (roll)
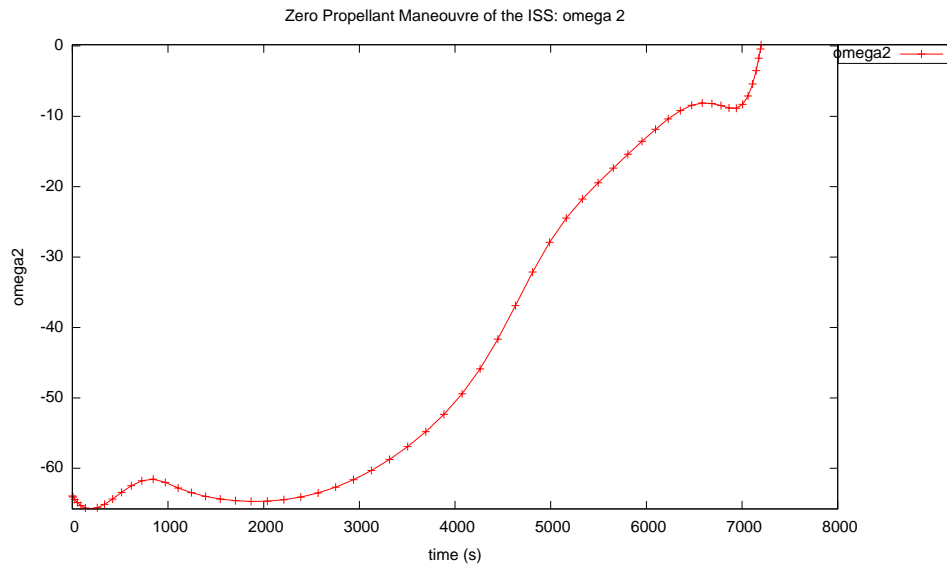
Figure 3.116: Angular speed ω_2 (pitch)
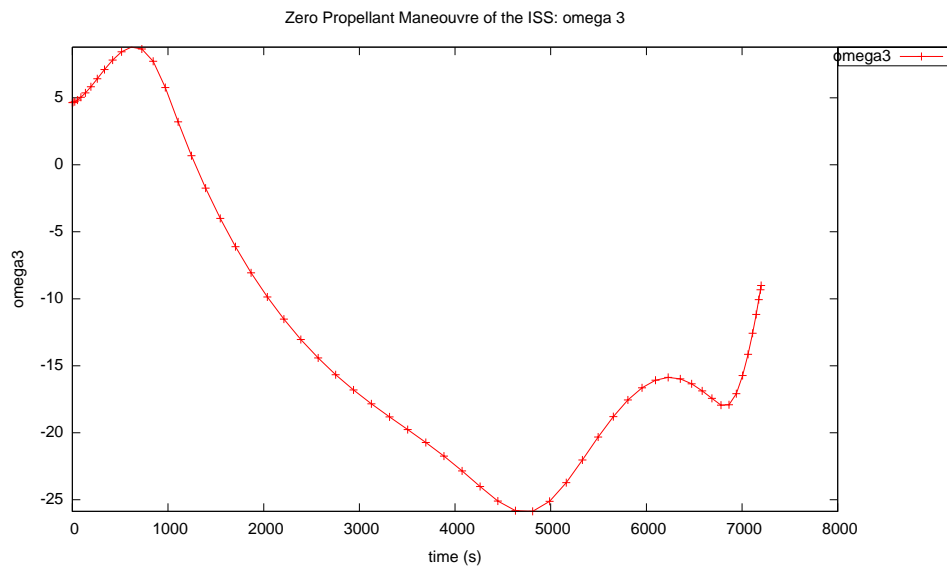
Figure 3.117: Angular speed ω_3 (yaw)
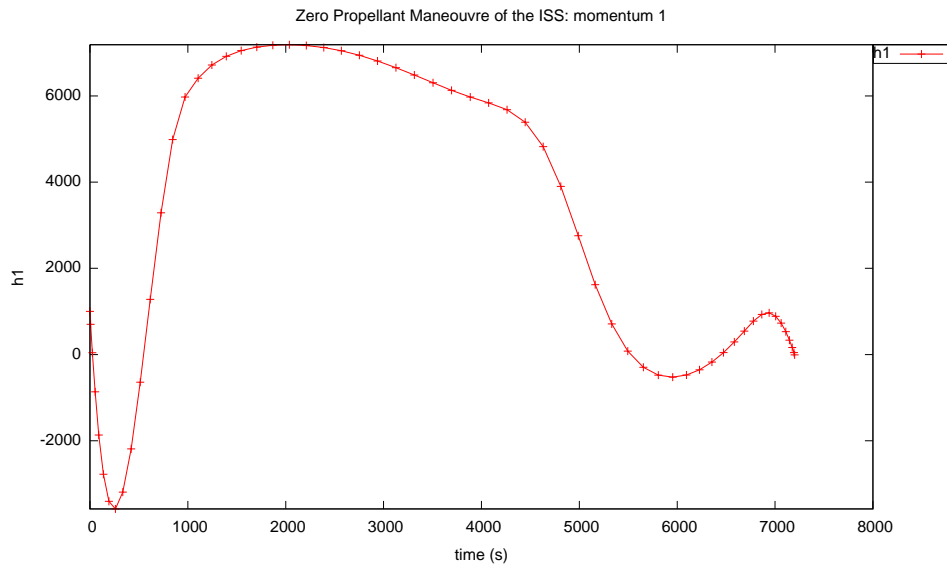
