PSOPT optimal control solver

User Manual (version 02.09.2022)

Software Release 5.0.2

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

Image courtesy of https://images.nasa.gov/

Disclaimer

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

Licensing Agreement

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

```
  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
```

Contents

Chapter 1

Introduction to $\mathcal{PSOPT}$

1.1 What is $\mathcal{PSOPT}$

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

- Single or multiphase problems
- Continuous time nonlinear dynamics
- General endpoint constraints
- Nonlinear path constraints (equalities or inequalities) on states and/or control variables
- Integral constraints
- Interior point constraints
- Bounds on controls and state variables
- General cost function with Lagrange and Mayer terms.
- Free or fixed initial and final conditions
- Linear or nonlinear linkages between phases
- Fixed or free initial time
- Fixed or free final time

- 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 Why 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 PSOPT user's group

A user's group has been created with the purpose of enabling users to share their experiences with using 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 PSOPT and to ask for help. Membership is free and open. The 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 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 What is new in Release 5

1. PSOPT now builds using CMake, which means it can be compiled on a range of platforms where CMake is available and where its dependencies can be installed.
2. PSOPT now uses Eigen3 for its interface and for internal linear algebra manipulations. Eigen3 is a free state-of-the-art linear algebra suite written in C++. With the use of Eigen3, varioud old dependencies (DMatrix, LUSOL, SparseSuite) have been removed.
3. The SNOPT interface is working once again.
4. Miscellatious bug fixes and improvements to the interface.

1.4 External software libraries required by $\mathcal{PSOPT}$

$\mathcal{PSOPT}$ relies on three main software packages to perform a number of tasks. Note that some of these packages have their own dependencies.

1.4.1 IPOPT

IPOPT is an open source C++ package for large-scale nonlinear optimization, which uses an interior point method [42]. IPOPT is the default nonlinear programming algorithm used by $\mathcal{PSOPT}$. The IPOPT web page is:

https://github.com/coin-or/Ipopt

The current release of $\mathcal{PSOPT}$ has been tested with IPOPT version 3.12.12, but other versions of IPOPT are likely to work as well. The source code of version 3.12.12 of IPOPT can be downloaded from:

https://www.coin-or.org/download/source/Ipopt/

The list of dependencies of IPOPT include a sparse linear solver, such as MUMPS or one of the HSL libraries, and the matrix odering algorithm METIS. Please read the IPOPT installation instructions for further details. Some Linux distributions allow IPOPT and its depdendencies to be installed through a package manager. The installation instructions for IPOPT can be found at:

https://coin-or.github.io/Ipopt/INSTALL.html

1.4.2 ADOL-C

ADOL-C is a library for the automatic differentiation of C++ code. It allows to compute automatically the gradients and sparse Jacobians required by $\mathcal{PSOPT}$. At the heart of the ADOL-C library is the `adouble` data type, which can be mostly treated as a C++ `double`. A copy of ADOL-C is included with the distribution of $\mathcal{PSOPT}$. Some current Linux distrubutions, such as Ubuntu, make it very easy to install the ADOL-C library and headers using a package manager. When installing $\mathcal{PSOPT}$, if ADOL-C is not detected, CMake will attempt to download and compile the source for both COLPACK and ADOL-C.

Further information about ADOL-C can be found at its webpage:

https://github.com/coin-or/ADOL-C

The current release of $\mathcal{PSOPT}$ has been tested with ADOL-C version 2.6.3 and COLPACK version 1.0.10.

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

If installing from source code, the versions of ADOL-C and COLPACK that the author has used to develop the current version of $\mathcal{PSOPT}$ can be downloaded from the following links, respectively:

www.coin-or.org/download/source/ADOL-C/ADOL-C-2.6.3.tgz

https://github.com/CSCsw/ColPack/releases/tag/v1.0.10.

Note that it is necessary to configure ADOL-C to use enable sparse linear algebra through the –enable-sparse option, and to also install the graph colouring library COLPACK. Instructions for compiling and installing ADOL-C and ColPack from source code on a Linux computer are given below:

```
$ wget --continue www.coin-or.org/download/source/
   ADOL-C/ADOL-C-2.6.3.tgz
$ tar zxvf ADOL-C-2.6.3.tgz
$ cd ADOL-C-2.6.3
$ mkdir ./ThirdParty
$ cd ./ThirdParty
$ wget --continue http://archive.ubuntu.com/ubuntu/pool/
   universe/c/colpack/colpack_1.0.10.orig.tar.gz
$ tar zxvf colpack_1.0.10.orig.tar.gz
$ mv ColPack-1.0.10 ColPack
$ cd ColPack
$ ./configure --prefix=/usr/local
$ make
$ sudo make install
$ cd $HOME/ADOL-C-2.6.3
$ ./configure --prefix=/usr/local --enable-sparse
    --with-colpack=/usr/local
$ make
$ sudo make install
```

The installation procedure from source code on MacOS is similar from the instructions given above for Linux, with the difference that it is necessary to add `CXX=g++-10` and `CC=gcc-10` to the `configure` commands.

1.4.3 EIGEN3

Eigen is a community maintained lightweight, comprehensive and powerful linear algebra package for C++. It consists only of header files, so there is no need to link to it, only to include the relevant header files. Eigen's documentation and source code can be accessed through its webpage:

http://eigen.tuxfamily.org

Previous versions of $\mathcal{PSOPT}$ used a single matrix object of type `DMatrix` (a matrix with double precision floating point elements). The current release of $\mathcal{PSOPT}$ uses instead objects of type `MatrixXd` (a type of matrix object defined by Eigen3 with double precision floating point elements), and `RowVectorXi` (a type of vector defined by Eigen3 with integer elements).

In some respects, Eigen works in a similar way as the old library DMatrix, which users of previous versions of $\mathcal{PSOPT}$ may have become familiar with. A key difference is that Eigen uses zero based indexing for matrices and vectors, whereas DMatrix works with one-based indexing. For example with Eigen, the top left element of a matrix `A` is accessed through `A(0,0)`, whereas with DMatrix the same element is accessed through `A(1,1)`.

On MacOS, Eigen can be easily installed using Homebrew. If using a Debian-based Linux distribution (such as Ubuntu), the following command should work:

```
$ sudo apt install libeigen3-dev
```

As an alternative, the Eigen source code can be installed using CMake, as follows:

```
$ wget --continue  https://gitlab.com/libeigen/eigen/-/archive/
    3.3.7/eigen-3.3.7.tar.gz
$ tar zxvf eigen-3.3.7.tar.gz
$ cd eigen-3.3.7
$ mkdir build
$ cd build
$ cmake ..
$ sudo make install
```

1.5 Optional software that can be used by $\mathcal{PSOPT}$

1.5.1 SNOPT

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 on SNOPT. A trial license for SNOPT can be obtained from:

http://www.sbsi-sol-optimize.com.

1.5.2 GNUplot

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

http://www.gnuplot.info

On MacOS, GNUplot can be easily installed with Homebrew. Some current Linux distrubutions, such as Ubuntu, make it very easy to install GNUplot using a package manager. If it is desired to generate pdf files with the plots using C++ commands, the GNUplot needs to be compiled together with the PDFlib library. The instructions to do the installation from source on a Ubuntu-like Linux system are given below:

```
$ wget --continue
  https://fossies.org/linux/misc/old/PDFlib-Lite-7.0.5p3.tar.gz
$ tar zxvf PDFlib-Lite-7.0.5p3.tar.gz
$ cd PDFlib-Lite-7.0.5p3
$ ./configure
$ make; sudo make install
$ sudo ldconfig
$ cd $HOME/Downloads
$ wget --continue  https://sourceforge.net/projects/gnuplot/
   files/gnuplot/4.4.0/gnuplot-4.4.0.tar.gz/download
$ mv download gnuplot-4.4.0.tar.gz
$ tar zxvf gnuplot-4.4.0.tar.gz
$ sudo apt-get -y install libx11-dev libxt-dev libreadline6-dev
   libgd-dev
$ cd gnuplot-4.4.0
$ ./configure -with-readline=gnu -without-tutorial
$ make;sudo make install
```

1.6 About CMake

CMake is an open-source, cross-platform software tool used to build, test and package software. CMake is used to control the software compilation process using platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used with a variety of compilers on different platforms.

The *PSOPT* build process requires the use of CMake version 3.12 or later. Please note that ealier versions of CMake are not compatible with the *PSOPT* build process. CMake can often be installed with the operating system's package manager, or in the case of MacOS, with `homebrew`. Readers are advised to check that they have a suitable version number. If necessary, CMake may be built from source. The source code for CMake and instructions to build it can be found at:

https://cmake.org/download/

1.7 About pkg-config

`pkg-config` is a helper tool used when compiling applications and libraries. The *PSOPT* build process requires the use of `pkg-config`, which can be installed using the package manager, or in the case of MacOS, using Homebrew. In particular, the build process expects to see `pkg-config` configuration files for IPOPT, ColPack and ADOL-C. These configuration files are usually installed under `/usr/local/lib/pkgconfig` or `/usr/lib/pkgconfig`. If these configuration files are not created during the build process for the above libraries, they can be created manually and be placed at the correct folder. If the `pkg-config` configuration files are being created manually, the contents of these files on the authors' computer are provided below as examples. Please note that the paths that are given in these files depend on the actual location where the different libraries have been installed.

For IPOPT (filename: `ipopt.pc`):

```
prefix=/usr/local
#prefix=${pcfiledir}/../..
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include/coin-or

Name: IPOPT
Description: Interior Point Optimizer
URL: https://github.com/coin-or/Ipopt
Version: 3.13.2
Cflags: -I${includedir}
Libs: -L${libdir} -lipopt
```

```
Requires.private: coinhsl coinmumps
```

For ColPack (filename: `ColPack.pc`):

```

prefix=/usr/local
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include/ColPack

Name: ColPack
Version: 1.0.10
Description: Graph Coloring Library
Requires:
Libs: -L${libdir} -lColPack -Wl,-rpath,${libdir}
-L/usr/local/lib  -Wl,-rpath,/usr/local/lib
Cflags: -I${includedir}
```

For ADOL-C (filename: `adolc.pc`):

```

prefix=/usr/local
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include

Name: adolc
Version: 2.6.3
Description: Algorithmic Differentiation Library for C/C++
Requires:
Libs: -L${libdir} -ladolc -Wl,-rpath,${libdir}
 -L/usr/local/lib -lColPack -Wl,-rpath,/usr/local/lib
Cflags: -I${includedir}
```

For EIGEN3 (filename: `eigen3.pc`):

```

prefix=/usr/local
```

```
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include

Name: Eigen3
Version: 3.3.77
Description: Numerical linear algebra library for C++
Requires:
Libs:  -Wl,-rpath,${libdir}  -L/usr/local/lib
Cflags: -I${includedir} -std=c++11
```

For SNOPT (filename: `snopt7.pc`):

```
prefix=/usr/local
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include/snopt7

Name: SNOPT7
Version: 7
Description: SNOPT NONLINEAR PROGRAMMING LIBRARY
Requires:
Libs: -L${libdir} -lsnopt7_cpp -Wl,-rpath,${libdir}
 -L/usr/local/lib  -Wl,-rpath,/usr/local/lib
Cflags: -I${includedir}
```

1.8 Supported platforms

The current release of $\mathcal{PSOPT}$ has been successfully compiled under the following operating systems:

- Ubuntu Linux version 20.04.
- OpenSUSE Linux 15.4 Leap
- Arch Linux 2022.08.05
- Manjaro Linux 21.3.7
- MacOS Catalina, using g++/gcc 10.2.0 installed with Homebrew .

With the use of CMake, it is likely that PSOPT will install properly on a wide range of Unix-like systems. Users are encouraged to share their installation experience through the user's group, for the benefit of other users.

1.9 GitHub repository and home page

The downloadable *PSOPT* distribution is held in its GitHub page:

https://github.com/PSOPT/psopt

A web page is maintained to provide general information about *PSOPT* :

http://www.psopt.org

1.10 Installing and compiling *PSOPT*

For example, on Ubuntu 20.04, all dependencies plus GNUplot can be installed as follows:

```
$ sudo apt-get install git
$ sudo apt-get install cmake
$ sudo apt-get install gfortran
$ sudo apt-get install g++
$ sudo apt-get install libboost-dev
$ sudo apt-get install libboost-system-dev
$ sudo apt-get install coinor-libipopt-dev
$ sudo apt-get install libcolpack-dev
$ sudo apt-get install libadolc-dev
$ sudo apt-get install gnuplot
$ sudo apt-get install libeigen3-dev
$ sudo apt-get install libblas-dev
$ sudo apt-get install liblapack-dev
```

Please note that a runtime error related to the adolc library is currently reported when building PSOPT under Ubuntu 22.04. It is suggested to avoid this version of Ubuntu for the moment. This is probably due to a misconfiguration of the libadolc-dev package in Ubuntu 22.04.

If you use Debian 11.4.0, all dependencies plus GNUplot can simply be installed as follows:

```
$ su
$ apt-get install git
$ apt-get install cmake
$ apt-get install gfortran
$ apt-get install g++
$ apt-get install libboost-dev
$ apt-get install libboost-system-dev
$ apt-get install coinor-libipopt-dev
$ apt-get install libcolpack-dev
$ apt-get install libadolc-dev
$ apt-get install gnuplot
$ apt-get install libeigen3-dev
$ apt-get install libblas-dev
$ apt-get install liblapack-dev
```

If you use OpenSuse Leap 15.4, all dependencies plus GNUplot can be installed as follows:

```
$ sudo zypper install git
$ sudo zypper install gnuplot
$ sudo zypper install libboost_system1_66_0-devel
$ sudo zypper install eigen3-devel
$ sudo zypper install ColPack-devel
$ sudo zypper install adolc-devel
$ sudo zypper install blas-devel
$ sudo zypper install lapack-devel
$ sudo zypper ar -f
    https://download.opensuse.org/repositories/science/15.4/science
$ sudo zypper install Ipopt-devel
$ sudo zypper install cmake
$ sudo zypper install gcc-c++
```

If you use Arch Linux 2022.08.05 or Manjaro 21.3.7, all dependencies plus GNUplot can be installed as follows:

```
$ sudo pacman -Syu
$ sudo pacman -S git base-devel
```

```
$ sudo pacman -S cmake
$ sudo pacman -S gnuplot
$ sudo pacman -S eigen
$ sudo pacman -S boost
$ sudo pacman -S blas
$ sudo pacman -S lapack
$ git clone https://aur.archlinux.org/yay.git
$ cd yay
$ makepkg -si;cd ..
$ yay -S coin-or-ipopt
$ yay -S colpack
$ yay -S adol-c
```

After installation of dependencies, a typical PSOPT build and installation on a suitable version of Linux (such as Ubuntu 20.04, Debian 11.4.0 or OpenSUSE Leap 15.4, Arch Linux 2022.08.05 or Manjaro 21.3.7), or other Unix-like operating system, follows the steps given below:

```
$ git clone https://github.com/PSOPT/psopt.git
$ cd psopt; mkdir build; cd build
$ cmake -DBUILD_EXAMPLES=ON ..
$ make
$ sudo make install
```

CMake will configure the system for compilation, and will install some of the dependencies. Subsequently, 'make' will compile the software and install it. On MacOS, it is necessary to specify the C++/C compilers to be the GNU g++/gcc version 10.2.0. This can be done by defining the CXX and CC variables passed to the cmake command, as follows:

```
$ cmake -DBUILD_EXAMPLES=ON -DCXX=g++-10 -DCC=gcc-10 ..
```

If SNOPT is being linked, then add `-DWITH_SNOPT_INTERFACE=ON` when invoking the `cmake` command. Finally, if debugging symbols are required, define `-DCMAKE_BUILD_TYPE=Debug` when calling the `cmake` command.

The installation process will create the following directory structure under the home folder.

```
psopt
-       - src
-       - examples
-       - doc
-       - include
-       - src
-       - examples
-       - doc
-       - build
```

If desired, the `build` folder can be placed outside the source tree. Note, however, that the examples that need to read data files assume that the build folder is placed under the psopt folder, something that can be easily corrected by adjusting the file location in the source code of those examples. The installation process will generate a number of exacutables within the various subdirectories under `build/examples`.

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

```
$ cd build/examples/launch
$ ./launch
```

Please note that the CMake build configuration expects IPOPT to be already installed on the system before $\mathcal{PSOPT}$ is built.

1.11 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

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

the event constraints:

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

the phase linkage constraints:

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

the bound constraints:

$$\begin{aligned}u_L^{(i)} \le u^i(t) \le u_U^{(i)}, \, t \in [t_0^{(i)}, t_f^{(i)}],\\ x_L^{(i)} \le x^i(t) \le x_U^{(i)}, \, t \in [t_0^{(i)}, t_f^{(i)}],\\ p_L^{(i)} \le p^{(i)} \le p_U^{(i)},\\ \underline{t}_0^{(i)} \le t_0^{(i)} \le \bar{t}_0^{(i)},\\ \underline{t}_f^{(i)} \le t_f^{(i)} \le \bar{t}_f^{(i)},\end{aligned}$$

and the following constraints:

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

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 [3].

1.12 Overview of the Legendre and Chebyshev pseudospectral methods

1.12.1 Introduction to pseudospectral optimal control

Pseudospectral methods were originally developed for the solution of partial differential equations and have become a widely applied computational tool in fluid dynamics [12, 11]. Moreover, over the last 15 years or so, pseudospectral techniques have emerged as important computational methods for solving optimal control problems [15, 16, 18, 33, 23]. 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 [39]. In cases where global collocation is unsuitable (for example, when the solution exhibits discontinuities), multi-domain pseudospectral techniques have been proposed, where the problem is divided into a number of subintervals and global collocation is performed along each subinterval [11].

Pseudospectral methods directly discretize the original optimal control problem to formulate a nonlinear programming problem, which is then solved numerically using a sparse nonlinear programming solver to find approximate local optimal solutions. Approximation theory and practice shows that pseudospectral methods are well suited for approximating smooth functions, integrations, and differentiations [10, 39], 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 [15]. Since then, authors such as Ross, Fahroo and co-workers have analysed, extended and applied the method. For instance, convergence analysis is presented in [24], while an extension of the method to multiphase problems is given in [33]. An application that has received publicity is the use of the Legendre pseudospectral method for generating real time trajectories for a NASA spacecraft maneouvre [23]. The Chebyshev pseudospectral method for optimal control problems was originally proposed in 1988 [41]. Fahroo and Ross proposed an alternative method for trajectory optimisation using Chebyshev polynomials [18].

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

1.13 Pseudospectral approximations

1.13.1 Interpolation and the Lagrange polynomial

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

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

This polynomial is given by:

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

where

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

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

1.13.2 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.

1.13.3 Legendre polynomials and numerical quadrature

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

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

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

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

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

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

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

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

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

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

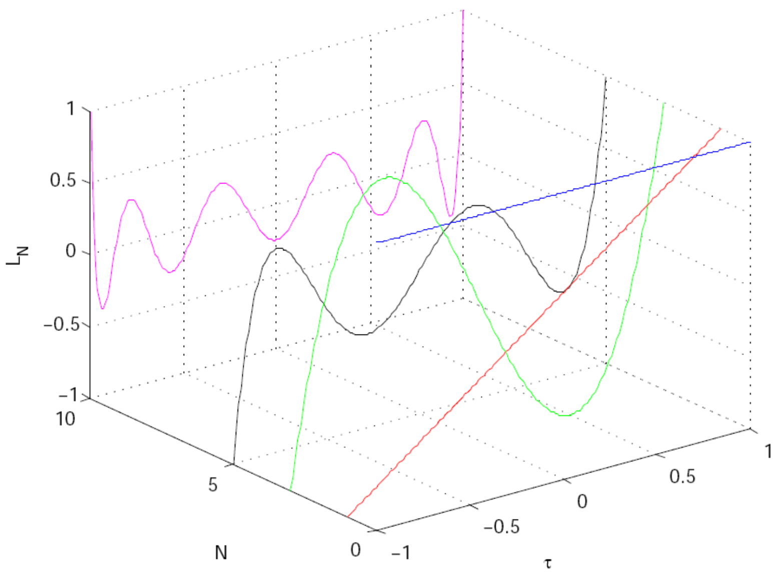

Figure 1.1: Illustration of the Legendre polynomials $L_N(\tau)$ for $N = 0, 1, 2, 4, 5, 10$.

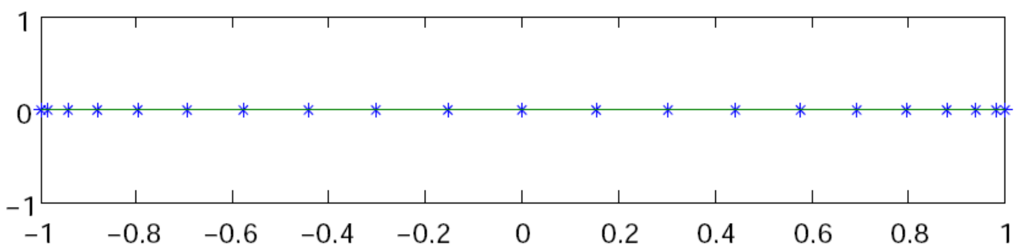

Figure 1.2: Illustration of the Legendre Gauss Lobatto (LGL) nodes for $N = 20$.

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

1.13.4 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 [21]) 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 [21]:

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

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

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

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

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

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

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

1.13.5 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.

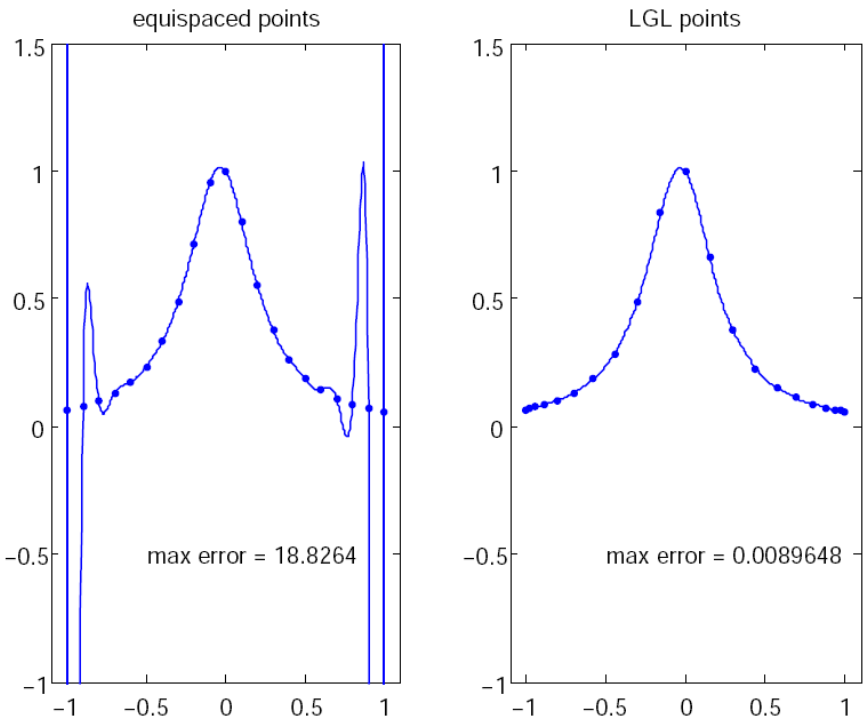

Figure 1.3: Illustration of polynomial interpolation over equispaced and LGL nodes

.

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*.

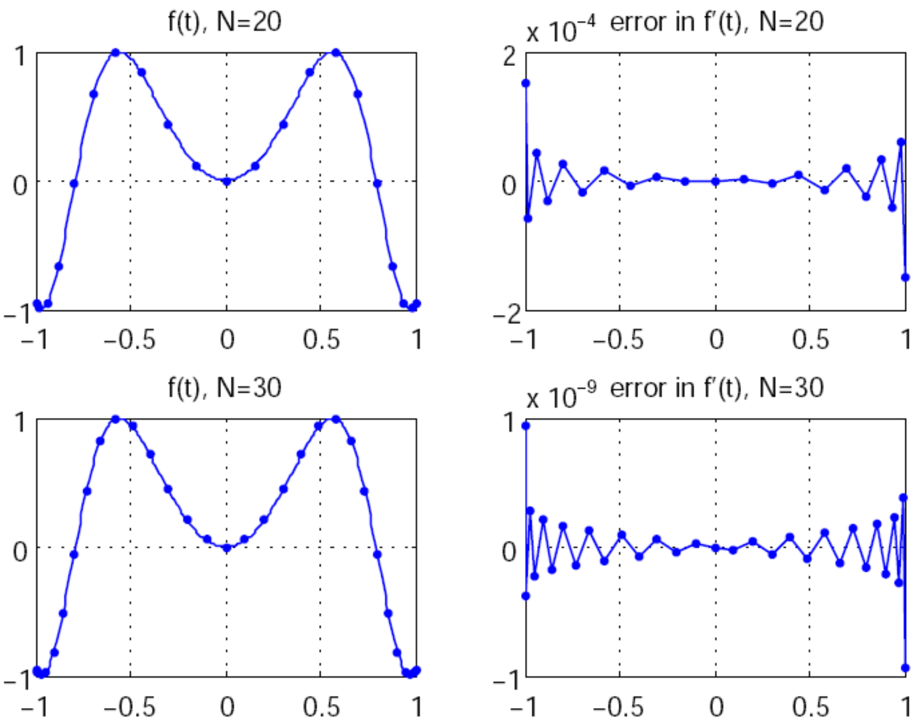

Figure 1.4: Legendre differentiation of $f(t) = \sin(5t^2)$ for $N = 20$ and $N = 30$.

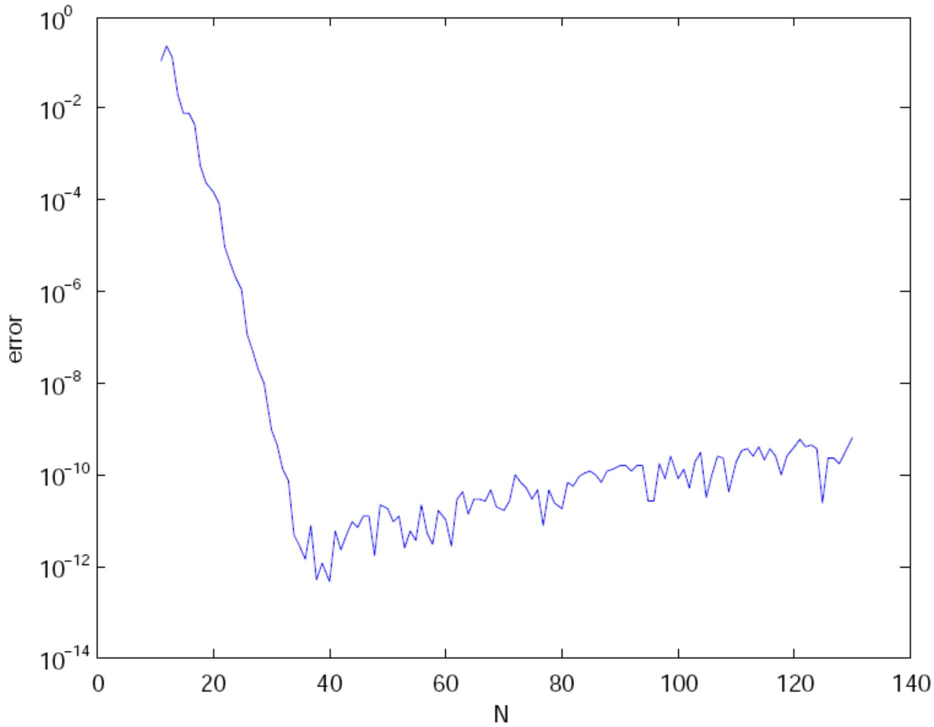

Figure 1.5: Maximum error in the Legendre differentiation of $f(t) = \sin(5t^2)$ as a function of N.

1.13.6 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

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

It should be noted that $\varphi_k(\tau_j) = 1$ if $k = j$ and $\varphi_k(\tau_j)$=0, if $k \neq j$, so that:

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

1.13.7 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

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

which is known as the differentiation matrix.

1.13.8 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:

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

1.13.9 Differentiation with reduced round-off errors

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

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

1.14 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:

$$\begin{aligned} \underline{t}_0 &\leq t_0 \leq \bar{t}_0, \\ \underline{t}_f &\leq t_f \leq \bar{t}_f, \\ t_f - t_0 &\geq 0, \end{aligned}$$

where

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

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:

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

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:

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

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

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

and

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

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

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

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

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

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

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

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

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

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

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

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

1.14.1 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 $\mathcal{P}_3$ [17]:

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

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 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 [17] 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 [30]:

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

where w_k are the weights given by (1.21), and $\tilde{\lambda}_k \in \Re^{n_x}$, $k = 0, \ldots, N$ are the KKTs multiplier associated with the collocation constraints $\mathrm{vec}(\zeta^N) = 0$. 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 1.14.1

1.14.2 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:

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

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:

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

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 $\mathcal{P}_1$.

1.15 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.

1.15.1 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:

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

1.15.2 Multi-phase case

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

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

the path constraints

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

the event constraints

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

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

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

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

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

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

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

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

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

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

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

1.15.3 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 [25], 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 [28]:

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

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 [35] 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 [35].

1.15.4 Remarks on parameter estimation

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

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

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

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

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

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

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

  The difference is:

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

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

1.16 Alternative local discretizations

Direct collocation methods that use local information to approximate the functions associated with an optimal control problem are well established [3]. Sometimes, it may be convenient for users to compare the performance and solutions obtained by means of the pseudospectral methods implemented in $\mathcal{PSOPT}$, with local discretization methods. Also, if a given problem cannot be solved by means of a pseudospectral discretization, the user has the option to try the local discretizations implemented in $\mathcal{PSOPT}$. The main impact of using a local discretization method as opposed to a pseudospectral discretization method, is that the resulting Jacobian and Hessian matrices needed by the NLP solver are more sparse with local methods, which facilitates the NLP solution. This becomes more noticeable as the number of grid points increases. The disadvantage of using a local method is that the spectral accuracy in the discretization of the differential constraints offered by pseudospectral methods is lost. Moreover, the accuracy of Gauss type integration employed in pseudospectral methods is also lost if pseudospectral grids are not used.

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

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

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

1.16.1 Trapezoidal method

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

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

where $\zeta(\tau_k) \in \Re^{n_x}$ is the vector of differential defect constraints at node τ_k, $k = 0, \ldots, N-1$, $h_k = \tau_{k+1} - \tau_k$, $f_k = f[(\tau_k), u(\tau_k), p, \tau_k]$, $f_{k+1} = f[x(\tau_{k+1}), u(\tau_{k+1}), p, \tau_{k+1}]$. This gives rise to $n_x N$ differential defect constraints. In this case, the decision vector for single phase problems is given by equation (1.39), so that it is the same as the one used in the Legendre and Chebyshev methods.

1.16.2 Hermite-Simpson method

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

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

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

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

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.

1.16.3 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.

1.16.4 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 1.14.1.

1.17 Limitations and known issues

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

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

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

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

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

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

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

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

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

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

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

Chapter 2

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

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

2.1 Interface data structures

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

- *Problem data structure:* This structure is used to specify problem information, including the number of phases and pointers to the relevant functions, as well as phase related information such as number of states, controls, parameters, number of grid points, bounds on variables (e.g. state bounds), and functions (e.g. path function bounds).
- *Algorithm data structure:* This is used to control the solution algorithm and to pass parameters to the NLP solver.
- *Solution data structure:* This is used to store the resulting variables of a $\mathcal{PSOPT}$ run.

2.2 Required functions

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

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:

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

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

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

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

```
adouble endpoint_cost(adouble* initial_states,
                      adouble* final_states,
                      adouble* parameters,
                      adouble& t0, adouble& tf,
                      adouble* xad, int iphase,
                      Workspace* workspace)
{
    adouble x1f = final_states[ 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 ];
```

```
    adoble 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);
```

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

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

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

2.3 Specifying a parameter estimation problem

To use the parameter estimation facilities implemented in $\mathcal{PSOPT}$ for problems where the observation function is defined, and where there is a set of observed data at given sampling points (see section 1.15). 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× `nsamples` matrix, `observations` is a `nobserved` × `nsamples` matrix, `residual_weights` is a 1× `nobserved` × `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, $[-\mathrm{inf}, 1]$, $[1, \mathrm{inf}]$ or $[-\mathrm{inf}, \mathrm{inf}]$.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 [13]. 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 [3]. Define the error in the differential equation as a function of time:

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

where $\tilde{x}$ is an interpolated value of the state vector given the grid point values of the state vector, $\dot{\tilde{x}}$ 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 [10]. Given a desired tolerance $\epsilon_{\max}$, this approximation is applied when the discretization error has been reduced for at least two iterations to find an extrapolated number of nodes which reduces the discretization error by a factor of 0.25.

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

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

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

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

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

Maximum increment factor, F:

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

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

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

Minimum number of extrapolation points:

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

Initial increment for the number of nodes, ΔN_0:

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

The mesh refinement algorithm is decribed below.

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

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

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

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

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

   (a) If $m < m_{\min}$ then $\Delta N_i = \Delta N_0$

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

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

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

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

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

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

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

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

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

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

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

Minimum increment factor, κ:

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

Maximum increment factor, ρ:

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

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

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

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 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 3.30 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` $\times$ `nypoints`. Each element `Z(i,j)` corresponds to the pair `( X(i), Y(j) )`

The function does not deal with sparse data. This function does not allow the use of automatic differentiation, so it should only be used with numerical differentiation.

2.10.22 `smooth_bilinear_interpolation` function

The function interpolates functions of two variables on a regular grid using the a smoothed bilinear interpolation method which allows the use of automatic differentiation. This is useful when the problem involves tabular data. The function prototype is as follows.

```
void smooth_bilinear_interpolation(adouble* z,
                                   adouble& x,
                                   adouble& y,
                                   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.

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` ×`n` square matrix stored columnwise in `adouble` array `a`. The result is returned in `adouble` array `ainv`, also using columnwise storage. The function prototype is as follows.

```
void inverse_ad(adouble* a,
                int n,
                adouble* ainv)
```

2.10.32 rk4_propagate function

This function may be used to generate an initial guess for the state trajectory by propagating the dynamics using 4th order Runge-Kutta integration. Note that no bounds are considered on states or controls and that any path constraints specified in function `dae()` are ignored. The user needs to specify a control trajectory and the corresponding time vector. The function prototype is as follows:

```
void rk4_propagate( void (*dae)(),
        MatrixXd& control_trajectory,
        MatrixXd& time_vector,
        MatrixXd& initial_state,
        MatrixXd& parameters,
        Prob & problem,
        int iphase,
        MatrixXd& state_trajectory);
```

- `dae` is a pointer to the problem's `dae` function;
- `control_trajectory` is a MatrixXd object of dimensions

  `problem.phases(iphase).ncontrols` × `M`

  with the initial guess for the controls.

- `time_vector` is a MatrixXd object of dimensions $1 \times$ `M` with the time instants that correspond to each element of `control_trajectory`.
- `initial_state` is a MatrixXd object of dimensions

  `problem.phases(iphase).nstates` $\times 1$

  with the value of the initial state vector.
- `parameters` is a MatrixXd object of dimensions

  `problem.phases(iphase).nparameters` $\times 1$

  with given values for the static parameters.
- `problem` is a `Prob` structure.
- `iphase` is the phase index.
- `state_trajectory` is a MatrixXd object with dimensions

  `problem.phases(iphase).nstates` $\times$ `M`

  which on output contains the result of the of the propagation. The values of the states correspond to the time vector `time_vector`.

2.10.33 `rkf_propagate` function

This function may be used to generate an initial guess for the state trajectory by propagating the dynamics using the Runge-Kutta-Fehlberg method with variable step size and relative local truncation error within a given tolerance. Note that no bounds are considered on states or controls and that any path constraints specified in function `dae()` are ignored. The user needs to specify a control trajectory and the corresponding time vector, as well as minimum and maximum values for the integration step size, and a tolerance. Note that the function throws an error if the minimum step size is violated. The function prototype is as follows:

```
void rkf_propagate( void (*dae)(),
        MatrixXd& control_trajectory,
        MatrixXd& time_vector,
        MatrixXd& initial_state,
        MatrixXd& parameters,
        double tolerance,
        double hmin,
        double hmax,
        Prob & problem,
        int iphase,
        MatrixXd& state_trajectory,
        MatrixXd& new_time_vector,
        MatrixXd& new_control_trajectory);
```

- `dae` is a pointer to the problem's `dae` function;
- `control_trajectory` is a MatrixXd object of dimensions

  `problem.phases(iphase).ncontrols` × `M`

  with the initial guess for the controls.
- `time_vector` is a MatrixXd object of dimensions 1 × `M` with the time instants that correspond to each element of `control_trajectory`.
- `initial_state` is a MatrixXd object of dimensions

  `problem.phases(iphase).nstates` × 1

  with the value of the initial state vector.
- `parameters` is a MatrixXd object of dimensions

  `problem.phases(iphase).nparameters` × 1

  with given values for the static parameters.
- `tolerance` is a positive value for the tolerance against which the maximum relative error in the state vector is compared.
- `hmin` is the minimum integration step size.
- `hmax` is the maximum integration step size.
- `problem` is a `Prob` structure.
- `iphase` is the phase index.
- `state_trajectory` is a MatrixXd object with dimensions

  `problem.phases(iphase).nstates` × `M`

  which on output contains the result of the of the propagation. The values of the states correspond to the time vector `new_time_vector`.
- `new_time_vector` is a MatrixXd object with dimensions 1 × `N`

  which on output contains the time values of the propagation.
- `new_control_trajectory` is a MatrixXd object with dimensions

  `problem.phases(iphase).nstates` × `N`

  which on output containts interpolated values of the control trajectory corresponding to the time vector `new_time_vector`. Linear interpolation is employed.

2.10.34 `resample_trajectory` function

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

```
void resample_trajectory(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)`.

2.10.35 `linspace` function

This function generate a sequence of real values and returns it in a MatrixXd object of dimensions `1` $\times$ `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.

2.10.36 `zeros` function

This function returns a MatrixXd object of dimensions `nrows` $\times$ `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

2.10.37 `ones` function

This function returns a MatrixXd object of dimensions `nrows` $\times$ `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

2.10.38 `eye` function

This function returns a MatrixXd object of dimensions `n` $\times$ `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

2.10.39 `GaussianRandom` function

This function returns a MatrixXd object of dimensions `nrows` $\times$ `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

2.10.40 Elementwise mathematical functions on `MatrixXd` objects

A number of functions have been implemented returning each a MatrixXd object of dimensions `nrows` $\times$ `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

$\mathcal{PSOPT}$ will by default produce output information on the screen as it runs. $\mathcal{PSOPT}$ will produce a short file with a summary of information named with the string provided in `algorithm.outfilelname`. This file contains the problem name, the total CPU time spent, the NLP solver used, the optimal value of the objective function, the values of the endpoint cost function and cost integrals, the initial and final time, the maximum discretization error, and the output string from the NLP solver.

Additionally, every time a $\mathcal{PSOPT}$ excutable is run, it will produce a file named `psopt_solution_$$$.txt` (`$$$` represents the characters of `problem.outfilename` which occur to the left of the file extension point "."). This file contains the problem name, time stamps, a summary of the algorithm options used, and results obtained, the final grid points, the final control variables, the final state variables, the final static parameter values. The file also contains a summary of all constraints functions associated with the NLP problem, including their final scaled value, bounds, and scaling factor used; a summary of the final NLP decision variables, including their final unscaled values, bounds and scaling factors used; and a summary of the mesh refinement process. An indication is given at the end of a constraint line, or decision variable line, if a scaled constraint function or scaled 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`.

Chapter 3

Examples of using $\mathcal{PSOPT}$

The following examples have been mostly selected from the literature such that their solutions can be compared with published results by consulting the references provided. Although source code is only shown here for a selection of the examples, the source code for all examples can be found in the $\mathcal{PSOPT}$ software distribution. Users are advised to study the source code of some of the examples before attempting to code their own problems.

3.1 Alp rider problem

Consider the following optimal control problem, which is known in the literature as the Alp rider problem [3]. It is known as Alp rider because the minimum of the objective function forces the states to ride the path constraint. Minimize the cost functional

$$J = \int_0^{20} (100(x_1^2 + x_2^2 + x_3^2 + x_4^2) + 0.01(u_1^2 + u_2^2))\mathrm{d}t \tag{3.1}$$

subject to the dynamic constraints

$$\begin{array}{rcl} \dot{x}_1 & = & -10x_1 + u_1 + u_2 \\ \dot{x}_2 & = & -2x_2 + u_1 + 2u_2 \\ \dot{x}_3 & = & -3x_3 + 5x_4 + u_1 - u_2 \\ \dot{x}_4 & = & 5x_3 - 3x_4 + u_1 + 3u_2 \end{array} \tag{3.2}$$

the path constraint

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 - 3p(t,3,12) - 3p(t,6,10) - 3p(t,10,16) - 8p(t,15,4) - 0.01 \leq 0 \tag{3.3}$$

where the exponential peaks are $p(t, a, b) = e^{-b(t-a)^2}$, and the boundary conditions are given by:

$$\begin{array}{rcl} x_1(0) & = & 2 \\ x_2(0) & = & 1 \\ x_3(0) & = & 2 \\ x_4(0) & = & 1 \\ x_1(20) & = & 2 \\ x_2(20) & = & 3 \\ x_3(20) & = & 1 \\ x_4(20) & = & -2 \end{array} \tag{3.4}$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and shown in Figures 3.1-3.4 and Figures 3.5-3.6, which contain the elements of the state and the control, respectively. Table 3.1 shows the mesh refinement history for this problem.

```
PSOPT results summary
=====================

Problem:  Alp rider problem
CPU time (seconds): 2.131610e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:11:15 2020

Optimal (unscaled) cost function value:  2.026363e+03
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  2.026363e+03
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.000000e+01
Phase 1 maximum relative local error: 3.703335e-03
NLP solver reports:  The problem has been solved!
```

3.2 Brachistochrone problem

Consider the following optimal control problem. Minimize the cost functional

$$J = t_f \tag{3.5}$$

Table 3.1: Mesh refinement statistics: Alp rider problem

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	$\epsilon_{\max}$	$\mathrm{CPU_a}$
1	LGL-CD	120	722	609	509	509	137	0	61080	2.211e-03	1.789e+00
2	LGL-CD	125	752	634	771	772	313	0	96500	3.414e-03	3.739e+00
3	LGL-CD	126	758	639	1233	1235	329	0	155610	3.703e-03	4.253e+00
$\mathrm{CPU_b}$	-	-	-	-	-	-	-	-	-	-	1.153e+01
-	-	-	-	-	2513	2516	779	0	313190	-	2.132e+01

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, $\epsilon_{\max}$ = maximum relative ODE error, $\mathrm{CPU_a}$ = CPU time in seconds spent by NLP algorithm, $\mathrm{CPU_b}$ = additional CPU time in seconds spent by PSOPT

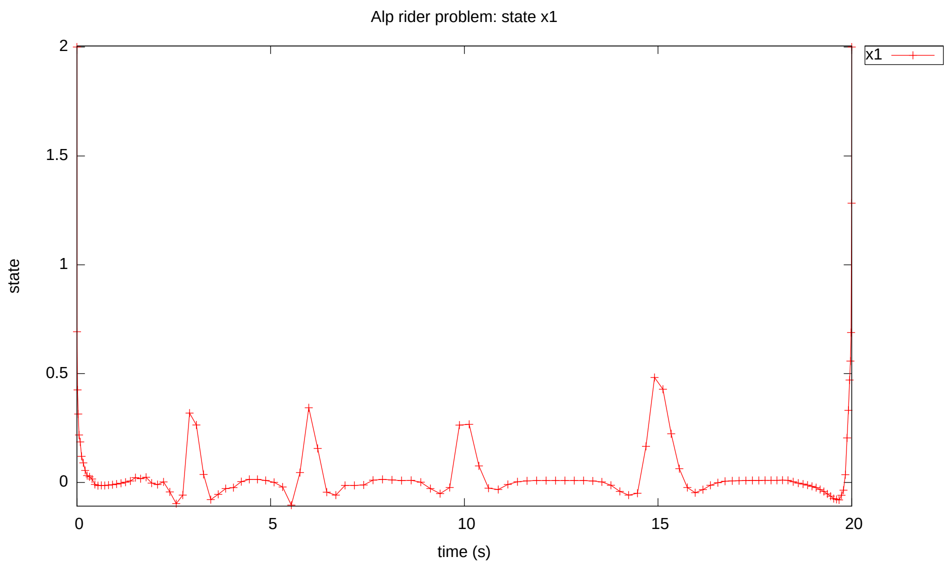

Figure 3.1: State $x_1(t)$ for the Alp rider problem

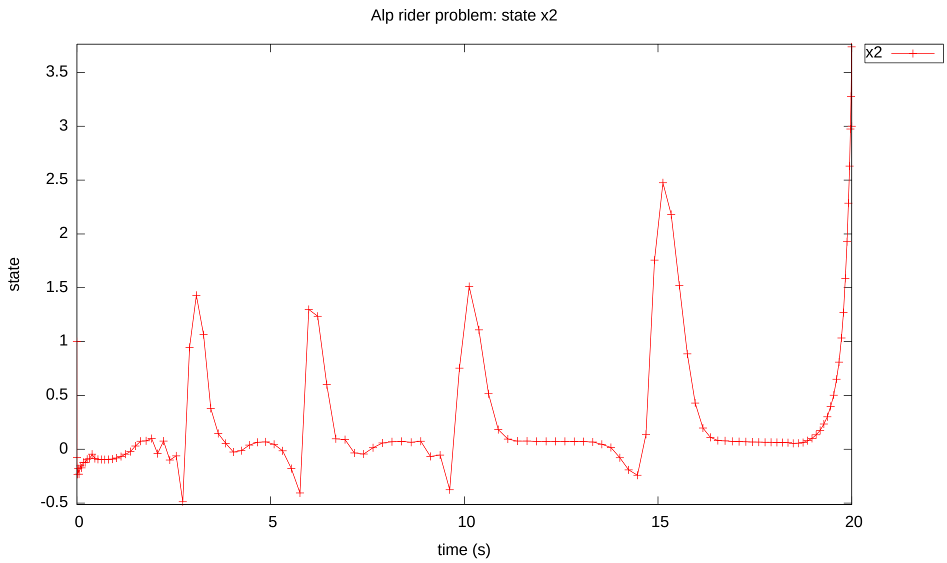

Figure 3.2: State $x_2(t)$ for the Alp rider problem

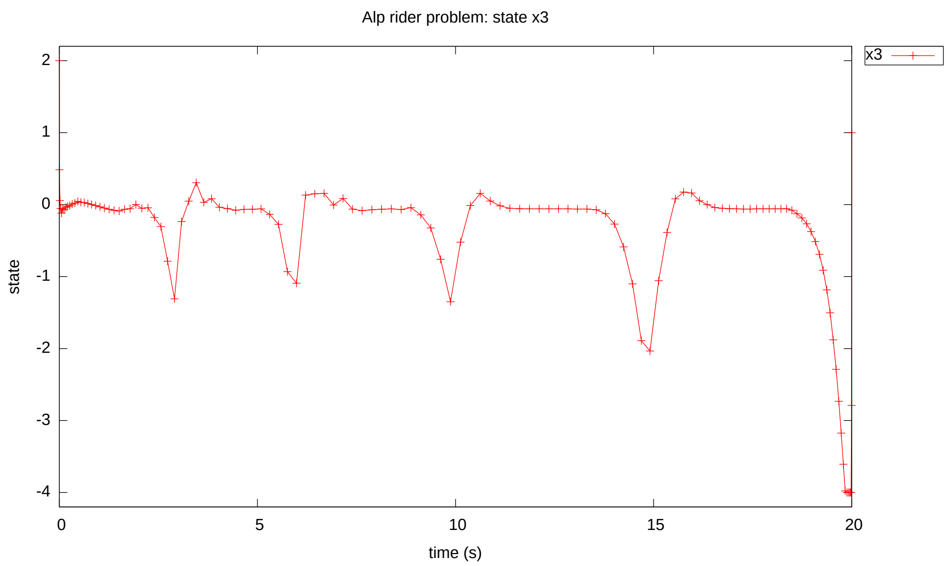

Figure 3.3: State $x_3(t)$ for the Alp rider problem

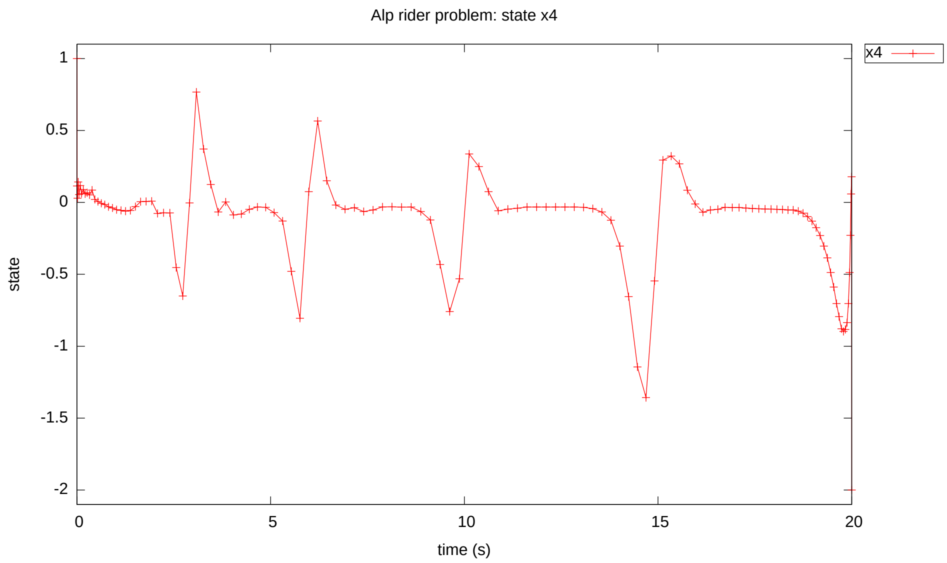

Figure 3.4: State $x_4(t)$ for the Alp rider problem

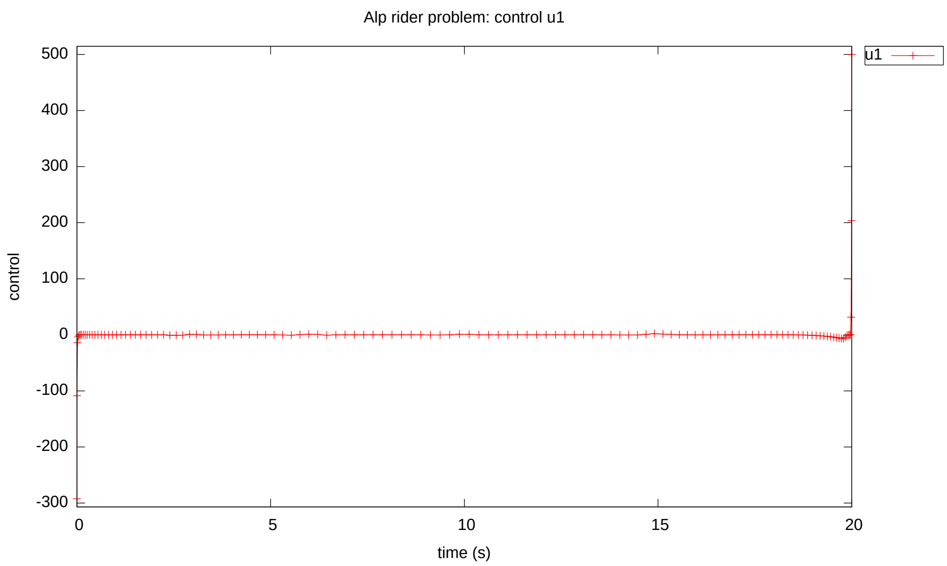

Figure 3.5: Control $u_1(t)$ for the Alp rider problem

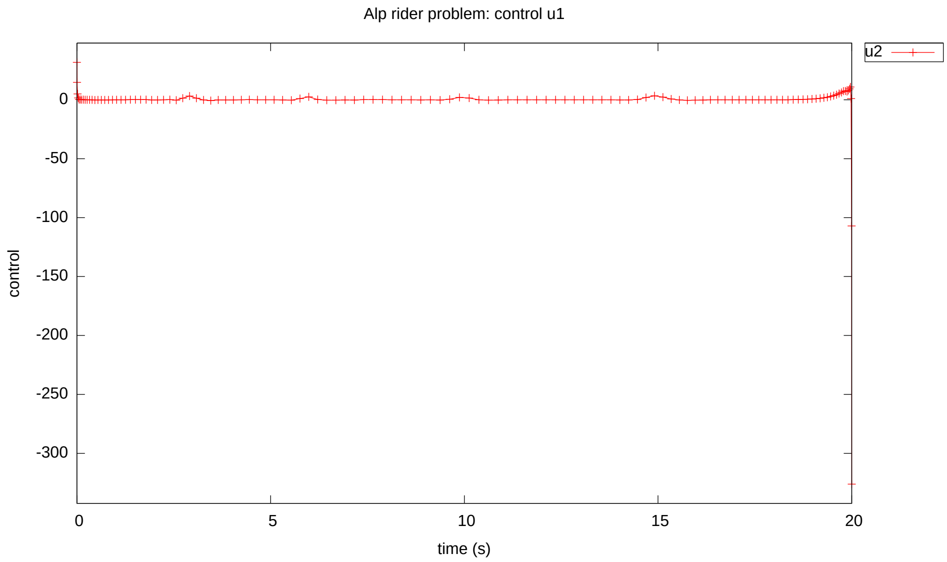

Figure 3.6: Control $u_2(t)$ for the Alp rider problem

subject to the dynamic constraints

$$\begin{aligned} \dot{x} &= v\sin(\theta) \\ \dot{y} &= v\cos(\theta) \\ \dot{v} &= g\cos(\theta) \end{aligned} \tag{3.6}$$

and the boundary conditions

$$\begin{aligned} x(0) &= 0 \\ y(0) &= 0 \\ v(0) &= 0 \\ x(t_f) &= 2 \\ y(t_f) &= 2 \end{aligned} \tag{3.7}$$

where $g = 9.8$. A version of this problem was originally formulated by Johann Bernoulli in 1696 and is referred to as the *Brachistochrone* problem. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
//////////////////        brac1.cxx       ///////////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example          ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:                  Brachistochrone problem   ////////////////
//////// Last modified:          04 January 2009           ////////////////
//////// Reference:              Bryson and Ho (1975)      ////////////////
//////// (See PSOPT handbook for full reference)           ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009         ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
```

