$\mathcal{PSOPT}$ User Manual

Victor Becerra
Email: v.m.becerra@ieee.org
www: http://www.psopt.net
March 22, 2025

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.

Contents

1. Introduction to $\mathcal{PSOPT}$

1.1 About $\mathcal{PSOPT}$

$\mathcal{PSOPT}$ 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
- 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. See https://projects.coin-or.org/Ipopt for further details.
- SNOPT: is a Sequential Quadratic Programming algorithm for the optimisation of constrained large-scale problems. See http://www.sbsi-sol-optimize.com/manuals/SNOPT-Manual.pdf for further details.

1.1.1 Reasons to use $\mathcal{PSOPT}$

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 free linear solver `MUMPS`.

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:

http://groups.google.com/group/psopt-users-group

1.2.1 About the author

Victor M. Becerra obtained his first degree in Electrical Engineering in 1990, and worked for two years in power systems analysis and control for a power generation and transmission company. 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. Between 2000 and 2015 he was an academic at the School of Systems Engineering, University of Reading, UK, where he became a Professor of Automatic Control in 2012. Between 2011 and 2012, he was seconded at the Ford Motor Company in Dunton, Essex, with funding by the Royal Academy of Engineering, where he developed methods for the calibration of gasoline engine oil temperature dynamic models. In 2015, he took the position of Professor of Power Systems Engineering at the University of Portsmouth, UK. He is a Senior Member of the IEEE, a Senior Member of the AIAA, and a Fellow of the Institute of Engineering and Technology. During his career, he has received research funding from the EPSRC, the Royal Academy of Engineering, the European Union, the Knowledge Transfer Partnership programme, Innovate UK and UK industry. He has published over 150 research papers and two books. His web site is:

https://researchportal.port.ac.uk/en/persons/victor-becerra

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 at his email address `victor.becerra@port.ac.uk` or `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 (bug tracking system: https://github.com/PSOPT/psopt/issues/)
- Sending corrections to the documentation, please use the above link.

- 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 $\mathcal{PSOPT}$, usually through pull requests in GitHub.
- 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. The recommended reference for $\mathcal{PSOPT}$ is:

  – 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

  and the following is the recommended form to cite this document:

  – Becerra, V.M. (2020). *PSOPT Optimal Control Solver User Manual. Release 5.* Available: https://github.com/PSOPT/psopt/blob/master/doc/
- Developing interfaces to other NLP solvers.

1.3 Rolling Release

From March 2025, $\mathcal{PSOPT}$ will feature a rolling release mode. Rolling release is a concept in software development of frequently delivering updates to applications. This is in contrast to a standard or point release development model which uses software versions which replace the previous version. Users can download the latest source code from the GitHub repository. The documentation will also be updated on a rolling release basis.

1.4 Other documents in this set

The $\mathcal{PSOPT}$ documentation is now made of this document, which introduces the software, its methods and its usage. A second document provides details and results from a number of examples. Finally, the GitHub page provides the latest installation instructions.

1.5 GitHub repository and home page

The downloadable $\mathcal{PSOPT}$ distribution is held in its GitHub page:

https://github.com/PSOPT/psopt

A web page is maintained to provide general information about $\mathcal{PSOPT}$:

http://www.psopt.net

1.6 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

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

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 [2].

1.7 Overview of the Legendre and Chebyshev pseudospectral methods

1.7.1 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 [7, 6]. Moreover, over the last 15 years or so, pseudospectral techniques have emerged as important computational methods for solving optimal control problems [9, 10, 12, 19, 14]. 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 [21]. 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 [6].

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 [5, 21], 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 [9]. Since then, authors such as Ross, Fahroo and co-workers have analysed, extended and applied the method. For instance, convergence analysis is presented in [15], while an extension of the method to multi-phase problems is given in [19]. An application that has received publicity is the use of the Legendre pseudospectral method for generating real time trajectories for a NASA spacecraft maneouvre [14]. The Chebyshev pseudospectral method for optimal control problems was originally proposed in 1988 [22]. Fahroo and Ross proposed an alternative method for trajectory optimisation using Chebyshev polynomials [12].

Some details on approximating continuous functions using Legendre and Chebyshev polynomials are given below. Interested readers are referred to [5] for further details.

1.8 Pseudospectral approximations

1.8.1 Interpolation and the Lagrange polynomial

It is a well known fact in numerical analysis [4] 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.

1.8.2 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 $L^2_w[-1, 1]$ 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 $f \in L^2_w[-1, 1]$ 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.

1.8.3 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 [5]. 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:

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

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

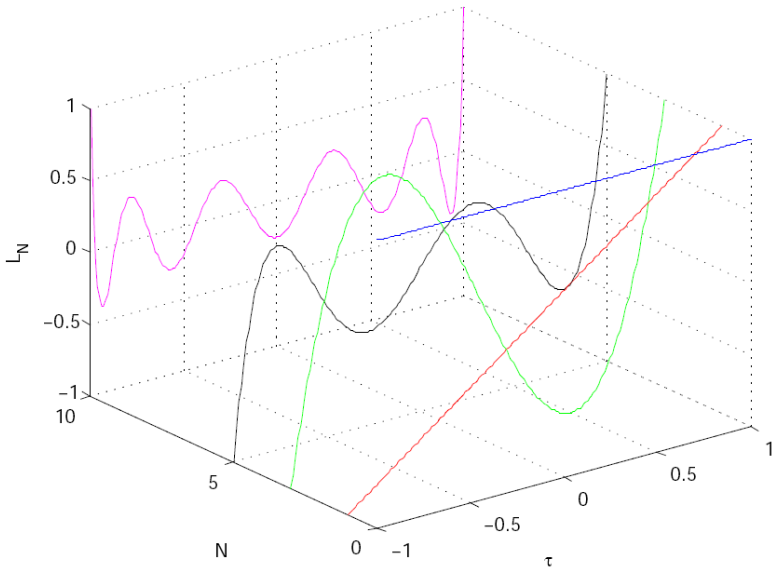

Figure 1.1: Illustration of the Legendre polynomials $L_N(\tau)$ for $N = 0, 1, 2, 4, 5, 10$.

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.