Figure 3.118: Momentum h_1 (roll)
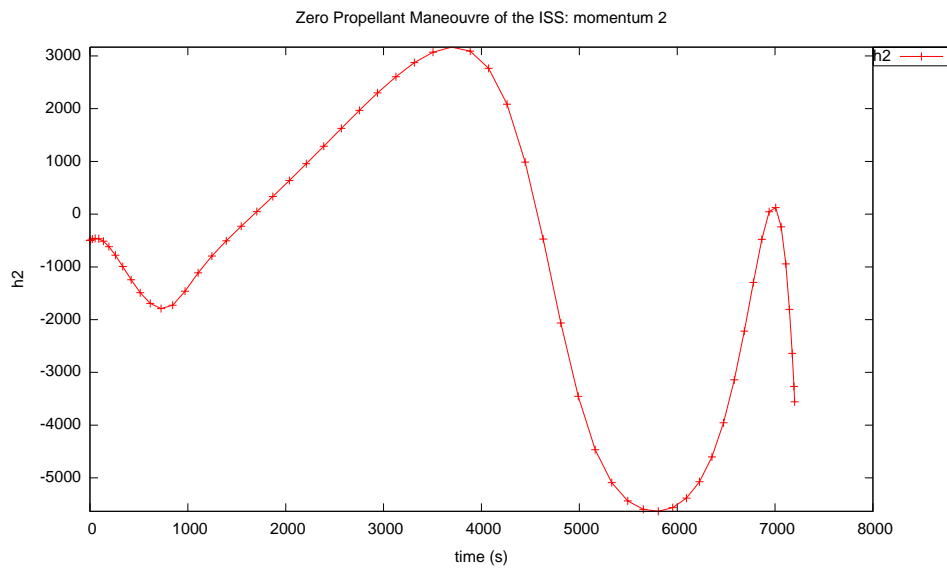
Figure 3.119: Momentum h_2 (pitch)
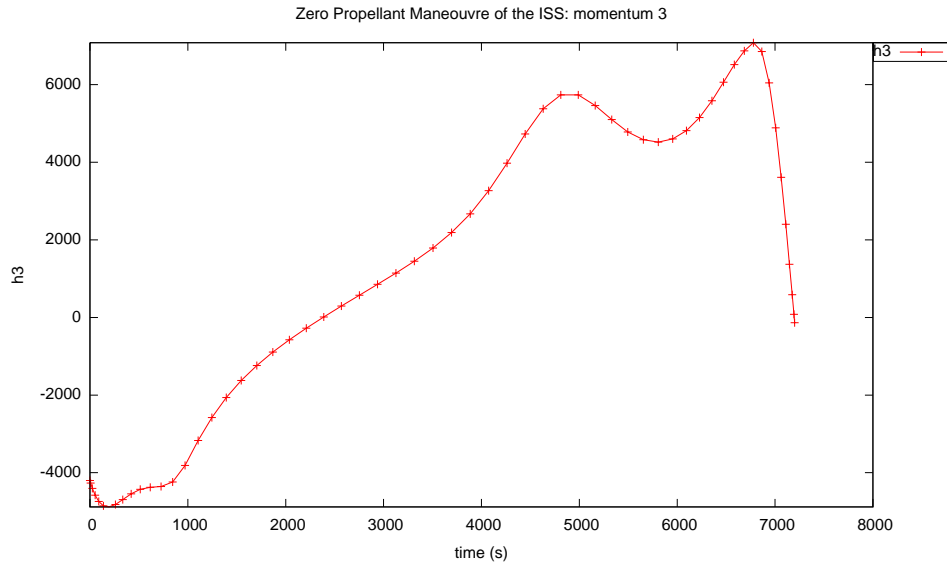
Figure 3.120: Momentum h_3 (yaw)
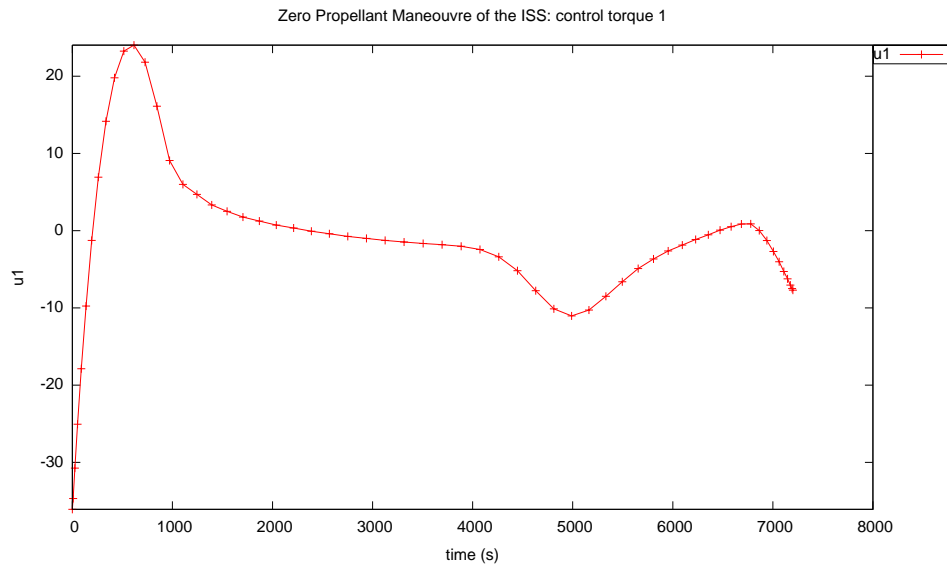