```
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                    ///////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return tf;
}

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    return  0.0;
}

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble xdot, ydot, vdot;

   adouble x = states[ 0 ];
   adouble y = states[ 1 ];
   adouble v = states[ 2 ];

   adouble theta = controls[ 0 ];

   xdot = v*sin(theta);
   ydot = v*cos(theta);
   vdot = 9.8*cos(theta);

   derivatives[ 0 ] = xdot;
   derivatives[ 1 ] = ydot;
   derivatives[ 2 ] = vdot;
}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x0 = initial_states[ 0 ];
   adouble y0 = initial_states[ 1 ];
   adouble v0 = initial_states[ 2 ];
   adouble xf = final_states[ 0];
   adouble yf = final_states[ 1];

   e[ 0 ] = x0;
   e[ 1 ] = y0;
   e[ 2 ] = v0;
   e[ 3 ] = xf;
   e[ 4 ] = yf;
}

///////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}
```

```
////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////


int main(void)
{

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name        = "Brachistochrone Problem";

    problem.outfilename                = "brac1.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

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

    psopt_level1_setup(problem);


/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 3;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 5;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes                    << 40;

    psopt_level2_setup(problem, algorithm);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states    <<  0,  0,  0;
    problem.phases(1).bounds.upper.states    << 20, 20, 20;

    problem.phases(1).bounds.lower.controls  << 0.0;
    problem.phases(1).bounds.upper.controls  << 2*pi;

    problem.phases(1).bounds.lower.events    << 0,  0,  0,  2,  2;
    problem.phases(1).bounds.upper.events    << 0,  0,  0,  2,  2;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 0.0;
    problem.phases(1).bounds.upper.EndTime      = 10.0;


////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;


////////////////////////////////////////////////////////////////////////////
```

```
//////////////////  set scaling factors (optional) ///////////////////////
///////////////////////////////////////////////////////////////////////////

//    problem.phases(1).scale.controls    = 1.0*ones(1,1);
//    problem.phases(1).scale.states      = 1.0*ones(3,1);
//    problem.phases(1).scale.events      = 1.0*ones(5,1);
//    problem.phases(1).scale.defects     = 1.0*ones(3,1);
//    problem.phases(1).scale.time        = 1.0;

//    problem.scale.objective             = 1.0;

///////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ///////////////////////
///////////////////////////////////////////////////////////////////////////



    MatrixXd x0(3,20);

    x0.row(0) = linspace(0.0,1.0, 20);
    x0.row(1) = linspace(0.0,1.0, 20);
    x0.row(2) = linspace(0.0,1.0, 20);

    problem.phases(1).guess.controls       = ones(1,20);
    problem.phases(1).guess.states         = x0;
    problem.phases(1).guess.time           = linspace(0.0, 2.0, 20);

///////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  //////////////////////////////
///////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;
//    algorithm.hessian   = "exact";
    algorithm.collocation_method          = "Legendre";
//    algorithm.mesh_refinement             = "automatic";




///////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   /////////////////
///////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

    if (solution.error_flag) exit(0);

///////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
///////////////////////////////////////////////////////////////////////////

    MatrixXd x  = solution.get_states_in_phase(1);
    MatrixXd u  = solution.get_controls_in_phase(1);
    MatrixXd t  = solution.get_time_in_phase(1);
    MatrixXd H           = solution.get_dual_hamiltonian_in_phase(1);
    MatrixXd lambda      = solution.get_dual_costates_in_phase(1);

///////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ///////////////////////
///////////////////////////////////////////////////////////////////////////

    Save(x,"x.dat");
    Save(u,"u.dat");
    Save(t,"t.dat");
    Save(lambda,"p.dat");


///////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
///////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name + ": states", "time (s)", "states", "x y v");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u");

    plot(t,H,problem.name + ": Hamiltonian", "time (s)", "H", "H");
```

```
    plot(t,lambda,problem.name + ": costates", "time (s)", "lambda", "lambda_1 lambda_2 lambda_3");

    plot(t,x,problem.name + ": states", "time (s)", "states", "x y v",
                                "pdf", "brac1_states.pdf");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u",
                            "pdf", "brac1_control.pdf");

    plot(t,H,problem.name + ": Hamiltonian", "time (s)", "H", "H",
                                "pdf", "brac1_hamiltonian.pdf");

    plot(t,lambda,problem.name + ": costates", "time (s)", "lambda", "lambda_1 lambda_2 lambda_3",
                                "pdf", "brac1_costates.pdf");


}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and shown in Figures 3.7, 3.8, which contain the elements of the state, and the control respectively.

```
PSOPT results summary
=====================

Problem:  Brachistochrone Problem
CPU time (seconds): 9.862140e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:12:16 2020

Optimal (unscaled) cost function value:  8.247591e-01
Phase 1 endpoint cost function value: 8.247591e-01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 8.247591e-01
Phase 1 maximum relative local error: 2.776833e-07
NLP solver reports:  The problem has been solved!
```

3.3 Breakwell problem

Consider the following optimal control problem, which is known in the literature as the Breakwell problem [8]. The problem benefits from having an analytical solution, which is reported (with some errors) in the book by Bryson and Ho (1975). Minimize the cost

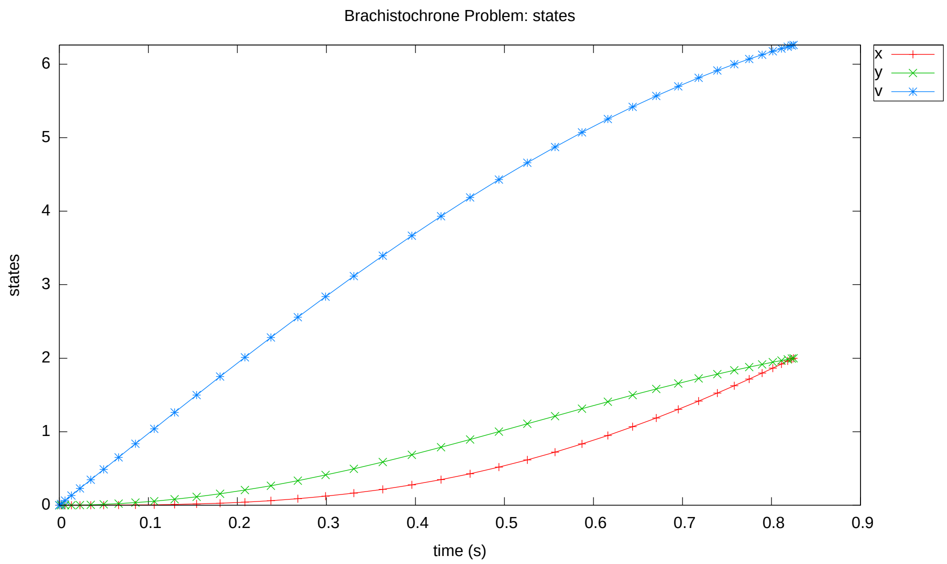

Figure 3.7: States for brachistochrone problem

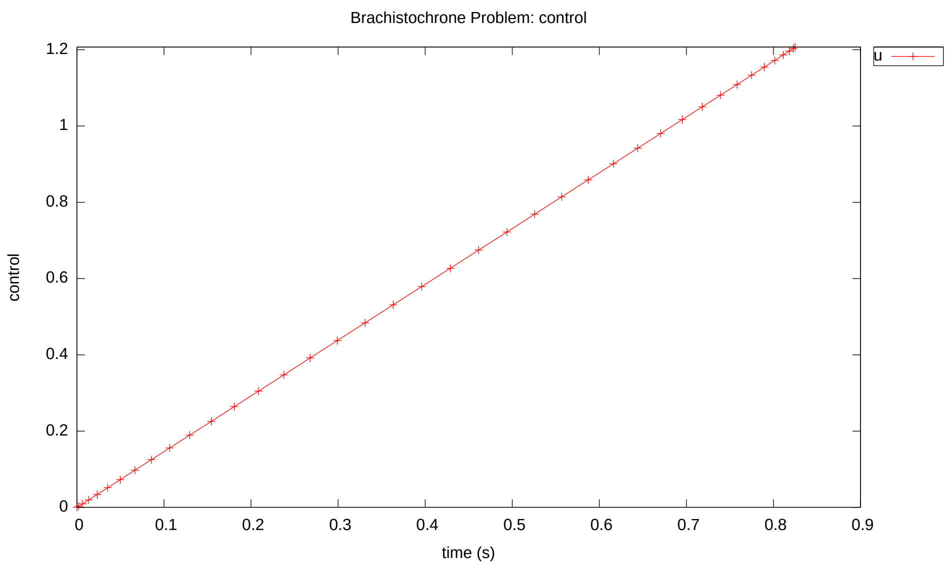

Figure 3.8: Control for brachistochrone problem

functional.

$$J = \int_0^{t_f} u(t)^2 \mathrm{d}t \tag{3.8}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= u \end{aligned} \tag{3.9}$$

the state dependent constraint

$$x(t) \leq l \tag{3.10}$$

where $l = 0.1$, $t_f = 1.$ and the boundary conditions

$$\begin{aligned} x(0) &= 0 \\ v(0) &= 1 \\ x(t_f) &= 0 \\ v(t_f) &= -1 \end{aligned} \tag{3.11}$$

The analytical solution of the problem (valid for $0 \leq l \leq 1/6$) is given by:

$$u(t) = \begin{cases} -\frac{2}{3l}(1 - \frac{t}{3l}), & 0 \leq t \leq 3l \\ 0, & 3l \leq t \leq 1 - 3l \\ -\frac{2}{3l}(1 - \frac{1-t}{3l}), & 1 - 3l \leq t \leq 1 \end{cases} \tag{3.12}$$

$$x(t) = \begin{cases} l\left(1 - (1 - \frac{t}{3l})^3\right), & 0 \leq t \leq 3l \\ l, & 3l \leq t \leq 1 - 3l \\ l\left(1 - (1 - \frac{1-t}{3l})^3\right), & 1 - 3l \leq t \leq 1 \end{cases} \tag{3.13}$$

$$v(t) = \begin{cases} \left(1 - \frac{t}{3l}\right)^2, & 0 \leq t \leq 3l \\ 0, & 3l \leq t \leq 1 - 3l \\ \left(1 - \frac{1-t}{3l}\right)^2, & 1 - 3l \leq t \leq 1 \end{cases} \tag{3.14}$$

$$\lambda_x(t) = \begin{cases} \frac{2}{9l^2}, & 0 \leq t \leq 3l \\ 0, & 3l \leq t \leq 1 - 3l \\ -\frac{2}{9l^2}, & 1 - 3l \leq t \leq 1 \end{cases} \tag{3.15}$$

$$\lambda_v(t) = \begin{cases} \frac{2}{3l}(1 - \frac{t}{3l}), & 0 \leq t \leq 3l \\ 0, & 3l \leq t \leq 1 - 3l \\ \frac{2}{3l}(1 - \frac{1-t}{3l}), & 1 - 3l \leq t \leq 1 \end{cases} \tag{3.16}$$

where $\lambda_x(t)$ and $\lambda_v(t)$ are the costates. The analytical optimal value of the objective function is $J = 4/(9l) = 4.4444444$.

The output from $\mathcal{PSOPT}$ is summarized in the following box and shown in Figures 3.9 and 3.10, which contain the elements of the state and the control, respectively, and Figure 3.11 which shows the costates. The figures include curves with the analytical solution for each variable, which is very close to the computed solution.

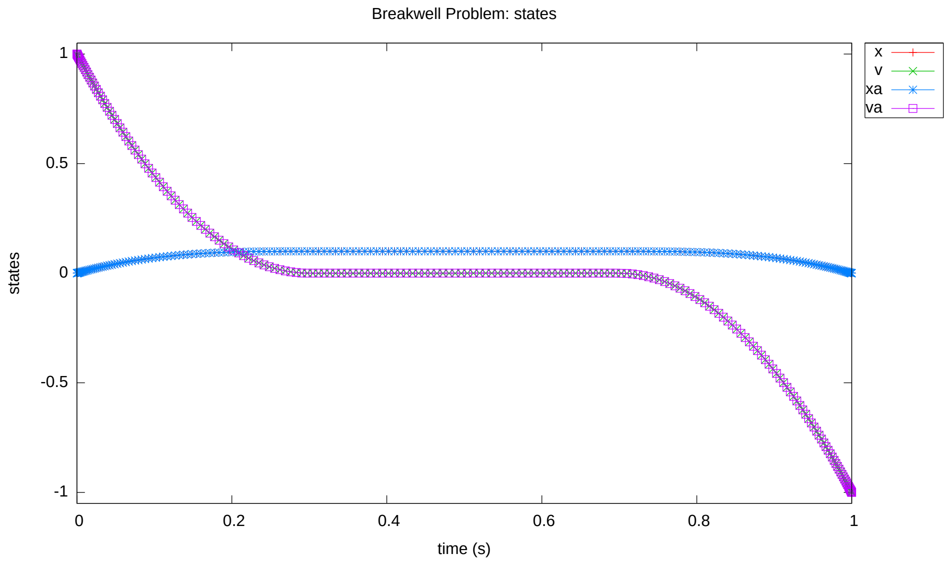

Figure 3.9: States for Breakwell problem

```
PSOPT results summary
=====================

Problem:  Breakwell Problem
CPU time (seconds): 3.911303e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:12:39 2020

Optimal (unscaled) cost function value:  4.444439e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  4.444439e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 1.488244e-06
NLP solver reports:  The problem has been solved!
```

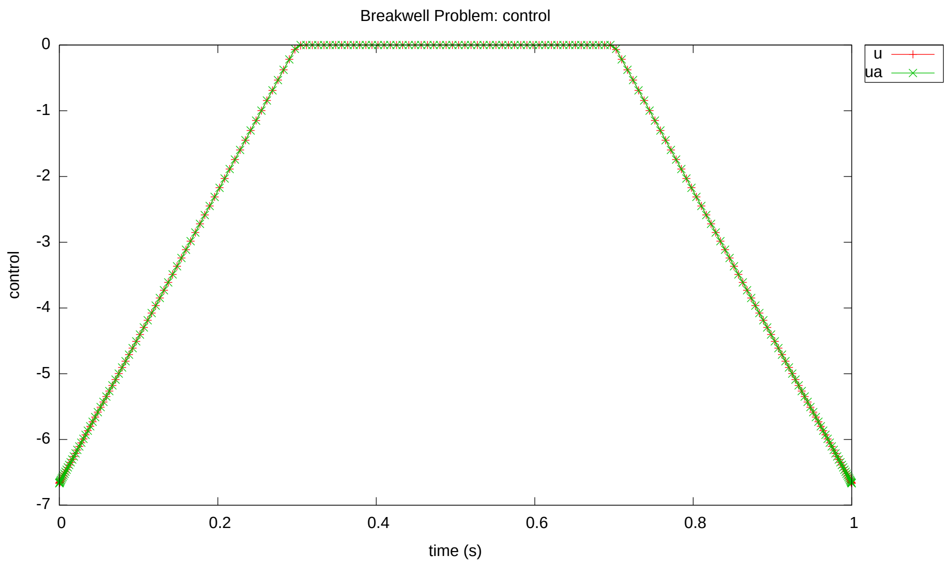

Figure 3.10: Control for Breakwell problem

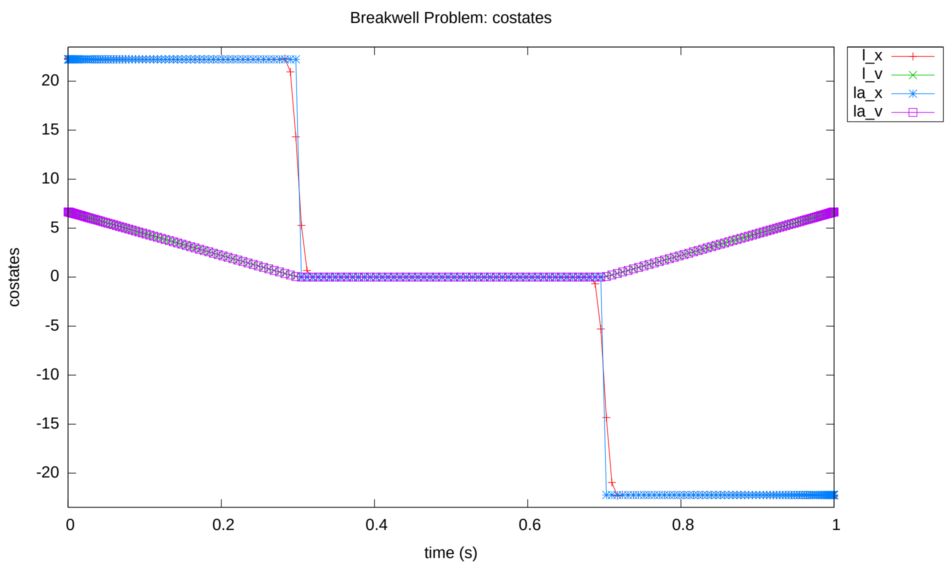

Figure 3.11: Costates for Breakwell problem

3.4 Bryson-Denham problem

Consider the following optimal control problem, which is known in the literature as the Bryson-Denham problem [7]. Minimize the cost functional

$$J = x_3(t_f) \tag{3.17}$$

subject to the dynamic constraints

$$\begin{array}{rcl} \dot{x}_1 & = & x_2 \\ \dot{x}_2 & = & u \\ \dot{x}_3 & = & \frac{1}{2}u^2 \end{array} \tag{3.18}$$

the state bound

$$0 \leq x_1 \leq 1/9 \tag{3.19}$$

and the boundary conditions

$$\begin{array}{rcl} x_1(0) & = & 0 \\ x_2(0) & = & 1 \\ x_3(0) & = & 0 \\ x_1(t_f) & = & 0 \\ x_2(t_f) & = & -1 \end{array} \tag{3.20}$$

The output from $\mathcal{PSOPT}$ is summarized in the following box and shown in Figures 3.12 and 3.13, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Bryson-Denham Problem
CPU time (seconds): 1.425490e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:13:12 2020

Optimal (unscaled) cost function value:  3.999539e+00
Phase 1 endpoint cost function value: 3.999539e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 6.474081e-01
Phase 1 maximum relative local error: 9.527663e-06
NLP solver reports:  The problem has been solved!
```

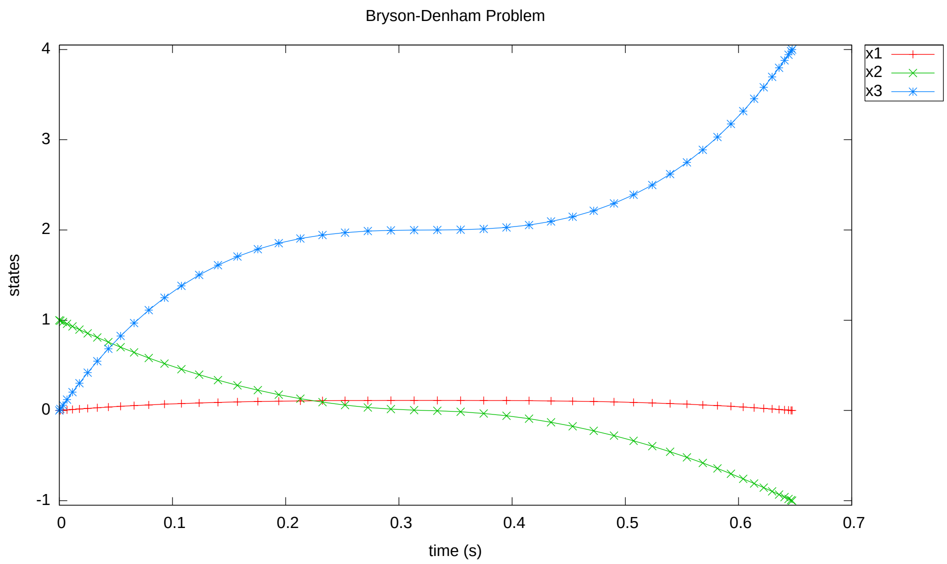

Figure 3.12: States for Bryson Denham problem

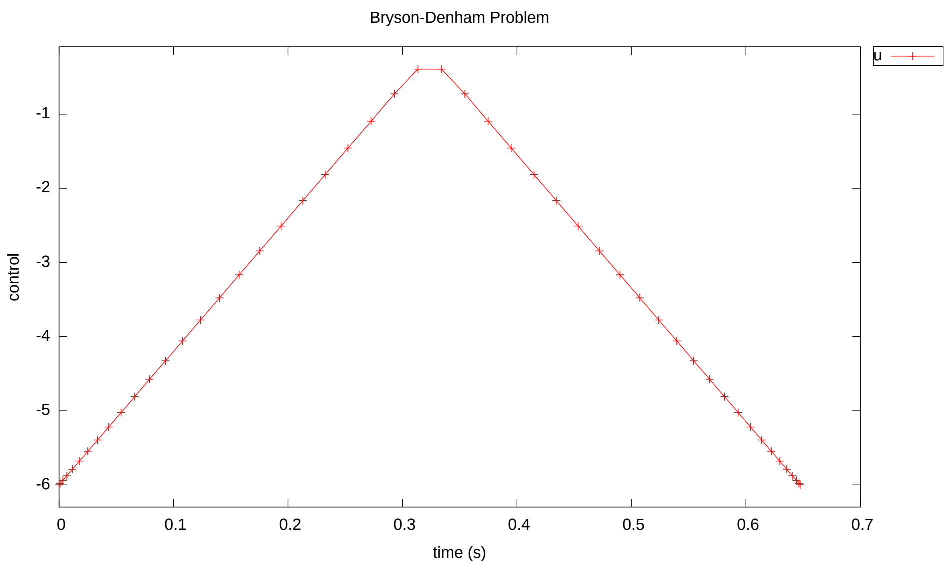

Figure 3.13: Control for Bryson Denham problem

3.5 Bryson's maximum range problem

Consider the following optimal control problem, which is known in the literature as the Bryson's maximum range problem [7]. Minimize the cost functional

$$J = -x(t_f) \tag{3.21}$$

subject to the dynamic constraints

$$\begin{array}{rcl} \dot{x} & = & vu_1 \\ \dot{y} & = & vu_2 \\ \dot{v} & = & a - gu_2 \end{array} \tag{3.22}$$

the path constraint

$$u_1^2 + u_2^2 = 1 \tag{3.23}$$

and the boundary conditions

$$\begin{array}{rcl} x(0) & = & 0 \\ y(0) & = & 0 \\ v(0) & = & 0 \\ y(t_f) & = & 0.1 \end{array} \tag{3.24}$$

where $t_f = 2$, $g = 1$ and $a = 0.5g$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
///////////////        bryson_max_range.cxx         /////////////////////
//////////////////////////////////////////////////////////////////////////
///////////////           PSOPT  Example            /////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:         Bryson maximum range problem      ////////////////
//////// Last modified: 05 January 2009                   ////////////////
//////// Reference:     Bryson and Ho (1975)              ////////////////
//////// (See PSOPT handbook for full reference)          ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////////
/////////////////// Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   adouble x = final_states[0];

   return (-x);
}

//////////////////////////////////////////////////////////////////////////
/////////////////// Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

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

```
{
    return  0.0;
}


//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble xdot, ydot, vdot;

   double g = 1.0;
   double a = 0.5*g;

   adouble x = states[ 0 ];
   adouble y = states[ 1 ];
   adouble v = states[ 2 ];

   adouble u1 = controls[ 0 ];
   adouble u2 = controls[ 1 ];

   xdot = v*u1;
   ydot = v*u2;
   vdot = a-g*u2;

   derivatives[ 0 ] = xdot;
   derivatives[ 1 ] = ydot;
   derivatives[ 2 ] = vdot;

   path[ 0 ] = (u1*u1) + (u2*u2);

}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

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

{
   adouble x0 = initial_states[ 0 ];
   adouble y0 = initial_states[ 1 ];
   adouble v0 = initial_states[ 2 ];
   adouble xf = final_states[ 0 ];
   adouble yf = final_states[ 1 ];

   e[ 0 ] = x0;
   e[ 1 ] = y0;
   e[ 2 ] = v0;
   e[ 3 ] = yf;

}



///////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}



////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

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

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name         = "Bryson Maximum Range Problem";
    problem.outfilename                = "brymr.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

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

    psopt_level1_setup(problem);

/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 3;
    problem.phases(1).ncontrols  = 2;
    problem.phases(1).nevents    = 4;
    problem.phases(1).npath      = 1;
    problem.phases(1).nodes         << 50;

    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare MatrixXd objects to store results /////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd x, u, t;
    MatrixXd lambda, H;

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////

    double xL = -10.0;
    double yL = -10.0;
    double vL = -10.0;
    double xU = 10.0;
    double yU = 10.0;
    double vU = 10.0;

    double u1L = -10.0;
    double u2L = -10.0;
    double u1U = 10.0;
    double u2U = 10.0;

    double x0 = 0.0;
    double y0 = 0.0;
    double v0 = 0.0;
    double yf = 0.1;


    problem.phases(1).bounds.lower.states(0) = xL;
    problem.phases(1).bounds.lower.states(1) = yL;
    problem.phases(1).bounds.lower.states(2) = vL;


    problem.phases(1).bounds.upper.states(0) = xU;
    problem.phases(1).bounds.upper.states(1) = yU;
    problem.phases(1).bounds.upper.states(2) = vU;


    problem.phases(1).bounds.lower.controls(0) = u1L;
    problem.phases(1).bounds.lower.controls(1) = u2L;
    problem.phases(1).bounds.upper.controls(0) = u1U;
    problem.phases(1).bounds.upper.controls(1) = u2U;

    problem.phases(1).bounds.lower.events(0) = x0;
    problem.phases(1).bounds.lower.events(1) = y0;
    problem.phases(1).bounds.lower.events(2) = v0;
    problem.phases(1).bounds.lower.events(3) = yf;
```

```
    problem.phases(1).bounds.upper.events(0) = x0;
    problem.phases(1).bounds.upper.events(1) = y0;
    problem.phases(1).bounds.upper.events(2) = v0;
    problem.phases(1).bounds.upper.events(3) = yf;

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



    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 2.0;
    problem.phases(1).bounds.upper.EndTime      = 2.0;



////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae             = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes                  = problem.phases(1).nodes(0);
    int ncontrols                       = problem.phases(1).ncontrols;
    int nstates                         = problem.phases(1).nstates;

    MatrixXd x_guess    =  zeros(nstates,nnodes);

    x_guess.row(0)  = x0*ones(1,nnodes);
    x_guess.row(1)  = y0*ones(1,nnodes);
    x_guess.row(2)  = v0*ones(1,nnodes);

    problem.phases(1).guess.controls       = zeros(ncontrols,nnodes);
    problem.phases(1).guess.states         = x_guess;
    problem.phases(1).guess.time           = linspace(0.0,2.0,nnodes);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-4;
    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
//    algorithm.mesh_refinement             = "automatic";
    algorithm.collocation_method          = "trapezoidal";
//    algorithm.defect_scaling = "jacobian-based";
    algorithm.ode_tolerance               = 1.e-6;



////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////


    x      = solution.get_states_in_phase(1);
    u      = solution.get_controls_in_phase(1);
    t      = solution.get_time_in_phase(1);
    lambda = solution.get_dual_costates_in_phase(1);
    H      = solution.get_dual_hamiltonian_in_phase(1);
```

```
////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ///////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(x, "x.dat");
    Save(u,"u.dat");
    Save(t,"t.dat");
    Save(lambda,"lambda.dat");
    Save(H,"H.dat");

////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name+": states", "time (s)", "states","x y v");

    plot(t,u,problem.name+": controls","time (s)", "controls", "u_1 u_2");

    plot(t,x,problem.name+": states", "time (s)", "states","x y v",
                             "pdf", "brymr_states.pdf");

    plot(t,u,problem.name+": controls","time (s)", "controls", "u_1 u_2",
                             "pdf", "brymr_controls.pdf");
}

////////////////////////////////////////////////////////////////////////////
//////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and shown in Figures 3.14 and 3.15, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Bryson Maximum Range Problem
CPU time (seconds): 5.738080e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:13:35 2020

Optimal (unscaled) cost function value:  -1.712313e+00
Phase 1 endpoint cost function value: -1.712313e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.000000e+00
Phase 1 maximum relative local error: 1.312361e-04
NLP solver reports:  The problem has been solved!
```

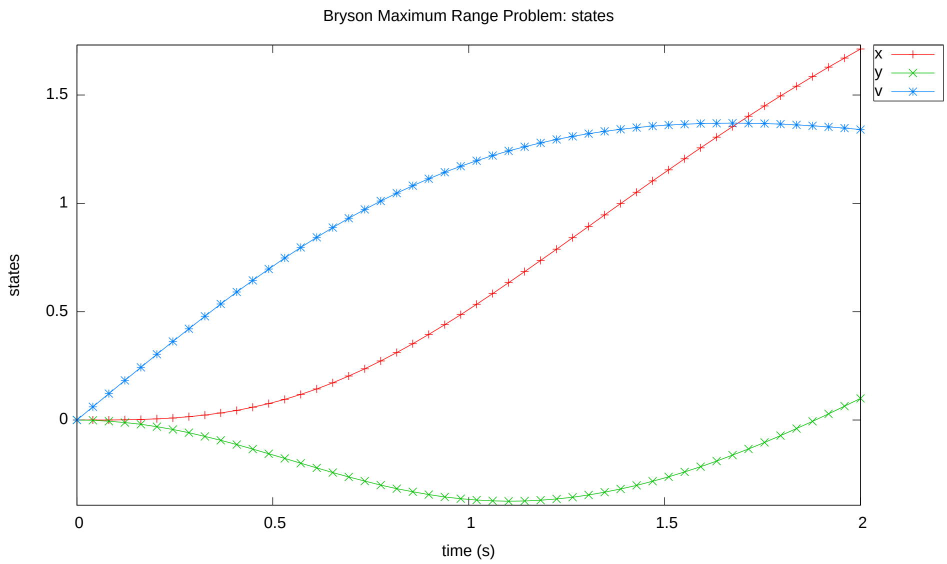

Figure 3.14: States for Bryson's maximum range problem

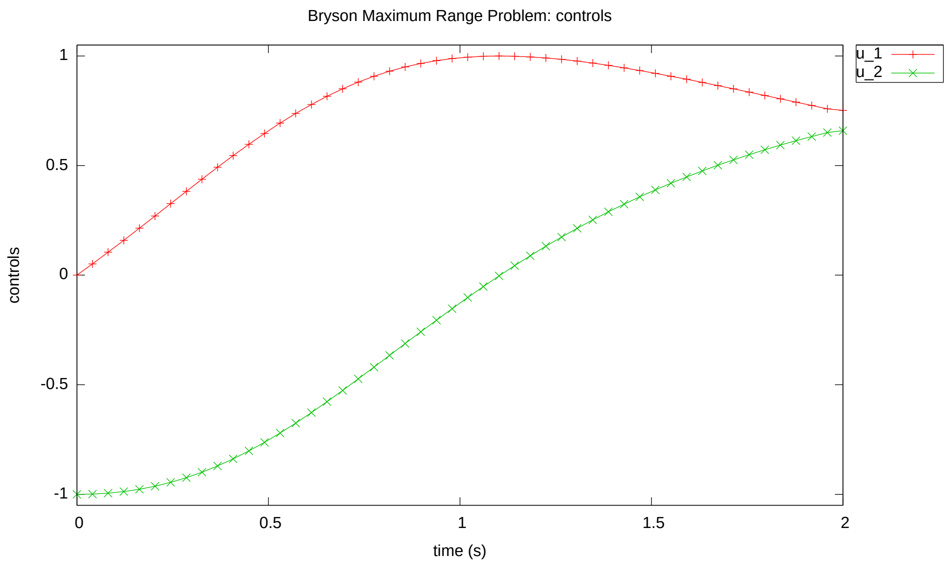

Figure 3.15: Controls for Bryson's maximum range problem

3.6 Catalyst mixing problem

Consider the following optimal control problem, which attempts to determine the optimal mixing policy of two catalysts along the length of a tubular plug flow reactor involving several reactions [36]. The catalyst mixing problem is a typical bang-singular-bang problem. Minimize the cost functional

$$J = -1 + x_1(t_f) + x_2(t_f) \tag{3.25}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= u(10x_2 - x_1) \\ \dot{x}_2 &= u(x_1 - 10x_2) - (1-u)x_2 \end{aligned} \tag{3.26}$$

the boundary conditions

$$\begin{aligned} x_1(0) &= 1 \\ x_2(0) &= 0 \\ x_1(t_f) &\leq 0.95 \end{aligned} \tag{3.27}$$

and the box constraints:

$$\begin{aligned} 0.9 &\leq x_1(t) \leq 1.0 \\ 0 &\leq x_2(t) \leq 0.1 \\ 0 &\leq u(t) \leq 1 \end{aligned} \tag{3.28}$$

where $t_f = 1$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.16 and 3.17, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Catalyst mixing roblem
CPU time (seconds): 1.369475e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:13:48 2020

Optimal (unscaled) cost function value:  -4.805320e-02
Phase 1 endpoint cost function value: -4.805320e-02
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 1.089793e-04
NLP solver reports:  The problem has been solved!
```

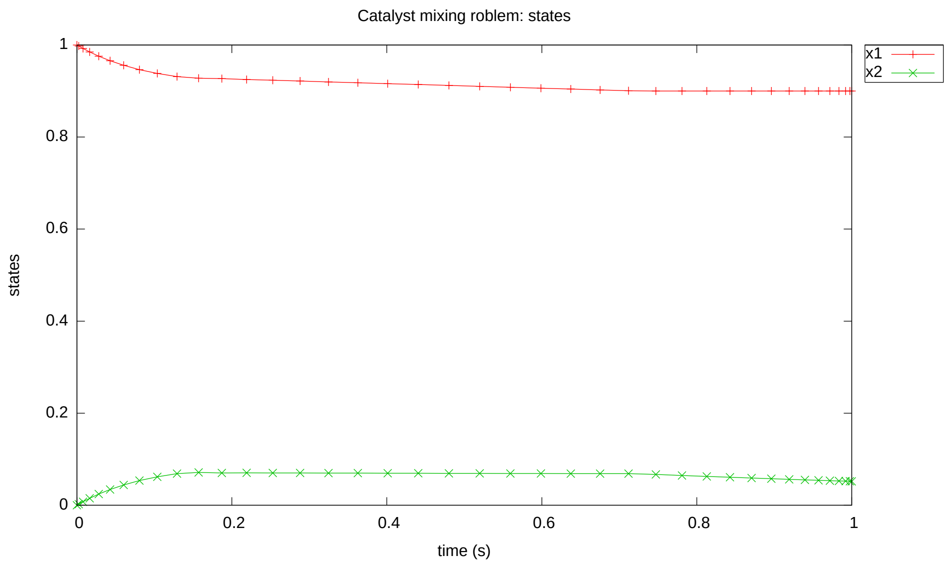

Figure 3.16: States for catalyist mixing problem

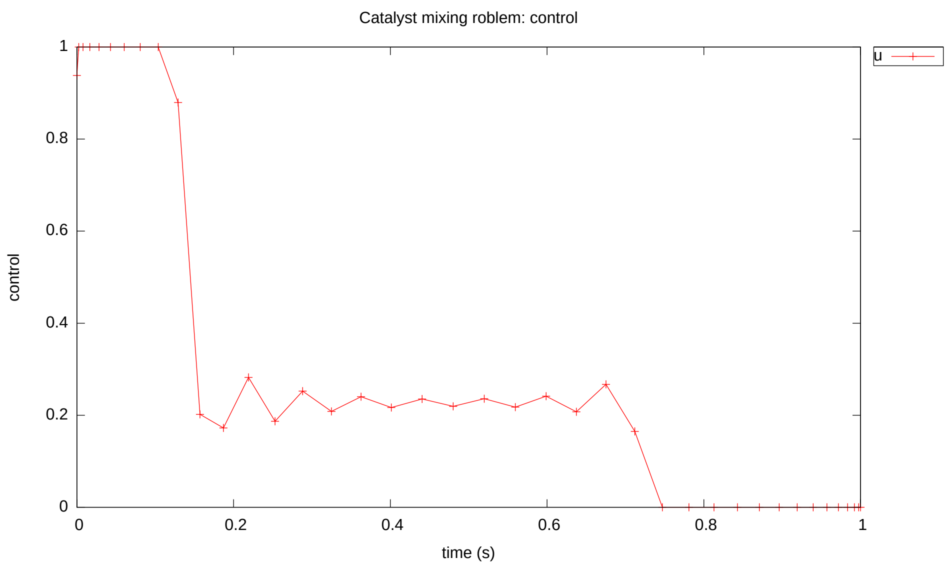

Figure 3.17: Control for catalyst mixing problem

3.7 Catalytic cracking of gas oil

Consider the following optimization problem, which involves finding optimal static parameters subject to dynamic constraints [14]. Minimize

$$J = \sum_{i=1}^{21} (y_1(t_i) - y_{m,1}(i))^2 + (y_2(t_i) - y_{m,2}(i))^2 \tag{3.29}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{y}_1 &= -(\theta_1 + \theta_3) y_1^2 \\ \dot{y}_2 &= \theta_1 y_1^2 - \theta_2 y_2 \end{aligned} \tag{3.30}$$

the parameter constraint

$$\begin{aligned} \theta_1 &\geq 0 \\ \theta_2 &\geq 0 \\ \theta_3 &\geq 0 \end{aligned} \tag{3.31}$$

Note that, given the nature of the problem, the parameter estimation facilities of $\mathcal{PSOPT}$ are used in this example. In this case, the observations function is simple:

$$g(x(t), u(t), p, t) = [y_1 \;\; y_2]^T$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below. The code includes the values of the measurement vectors $y_{m,1}$, and $y_{m,2}$, as well as the vector of sampling instants $\theta_i, i = 1, \ldots, 21$.

```
//////////////////////////////////////////////////////////////////////////
////////////////////        cracking.cxx       ///////////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////           PSOPT  Example            ///////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:  Catalytic Cracking of Gas Oil            ////////////////
//////// Last modified: 15 January 2009                   ////////////////
//////// Reference:     User's guide for DIRCOL     ////////////////
//////// (See PSOPT handbook for full reference)           ///////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////////
/////////////////////  Define the observation function //////////
//////////////////////////////////////////////////////////////////////////



void  observation_function( adouble* observations,
                            adouble* states, adouble* controls,
                            adouble* parameters, adouble& time, int k,
                            adouble* xad, int iphase, Workspace* workspace)
{

      observations[ 0 ] = states[ 0 ];
```

```
      observations[ 1 ] = states[ 1 ];
}



//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

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

   adouble y1 = states[0];
   adouble y2 = states[1];

   adouble theta1 = parameters[ 0 ];
   adouble theta2 = parameters[ 1 ];
   adouble theta3 = parameters[ 2 ];

   derivatives[0] = -(theta1 + theta3)*y1*y1;
   derivatives[1] =  theta1*y1*y1 - theta2*y2;

}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   // No events

}


///////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{

   MatrixXd y1meas(1,21), y2meas(1,21), tmeas(1,21);
   // Measured values of y1
   y1meas <<     1.0,0.8105,0.6208,0.5258,0.4345,0.3903,0.3342,0.3034, \
                  0.2735,0.2405,0.2283,0.2071,0.1669,0.153,0.1339,0.1265, \
                  0.12,0.099,0.087,0.077,0.069;
   // Measured values of y2
   y2meas <<  0.0,0.2,0.2886,0.301,0.3215,0.3123,0.2716,0.2551,0.2258, \
                 0.1959,0.1789,0.1457,0.1198,0.0909,0.0719,0.0561,0.046, \
                0.028,0.019,0.014,0.01;
   // Sampling instants
   tmeas  <<  0.0,0.025,0.05,0.075,0.1,0.125,0.15,0.175,0.2,0.225,0.25, \
              0.3,0.35,0.4,0.45,0.5,0.55,0.65,0.75,0.85,0.95;



////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

```
    problem.name          = "Catalytic cracking of gas oil";
    problem.outfilename                 = "cracking.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

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

    psopt_level1_setup(problem);


/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup ////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 2;
    problem.phases(1).ncontrols  = 0;
    problem.phases(1).nevents    = 0;
    problem.phases(1).npath      = 0;
    problem.phases(1).nparameters         = 3;
    problem.phases(1).nodes          << 80;
    problem.phases(1).nobserved   = 2;
    problem.phases(1).nsamples    = 21;

    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////////
////////////  Enter estimation information                     ////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd observations(2, 21);

    observations << y1meas, y2meas;

    problem.phases(1).observation_nodes      = tmeas;
    problem.phases(1).observations           = observations;
    problem.phases(1).residual_weights       = ones(2,21);


////////////////////////////////////////////////////////////////////////////
//////////////////  Declare DMatrix objects to store results //////////////
////////////////////////////////////////////////////////////////////////////

    DMatrix x, p, t;

////////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states(0) =  0.0;
    problem.phases(1).bounds.lower.states(1) =  0.0;


    problem.phases(1).bounds.upper.states(0) =  2.0;
    problem.phases(1).bounds.upper.states(1) =  2.0;


    problem.phases(1).bounds.lower.parameters(0) = 0.0;
    problem.phases(1).bounds.lower.parameters(1) = 0.0;
    problem.phases(1).bounds.lower.parameters(2) = 0.0;
    problem.phases(1).bounds.upper.parameters(0) = 20.0;
    problem.phases(1).bounds.upper.parameters(1) = 20.0;
    problem.phases(1).bounds.upper.parameters(2) = 20.0;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 0.95;
    problem.phases(1).bounds.upper.EndTime      = 0.95;

////////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;
    problem.observation_function = & observation_function;
```

```
////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess //////////////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd state_guess(2, 40);

    state_guess.row(0) =  linspace(1.0,0.069, 40);
    state_guess.row(1) =  linspace(0.30,0.01,  40);


    problem.phases(1).guess.states         = state_guess;
    problem.phases(1).guess.time           = linspace(0.0, 0.95, 40);
    problem.phases(1).guess.parameters     = zeros(3,1);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////

    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.collocation_method          = "Hermite-Simpson";
    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;
//    algorithm.jac_sparsity_ratio          = 0.52;

////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////

    x = solution.get_states_in_phase(1);
    t = solution.get_time_in_phase(1);
    p = solution.get_parameters_in_phase(1);


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(x,"x.dat");
    Save(t,"t.dat");
    cout << "\n Estimated parameters\n" << p << endl;
//    Print(p,"Estimated parameters");


////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name, "time (s)", "states", "y1 y2");
    plot(t,x,problem.name, "time (s)", "states", "y1 y2",
                           "pdf", "cracking_states.pdf");

}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and shown in Figure 3.18, which shows the states of the system. The optimal parameters found were:

$$\begin{aligned} \theta_1 &= 11.40825702 \\ \theta_2 &= 8.123367918 \\ \theta_3 &= 1.668727477 \end{aligned} \tag{3.32}$$

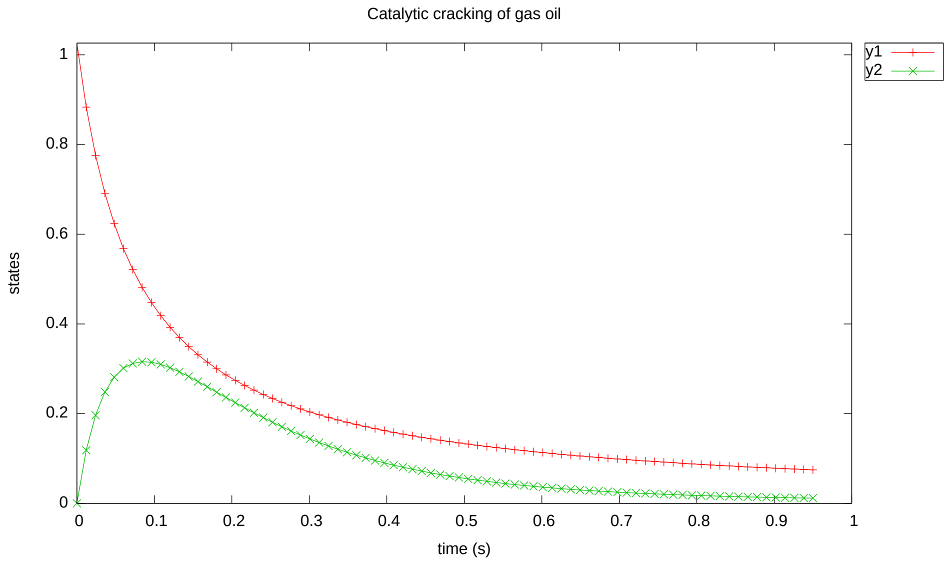

Figure 3.18: States for catalytic cracking of gas oil problem

```
PSOPT results summary
=====================

Problem:  Catalytic cracking of gas oil
CPU time (seconds): 6.313210e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:18:00 2020

Optimal (unscaled) cost function value:  4.319519e-03
Phase 1 endpoint cost function value: 4.319519e-03
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 9.500000e-01
Phase 1 maximum relative local error: 4.414787e-04
NLP solver reports:  The problem has been solved!
```

3.8 Coulomb friction

Consider the following optimal control problem, which consists of a system that exhibits Coulomb friction [27]. Minimize the cost:

$$J = t_f \tag{3.33}$$

subject to the dynamic constraints

$$\begin{array}{rcl} \ddot{q}_1 & = & (-(k_1 - k_2)q_1 + k_2 q_2 - \mu \text{sign}(\dot{q}_1) + u_1)/m_1 \\ \ddot{q}_2 & = & (k_2 q_1 - k_2 q_2 - \mu \text{sign}(\dot{q}_2) + u_2)/m_2 \end{array} \tag{3.34}$$

and the boundary conditions

$$\begin{array}{rcl} q_1(0) & = & 0 \\ \dot{q}_1(0) & = & -1 \\ q_2(0) & = & 0 \\ \dot{q}_2(0) & = & -2 \\ q_1(t_f) & = & 1 \\ \dot{q}_1(t_f) & = & 0 \\ q_2(t_f) & = & 2 \\ \dot{q}_2(t_f) = 0 & & \end{array} \tag{3.35}$$

where $k_1 = 0.95$, k_2=0.85, $\mu = 1.0$, m_1=1.1, m_2=1.2.