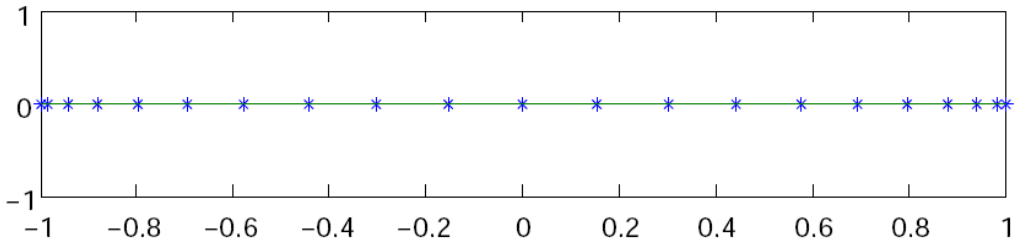

Figure 1.2: Illustration of the Legendre Gauss Lobatto (LGL) nodes for $N = 20$.

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:

$$w_k = \frac{2}{N(N+1)}\frac{1}{[L_N(\tau_k)]^2},\ k = 0,\ldots,N. \tag{1.5}$$

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 $\mathcal{O}(10^{-7})$.

1.8.4 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 [13]) that with these weights, function $f : [-1, 1] \to \Re$ 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 [13]:

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

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* [5].

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.

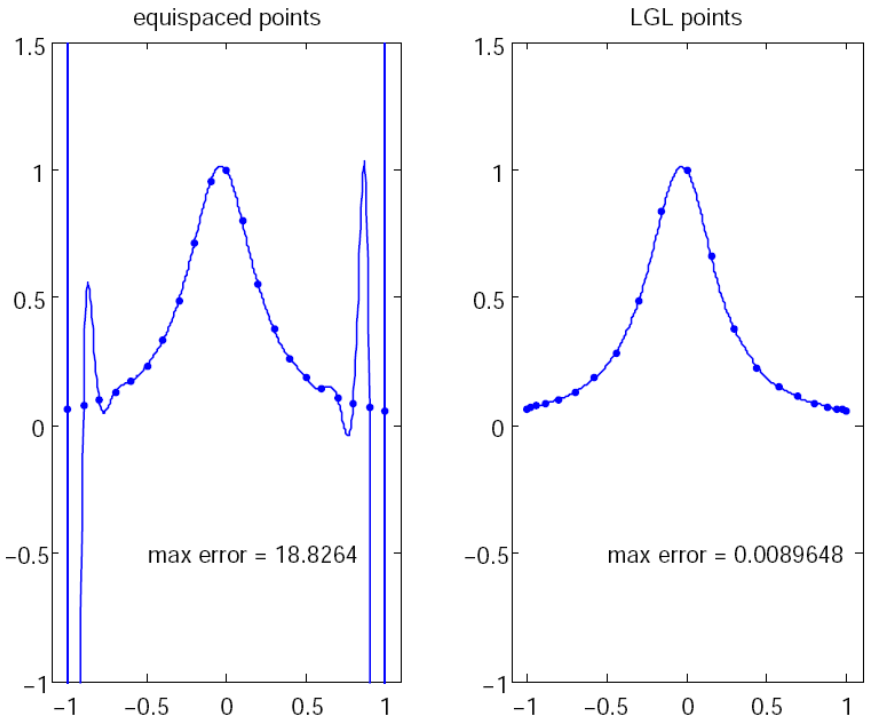

Figure 1.3: Illustration of polynomial interpolation over equispaced and LGL nodes

.

1.8.5 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:

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

where

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

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*.

1.8.6 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:

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

where

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

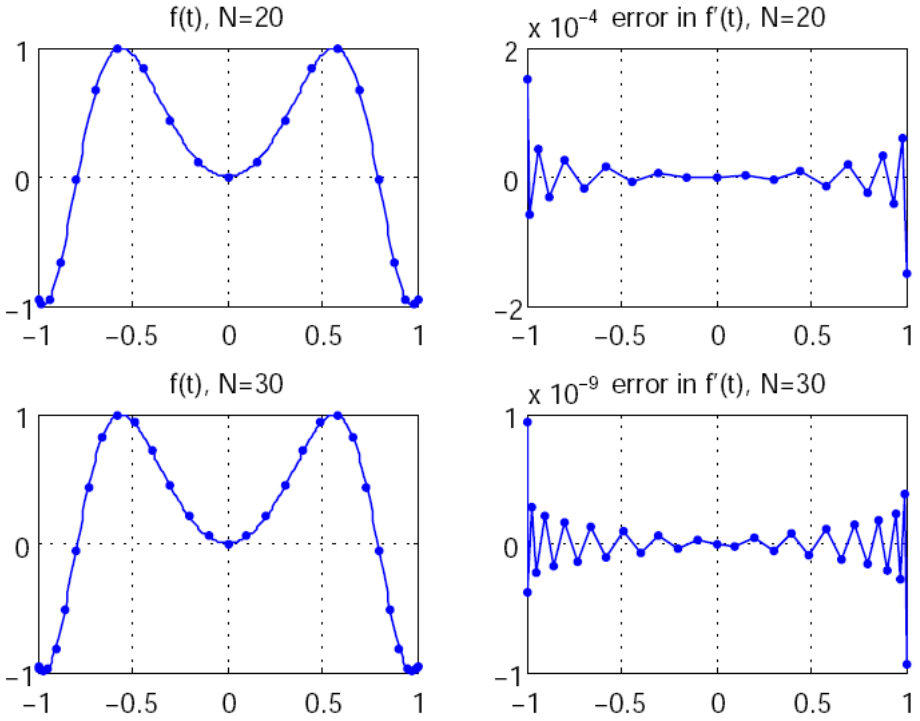

Figure 1.4: Legendre differentiation of $f(t) = \sin(5t^2)$ for $N = 20$ and $N = 30$.