Figure 3.121: Control torque u_1 (roll)
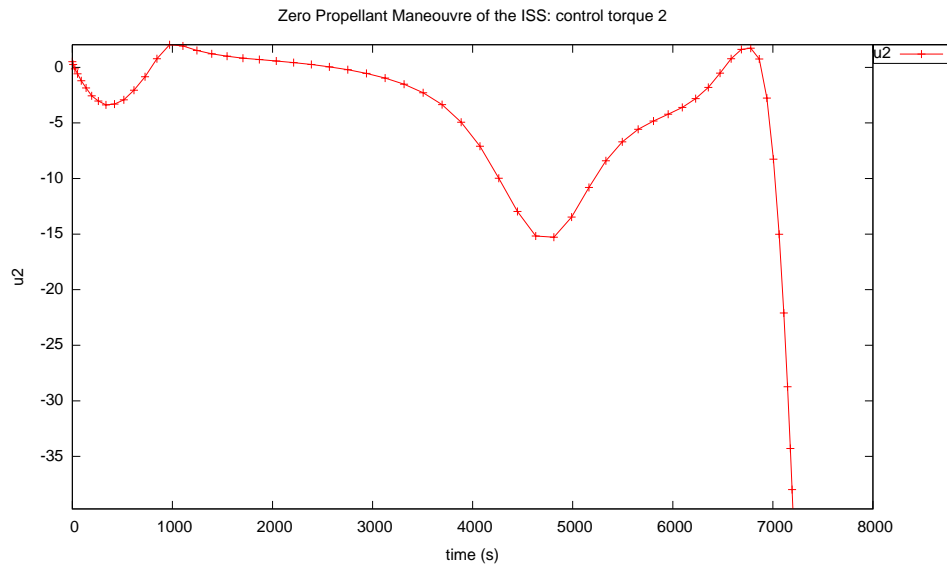
Figure 3.122: Control torque u_2 (pitch)
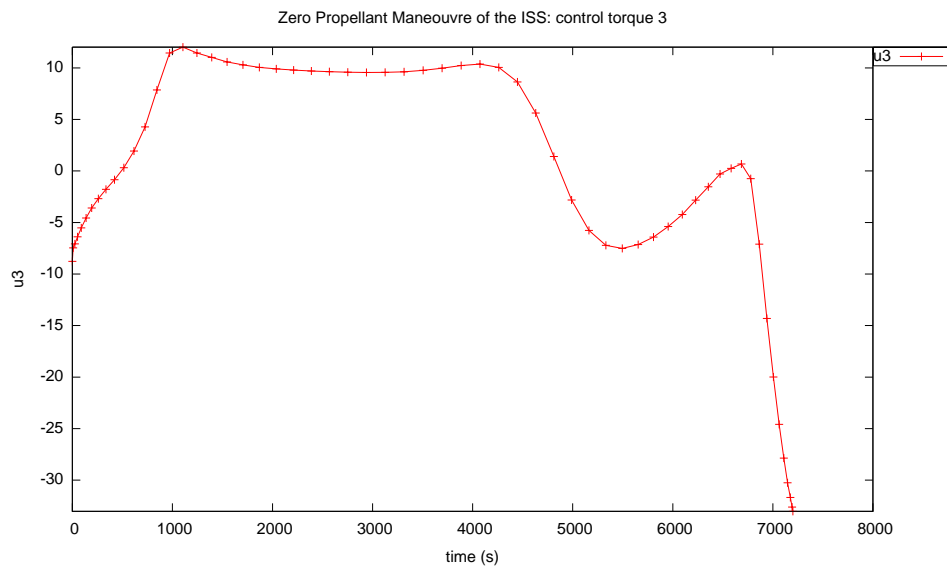
Figure 3.123: Control torque u_3 (yaw)