The output from $\mathcal{PSOPT}$ summarised in the box below and shown in Figures 3.19 and 3.20, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Coulomb friction problem
CPU time (seconds): 3.060764e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:17:48 2020

Optimal (unscaled) cost function value:  2.104992e+00
Phase 1 endpoint cost function value: 2.104992e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.104992e+00
Phase 1 maximum relative local error: 7.858415e-03
NLP solver reports:  The problem has been solved!
```

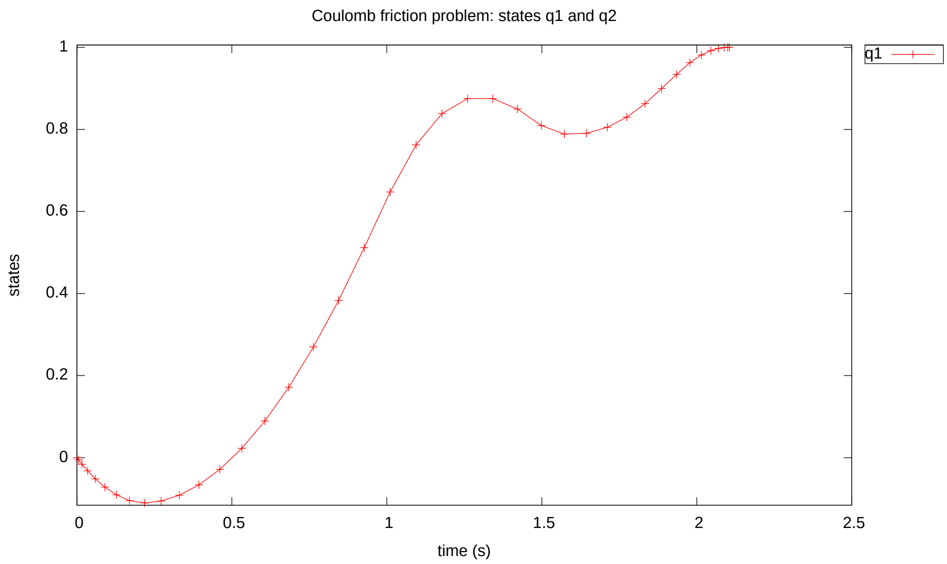

Figure 3.19: States for Coulomb friction problem

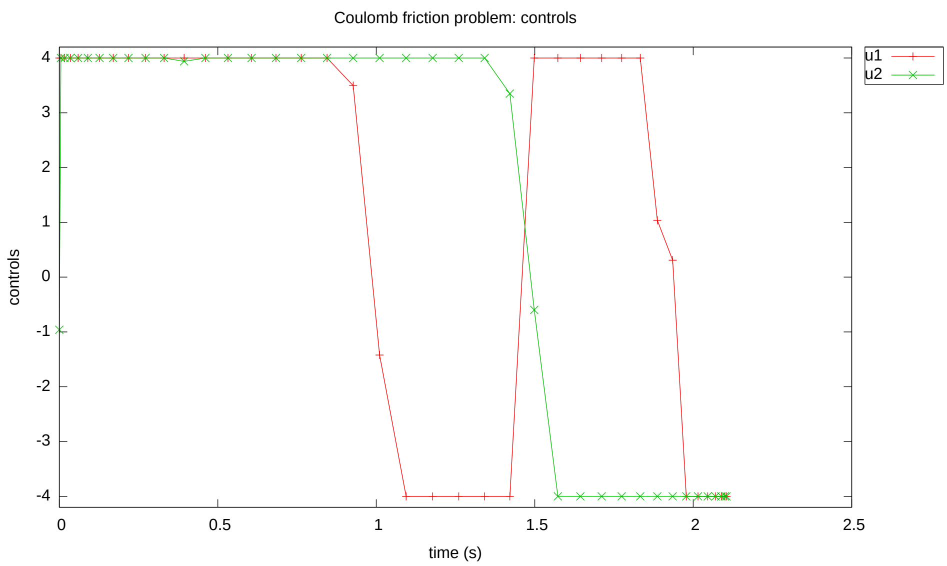

Figure 3.20: Controls for Coulomb friction problem

3.9 DAE index 3 parameter estimation problem

Consider the following parameter estimation problem, which involves a differential-algebraic equation of index 3 with four differential states and one algebraic state [35].

The dynamics consists of the differential equations

$$\begin{aligned} \dot{x}_1(t) &= x_3(t) \\ \dot{x}_2(t) &= x_4(t) \\ \dot{x}_3(t) &= \lambda(t)x_1(t) \\ \dot{x}_4(t) &= \lambda(t)x_2(t) \end{aligned} \tag{3.36}$$

and the algebraic equation

$$0 = L^2 - x_1(t)^2 - x_2(t)^2 \tag{3.37}$$

where $x_j(t), j = 1, \ldots, 4$ are the differential states, $\lambda(t)$ is an algebraic state (note that algebraic states are treated as control variables), and L is a parameter to be estimated.

The observations function is given by:

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= x_2 \end{aligned} \tag{3.38}$$

And the following least squares objective is minimised:

$$J = \sum_{k=1}^{n_s} \left[(y_1(t_k) - \hat{y}_1(t_k))^2 + (y_2(t_k) - \hat{y}_2(t_k))^2 \right] \tag{3.39}$$

where $n_s = 20$, $t_1 = 0.5$ and $t_{20} = 10.0$.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
//////////////////        dae_i3.cxx      ////////////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example           ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:  DAE Index 3                            ////////////////
//////// Last modified:  07 June 2011                 ////////////////
//////// Reference:     Schittkowski (2002)          ////////////////
//////// (See PSOPT handbook forf full reference)           ///////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2011        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the observation function //////////
//////////////////////////////////////////////////////////////////////////
```

```
void  observation_function( adouble* observations,
                            adouble* states, adouble* controls,
                            adouble* parameters, adouble& time, int k,
                            adouble* xad, int iphase, Workspace* workspace)
{
      observations[ 0 ] = states[ 0 ];
      observations[ 1 ] = states[ 1 ];

}



//////////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

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

    // Variables
       adouble x1, x2, x3, x4, L, OMEGA, LAMBDA;
       adouble dx1, dx2, dx3, dx4;

    // Differential states
       x1 = states[0];
       x2 = states[1];
       x3 = states[2];
       x4 = states[3];

    // Algebraic variables
       LAMBDA = controls[0];


    // Parameters
       L     = parameters[0];
    // Differential equations

      dx1 = x3;

      dx2 = x4;

      dx3 = LAMBDA*x1;

      dx4 = LAMBDA*x2;

      derivatives[ 0 ] = dx1;
      derivatives[ 1 ] = dx2;
      derivatives[ 2 ] = dx3;
      derivatives[ 3 ] = dx4;


     // algebraic equation

      path[ 0 ] = L*L - x1*x1 - x2*x2;

}

////////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
       // no events

       return;

}


///////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}
```

```
////////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name           =       "DAE Index 3";
    problem.outfilename          =       "dae_i3.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

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

    psopt_level1_setup(problem);


/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates     = 4;
    problem.phases(1).ncontrols   = 1;
    problem.phases(1).nevents     = 0;
    problem.phases(1).npath       = 1;
    problem.phases(1).nparameters       = 1;
    problem.phases(1).nodes         << 30;
    problem.phases(1).nobserved         = 2;
    problem.phases(1).nsamples          = 20;

    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////////
////////////  Load data for parameter estimation                ////////////
////////////////////////////////////////////////////////////////////////////

   int iphase = 1;
   load_parameter_estimation_data(problem, iphase, "../../../examples/dae_i3/dae_i3.dat");

   Print(problem.phases(1).observation_nodes, "observation nodes");
   Print(problem.phases(1).observations, "observations");
   Print(problem.phases(1).residual_weights, "weights");


////////////////////////////////////////////////////////////////////////////
//////////////////  Declare MatrixXd objects to store results /////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd x, u, p, t;

////////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states(0) = -2.0;
    problem.phases(1).bounds.lower.states(1) = -2.0;
    problem.phases(1).bounds.lower.states(2) = -2.0;
    problem.phases(1).bounds.lower.states(3) = -2.0;

    problem.phases(1).bounds.upper.states(0) = 2.0;
    problem.phases(1).bounds.upper.states(1) = 2.0;
    problem.phases(1).bounds.upper.states(2) = 2.0;
    problem.phases(1).bounds.upper.states(3) = 2.0;

    problem.phases(1).bounds.lower.controls(0) = -10.0;
```

```
    problem.phases(1).bounds.upper.controls(0) =  10.0;

    problem.phases(1).bounds.lower.parameters(0)  = 0.0;
    problem.phases(1).bounds.upper.parameters(0)  = 5.0;


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

    problem.phases(1).bounds.lower.StartTime    = 0.5;
    problem.phases(1).bounds.upper.StartTime    = 0.5;

    problem.phases(1).bounds.lower.EndTime      = 10.0;
    problem.phases(1).bounds.upper.EndTime      = 10.0;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;
    problem.observation_function = & observation_function;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes =     (int) problem.phases(1).nsamples;

    MatrixXd state_guess(4, nnodes);
    MatrixXd control_guess(1,nnodes);
    MatrixXd param_guess(1,1);

    state_guess <<   problem.phases(1).observations.row(0),
     problem.phases(1).observations.row(1),
     ones(1,nnodes),
     ones(1,nnodes);

    control_guess = zeros(1,nnodes);

    param_guess << 0.5;


    problem.phases(1).guess.states        = state_guess;
    problem.phases(1).guess.time          = problem.phases(1).observation_nodes;
    problem.phases(1).guess.parameters    = param_guess;
    problem.phases(1).guess.controls      = control_guess;



////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////

    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.collocation_method          = "Legendre";


////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////

    x = solution.get_states_in_phase(1);
    u = solution.get_controls_in_phase(1);
    t = solution.get_time_in_phase(1);
    p = solution.get_parameters_in_phase(1);


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

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

```
    Save(u,"u.dat");
    Save(t,"t.dat");
    Print(p,"Estimated parameter");


////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

     MatrixXd tm;
     MatrixXd ym;

     tm = problem.phases(1).observation_nodes;
     ym = problem.phases(1).observations;


     plot(t,x.row(0) ,tm,ym.row(0) ,problem.name, "time (s)", "state x1", "x1 yhat1");
     plot(t,x.row(1) ,tm,ym.row(1) ,problem.name, "time (s)", "state x2", "x2 yhat2");
     plot(t,u,problem.name, "time (s)", "algebraic state u", "u");

     plot(t,x.row(0),tm,ym.row(0),problem.name, "time (s)", "state x1", "x1 yhat1",
  "pdf", "x1.pdf");
     plot(t,x.row(1),tm,ym.row(1),problem.name, "time (s)", "state x2", "x2 yhat2",
  "pdf", "x2.pdf");
     plot(t,u,problem.name, "time (s)", "algebraic state lambda", "lambda", "pdf", "lambda.pdf");


}




////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ summarised in the box below and shown in Figures 3.21 and 3.22, which compare the observations with the estimated outputs, and 3.23, which shows the algebraic state. The exact solution to the problem is $L = 1$ and $\lambda(t) = -1$. The numerical solution obtained is $L = 1.000000188$ and $\lambda(t) = -0.999868$. The 95% confidence interval for the estimated parameter is $[0.9095289, \ 1.090471]$

```
PSOPT results summary
=====================

Problem:  DAE Index 3
CPU time (seconds): 2.417066e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:18:43 2020

Optimal (unscaled) cost function value:  4.281388e-16
Phase 1 endpoint cost function value: 4.281388e-16
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 5.000000e-01
Phase 1 final time: 1.000000e+01
```

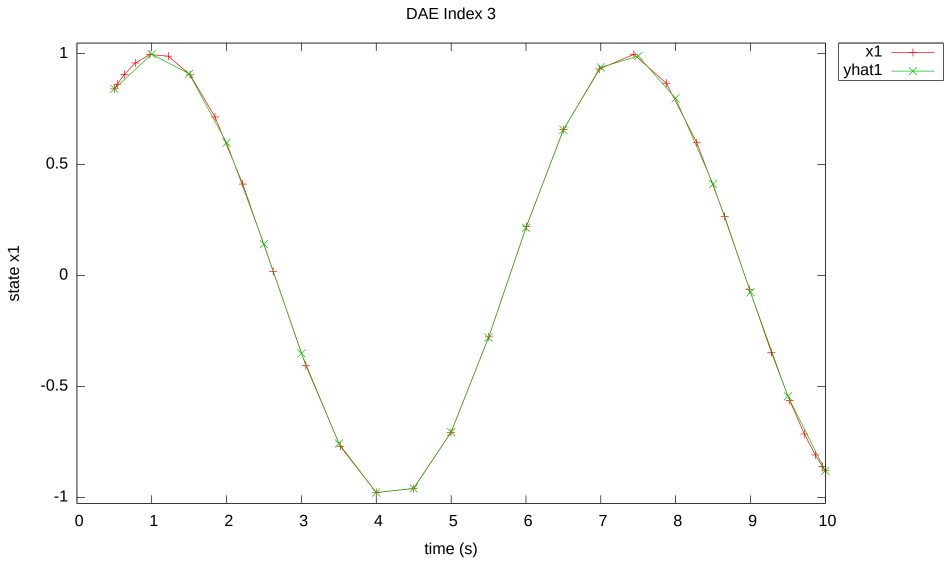

Figure 3.21: State x_1 and observations

```
Phase 1 maximum relative local error: 1.542988e-08
NLP solver reports:  The problem has been solved!
```

3.10 Delayed states problem 1

Consider the following optimal control problem, which consists of a linear system with delays in the state equations [27]. Minimize the cost functional:

$$J = x_3(t_f) \tag{3.40}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2(t) \\ \dot{x}_2 &= -10x_1(t) - 5x_2(t) - 2x_1(t-\tau) - x_2(t-\tau) + u(t) \\ \dot{x}_3 &= 0.5(10x_1^2(t) + x_2^2(t) + u^2(t)) \end{aligned} \tag{3.41}$$

and the boundary conditions

$$\begin{aligned} x_1(0) &= 1 \\ x_2(0) &= 1 \\ x_3(0) &= 0 \end{aligned} \tag{3.42}$$

where $t_f = 5$ and $\tau = 0.25$.

The output from $\mathcal{PSOPT}$ summarised in the box below and shown in Figures 3.24 and 3.25, which contain the elements of the state and the control, respectively.

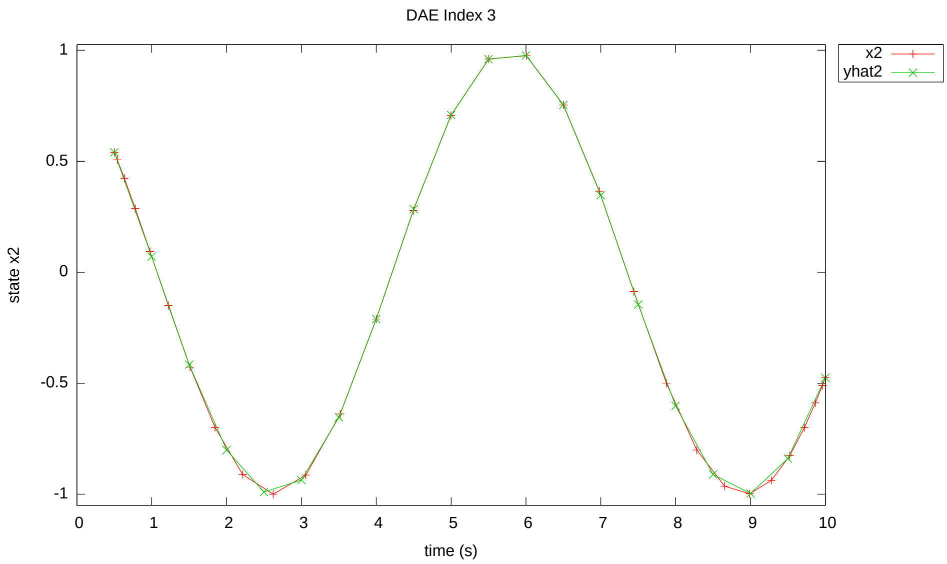

Figure 3.22: State x_2 and observations

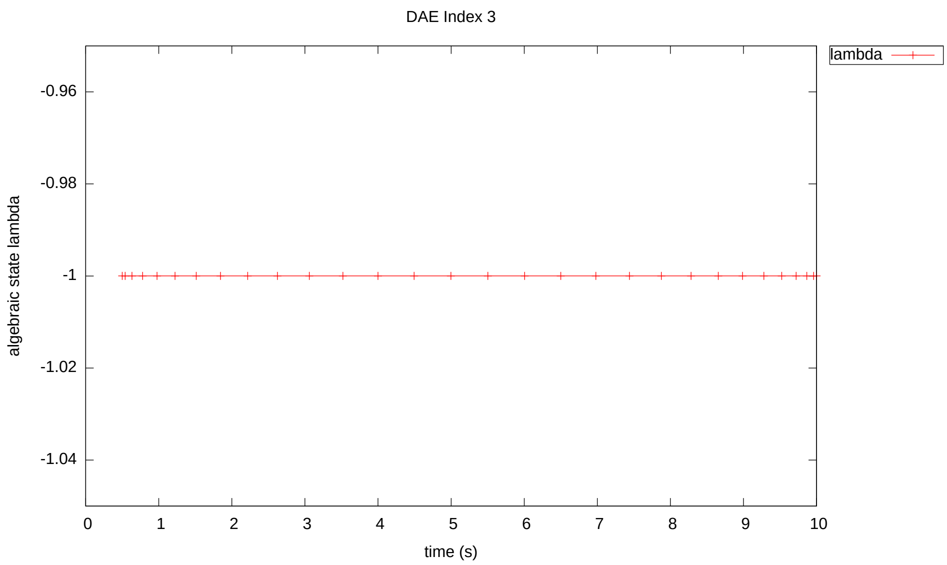

Figure 3.23: Algebraic state $\lambda(t)$

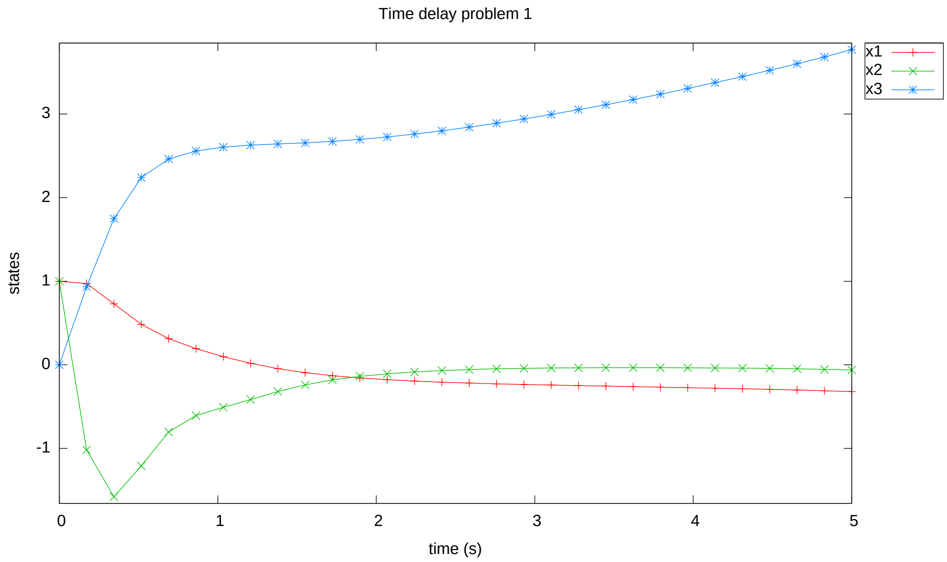

Figure 3.24: States for time delay problem 1

```
PSOPT results summary
=====================

Problem:  Time delay problem 1
CPU time (seconds): 9.077910e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:18:59 2020

Optimal (unscaled) cost function value:  3.770849e+00
Phase 1 endpoint cost function value: 3.770849e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.000000e+00
Phase 1 maximum relative local error: 1.838200e-02
NLP solver reports:  The problem has been solved!
```

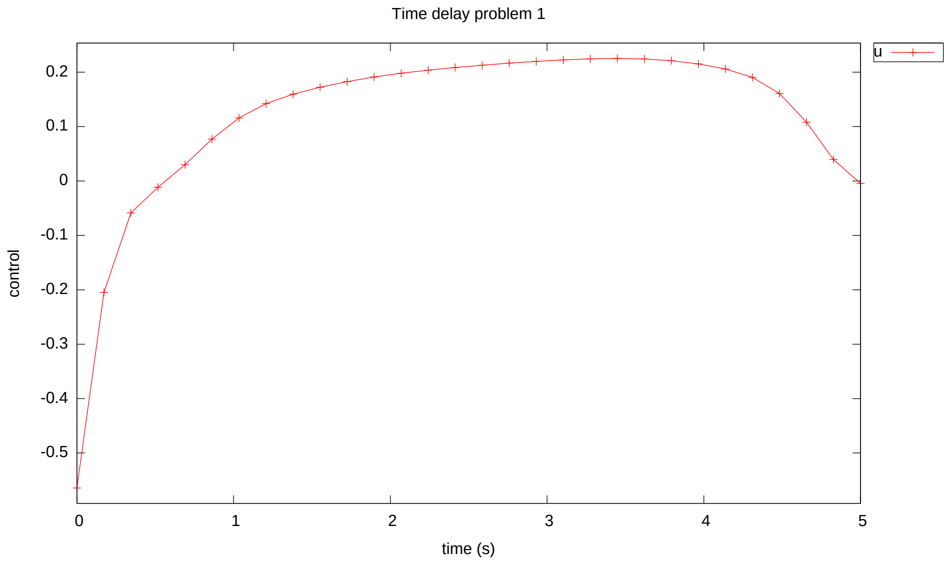

Figure 3.25: Control for time delay problem 1

3.11 Dynamic MPEC problem

Consider the following optimal control problem, which involves special handling of a system with a discontinuous right hand side [4]. Minimize the cost functional:

$$J = [y(2) - 5/3]^2 + \int_0^2 y^2(t)\mathrm{d}t \tag{3.43}$$

subject to

$$\dot{y} \;=\; 2 - \mathrm{sgn}(y) \tag{3.44}$$

and the boundary condition

$$y(0) \;=\; -1 \tag{3.45}$$

Note that there is no control variable, and the analytical solution of this problem satisfies $\dot{y}(t) = 3$, $0 \le t \le 1/3$, and $\dot{y}(t) = 1$, $1/3 \le t \le 2$.

In order to handle the discontinuous right hand side, the problem is converted into the following equivalent problem, which has three algebraic (control) variables. This type of problem is known in the literature as a dynamic MPEC problem.

$$J = [y(2) - 5/3]^2 + \int_0^2 \left(y^2(t) + \rho\left\{p(t)[s(t)+1] + q(t)[1 - s(t)]\right\}\right)\mathrm{d}t \tag{3.46}$$

subject to

$$\begin{aligned} \dot{y} &= 2 - \mathrm{sgn}(y) \\ 0 &= -y(t) - p(t) + q(t) \end{aligned} \tag{3.47}$$

the boundary condition

$$y(0) \quad = \quad -1 \tag{3.48}$$

and the bounds:

$$\begin{aligned} -1 &\leq s(t) \leq 1, \\ 0 &\leq p(t), \\ 0 &\leq q(t). \end{aligned} \tag{3.49}$$

The output from $\mathcal{PSOPT}$ summarised in the box below and shown in Figures 3.26, 3.27, 3.28, and 3.29.

```
PSOPT results summary
=====================

Problem:  Dynamic MPEC problem
CPU time (seconds): 1.978309e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:24:07 2020

Optimal (unscaled) cost function value:  1.653481e+00
Phase 1 endpoint cost function value: 3.809169e-07
Phase 1 integrated part of the cost:  1.653481e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.000000e+00
Phase 1 maximum relative local error: 7.761572e-07
NLP solver reports:  The problem has been solved!
```

3.12 Geodesic problem

This problem is about calculating the geodesic curve [1] that joins two points on Earth using optimal control. The problem is posed in the form of estimating the shortest fligh path for an airliner to fly from New Yorks's JFK to London's LHR airport.

The formulation is as follows. Find the trajectories for the elevation and azimuth angles $\theta(t)$ and $\phi(t) \in [0, t_f]$ to minimize the cost functional

$$J = \int_0^{t_f} \sqrt{\dot{x}^2 + \dot{y}^2 + \dot{z}^2} dt \tag{3.50}$$

[1] See http://mathworld.wolfram.com/Geodesic.html

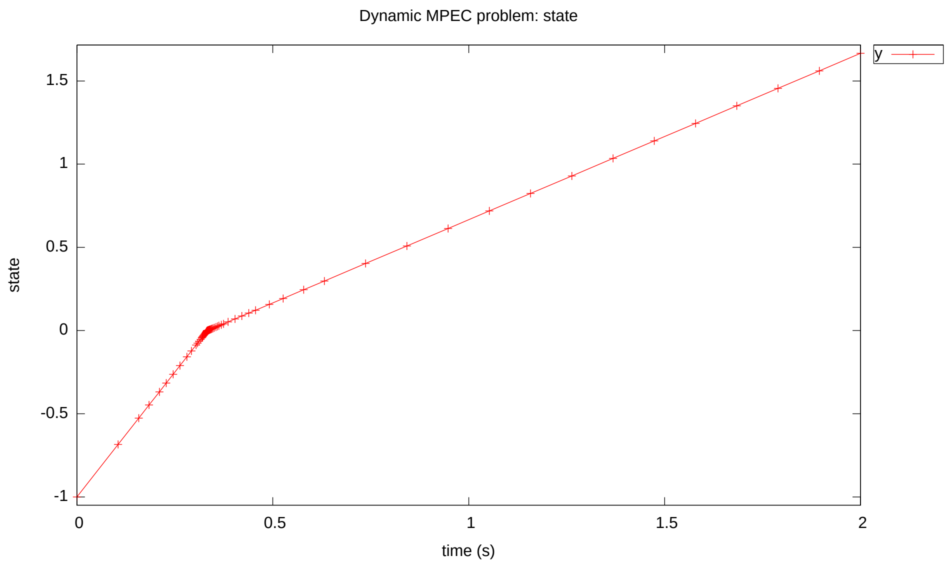

Figure 3.26: State y for dynamic MPEC problem

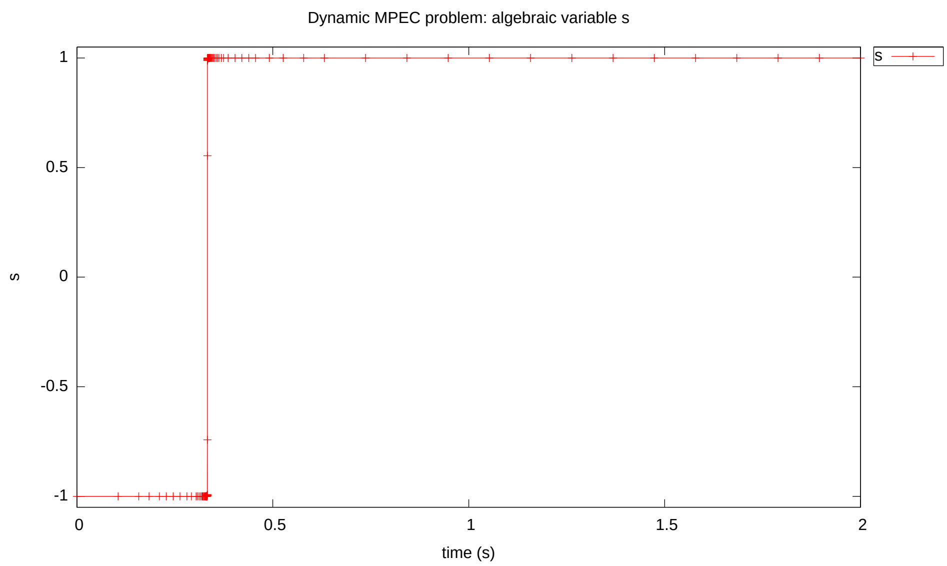

Figure 3.27: Algebraic variable s for dynamic MPEC problem

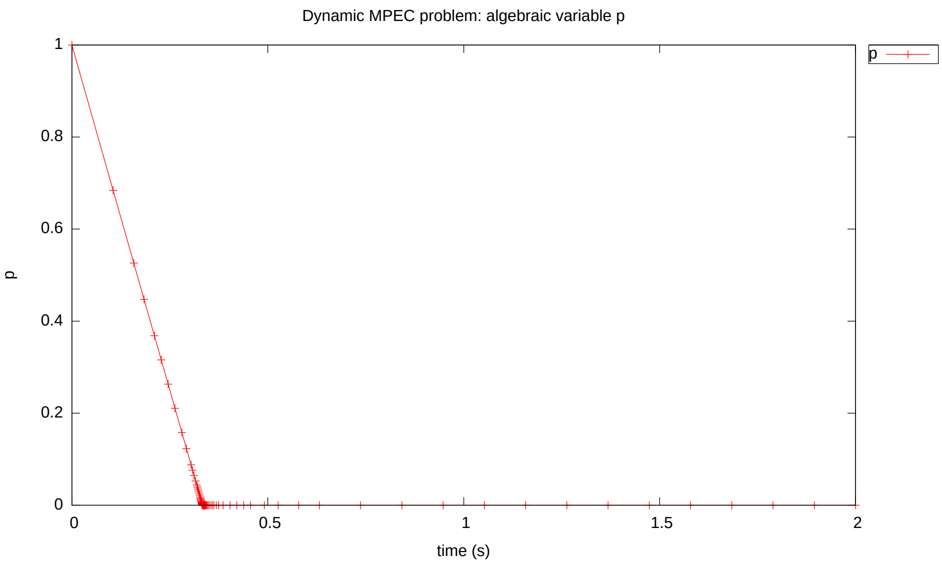

Figure 3.28: Algebraic variable p for dynamic MPEC problem

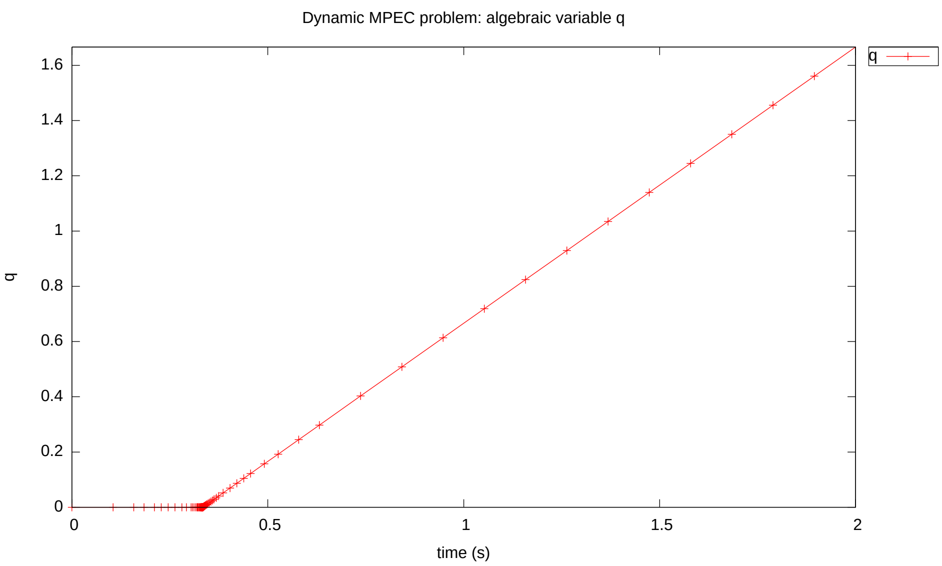

Figure 3.29: Algebraic variable q for dynamic MPEC problem

subject to the dynamic constraints

$$\begin{array}{rcl} \dot{x} & = & V\sin(\theta)\cos(\phi) \\ \dot{y} & = & V\sin(\theta)\sin(\phi) \\ \dot{z} & = & V\cos(\theta) \end{array} \tag{3.51}$$

The path constraint, which corresponds to the Earth's spheroid [2] shape:

$$\frac{x^2}{a^2} + \frac{y^2}{a^2} + \frac{z^2}{b^2} - 1.0 = 0 \tag{3.52}$$

the boundary conditions, which correspond to the geographical coordinates of LHR (51.4700° N, 0.4543° W) and JFK (40.6413° N, 73.7781° W)

$$\begin{array}{rcl} x(0) & = & x_0 \\ y(0) & = & y_0 \\ z(0) & = & z_0 \\ x(t_f) & = & x_f \\ y(t_f) & = & y_f \\ z(t_f) & = & z_f \end{array} \tag{3.53}$$

and the control bounds

$$\begin{array}{rcccl} 0 & \leq & \theta(t) & \leq & \pi \\ 0 & \leq & \phi(t) & \leq & 2\pi \end{array} \tag{3.54}$$

where x, y, z are the Cartesian coordinates (in km) with origin on the centre of Earth, t is time in hours, $V = 900$ km/h corresponds to the cruising speed of a typical airliner, $a = 6384$ km is the Earth's semi-major axis, and $b = 6353$ km is the Earth's semi-minor axis, which is the length of the Earth's axis of rotation from the north pole to the south pole. For simplicity, the altitude of the aircraft is neglected.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.30, 3.31 and 3.32, which show the flight path, the elements of the state vector, and the elements of the control vector, respectively. Note that $\mathcal{PSOPT}$ predicts that the length of the shortest flightpath is 5,540.4 km, and the flight time is 6 hours 9 min.

```
PSOPT results summary
=====================

Problem:  Geodesic problem
CPU time (seconds): 4.393502e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
```

[2]See http://mathworld.wolfram.com/Ellipsoid.html

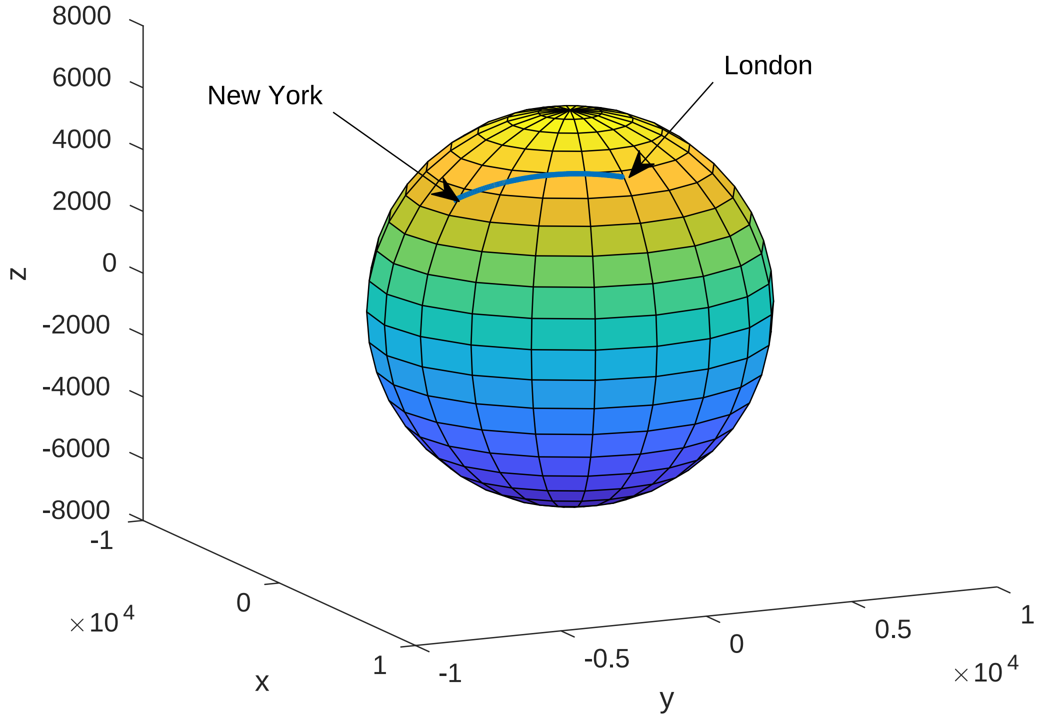

Figure 3.30: Flight path for geodesic problem

```
Date and time of this run:  Wed Sep 23 12:19:16 2020

Optimal (unscaled) cost function value:  5.540405e+03
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.540405e+03
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 6.156005e+00
Phase 1 maximum relative local error: 7.825546e-05
NLP solver reports:  The problem has been solved!
```

3.13 Goddard rocket maximum ascent problem

Consider the following optimal control problem, which is known in the literature as the Goddard rocket maximum ascent problem [6]. Find t_f and $T(t) \in [t_0, t_f]$ to minimize

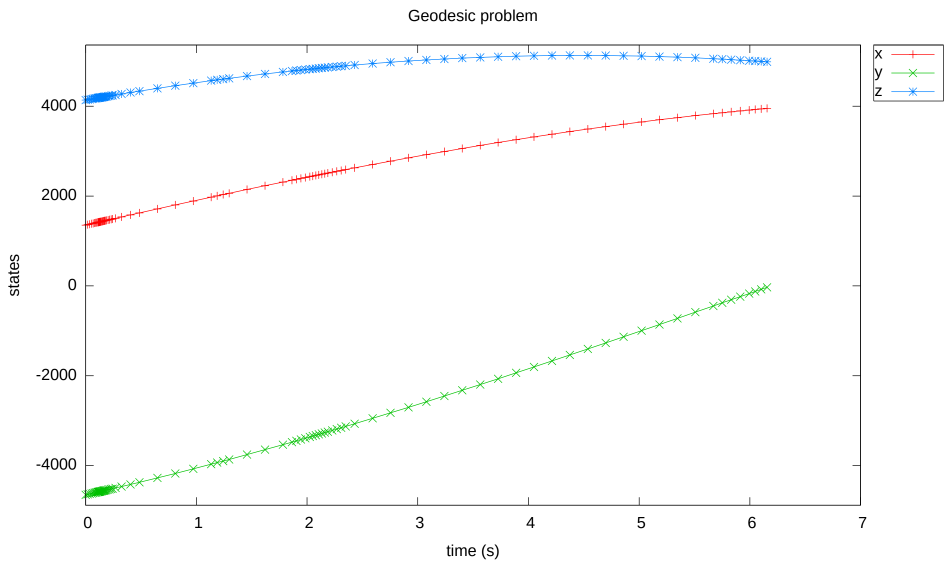

Figure 3.31: States for geodesic problem

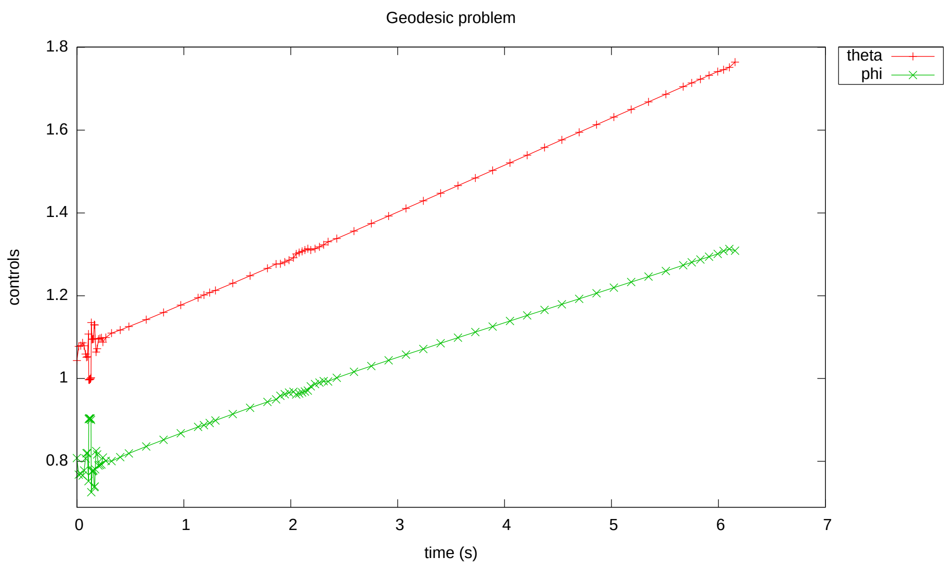

Figure 3.32: Controls for geodesic problem

the cost functional

$$J = h(t_f) \tag{3.55}$$

subject to the dynamic constraints

$$\begin{array}{rcl} \dot{v} & = & \frac{1}{m}(T - D) - g \\ \dot{h} & = & v \\ \dot{m} & = & -\frac{T}{c} \end{array} \tag{3.56}$$

the boundary conditions:

$$\begin{array}{rcl} v(0) & = & 0 \\ h(0) & = & 1 \\ m(0) & = & 1 \\ m(t_f) & = & 0.6 \end{array} \tag{3.57}$$

the state bounds:

$$\begin{array}{rcl} 0.0 \leq & v(t) & \leq 2.0 \\ 1.0 \leq & h(t) & \leq 2.0 \\ 0.6 \leq & m(t) & \leq 1.0 \end{array} \tag{3.58}$$

and the control bounds

$$0 \leq T(t) \leq 3.5 \tag{3.59}$$

where

$$\begin{array}{rcl} D & = & D_0 v^2 \exp(-\beta h) \\ g & = & 1/(h^2) \end{array}, \tag{3.60}$$

$D_0 = 310$, $\beta = 500$, and $c = 0.5$, $0.1 \leq t_f \leq 1$.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.33 and 3.34, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Goddard Rocket Maximum Ascent
CPU time (seconds): 1.615035e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:21:05 2020

Optimal (unscaled) cost function value:  -1.025336e+00
Phase 1 endpoint cost function value: -1.025336e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.605347e-01
Phase 1 maximum relative local error: 6.877099e-04
NLP solver reports:  The problem has been solved!
```

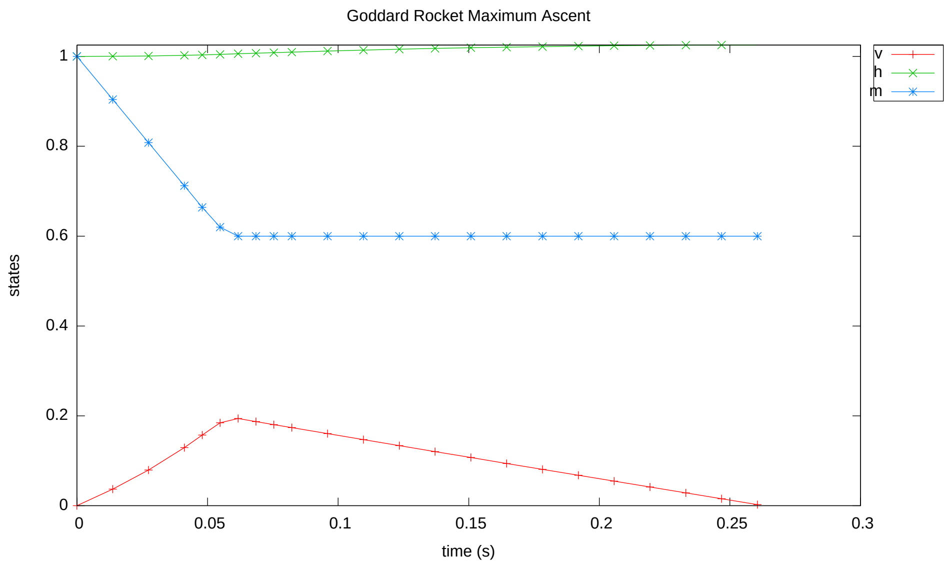

Figure 3.33: States for Goddard rocket problem

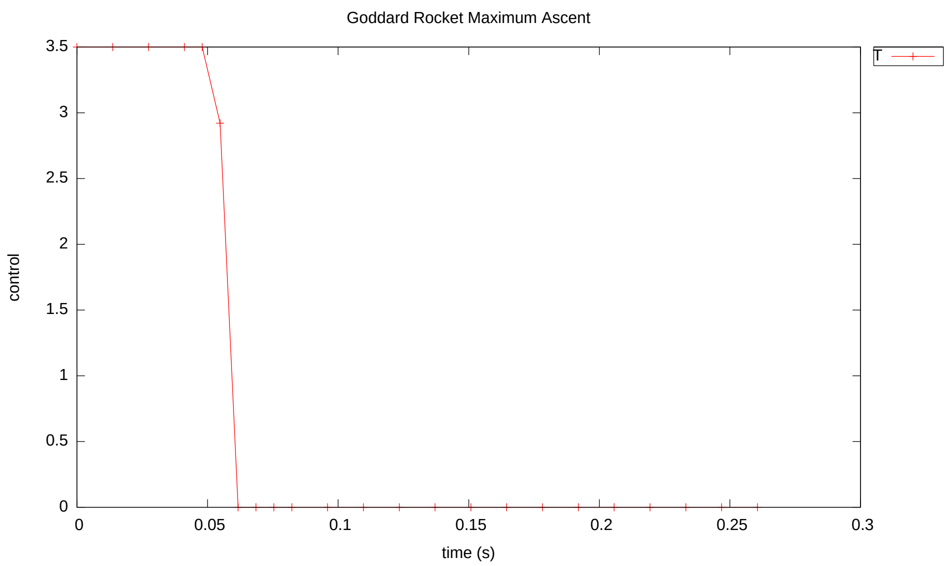

Figure 3.34: Control for Goddard rocket problem

3.14 Hang glider

This problem is about the range maximisation of a hang glider in the presence of a specified thermal draft [4]. Find t_f and $C_L(t), t \in [0, t_f]$, to minimise,

$$J = x(t_f) \tag{3.61}$$

subject to the dynamic constraints

$$\begin{array}{rcl} \dot{x} & = & v_x \\ \dot{y} & = & v_y \\ \dot{v}_x & = & \frac{1}{m}(-L\sin\eta - D\cos\eta) \\ \dot{v}_y & = & \frac{1}{m}(L\cos\eta - D\sin\eta - W) \end{array} \tag{3.62}$$

where

$$\begin{aligned} C_D &= C_0 + kC_L^2 \\ v_r &= \sqrt{v_x^2 + v_y^2} \\ D &= \frac{1}{2}C_D\rho S v_r^2 \\ L &= \frac{1}{2}C_L\rho S v_r^2 \\ X &= \left(\frac{x}{R} - 2.5\right)^2 \\ u_a &= u_M(1 - X)\exp(-X) \\ V_y &= v_y - ua \\ \sin\eta &= \frac{V_y}{v_r} \\ \cos\eta &= \frac{v_x}{v_r} \\ W &= mg \end{aligned} \tag{3.63}$$

The control is bounded as follows:

$$0 \le C_L \le 1.4 \tag{3.64}$$

and the following boundary conditions:

$$\begin{array}{rr} x(0) = 0, & x(t_f) = \text{free} \\ y(0) = 1000, & y(t_f) = 900 \\ v_x(0) = 13.227567500, & v_x(t_f) = 13.227567500 \\ v_y(0) = -1.2875005200, & v_y(t_f) = -1.2875005200 \end{array} \tag{3.65}$$

With the following parameter values:

$$\begin{array}{rr} u_M = 2.5, & m = 100.0 \\ R = 100.0, & S = 14, \\ C_0 = 0.034, & \rho = 1.13 \\ k = 0.069662, & g = 9.80665 \end{array} \tag{3.66}$$

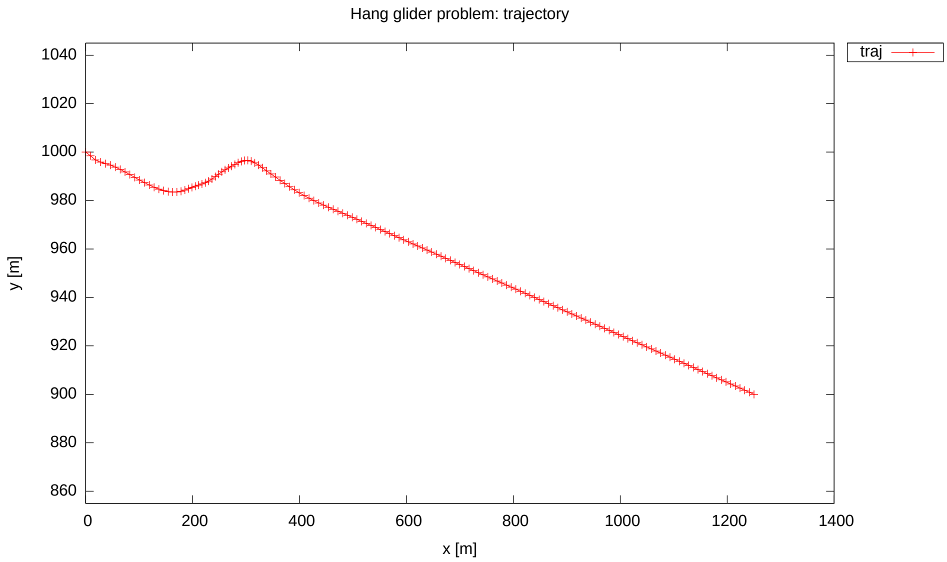

Figure 3.35: $x - y$ trajectory for hang glider

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.35, 3.36 and 3.37.

```
PSOPT results summary
=====================

Problem:  Hang glider problem
CPU time (seconds): 4.201823e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 13:26:04 2020

Optimal (unscaled) cost function value:  -1.250183e+03
Phase 1 endpoint cost function value: -1.250183e+03
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 9.854617e+01
Phase 1 maximum relative local error: 1.498813e-01
NLP solver reports:  The problem has been solved!
```

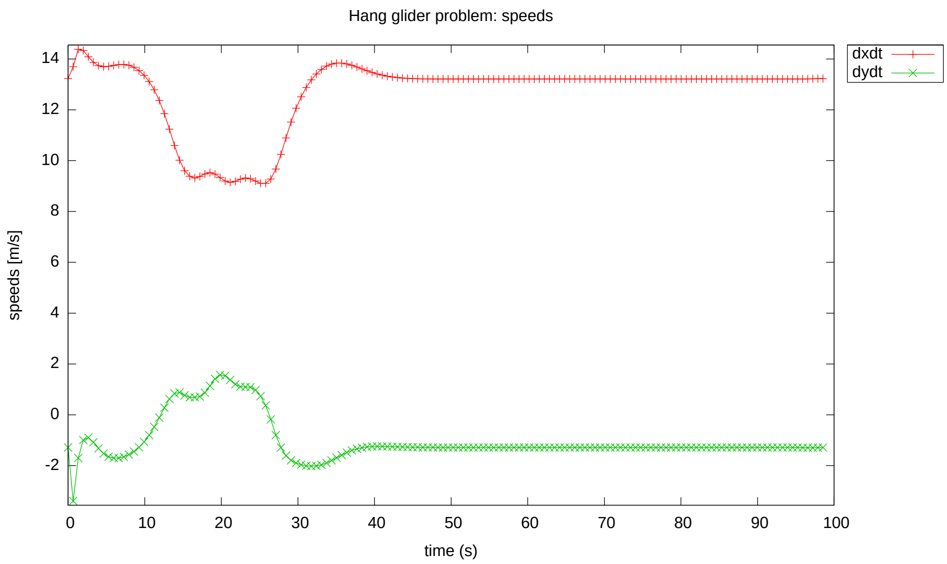

Figure 3.36: Velocities for hang glider

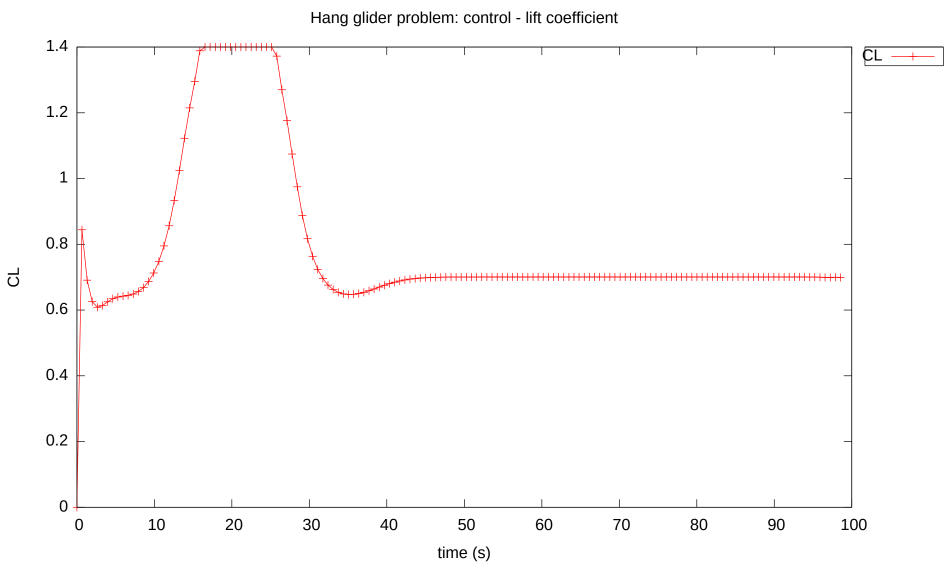

Figure 3.37: Lift coefficient for hang glider problem

3.15 Hanging chain problem

Consider the following optimal control problem, which includes an integral constraint. Minimize the cost functional

$$J = \int_0^{t_f} \left[x\sqrt{1+(\dot{x})^2} \right] dt \tag{3.67}$$

subject to the dynamic constraint

$$\dot{x} = u \tag{3.68}$$

the integral constraint:

$$\int_0^{t_f} \left[\sqrt{1+\left(\frac{dx}{dt}\right)^2} \right] dt = 4 \tag{3.69}$$

the boundary conditions

$$\begin{array}{rcl} x(0) & = & 1 \\ x(t_f) & = & 3 \end{array} \tag{3.70}$$

and the bounds:

$$\begin{array}{c} -20 \leq u(t) \leq 20 \\ -10 \leq x(t) \leq 10 \end{array} \tag{3.71}$$

where $t_f = 1$.

The output from $\mathcal{PSOPT}$ is summarized in the text box below and in Figure 3.38, which illustrates the shape of the hanging chain.

```
PSOPT results summary
=====================

Problem:  Hanging chain problem
CPU time (seconds): 9.451230e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:13:58 2020

Optimal (unscaled) cost function value:  5.068480e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.068480e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 2.429792e-06
NLP solver reports:  The problem has been solved!
```

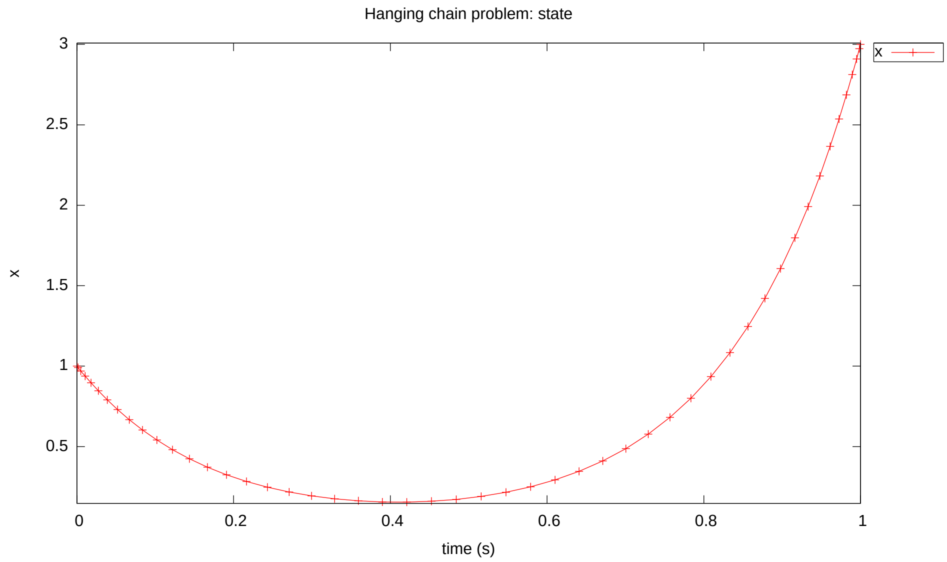

Figure 3.38: State for hanging chain problem

3.16 Heat difussion problem

This example can be viewed as a simplified model for the heating of a probe in a kiln [3]. The dynamics are a spatially discretized form of a partial differential equation, which is obtained by using the method of the lines. The problem is formulated on the basis of the state vector $\mathbf{x} = [x_1, \ldots, x_M]^T$ and the control vector $\mathbf{u} = [v_1, v_2, v_3]^T$, as follows

$$\min_{\mathbf{u}(t)} \; J = \frac{1}{2}\int_0^T \left\{(x_N(t) - x_d(t))^2 + \gamma v_1(t)^2\right\} \mathrm{d}t$$

subject to the differential constraints

$$\dot{x}_1 = \frac{1}{(a_1 + a_2 x_1)}\left[q_1 + \frac{1}{\delta^2}(a_3 + a_4 x_1)(x_2 - 2x_1 + v_2) + a_4\left(\frac{x_2 - x_1}{2\delta}\right)^2\right]$$

$$\dot{x}_i = \frac{1}{(a_1 + a_2 x_i)}\left[q_i + \frac{1}{\delta^2}(a_3 + a_4 x_i)(x_{i+1} - 2x_i + x_{i-1}) + a_4\left(\frac{x_{i+1} - x_{i-1}}{2\delta}\right)^2\right]$$

for $i = 2, \ldots, M-1$

$$\dot{x}_M = \frac{1}{(a_1 + a_2 x_M)}\left[q_M + \frac{1}{\delta^2}(a_3 + a_4 x_M)(v_3 - 2x_N + x_{M-1}) + a_4\left(\frac{v_3 - x_{M-1}}{2\delta}\right)^2\right]$$

the path constraints

$$0 = g(x_1 - v_1) - \frac{1}{2\delta}(a_3 + a_4 x_1)(x_2 - v_2)$$

$$0 = \frac{1}{2\delta}(a_3 + a + 4x_M)(v_3 - x_{M-1})$$

the control bounds

$$u_L \leq v_1 \leq u_U$$

and the initial conditions for the states:

$$x_i(0) = 2 + \cos(\pi z_i)$$

where

$$\begin{aligned}
z_i &= \frac{i-1}{M-1}, \ i = 1, \ldots, M \\
x_d(t) &= 2 - e^{\rho t} \\
q(z,t) &= \left[\rho(a_1 + 2a_2) + \pi^2(a_3 + 2a_4)\right] e^{\rho t} \cos(\pi z) \\
&\quad - a_4 \pi^2 e^{2\pi t} + (2a_4\pi^2 + \rho a_2) e^{2\rho t} \cos^2(\pi z) \\
q_i &\equiv q(z_i, t), \ i = 1, \ldots, M
\end{aligned}$$

with the parameter values $a_1 = 4$, $a_2 = 1$, $a_3 = 4$, $a_4 = -1$, $u_U = 0.1$, $\rho = -1$, $T = 0.5$, $\gamma = 10^{-3}$, $g = 1$, $u_L = -\infty$.

A spatial discretization given by $M = 10$ was used. The problem was solved initially by using first 50 nodes, then the mesh was refined to 60 nodes, and an interpolation of the previous solution was employed as an initial guess for the new solution.

The output from $\mathcal{PSOPT}$ is summarized the box below. Figure 3.39 shows the control variable v_1 as a function of time. Figure 3.40 shows the resulting temperature distribution.