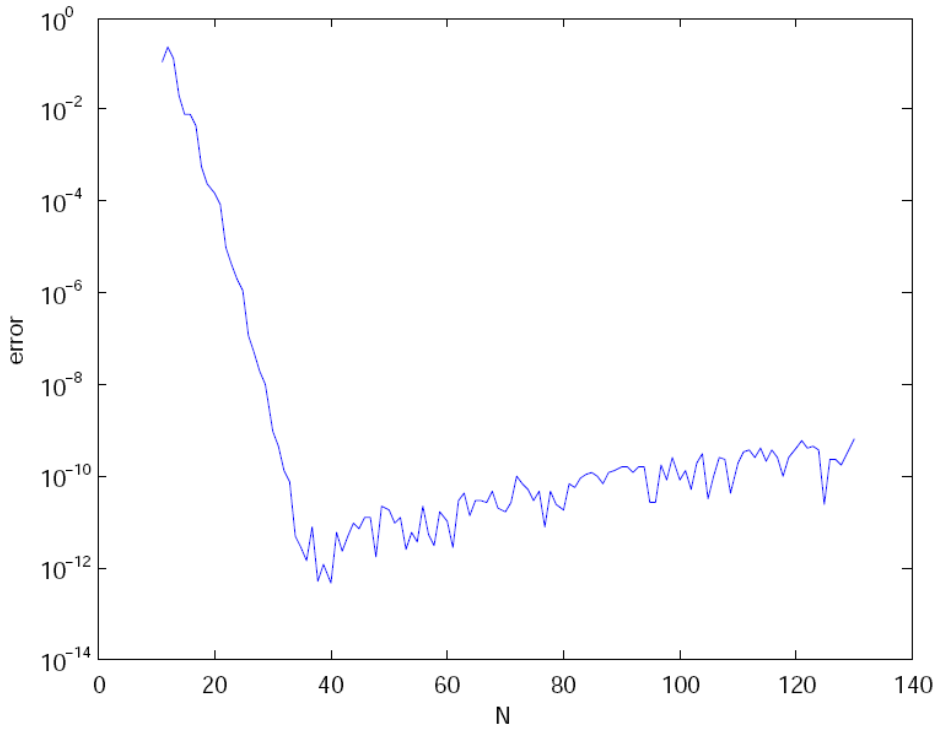

Figure 1.5: Maximum error in the Legendre differentiation of $f(t) = \sin(5t^2)$ as a function of N.

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:

$$f^N(\tau_k) = f(\tau_k) \tag{1.17}$$

1.8.7 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

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

which is known as the differentiation matrix.

1.8.8 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:

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

1.8.9 Differentiation with reduced round-off errors

The following differentiation matrix, which offers reduced round-off errors [5], is employed optionally by $\mathcal{PSOPT}$. It can be used both with Legrendre and Chebyshed points.

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

1.9 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

$$\begin{aligned}
&u : [t_0, t_f] \to \mathcal{R}^{n_u} \\
&x : [t_0, t_f] \to \mathcal{R}^{n_x} \\
&p \in \mathcal{R}^{n_p} \\
&\varphi : \mathcal{R}^{n_x} \times \mathcal{R}^{n_x} \times \mathcal{R}^{n_p} \times \mathcal{R} \times \mathcal{R} \to \mathcal{R} \\
&L : \mathcal{R}^{n_x} \times \mathcal{R}^{n_u} \times \mathcal{R}^{n_p} \times [t_0, t_f] \to \mathcal{R} \\
&f : \mathcal{R}^{n_x} \times \mathcal{R}^{n_u} \times \mathcal{R}^{n_p} \times [t_0, t_f] \to \mathcal{R}^{n_x} \\
&h : \mathcal{R}^{n_x} \times \mathcal{R}^{n_u} \times \mathcal{R}^{n_p} \times [t_0, t_f] \to \mathcal{R}^{n_h} \\
&e : \mathcal{R}^{n_x} \times \mathcal{R}^{n_u} \times \mathcal{R}^{n_x} \times \mathcal{R}^{n_u} \times \Re^{n_p} \times \mathcal{R} \times \mathcal{R} \to \mathcal{R}^{n_e}
\end{aligned} \tag{1.25}$$

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:

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

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:

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

$$\underline{t}_f \leq t_f \leq \bar{t}_f,$$
$$t_f - t_0 \geq 0,$$

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 (1.12).

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

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

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:

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

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:

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

and the bounds on the decision vector are given by:

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

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.

1.9.1 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 $\mathcal{P}_3$ [11]:

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

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 [11] 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 [18]:

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

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 co-states at $\tau = \pm 1$ are found using linear extrapolation. The costate estimates are also smoothed as described in 1.9.1

1.9.2 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 programming 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:

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

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 $\mathcal{P}_1$.

1.10 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.

1.10.1 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 $g : \mathcal{R}^{n_x} \times \mathcal{R}^{n_u} \times \mathcal{R}^{n_p} \times \mathcal{R} \to \mathcal{R}^{n_o}$ 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:

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

1.10.2 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 $\left\{\tilde{y}^{(i)}\right\}_{k=1}^{n_s^{(i)}}$ 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:

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

1.10.3 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 [16], 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 [17]:

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

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 [20] 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 [20].

1.10.4 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 [3], 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) may modify the numbers by eliminating redundant constraints or decision variables. These modifications are also visible when a problem is run.

1.11 Alternative local discretizations

Direct collocation methods that use local information to approximate the functions associated with an optimal control problem are well established [2]. 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 [2]. 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 multi-phase problems.

1.11.1 Trapezoidal method

With the trapezoidal method [2], 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.

1.11.2 Hermite-Simpson method

With the Hermite-Simpson method [2], the defect constraints are computed as follows:

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

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

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

with

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

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

1.11.3 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:

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

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.

1.11.4 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 1.9.1.

1.12 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 [1]. 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.

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.

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,
```

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

```
                    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[ 0 ];
    adouble x2f = final_states[ 1 ];

    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:

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

```
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 + 0.01*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:

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -3\exp(x_1) - 4x_2 + u \\ 0 &\leq x_1^2 + x_2^2 \leq 1 \end{aligned} \tag{2.3}$$

```
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 ] = -3*exp(x1)-4*x2+u;
    path[        0 ] = x1*x1 + x2*x2
}
```

Note that the bounds on the path constraints are specified separately, possibly in the `main()` function, as follows:

```
problem.phases(1).bounds.upper.path(0) = 0.0;
problem.phases(1).bounds.lower.path(0) = 1.0;
```

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)
```

Please note, if the initial or final values of the control variables within a phase need to be accessed within the `events()` function, this can be done through the functions `get_initial_controls()` and `get_final_controls()`. See sections 2.10.11 and 2.10.12.

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

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

```
void events(adouble* e,
            adouble* initial_states,
            adouble* final_states,
            adouble* parameters,
            adouble& t0,
            adouble& tf,
            int iphase,
            Workspace* workspace)
{
    adouble x1i = initial_states[   0 ];
    adouble x2i = initial_states[   1 ];
    adouble x1f = final_states[     0 ];
    adouble x2f = final_states[     1 ];


    e[ 0 ] = x1i;
    e[ 1 ] = x2i;
    e[ 2 ] = x1f*x2f;
```

```
}
```

Note that the bounds on the event constraints are specified separately, possibly in the `main()` function:

```
problem.phases(1).bounds.lower.events(0) = 1.0;
problem.phases(1).bounds.upper.events(0)= 1.0;
problem.phases(1).bounds.lower.events(1)= 2.0;
problem.phases(1).bounds.upper.events(1)= 2.0;
problem.phases(1).bounds.lower.events(2)=-0.1;
problem.phases(1).bounds.upper.events(2)=0.1;
```

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:

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

These type of state and time continuity constraint can be entered automatically by using the `auto_link` function as shown below. Note that phase linkage bounds are set to zero by default. If they are different from zero, they need to be specified explicitly as shown in the second example 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.