426
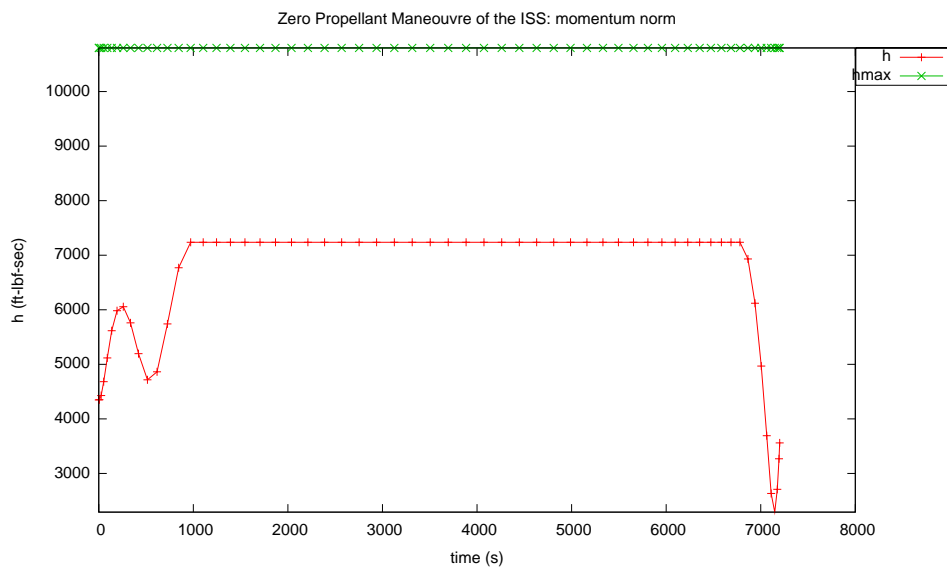
Figure 3.124: Momentum norm $||\mathbf{h}(t)||$