```
PSOPT results summary
=====================

Problem:  Heat diffusion process
CPU time (seconds): 2.741054e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:21:17 2020

Optimal (unscaled) cost function value:  4.372841e-05
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  4.372841e-05
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.000000e-01
Phase 1 maximum relative local error: 1.749463e-04
NLP solver reports:  The problem has been solved!
```

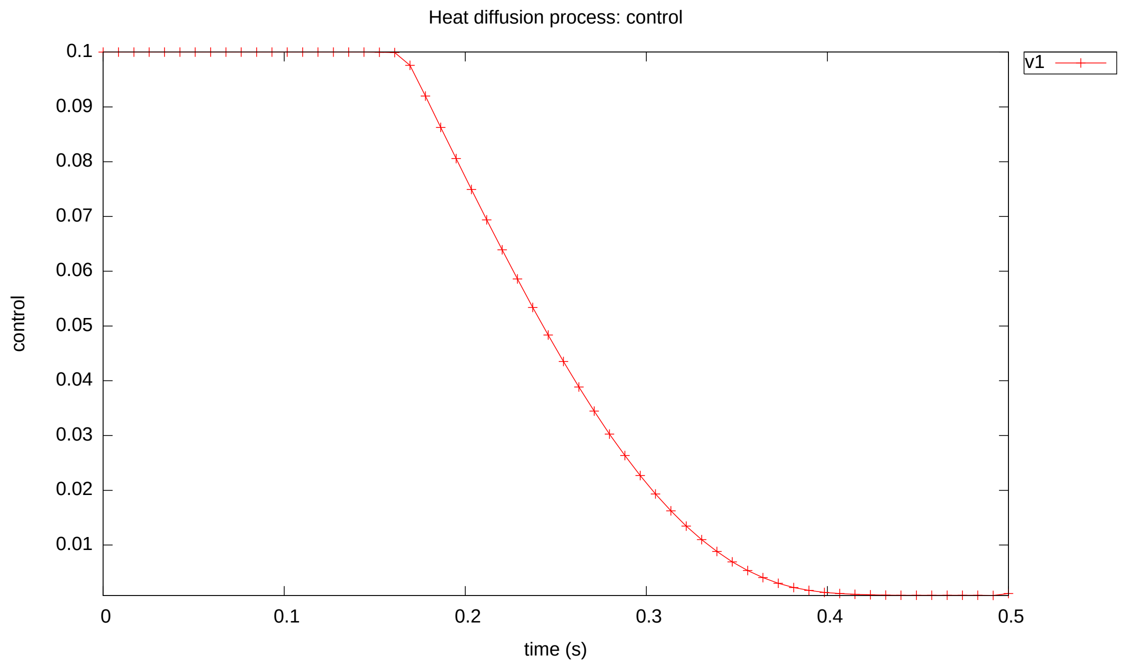

Figure 3.39: Optimal control distribution for the heat diffusion process

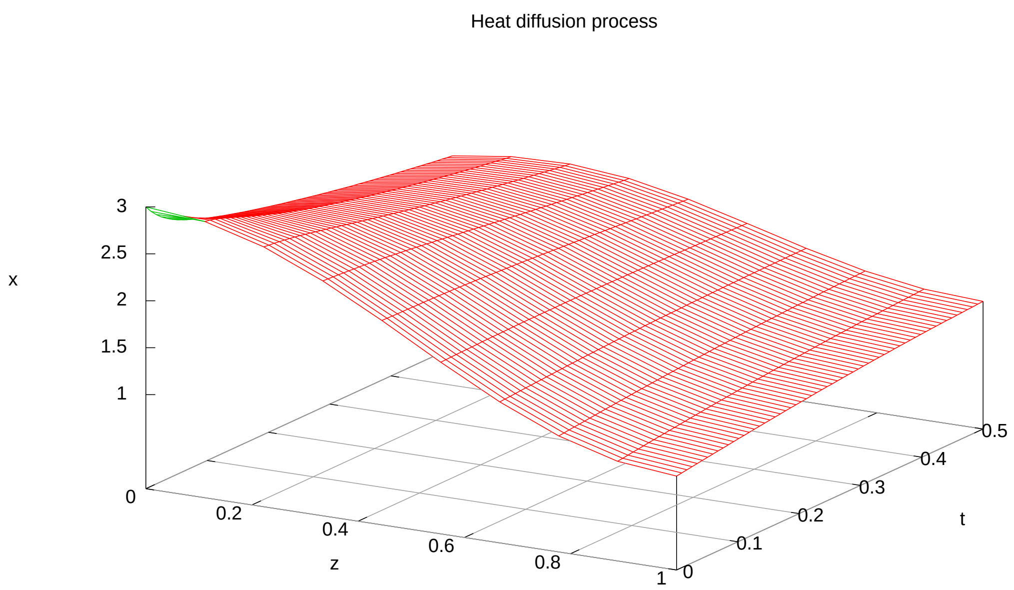

Figure 3.40: Optimal temperature distribution for the heat diffusion process

3.17 Hypersensitive problem

Consider the following optimal control problem, which is known in the literature as the hypesensitive optimal control problem [32]. Minimize the cost functional

$$J = \frac{1}{2}\int_0^{t_f} [x^2 + u^2] dt \tag{3.72}$$

subject to the dynamic constraint

$$\dot{x} = -x^3 + u \tag{3.73}$$

and the boundary conditions

$$\begin{array}{rcl} x(0) & = & 1.5 \\ x(t_f) & = & 1 \end{array} \tag{3.74}$$

where $t_f = 50$.

The output from $\mathcal{PSOPT}$ is summarized the box below and shown in the following plots that contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Hypersensitive problem
CPU time (seconds): 1.395726e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:21:29 2020

Optimal (unscaled) cost function value:  1.330826e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  1.330826e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.000000e+01
Phase 1 maximum relative local error: 5.728530e-04
NLP solver reports:  The problem has been solved!
```

3.18 Interior point constraint problem

Consider the following optimal control problem, which involves a scalar system with an interior point constraint on the state [22]. Minimize the cost functional

$$J = \int_0^1 [x^2 + u^2] dt \tag{3.75}$$

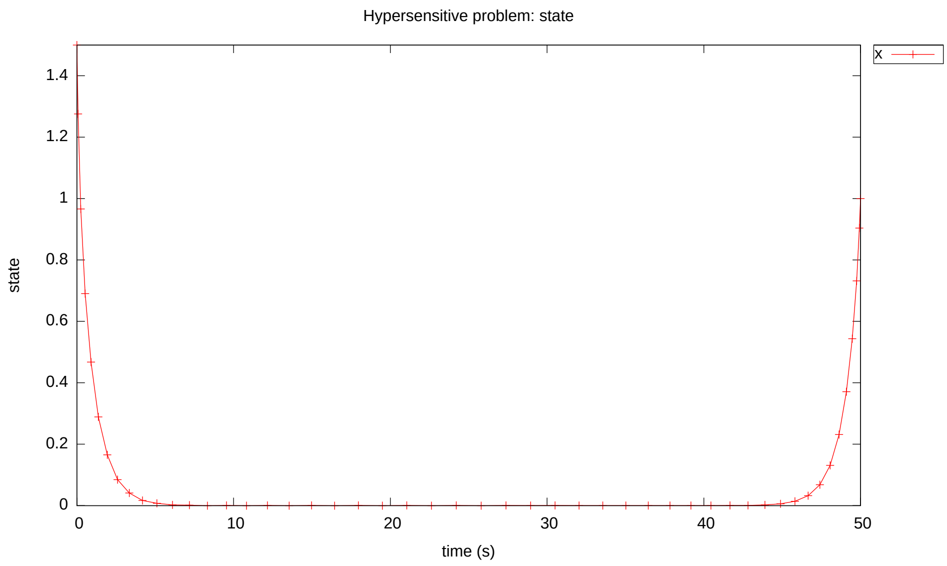

Figure 3.41: State for hypersensitive problem

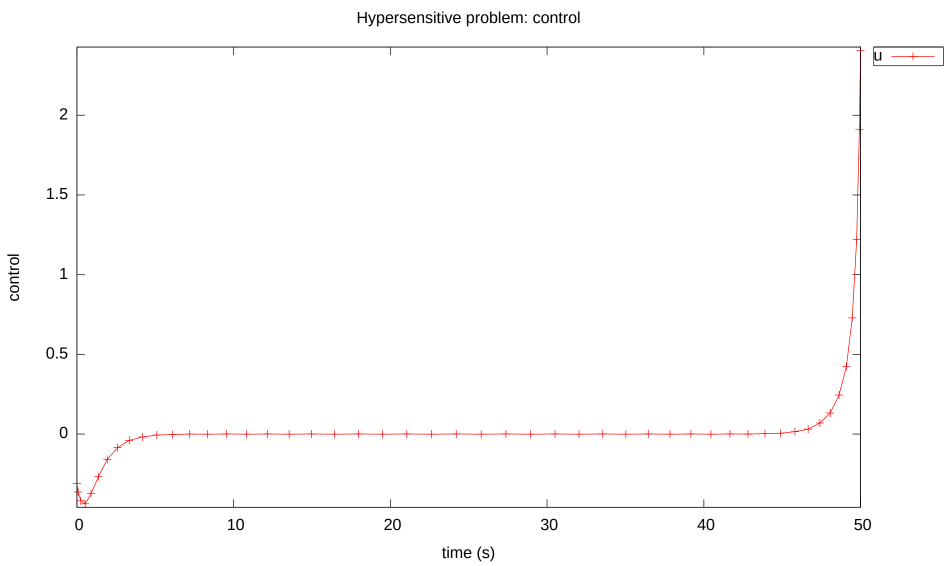

Figure 3.42: Control for hypersensitive problem

subject to the dynamic constraint

$$\dot{x} = u, \tag{3.76}$$

the boundary conditions

$$\begin{array}{rcl} x(0) & = & 1, \\ x(1) & = & 0.75, \end{array} \tag{3.77}$$

and the interior point constraint:

$$x(0.75) = 0.9. \tag{3.78}$$

The problem is divided into two phases and the interior point constraint is accommodated as an event constraint at the end of the first phase.

The output from $\mathcal{PSOPT}$ is summarized the box below and shown in the following plots that contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Problem with interior point constraint
CPU time (seconds): 4.141470e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:21:39 2020

Optimal (unscaled) cost function value:  9.205314e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  6.607877e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 7.500000e-01
Phase 1 maximum relative local error: 2.349978e-08
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  2.597438e-01
Phase 2 initial time: 7.500000e-01
Phase 2 final time: 1.000000e+00
Phase 2 maximum relative local error: 6.996100e-09
NLP solver reports:  The problem has been solved!
```

3.19 Isoperimetric constraint problem

Consider the following optimal control problem, which includes an integral constraint. Minimize the cost functional

$$J = \int_0^{t_f} x^2(t)dt \tag{3.79}$$

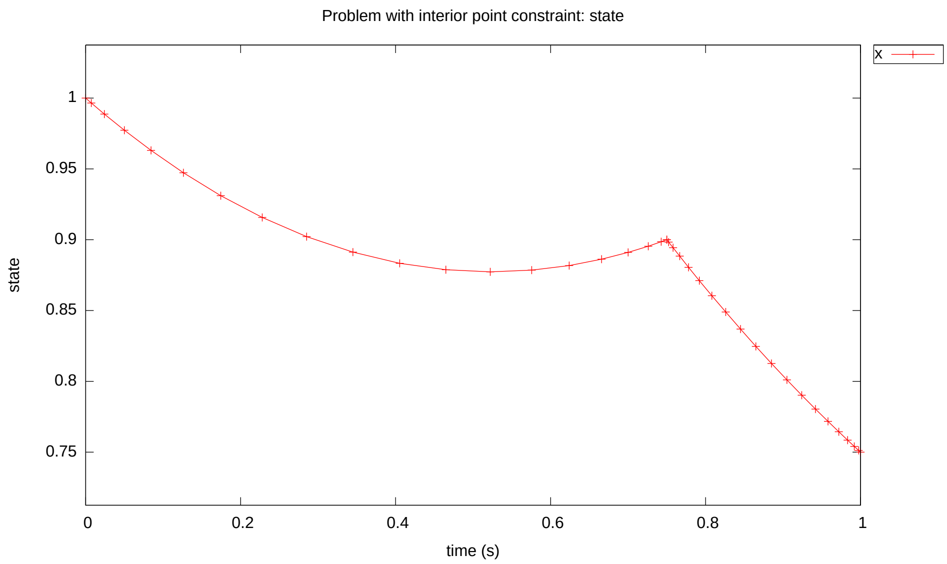

Figure 3.43: State for interior point constraint problem

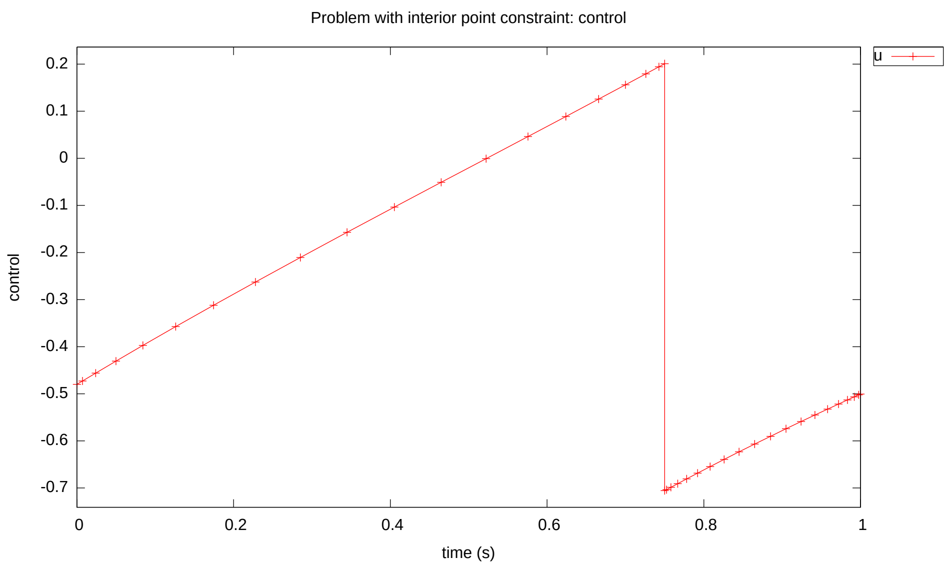

Figure 3.44: Control for interior point constraint problem

subject to the dynamic constraint

$$\dot{x} = -\sin(x) + u \tag{3.80}$$

the integral constraint:

$$\int_0^{t_f} u^2(t)dt = 10 \tag{3.81}$$

the boundary conditions

$$\begin{aligned} x(0) &= 1 \\ x(t_f) &= 0 \end{aligned} \tag{3.82}$$

and the bounds:

$$\begin{aligned} -4 \leq u(t) \leq 4 \\ -10 \leq x(t) \leq 10 \end{aligned} \tag{3.83}$$

where $t_f = 1$. The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
//////////////////        isoperimetric.cxx        //////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example             ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:                 Isoperimetric problem     ////////////////
//////// Last modified:         29 January 2009           ////////////////
//////// Reference:                                       ////////////////
////////                                                  ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return 0.0;
}

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    adouble x = states[0];

    adouble L = x;

    return  L;
}

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

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

```
        adouble* xad, int iphase, Workspace* workspace)
{
   adouble xdot, ydot, vdot;

   adouble x = states[ 0 ];

   adouble u = controls[ 0 ];

   derivatives[0] = -sin(x) + u;

}


////////////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand of the integral constraint ///////
////////////////////////////////////////////////////////////////////////////

adouble integrand( adouble* states, adouble* controls, adouble* parameters,
                    adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
   adouble g;
   adouble u = controls[ 0 ];

   g =  u*u ;

   return g;

}


////////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x0 = initial_states[ 0 ];
   adouble xf = final_states[   0 ];
   adouble Q;

   // Compute the integral to be constrained
   Q = integrate( integrand, xad, iphase, workspace );

   e[ 0 ] = x0;
   e[ 1 ] = xf;
   e[ 2 ] = Q;

}



///////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


////////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////


int main(void)
{

////////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

```
    problem.name        = "Isoperimetric constraint problem";

    problem.outfilename                 = "isoperimetric.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

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

    psopt_level1_setup(problem);


/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
/////////////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates    = 1;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 3;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes << 50;

    psopt_level2_setup(problem, algorithm);



////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.phases(1).bounds.lower.states << -10;
    problem.phases(1).bounds.upper.states << 10;

    problem.phases(1).bounds.lower.controls << -4.0;
    problem.phases(1).bounds.upper.controls <<  4.0;

    problem.phases(1).bounds.lower.events    <<  1.0, 0.0, 10.0;


    problem.phases(1).bounds.upper.events    <<  1.0, 0.0, 10.0;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.0;
    problem.phases(1).bounds.upper.EndTime      = 1.0;


////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost                    = &integrand_cost;
    problem.endpoint_cost                       = &endpoint_cost;
    problem.dae                             = &dae;
    problem.events                          = &events;
    problem.linkages                         = &linkages;




////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.phases(1).guess.controls       = zeros(1,30);
    problem.phases(1).guess.states         = zeros(1,30);
    problem.phases(1).guess.time           = linspace(0.0,1.0,30);

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
```

```
    algorithm.nlp_iter_max              = 1000;
    algorithm.nlp_tolerance             = 1.e-6;

////////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

    if (solution.error_flag) exit(0);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////

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

////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

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


////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name + ": state", "time (s)", "x", "x");

    plot(t,u,problem.name + ": control", "time (s)", "u", "u");

    plot(t,x,problem.name + ": state", "time (s)", "x", "x",
                            "pdf", "isop_state.pdf");

    plot(t,u,problem.name + ": control", "time (s)", "u", "u",
                            "pdf", "isop_control.pdf");



}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from *PSOPT* is summarized in the text box below and in Figures 3.45 and 3.46, which show the optimal state and control, respectively.

```
PSOPT results summary
=====================

Problem:  Isoperimetric constraint problem
CPU time (seconds): 9.701080e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:21:52 2020

Optimal (unscaled) cost function value:  -3.755058e-01
Phase 1 endpoint cost function value: 0.000000e+00
```

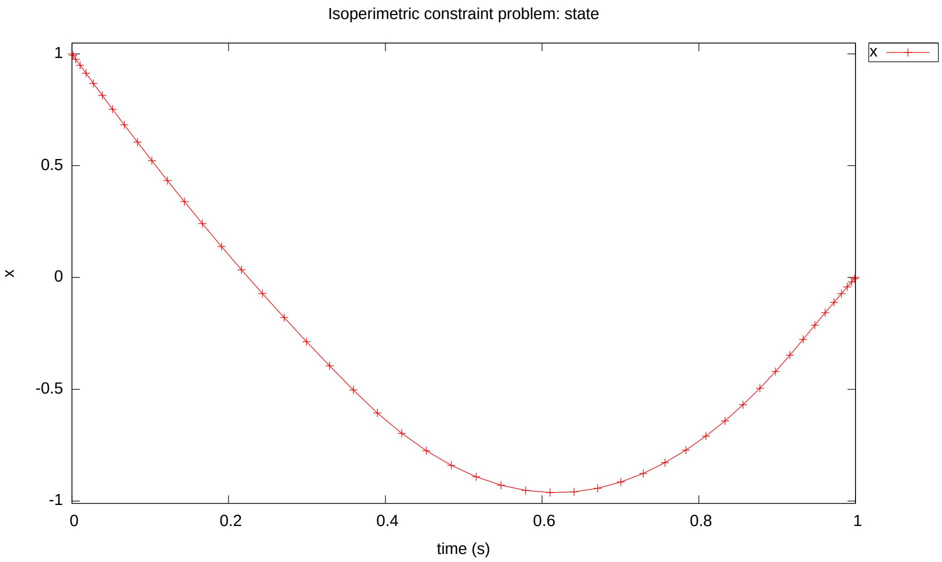

Figure 3.45: State for isoperimetric constraint problem

```
Phase 1 integrated part of the cost:  -3.755058e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 3.106347e-05
NLP solver reports:  The problem has been solved!
```

3.20 Lambert's problem

This example demonstrates the use of the $\mathcal{PSOPT}$ for a classical orbit determination problem, namely the determination of an orbit from two position vectors and time (Lambert's problem) [40]. The problem is formulated as follows. Find $\mathbf{r}(t) \in [0, t_f]$ and $\mathbf{v}(t) \in [0, t_f]$ to minimise:

$$J = 0 \tag{3.84}$$

subject to

$$\begin{aligned} \dot{\mathbf{r}} &= \mathbf{v} \\ \dot{\mathbf{v}} &= -\mu \frac{\mathbf{r}}{||\mathbf{r}||^3} \end{aligned} \tag{3.85}$$

with the boundary conditions:

$$\begin{aligned} \mathbf{r}(\mathbf{0}) &= [15945.34\text{E}3, 0.0, 0.0]^T \\ \mathbf{r}(t_f) &= [12214.83899\text{E}3, 10249.46731\text{E}3, 0.0]^T \end{aligned} \tag{3.86}$$

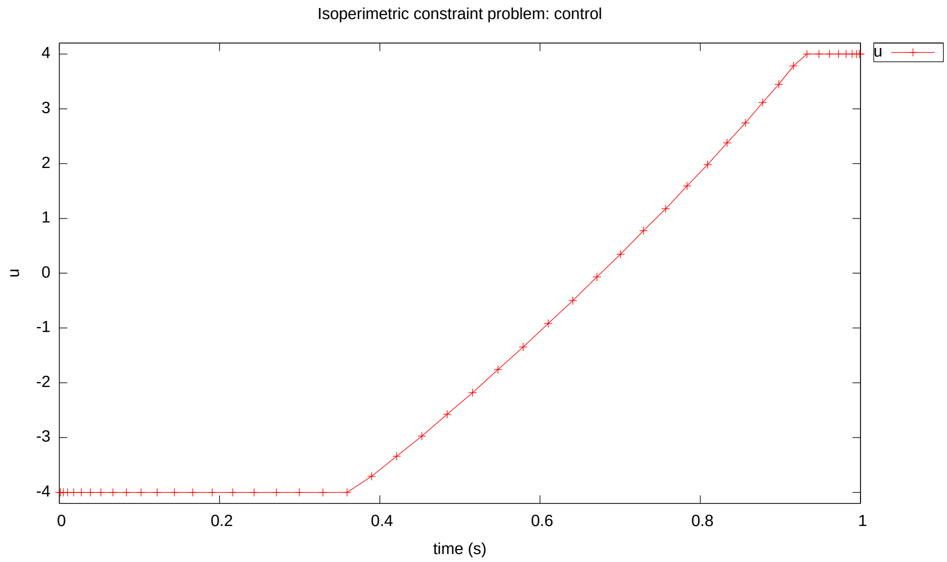

Figure 3.46: Control for isoperimetric constraint problem

where $\mathbf{r} = [x, y, z]^T$ (m) is a cartesian position vector, and $\mathbf{v} = [v_x, v_z, v_z]^T$ is the corresponding velocity vector, $\mu = GM_e$, G ($\text{m}^3/(\text{kg s}^2)$) is the universal gravitational constant and M_e (kg) is the mass of Earth.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
////////////////        lambert.cxx              /////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example             ////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
//////// Title: Lambert problem                           ////////////////
//////// Last modified: 27 December 2009                  ////////////////
//////// Reference:     Vallado (2001), page 470          ////////////////
//////// (See PSOPT handbook for full reference)           ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"


//////////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   return 0.0;
}

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////
```

```
adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    return  0.0;
}


//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

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

   // Define constants:

   double G =   6.6720e-11; // Universal gravity constant [m^3/(kg s^2)]
   double Me =  5.9742e24;   // Mass of earth;[kg]

   double mu   = G*Me;       // [m^3/sec^2]

   adouble r[3];
   adouble v[3];

   // Extract individual variables

   r[0] = states[ 0 ];
   r[1] = states[ 1 ];
   r[2] = states[ 2 ];

   v[0] = states[ 3 ];
   v[1] = states[ 4 ];
   v[2] = states[ 5 ];


   adouble rdd[3];

   adouble rr = sqrt( r[0]*r[0]+r[1]*r[1]+r[2]*r[2] );

   adouble r3 = pow(rr,3.0);

   rdd[0] = -mu*r[0]/r3;
   rdd[1] = -mu*r[1]/r3;
   rdd[2] = -mu*r[2]/r3;

   derivatives[ 0 ] = v[0];
   derivatives[ 1 ] = v[1];
   derivatives[ 2 ] = v[2];
   derivatives[ 3 ] = rdd[0];
   derivatives[ 4 ] = rdd[1];
   derivatives[ 5 ] = rdd[2];

}

////////////////////////////////////////////////////////////////////////////
////////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

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

{

   adouble ri1 = initial_states[ 0 ];
   adouble ri2 = initial_states[ 1 ];
   adouble ri3 = initial_states[ 2 ];




   adouble rf1 = final_states[ 0 ];
   adouble rf2 = final_states[ 1 ];
   adouble rf3 = final_states[ 2 ];

   e[ 0 ] = ri1;
   e[ 1 ] = ri2;
   e[ 2 ] = ri3;

   e[ 3 ] = rf1;
```

```
   e[ 4 ] = rf2;
   e[ 5 ] = rf3;

}



////////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
////////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
 //  auto_link_multiple(linkages, xad, N_PHASES);
}



////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

    MSdata msdata;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name                   = "Lambert problem";
    problem.outfilename                 = "lambert.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

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

    psopt_level1_setup(problem);

/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information &  do level 2 setup ////////////
/////////////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates    = 6;
    problem.phases(1).ncontrols  = 0;
    problem.phases(1).nevents    = 6;
    problem.phases(1).nparameters   = 6;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes         << 100;

    psopt_level2_setup(problem, algorithm);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////

    double r1i = 15945.34e3; // m
    double r2i = 0.0;
    double r3i = 0.0;

    double r1f = 12214.83899e3; //m
    double r2f = 10249.46731e3; //m
    double r3f = 0.0;

    double TF = 76.0*60.0; // seconds

    problem.phases(1).bounds.lower.states(0) = -10*max(r1i,r1f);
    problem.phases(1).bounds.lower.states(1) = -10*max(r2i,r2f);
```

```
    problem.phases(1).bounds.lower.states(2) = -10*max(r3i,r3f);
    problem.phases(1).bounds.upper.states(0) = 10*max(r1i,r1f);
    problem.phases(1).bounds.upper.states(1) = 10*max(r2i,r2f);
    problem.phases(1).bounds.upper.states(2) = 10*max(r3i,r3f);

    problem.phases(1).bounds.lower.states(3) = -10*max(r1i,r1f)/TF;
    problem.phases(1).bounds.lower.states(4) = -10*max(r1i,r1f)/TF;;
    problem.phases(1).bounds.lower.states(5) = -10*max(r1i,r1f)/TF;;
    problem.phases(1).bounds.upper.states(3) = 10*max(r1i,r1f)/TF;
    problem.phases(1).bounds.upper.states(4) = 10*max(r2i,r2f)/TF;
    problem.phases(1).bounds.upper.states(5) = 10*max(r3i,r3f)/TF;


    problem.phases(1).bounds.lower.events(0) = r1i;
    problem.phases(1).bounds.upper.events(0) = r1i;

    problem.phases(1).bounds.lower.events(1) = r2i;
    problem.phases(1).bounds.upper.events(1) = r2i;

    problem.phases(1).bounds.lower.events(2) = r3i;
    problem.phases(1).bounds.upper.events(2) = r3i;

    problem.phases(1).bounds.lower.events(3) = r1f;
    problem.phases(1).bounds.upper.events(3) = r1f;

    problem.phases(1).bounds.lower.events(4) = r2f;
    problem.phases(1).bounds.upper.events(4) = r2f;

    problem.phases(1).bounds.lower.events(5) = r3f;
    problem.phases(1).bounds.upper.events(5) = r3f;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = TF;
    problem.phases(1).bounds.upper.EndTime      = TF;


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem names and units    //////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).name.states(1) = "x position";
    problem.phases(1).name.states(2) = "y position";
    problem.phases(1).name.states(3) = "z position";
    problem.phases(1).name.states(4) = "x velocity";
    problem.phases(1).name.states(5) = "y velocity";
    problem.phases(1).name.states(6) = "z velocity";

    problem.phases(1).units.states(1) = "m";
    problem.phases(1).units.states(2) = "m";
    problem.phases(1).units.states(3) = "m";
    problem.phases(1).units.states(4) = "m";
    problem.phases(1).units.states(5) = "m/s";
    problem.phases(1).units.states(6) = "m/s";




////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost   = &endpoint_cost;
    problem.dae              = &dae;
    problem.events    = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes      = 20;
    int ncontrols                       = problem.phases(1).ncontrols;
    int nstates                         = problem.phases(1).nstates;


    MatrixXd x_guess    =  zeros(nstates,nnodes);
```

```
    MatrixXd time_guess =  linspace(0.0,TF,nnodes);



    x_guess << linspace(r1i,r1f, nnodes),
               linspace(r2i,r2f,nnodes),
      linspace(r3i,r3f,nnodes),
     linspace(r1i,r1f,nnodes)/TF,
     linspace(r2i,r2f,nnodes)/TF,
     linspace(r3i,r3f, nnodes)/TF;


    problem.phases(1).guess.states         = x_guess;
    problem.phases(1).guess.time           = time_guess;


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;
    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.defect_scaling              = "jacobian-based";
    algorithm.collocation_method          = "Hermite-Simpson";




////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////


    MatrixXd x, u, t, xi, ui, ti;

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



////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

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

////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd r1 = x.row(0);
    MatrixXd r2 = x.row(1);
    MatrixXd r3 = x.row(2);

    MatrixXd v1 = x.row(3);
    MatrixXd v2 = x.row(4);
    MatrixXd v3 = x.row(5);

    MatrixXd vi(3,1), vf(3,1);

    vi(0) = v1(0);
    vi(1) = v2(1);
    vi(2) = v3(2);

    vf(0) = v1(length(v1)-1);
    vf(1) = v2(length(v1)-1);
    vf(2) = v3(length(v1)-1);

    Print(vi,"Initial velocity vector [m/s]");
```

```
    Print(vf,"Final velocity vector [m/s]");


    plot(t,r1,problem.name+": x", "time (s)", "x","x");

    plot(t,r2,problem.name+": y", "time (s)", "x","y");

    plot(t,r3,problem.name+": z", "time (s)", "z","z");

    plot(r1,r2,problem.name+": x-y", "x (m)", "y (m)","y");

    plot(r1,r2,problem.name+": x-y trajectory", "x (m)", "y (m)","y", "pdf", "lambert_xy.pdf");

    plot3(r1,r2,r3,problem.name+": trajectory","x (m)", "y (m)", "z (m)");


}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the text box below and in Figure 3.47, which show the trajectory from $\mathbf{r}(0)$ to $\mathbf{r}(t_f)$, respectively.

```
PSOPT results summary
=====================

Problem:  Lambert problem
CPU time (seconds): 1.600679e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:22:06 2020

Optimal (unscaled) cost function value:  0.000000e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 4.560000e+03
Phase 1 maximum relative local error: 3.437571e-05
NLP solver reports:  The problem has been solved!
```

The resulting initial and final velocity vectors are:

$$\begin{aligned} \mathbf{v}(0) &= [2058.902605, 2915.961924, -6.878790137\text{E}-13]^T \\ \mathbf{v}(t_f) &= [-3451.55505, 910.3192974, -6.878787164\text{E}-13]^T \end{aligned} \tag{3.87}$$

3.21 Lee-Ramirez bioreactor

Consider the following optimal control problem, which is known in the literature as the Lee-Ramirez bioreactor [27, 34]. Find t_f and $u(t) \in [0, t_f]$ to minimize the cost functional

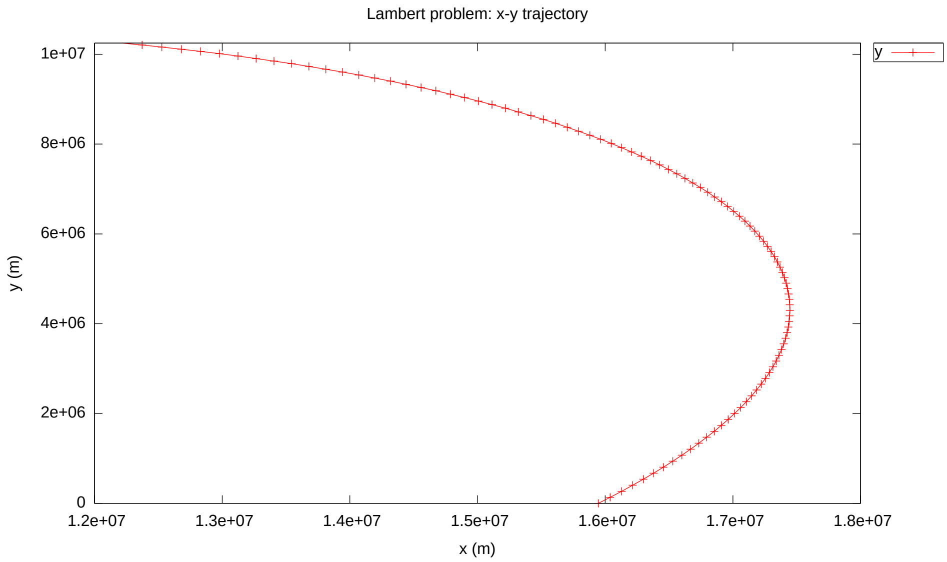

Figure 3.47: Trajectory between the initial and final positions for Lambert's problem

$$J = -x_1(t_f)x_4(t_f) + \int_0^{t_f} \rho[\dot{u}_1(t)^2 + \dot{u}_2(t)^2]dt \tag{3.88}$$

subject to the dynamic constraints

$$\begin{aligned}
\dot{x}_1 &= u_1 + u_2; \\
\dot{x}_2 &= g_1 x_2 - \frac{u_1 + u_2}{x_1} x_2; \\
\dot{x}_3 &= 100\frac{u_1}{x_1} - \frac{u_1 + u_2}{x_1} x_3 - (g_1/0.51) x_2; \\
\dot{x}_4 &= R_{fp} x_2 - \frac{u_1 + u_2}{x_1} x_4; \\
\dot{x}_5 &= 4\frac{u_2}{x_1} - \frac{u_1 + u_2}{x_1} x_5; \\
\dot{x}_6 &= -k_1 x_6; \\
\dot{x}_7 &= k_2(1 - x_7).
\end{aligned} \tag{3.89}$$

where $t_f = 10$, $\rho = 1/N$, and N is the number of discretization nodes,

$$\begin{aligned}
k_1 &= 0.09x_5/(0.034 + x_5); \\
k_2 &= k_1; \\
g_1 &= (x_3/(14.35 + x_3(1.0 + x_3/111.5)))(x_6 + 0.22x_7/(0.22 + x_5)); \\
R_{fp} &= (0.233x_3/(14.35 + x_3(1.0 + x_3/111.5)))((0.0005 + x_5)/(0.022 + x_5));
\end{aligned} \tag{3.90}$$

the initial conditions:

$$\begin{array}{rcl} x_1(0) & = & 1 \\ x_2(0) & = & 0.1 \\ x_3(0) & = & 40 \\ x_4(0) & = & 0 \\ x_5(0) & = & 0 \\ x_6(0) & = & 1.0 \\ x_7(0) & = & 0 \end{array} \tag{3.91}$$

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.48 and 3.49, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Lee-Ramirez bioreactor
CPU time (seconds): 2.640299e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:12:03 2020

Optimal (unscaled) cost function value:  -6.163108e+00
Phase 1 endpoint cost function value: -6.166014e+00
Phase 1 integrated part of the cost:  2.905549e-03
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+01
Phase 1 maximum relative local error: 8.617937e-02
NLP solver reports:  The problem has been solved!
```

3.22 Li's parameter estimation problem

This is a parameter estimation problem with two parameters and three observed variables, which is presented b Li *et. al* [26].

The dynamic equations are given by:

$$\frac{dx}{dt} = M(t,p)x + f(t), \ t \in [0,\pi] \tag{3.92}$$

with boundary condition:

$$x(0) + x(\pi) = (1 + e^{\pi})\,[1,\,1,\,1]^T$$

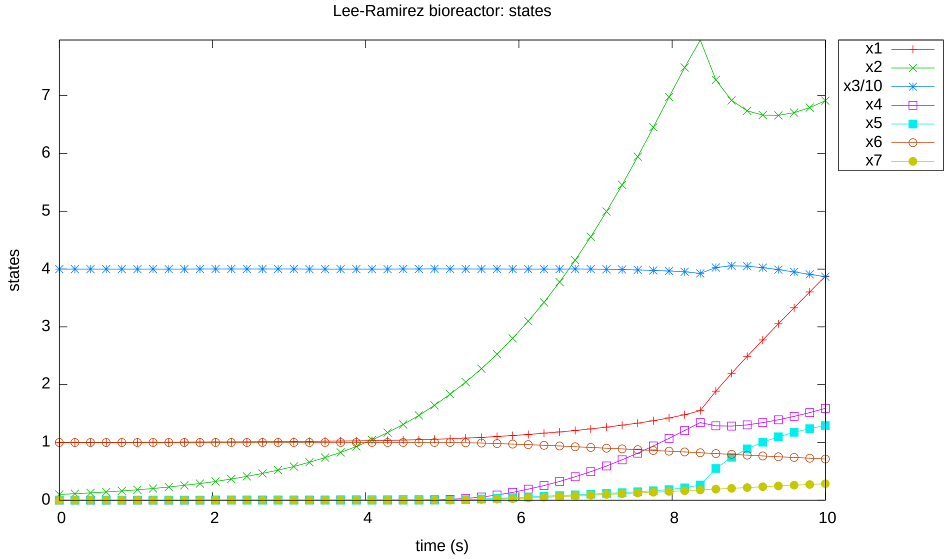

Figure 3.48: States for the Lee-Ramirez bioreactor problem

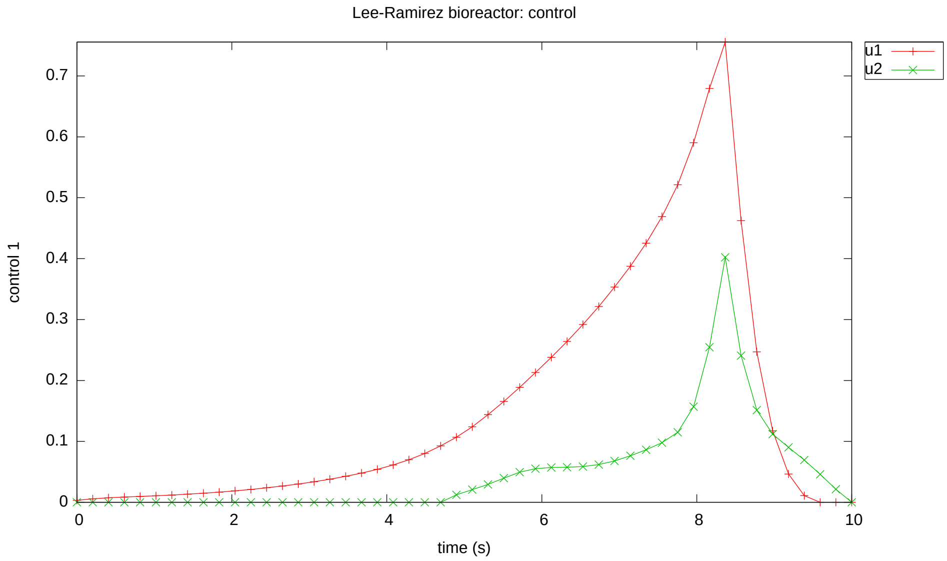

Figure 3.49: Control for the Lee-Ramirez bioreactor problem

Table 3.2: Estimated parameter values and 95 percent statistical confidence limits on estimated parameters

Parameter	Low Confidence Limit	Value	High Confidence Limit
p_1	1.907055e+01	1.907712e+01	1.908369e+01
p_2	9.984900e-01	9.984990e-01	9.985080e-01

where

$$M(t,p) = \begin{bmatrix} p_2 - p_1\cos(p_2 t) & 0 & p_2 + p_1\sin(p_2 t) \\ 0 & p_1 & 0 \\ -p_2 + p_1\sin(p_2 t) & 0 & p_2 + p_1\cos(p_2 t) \end{bmatrix} \tag{3.93}$$

and

$$f(t) = \begin{bmatrix} -1 + 19(\cos(t) - \sin(t)) \\ -18 \\ 1 - 19(\cos(t) + \sin(t)) \end{bmatrix} \tag{3.94}$$

and the observation functions are:

$$\begin{aligned} g_1 &= x_1 \\ g_2 &= x_2 \\ g_3 &= x_3 \end{aligned} \tag{3.95}$$

The trajectories of the dynamic system is characterised by rapidly varying fast and slow components if the difference between the two parameters p_1 and p_2 is large, which may cause numerical problems to some ODE solvers.

The estimation data set is generated by adding Gaussian noise with standard deviation 1 around the solution $[x_1(t), x_2(t), x_3(t)]^T = [e^t,\, e^t,\, e^t]^T$, with $N = 33$ equidistant samples within the interval $t = [0,\, \pi]$. The true values of the parameters are $p_1 = 19$ and $p_2 = 1$. The weights of the three observations are the same and equal to one.

The solution is found using Legendre discretisation with 40 grid points. The estimated parameter values and their 95% confidence limits for $n_s = 129$ samples are shown in Table 3.22. Figure 3.50 shows the observations as well as the estimated values of variable x_1.

3.23 Linear tangent steering problem

Consider the following optimal control problem, which is known in the literature as the linear tangent steering problem [3]. Find t_f and $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = t_f \tag{3.96}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= a\cos(u) \\ \dot{x}_3 &= x_4 \\ \dot{x}_4 &= a\sin(u) \end{aligned} \tag{3.97}$$

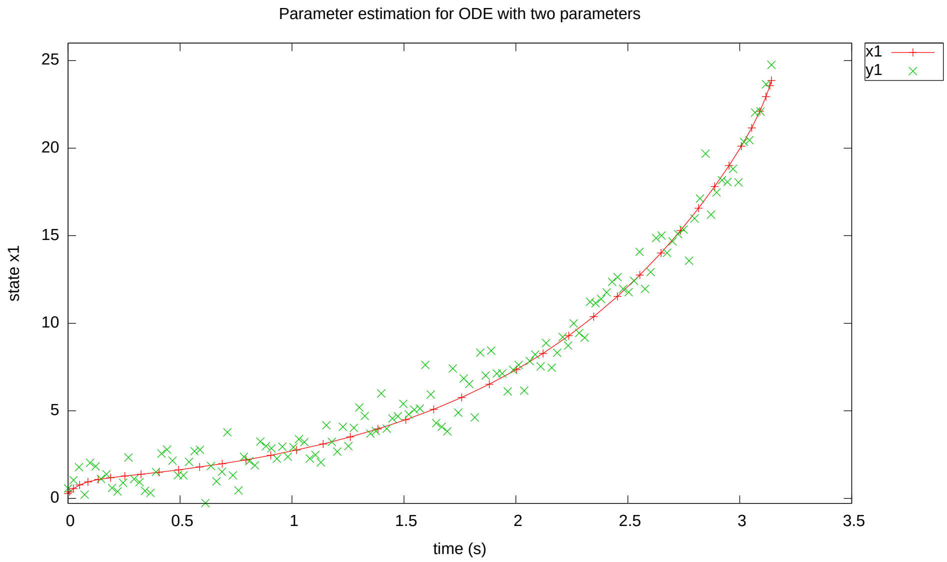

Figure 3.50: Observations and estimated state $x_1(t)$

the boundary conditions:

$$
\begin{aligned}
x_1(0) &= 0 \\
x_2(0) &= 0 \\
x_3(0) &= 0 \\
x_4(0) &= 0 \\
x_2(t_f) &= 45.0 \\
x_3(t_f) &= 5.0 \\
x_4(t_f) &= 0.0
\end{aligned}
\tag{3.98}
$$

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.51 and 3.52, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Linear Tangent Steering Problem
CPU time (seconds): 7.697350e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:23:17 2020

Optimal (unscaled) cost function value:  5.545709e-01
Phase 1 endpoint cost function value: 5.545709e-01
```

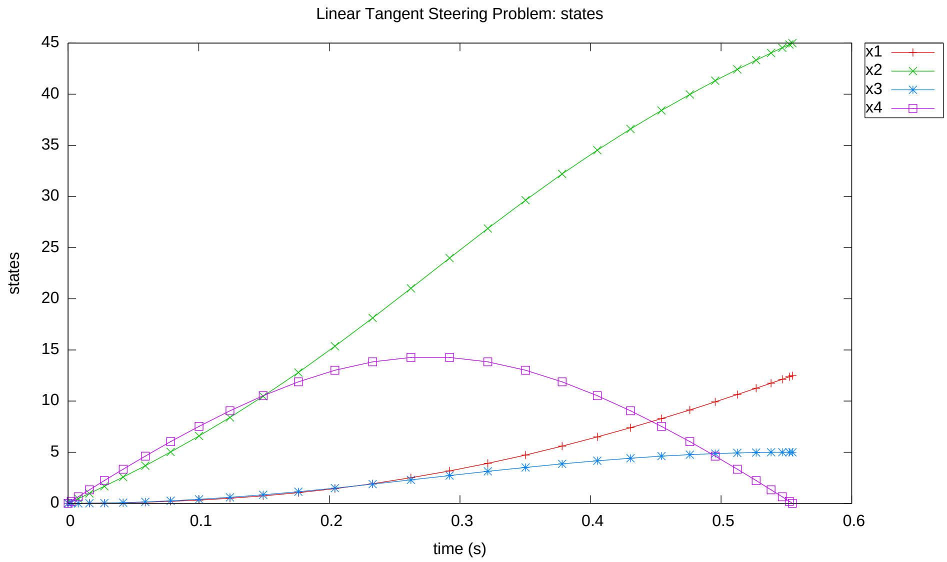

Figure 3.51: States for the linear tangent steering problem