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

$$\begin{aligned} 0 = \Psi_1 &= x_1^{(1)}(t_f^{(1)}) - \sin[x_1^{(2)}(t_i^{(2)})] \\ 0 = \Psi_2 &= x_2^{(1)}(t_f^{(1)}) - \cos[x_2^{(2)}(t_i^{(2)})] \\ 10 \leq \Psi_3 &= t_f^{(1)} - t_i^{(2)} \leq 100 \end{aligned} \tag{2.6}$$

```
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[0]=xf_p0[0]-sin(xi_p1[0]);
     linkages[1]=xf_p0[1]-cos(xi_p1[1]);
     linkages[2]=tf_p0 - ti_p1;
}
```

Note that the bounds are specified separately, possibly in the `main()` function:

```
problem.bounds.lower.linkage(0)= 0.0;
problem.bounds.upper.linkage(0)= 0.0;
problem.bounds.lower.linkage(1)= 0.0;
problem.bounds.upper.linkage(1)= 0.0;
problem.bounds.lower.linkage(2)= 10.0;
problem.bounds.upper.linkage(2)= 100.0;
```

2.2.6 Using the power of Eigen from within the user functions

Eigen is a C++ template library for linear algebra, and thus it is possible to use this to create Eigen matrices with `adouble` elements, and to make operations with them from within the user functions. Recall that `adouble` is a type defined by the automatic differentiation library ADOL-C. This has the potential of simplifying the implementation of user functions.

As an example, consider the `dae()` function. The list of arguments of this function includes arrays of type adouble named `states`, `controls`, `path` and `derivatives`. It is possible, for example, in the case of a problem with 3 states, 2 controls and 2 path constraints, to map these arrays into Eigen objects as follows:

```
Eigen::Map<Eigen::Matrix<adouble, 3, 1>> x(states, 3);
Eigen::Map<Eigen::Matrix<adouble, 2, 1>> u(controls, 2);
Eigen::Map<Eigen::Matrix<adouble, 2, 1>> p(path, 2);
Eigen::Map<Eigen::Matrix<adouble, 2, 1>> dx(derivatives, 3);
```

The user can then use these objects with the syntax and compatible functions that are available through Eigen. For example, assume that the dynamics are given by the linear equation $\dot{x} = Ax + Bu$, with A and B being matrices of the appropriate dimmension. One could write the following code inside the `dae()` function:

```
// Define the model matrices A and B
Eigen::Matrix<adouble, 3, 3> A;
Eigen::Matrix<adouble, 3, 2> B;
// Assume that values are assigned to the elements of A and B.
// Assume that x, u and dx are defined as above.
// The following code performs the necessary matrix operations
// and assigns the result to the derivatives array.
dx = A*x + B*u;
```

2.2.7 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);
```

Please note, $\mathcal{PSOPT}$ is able to solve problems with multiple phases, and some parameters need to be passed for each particular phase. In the $\mathcal{PSOPT}$ interface, the numbering of phases starts from 1, so that the first phase of the problem is phase 1, and problems with only a single phase have their phase number identified as 1.

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;
```

where `iphase` represents the number of the phase for which the parameters are being entered.

To specify the number of nodes (time instants) in the solution grid, there are different possibilities. If a single solution is required with a fixed number of nodes, this can be specified as follows:

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

If a sequence of nodes needs to be tried, so that manual mesh refinement is perform with an increasing number of nodes from solution to solution, and using the previous solution as a guess, then this can be specified in the example below:

```
problem.phases(iphase).nodes=(RowVectorXi(2)<<50, 60).finished();
```

Note, the number 2 in `RowVectorXi(2)` indicates the number of elements in the sequence, and the sequence of nodes itself is specified by the values $50, 60$. It is also possible to define the number of nodes using a previously defined variable of type `RowVectorXd`, as in the followign example:

```
RowVectorXi my_vector << 50, 60;
problem.phases(iphase).nodes = my_vector;
```

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 0 and `problem.phases(iphase).nstates-1`.

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 0 and `problem.phases(iphase).ncontrols`-1.

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 0 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 0 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 `MatrixXd` 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.

```
MatrixXd 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:

```
MatrixXd xGuess(2,20);
xGuess.row(0) = linspace( 10, 15, 20); // First row
xGuess.row(1) = 10*ones( 1, 20); // Second row
```

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

```
MatrixXd 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;
```

Passing user data to the interface functions

It is possible to pass user data to the different interface functions by using a void pointer that is a member of the `problem` data structure. This is good programming practice, as it helps avoid the use of global or static variables.

For example, the user could define a data structure to encapsulate some key data values, such as:

```
typedef struct{

double c1;
double c2;
double c3;

} Mydata;
```

In the `main` function, an instance of this data structure can be allocated:

```
Mydata* md = (Mydata*) malloc(sizeof(Mydata));
```

and its elements given values:

```
md->c1 = 1.0;
md->c2 = 100.0;
md->c3 = 1000.0;
```

In the `main()` function, after the problem data structure has been declared, then the pointer to the instance of the `Mydata` data structure can be stored in the `user_data` member of the `problem` data structure, as follows:

```
problem.user_data = (void*) md;
```

Finally, the user can access this data from within the interface functions. For example, to access the data from within the `integrand_cost` function, the following can be done:

```
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 ];

Mydata* md = (Mydata*) workspace->problem->user_data;

double c1 = md->c1;
double c2 = md->c2;
double c3 = md->c3;

return ( c1*x1*x1 + c2*x2*x2 + c3*u*u );

}
```

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;
algorithm.ipopt_linear_solver                   = STRING
```

Note that:

- `algorithm.nlp_method` takes the (only) option "IPOPT" (default).
- `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, this is the 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).
- `algorithm.ipopt_linear_solver` is a string indicating what linear solver should be used by IPOPT. The default value is "mumps", but other solvers are possible (see the IPOPT documentation). Any specified solver should be linked to the executable program as a dynamic or static library.
  - ma27: use the Harwell routine MA27
  - ma57: use the Harwell routine MA57

- ma77: use the Harwell routine MA77
- ma86: use the Harwell routine MA86
- ma97: use the Harwell routine MA97
- pardiso: use the Pardiso package
- wsmp: use WSMP package
- mumps: use MUMPS package (default)

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:

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

Save(x,"x.dat");
```

```
Save(u,"u.dat");
Save(t,"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(MatrixXd& x,
          MatrixXd& 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(MatrixXd& x1, MatrixXd& y1,
          MatrixXd& x2, MatrixXd& y2,
          const string& title,
          char* xlabel,
          char* ylabel,
          char* legend=NULL,
          char* terminal=NULL,
          char* output=NULL);

void plot(MatrixXd& x1, MatrixXd& y1,
          MatrixXd& x2, MatrixXd& y2,
          MatrixXd& x3, MatrixXd& 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 MatrixXd object "u", and assuming that the corresponding time vector has been stored in MatrixXd 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 GNUplot. 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(MatrixXd& x1, MatrixXd& y1,
            MatrixXd& x2, MatrixXd& 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");
```

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(MatrixXd& x,
               MatrixXd& 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 MatrixXd object "y", and assuming that the corresponding time vector has been stored in MatrixXd 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(MatrixXd& x,
          MatrixXd& y,
          MatrixXd& 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 MatrixXd 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(MatrixXd& x,
           MatrixXd& y,
           MatrixXd& 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 MatrixXd 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(MatrixXd& theta,
           MatrixXd& 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 MatrixXd 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(MatrixXd& theta,
           MatrixXd& r,
           MatrixXd& theta2,
           MatrixXd& r2,
           const string& title,
           char* legend=NULL,
           char* terminal=NULL,
           char* output=NULL);
```

For three polar curves.

```
void polar(MatrixXd& theta,
           MatrixXd& r,
           MatrixXd& theta2,
           MatrixXd& r2,
           MatrixXd& theta3,
           MatrixXd& 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.

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

`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 1.10). 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      = MatrixXd OBJECT;
problem.phases(iphase).observations           = MatrixXd OBJECT;
problem.phases(iphase).residual_weights       = MatrixXd OBJECT
```

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  );
```

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 multipled 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, $[-\text{inf}, 1]$, $[1, \text{inf}]$ or $[-\text{inf}, \text{inf}]$.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 the derivatives required by the nonlinear programming algorithm as follows.

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 [8]. 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. It is assumed that the Jacobian of the constraint function $G(y)$ is divided into constant and variable terms as follows:

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

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.

If SNOPT is being used, then its internal algorithms for numerical differentiation implemented are employed.

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 [2]. 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, $\tilde{x}$ 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

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

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 MatrixXd 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 the parameter:

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

For example, to try the sequence 40, 50 and 80 nodes in phase `iphase`, then the following commandd specify that:

```
RowVectorXi my_vector << 40, 50, 80;
problem.phases(iphase).nodes = my_vector;
```

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 [5]. 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

$$\Delta N_{\max} = F N_i$$

   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_refinemer` 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 [2]. 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;
```

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 $\epsilon_\alpha \leq 2\bar{\epsilon}^{(i,m)}$, 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:

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

   where

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

   where 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 $0 \leq I_\alpha < M_1$, 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 constraints in each phase. The

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

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                  = RowVectorXi OBJECT
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.

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 = MatrixXd object;
problem.bounds.upper.times = MatrixXd object;
```

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

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

For example, consider the following code:

```
MatrixXd tlower << 10.0, 20.0, 30.0;
MatrixXd tupper << 15.0, 25.0, 35.0;
problem.bounds.lower.times =  tlower;
problem.bounds.upper.times =  tupper;
```

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 2.2.7. 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 **??** 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);
```

- `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);
```

- `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,
                      MatrixXd& controls,
                      MatrixXd& states,
                      MatrixXd& param,
                      MatrixXd& 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,
                          MatrixXd& pointx,
                          MatrixXd& 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 MatrixXd object of independent data points.
- `pointy`: is a MatrixXd 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,
                                   MatrixXd& pointx,
                                   MatrixXd& 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 MatrixXd object of independent data points.
- `pointy`: is a MatrixXd 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,
                               MatrixXd& pointx,
                               MatrixXd& 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 MatrixXd object of independent data points.
- `pointy`: is a MatrixXd 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,
                            MatrixXd& X,
                            MatrixXd& Y,
                            MatrixXd& 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 (MatrixXd object) of dimension `nxpoints` $\times$ 1.
- `Y`: is a vector (MatrixXd object) of dimension `nypoints` $\times$ 1.

- `Z`: is a matrix (MatrixXd object) of dimensions `nxpoints` × `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,
                                   MatrixXd& X,
                                   MatrixXd& Y,
                                   MatrixXd& 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 (MatrixXd object) of dimension `nxpoints` × 1.
- `Y`: is a vector (MatrixXd object) of dimension `nypoints` × 1.
- `Z`: is a matrix (MatrixXd object) of dimensions `nxpoints` × `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,
                             MatrixXd& X,
                             MatrixXd& Y,
                             MatrixXd& 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 (MatrixXd object) of dimension `nxpoints` $\times$ 1.
- `Y`: is a vector (MatrixXd object) of dimension `nypoints` $\times$ 1.
- `Z`: is a matrix (MatrixXd 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:

$$Q_l \leq \int_{t_0^{(i)}}^{t_f^{(i)}} g[x^{(i)}(t), u^{(i)}(t), p^{(i)}, t]dt \leq Q_u \tag{2.12}$$

Function `integrate` has the following prototype:

```
adouble integrate(
  adouble (*integrand)(adouble*,
                       adouble*,
                       adouble*,
                       adouble&,
                       adouble*,int),
  adouble* xad,
  int iphase )
```

- `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)
```

- `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 MatrixXd& A,
                const adouble* x,
                int nx,
                adouble* y);
```

This function multiplies a constant matrix stored in MatrixXd 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` $\times$ `nb`) matrix stored in column-major format column in `adouble` array `Apr`, by the (`nb` $\times$ `mb`) matrix stored in column-major format in `adouble` array `Bpr`. The result is stored in column-major format 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 `resample_trajectory` function

This function resamples a trajectory given new values of the time vector using natural cubic spline interpolation.

```
void resample_trajectory(MatrixXd& Y,
                         MatrixXd& t,
                         MatrixXd& Ydata,
                         MatrixXd& tdata )
```

- `Y` is, on output, a MatrixXd object with dimensions $n_y \times N$ with the interpolated values of the dependent variable.
- `t` is a MatrixXd 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 MatrixXd object of dimensions $n_y \times M$ with the data values of the dependent variable.
- `tdata` is a MatrixXd 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)`.

2.10.33 `linspace` function

This function generate a sequence of real values and returns it in a MatrixXd object of dimensions `1` × `N`.

```
MatrixXd linspace(double X1, double X2, long N);
```

- `X1` is the initial value of the sequence
- `X1` is the final value of the sequence
- `N` is the number of elements of the sequence.

2.10.34 `zeros` function

This function returns a MatrixXd object of dimensions `nrows` × `ncols`, having a zero value in all its elements.