427

Acknowledgements

The author is grateful to Dr. Martin Otter from the Institute for Robotics and System Dynamics, DLR, Germany, for kindly allowing the publication of a translated version of the DLR model 2 of the Manutec R3 robot (original Fortran subroutine R3M2SI) with the distribution of $\mathcal{PSOPT}$.

The author is also grateful to Naz Bedrossian from the Draper Laboratory (USA) for facilitating the thesis of S. Bhatt, where the dynamic model of the International Space Station is described.

References

[1] N.S. Bedrossian, S. Bhatt, W. Kang, and I.M. Ross. Zero Propellant Maneuver Guidance. *IEEE Control Systems Magazine*, 29:53–73, 2009.

[2] D. A. Benson. *A Gauss Pseudospectral Transcription for Optimal Control*. PhD thesis, MIT, Department of Aeronautics and Astronautics, Cambridge, Mass., 2004.

[3] J. T. Betts. *Practical Methods for Optimal Control Using Nonlinear Programming*. SIAM, 2001.

[4] J. T. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. SIAM, 2010.

[5] S.A. Bhatt. Optimal reorientation of spacecraft using only control moment gyroscopes. Master's thesis, Rice University, Houston, Texas, 2007.

[6] A.E. Bryson. *Dynamic Optimization*. Addison-Wesley, 1999.

[7] A.E. Bryson, M.N. Desai, and W.C. Hoffman. Energy-State Approximation in Performance Optimization of Supersonic Aircraft. *Journal of Aircraft*, 6:481–488, 1969.

[8] A.E. Bryson and Yu-Chi Ho. *Applied Optimal Control*. Halsted Press, 1975.

[9] R.L. Burden and J.D. Faires. *Numerical Analysis*. Thomson Brooks/Cole, 2005.

[10] C. Canuto, M.Y. Hussaini, A. Quarteroni, and T.A. Zang. *Spectral Methods: Fundamentals in Single Domains*. Springer-Verlag, Berlin, 2006.

[11] C. Canuto, M.Y. Hussaini, A. Quarteroni, and T.A. Zang. *Spectral Methods: Evolution to Complex Geometries and Applications to Fluid Dynamics*. Springer, Heidelberg, Germany, 2007.

[12] C.G. Canuto, M.Y. Hussaini, A. Quarteroni A., and T.A. Zang. *Spectral Methods in Fluid Dynamics*. Springer-Verlag, 1988.

[13] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the Estimation of Sparse Jacobian Matrices. *Journal of the Institute of Mathematics and Applications*, 13:117–120, 1974.

[14] T. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[15] E. D. Dolan and J. J. More. *Benchmarking optimization software with COPS 3.0*. Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439, 2004.

[16] G. Elnagar, M. A. Kazemi, and M. Razzaghi. The Pseudospectral Legendre Method for Discretizing Optimal Control Problems. *IEEE Transactions on Automatic Control*, 40:1793–1796, 1995.

[17] F. Fahroo and I.M. Ross. Costate Estimation by a Legendre Pseudospectral Method. *Journal of Guidance, Control, and Dynamics*, 24:270–277, 2001.

[18] F. Fahroo and I.M. Ross. Costate Estimation by a Legendre Pseudospectral Method. *Journal of Guidance, Control, and Dynamics*, 24:270–277, 2002.

[19] F. Fahroo and I.M. Ross. Direct Trajectory Optimization by a Chebyshev Pseudospectral Method. *Journal of Guidance Control and Dynamics*, 25, 2002.

[20] J. Franke and M. Otter. The manutec r3 benchmark models for the dynamic simulation of robots. Technical report, Institute for Robotics and System Dynamics, DLR Oberpfaffenhofen, 1993.

[21] P. E. Gill, Walter Murray, and M. A. Saunders. SNOPT: An SQP algorithm for Large-Scale Constrained Optimization. *SIAM Review*, 47(1), 2001.

[22] P.E. Gill, W. Murray, M.A. Saunders, and M.H. Wright. Maintaining LU factors of a general sparse matrix. *Linear Algebra and its Applications*, 88/89:239–270, 1987.