```
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.545709e-01
Phase 1 maximum relative local error: 1.842632e-07
NLP solver reports:  The problem has been solved!
```

3.24 Low thrust orbit transfer

The goal of this problem is to compute an optimal low thrust policy for an spacecraft to go from a standard space shuttle park orbit to a specified final orbit, while maximising the final weight of the spacecraft. The problem is described in detail by Betts [3]. The problem is formulated as follows. Find $\mathbf{u}(t) = [u_r(t), u_\theta(t), u_h(t)]^T, t \in [0, t_f]$, the unknown throtle parameter τ, and the final time t_f, such that the following objective function is minimised:

$$J = -w(t_f) \tag{3.99}$$

subject to the dynamic constraints:

$$\begin{aligned} \dot{\mathbf{y}} &= \mathbf{A}(\mathbf{y})\Delta + \mathbf{b} \\ \dot{w} &= -T[1 + 0.01\tau]/I_{sp} \end{aligned} \tag{3.100}$$

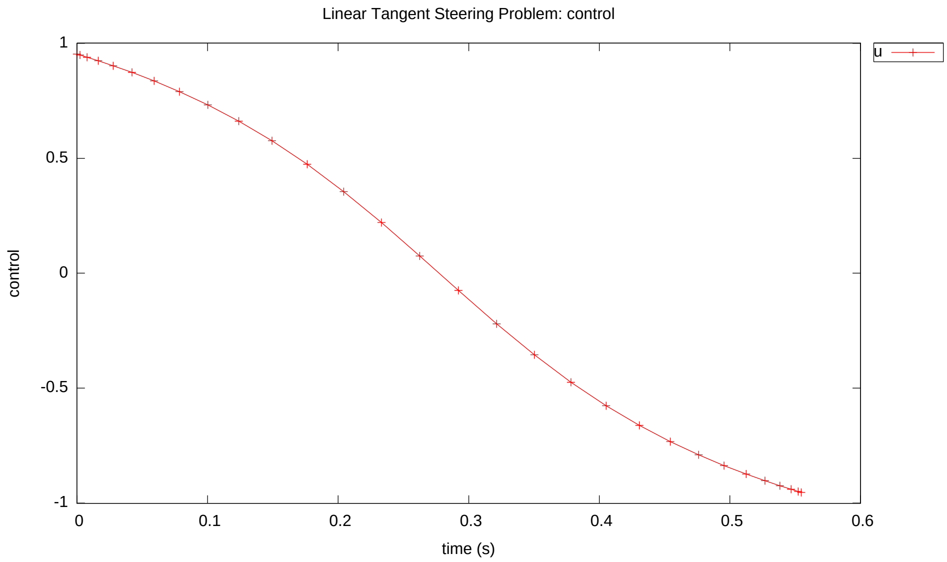

Figure 3.52: Control for the linear tangent steering problem

the path constraint:

$$||u(t)||^2 = 1 \tag{3.101}$$

and the parameter bounds:

$$\tau_L \leq \tau \leq 0 \tag{3.102}$$

where $\mathbf{y} = [p, f, g, h, k, L, w]^T$ is the vector of modified equinoctial elements, $w(t)$ is the weight of the spacecraft, I_{sp} is the specific impulse of the engine, expressions for $\mathbf{A}(\mathbf{y})$ and $\mathbf{b}$ are given in [3], the disturbing acceleration Δ is given by:

$$\Delta = \Delta_g + \Delta_T \tag{3.103}$$

where Δ_g is the gravitational disturbing acceleration due to the oblatness of Earth (given in [3]), and Δ_T is the thurst acceleration, given by:

$$\Delta_T = \frac{g_0 T[1 + 0.01\tau]}{w}\mathbf{u}$$

where T is the maximum thrust, and g_0 is the mass to weight conversion factor.

The boundary conditions of the problem are given by:

$$\begin{aligned}
p(t_f) &= 40007346.015232 \text{ ft} \\
\sqrt{f(t_f)^2 + g(t_f)^2} &= 0.73550320568829 \\
\sqrt{h(t_f)^2 + k(t_f)^2} &= 0.61761258786099 \\
f(t_f)h(t_f) + g(t_f)k(t_f) &= 0 \\
g(t_f)h(t_f) - k(t_f)f(t_f) &= 0 \\
p(0) &= 21837080.052835\text{ft} \\
f(0) &= 0 \\
g(0) &= 0 \\
h(0) &= 0 \\
h(0) &= 0 \\
k(0) &= 0 \\
L(0) &= \pi \text{ (rad)} \\
w(0) &= 1 \text{ (lb)}
\end{aligned} \tag{3.104}$$

and the values of the parameters are: $g_0 = 32.174$ (ft/sec^2), $I_{sp} = 450$ (sec), $T = 4.446618 \times 10^{-3}$ (lb), $\mu = 1.407645794 \times 10^{16}$ (ft^3/sec^2), $R_e = 20925662.73$ (ft), $J_2 = 1082.639 \times 10^{-6}$, $J_3 = -2.565 \times 10^{-6}$, $J_4 = -1.608 \times 10^{-6}$, $\tau_L = -50$.

An initial guess was computed by forward propagation from the initial conditions, assuming that the direction of the thrust vector is parallel to the cartesian velocity vector, such that the initial control input was computed as follows:

$$\mathbf{u}(t) = \mathbf{Q}_r^T \frac{\mathbf{v}}{||\mathbf{v}||} \tag{3.105}$$

where $\mathbf{Q}_r$ is a matrix whose columns are the directions of the rotating radial frame:

$$\mathbf{Q}_r = \begin{bmatrix} \mathbf{i}_r & \mathbf{i}_\theta & \mathbf{i}_h \end{bmatrix} = \begin{bmatrix} \frac{\mathbf{r}}{||\mathbf{r}||} & \frac{(\mathbf{r}\times\mathbf{v})\times\mathbf{r}}{||\mathbf{r}\times\mathbf{v}||||\mathbf{r}||} & \frac{(\mathbf{r}\times\mathbf{v})}{||\mathbf{r}\times\mathbf{v}||} \end{bmatrix} \tag{3.106}$$

The problem was solved using local collocation (trapezoidal followed by Hermite-Simpson) with automatic mesh refinement. The $\mathcal{PSOPT}$ code that solves the problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
////////////////        low_thrust.cxx              //////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example           /////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title: Low thrust orbit transfer problem         ////////////////
//////// Last modified: 16 February 2009                  ////////////////
//////// Reference:     Betts  (2001)                ////////////////
//////// (See PSOPT handbook for full reference)           ///////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
```

```
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                    ///////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"



//////////////////////////////////////////////////////////////////////////
//////////////////  Define auxiliary functions                 //////////
//////////////////////////////////////////////////////////////////////////

adouble legendre_polynomial( adouble x, int n)
{
// This function computes the value of the legendre polynomials
// for a given value of the argument x and for n=0...5 only

  adouble retval=0.0;

  switch(n) {
     case 0:
         retval=1.0; break;
     case 1:
         retval= x; break;
     case 2:
         retval= 0.5*(3.0*pow(x,2)-1.0); break;
     case 3:
         retval= 0.5*(5.0*pow(x,3)- 3*x); break;
     case 4:
         retval= (1.0/8.0)*(35.0*pow(x,4) - 30.0*pow(x,2) + 3.0); break;
     case 5:
         retval= (1.0/8.0)*(63.0*pow(x,5) - 70.0*pow(x,3) + 15.0*x); break;
     default:
         error_message("legendre_polynomial(x,n) is limited to n=0...5");
  }

  return retval;

}

adouble legendre_polynomial_derivative( adouble x, int n)
{
// This function computes the value of the legendre polynomial derivatives
// for a given value of the argument x and for n=0...5 only.

  adouble retval=0.0;

  switch(n) {
     case 0:
         retval=0.0; break;
     case 1:
         retval= 1.0; break;
     case 2:
         retval= 0.5*(2.0*3.0*x); break;
     case 3:
         retval= 0.5*(3.0*5.0*pow(x,2)-3.0); break;
     case 4:
         retval= (1.0/8.0)*(4.0*35.0*pow(x,3) - 2.0*30.0*x ); break;
     case 5:
         retval= (1.0/8.0)*(5.0*63.0*pow(x,4) - 3.0*70.0*pow(x,2) + 15.0); break;
     default:
         error_message("legendre_polynomial_derivative(x,n) is limited to n=0...5");
  }

  return retval;

}


void compute_cartesian_trajectory(const MatrixXd& x, MatrixXd& xyz )
{

   int npoints = x.cols();
   xyz.resize(3,npoints);

   for(int i=0; i<npoints;i++) {


   double p = x(0,i);
   double f = x(1,i);
   double g = x(2,i);
```

```
   double h = x(3,i);
   double k = x(4,i);
   double L = x(5,i);

   double q      =  1.0 + f*cos(L) + g*sin(L);
   double r      =  p/q;
   double alpha2 = h*h - k*k;
   double X      = sqrt( h*h + k*k );
   double s2     = 1 + X*X;

   double r1 = r/s2*( cos(L) + alpha2*cos(L) + 2*h*k*sin(L));
   double r2 = r/s2*( sin(L) - alpha2*sin(L) + 2*h*k*cos(L));
   double r3 = 2*r/s2*( h*sin(L) - k*cos(L) );

   xyz(0,i) = r1;
   xyz(1,i) = r2;
   xyz(2,i) = r3;

   }

}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
////////////////////////////////////////////////////////////////////////////

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

   if (iphase == 1) {
   adouble w = final_states[6];
   return (-w);
   }
   else {
   return (0);
   }
}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
////////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    return  0.0;
}


////////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   // Local integers
   int i, j;
   // Define constants:
   double Isp  = 450.0;            // [sec]
   double mu   = 1.407645794e16;   // [f2^2/sec^2]
   double g0   = 32.174;           // [ft/sec^2]
   double T    = 4.446618e-3;      // [lb]
   double Re   = 20925662.73;      // [ft]
   double J[5];
   J[2] =  1082.639e-6;
   J[3] = -2.565e-6;
   J[4] = -1.608e-6;

   // Extract individual variables

   adouble p = states[ 0 ];
   adouble f = states[ 1 ];
   adouble g = states[ 2 ];
   adouble h = states[ 3 ];
   adouble k = states[ 4 ];
   adouble L = states[ 5 ];
   adouble w = states[ 6 ];
```

```
adouble* u  = controls;

adouble tau  = parameters[ 0 ];

// Define some dependent variables

adouble q      =  1.0 + f*cos(L) + g*sin(L);
adouble r      =  p/q;
adouble alpha2 = h*h - k*k;
adouble X      = sqrt( h*h + k*k );
adouble s2     = 1 + X*X;

// r and v

adouble r1 = r/s2*( cos(L) + alpha2*cos(L) + 2*h*k*sin(L));
adouble r2 = r/s2*( sin(L) - alpha2*sin(L) + 2*h*k*cos(L));
adouble r3 = 2*r/s2*( h*sin(L) - k*cos(L) );

adouble rvec[3];

rvec[ 0 ] = r1; rvec[ 1] = r2; rvec[ 2 ] = r3;

adouble v1 = -(1.0/s2)*sqrt(mu/p)*(  sin(L) + alpha2*sin(L) - 2*h*k*cos(L) + g - 2*f*h*k + alpha2*g);
adouble v2 = -(1.0/s2)*sqrt(mu/p)*( -cos(L) + alpha2*cos(L) + 2*h*k*sin(L) - f + 2*g*h*k + alpha2*f);
adouble v3 =  (2.0/s2)*sqrt(mu/p)*(h*cos(L) +      k*sin(L) + f*h + g*k);

adouble vvec[3];

vvec[ 0 ] = v1; vvec[ 1 ] = v2; vvec[ 2 ] = v3;

// compute Qr

adouble ir[3], ith[3], ih[3];
adouble rv[3];
adouble rvr[3];

cross( rvec, vvec, rv );

cross( rv, rvec, rvr );

adouble norm_r = sqrt( dot(rvec, rvec, 3) );

adouble norm_rv = sqrt( dot(rv, rv, 3) );

for (i=0; i<3; i++) {

   ir[i]  = rvec[i]/norm_r;

   ith[i] = rvr[i]/( norm_rv*norm_r );

   ih[i]  = rv[i]/norm_rv;

}

adouble Qr1[3], Qr2[3], Qr3[3];

for(i=0; i< 3; i++)
{
     // Columns of matrix Qr
     Qr1[i] = ir[i];
     Qr2[i] = ith[i];
     Qr3[i] = ih[i];
}

// Compute in

adouble en[3];
en[ 0 ] = 0.0; en[ 1 ]= 0.0; en[ 2 ] = 1.0;

adouble enir = dot(en,ir,3);

adouble in[3];

for(i=0;i<3;i++) {
   in[i] = en[i] - enir*ir[i];
}

adouble norm_in = sqrt( dot( in, in, 3 ) );

for(i=0;i<3;i++) {
   in[i] = in[i]/norm_in;
}
```

```
   // Geocentric latitude angle:

   adouble sin_phi =  rvec[ 2 ]/ sqrt( dot(rvec,rvec,3) ) ;
   adouble cos_phi =  sqrt(1.0- pow(sin_phi,2.0));

   adouble deltagn = 0.0;
   adouble deltagr = 0.0;
   for (j=2; j<=4;j++) {
     adouble Pdash_j = legendre_polynomial_derivative( sin_phi, j );
     adouble P_j     = legendre_polynomial( sin_phi, j );
     deltagn += -mu*cos_phi/(r*r)*pow(Re/r,j)*Pdash_j*J[j];
     deltagr += -mu/(r*r)* (j+1)*pow( Re/r,j)*P_j*J[j];
   }

   // Compute vector delta_g

   adouble delta_g[3];
   for (i=0;i<3;i++) {
       delta_g[i] = deltagn*in[i] - deltagr*ir[i];
   }

   // Compute vector DELTA_g

   adouble DELTA_g[3];

   DELTA_g[ 0 ] = dot(Qr1, delta_g,3);
   DELTA_g[ 1 ] = dot(Qr2, delta_g,3);
   DELTA_g[ 2 ] = dot(Qr3, delta_g,3);

   // Compute DELTA_T

   adouble DELTA_T[3];



   for(i=0;i<3;i++) {
      DELTA_T[i] =  g0*T*(1.0+0.01*tau)*u[i]/w;
   }

   // Compute DELTA

   adouble DELTA[3];

   for(i=0;i<3;i++) {
      DELTA[i] =  DELTA_g[i] + DELTA_T[i];
   }

   adouble delta1= DELTA[ 0 ];
   adouble delta2= DELTA[ 1 ];
   adouble delta3= DELTA[ 2 ];


  // derivatives

   adouble pdot =                                  2*p/q*sqrt(p/mu)              * delta2;
   adouble fdot =  sqrt(p/mu)*sin(L) * delta1 + sqrt(p/mu)*(1.0/q)*((q+1.0)*cos(L)+f) * delta2
                -  sqrt(p/mu)*(g/q)*(h*sin(L)-k*cos(L))  * delta3;
   adouble gdot = -sqrt(p/mu)*cos(L) * delta1 + sqrt(p/mu)*(1.0/q)*((q+1.0)*sin(L)+g) * delta2
                +  sqrt(p/mu)*(f/q)*(h*sin(L)-k*cos(L))  * delta3;
   adouble hdot =  sqrt(p/mu)*s2*cos(L)/(2.0*q)            * delta3;
   adouble kdot =  sqrt(p/mu)*s2*sin(L)/(2.0*q)            * delta3;
   adouble Ldot =  sqrt(p/mu)*(1.0/q)*(h*sin(L)-k*cos(L))* delta3     + sqrt(mu*p)*pow( (q/p),2.);

   adouble wdot = -T*(1.0+0.01*tau)/Isp;

   derivatives[ 0 ] = pdot;
   derivatives[ 1 ] = fdot;
   derivatives[ 2 ] = gdot;
   derivatives[ 3 ] = hdot;
   derivatives[ 4 ] = kdot;
   derivatives[ 5 ] = Ldot;
   derivatives[ 6 ] = wdot;

   path[ 0 ] = pow( u[0] , 2)  + pow( u[1], 2) + pow( u[2], 2);

}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
```

```
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{


   int offset;

   adouble pti = initial_states[ 0 ];
   adouble fti = initial_states[ 1 ];
   adouble gti = initial_states[ 2 ];
   adouble hti = initial_states[ 3 ];
   adouble kti = initial_states[ 4 ];
   adouble Lti = initial_states[ 5 ];
   adouble wti = initial_states[ 6 ];


   adouble ptf = final_states[ 0 ];
   adouble ftf = final_states[ 1 ];
   adouble gtf = final_states[ 2 ];
   adouble htf = final_states[ 3 ];
   adouble ktf = final_states[ 4 ];
   adouble Ltf = final_states[ 5 ];


   if (iphase==1) {
    e[ 0 ]  = pti;
    e[ 1 ]  = fti;
    e[ 2 ]  = gti;
    e[ 3 ]  = hti;
    e[ 4 ]  = kti;
    e[ 5 ]  = Lti;
    e[ 6 ]  = wti;
   }

   if (1 == 1) offset = 7;
   else offset = 0;

   if (iphase == 1 ) {
    e[ offset + 0 ]  = ptf;
    e[ offset + 1 ]  = sqrt( ftf*ftf + gtf*gtf );
    e[ offset + 2 ]  = sqrt( htf*htf + ktf*ktf );
    e[ offset + 3 ] = ftf*htf + gtf*ktf;
    e[ offset + 4 ] = gtf*htf - ktf*ftf;
   }

}



///////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
 //  auto_link_multiple(linkages, xad, 1);
}



////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

    MSdata msdata;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

```
    problem.name          = "Low thrust transfer problem";
    problem.outfilename                 = "lowthrust.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

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

    psopt_level1_setup(problem);

/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 7;
    problem.phases(1).ncontrols  = 3;
    problem.phases(1).nparameters         = 1;
    problem.phases(1).nevents             = 12;
    problem.phases(1).npath      = 1;
    problem.phases(1).nodes               << 80;

    psopt_level2_setup(problem, algorithm);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////

    double tauL = -50.0;
    double tauU =   0.0;


    double pti = 21837080.052835;
    double fti = 0.0;
    double gti = 0.0;
    double hti = -0.25396764647494;
    double kti = 0.0;
    double Lti = pi;
    double wti = 1.0;

    double wtf_guess;

    double  SISP = 450.0;
    double  DELTAV = 22741.1460;
    double  CM2W = 32.174;

    wtf_guess  = wti*exp(-DELTAV/(CM2W*SISP));

    double ptf            = 40007346.015232;
    double event_final_9  = 0.73550320568829;
    double event_final_10  = 0.61761258786099;
    double event_final_11 = 0.0;
    double event_final_12_upper = 0.0;
    double event_final_12_lower = -10.0;



    problem.phases(1).bounds.lower.parameters <<  tauL;

    problem.phases(1).bounds.upper.parameters <<  tauU;

    problem.phases(1).bounds.lower.states << 10.e6, -0.20, -0.10, -1.0, -0.20, pi, 0.0;

    problem.phases(1).bounds.upper.states << 60.e6, 0.20, 1.0, 1.0, 0.20, 20*pi, 2.0;

    problem.phases(1).bounds.lower.controls << -1.0, -1.0, -1.0;

    problem.phases(1).bounds.upper.controls << 1.0, 1.0, 1.0;

    problem.phases(1).bounds.lower.events << pti, fti, gti, hti, kti, Lti, wti, ptf, event_final_9, event_final_10, event_final_11, event_final_12_lower;

    problem.phases(1).bounds.upper.events << pti, fti, gti, hti, kti, Lti, wti, ptf, event_final_9, event_final_10, event_final_11, event_final_12_upper;

    double EQ_TOL = 0.001;

    problem.phases(1).bounds.upper.path << 1.0+EQ_TOL;
    problem.phases(1).bounds.lower.path << 1.0-EQ_TOL;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
```

```
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 50000.0;
    problem.phases(1).bounds.upper.EndTime      = 100000.0;


////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae             = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes     = 141;
    int ncontrols          = problem.phases(1).ncontrols;
    int nstates            = problem.phases(1).nstates;

    MatrixXd u_guess    =  zeros(ncontrols,nnodes);
    MatrixXd x_guess    =  zeros(nstates,nnodes);
    MatrixXd time_guess =  linspace(0.0,86810.0,nnodes);

    MatrixXd param_guess = -25.0*ones(1,1);

    u_guess    = load_data("../../../examples/low_thrust/U0.dat",ncontrols, nnodes );
    x_guess    = load_data("../../../examples/low_thrust/X0.dat",nstates  , nnodes );
    time_guess = load_data("../../../examples/low_thrust/T0.dat",1        , nnodes );

    auto_phase_guess(problem, u_guess, x_guess, param_guess, time_guess);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;
    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.defect_scaling              = "jacobian-based";
    algorithm.jac_sparsity_ratio          =  0.11; // 0.05;
    algorithm.collocation_method          = "trapezoidal";
    algorithm.mesh_refinement             = "automatic";
    algorithm.mr_max_increment_factor     = 0.2;




////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////


    MatrixXd x, u, t;

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


    t = t/3600.0;


    MatrixXd tau = solution.get_parameters_in_phase(1);

    Print(tau,"tau");
```

```
////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

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

////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd x1 = x.row(0)/1.e6;
    MatrixXd x2 = x.row(1);
    MatrixXd x3 = x.row(2);
    MatrixXd x4 = x.row(3);
    MatrixXd x5 = x.row(4);
    MatrixXd x6 = x.row(5);
    MatrixXd x7 = x.row(6);
    MatrixXd u1 = u.row(0);
    MatrixXd u2 = u.row(1);
    MatrixXd u3 = u.row(2);

    plot(t,x1,problem.name+": states", "time (h)", "p (1000000 ft)","p (1000000 ft)");

    plot(t,x2,problem.name+": states", "time (h)", "f","f");

    plot(t,x3,problem.name+": states", "time (h)", "g","g");

    plot(t,x4,problem.name+": states", "time (h)", "h","h");

    plot(t,x5,problem.name+": states", "time (h)", "k","k");

    plot(t,x6,problem.name+": states", "time (h)", "L (rev)","L (rev)");

    plot(t,x7,problem.name+": states", "time (h)", "w (lb)","w (lb)");

    plot(t,u1,problem.name+": controls","time (h)", "ur", "ur");

    plot(t,u2,problem.name+": controls","time (h)", "ut", "ut");

    plot(t,u3,problem.name+": controls","time (h)", "uh", "uh");

    plot(t,x1,problem.name+": states", "time (h)", "p (1000000 ft)","p (1000000 ft)",
         "pdf","lowthr_x1.pdf");

    plot(t,x2,problem.name+": states", "time (h)", "f","f",
   "pdf","lowthr_x2.pdf");

    plot(t,x3,problem.name+": states", "time (h)", "g","g",
   "pdf","lowthr_x3.pdf");

    plot(t,x4,problem.name+": states", "time (h)", "h","h",
   "pdf","lowthr_x4.pdf");

    plot(t,x5,problem.name+": states", "time (h)", "k","k",
             "pdf","lowthr_x5.pdf");

    plot(t,x6,problem.name+": states", "time (h)", "L (rev)","L (rev)",
   "pdf","lowthr_x6.pdf");

    plot(t,x7,problem.name+": states", "time (h)", "w (lb)","w (lb)",
   "pdf","lowthr_x7.pdf");


    plot(t,u1,problem.name+": controls","time (h)", "ur", "ur",
   "pdf","lowthr_u1.pdf");

    plot(t,u2,problem.name+": controls","time (h)", "ut", "ut",
   "pdf","lowthr_u2.pdf");

    plot(t,u3,problem.name+": controls","time (h)", "uh", "uh",
   "pdf","lowthr_u3.pdf");

    MatrixXd r;

    compute_cartesian_trajectory(x,r);

    double ft2km = 0.0003048;

    r = r*ft2km;
```

```
    MatrixXd rnew, tnew;

    tnew = linspace(0.0, t(length(t)-1), 1000);

    resample_trajectory( rnew, tnew, r, t );


    plot3(rnew.row(0), rnew.row(1), rnew.row(2),
        "Low thrust transfer trajectory", "x (km)", "y (km)", "z (km)",
         NULL, NULL, "30,97");

    plot3(rnew.row(0), rnew.row(1), rnew.row(2),
       "Low thrust transfer trajectory", "x (km)", "y (km)", "z (km)",
       "pdf", "trajectory.pdf", "30,97");


}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from *PSOPT* is summarised in the box below and shown in Figures 3.53, to 3.58 and 3.59 to 3.61, which contain the modified equinoctial elements and the controls, respectively.

```
PSOPT results summary
=====================

Problem:  Low thrust transfer problem
CPU time (seconds): 2.407956e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:23:00 2020

Optimal (unscaled) cost function value:  -2.203403e-01
Phase 1 endpoint cost function value: -2.203403e-01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 8.684257e+04
Phase 1 maximum relative local error: 3.513503e-04
NLP solver reports:  The problem has been solved!
```

3.25 Manutec R3 robot

The DLR model 2 of the Manutec r3 robot, reported and validated by Otter and co-workers [29, 19], describes the motion of three links of the robot as a function of the control input signals of the robot drive:

Table 3.3: Mesh refinement statistics: Low thrust transfer problem

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	ϵ_{max}	CPU_a
1	TRP	80	803	653	333	333	187	0	52947	2.207e-03	3.981e+00
2	TRP	96	963	781	124	125	119	0	23875	2.266e-03	2.492e+00
3	H-S	107	1391	975	117	118	112	0	37642	1.179e-03	4.279e+00
4	H-S	115	1495	1047	129	130	125	0	44590	3.514e-04	4.654e+00
CPU_b	-	-	-	-	-	-	-	-	-	-	8.675e+00
-	-	-	-	-	703	706	543	0	159054	-	2.408e+01

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, ϵ_{max} = maximum relative ODE error, CPU_a = CPU time in seconds spent by NLP algorithm, CPU_b = additional CPU time in seconds spent by PSOPT

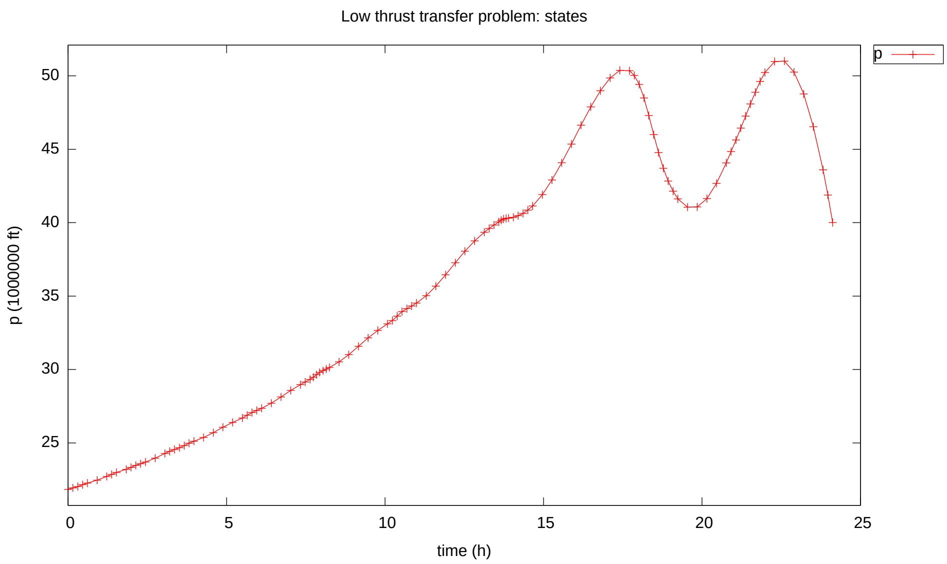

Figure 3.53: Modified equinoctial element p

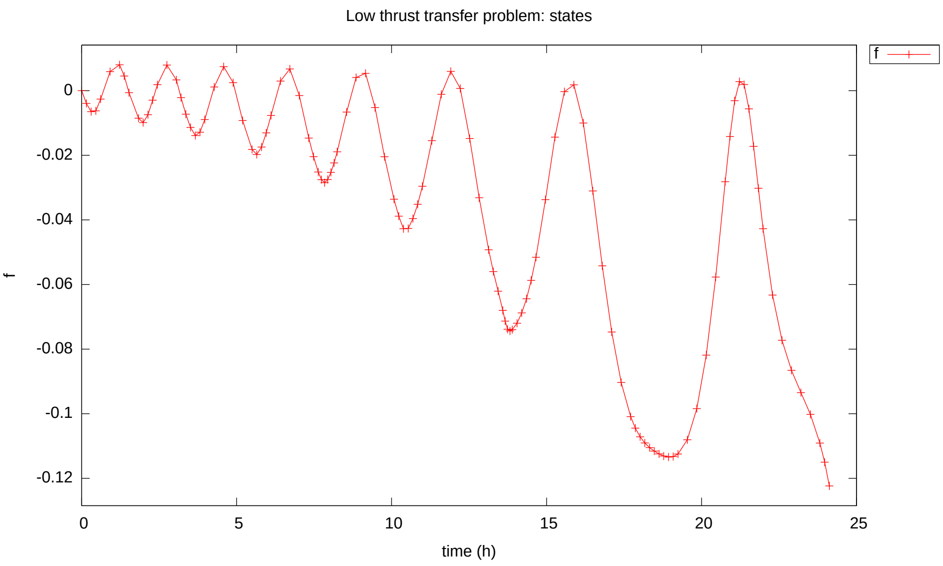

Figure 3.54: Modified equinoctial element f

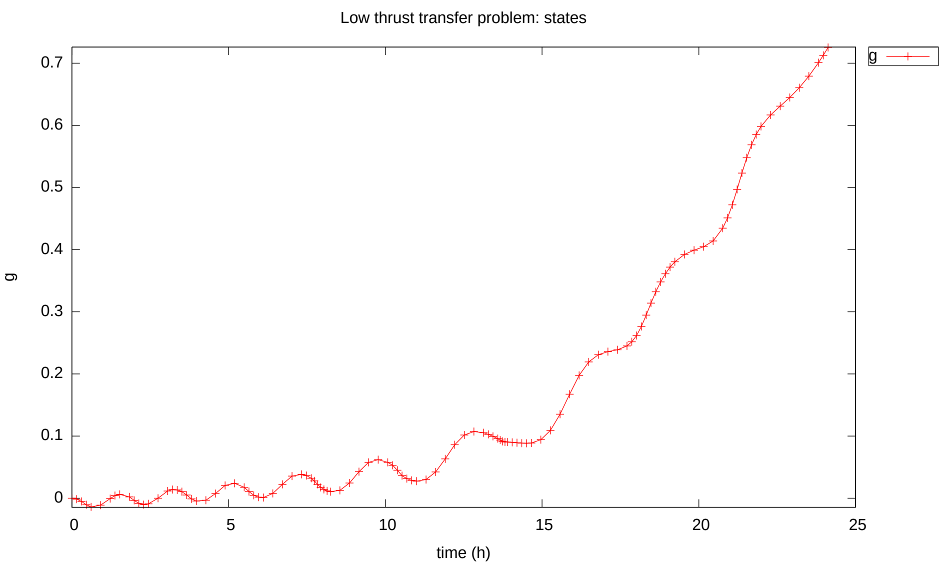

Figure 3.55: Modified equinoctial element g

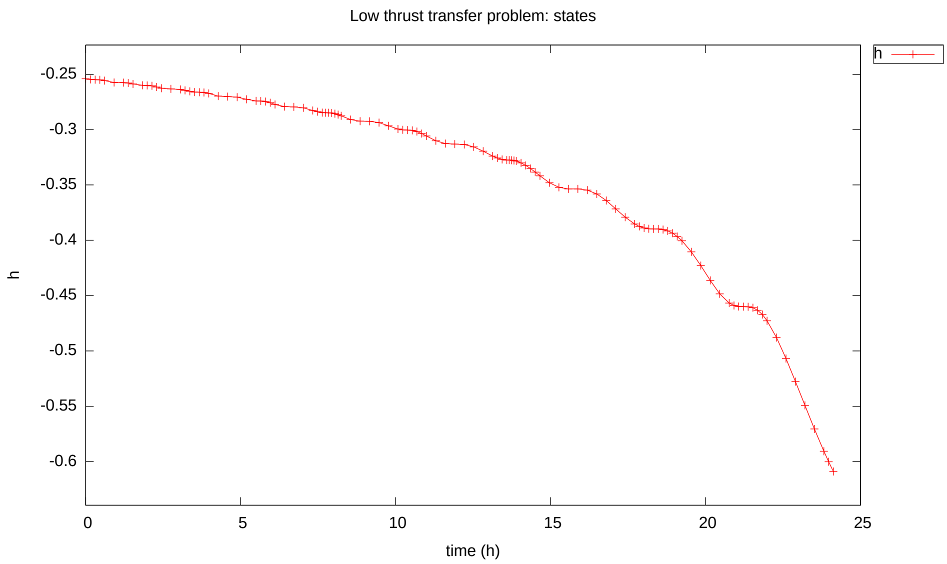

Figure 3.56: Modified equinoctial element h

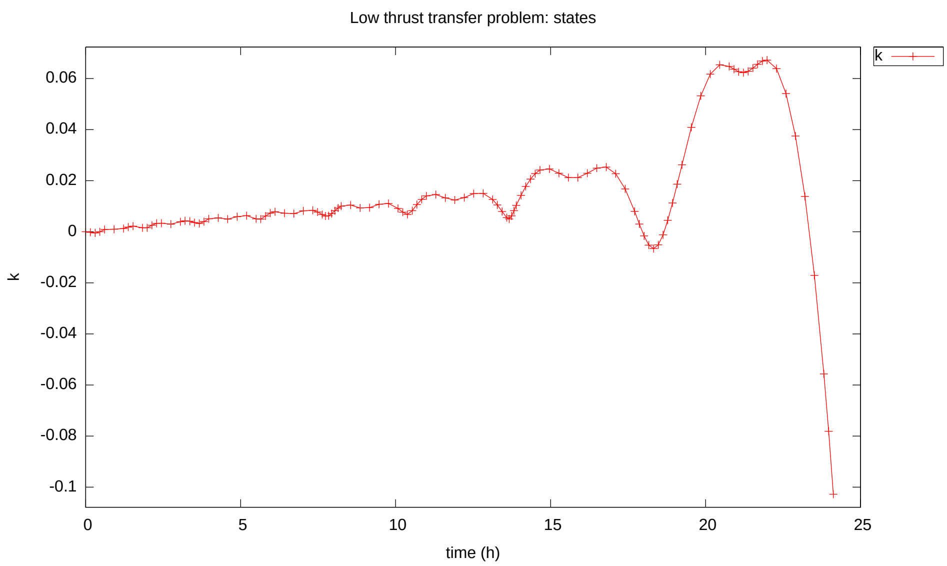

Figure 3.57: Modified equinoctial element k

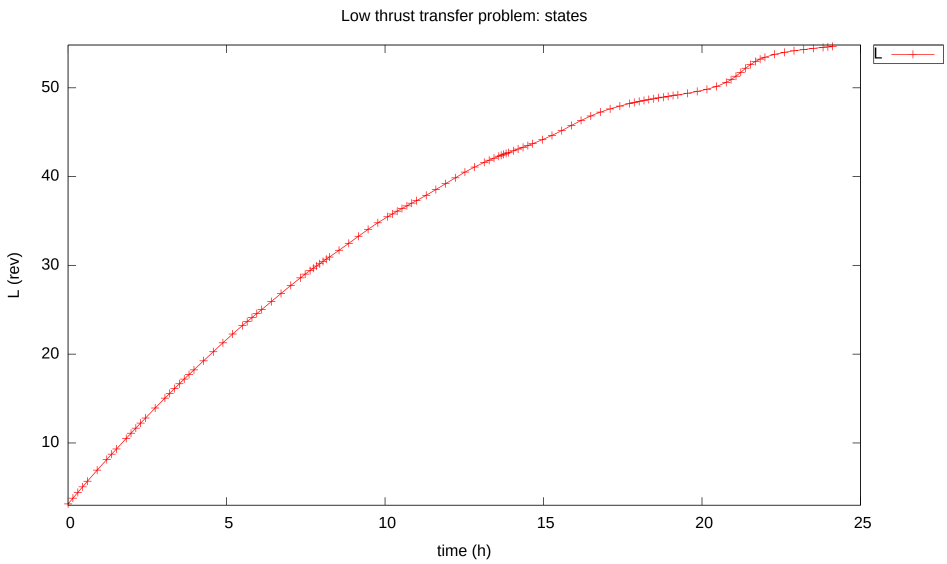

Figure 3.58: Modified equinoctial element L

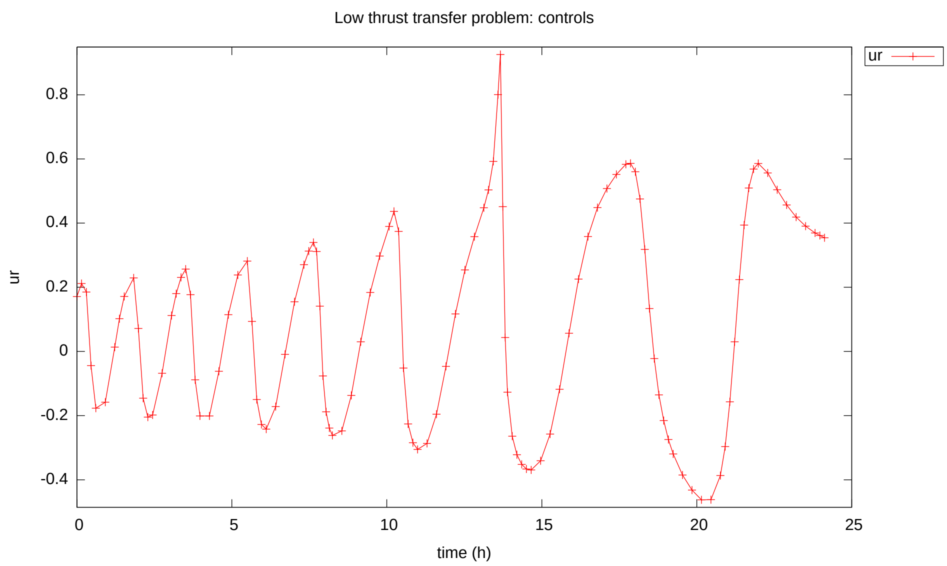

Figure 3.59: Radial component of the thrust direction vector, u_r

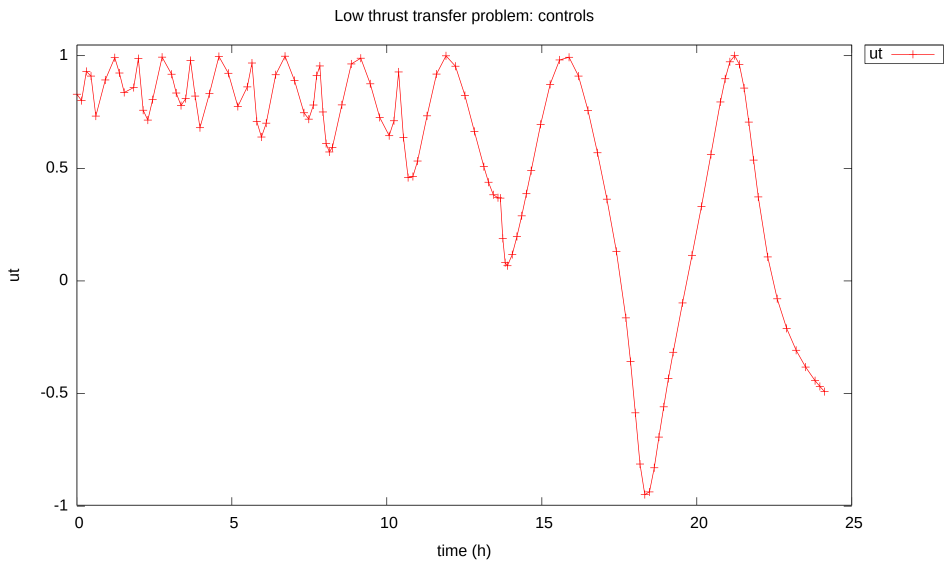

Figure 3.60: Tangential component of the thrust direction vector, u_t

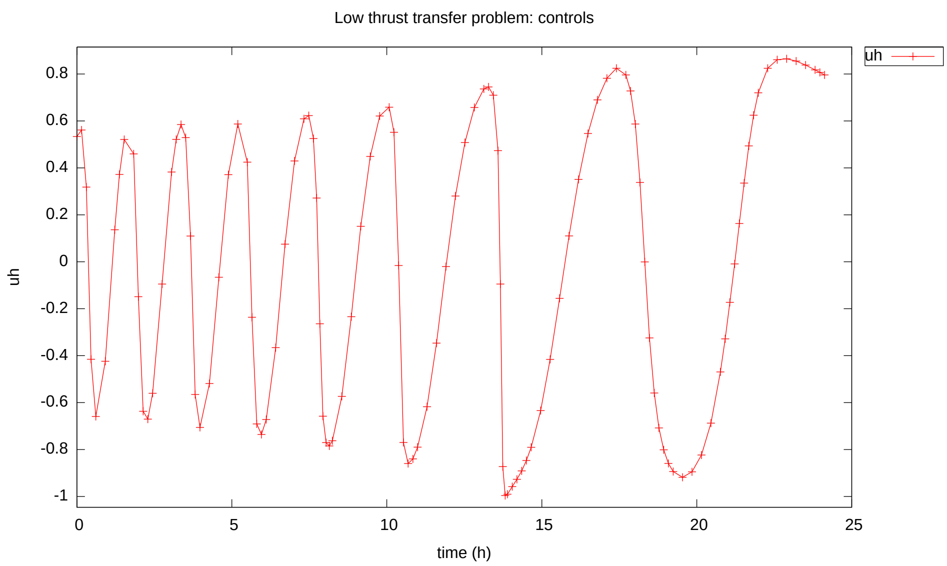

Figure 3.61: Normal component of the thrust direction vector, u_h

$$\mathbf{M}(\mathbf{q}(t))\ddot{\mathbf{q}}(t) = \mathbf{V}(\mathbf{q}(t), \dot{\mathbf{q}}(t)) + \mathbf{G}(\mathbf{q}(t)) + \mathbf{D}\mathbf{u}(t)$$

where $\mathbf{q} = [q_1(t), q_2(t), q_3(t)]^T$ is the vector of relative angles between the links, the normalized torque controls are $\mathbf{u}(t) = [u_1(t), u_2(t), u_3(t)]^T$, $\mathbf{D}$ is a diagonal matrix with constant values, $\mathbf{M}(\mathbf{q})$ is a symmetric inertia matrix, $\mathbf{V}(\mathbf{q}(t), \dot{\mathbf{q}}(t))$ are the torques caused by coriolis and centrifugal forces, $\mathbf{G}(\mathbf{q}(t))$ are gravitational torques. The model is described in detail in [29] and is fully included in the code for this example[3].

The example reported here consists of a minimum energy point to point trajectory, so that the objective is to find t_f and $\mathbf{u}(t) = [u_1(t), u_2(t), u_3(t)]^T$, $t \in [0, t_f]$ to minimise:

$$J = \int_0^{t_f} \mathbf{u}(t)^T \mathbf{u}(t) \mathrm{d}t \tag{3.107}$$

The boundary conditions associated with the problem are:

$$\begin{aligned}
\mathbf{q}(0) &= \begin{bmatrix} 0 & -1.5 & 0 \end{bmatrix}^T \\
\dot{\mathbf{q}}(0) &= \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T \\
\mathbf{q}(t_f) &= \begin{bmatrix} 1.0 & -1.95 & 1.0 \end{bmatrix}^T \\
\dot{\mathbf{q}}(t_f) &= \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T
\end{aligned} \tag{3.108}$$

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.62, 3.63 and 3.64, which contain the elements of the position vector $\mathbf{q}(t)$, the velocity vector $\dot{\mathbf{q}}(t)$, and the controls $\mathbf{u}(t)$, respectively. The mesh refinement process is described in Table 3.4.

```
PSOPT results summary
=====================

Problem:  Manutec R3 robot problem
CPU time (seconds): 4.293625e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:23:30 2020

Optimal (unscaled) cost function value:  2.040420e+01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  2.040420e+01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.300000e-01
```

[3] Dr. Martin Otter from DLR, Germany, has kindly authorised the author to publish a translated form of subroutine R3M2SI as part of the $\mathcal{PSOPT}$ distribution.

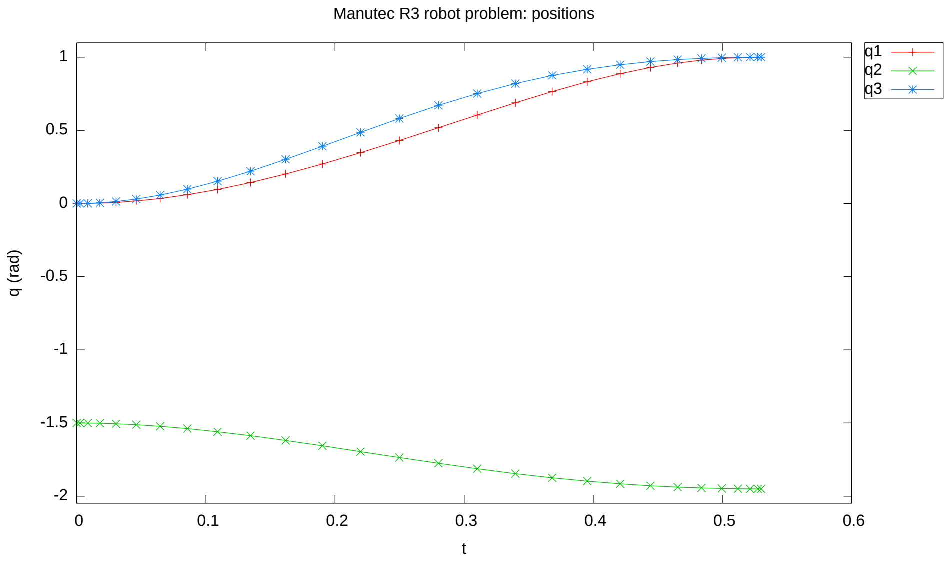

Figure 3.62: States q_1, q_2 and q_3 for the Manutec R3 robot minimum energy problem

```
Phase 1 maximum relative local error: 2.599460e-05
NLP solver reports:  The problem has been solved!
```

3.26 Minimum swing control for a container crane

Consider the following optimal control problem [38], which seeks to minimise the load swing of a container crane, while the load is transferred from one location to another. Find $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = 4.5 \int_0^{t_f} \left[x_3^2(t) + x_6^2(t)\right] dt \tag{3.109}$$

subject to the dynamic constraints

$$\begin{aligned}
\dot{x}_1 &= 9x_4 \\
\dot{x}_2 &= 9x_5 \\
\dot{x}_3 &= 9x_6 \\
\dot{x}_4 &= 9(u_1 + 17.2656x_3) \\
\dot{x}_5 &= 9u_2 \\
\dot{x}_6 &= -\frac{9}{x_2}\left[u_1 + 27.0756x_3 + 2x_5x_6\right]
\end{aligned} \tag{3.110}$$

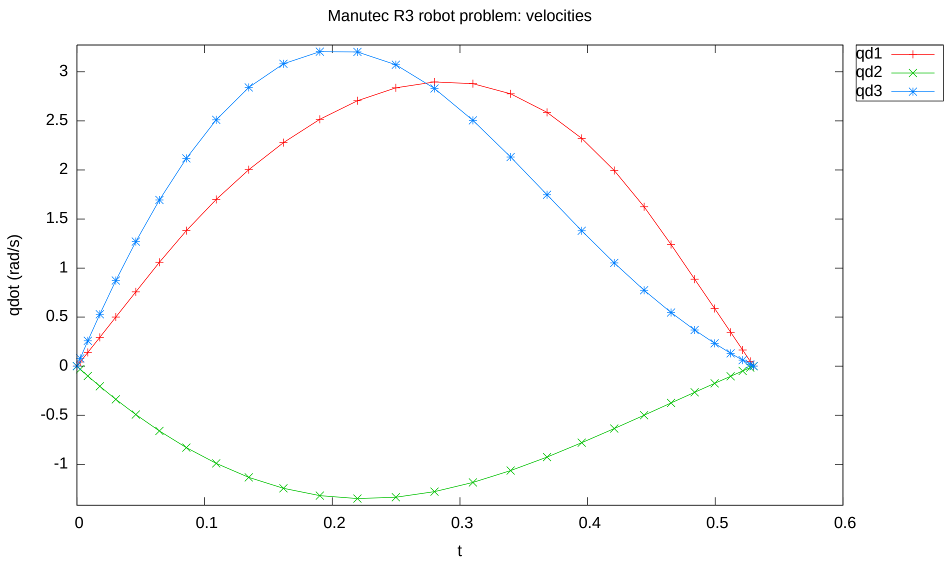

Figure 3.63: States $\dot{q}_1, \dot{q}_2$ and $\dot{q}_3$ for the Manutec R3 robot minimum energy problem

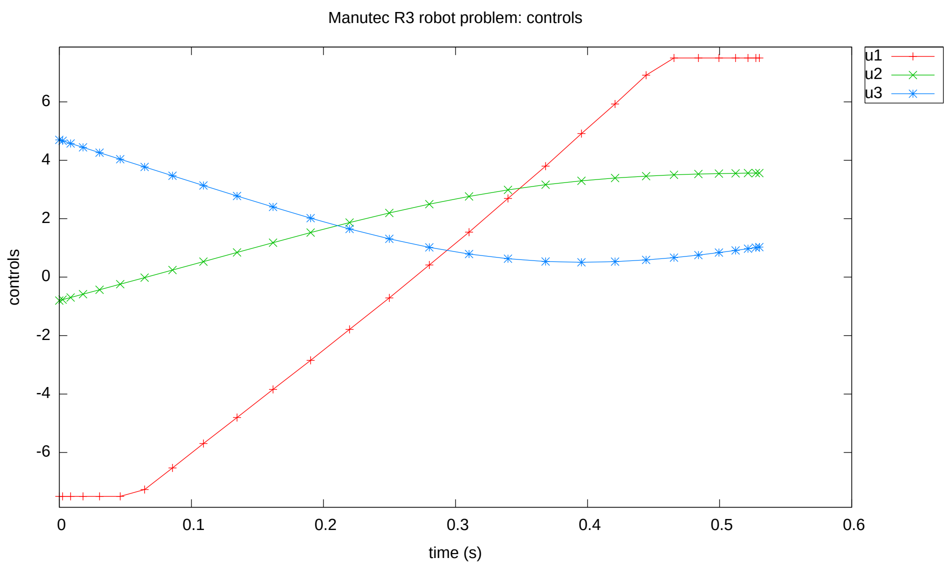

Figure 3.64: Controls u_1, u_2 and u_3 for the Manutec R3 robot minimum energy problem

Table 3.4: Mesh refinement statistics: Manutec R3 robot problem

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	$\epsilon_{\max}$	$\mathrm{CPU_a}$
1	LGL-ST	20	182	133	43	43	35	0	860	4.676e-05	4.909e-01
2	LGL-ST	25	227	163	34	35	32	0	875	3.636e-05	2.298e-01
3	LGL-ST	26	236	169	36	37	31	0	962	2.948e-05	2.925e-01
4	LGL-ST	27	245	175	36	37	32	0	999	5.015e-05	9.704e-01
5	LGL-ST	28	254	181	15	16	15	0	448	2.599e-05	4.414e-01
$\mathrm{CPU_b}$	-	-	-	-	-	-	-	-	-	-	1.869e+00
-	-	-	-	-	164	168	145	0	4144	-	4.294e+00

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, $\epsilon_{\max}$ = maximum relative ODE error, $\mathrm{CPU_a}$ = CPU time in seconds spent by NLP algorithm, $\mathrm{CPU_b}$ = additional CPU time in seconds spent by PSOPT

the boundary conditions

$$\begin{array}{rclrcl} x_1(0) & = & 0 & x_1(t_f) & = & 10 \\ x_2(0) & = & 22 & x_2(t_f) & = & 14 \\ x_3(0) & = & 0 & x_3(t_f) & = & 0 \\ x_4(0) & = & 0 & x_4(t_f) & = & 2.5 \\ x_5(0) & = & -1 & x_5(t_f) & = & 0 \\ x_6(0) & = & 0 & x_6(t_f) & = & 0 \end{array}, \tag{3.111}$$

and the bounds

$$\begin{aligned} -2.83374 \leq & u_1(t) \leq 2.83374, \\ -0.80865 \leq & u_2(t) \leq 0.71265, \\ -2.5 \leq & x_4(t) \leq 2.5, \\ -1 \leq & x_5(t) \leq 1. \end{aligned} \tag{3.112}$$

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.65, 3.66 and 3.67, which contain the elements of the state x_1 to x_3, x_4 to x_6, and the controls, respectively.

```
PSOPT results summary
=====================

Problem:  Minimum swing control for a container crane
CPU time (seconds): 2.771775e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:18:28 2020
```

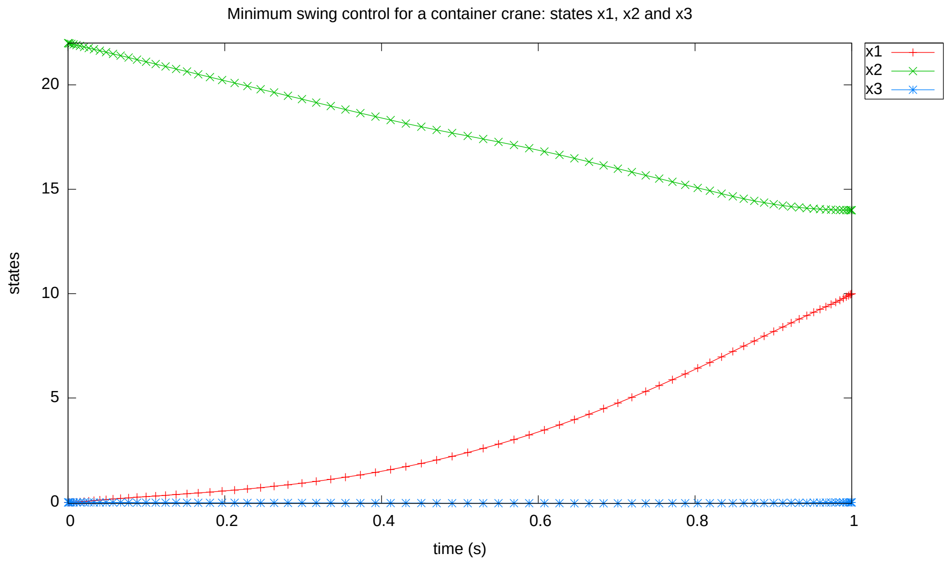

Figure 3.65: States x_1, x_2 and x_3 for minimum swing crane control problem

```
Optimal (unscaled) cost function value:  5.151279e-03
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.151279e-03
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 9.820877e-05
NLP solver reports:  The problem has been solved!
```

3.27 Minimum time to climb for a supersonic aircraft

Consider the following optimal control problem, which finds the minimum time to climb to a given altitude for a supersonic aircraft [4]. Minimize the cost functional

$$J = t_f \tag{3.113}$$

subject to the dynamic constraints

$$\begin{aligned}
\dot{h} &= v \sin\gamma \\
\dot{v} &= \tfrac{1}{m}\left[T(M,h)\cos\alpha - D\right] - \tfrac{\mu}{(R_e+h)^2}\sin\gamma \\
\dot{\gamma} &= \tfrac{1}{mv}\left[T(M,h)\sin\alpha + L\right] + \cos\gamma\left[\tfrac{v}{(R_e+h)} - \tfrac{\mu}{v(R_e+h)^2}\right] \\
\dot{w} &= \tfrac{-T(M,h)}{I_{sp}}
\end{aligned} \tag{3.114}$$

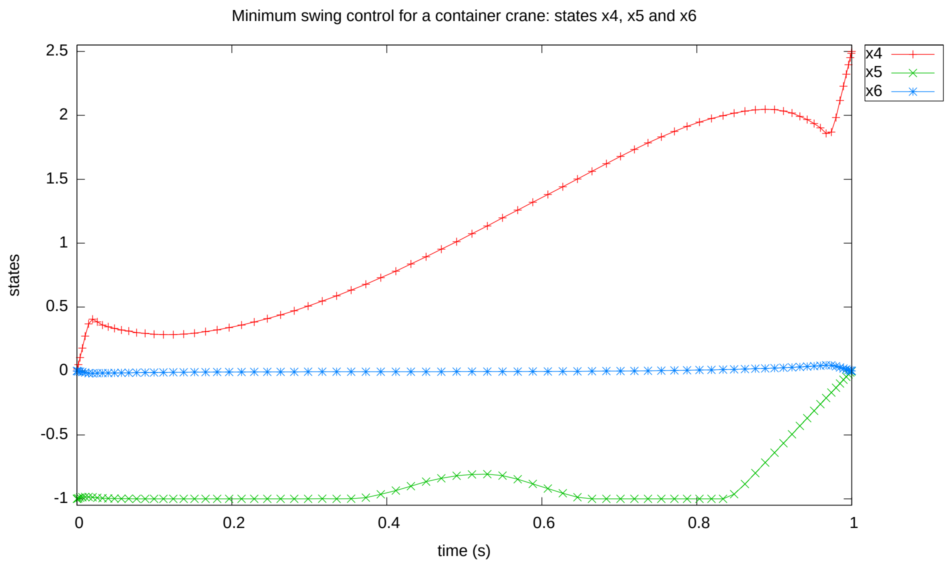

Figure 3.66: States x_4, x_5 and x_6 for minimum swing crane control problem

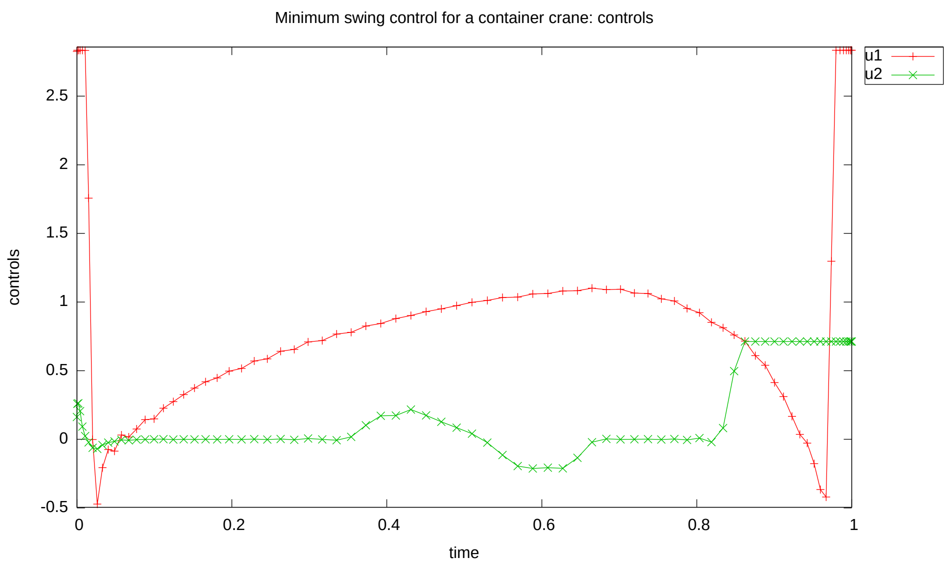

Figure 3.67: Controls for minimum swing crane control problem

where h is the altitude (ft), v is the velocity (ft/s), γ is the flight path angle (rad), w is the weight (lb), L is the lift force, D is the drag force (lb), T is the thrust (lb), $M = v/c$ is the mach number, $m = w/g_0$ (slug) is the mass, $c(h)$ is the speed of sound (ft/s), R_e is the radious of Earth, and μ is the gravitational constant. The control input α is the angle of attack (rad).

The speed of sound is given by:

$$c = 20.0468\sqrt{\theta} \tag{3.115}$$

where $\theta = \theta(h)$ is the atmospheric temperature (K).

The aerodynamic forces are given by:

$$\begin{aligned} D &= \frac{1}{2} C_D S \rho v^2 \\ L &= \frac{1}{2} C_L S \rho v^2 \end{aligned} \tag{3.116}$$

where

$$\begin{aligned} C_L &= c_{L\alpha}(M)\alpha \\ C_D &= c_{D0}(M) + \eta(M) c_{L\alpha}(M)\alpha^2 \end{aligned} \tag{3.117}$$

where C_L and C_D are aerodynamic lift and drag coefficients, S is the aerodynamic reference area of the aircraft, and $\rho = \rho(h)$ is the air density.