```
MatrixXd zeros(long nrows, long ncols)
```

- `nrows` is the number of rows of the matrix
- `ncols` is the number of columns of the matrix

2.10.35 `ones` function

This function returns a MatrixXd object of dimensions `nrows` × `ncols`, having a value of 1 in all its elements.

```
MatrixXd ones(long nrows, long ncols)
```

- `nrows` is the number of rows of the matrix
- `ncols` is the number of columns of the matrix

2.10.36 `eye` function

This function returns a MatrixXd object of dimensions `n` × `n`, having a value of 1 in all its diagonal elements and zeros elsewhere, this is, an identity matrix.

```
MatrixXd eye(long n)
```

- `n` is the number of rows and columns of the matrix

2.10.37 `GaussianRandom` function

This function returns a MatrixXd object of dimensions `nrows` × `ncols` and each of its elements has a random value obeying a Gaussian distribution with a mean of 0 and a standard deviation of 1.

```
MatrixXd RandomGaussian(long nrows, long ncols)
```

- `nrows` is the number of rows of the matrix
- `ncols` is the number of columns of the matrix

2.10.38 Elementwise mathematical functions on `MatrixXd` objects

A number of functions have been implemented returning each a MatrixXd object of dimensions `nrows` × `ncols` with an element-wise evaluation of a number of mathematical functions. The generic function prototype is as follows:

```
MatrixXd <Function_Name>(MatrixXd& m)
```

- `m` is the input matrix, which is the argument of the mathematical function being used. Note that the output matrix has the same dimensions of the input matrix.

The following functions are implemented:

- `sin`: trigonometic sine function
- `cos`: trigonnometric cosine function
- `tan`: trigonometric tangent function
- `asin`: arc sine function function
- `acos`: arc cosine function
- `atan`: arc tangent function
- `sinh`: hyperbolic sine function
- `cosh`: hyperbolic cosine function
- `tanh`: hyperbolic tangent function
- `exp`: exponential function
- `log`: natural logarithm function
- `log10`: base-10 logarithm function
- `sqrt`: square root function

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

PSOPT will by default produce output information on the screen as it runs. *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 *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 decision 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 estimation 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:

`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

After modifying the `user.cxx` code, open a command prompt and cd the user example directory under the build tree and execute the `make` command:

```
$ cd psopt/build/examples/user
$ make
```

If no compilation errors occur, an executable named `user` should be created in the directory `psopt/PSOPT/examples/user`.

Acknowledgements

The author is very grateful to Philipp Waxweiler from DLR, Germany, for developing the CMake build process for $\mathcal{PSOPT}$ and for providing suggestions for improving the code, including the use of Eigen3.

The author is thankful to 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 by S. Bhatt, where the dynamic model of the International Space Station is described.

Finally, the author is indebted to all users who have provided feedback on $\mathcal{PSOPT}$ since its first release.

References

[1] D. A. Benson. *A Gauss Pseudospectral Transcription for Optimal Control*. PhD thesis, MIT, Department of Aeronautics and Astronautics, Cambridge, Mass., 2004.

[2] J. T. Betts. *Practical Methods for Optimal Control Using Nonlinear Programming*. SIAM, 2001.

[3] J. T. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. SIAM, 2010.

[4] R.L. Burden and J.D. Faires. *Numerical Analysis*. Thomson Brooks/Cole, 2005.

[5] C. Canuto, M.Y. Hussaini, A. Quarteroni, and T.A. Zang. *Spectral Methods: Fundamentals in Single Domains*. Springer-Verlag, Berlin, 2006.

[6] 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.

[7] C.G. Canuto, M.Y. Hussaini, A. Quarteroni A., and T.A. Zang. *Spectral Methods in Fluid Dynamics*. Springer-Verlag, 1988.

[8] 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.

[9] 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.

[10] F. Fahroo and I.M. Ross. Costate Estimation by a Legendre Pseudospectral Method. *Journal of Guidance, Control, and Dynamics*, 24:270–277, 2001.

[11] F. Fahroo and I.M. Ross. Costate Estimation by a Legendre Pseudospectral Method. *Journal of Guidance, Control, and Dynamics*, 24:270–277, 2002.

[12] F. Fahroo and I.M. Ross. Direct Trajectory Optimization by a Chebyshev Pseudospectral Method. *Journal of Guidance Control and Dynamics*, 25, 2002.

[13] J. Hesthaven, S. Gottlieb, and D. Gottlieb. *Spectral Methods for Time-Dependent Problems*. Cambridge University Press, Cambridge, 2007.

[14] W. Kang and N. Bedrossian. Pseudospectral Optimal Control Theory Makes Debut Flight, Saves nasa $1m in Under Three Hours. *SIAM News*, 40, 2007.

[15] 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.

[16] 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.

[17] S. Marsili-Libelli, S. Guerrizio, and N. Checchi. Confidence regions of estimated parameters for ecological systems. *Ecological Modelling*, 165:127–146, 2003.

[18] J.A. Pietz. Pseudospectral Collocation Methods for the Direct transcription of Optimal Control Problems. Master's thesis, Rice University, Houston, Texas, 2003.

[19] 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.

[20] K. Schittkowski. *Numerical Data Fitting in Dynamical Systems*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002.

[21] Lloyd N. Trefethen. *Spectral Methods in MATLAB*. SIAM, Philadelphia, 2000.

[22] J. Vlassenbroeck and R. Van Doreen. A Chebyshev Technique for Solving Nonlinear Optimal Control Problems. *IEEE Transactions on Automatic Control*, 33:333–340, 1988.

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.

```
  GNU LESSER GENERAL PUBLIC LICENSE
       Version 2.1, February 1999

 Copyright (C) 1991, 1999 Free Software Foundation, Inc.
 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL.  It also counts
 as the successor of the GNU Library Public License, version 2, hence
 the version number 2.1.]

    Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
Licenses are intended to guarantee your freedom to share and change
free software--to make sure the software is free for all its users.

  This license, the Lesser General Public License, applies to some
specially designated software packages--typically libraries--of the
Free Software Foundation and other authors who decide to use it.  You
can use it too, but we suggest you first think carefully about whether
this license or the ordinary General Public License is the better
strategy to use in any particular case, based on the explanations below.

  When we speak of free software, we are referring to freedom of use,
not price.  Our General Public Licenses are designed to make sure that
you have the freedom to distribute copies of free software (and charge
for this service if you wish); that you receive source code or can get
it if you want it; that you can change the software and use pieces of
it in new free programs; and that you are informed that you can do
these things.