[23] Q. Gong, F. Fahroo, and I.M. Ross. Spectral algorithms for pseudospectral methods in optimal control. *Journal of Guidance Control and Dynamics*, 31:460–471, 2008.

[24] J. Hesthaven, S. Gottlieb, and D. Gottlieb. *Spectral Methods for Time-Dependent Problems*. Cambridge University Press, Cambridge, 2007.

[25] L.S. Jennings, M.E. Fisher, K.L. Teo, and C.J. Goh. *MISER3 Optimal Control Software Version 2.0 Theory and User Manual.* Department of Mathematics, The University of Western Australia, 2002.

[26] W. Kang and N. Bedrossian. Pseudospectral Optimal Control Theory Makes Debut Flight, Saves nasa $1m in Under Three Hours. *SIAM News*, 40, 2007.

[27] W. Kang, Q. Gong, I. M. Ross, and F. Fahroo. On the Convergence of Nonlinear Optimal Control Using Pseudospectral Methods for Feedback Linearizable Systems. *International Journal of Robust and Nonlinear Control*, 17:1251–1277, 2007.

[28] E. Kostina, M.A. Saunders, and I. Schierle. Computation of covariance matrices for constrained parameter estimation problems using lsqr. Technical Report SOL-2009-1, Standford University, System Optimization Laboratory, 2009.

[29] Z. Li, M. R. Osborne, and T. Prvan. Parameter estimation of ordinary differential equations. *IMA Journal of Numerical Analysis*, 25:264–285, 2005.

[30] R. Luus. *Iterative Dynamic Programming.* Chapman and Hall / CRC, 2002.

[31] S. Marsili-Libelli, S. Guerrizio, and N. Checchi. Confidence regions of estimated parameters for ecological systems. *Ecological Modelling*, 165:127–146, 2003.

[32] M. Otter and S. Turk. The dfvrl models 1 and 2 of the manutec r3 robot. Technical report, Institute for Robotics and System Dynamics, DLR Oberpfaffenhofen, Germany, 1987.

[33] J.A. Pietz. Pseudospectral Collocation Methods for the Direct transcription of Optimal Control Problems. Master's thesis, Rice University, Houston, Texas, 2003.

[34] A.V. Rao, D. Benson, G. Huntington, and C. Francolin. *User's manual for GPOPS version 1.3: A Matlab package for dynamic optimization using the Gauss pseudospectral method.* 2008.

[35] A.V. Rao and K.D. Mease. Eigenvector Approximate Dichotomic Basis Method for Solving Hyper- sensitive Optimal Control Problems. *Optimal Control Applications and Methods*, 21:1–19, 2000.

[36] I.M. Ross and F. Fahroo. Pseudospectral Knotting Methods for Solving Nonsmooth Optimal Control Problems. *Journal of Guidance Control and Dynamics*, 27:397–405, 2004.

[37] P. E. Rutquist and M. M. Edvall. *PROPT Matlab Optimal Control Software.* TOMLAB Optimization, 2009.

[38] K. Schittkowski. *Numerical Data Fitting in Dynamical Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002.

[39] O. Von Stryk. *User's guide for DIRCOL (Version 2.1): A direct collocation method for the numerical solution of optimal control problems.* Technical Report, Technische Universitat Munchen, 1999.

[40] S. Subchan and R. Zbikowski. *Computational optimal control: tools and practice.* Wiley, 2009.

[41] K.L. Teo, C.J. Goh, and K.H. Wong. *A Unified Computational Approach to Optimal Control Problems.* Wiley, New York, 1991.

[42] Lloyd N. Trefethen. *Spectral Methods in MATLAB.* SIAM, Philadelphia, 2000.

[43] D.A. Vallado. *Fundamentals of Astrodynamics and Applications.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.

[44] J. Vlassenbroeck and R. Van Doreen. A Chebyshev Technique for Solving Nonlinear Optimal Control Problems. *IEEE Transactions on Automatic Control*, 33:333–340, 1988.

[45] A. Wächter and L. T. Biegler. On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming. *Mathematical Programming*, 106:25–57, 2006.