The boundary conditions are given by:

$$\begin{aligned} h(0) &= 0 \text{ (ft)}, \\ h(t_f) &= 65600.0 \text{ (ft)} \\ v(0) &= 424.260 \text{ (ft/s)}, \\ v(t_f) &= 968.148 \text{ (ft/s)} \\ \gamma(0) &= \gamma(t_f) = 0 \text{ (rad)} \\ w(0) &= 42000.0 \text{ lb} \end{aligned} \tag{3.118}$$

The parameter values are given by:

$$\begin{aligned} S &= 530 \text{ (ft}^2\text{)}, \\ I_{sp} &= 1600.0 \text{ (sec)} \\ \mu &= 0.14046539 \ \times 10^{17} \text{ (ft}^3/\text{s}^2\text{)}, \\ g_0 &= 32.174 \text{ (ft/s}^2\text{)} \\ R_e &= 20902900 \text{ (ft)} \end{aligned} \tag{3.119}$$

The variables $c_{L\alpha}(M)$, $c_{D0}(M)$, $\eta(M)$ are interpolated from 1-D tabular data which is given in the code and also in [4], using spline interpolation, while the thrust $T(M,h)$ is interpolated from 2-D tabular data given in the code and in [4], using 2D spline interpolation.

The air density ρ and the atmospheric temperature θ were calculated using the US Standard Atmosphere Model 1976[4], based on the standard temperature of 15 (deg C) at zero altitude and the standard air density of 1.22521 (slug/ft^3) at zero altitude.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
//////////////////            climb.cxx             ////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example           ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title: Minimum time to climb of a supersonic aircraft //////////
//////// Last modified: 12 January 2009                 ////////////////
//////// Reference:     GPOPS Manual   ////////////////
//////// (See PSOPT handbook for full reference)           ///////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"




//////////////////////////////////////////////////////////////////////////
/////////  Declare an auxiliary structure to hold local constants ////////
//////////////////////////////////////////////////////////////////////////

struct Constants {
  double g0;
  double S;
  double Re;
  double Isp;
  double mu;
  MatrixXd* CLa_table;
  MatrixXd* CD0_table;
  MatrixXd* eta_table;
  MatrixXd* T_table;
  MatrixXd* M1;
  MatrixXd* M2;
  MatrixXd* h1;
  MatrixXd* htab;
  MatrixXd* ttab;
  MatrixXd* ptab;
  MatrixXd* gtab;
};

typedef struct Constants Constants_;

void atmosphere(adouble* alt,adouble* sigma,adouble* delta,adouble* theta, Constants_& CONSTANTS);

void atmosphere_model(adouble* rho, adouble* M, adouble v, adouble h, Constants_& CONSTANTS);


//////////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
return tf;
}

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
```

[4] see http://www.pdas.com/programs/atmos.f90

```
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)

{
    return  0.0;
}



//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

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

  Constants_& CONSTANTS = *( (Constants_ *) workspace->problem->user_data );

  adouble alpha = controls[ 0]; // Angle of attack (rad)


  adouble h     =   states[ 0 ]; // Altitude (ft)
  adouble v     =   states[ 1 ]; // Velocity (ft/s)
  adouble gamma =   states[ 2 ]; // Flight path angle (rad)
  adouble w     =   states[ 3 ]; // weight (lb)


  double g0   = CONSTANTS.g0;
  double S    = CONSTANTS.S;
  double Re   = CONSTANTS.Re;
  double Isp  = CONSTANTS.Isp;
  double mu   = CONSTANTS.mu;

  MatrixXd& M1         = *CONSTANTS.M1;
  MatrixXd& M2         = *CONSTANTS.M2;
  MatrixXd& h1         = *CONSTANTS.h1;
  MatrixXd& CLa_table  = *CONSTANTS.CLa_table;
  MatrixXd& CD0_table  = *CONSTANTS.CD0_table;
  MatrixXd& eta_table  = *CONSTANTS.eta_table;
  MatrixXd& T_table    = *CONSTANTS.T_table;

  int lM1 = length(M1);

  adouble rho;
  adouble   m = w/g0;
  adouble   M;

  atmosphere_model( &rho, &M, v, h, CONSTANTS);

  adouble CL_a, CD0, eta, T;


  spline_interpolation( &CL_a, M, M1, CLa_table,  lM1);
  spline_interpolation( &CD0,  M, M1, CD0_table, lM1);
  spline_interpolation( &eta,  M, M1, eta_table, lM1);
  spline_2d_interpolation(&T, M, h, M2, h1, T_table, workspace);

//  smooth_linear_interpolation( &CL_a, M, M1, CLa_table,  lM1);
//  smooth_linear_interpolation( &CD0,  M, M1, CD0_table, lM1);
//  smooth_linear_interpolation( &eta,  M, M1, eta_table, lM1);
//  smooth_bilinear_interpolation(&T, M, h, M2, h1, T_table);


//  linear_interpolation( &CL_a, M, M1, CLa_table,  lM1);
//  linear_interpolation( &CD0,  M, M1, CD0_table, lM1);
//  linear_interpolation( &eta,  M, M1, eta_table, lM1);
//  bilinear_interpolation(&T, M, h, M2, h1, T_table);


   adouble CL = CL_a*alpha;
   adouble CD = CD0 + eta*CL_a*alpha*alpha;

   adouble D = 0.5*CD*S*rho*v*v;
   adouble L = 0.5*CL*S*rho*v*v;


   adouble hdot     = v*sin(gamma);
   adouble vdot     = 1.0/m*(T*cos(alpha)-D) - mu/pow(Re+h,2.0)*sin(gamma);
   adouble gammadot = (1.0/(m*v))*(T*sin(alpha)+L) + cos(gamma)*(v/(Re+h)-mu/(v*pow(Re+h,2.0)));
   adouble wdot     = -T/Isp;
```

```
   derivatives[ 0 ] = hdot;
   derivatives[ 1 ] = vdot;
   derivatives[ 2 ] = gammadot;
   derivatives[ 3 ] = wdot;


}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

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



      adouble h0     = initial_states[0];
      adouble v0     = initial_states[1];
      adouble gamma0 = initial_states[2];
      adouble w0     = initial_states[3];

      adouble hf     = final_states[0];
      adouble vf     = final_states[1];
      adouble gammaf = final_states[2];

      e[ 0 ] = h0;
      e[ 1 ] = v0;
      e[ 2 ] = gamma0;
      e[ 3 ] = w0;
      e[ 4 ] = hf;
      e[ 5 ] = vf;
      e[ 6 ] = gammaf;

}


///////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

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

  // Single phase problem

}



////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{
////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name = "Minimum time to climb for a supersonic aircraft";
    problem.outfilename = "climb.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

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

    psopt_level1_setup(problem);
```

```
////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates   = 4;
    problem.phases(1).ncontrols = 1;
    problem.phases(1).nevents   = 7;
    problem.phases(1).npath     = 0;

    problem.phases(1).nodes     = (RowVectorXi(1) << 30).finished();


    psopt_level2_setup(problem, algorithm);

///////////////////////////////////////////////////////////////////////////
//////////////////  Declare an instance of Constants structure /////////////
///////////////////////////////////////////////////////////////////////////


    Constants_ CONSTANTS;

    problem.user_data = (void*) &CONSTANTS;

///////////////////////////////////////////////////////////////////////////
//////////////////  Declare MatrixXd objects to store results //////////////
///////////////////////////////////////////////////////////////////////////

    MatrixXd x, u, t, H;



///////////////////////////////////////////////////////////////////////////
///////////////////  Initialize CONSTANTS and //////////////////////////////
//////////////////  declare local variables  //////////////////////////////
///////////////////////////////////////////////////////////////////////////

   CONSTANTS.g0   = 32.174; // ft/s^2
   CONSTANTS.S    = 530.0; // ft^2
   CONSTANTS.Re   = 20902900.0; // ft
   CONSTANTS.Isp  = 1600.00; //s
   CONSTANTS.mu   = 0.14076539E17; // ft^3/s^2


   MatrixXd M1(1,9);
   M1 << 0.E0, .4E0, .8E0, .9E0, 1.E0, 1.2E0, 1.4E0, 1.6E0, 1.8E0;

   MatrixXd M2(1,10);
   M2 << 0.E0, .2E0, .4E0, .6E0, .8E0, 1.E0, 1.2E0, 1.4E0, 1.6E0, 1.8E0;

   MatrixXd h1(1,10);
   h1 << 0.E0, 5E3, 10.E3, 15.E3, 20.E3, 25.E3, 30.E3, 40.E3, 50.E3, 70.E3;

   MatrixXd CLa_table(1,9);
   CLa_table << 3.44E0, 3.44E0, 3.44E0, 3.58E0, 4.44E0, 3.44E0, 3.01E0, 2.86E0, 2.44E0;

   MatrixXd CD0_table(1,9);
   CD0_table << .013E0, .013E0, .013E0, .014E0, .031E0, 0.041E0, .039E0, .036E0, .035E0;

   MatrixXd eta_table(1,9);
   eta_table << .54E0, .54E0, .54E0, .75E0, .79E0, .78E0, .89E0, .93E0, .93E0;

   MatrixXd T_table(10,10);
   T_table << 24200., 24000.,  20300.,  17300.,14500.,12200.,10200.,5700.,3400.,100.,
   28000., 24600.,  21100.,  18100.,15200.,12800.,10700.,6500.,3900.,200.,
   28300., 25200.,  21900.,  18700.,15900.,13400.,11200.,7300.,4400.,400.,
   30800., 27200.,  23800.,  20500.,17300.,14700.,12300.,8100.,4900.,800.,
   34500., 30300.,  26600.,  23200.,19800.,16800.,14100.,9400.,5600.,1100.,
   37900., 34300.,  30400.,  26800.,23300.,19800.,16800.,11200.,6800.,1400.,
   36100., 38000.,  34900.,  31300.,27300.,23600.,20100.,13400.,8300.,1700.,
   34300., 36600.,  38500.,  36100.,31600.,28100.,24200.,16200.,10000.,2200.,
   32500., 35200.,  42100.,  38700.,35700.,32000.,28100.,19300.,11900.,2900.,
   30700., 33800.,  45700.,  41300.,39800.,34600.,31100.,21700.,13300.,3100. ;

   MatrixXd htab(1,8);
   htab <<      0.0, 11.0, 20.0, 32.0, 47.0, 51.0, 71.0, 84.852;
   MatrixXd ttab(1,8);
   ttab <<      288.15, 216.65, 216.65, 228.65, 270.65, 270.65, 214.65, 186.946;
   MatrixXd ptab(1,8);
   ptab <<      1.0, 2.233611E-1, 5.403295E-2, 8.5666784E-3, 1.0945601E-3,
                                  6.6063531E-4, 3.9046834E-5, 3.68501E-6;
   MatrixXd gtab(1,8);
   gtab << -6.5, 0.0, 1.0, 2.8, 0.0, -2.8, -2.0, 0.0;
```

```
//   M1.Print("M1");

//   M2.Print("M2");

//   h1.Print("h1");

//   CLa_table.Print("CLa_table");

//   CD0_table.Print("CD0_table");

//   eta_table.Print("eta_table");

//   T_table.Print("T_table");


   CONSTANTS.M1         = &M1;
   CONSTANTS.M2         = &M2;
   CONSTANTS.h1         = &h1;
   CONSTANTS.CLa_table  = &CLa_table;
   CONSTANTS.CD0_table  = &CD0_table;
   CONSTANTS.eta_table  = &eta_table;
   CONSTANTS.T_table    = &T_table;
   CONSTANTS.htab       = &htab;
   CONSTANTS.ttab       = &ttab;
   CONSTANTS.ptab       = &ptab;
   CONSTANTS.gtab       = &gtab;

   double h0     = 0.0;
   double hf     = 65600.0;
   double v0     = 424.26;
   double vf     = 968.148;
   double gamma0 = 0.0;
   double gammaf = 0.0;
   double w0     = 42000.0;

   double hmin =  0;
   double hmax =  69000.0;
   double vmin =  1.0;
   double vmax =  2000.0;
   double gammamin = -89.0*pi/180.0; // -89.0*pi/180.0;
   double gammamax =  89.0*pi/180.0; //  89.0*pi/180.0;
   double wmin     = 0.0;
   double wmax     = 45000.0;
   double alphamin = -20.0*pi/180.0;
   double alphamax =  20.0*pi/180.0;

   double t0min = 0.0;
   double t0max = 0.0;
   double tfmin = 200.0;
   double tfmax = 500.0;


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////

    int iphase =  1;

    problem.phases(iphase).bounds.lower.StartTime    = t0min;
    problem.phases(iphase).bounds.upper.StartTime    = t0max;

    problem.phases(iphase).bounds.lower.EndTime      = tfmin;
    problem.phases(iphase).bounds.upper.EndTime      = tfmax;


    problem.phases(iphase).bounds.lower.states(0) = hmin;
    problem.phases(iphase).bounds.upper.states(0) = hmax;
    problem.phases(iphase).bounds.lower.states(1) = vmin;
    problem.phases(iphase).bounds.upper.states(1) = vmax;
    problem.phases(iphase).bounds.lower.states(2) = gammamin;
    problem.phases(iphase).bounds.upper.states(2) = gammamax;
    problem.phases(iphase).bounds.lower.states(3) = wmin;
    problem.phases(iphase).bounds.upper.states(3) = wmax;


    problem.phases(iphase).bounds.lower.controls(0) = alphamin;
    problem.phases(iphase).bounds.upper.controls(0) = alphamax;

    // The following bounds fix the initial and final state conditions

    problem.phases(iphase).bounds.lower.events(0) = h0;
    problem.phases(iphase).bounds.upper.events(0) = h0;
```

```
    problem.phases(iphase).bounds.lower.events(1) = v0;
    problem.phases(iphase).bounds.upper.events(1) = v0;
    problem.phases(iphase).bounds.lower.events(2) = gamma0;
    problem.phases(iphase).bounds.upper.events(2) = gamma0;
    problem.phases(iphase).bounds.lower.events(3) = w0;
    problem.phases(iphase).bounds.upper.events(3) = w0;
    problem.phases(iphase).bounds.lower.events(4) = hf;
    problem.phases(iphase).bounds.upper.events(4) = hf;
    problem.phases(iphase).bounds.lower.events(5) = vf;
    problem.phases(iphase).bounds.upper.events(5) = vf;
    problem.phases(iphase).bounds.lower.events(6) = gammaf;
    problem.phases(iphase).bounds.upper.events(6) = gammaf;


////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes = problem.phases(iphase).nodes(0);


    MatrixXd stateGuess(4,nnodes);

    stateGuess.row(0) = linspace(h0,hf,nnodes);
    stateGuess.row(1) = linspace(v0,vf,nnodes);
    stateGuess.row(2) = linspace(gamma0,gammaf,nnodes);
    stateGuess.row(3) = linspace(w0,0.8*w0,nnodes);



    problem.phases(1).guess.controls       = zeros(1,nnodes);
    problem.phases(1).guess.states         = stateGuess;
    problem.phases(1).guess.time           = linspace(t0min, tfmax, nnodes);


////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "numerical";
    algorithm.collocation_method           = "trapezoidal";
    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;
    algorithm.mesh_refinement              = "automatic";
    algorithm.mr_max_iterations            = 4;
    algorithm.defect_scaling               = "jacobian-based";




////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////////////


    psopt(solution, problem, algorithm);


////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////

    x  = solution.get_states_in_phase(1);
    u  = solution.get_controls_in_phase(1);
    t  = solution.get_time_in_phase(1);
    H       = solution.get_dual_hamiltonian_in_phase(1);

    MatrixXd h     = x.row(0);
```

```
    MatrixXd v     = x.row(1);
    MatrixXd gamma = x.row(2);
    MatrixXd w     = x.row(3);

////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

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


////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////


    plot(t,h/1000.0,problem.name + ": altitude", "time (s)", "altitude (x1,000 ft)", "h");
    plot(t,v/100.0,problem.name + ": velocity", "time (s)", "velocity (x100 ft/s)", "v");
    plot(t,gamma*180/pi,problem.name + ": flight path angle", "time (s)", "gamma (deg)", "gamma");
    plot(t,w/10000.0,problem.name + ": weight", "time (s)", "w (x10,000 lb)", "w");
    plot(t,u*180/pi,problem.name + ": angle of attack", "time (s)", "alpha (deg)", "alpha");

    plot(t,h/1000.0,problem.name + ": altitude", "time (s)", "altitude (x1,000 ft", "h",
                                   "pdf","climb_altitude.pdf");
    plot(t,v/100.0,problem.name + ": velocity", "time (s)", "velocity (x100 ft/s)", "v",
                            "pdf","climb_velocity.pdf");
    plot(t,gamma*180/pi,problem.name + ": flight path angle", "time (s)", "gamma (deg)", "gamma",
                                     "pdf","climb_fpa.pdf");
    plot(t,w/10000.0,problem.name + ": weight", "time (s)", "w (x10,000 lb)", "w", "pdf",
                                      "weight.pdf");
    plot(t,u*180/pi,problem.name + ": angle of attack", "time (s)", "alpha (deg)", "alpha",
                                      "pdf", "alpha.pdf");


}


void atmosphere(adouble* alt,adouble* sigma,adouble* delta,adouble* theta, Constants_& CONSTANTS)
// US Standard Atmosphere Model 1976
// Adopted from original Fortran 90 code by Ralph Carmichael
// Fortran code located at: http://www.pdas.com/programs/atmos.f90
{
/*!   -----------------------------------------------------------------------
! PURPOSE - Compute the properties of the 1976 standard atmosphere to 86 km.
! AUTHOR - Ralph Carmichael, Public Domain Aeronautical Software
! NOTE - If alt > 86, the values returned will not be correct, but they will
!   not be too far removed from the correct values for density.
!   The reference document does not use the terms pressure and temperature
!   above 86 km.
  IMPLICIT NONE
!============================================================================
!    A R G U M E N T S                                                      |
!============================================================================
  alt        ! geometric altitude, km.
  sigma      ! density/sea-level standard density
  delta      ! pressure/sea-level standard pressure
  theta      ! temperature/sea-level standard temperature
*/

/*!============================================================================
!    L O C A L   C O N S T A N T S                                          |
!============================================================================
*/
  double REARTH = 6369.0;                 // radius of the Earth (km)
  double GMR = 34.163195;                 // hydrostatic constant
  int NTAB=8;       // number of entries in the defining tables
/*!============================================================================
!    L O C A L   V A R I A B L E S                                          |
!============================================================================
*/
  int i,j,k;                                                  // counters
  adouble h;                                      // geopotential altitude (km)
  adouble tgrad, tbase;      // temperature gradient and base temp of this layer
  adouble tlocal;                                           // local temperature
  adouble deltah;                                 // height above base of this layer
/*!============================================================================
!    L O C A L   A R R A Y S   ( 1 9 7 6   S T D.  A T M O S P H E R E )  |
!============================================================================
*/
```

```
  MatrixXd& htab = *CONSTANTS.htab;
  MatrixXd& ttab = *CONSTANTS.ttab;
  MatrixXd& ptab = *CONSTANTS.ptab;
  MatrixXd& gtab = *CONSTANTS.gtab;


//!----------------------------------------------------------------------------
  h=(*alt)*REARTH/((*alt)+REARTH);      //convert geometric to geopotential altitude

  i=1;
  j=NTAB;                                       // setting up for binary search
  while (j<=i+1) {
    k=(i+j)/2;                                              // integer division
    if (h < htab(k-1)) {
      j=k;
    } else {
       i=k;
    }
  }

  tgrad=gtab(i-1);                                     // i will be in 1...NTAB-1
  tbase=ttab(i-1);
  deltah=h-htab(i-1);
  tlocal=tbase+tgrad*deltah;
  *theta=tlocal/ttab(0);                                    // temperature ratio

  if (tgrad == 0.0) {                                  //  pressure ratio
    *delta=ptab(i-1)*exp(-GMR*deltah/tbase);
  } else {
    *delta=ptab(i-1)*pow(tbase/tlocal, GMR/tgrad);
  }

  *sigma=(*delta)/(*theta);                                        // density ratio
  return;

}


void atmosphere_model(adouble* rho, adouble* M, adouble v, adouble h, Constants_& CONSTANTS)
{
   double feet2meter = 0.3048;
   double kgperm3_to_slug_per_feet3 = 0.062427960841/32.174049;
   adouble alt, sigma, delta, theta;
   alt = h.value()*feet2meter/1000.0;

   // Call the standard atmosphere model 1976

   atmosphere(&alt, &sigma, &delta, &theta, CONSTANTS);

   adouble rho1 = 1.22521 * sigma; // Multiply by standard density at zero altitude and 15 deg C.

   rho1 = rho1*kgperm3_to_slug_per_feet3;

   *rho = rho1;

   adouble T;
   adouble mach;

   double TempStandardSeaLevel = 288.15; // in K, or 15 deg C.

   T = theta*TempStandardSeaLevel;

   adouble a = 20.0468 * sqrt(T); // Speed of sound in m/s.

   a = a/feet2meter;  // Speed of sound in ft/s

   mach = v/a;

   *M = mach;


  return;
}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and Figures 3.68, to 3.72.

Table 3.5: Mesh refinement statistics: Minimum time to climb for a supersonic aircraft

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	$\epsilon_{\max}$	CPU_a
1	TRP	30	152	128	14008	2506	46	0	147854	3.968e-02	8.689e+00
2	TRP	42	212	176	26234	3431	62	0	284773	2.070e-02	1.864e+01
3	H-S	58	349	240	47152	4943	68	0	850196	1.139e-02	5.454e+01
4	H-S	77	463	316	65245	6014	71	0	1377206	1.898e-03	8.888e+01
CPU_b	-	-	-	-	-	-	-	-	-	-	2.756e+01
-	-	-	-	-	152639	16894	247	0	2660029	-	1.983e+02

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, $\epsilon_{\max}$ = maximum relative ODE error, CPU_a = CPU time in seconds spent by NLP algorithm, CPU_b = additional CPU time in seconds spent by PSOPT

The results can be compared with those presented in [4]. Table 3.1 shows the mesh refinement history for this problem.

```
PSOPT results summary
=====================

Problem:  Minimum time to climb for a supersonic aircraft
CPU time (seconds): 1.983098e+02
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 13:38:34 2020

Optimal (unscaled) cost function value:  3.188146e+02
Phase 1 endpoint cost function value: 3.188146e+02
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 3.188146e+02
Phase 1 maximum relative local error: 1.897601e-03
NLP solver reports:  The problem has been solved!
```

3.28 Missile terminal burn maneouvre

This example illustrates the design of a missile trajectory to strike a specified target from given initial conditions in minimum time [37]. Figure 3.28 shows the variables associated with the dynamic model of the missile employed in this example, where γ is the flight

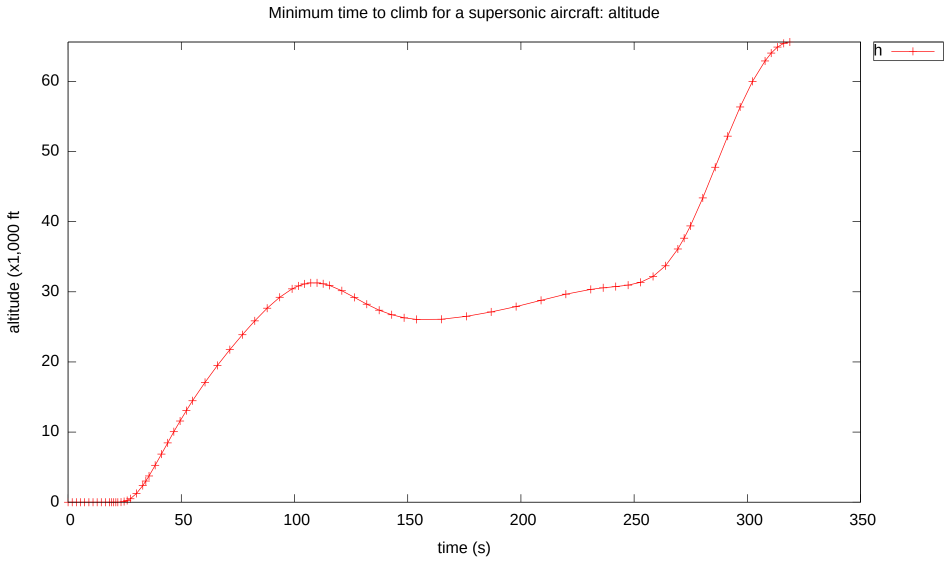

Figure 3.68: Altitude for minimum time to climb problem

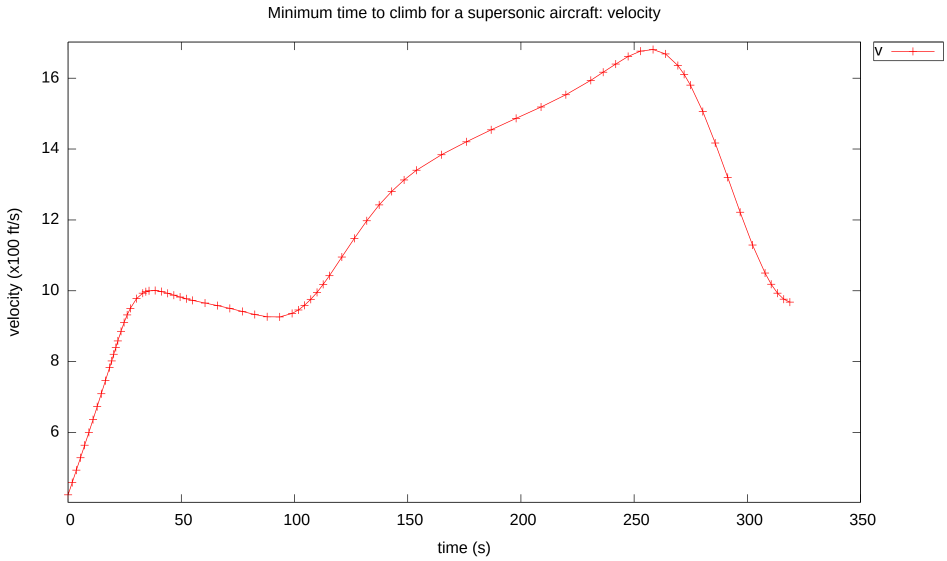

Figure 3.69: Velocity for minimum time to climb problem

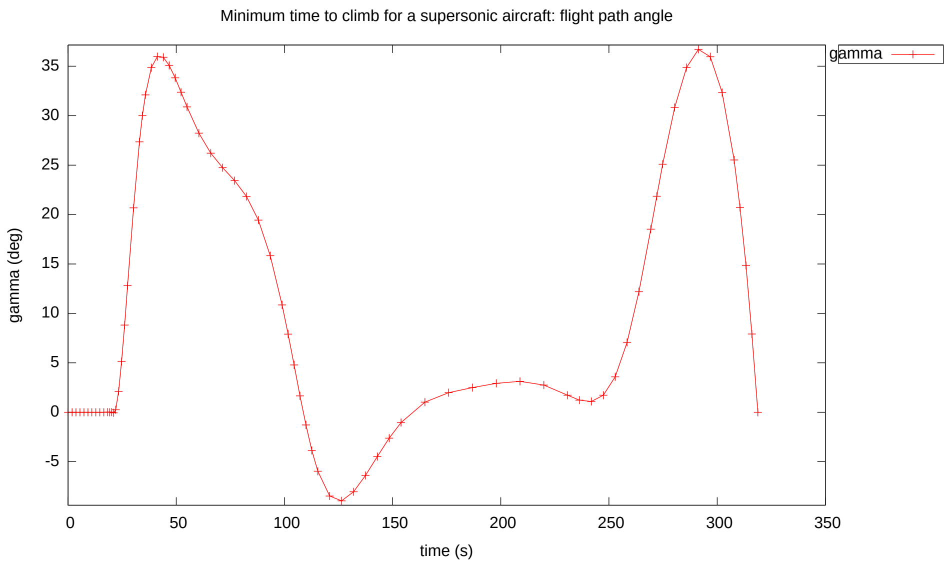

Figure 3.70: Flight path angle for minimum time to climb problem

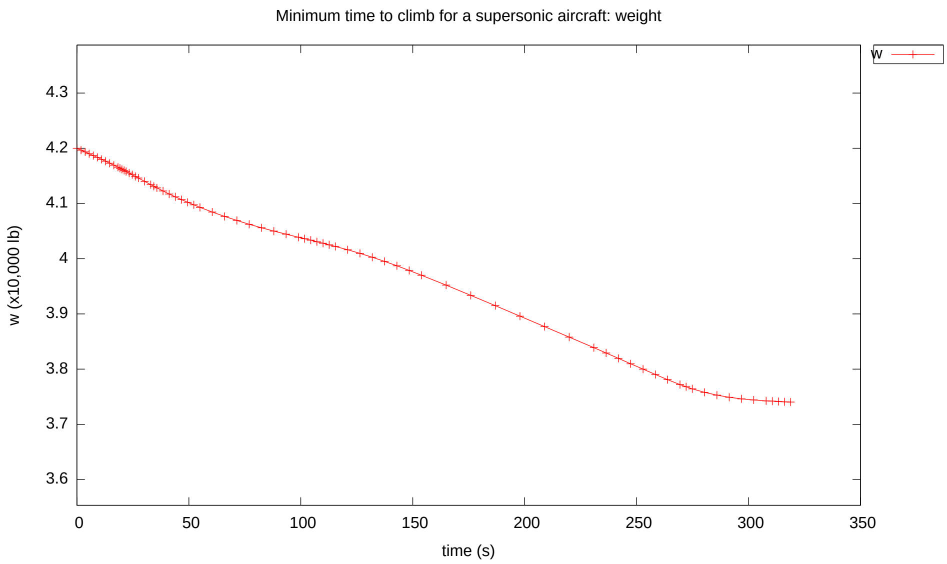

Figure 3.71: Weight for minimum time to climb problem

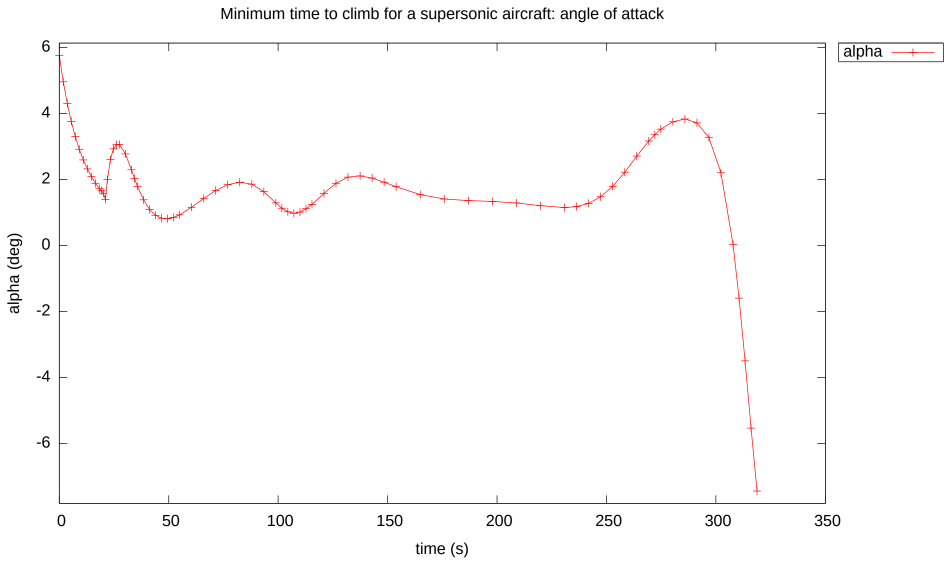

Figure 3.72: Angle of attack (α) for minimum time to climb problem

path angle, α is the angle of attack, V is the missile speed, x is the longitudinal position, h is the altitude, D is the axial aerodynamic force, L is the normal aerodynamic force, and T is the thrust.

The equations of motion of the missile are given by:

$$\begin{aligned}
\dot{\gamma} &= \frac{T-D}{mg}\sin\alpha + \frac{L}{mV}\cos\alpha - \frac{g\cos\gamma}{V} \\
\dot{V} &= \frac{T-D}{m}\cos\alpha - \frac{L}{m}\sin\alpha - g\cos\gamma \\
\dot{x} &= V\cos\gamma \\
\dot{h} &= V\sin\gamma
\end{aligned}$$

where

$$\begin{aligned}
D &= \frac{1}{2}C_d\rho V^2 Sref \\
C_d &= A_1\alpha^2 + A_2\alpha + A_3 \\
L &= \frac{1}{2}C_l\rho V^2 Sref \\
C_l &= B_1\alpha + B_2 \\
\rho &= C_1h^2 + C_2h + C_3
\end{aligned}$$

where all the model parameters are given in Table 3.6. The initial conditions for the state variables are:

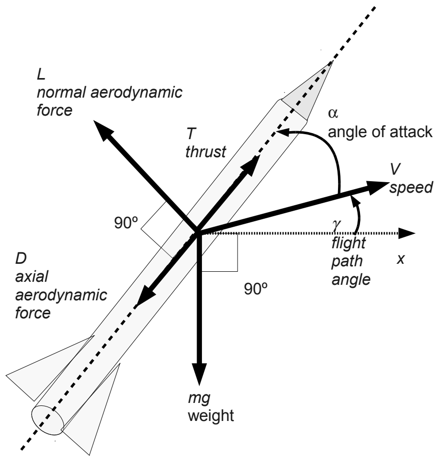

Figure 3.73: Ilustration of the variables associated with the missile model

Table 3.6: Parameters values of the missile model

Parameter	Value	Units
m	1005	kg
g	9.81	$\mathrm{m/s^2}$
S_{ref}	0.3376	$\mathrm{m^2}$
A_1	-1.9431	
A_2	-0.1499	
A_3	0.2359	
B_1	21.9	
B_2	0	
C_1	3.312×10^{-9}	$\mathrm{kg/m^5}$
C_2	-1.142×10^{-4}	$\mathrm{kg/m^4}$
C_3	1.224	$\mathrm{kg/m^3}$

$$\begin{aligned}
\gamma(0) &= 0 \\
V(0) &= 272\mathrm{m/s} \\
x(0) &= 0\mathrm{m} \\
h(0) &= 30\mathrm{m}
\end{aligned}$$

The terminal conditions on the states are:

$$\begin{aligned}
\gamma(t_f) &= -\pi/2 \\
V(t_f) &= 310\mathrm{m/s} \\
x(t_f) &= 10000\mathrm{m} \\
h(t_f) &= 0\mathrm{m}
\end{aligned}$$

The problem constraints are given by:

$$\begin{aligned}
200 \leq & V \leq 310 \\
1000 \leq & T \leq 6000 \\
-0.3 \leq & \alpha \leq 0.3 \\
-4 \leq & \frac{L}{mg} \leq 4 \\
& h \geq 30 \ (\text{for}\, x \leq 7500\mathrm{m}) \\
& h \geq 0 \ (\text{for}\, x > 7500\mathrm{m})
\end{aligned}$$

Note that the path constraints on the altitude are non-smooth. Given that non-smoothness causes problems with nonlinear programming, the constraints on the altitude were approximated by a single smooth constraint:

$$\mathcal{H}_\epsilon(x - 7500))h(t) + [1 - \mathcal{H}_\epsilon(x - 7500)][h(t) - 30] \geq 0$$

where $\mathcal{H}_\epsilon(z)$ is a smooth version of the Heaviside function, which is computed as follows:

$$\mathcal{H}_\epsilon(z) = 0.5(1 + \tanh(z/\epsilon))$$

where $\epsilon > 0$ is a small number.

The problem is solved by using automatic mesh refinement starting with 50 nodes. The final solution, which is found after six mesh refinement iterations, has 85 nodes. Figure 3.74 shows the missile altitude as a function of the longitudinal position. Figures 3.75 and 3.76 show, respectively, the missile speed and angle of attack as functions of time. The output from $\mathcal{PSOPT}$ is summarised in the box below.

```
PSOPT results summary
=====================

Problem:  Missile problem
CPU time (seconds): 1.857816e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:24:21 2020

Optimal (unscaled) cost function value:  4.091755e+01
Phase 1 endpoint cost function value: 4.091755e+01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 4.091755e+01
Phase 1 maximum relative local error: 4.930020e-04
NLP solver reports:  The problem has been solved!
```

3.29 Moon lander problem

Consider the following optimal control problem, which is known in the literature as the moon lander problem [34]. Find t_f and $T(t) \in [0, t_f]$ to minimize the cost functional

$$J = \int_0^{t_f} T(t)dt \tag{3.120}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{h} &= v \\ \dot{v} &= -g + T/m \\ \dot{m} &= -T/E \end{aligned} \tag{3.121}$$

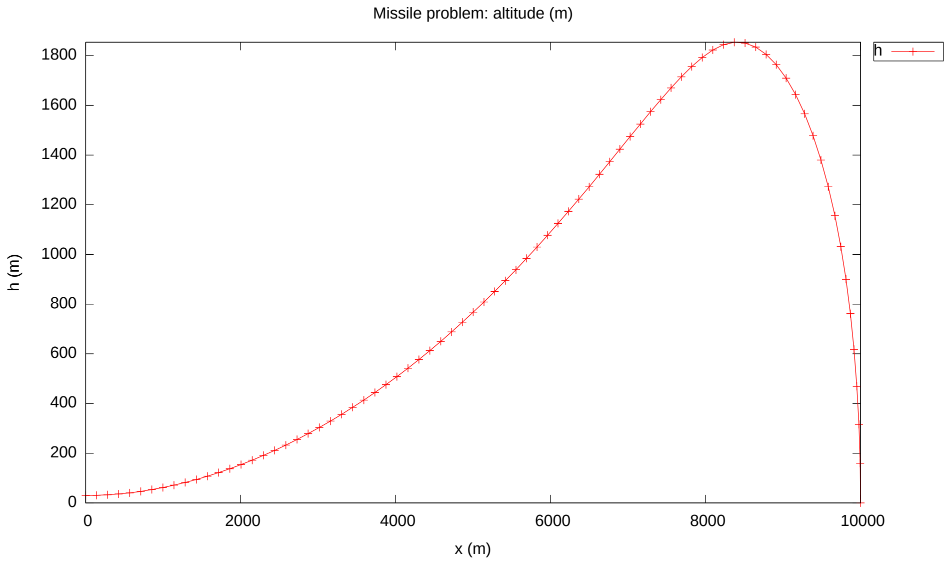

Figure 3.74: Missile altitude and a function of the longitudinal position

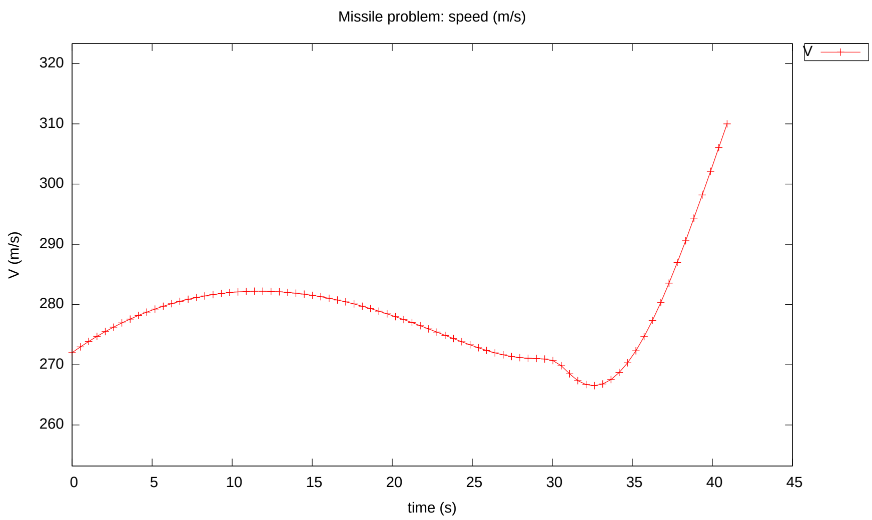

Figure 3.75: Missile speed as a function of time

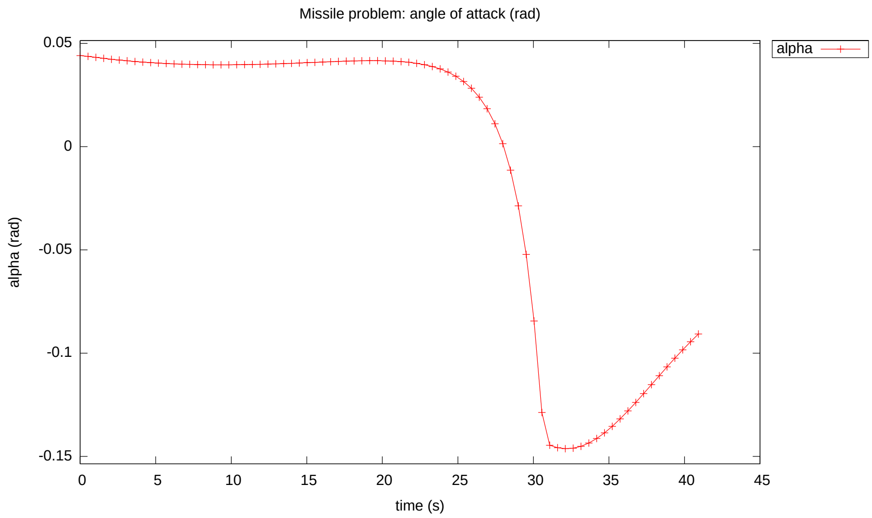

Figure 3.76: Missile angle of attack as a function of time

the boundary conditions:

$$\begin{array}{rcl} h(0) & = & 1 \\ v(0) & = & -0.783 \\ m(0) & = & 1 \\ h(t_f) & = & 0.0 \\ v(t_f) & = & 0.0 \end{array} \tag{3.122}$$

and the bounds

$$\begin{array}{rr} 0 \leq T(t) & \leq 1.227 \\ -20 \leq h(t) & \leq 20 \\ -20 \leq v(t) & \leq 20 \\ 0.01 \leq m(t) & \leq 1 \\ 0 \leq t_f & \leq 1000 \end{array} \tag{3.123}$$

where $g = 1.0$, and $E = 2.349$.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.77 and 3.78, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Moon Lander Problem
CPU time (seconds): 1.798258e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
```

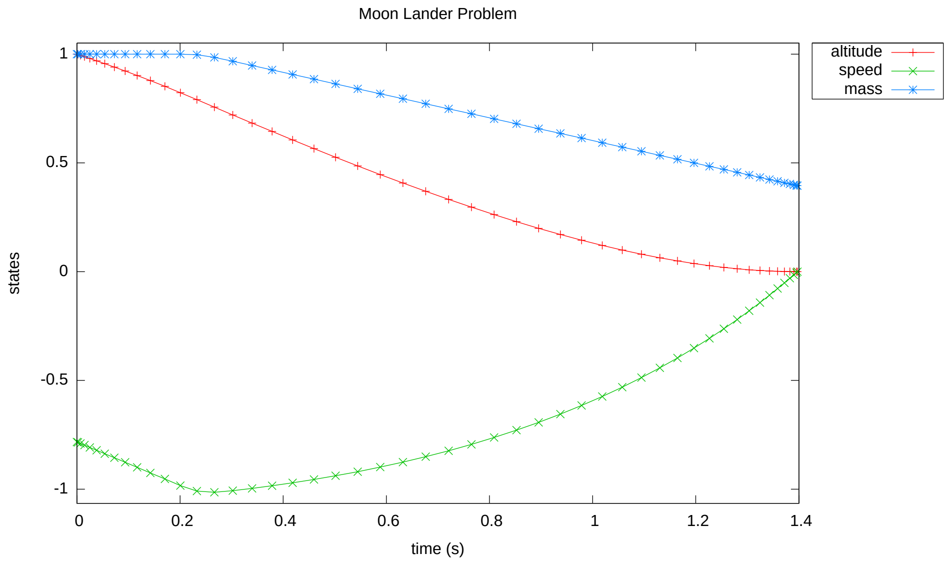

Figure 3.77: States for moon lander problem

```
Date and time of this run:  Wed Sep 23 12:23:56 2020

Optimal (unscaled) cost function value:  1.420408e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  1.420408e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.397050e+00
Phase 1 maximum relative local error: 9.568728e-05
NLP solver reports:  The problem has been solved!
```

3.30 Multi-segment problem

Consider the following optimal control problem, where the optimal control has a characteristic stepped shape [20]. Find $u(t) \in [0, 3]$ to minimize the cost functional

$$J = \int_0^3 x(t)dt \tag{3.124}$$

subject to the dynamic constraints

$$\dot{x} = u \tag{3.125}$$

the boundary conditions:

$$\begin{aligned} x(0) &= 1 \\ x(3) &= 1 \end{aligned} \tag{3.126}$$

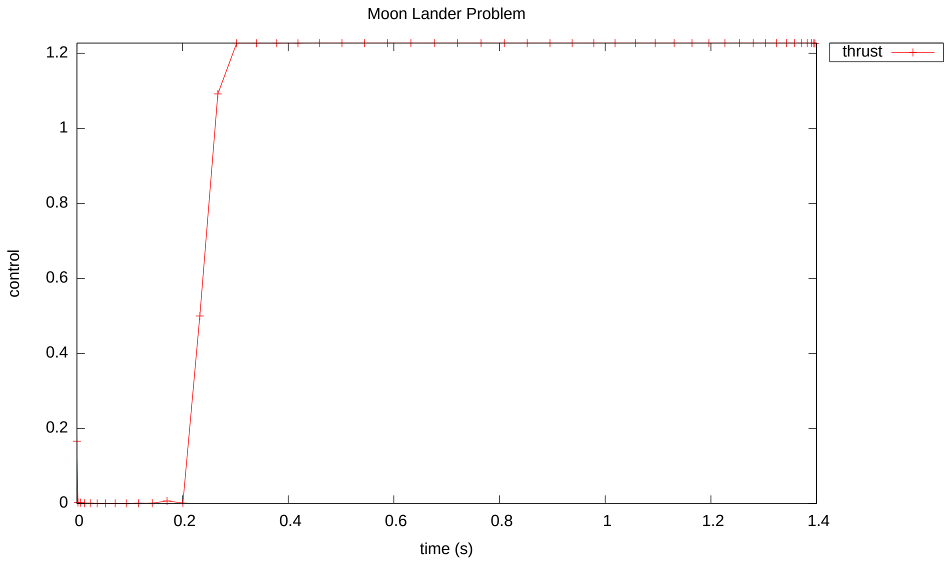

Figure 3.78: Control for moon lander problem

and the bounds

$$\begin{aligned} -1 \leq u(t) &\leq 1 \\ x(t) \geq 0 & \end{aligned} \tag{3.127}$$

The analytical optimal control is given by:

$$u(t) = \begin{cases} -1, & t \in [0, 1) \\ 0, & t \in [1, 2] \\ 1, & t \in (2, 3] \end{cases} \tag{3.128}$$

The problem has been solved using the multi-segment paradigm. Three segments are defined in the code, such that the initial time is fixed at $t_0^{(1)} = 0$, the final time is fixed at $t_f^{(3)} = 3$, and the intermediate junction times are $t_f^{(1)} = 1$, and $t_f^{(2)} = 2$.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
////////////////           steps.cxx                 /////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT example             /////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:  Steps problem                     ////////////////
//////// Last modified: 12 July 2009                      ////////////////
//////// Reference:      Gong, Farhoo, and Ross (2008)    ////////////////
////////                                                  ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////
```

```
#include "psopt.h"


//////////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////


adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return 0.0;
}

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
   adouble x = states[ 0 ];

   return (x);
}


//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE’s ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////


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

    adouble u = controls[CINDEX(1)];

    derivatives[ 0 ]  = u;

}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

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

{

    adouble x1_i  = initial_states[ 0 ];

    adouble x1_f  = final_states[   0 ];


   if ( iphase==1 ) {
    e[ 0 ] =  x1_i;
   }
   else if ( iphase==3 ) {
    e[ 0 ] =  x1_f;
   }

}



///////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

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

}



////////////////////////////////////////////////////////////////////////////
```

```
////////////////////  Define the main routine ////////////////////////////////
//////////////////////////////////////////////////////////////////////////////

int main(void)
{

//////////////////////////////////////////////////////////////////////////////
////////////////////  Declare key structures ////////////////////////////////
//////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;
    MSdata msdata;

//////////////////////////////////////////////////////////////////////////////
////////////////////  Register problem name  ////////////////////////////////
//////////////////////////////////////////////////////////////////////////////

    problem.name                   = "Steps problem";
    problem.outfilename                 = "steps.txt";

//////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
//////////////////////////////////////////////////////////////////////////////

    msdata.nsegments       = 3;
    msdata.nstates         = 1;
    msdata.ncontrols       = 1;
    msdata.nparameters     = 0;
    msdata.npath           = 0;
    msdata.ninitial_events = 1;
    msdata.nfinal_events   = 1;
    msdata.nodes           << 20; // nodes per segment

    multi_segment_setup(problem, algorithm, msdata );

//////////////////////////////////////////////////////////////////////////////
////////////////////  Enter problem bounds information //////////////////////
//////////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.controls(0) = -1.0;
    problem.phases(1).bounds.upper.controls(0) =  1.0;
    problem.phases(1).bounds.lower.states(0) = 0.0;
    problem.phases(1).bounds.upper.states(0) = 5.0;
    problem.phases(1).bounds.lower.events(0) = 1.0;
    problem.phases(3).bounds.lower.events(0) = 1.0;


    problem.phases(1).bounds.upper.events=problem.phases(1).bounds.lower.events;
    problem.phases(3).bounds.upper.events=problem.phases(3).bounds.lower.events;



    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(3).bounds.lower.EndTime      = 3.0;
    problem.phases(3).bounds.upper.EndTime      = 3.0;

//    problem.bounds.lower.times = "[0.0, 1.0, 2.0, 3.0]";
//    problem.bounds.upper.times = "[0.0, 1.0, 2.0, 3.0]";

problem.bounds.lower.times.resize(1,4);
problem.bounds.upper.times.resize(1,4);

    problem.bounds.lower.times << 0.0, 1.0, 2.0, 3.0;
    problem.bounds.upper.times << 0.0, 1.0, 2.0, 3.0;

    auto_phase_bounds(problem);



//////////////////////////////////////////////////////////////////////////////
////////////////////  Register problem functions  ///////////////////////////
//////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae             = &dae;
```

```
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes      = problem.phases(1).nodes(0);
    int ncontrols                       = problem.phases(1).ncontrols;
    int nstates                         = problem.phases(1).nstates;

    MatrixXd state_guess     =  zeros(nstates,nnodes);
    MatrixXd control_guess   =  zeros(ncontrols,nnodes);
    MatrixXd time_guess     =  linspace(0.0,3.0,nnodes);
    MatrixXd param_guess;



    state_guess   = linspace(1.0, 1.0, nnodes);

    control_guess = zeros(1,nnodes);


    auto_phase_guess(problem, control_guess, state_guess, param_guess, time_guess);

////////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;
    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.hessian                     = "exact";
    algorithm.mesh_refinement             = "automatic";
    algorithm.ode_tolerance               = 1.e-5;


////////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////


    MatrixXd x, u, t, x_ph1, u_ph1, t_ph1, x_ph2, u_ph2, t_ph2, x_ph3, u_ph3, t_ph3;

    x_ph1      = solution.get_states_in_phase(1);
    u_ph1      = solution.get_controls_in_phase(1);
    t_ph1      = solution.get_time_in_phase(1);

    x_ph2      = solution.get_states_in_phase(2);
    u_ph2      = solution.get_controls_in_phase(2);
    t_ph2      = solution.get_time_in_phase(2);

    x_ph3      = solution.get_states_in_phase(3);
    u_ph3      = solution.get_controls_in_phase(3);
    t_ph3      = solution.get_time_in_phase(3);


    x.resize(1, length(t_ph1)+length(t_ph2)+length(t_ph3) );
    u.resize(1, length(t_ph1)+length(t_ph2)+length(t_ph3) );
    t.resize(1, length(t_ph1)+length(t_ph2)+length(t_ph3) );

    x << x_ph2, x_ph2, x_ph3;
    u << u_ph1, u_ph2, u_ph3;
    t << t_ph1, t_ph2, t_ph3;


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

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

```
//////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
//////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name+": state","time (s)", "x", "x");

    plot(t,u,problem.name+": control","time (s)", "u", "u");

    plot(t,x,problem.name+": state","time (s)", "x", "x",
                             "pdf", "steps_state.pdf");


    plot(t,u,problem.name+": control","time (s)", "u", "u",
                             "pdf", "steps_control.pdf");


}

//////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
//////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.79 and 3.80, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Steps problem
CPU time (seconds): 4.224880e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:28:21 2020

Optimal (unscaled) cost function value:  1.000000e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.000001e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 4.165217e-08
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  5.787927e-08
Phase 2 initial time: 1.000000e+00
Phase 2 final time: 2.000000e+00
Phase 2 maximum relative local error: 2.914144e-07
Phase 3 endpoint cost function value: 0.000000e+00
Phase 3 integrated part of the cost:  5.000001e-01
Phase 3 initial time: 2.000000e+00
Phase 3 final time: 3.000000e+00
Phase 3 maximum relative local error: 4.172397e-08
NLP solver reports:  The problem has been solved!
```

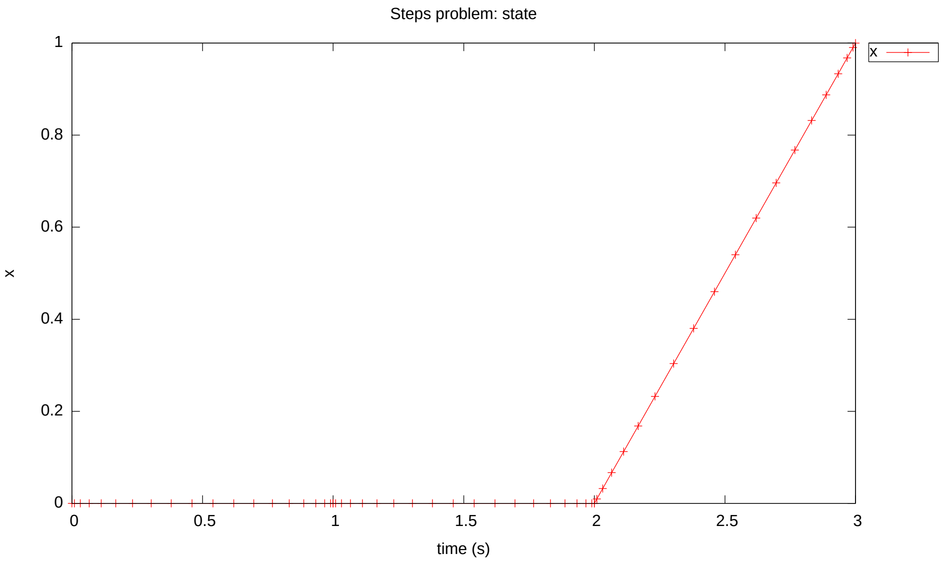

Figure 3.79: State trajectory for the multi-segment problem

3.31 Notorious parameter estimation problem

Consider the following parameter estimation problem, which is known to be challenging to single-shooting methods because of internal instability of the differential equations [35]. Find $p \in \Re$ to minimize

$$J = \sum_{i=1}^{200} (y_1(t_i) - \tilde{y}_1(i))^2 + (y_2(t_i) - \tilde{y}_2(i))^2 \tag{3.129}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{y}_1 &= y_2 \\ \dot{y}_2 &= \mu^2 y_1 - (\mu^2 + p^2)\sin(pt) \end{aligned} \tag{3.130}$$

where $\mu = 60.0$, $y_1(0) = 0$, $y_2(0) = \pi$. The parameter estimation facilities of $\mathcal{PSOPT}$ are used in this example. In this case, the observations function is:

$$g(x(\theta_k), u(\theta_k), p, \theta_k) = [y_1(\theta_k) \;\; y_2(\theta_k)]^T$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below. The code includes the generation of the measurement vectors $\tilde{y}_1$, and $\tilde{y}_2$ by adding Gaussian noise with standard

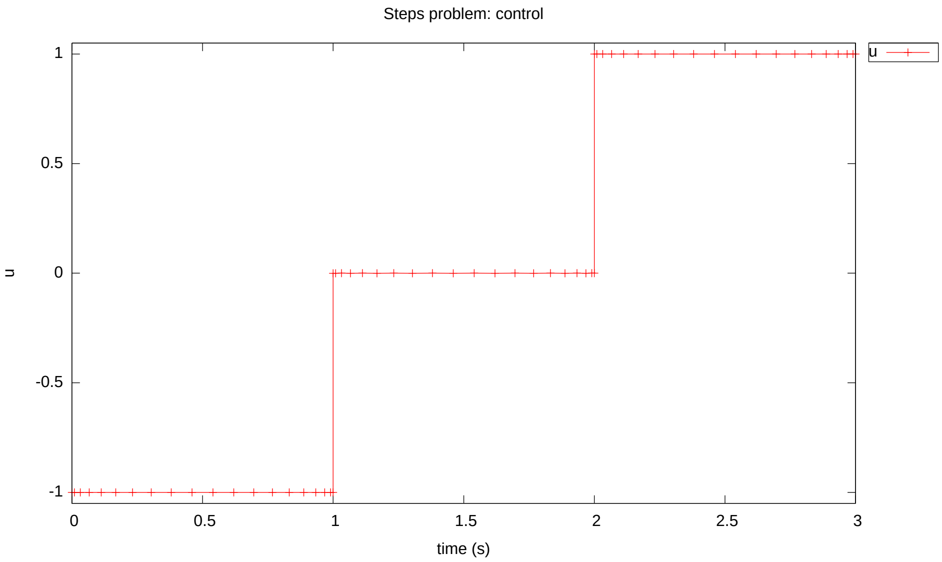

Figure 3.80: Control trajectory for the multi-segment problem

deviation 0.05 to the exact solution of the problem with $p = \pi$, which is given by:

$$y_1(t) = \sin(\pi t)$$
$$y_2(t) = \pi \cos(\pi t)$$

The code also defines the vector of sampling instants $\theta_i, i = 1, \ldots, 200$ as a uniform random samples in the interval $[0, 1]$.