  To protect your rights, we need to make restrictions that forbid
distributors to deny you these rights or to ask you to surrender these
rights.  These restrictions translate to certain responsibilities for
you if you distribute copies of the library or if you modify it.

  For example, if you distribute copies of the library, whether gratis
or for a fee, you must give the recipients all the rights that we gave
you.  You must make sure that they, too, receive or can get the source
code.  If you link other code with the library, you must provide
```

```
complete object files to the recipients, so that they can relink them
with the library after making changes to the library and recompiling
it.  And you must show them these terms so they know their rights.

  We protect your rights with a two-step method: (1) we copyright the
library, and (2) we offer you this license, which gives you legal
permission to copy, distribute and/or modify the library.

  To protect each distributor, we want to make it very clear that
there is no warranty for the free library.  Also, if the library is
modified by someone else and passed on, the recipients should know
that what they have is not the original version, so that the original
author's reputation will not be affected by problems that might be
introduced by others.

  Finally, software patents pose a constant threat to the existence of
any free program.  We wish to make sure that a company cannot
effectively restrict the users of a free program by obtaining a
restrictive license from a patent holder.  Therefore, we insist that
any patent license obtained for a version of the library must be
consistent with the full freedom of use specified in this license.

  Most GNU software, including some libraries, is covered by the
ordinary GNU General Public License.  This license, the GNU Lesser
General Public License, applies to certain designated libraries, and
is quite different from the ordinary General Public License.  We use
this license for certain libraries in order to permit linking those
libraries into non-free programs.

  When a program is linked with a library, whether statically or using
a shared library, the combination of the two is legally speaking a
combined work, a derivative of the original library.  The ordinary
General Public License therefore permits such linking only if the
entire combination fits its criteria of freedom.  The Lesser General
Public License permits more lax criteria for linking other code with
the library.

  We call this license the "Lesser" General Public License because it
does Less to protect the user's freedom than the ordinary General
Public License.  It also provides other free software developers Less
of an advantage over competing non-free programs.  These disadvantages
are the reason we use the ordinary General Public License for many
libraries.  However, the Lesser license provides advantages in certain
special circumstances.

  For example, on rare occasions, there may be a special need to
encourage the widest possible use of a certain library, so that it becomes
a de-facto standard.  To achieve this, non-free programs must be
allowed to use the library.  A more frequent case is that a free
library does the same job as widely used non-free libraries.  In this
case, there is little to gain by limiting the free library to free
software only, so we use the Lesser General Public License.

  In other cases, permission to use a particular library in non-free
programs enables a greater number of people to use a large body of
free software.  For example, permission to use the GNU C Library in
non-free programs enables many more people to use the whole GNU
operating system, as well as its variant, the GNU/Linux operating
system.

  Although the Lesser General Public License is Less protective of the
```

```
users' freedom, it does ensure that the user of a program that is
linked with the Library has the freedom and the wherewithal to run
that program using a modified version of the Library.

  The precise terms and conditions for copying, distribution and
modification follow.  Pay close attention to the difference between a
"work based on the library" and a "work that uses the library".  The
former contains code derived from the library, whereas the latter must
be combined with the library in order to run.

  GNU LESSER GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License Agreement applies to any software library or other
program which contains a notice placed by the copyright holder or
other authorized party saying it may be distributed under the terms of
this Lesser General Public License (also called "this License").
Each licensee is addressed as "you".

  A "library" means a collection of software functions and/or data
prepared so as to be conveniently linked with application programs
(which use some of those functions and data) to form executables.

  The "Library", below, refers to any such software library or work
which has been distributed under these terms.  A "work based on the
Library" means either the Library or any derivative work under
copyright law: that is to say, a work containing the Library or a
portion of it, either verbatim or with modifications and/or translated
straightforwardly into another language.  (Hereinafter, translation is
included without limitation in the term "modification".)

  "Source code" for a work means the preferred form of the work for
making modifications to it.  For a library, complete source code means
all the source code for all modules it contains, plus any associated
interface definition files, plus the scripts used to control compilation
and installation of the library.

  Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running a program using the Library is not restricted, and output from
such a program is covered only if its contents constitute a work based
on the Library (independent of the use of the Library in a tool for
writing it).  Whether that is true depends on what the Library does
and what the program that uses the Library does.

  1. You may copy and distribute verbatim copies of the Library's
complete source code as you receive it, in any medium, provided that
you conspicuously and appropriately publish on each copy an
appropriate copyright notice and disclaimer of warranty; keep intact
all the notices that refer to this License and to the absence of any
warranty; and distribute a copy of this License along with the
Library.

  You may charge a fee for the physical act of transferring a copy,
and you may at your option offer warranty protection in exchange for a
fee.

  2. You may modify your copy or copies of the Library or any portion
of it, thus forming a work based on the Library, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:
```

```
    a) The modified work must itself be a software library.

    b) You must cause the files modified to carry prominent notices
    stating that you changed the files and the date of any change.

    c) You must cause the whole of the work to be licensed at no
    charge to all third parties under the terms of this License.

    d) If a facility in the modified Library refers to a function or a
    table of data to be supplied by an application program that uses
    the facility, other than as an argument passed when the facility
    is invoked, then you must make a good faith effort to ensure that,
    in the event an application does not supply such function or
    table, the facility still operates, and performs whatever part of
    its purpose remains meaningful.

    (For example, a function in a library to compute square roots has
    a purpose that is entirely well-defined independent of the
    application.  Therefore, Subsection 2d requires that any
    application-supplied function or table used by this function must
    be optional: if the application does not supply it, the square
    root function must still compute square roots.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Library,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Library, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote
it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Library.

In addition, mere aggregation of another work not based on the Library
with the Library (or with a work based on the Library) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may opt to apply the terms of the ordinary GNU General Public
License instead of this License to a given copy of the Library.  To do
this, you must alter all the notices that refer to this License, so
that they refer to the ordinary GNU General Public License, version 2,
instead of to this License.  (If a newer version than version 2 of the
ordinary GNU General Public License has appeared, then you can specify
that version instead if you wish.)  Do not make any other change in
these notices.

  Once this change is made in a given copy, it is irreversible for
that copy, so the ordinary GNU General Public License applies to all
subsequent copies and derivative works made from that copy.

  This option is useful when you wish to copy part of the code of
the Library into a program that is not a library.
```

```
  4. You may copy and distribute the Library (or a portion or
derivative of it, under Section 2) in object code or executable form
under the terms of Sections 1 and 2 above provided that you accompany
it with the complete corresponding machine-readable source code, which
must be distributed under the terms of Sections 1 and 2 above on a
medium customarily used for software interchange.