```
//////////////////////////////////////////////////////////////////////////
//////////////////        notorious.cxx       ///////////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example            ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:  Bock's notorious parameter estimation problem //////////
//////// Last modified: 08 April 2011                   ////////////////
//////// Reference:     Schittkowskit (2002)          ////////////////
//////// (See PSOPT handbook for full reference)        ///////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"



//////////////////////////////////////////////////////////////////////////
///////////////////  Define the observation function //////////
//////////////////////////////////////////////////////////////////////////



void  observation_function( adouble* observations,
                            adouble* states, adouble* controls,
```

```
                            adouble* parameters, adouble& time, int k,
                            adouble* xad, int iphase, Workspace* workspace)
{

      observations[ 0 ] = states[ 0 ];
      observations[ 1 ] = states[ 1 ];
}




//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////


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

   adouble x1 = states[ 0 ];
   adouble x2 = states[ 1 ];


   adouble p  = parameters[ 0 ];
   adouble t  = time;

   double mu = 60.0;

   derivatives[0] =  x2;
   derivatives[1] =  mu*mu*x1 - (mu*mu + p*p)*sin(p*t);


}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

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

   e[ 0 ] =  initial_states[0];
   e[ 1 ] =  initial_states[1];


}


///////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}

using namespace std;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{
   int nobs =200;

   // Generate true solution at sampling points and add noise

   double sigma = 0.05;

   MatrixXd y1m, y2m;


//   theta = randu(1,nobs);

   MatrixXd noise1= GaussianRandom(1,nobs);
```

```
   MatrixXd noise2= GaussianRandom(1,nobs);

   MatrixXd theta = (MatrixXd::Random(1,nobs)+ones(1,nobs))/2.0;

   sort(theta);

   MatrixXd ss = (pi*theta).array().sin();
   MatrixXd cc = (pi*theta).array().cos();

   y1m =  ss + sigma*noise1;

   y2m =  pi*cc + sigma*noise2;



////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name           = "Bocks notorious parameter estimation problem";
    problem.outfilename                 = "notorious.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

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

    psopt_level1_setup(problem);


/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup ////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 2;
    problem.phases(1).ncontrols  = 0;
    problem.phases(1).nevents    = 2;
    problem.phases(1).npath      = 0;
    problem.phases(1).nparameters         = 1;
    problem.phases(1).nodes          << 80;
    problem.phases(1).nobserved             = 2;
    problem.phases(1).nsamples              = nobs;


    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////////
////////////  Enter estimation information                      ////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).observation_nodes      = theta;
    problem.phases(1).observations           << y1m,
                                                y2m;



////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states(0) =  -10.0;
    problem.phases(1).bounds.lower.states(1) =  -100.0;



    problem.phases(1).bounds.upper.states(0) =  10.0;
    problem.phases(1).bounds.upper.states(1) =  100.0;


    problem.phases(1).bounds.lower.parameters(0) =  -10.0;
```

```
    problem.phases(1).bounds.upper.parameters(0) =   10.0;

    problem.phases(1).bounds.lower.events(0) =  0.0;
    problem.phases(1).bounds.upper.events(0) =  0.0;

    problem.phases(1).bounds.lower.events(1) =  pi;
    problem.phases(1).bounds.upper.events(1) =  pi;



    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.0;
    problem.phases(1).bounds.upper.EndTime      = 1.0;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;
    problem.observation_function  = & observation_function;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes = problem.phases(1).nodes(0);


    problem.phases(1).guess.states         = zeros(2,nnodes);
    problem.phases(1).guess.time           = linspace(0.0, 1.0, nnodes);
    problem.phases(1).guess.parameters(0)  = 0.0;


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////

    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.collocation_method          = "trapezoidal";
    algorithm.nlp_iter_max                =  200;
    algorithm.nlp_tolerance               = 1.e-4;
//    algorithm.mesh_refinement             = "automatic";
//    algorithm.ode_tolerance               = 1.e-6;



////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////////////


    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare DMatrix objects to store results //////////////
////////////////////////////////////////////////////////////////////////////

    DMatrix states, x1, x2, p, t;


////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////

    states = solution.get_states_in_phase(1);
    t      = solution.get_time_in_phase(1);
    p      = solution.get_parameters_in_phase(1);
    x1     = states.row(0);
    x2     = states.row(1);


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

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

```
    Save(t,"t.dat");
    Print(p,"Estimated parameter");
    printf( "\nParameter error %e\n", abs(p(0)-pi) );


////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    plot(theta,y1m,t,x1,problem.name, "time (s)", "observed x1", "x1m x1");
    plot(theta,y2m,t,x2,problem.name, "time (s)", "observed x2", "x2m x2");



}

////////////////////////////////////////////////////////////////////////////
//////////////////////      END OF FILE     ////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below. The optimal parameter found was $p = 3.141180$, which is an approximation of π with an error of the order of 10^{-4}. The 95% confidence interval of the estimated parameter is $[3.132363, 3.149998]$.

```
PSOPT results summary
=====================

Problem:  Bocks notorious parameter estimation problem
CPU time (seconds): 8.947020e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:24:33 2020

Optimal (unscaled) cost function value:  8.806615e-01
Phase 1 endpoint cost function value: 8.806615e-01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 5.104843e-04
NLP solver reports:  The problem has been solved!
```

3.32 Predator-prey parameter estimation problem

This is a well known model that describes the behaviour of predator and prey species of an ecological system. The Letka-Volterra model system consist of two differential equations [35].

Table 3.7: Estimated parameter values and 95 percent statistical confidence limits on estimated parameters

Parameter	Low Confidence Limit	Value	High Confidence Limit
p_1	7.166429e-01	9.837490e-01	1.250855e+00
p_2	7.573469e-01	9.803930e-01	1.203439e+00
p_3	7.287846e-01	1.016900e+00	1.305015e+00
p_4	6.914964e-01	1.022702e+00	1.353909e+00

The dynamic equations are given by:

$$\begin{aligned} \dot{x}_1 &= -p_1 x_1 + p_2 x_1 x_2 \\ \dot{x}_2 &= p_3 x_2 - p_4 x_1 x_2 \end{aligned} \tag{3.131}$$

with boundary condition:

$$\begin{aligned} x_1(0) &= 0.4 \\ x_2(0) &= 1 \end{aligned}$$

The observation functions are:

$$\begin{aligned} g_1 &= x_1 \\ g_2 &= x_2 \end{aligned} \tag{3.132}$$

The measured data, with consists of $n_s = 10$ samples over the interval $t \in [0, 10]$, was constructed from simulations with parameter values $[p_1, p_2, p_3, p_4] = [1, 1, 1, 1]$ with added noise. The weights of both observations are the same and equal to one.

The solution is found using local discretisation (trapezoidal, Hermite-Simpson) and automatic mesh refinement, starting with 20 grid points with ODE tolerance 10^{-4}. The estimated parameter values and their 95% confidence limits are shown in Table 3.32. Figure 3.81 shows the observations as well as the estimated values of variables x_1 and x_2. The mesh statistics can be seen in Table 3.8

3.33 Rayleigh problem with mixed state-control path constraints

Consider the following optimal control problem, which involves a path constraint in which the control and the state appear explicitly [4]. Find $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = \int_0^{t_f} \left[x_1(t)^2 + u(t)^2 \right] \mathrm{d}t \tag{3.133}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -x_1 + x_2(1.4 - p x_2^2) + 4u\sin(\theta) \end{aligned} \tag{3.134}$$

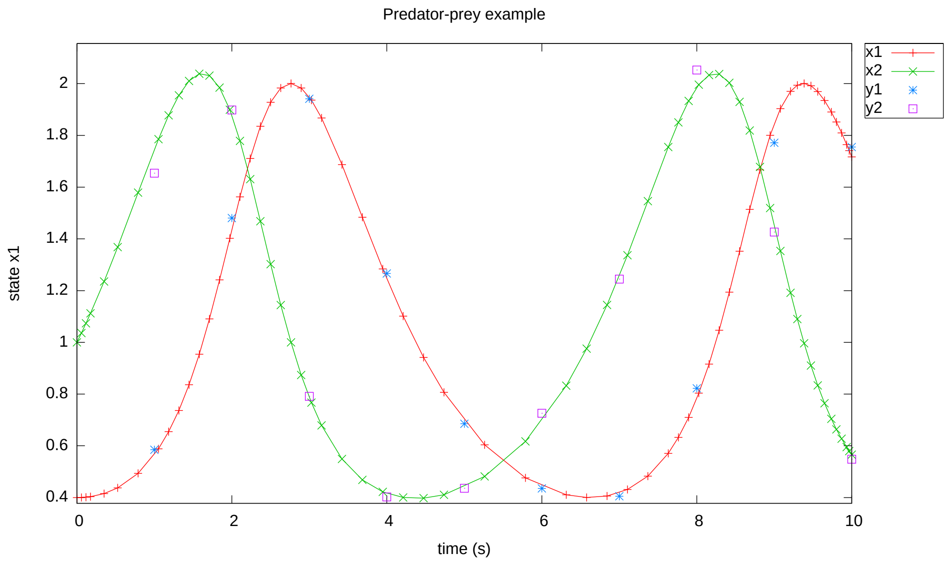

Figure 3.81: Observations y_1, y_2 and estimated states $x_1(t)$ and $x_2(t)$

Table 3.8: Mesh refinement statistics: Predator-prey example

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	$\epsilon_{\max}$	$\mathrm{CPU_a}$
1	TRP	20	46	43	20	20	20	0	780	1.615e-02	1.351e-01
2	TRP	28	62	59	11	11	11	0	605	8.919e-03	1.362e-01
3	H-S	39	84	81	10	10	10	0	1150	1.672e-03	2.398e-02
4	H-S	54	114	111	22	22	14	0	3520	1.268e-04	5.024e-02
5	H-S	61	128	125	8	8	8	0	1448	4.352e-05	2.649e-02
$\mathrm{CPU_b}$	-	-	-	-	-	-	-	-	-	-	4.315e-01
-	-	-	-	-	71	71	63	0	7503	-	8.035e-01

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, $\epsilon_{\max}$ = maximum relative ODE error, $\mathrm{CPU_a}$ = CPU time in seconds spent by NLP algorithm, $\mathrm{CPU_b}$ = additional CPU time in seconds spent by PSOPT

The path constraint:

$$u + \frac{x_1}{6} \leq 0 \tag{3.135}$$

and the boundary conditions:

$$\begin{aligned} x_1(0) &= -5 \\ x_2(0) &= -5 \end{aligned} \tag{3.136}$$

where $t_f = 4.5$, and $p = 0.14$.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.82, 3.83, 3.85, 3.85, 3.85, which show, respectively, the trajectories of the states, control, costates and path constraint multiplier. The results are comparable to those presented by [4].

```
PSOPT results summary
=====================

Problem:  Rayleigh problem
CPU time (seconds): 7.166780e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:26:21 2020

Optimal (unscaled) cost function value:  4.477625e+01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  4.477625e+01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 4.500000e+00
Phase 1 maximum relative local error: 2.329140e-03
NLP solver reports:  The problem has been solved!
```

3.34 Obstacle avoidance problem

Consider the following optimal control problem, which involves finding an optimal trajectory for a particle to travel from A to B while avoiding two forbidden regions [34]. Find $\theta(t) \in [0, t_f]$ to minimize the cost functional

$$J = \int_0^{t_f} \left[\dot{x}(t)^2 + \dot{y}(t)^2 \right] dt \tag{3.137}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x} &= V \cos(\theta) \\ \dot{y} &= V \sin(\theta) \end{aligned} \tag{3.138}$$

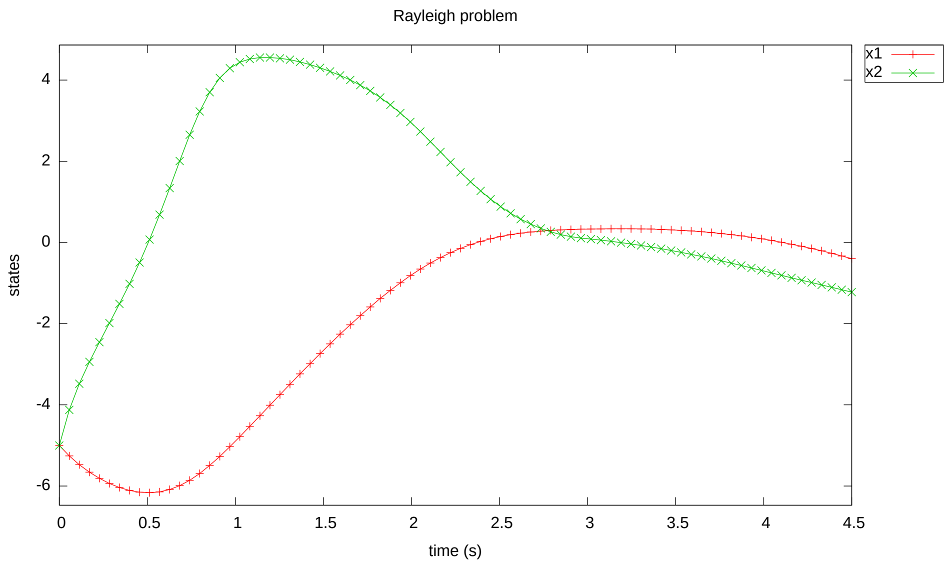

Figure 3.82: States for Rayleigh problem

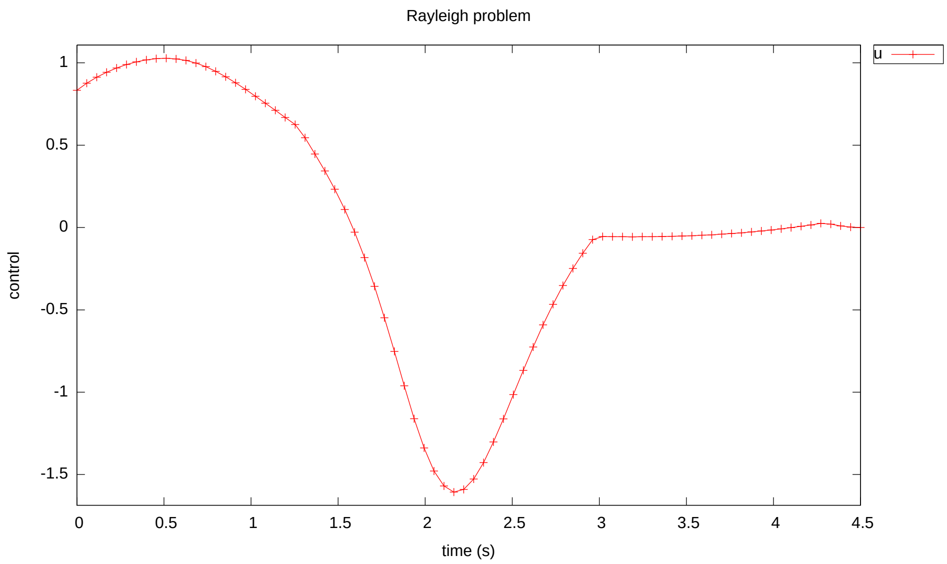

Figure 3.83: Optimal control for Rayleigh problem

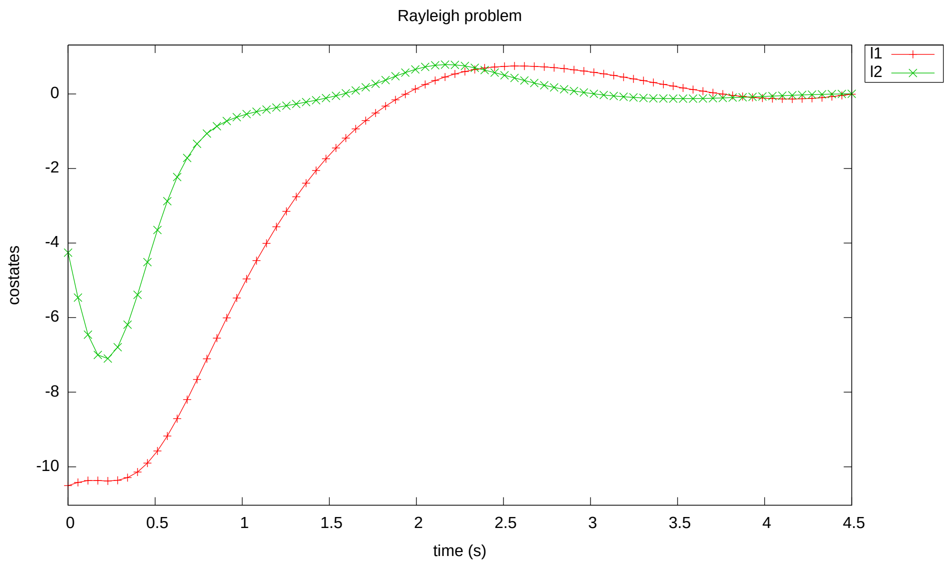

Figure 3.84: Costates for Rayleigh problem

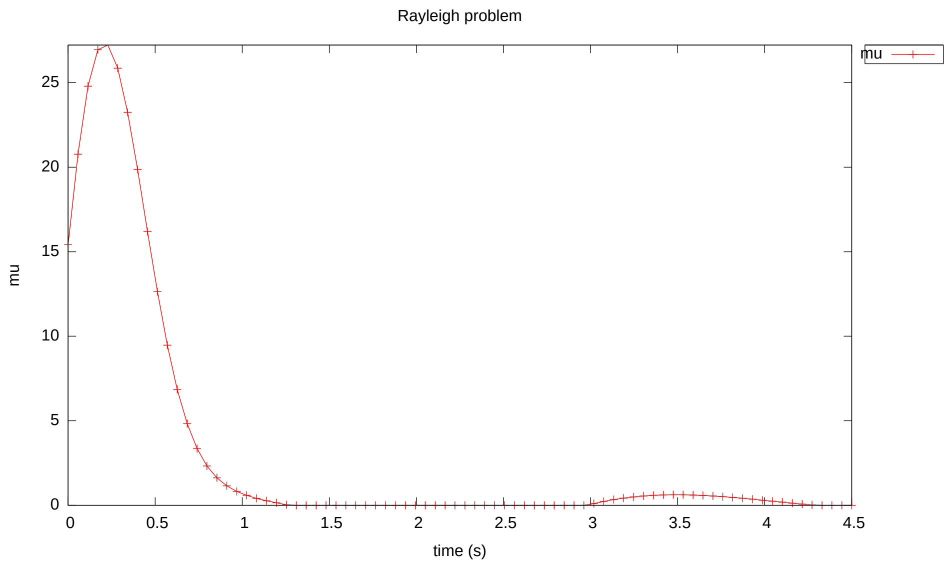

Figure 3.85: Path constraint multiplier for Rayleigh problem

The path constraints:

$$
\begin{aligned}
(x(t)-0.4)^2+(y(t)-0.5)^2 &\geq 0.1 \\
(x(t)-0.8)^2+(y(t)-1.5)^2 &\geq 0.1,
\end{aligned} \tag{3.139}
$$

and the boundary conditions:

$$
\begin{array}{rcl}
x(0) & = & 0 \\
y(0) & = & 0 \\
x(t_f) & = & 1.2 \\
y(t_f) & = & 1.6
\end{array} \tag{3.140}
$$

where $t_f = 1.0$, and $V = 2.138$.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figure 3.86, which illustrates the optimal (x, y) trajectory of the particle.

```
PSOPT results summary
=====================

Problem:  Obstacle avoidance problem
CPU time (seconds): 6.441073e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 13:29:46 2020

Optimal (unscaled) cost function value:  4.571044e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  4.571044e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 9.265077e-03
NLP solver reports:  The problem has been solved!
```

3.35 Reorientation of an asymmetric rigid body

Consider the following optimal control problem, which consists of the reorientation of an asymmetric rigid body in minimum time [4]. Find t_f, $\hat{\mathbf{u}}(t) = [u_1(t), u_2(t), u_3(t), q_4(t)]^T$ to minimize the cost functional

$$
J = t_f \tag{3.141}
$$

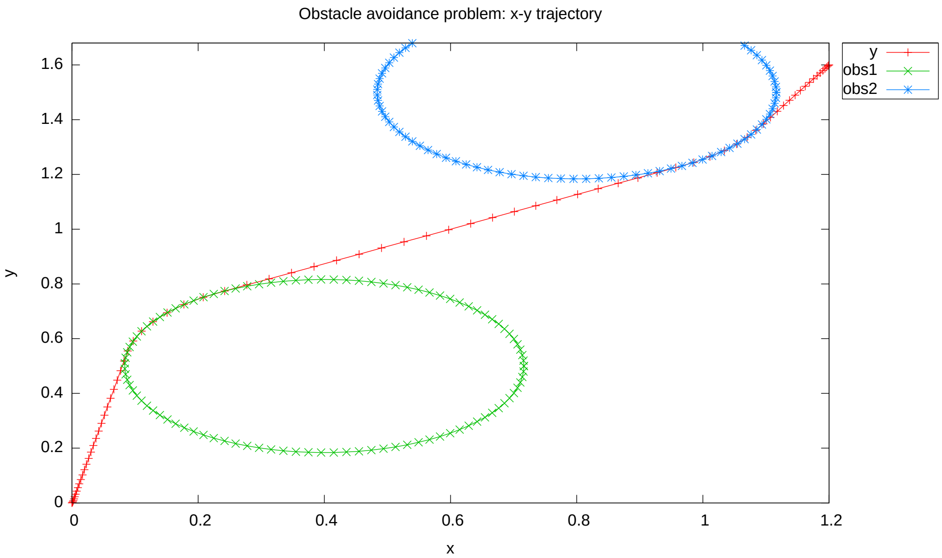

Figure 3.86: Optimal (x, y) trajectory for obstacle avoidance problem

subject to the dynamic constraints

$$\begin{aligned}
\dot{q}_1 &= \frac{1}{2}\left[\omega_1 q_4 - \omega_2 q_3 + \omega_3 q_2\right] \\
\dot{q}_2 &= \frac{1}{2}\left[\omega_1 q_3 + \omega_2 q_4 - \omega_3 q_1\right] \\
\dot{q}_3 &= \frac{1}{2}\left[-\omega_1 q_2 + \omega_2 q_1 + \omega_3 q_4\right] \\
\dot{\omega}_1 &= \frac{u_1}{I_x} - \left[\frac{I_z - I_y}{I_x}\omega_2\omega_3\right] \\
\dot{\omega}_2 &= \frac{u_2}{I_y} - \left[\frac{I_x - I_z}{I_y}\omega_1\omega_3\right] \\
\dot{\omega}_3 &= \frac{u_3}{I_z} - \left[\frac{I_y - I_x}{I_z}\omega_1\omega_2\right]
\end{aligned} \tag{3.142}$$

The path constraint:

$$0 = q_1^2 + q_2^2 + q_3^2 + q_4^2 - 1 \tag{3.143}$$

the boundary conditions:

$$
\begin{aligned}
q_1(0) &= 0, \\
q_2(0) &= 0, \\
q_3(0) &= 0, \\
q_4(0) &= 1.0 \\
q_1(t_f) &= \sin\frac{\phi}{2}, \\
q_2(t_f) &= 0, \\
q_3(t_f) &= 0, \\
q_4(t_f) &= \cos\frac{\phi}{2} \\
\omega_1(0) &= 0, \\
\omega_2(0) &= 0, \\
\omega_3(0) &= 0, \\
\omega_1(t_f) &= 0, \\
\omega_2(t_f) &= 0, \\
\omega_3(t_f) &= 0,
\end{aligned}
\tag{3.144}
$$

where $\phi = 150\,\mathrm{deg}$ is the Euler axis rotation angle, $\mathbf{q} = [q_1, q_2, q_3, q_4]^T$ is the quarternion vector, $\omega = [\omega_1, \omega_2, \omega_3]^T$ is the angular velocity vector, and $\mathbf{u} = [u_1, u_2, u_3]^T$ is the control vector. Note that in the implementation, variable $q_4(t)$ is treated as an algebraic variable (i.e. as a control variable).

The variable bounds and other parameters are given in the code.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.87 to 3.88, which contain the elements of the quarternion vector $\mathbf{q}$, and the control vector $\mathbf{u} = [u_1, u_2, u_3]^T$, respectively.

```
PSOPT results summary
=====================

Problem:  Reorientation of a rigid body
CPU time (seconds): 4.627959e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:27:17 2020

Optimal (unscaled) cost function value:  2.863096e+01
Phase 1 endpoint cost function value: 2.863096e+01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
```

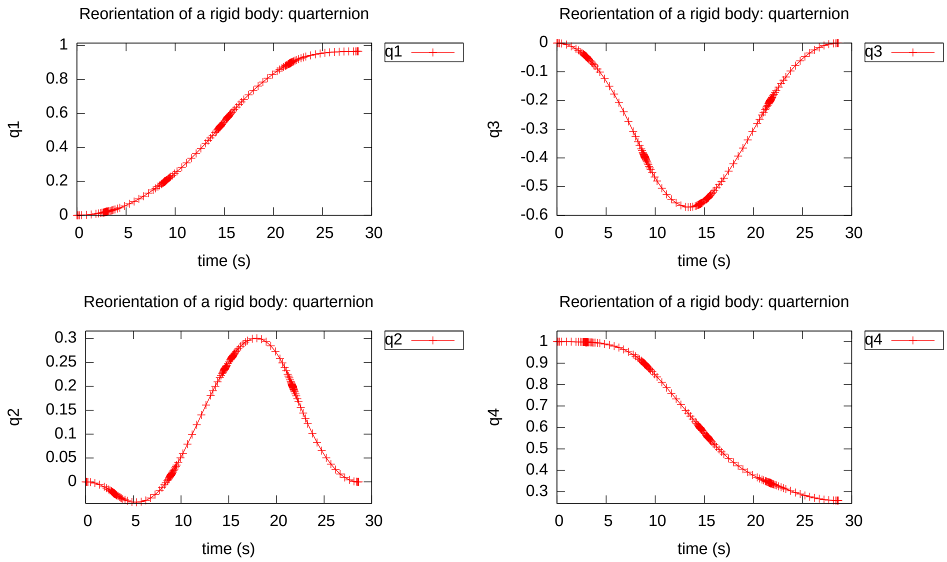

Figure 3.87: Quarternion vector elements for the reorientation problem

```
Phase 1 final time: 2.863096e+01
Phase 1 maximum relative local error: 1.393507e-05
NLP solver reports:  The problem has been solved!
```

3.36 Shuttle re-entry problem

Consider the following optimal control problem, which is known in the literature as the shuttle re-entry problem [3]. Find t_f, $\alpha(t)$ and $\beta(t) \in [0, t_f]$ to minimize the cost functional

$$J = -\frac{180}{\pi}\theta(t_f) \tag{3.145}$$

subject to the dynamic constraints

$$\begin{aligned}
\dot{h} &= v\sin(\gamma) \\
\dot{\phi} &= \tfrac{v}{r}\cos(\gamma)\sin(\psi)/\cos(\theta) \\
\dot{m} &= \tfrac{v}{r}\cos(\gamma)\cos(\psi) \\
\dot{v} &= -\tfrac{D}{m} - g\sin(\gamma) \\
\dot{\gamma} &= \tfrac{L}{mv}\cos(\beta) + \cos(\gamma)(\tfrac{v}{r} - \tfrac{g}{v}) \\
\dot{\psi} &= \tfrac{1}{mv\cos(\gamma)}L\sin(\beta) + \tfrac{v}{r\cos(\theta)}\cos(\gamma)\sin(\psi)\sin(\theta)
\end{aligned} \tag{3.146}$$

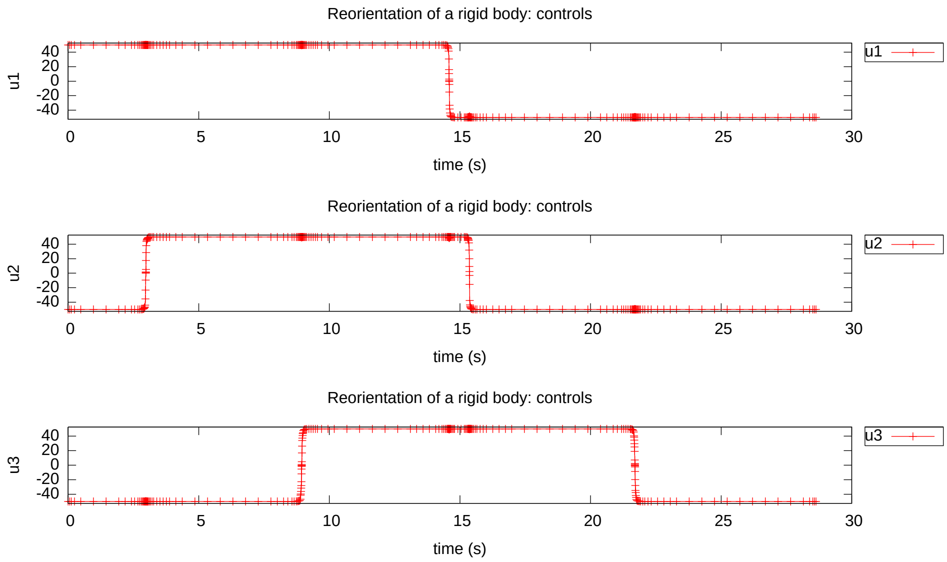

Figure 3.88: Control vector elements for the reorientation problem

the boundary conditions:

$$\begin{aligned} h(0) &= 260000.0 \\ \phi(0) &= -0.6572 \\ \theta(0) &= 0.0 \\ v(0) &= 25600.0 \\ \gamma(0) &= -0.0175 \\ h(t_f) &= 80000.0 \\ v(t_f) &= 2500.0 \\ \gamma(t_f) &= -0.0873 \end{aligned} \tag{3.147}$$

The variable bounds and other parameters are given in the code.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.89 to 3.96, which contain the elements of the state and the control vectors.

```
PSOPT results summary
=====================

Problem:  Shuttle re-entry problem
CPU time (seconds): 9.641502e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:27:38 2020

Optimal (unscaled) cost function value:  -3.414119e+01
```

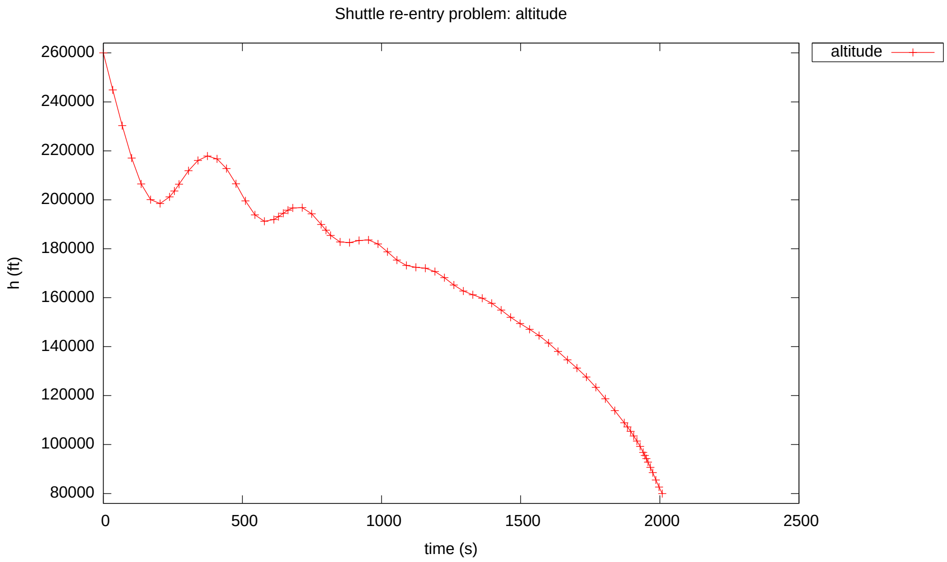

Figure 3.89: Altitude $h(t)$ for the shuttle re-entry problem

```
Phase 1 endpoint cost function value: -3.414119e+01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.008600e+03
Phase 1 maximum relative local error: 5.879622e-04
NLP solver reports:  The problem has been solved!
```

3.37 Singular control problem

Consider the following optimal control problem, whose solution is known to have a singular arc [27, 34]. Find $u(t), t \in [0, 1]$ to minimize the cost functional

$$J = \int_0^1 [x_1^2 + x_2^2 + 0.0005(x_2 + 16x_5 - 8 - 0.1x_3u^2)^2]dt \tag{3.148}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -x_3u + 16t - 8 \\ \dot{x}_3 &= u \end{aligned} \tag{3.149}$$

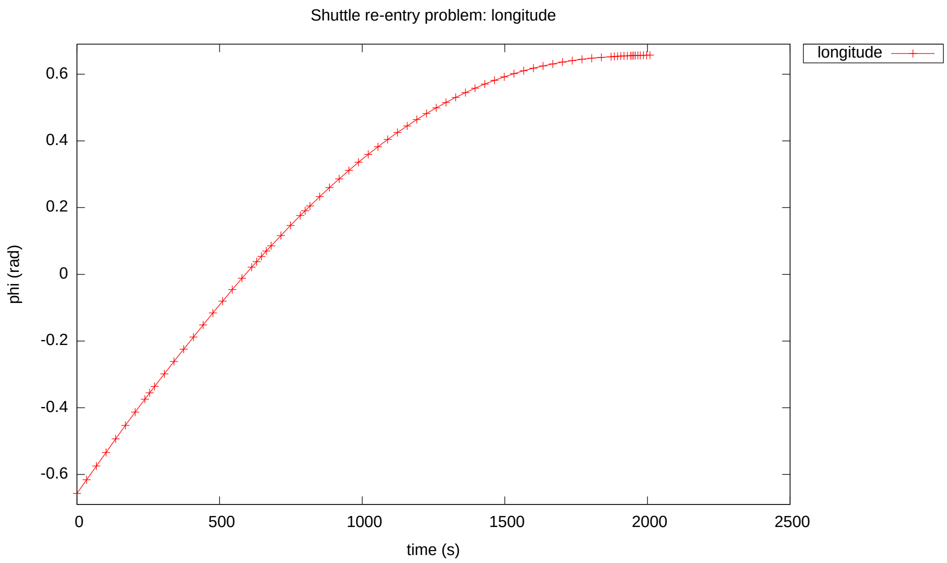

Figure 3.90: Longitude $\phi(t)$ for the shuttle re-entry problem

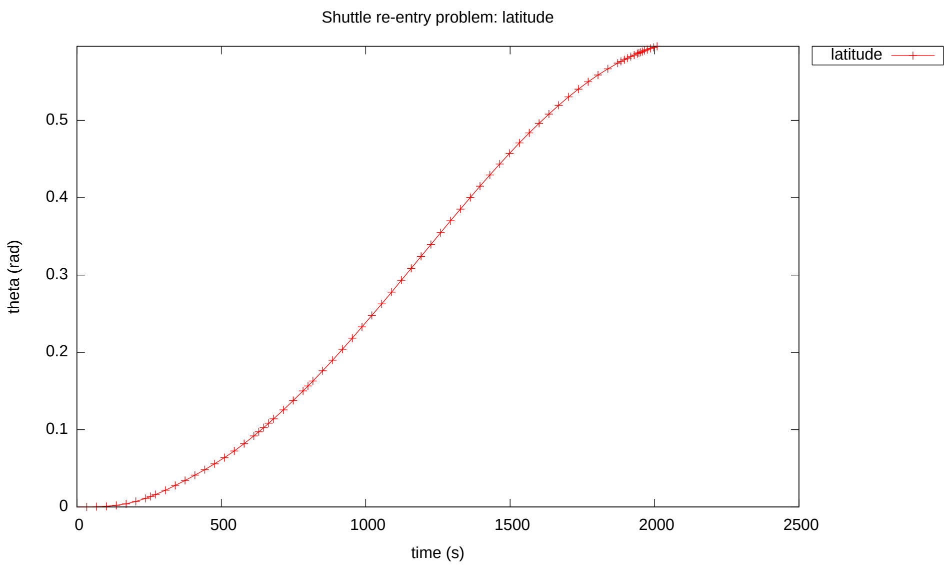

Figure 3.91: Latitude $\theta(t)$ for the shuttle re-entry problem

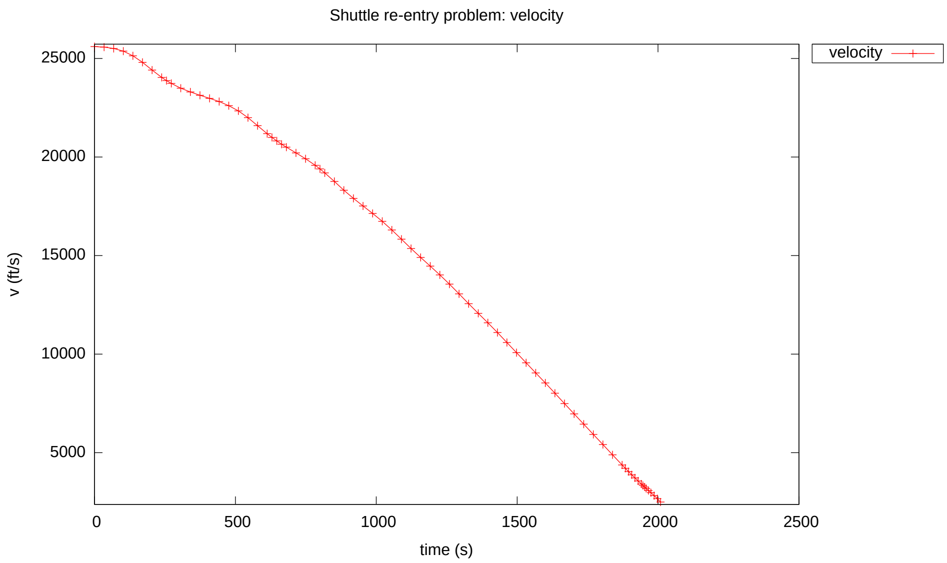

Figure 3.92: Velocity $v(t)$ for the shuttle re-entry problem

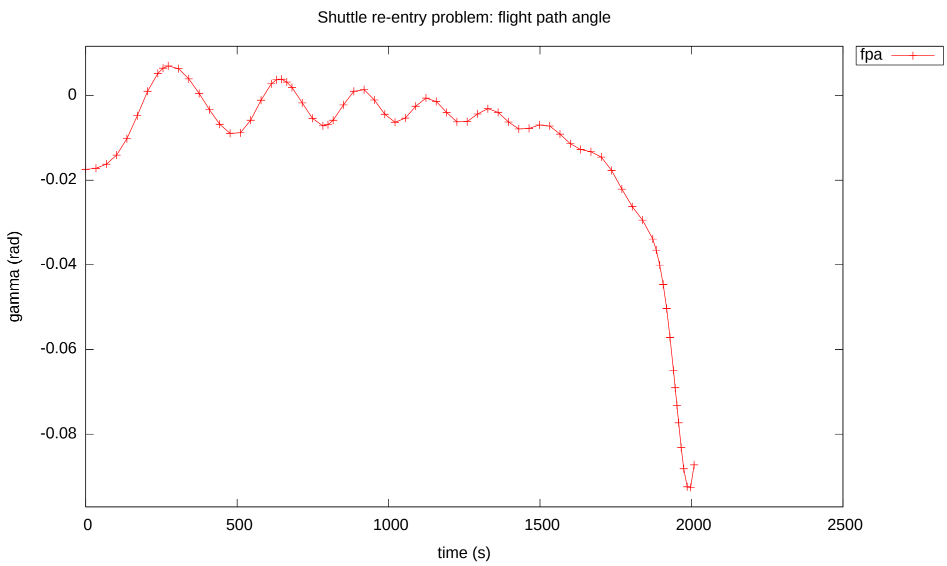

Figure 3.93: Flight path angle $\gamma(t)$ for the shuttle re-entry problem

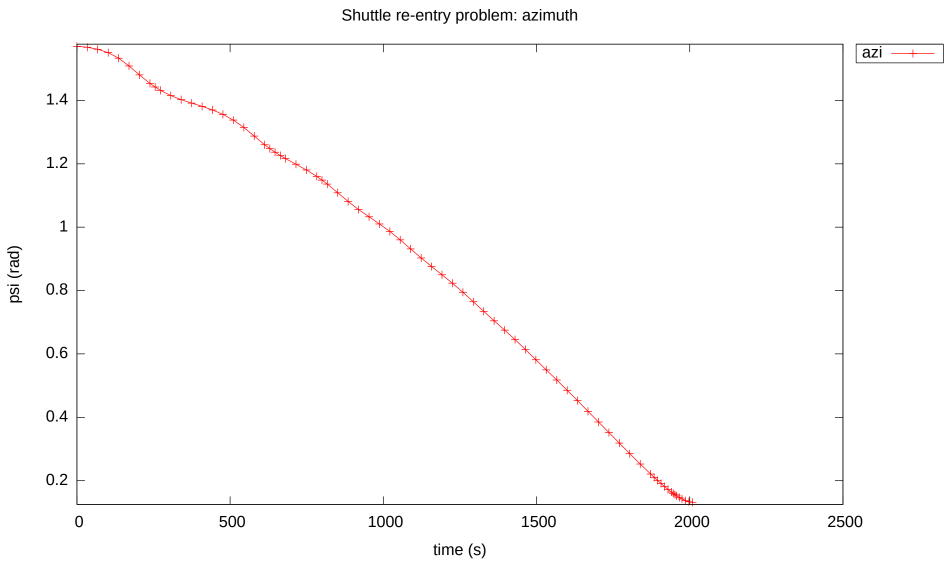

Figure 3.94: Azimuth $\psi(t)$ for the shuttle re-entry problem

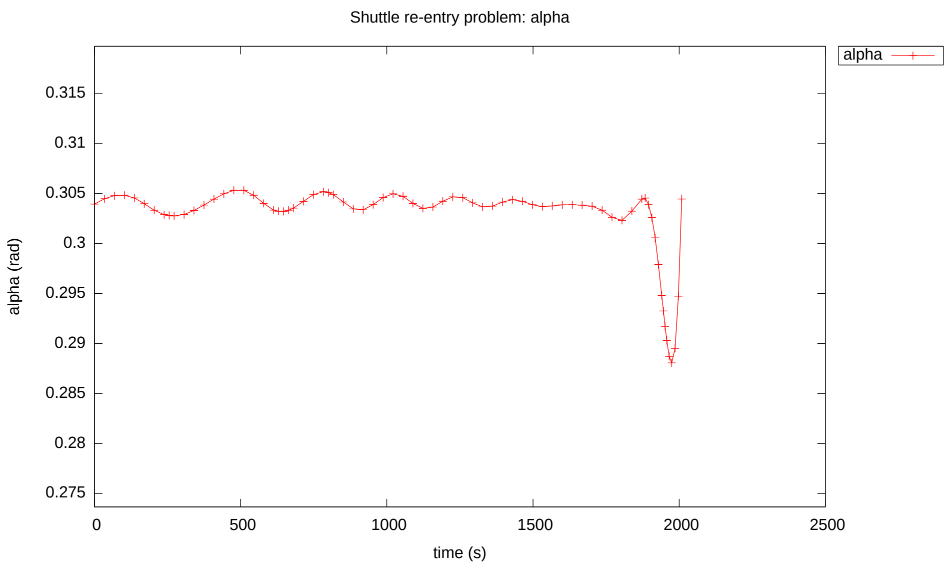

Figure 3.95: Angle of attack $\alpha(t)$ for the shuttle re-entry problem

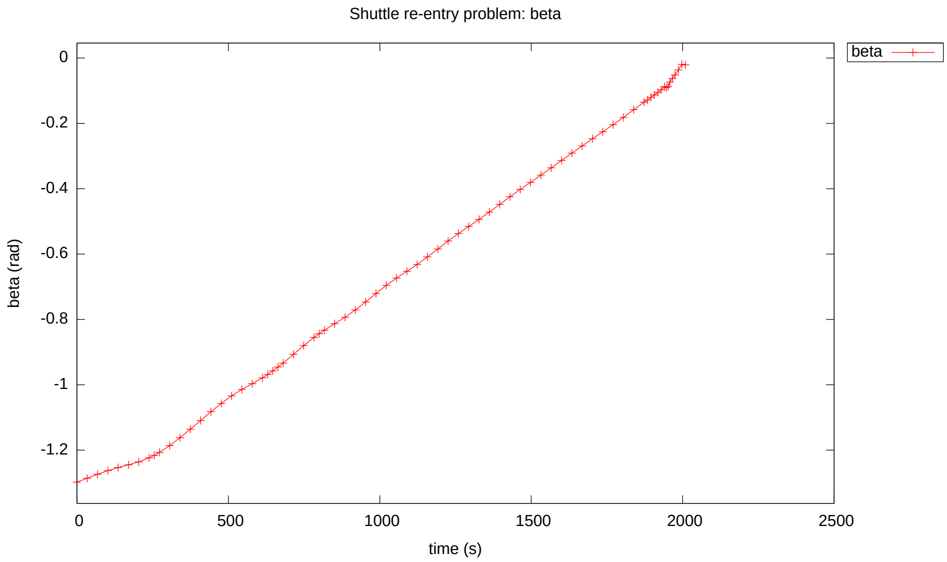

Figure 3.96: Bank angle $\beta(t)$ for the shuttle re-entry problem

the boundary conditions:

$$\begin{aligned} x_1(0) &= 0 \\ x_2(0) &= -1 \\ x_3(0) &= \sqrt{5} \end{aligned} \tag{3.150}$$

and the control bounds

$$-4 \leq u(t) \quad \leq 10 \tag{3.151}$$

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.97 and 3.98, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Singular control 5
CPU time (seconds): 1.654943e+01
NLP solver used:   IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:28:03 2020

Optimal (unscaled) cost function value:  1.192620e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  1.192620e-01
Phase 1 initial time: 0.000000e+00
```

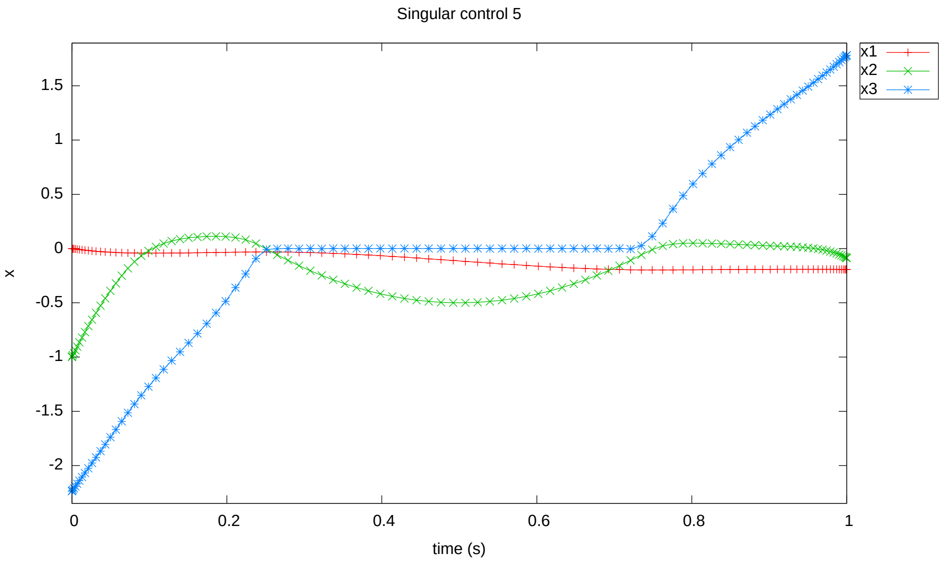

Figure 3.97: States for singular control problem

```
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 2.552542e-04
NLP solver reports:  The problem has been solved!
```