  If distribution of object code is made by offering access to copy
from a designated place, then offering equivalent access to copy the
source code from the same place satisfies the requirement to
distribute the source code, even though third parties are not
compelled to copy the source along with the object code.

  5. A program that contains no derivative of any portion of the
Library, but is designed to work with the Library by being compiled or
linked with it, is called a "work that uses the Library".  Such a
work, in isolation, is not a derivative work of the Library, and
therefore falls outside the scope of this License.

  However, linking a "work that uses the Library" with the Library
creates an executable that is a derivative of the Library (because it
contains portions of the Library), rather than a "work that uses the
library".  The executable is therefore covered by this License.
Section 6 states terms for distribution of such executables.

  When a "work that uses the Library" uses material from a header file
that is part of the Library, the object code for the work may be a
derivative work of the Library even though the source code is not.
Whether this is true is especially significant if the work can be
linked without the Library, or if the work is itself a library.  The
threshold for this to be true is not precisely defined by law.

  If such an object file uses only numerical parameters, data
structure layouts and accessors, and small macros and small inline
functions (ten lines or less in length), then the use of the object
file is unrestricted, regardless of whether it is legally a derivative
work.  (Executables containing this object code plus portions of the
Library will still fall under Section 6.)

  Otherwise, if the work is a derivative of the Library, you may
distribute the object code for the work under the terms of Section 6.
Any executables containing that work also fall under Section 6,
whether or not they are linked directly with the Library itself.

  6. As an exception to the Sections above, you may also combine or
link a "work that uses the Library" with the Library to produce a
work containing portions of the Library, and distribute that work
under terms of your choice, provided that the terms permit
modification of the work for the customer's own use and reverse
engineering for debugging such modifications.

  You must give prominent notice with each copy of the work that the
Library is used in it and that the Library and its use are covered by
this License.  You must supply a copy of this License.  If the work
during execution displays copyright notices, you must include the
copyright notice for the Library among them, as well as a reference
directing the user to the copy of this License.  Also, you must do one
of these things:

    a) Accompany the work with the complete corresponding
```

```
    machine-readable source code for the Library including whatever
    changes were used in the work (which must be distributed under
    Sections 1 and 2 above); and, if the work is an executable linked
    with the Library, with the complete machine-readable "work that
    uses the Library", as object code and/or source code, so that the
    user can modify the Library and then relink to produce a modified
    executable containing the modified Library.  (It is understood
    that the user who changes the contents of definitions files in the
    Library will not necessarily be able to recompile the application
    to use the modified definitions.)

    b) Use a suitable shared library mechanism for linking with the
    Library.  A suitable mechanism is one that (1) uses at run time a
    copy of the library already present on the user's computer system,
    rather than copying library functions into the executable, and (2)
    will operate properly with a modified version of the library, if
    the user installs one, as long as the modified version is
    interface-compatible with the version that the work was made with.

    c) Accompany the work with a written offer, valid for at
    least three years, to give the same user the materials
    specified in Subsection 6a, above, for a charge no more
    than the cost of performing this distribution.

    d) If distribution of the work is made by offering access to copy
    from a designated place, offer equivalent access to copy the above
    specified materials from the same place.

    e) Verify that the user has already received a copy of these
    materials or that you have already sent this user a copy.

  For an executable, the required form of the "work that uses the
Library" must include any data and utility programs needed for
reproducing the executable from it.  However, as a special exception,
the materials to be distributed need not include anything that is
normally distributed (in either source or binary form) with the major
components (compiler, kernel, and so on) of the operating system on
which the executable runs, unless that component itself accompanies
the executable.

  It may happen that this requirement contradicts the license
restrictions of other proprietary libraries that do not normally
accompany the operating system.  Such a contradiction means you cannot
use both them and the Library together in an executable that you
distribute.

  7. You may place library facilities that are a work based on the
Library side-by-side in a single library together with other library
facilities not covered by this License, and distribute such a combined
library, provided that the separate distribution of the work based on
the Library and of the other library facilities is otherwise
permitted, and provided that you do these two things:

    a) Accompany the combined library with a copy of the same work
    based on the Library, uncombined with any other library
    facilities.  This must be distributed under the terms of the
    Sections above.

    b) Give prominent notice with the combined library of the fact
    that part of it is a work based on the Library, and explaining
    where to find the accompanying uncombined form of the same work.
```

```
  8. You may not copy, modify, sublicense, link with, or distribute
the Library except as expressly provided under this License.  Any
attempt otherwise to copy, modify, sublicense, link with, or
distribute the Library is void, and will automatically terminate your
rights under this License.  However, parties who have received copies,
or rights, from you under this License will not have their licenses
terminated so long as such parties remain in full compliance.

  9. You are not required to accept this License, since you have not
signed it.  However, nothing else grants you permission to modify or
distribute the Library or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Library (or any work based on the
Library), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Library or works based on it.

  10. Each time you redistribute the Library (or any work based on the
Library), the recipient automatically receives a license from the
original licensor to copy, distribute, link with or modify the Library
subject to these terms and conditions.  You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties with
this License.

  11. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Library at all.  For example, if a patent
license would not permit royalty-free redistribution of the Library by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any
particular circumstance, the balance of the section is intended to apply,
and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system which is
implemented by public license practices.  Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  12. If the distribution and/or use of the Library is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Library under this License may add
an explicit geographical distribution limitation excluding those countries,
```

so that distribution is permitted only in or among countries not thus
excluded. In such case, this License incorporates the limitation as if
written in the body of this License.

  13. The Free Software Foundation may publish revised and/or new
versions of the Lesser General Public License from time to time.
Such new versions will be similar in spirit to the present version,
but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library
specifies a version number of this License which applies to it and
"any later version", you have the option of following the terms and
conditions either of that version or of any later version published by
the Free Software Foundation. If the Library does not specify a
license version number, you may choose any version ever published by
the Free Software Foundation.

  14. If you wish to incorporate parts of the Library into other free
programs whose distribution conditions are incompatible with these,
write to the author to ask for permission. For software which is
copyrighted by the Free Software Foundation, write to the Free
Software Foundation; we sometimes make exceptions for this. Our
decision will be guided by the two goals of preserving the free status
of all derivatives of our free software and of promoting the sharing
and reuse of software generally.

    NO WARRANTY

  15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO
WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW.
EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR
OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY
KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE
LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME
THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN
WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY
AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU
FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR
CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE
LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING
RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A
FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF
SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGES.

     END OF TERMS AND CONDITIONS