3.38 Time varying state constraint problem

Consider the following optimal control problem, which involves a time varying state constraint [38]. Find $u(t) \in [0, 1]$ to minimize the cost functional

$$J = \int_0^1 [x_1^2(t) + x_2^2(t) + 0.005u^2(t)]dt \tag{3.152}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -x_2 + u \end{aligned} \tag{3.153}$$

the boundary conditions:

$$\begin{aligned} x_1(0) &= 0 \\ x_2(0) &= -1 \end{aligned} \tag{3.154}$$

and the path constraint

$$x_2 \leq 8(t - 0.5)^2 - 0.5 \tag{3.155}$$

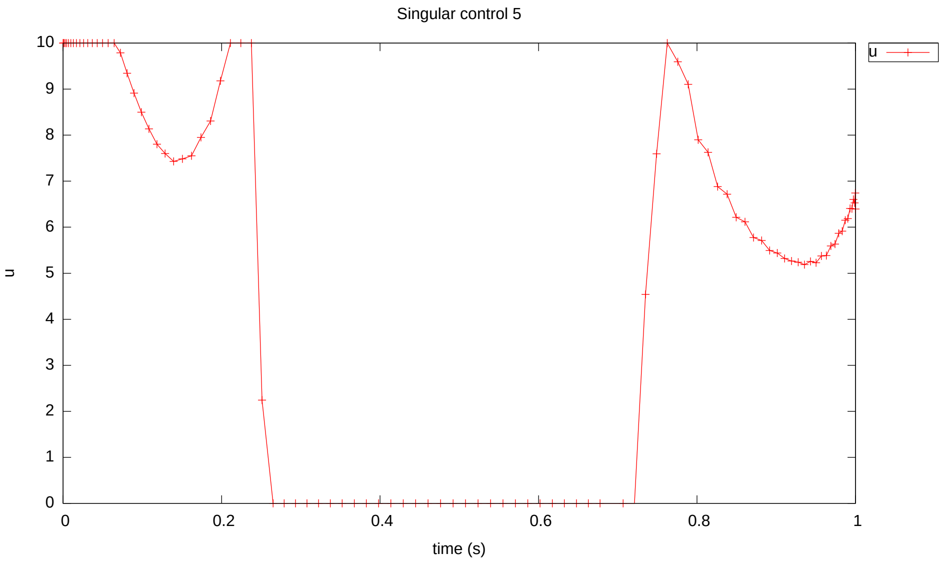

Figure 3.98: Control for singular control problem

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.99 and 3.100, which contain the elements of the states with the boundary of the constraint on x_2, and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Time varying state constraint problem
CPU time (seconds): 1.629702e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:28:13 2020

Optimal (unscaled) cost function value:  1.698180e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  1.698180e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 3.216699e-05
NLP solver reports:  The problem has been solved!
```

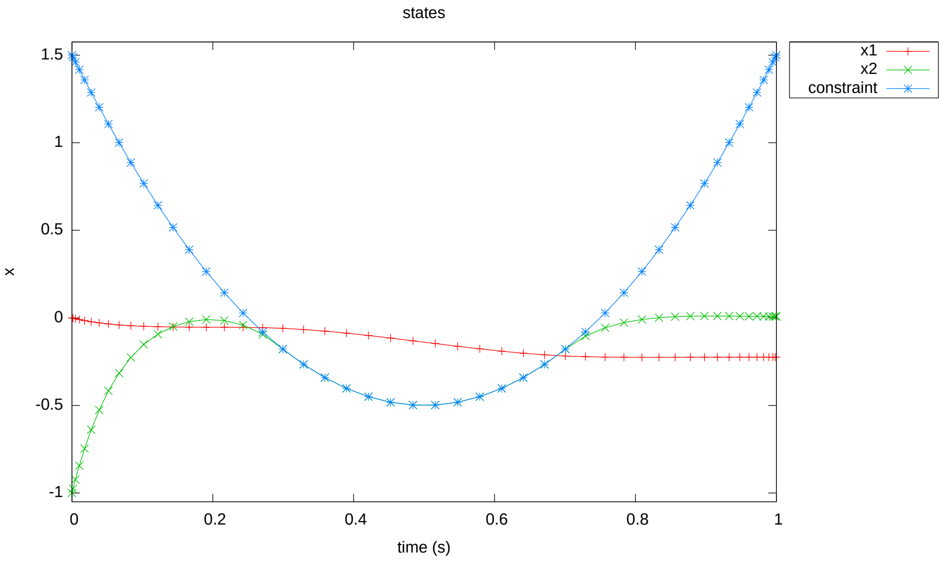

Figure 3.99: States for time-varying state constraint problem

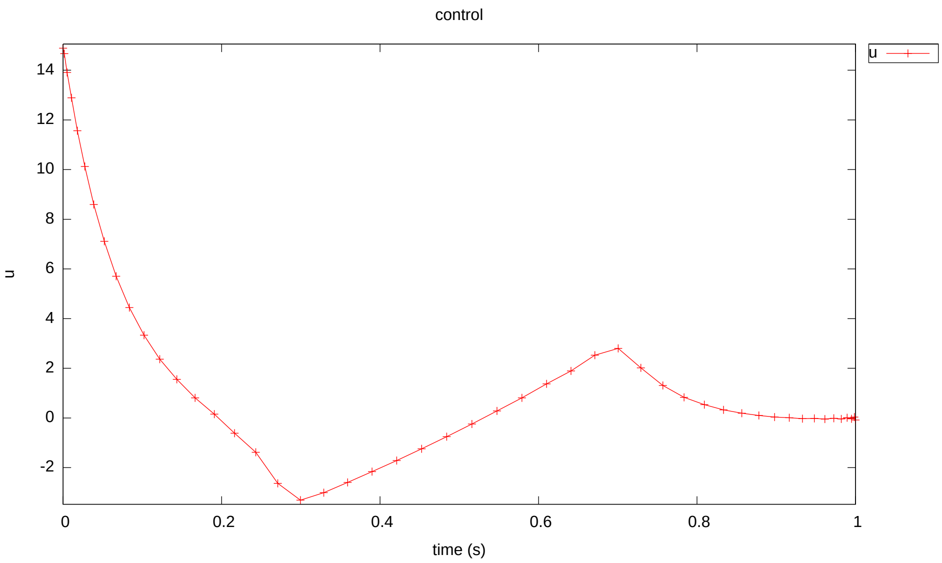

Figure 3.100: Control for time-varying state constraint problem

3.39 Two burn orbit transfer

The goal of this problem is to compute a trajectory for an spacecraft to go from a standard space shuttle park orbit to a geosynchronous final orbit. It is assumed that the engines operate over two short periods during the mission, and it is desired to compute the timing and duration of the burn periods, as well as the instantaneous direction of the thrust during these two periods, to maximise the final weight of the spacecraft. The problem is described in detail by Betts [3]. The mission then involves four phases: coast, burn, coast and burn. The problem is formulated as follows. Find $\mathbf{u}(t) = [\theta(t), \phi(t)]^T, t \in [t_f^{(1)}, t_f^{(2)}]$ and $t \in [t_f^{(3)}, t_f^{(4)}]$, and the instants $t_f^{(1)}, t_f^{(2)}, t_f^{(3)}, t_f^{(4)}$ such that the following objective function is minimised:

$$J = -w(t_f) \tag{3.156}$$

subject to the dynamic constraints for phases 1 and 3:

$$\dot{\mathbf{y}} = \mathbf{A}(\mathbf{y})\Delta_g + \mathbf{b} \tag{3.157}$$

the following dynamic constraints for phases 2 and 4:

$$\begin{aligned}\dot{\mathbf{y}} &= \mathbf{A}(\mathbf{y})\Delta + \mathbf{b} \\ \dot{w} &= -T/I_{sp}\end{aligned} \tag{3.158}$$

and the following linkages between phases

$$\begin{aligned}\mathbf{y}(t_f^{(1)}) &= \mathbf{y}(t_0^{(2)}) \\ \mathbf{y}(t_f^{(2)}) &= \mathbf{y}(t_0^{(3)}) \\ \mathbf{y}(t_f^{(3)}) &= \mathbf{y}(t_0^{(4)}) \\ t_f^{(1)} &= t_0^{(2)} \\ t_f^{(2)} &= t_0^{(3)} \\ t_f^{(3)} &= t_0^{(4)} \\ w(t_f^{(2)}) &= w(t_0^{(4)})\end{aligned} \tag{3.159}$$

where $\mathbf{y} = [p, f, g, h, k, L, w]^T$ is the vector of modified equinoctial elements, w is the spacecraft weight, I_{sp} is the specific impulse of the engine, T is the maximum thrust, expressions for $\mathbf{A}(\mathbf{y})$ and $\mathbf{b}$ are given in [3]. the disturbing acceleration is $\Delta = \Delta_g + \Delta_T$, where Δ_g is the gravitational disturbing acceleration due to the oblatness of Earth (given in [3]), and Δ_T is the thurst acceleration, given by:

$$\Delta_T = \mathbf{Q}_r\mathbf{Q}_v \begin{bmatrix} T_a \cos\theta\cos\phi \\ T_a \cos\theta\sin\phi \\ T_a \sin\theta \end{bmatrix} \tag{3.160}$$

where $T_a(t) = g_0 T/w(t)$, g_0 is a constant, θ is the pitch angle and ϕ is the yaw angle of the thurst, matrix $\mathbf{Q}_v$ is given by:

$$\mathbf{Q}_v = \left[\frac{\mathbf{v}}{||\mathbf{v}||}, \frac{\mathbf{v} \times r}{||\mathbf{v} \times \mathbf{r}||}, \frac{\mathbf{v}}{||\mathbf{v}||} \times \frac{\mathbf{v} \times r}{||\mathbf{v} \times \mathbf{r}||} \right] \tag{3.161}$$

matrix $\mathbf{Q}_r$ is given by:

$$\mathbf{Q}_r = \begin{bmatrix} \mathbf{i}_r & \mathbf{i}_\theta & \mathbf{i}_h \end{bmatrix} = \begin{bmatrix} \frac{\mathbf{r}}{||\mathbf{r}||} & \frac{(\mathbf{r}\times\mathbf{v})\times\mathbf{r}}{||\mathbf{r}\times\mathbf{v}||||\mathbf{r}||} & \frac{(\mathbf{r}\times\mathbf{v})}{||\mathbf{r}\times\mathbf{v}||} \end{bmatrix} \tag{3.162}$$

The boundary conditions of the problem are given by:

$$\begin{aligned}
p(0) &= 218327080.052835 \\
f(0) &= 0 \\
g(0) &= 0 \\
h(0) &= 0 \\
h(0) &= 0 \\
k(0) &= 0 \\
L(0) &= \pi \text{ (rad)} \\
w(0) &= 1 \text{ (lb)} \\
p(t_f) &= 19323/\sigma + R_e \\
f(t_f) &= 0 \\
g(t_f) &= 0 \\
h(t_f) &= 0 \\
k(t_f) &= 0
\end{aligned} \tag{3.163}$$

and the values of the parameters are: $g_0 = 32.174$ (ft/sec^2), $I_{sp} = 300$ (sec), $T = 1.2$ (lb), $\mu = 1.407645794 \times 10^{16}$ (ft^3/sec^2), $R_e = 20925662.73$ (ft), $\sigma = 1.0/6076.1154855643$, $J_2 = 1082.639 \times 10^{-6}$, $J_3 = -2.565 \times 10^{-6}$, $J_4 = -1.608 \times 10^{-6}$.

An initial guess was computed by forward propagation from the initial conditions, assuming the following guesses for the controls and burn periods [3]:

$$\begin{aligned}
\mathbf{u}(t) &= \begin{bmatrix} 0.148637 \times 10^{-2}, & -9.08446 \end{bmatrix}^T \quad t \in [2840, 21650] \\
\mathbf{u}(t) &= \begin{bmatrix} -0.136658 \times 10^{-2}, & 49.7892 \end{bmatrix} \quad t \in [21650, 21700]
\end{aligned} \tag{3.164}$$

The problem was solved using local collocation (trapezoidal followed by Hermite-Simpson) with automatic mesh refinement.

The output from $\mathcal{PSOPT}$ is summarised in the box below. The controls during the burn periods are shown Figures 3.101 to 3.104, which show the control variables during phases 2 and 4, and Figure 3.105, which shows the trajectory in cartesian co-ordinates.

```
PSOPT results summary
=====================

Problem:  Two burn transfer problem
CPU time (seconds): 8.360238e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:28:38 2020

Optimal (unscaled) cost function value:  -2.367249e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.609965e+03
Phase 1 maximum relative local error: 1.450810e-06
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  0.000000e+00
Phase 2 initial time: 2.609965e+03
Phase 2 final time: 2.751427e+03
Phase 2 maximum relative local error: 1.260956e-06
Phase 3 endpoint cost function value: 0.000000e+00
Phase 3 integrated part of the cost:  0.000000e+00
Phase 3 initial time: 2.751427e+03
Phase 3 final time: 2.163413e+04
Phase 3 maximum relative local error: 1.304951e-05
Phase 4 endpoint cost function value: -2.367249e-01
Phase 4 integrated part of the cost:  0.000000e+00
Phase 4 initial time: 2.163413e+04
Phase 4 final time: 2.168348e+04
Phase 4 maximum relative local error: 5.531391e-06
NLP solver reports:  The problem has been solved!
```

3.40 Two link robotic arm

Consider the following optimal control problem [27]. Find t_f, and $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = t_f \tag{3.165}$$

Table 3.9: Mesh refinement statistics: Two burn transfer problem

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	$\epsilon_{\max}$	$\mathrm{CPU_a}$
1	TRP	40	308	298	20	20	20	0	1520	4.942e-02	3.986e-01
2	TRP	56	428	402	24	25	23	0	2700	4.130e-03	2.630e-01
3	H-S	76	650	532	38	39	37	0	8580	1.571e-04	8.635e-01
4	H-S	104	888	714	65	66	61	0	20064	2.320e-05	1.938e+00
5	H-S	138	1162	932	74	75	74	0	30450	1.305e-05	2.728e+00
$\mathrm{CPU_b}$	-	-	-	-	-	-	-	-	-	-	2.169e+00
-	-	-	-	-	221	225	215	0	63314	-	8.360e+00

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, $\epsilon_{\max}$ = maximum relative ODE error, $\mathrm{CPU_a}$ = CPU time in seconds spent by NLP algorithm, $\mathrm{CPU_b}$ = additional CPU time in seconds spent by PSOPT

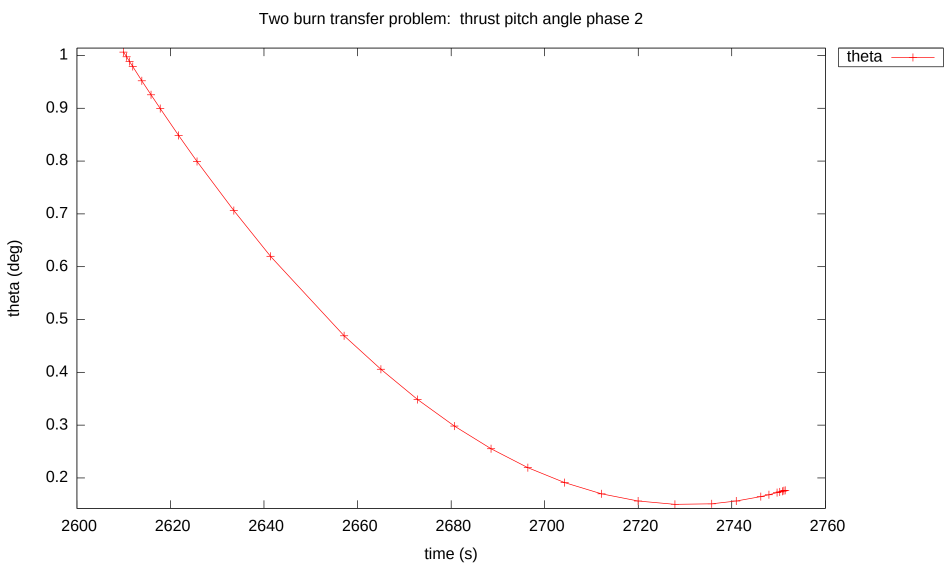

Figure 3.101: Pitch angle during phase 2

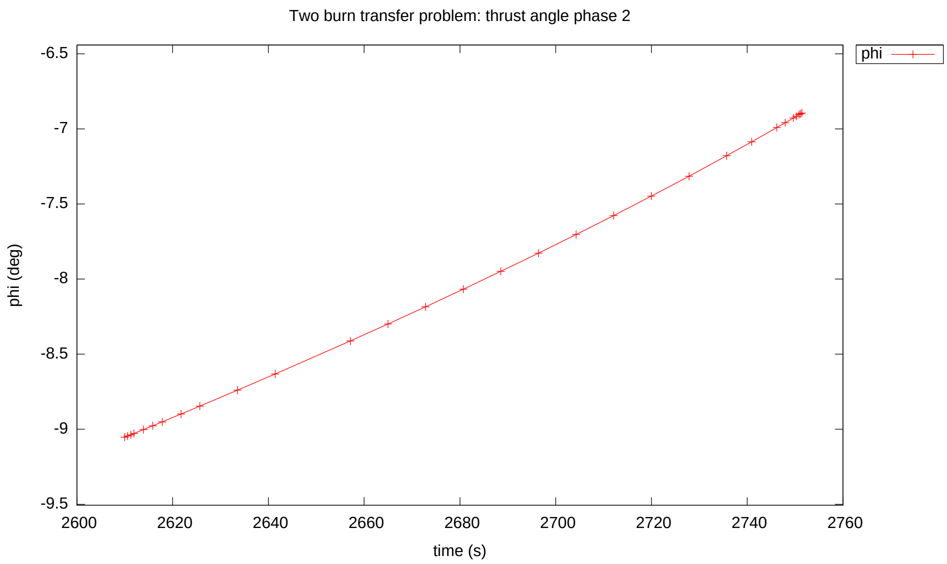

Figure 3.102: Yaw angle during phase 2

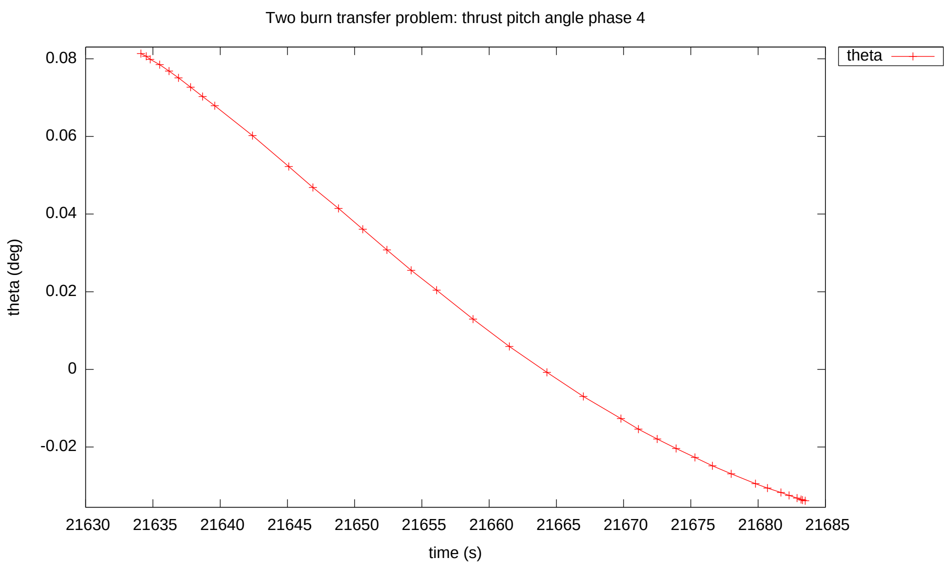

Figure 3.103: Pitch angle during phase 4

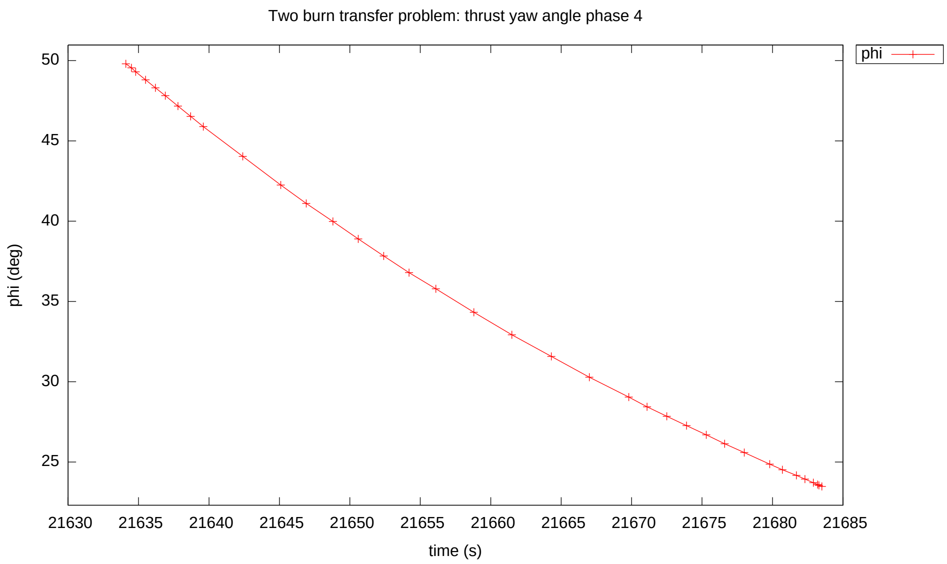

Figure 3.104: Yaw angle during phase 4

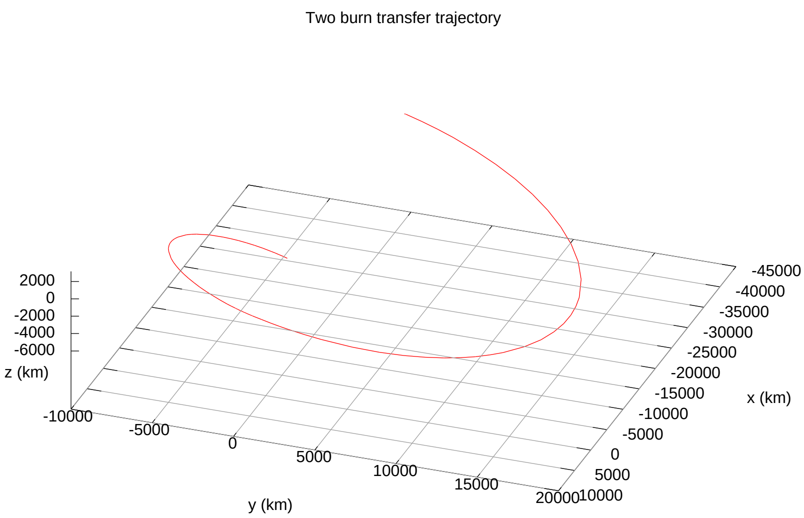

Figure 3.105: Two burn transfer trajectory

subject to the dynamic constraints

$$
\begin{aligned}
\dot{x}_1 &= \frac{\sin(x_3)(\frac{9.0}{4.0}\cos(x_3)x_1^2+2*x_2^2)+\frac{4.0}{3.0}(u_1-u_2)-\frac{3.0}{2.0}\cos(x_3)u_2}{\frac{31.0}{36.0}+\frac{9.0}{4.0\sin^2(x_3)}} \\
\dot{x}_2 &= \frac{-(\sin(x_3)*(\frac{7.0}{2.0}*x_1^2+\frac{9.0}{4.0}\cos(x_3)x_2^2)-\frac{7.0}{3.0}u_2+\frac{3.0}{2.0}\cos(x_3)(u_1-u_2))}{\frac{31.0}{36.0}+\frac{9.0}{4.0\sin^2(x_3)}} \\
\dot{x}_3 &= x_2 - x_1 \\
\dot{x}_4 &= x_1
\end{aligned} \tag{3.166}
$$

the boundary conditions:

$$
\begin{array}{rclrcl}
x_1(0) &=& 0 & x_1(t_f) &=& 0 \\
x_2(0) &=& 0 & x_2(t_f) &=& 0 \\
x_3(0) &=& 0.5 & x_3(t_f) &=& 0.5 \\
x_4(0) &=& 0.0 & x_4(t_f) &=& 0.522
\end{array} \tag{3.167}
$$

The control bounds:

$$
\begin{aligned}
-1 \leq & u_1(t) \leq 1 \\
-1 \leq & u_2(t) \leq 1
\end{aligned} \tag{3.168}
$$

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.106 and 3.107, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Two link robotic arm
CPU time (seconds): 2.047532e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:28:55 2020

Optimal (unscaled) cost function value:  2.988662e+00
Phase 1 endpoint cost function value: 2.988662e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.988662e+00
Phase 1 maximum relative local error: 3.815625e-04
NLP solver reports:  The problem has been solved!
```

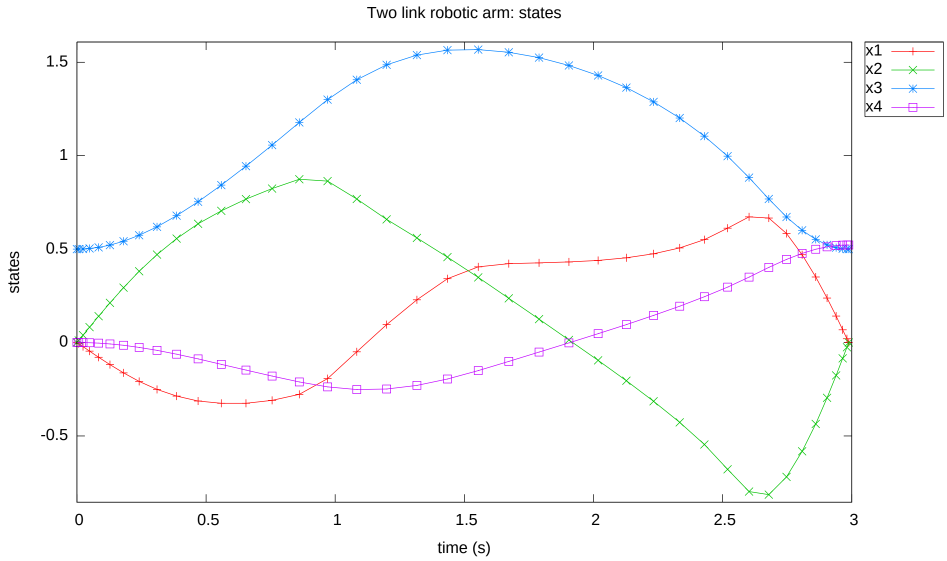

Figure 3.106: States for two-link robotic arm problem

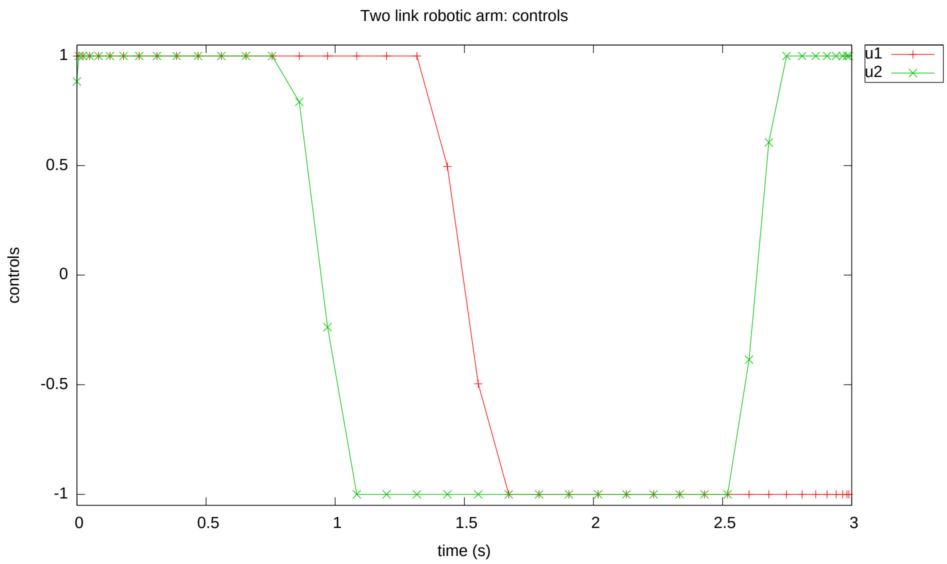

Figure 3.107: Controls for two link robotic arm problem

3.41 Two-phase path tracking robot

Consider the following two-phase optimal control problem, which consists of a robot following a specified path [36, 34]. Find $u(t) \in [0, 2]$ to minimize the cost functional

$$\begin{aligned} J = \int_0^2 &[100(x_1 - x_{1,ref})^2 + 100(x_2 - x_{2,ref})^2 \\ &+500(x_3 - x_{3,ref})^2 + 500(x_4 - x_{4,ref})^2]dt \end{aligned} \tag{3.169}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_3 \\ \dot{x}_2 &= x_4 \\ \dot{x}_3 &= u_1 \\ \dot{x}_4 &= u_2 \end{aligned} \tag{3.170}$$

the boundary conditions:

$$\begin{array}{rclrcl} x_1(0) &=& 0 & x_1(2) &=& 0.5 \\ x_2(0) &=& 0 & x_2(2) &=& 0.5 \\ x_3(0) &=& 0.5 & x_3(2) &=& 0 \\ x_4(0) &=& 0.0 & x_4(2) &=& 0.5 \end{array} \tag{3.171}$$

where the reference signals are given by:

$$\begin{aligned} x_{1,ref} &= \frac{t}{2}\,(0 \le t < 1), \frac{1}{2}\,(1 \le t \le 2) \\ x_{2,ref} &= 0\,(0 \le t < 1), \frac{t-1}{2}\,(1 \le t \le 2) \\ x_{3,ref} &= \frac{1}{2}\,(0 \le t < 1), 0\,(1 \le t \le 2) \\ x_{4,ref} &= 0\,(0 \le t < 1), \frac{1}{2}\,(1 \le t \le 2) \end{aligned} \tag{3.172}$$

Note that the first phase covers the period $t \in [0, 1]$, while the second phase covers the period $t \in [1, 2]$.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.108 and 3.109, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Two phase path tracking robot
CPU time (seconds): 8.766510e-01
NLP solver used:  IPOPT
```

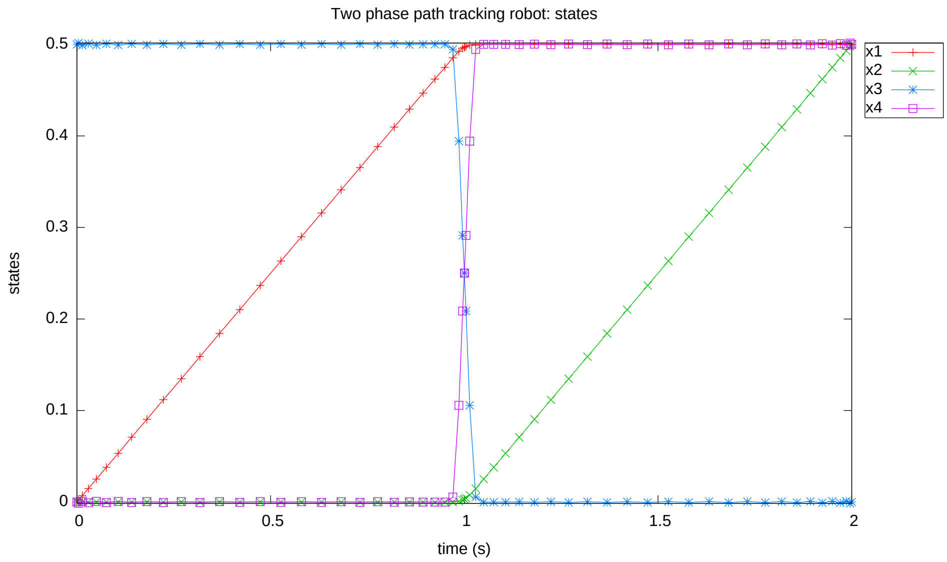

Figure 3.108: States for two-phase path tracking robot problem

```
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:29:12 2020

Optimal (unscaled) cost function value:  1.042568e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.212840e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 3.443287e-04
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  5.212840e-01
Phase 2 initial time: 1.000000e+00
Phase 2 final time: 2.000000e+00
Phase 2 maximum relative local error: 3.443298e-04
NLP solver reports:  The problem has been solved!
```

3.42 Two-phase Schwartz problem

Consider the following two-phase optimal control problem [34]. Find $u(t) \in [0, 2.9]$ to minimize the cost functional

$$J = 5(x_1(t_f)^2 + x_2(t_f)^2) \tag{3.173}$$

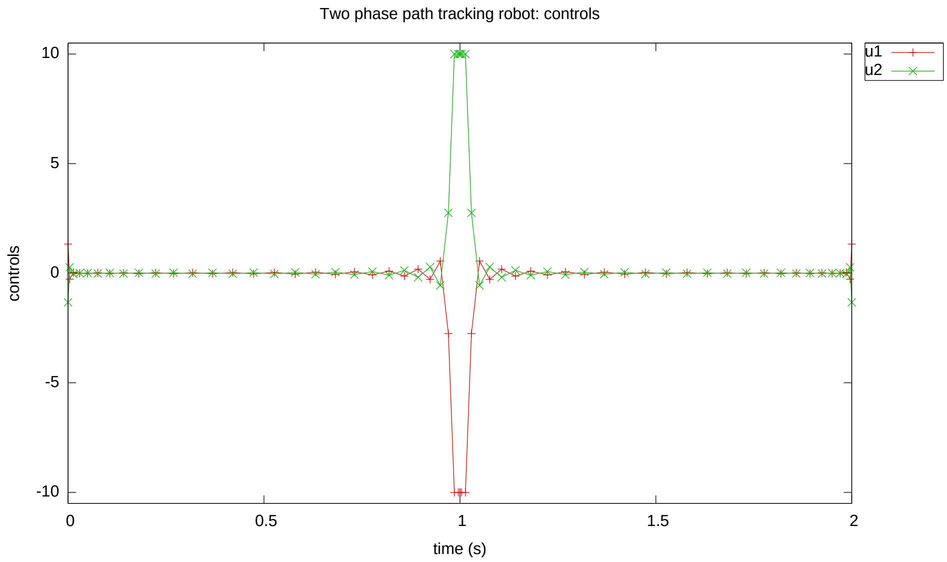

Figure 3.109: Control for two phase path tracking robot problem

subject to the dynamic constraints

$$\begin{aligned}\dot{x}_1 &= x_2 \\ \dot{x}_2 &= u - 0.1(1 + 2x_1^2)x_2\end{aligned} \tag{3.174}$$

the boundary conditions:

$$\begin{aligned}x_1(0) &= 1 \\ x_2(0) &= 1\end{aligned} \tag{3.175}$$

and the constraints for $t < 1$:

$$\begin{aligned}1-9(x_1 - 1)^2 - \left(\frac{x_2 - 0.4}{0.3}\right)^2 \leq 0 \\ -0.8 \leq x_2 \\ -1 \leq u \leq 1\end{aligned} \tag{3.176}$$

The problem has been divided into two phases. The first phase covers the period $t \in [0, 1]$, while the second phase covers the period $t \in [1, 2.9]$.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.110 and 3.111, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================
```

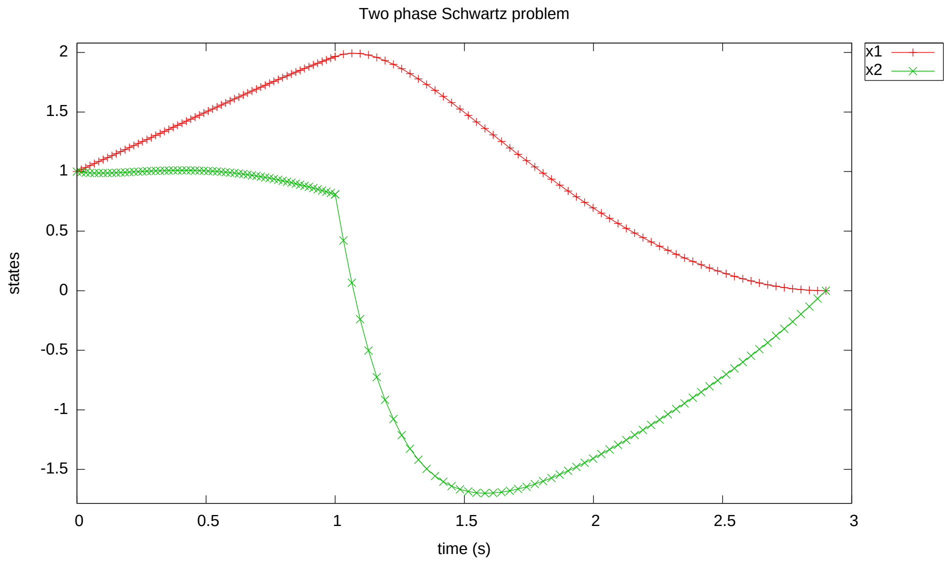

Figure 3.110: States for two-phase Schwartz problem

```
Problem:  Two phase Schwartz problem
CPU time (seconds): 6.095880e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:29:23 2020

Optimal (unscaled) cost function value:  3.138721e-16
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 3.948801e-03
Phase 2 endpoint cost function value: 3.138721e-16
Phase 2 integrated part of the cost:  0.000000e+00
Phase 2 initial time: 1.000000e+00
Phase 2 final time: 2.900000e+00
Phase 2 maximum relative local error: 2.254848e-02
NLP solver reports:  The problem has been solved!
```

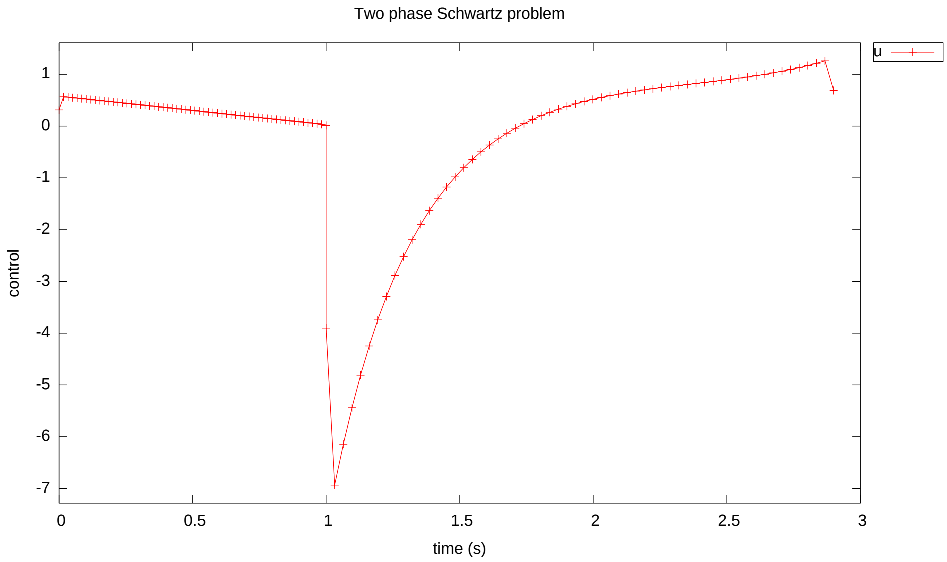

Figure 3.111: Control for two-phase Schwartz problem

3.43 Vehicle launch problem

This problem consists of the launch of a space vehicle. See [31, 2] for a full description of the problem. Only a brief description is given here. The flight of the vehicle can be divided into four phases, with dry masses ejected from the vehicle at the end of phases 1, 2 and 3. The final times of phases 1, 2 and 3 are fixed, while the final time of phase 4 is free. The optimal control problem is to find the control, $\mathbf{u}$, that minimizes the cost function

$$J = -m^{(4)}(t_f) \tag{3.177}$$

In other words, it is desired to maximise the vehicle mass at the end of phase 4. The dynamics are given by:

$$\begin{aligned} \dot{\mathbf{r}} &= \mathbf{v} \\ \dot{\mathbf{v}} &= -\frac{\mu}{\|\mathbf{r}\|^3}\mathbf{r} + \frac{T}{m}\mathbf{u} + \frac{\mathbf{D}}{m} \\ \dot{m} &= -\frac{T}{g_0 I_{sp}} \end{aligned} \tag{3.178}$$

where $\mathbf{r}(t) = \begin{bmatrix} x(t) & y(t) & z(t) \end{bmatrix}^T$ is the position, $\mathbf{v} = \begin{bmatrix} v_x(t) & v_y(t) & v_z(t) \end{bmatrix}^T$ is the Cartesian ECI velocity, μ is the gravitational parameter, T is the vacuum thrust, m is the mass, g_0 is the acceleration due to gravity at sea level, I_{sp} is the specific impulse of the engine, $\mathbf{u} = \begin{bmatrix} u_x & u_y & u_z \end{bmatrix}^T$ is the thrust direction, and $\mathbf{D} = \begin{bmatrix} D_x & D_y & D_z \end{bmatrix}^T$ is the drag force, which is given by:

$$\mathbf{D} = -\frac{1}{2} C_D A_{ref} \rho \|\mathbf{v}_{rel}\| \mathbf{v}_{rel} \tag{3.179}$$

where C_D is the drag coefficient, A_{ref} is the reference area, ρ is the atmospheric density, and $\mathbf{v}_{rel}$ is the Earth relative velocity, where $\mathbf{v}_{rel}$ is given as

$$\mathbf{v}_{rel} = \mathbf{v} - \boldsymbol{\omega} \times \mathbf{r} \tag{3.180}$$

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

$$\rho = \rho_0 \exp[-h/h_0] \tag{3.181}$$

where ρ_0 is the atmospheric density at sea level, $h = \|\mathbf{r}\| - R_e$ is the altitude, R_e is the equatorial radius of the Earth, and h_0 is the density scale height. The numerical values for these constants can be found in the code.

The vehicle starts on the ground at rest (relative to the Earth) at time t_0, so that the initial conditions are

$$\begin{aligned} \mathbf{r}(t_0) &= \mathbf{r}_0 = \left[\begin{array}{ccc} 5605.2 & 0 & 3043.4 \end{array}\right]^T \quad \text{km} \\ \mathbf{v}(t_0) &= \mathbf{v}_0 = \left[\begin{array}{ccc} 0 & 0.4076 & 0 \end{array}\right]^T \quad \text{km/s} \\ m(t_0) &= m_0 = 301454 \quad \text{kg} \end{aligned} \tag{3.182}$$

The terminal constraints define the target transfer orbit, which is defined in orbital elements as

$$\begin{aligned} a_f &= 24361.14 \text{ km}, \\ e_f &= 0.7308, \\ i_f &= 28.5 \text{ deg}, \\ \Omega_f &= 269.8 \text{ deg}, \\ \omega_f &= 130.5 \text{ deg} \end{aligned} \tag{3.183}$$

There is also a path constraint associated with this problem:

$$||\mathbf{u}||^2 = 1 \tag{3.184}$$

The following linkage constraints force the position and velocity to be continuous and also account for discontinuity in the mass state due to the ejections at the end of phases 1, 2 and 3:

$$\begin{aligned} \mathbf{r}^{(p)}(t_f) - \mathbf{r}^{(p+1)}(t_0) &= \mathbf{0}, \\ \mathbf{v}^{(p)}(t_f) - \mathbf{v}^{(p+1)}(t_0) &= \mathbf{0}, \qquad (p = 1, \ldots, 3) \\ m^{(p)}(t_f) - m_{dry}^{(p)} - m^{(p+1)}(t_0) &= 0 \end{aligned} \tag{3.185}$$

where the superscript (p) represents the phase number.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
/////////////////           launch.cxx           //////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////          PSOPT  Example            ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
```

```
//////// Title:         Multiphase vehicle launch         ////////////////
//////// Last modified: 05 January 2009                   ////////////////
//////// Reference:     GPOPS Manual                      ////////////////
//////// (See PSOPT handbook for full reference)          ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"


//////////////////////////////////////////////////////////////////////////
//////////////////  Declare auxiliary functions  ////////////////////////
//////////////////////////////////////////////////////////////////////////

void oe2rv(MatrixXd& oe, double mu, MatrixXd* ri, MatrixXd* vi);

void rv2oe(adouble* rv, adouble* vv, double mu, adouble* oe);


//////////////////////////////////////////////////////////////////////////
/////////  Declare an auxiliary structure to hold local constants  ///////
//////////////////////////////////////////////////////////////////////////

struct Constants {
  MatrixXd* omega_matrix;
  double mu;
  double cd;
  double sa;
  double rho0;
  double H;
  double Re;
  double g0;
  double thrust_srb;
  double thrust_first;
  double thrust_second;
  double ISP_srb;
  double ISP_first;
  double ISP_second;
};

typedef struct Constants Constants_;


//////////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    adouble retval;
    adouble mass_tf = final_states[6];

    if (iphase < 4)
      retval = 0.0;

    if (iphase== 4)
      retval = -mass_tf;

    return retval;
}

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

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

{
    return  0.0;
}



//////////////////////////////////////////////////////////////////////////
```

```
//////////////////  Define the DAE's ////////////////////////////////////
////////////////////////////////////////////////////////////////////////

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

    int j;

    Constants_& CONSTANTS = *( (Constants_ *) workspace->problem->user_data );

    adouble* x = states;
    adouble* u = controls;

    adouble  r[3]; r[0]=x[0]; r[1]=x[1]; r[2]=x[2];

    adouble  v[3]; v[0]=x[3]; v[1]=x[4]; v[2]=x[5];

    adouble  m = x[6];

    double T_first, T_second, T_srb, T_tot, m1dot, m2dot, mdot;

    adouble rad = sqrt( dot( r, r, 3)  );

    MatrixXd& omega_matrix = *CONSTANTS.omega_matrix;

    adouble vrel[3];
    for (j=0;j<3;j++)
      vrel[j] =  v[j] - omega_matrix(j,0)*r[0] -omega_matrix(j,1)*r[1] - omega_matrix(j,2)*r[2];


    adouble speedrel = sqrt( dot(vrel,vrel,3) );
    adouble altitude = rad-CONSTANTS.Re;

    adouble rho = CONSTANTS.rho0*exp(-altitude/CONSTANTS.H);
    double   a1 = CONSTANTS.rho0*CONSTANTS.sa*CONSTANTS.cd;
    adouble  a2 = a1*exp(-altitude/CONSTANTS.H);
    adouble  bc = (rho/(2*m))*CONSTANTS.sa*CONSTANTS.cd;



    adouble bcspeed    = bc*speedrel;

    adouble Drag[3];
    for(j=0;j<3;j++) Drag[j] =  - (vrel[j]*bcspeed);



    adouble muoverradcubed = (CONSTANTS.mu)/(pow(rad,3));
    adouble grav[3];
    for(j=0;j<3;j++) grav[j] = -muoverradcubed*r[j];


   if (iphase==1) {
     T_srb = 6*CONSTANTS.thrust_srb;
     T_first = CONSTANTS.thrust_first;
     T_tot = T_srb+T_first;
     m1dot = -T_srb/(CONSTANTS.g0*CONSTANTS.ISP_srb);
     m2dot = -T_first/(CONSTANTS.g0*CONSTANTS.ISP_first);
     mdot = m1dot+m2dot;
   }
   else if (iphase==2) {
     T_srb = 3*CONSTANTS.thrust_srb;
     T_first = CONSTANTS.thrust_first;
     T_tot = T_srb+T_first;
     m1dot = -T_srb/(CONSTANTS.g0*CONSTANTS.ISP_srb);
     m2dot = -T_first/(CONSTANTS.g0*CONSTANTS.ISP_first);
     mdot = m1dot+m2dot;
   }
   else if (iphase==3) {
     T_first = CONSTANTS.thrust_first;
     T_tot = T_first;
     mdot = -T_first/(CONSTANTS.g0*CONSTANTS.ISP_first);
   }
   else if (iphase==4) {
     T_second = CONSTANTS.thrust_second;
     T_tot = T_second;
     mdot = -T_second/(CONSTANTS.g0*CONSTANTS.ISP_second);
   }
```

```
   adouble Toverm = T_tot/m;

   adouble thrust[3];

   for(j=0;j<3;j++) thrust[j] = Toverm*u[j];

   adouble rdot[3];
   for(j=0;j<3;j++) rdot[j] = v[j];

   adouble vdot[3];
   for(j=0;j<3;j++)  vdot[j] = thrust[j]+Drag[j]+grav[j];


   derivatives[0] = rdot[0];
   derivatives[1] = rdot[1];
   derivatives[2] = rdot[2];
   derivatives[3] = vdot[0];
   derivatives[4] = vdot[1];
   derivatives[5] = vdot[2];
   derivatives[6] = mdot;

   path[0] = dot( controls, controls, 3);

}

////////////////////////////////////////////////////////////////////////////
/////////////////// Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

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


   Constants_& CONSTANTS = *( (Constants_ *) workspace->problem->user_data );

   adouble rv[3]; rv[0]=final_states[0]; rv[1]=final_states[1]; rv[2]=final_states[2];
   adouble vv[3]; vv[0]=final_states[3]; vv[1]=final_states[4]; vv[2]=final_states[5];

   adouble oe[6];

   int j;

   if(iphase==1) {
       // These events are related to the initial state conditions in phase 1
       for(j=0;j<7;j++) e[j] = initial_states[j];
   }

   if (iphase==4) {
     // These events are related to the final states in phase 4
     rv2oe( rv, vv, CONSTANTS.mu, oe );
     for(j=0;j<5;j++) e[j]=oe[j];

   }



}


///////////////////////////////////////////////////////////////////////////
/////////////////// Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

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

    double m_tot_first   = 104380.0;
    double m_prop_first  = 95550.0;
    double m_dry_first   = m_tot_first-m_prop_first;
    double m_tot_srb     = 19290.0;
    double m_prop_srb    = 17010.0;
    double m_dry_srb     = m_tot_srb-m_prop_srb;

    int index=0;

    auto_link(linkages, &index, xad, 1, 2, workspace );
    linkages[index-2]-= 6*m_dry_srb;
    auto_link(linkages, &index, xad, 2, 3, workspace );
    linkages[index-2]-= 3*m_dry_srb;
    auto_link(linkages, &index, xad, 3, 4, workspace );
```

```
    linkages[index-2]-= m_dry_first;

}



////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{
////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name         = "Multiphase vehicle launch";
    problem.outfilename                 = "launch.txt";

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare an instance of Constants structure /////////////
////////////////////////////////////////////////////////////////////////////


    Constants_ CONSTANTS;

    problem.user_data = (void*) &CONSTANTS;



////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases    = 4;
    problem.nlinkages                   = 24;

    psopt_level1_setup(problem);

/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates   = 7;
    problem.phases(1).ncontrols = 3;
    problem.phases(1).nevents   = 7;
    problem.phases(1).npath     = 1;

    problem.phases(2).nstates   = 7;
    problem.phases(2).ncontrols = 3;
    problem.phases(2).nevents   = 0;
    problem.phases(2).npath     = 1;


    problem.phases(3).nstates   = 7;
    problem.phases(3).ncontrols = 3;
    problem.phases(3).nevents   = 0;
    problem.phases(3).npath     = 1;

    problem.phases(4).nstates   = 7;
    problem.phases(4).ncontrols = 3;
    problem.phases(4).nevents   = 5;
    problem.phases(4).npath     = 1;

    problem.phases(1).nodes     = (RowVectorXi(2) << 15, 18).finished();
    problem.phases(2).nodes     = (RowVectorXi(2) << 15, 18).finished();
    problem.phases(3).nodes     = (RowVectorXi(2) << 15, 18).finished();
    problem.phases(4).nodes     = (RowVectorXi(2) << 20, 25).finished();





    psopt_level2_setup(problem, algorithm);
```

```
////////////////////////////////////////////////////////////////////////////
///////////////////  Declare MatrixXd objects to store results /////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd x, u, t, H;

////////////////////////////////////////////////////////////////////////////
///////////////////  Initialize CONSTANTS and //////////////////////////////
///////////////////  declare local variables  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    double omega            = 7.29211585e-5; // Earth rotation rate (rad/s)
    MatrixXd omega_matrix(3,3);

    omega_matrix(0,0) = 0.0;   omega_matrix(0,1) = -omega;   omega_matrix(0,2)=0.0;
    omega_matrix(1,0) = omega; omega_matrix(1,1) = 0.0;      omega_matrix(1,2)=0.0;
    omega_matrix(2,0) = 0.0;   omega_matrix(2,1) = 0.0;      omega_matrix(2,2)=0.0;

    CONSTANTS.omega_matrix = &omega_matrix; // Rotation rate matrix (rad/s)
    CONSTANTS.mu = 3.986012e14;       // Gravitational parameter (m^3/s^2)
    CONSTANTS.cd = 0.5;               // Drag coefficient
    CONSTANTS.sa = 4*pi;              // Surface area (m^2)
    CONSTANTS.rho0 = 1.225;           // sea level gravity (kg/m^3)
    CONSTANTS.H = 7200.0;             // Density scale height (m)
    CONSTANTS.Re = 6378145.0;         // Radius of earth (m)
    CONSTANTS.g0 = 9.80665;           // sea level gravity (m/s^2)

    double lat0 = 28.5*pi/180.0;              // Geocentric Latitude of Cape Canaveral
    double x0 = CONSTANTS.Re*cos(lat0);      // x component of initial position
    double z0 = CONSTANTS.Re*sin(lat0);      // z component of initial position
    double y0 = 0;
    MatrixXd r0(3,1); r0 <<  x0, y0, z0;
    MatrixXd v0 = omega_matrix*r0;

    double bt_srb = 75.2;
    double bt_first = 261.0;
    double bt_second = 700.0;

    double t0 = 0;
    double t1 = 75.2;
    double t2 = 150.4;
    double t3 = 261.0;
    double t4 = 961.0;

    double m_tot_srb     = 19290.0;
    double m_prop_srb    = 17010.0;
    double m_dry_srb     = m_tot_srb-m_prop_srb;
    double m_tot_first   = 104380.0;
    double m_prop_first  = 95550.0;
    double m_dry_first   = m_tot_first-m_prop_first;
    double m_tot_second  = 19300.0;
    double m_prop_second = 16820.0;
    double m_dry_second  = m_tot_second-m_prop_second;
    double m_payload     = 4164.0;
    double thrust_srb    = 628500.0;
    double thrust_first  = 1083100.0;
    double thrust_second = 110094.0;
    double mdot_srb      = m_prop_srb/bt_srb;
    double ISP_srb       = thrust_srb/(CONSTANTS.g0*mdot_srb);
    double mdot_first    = m_prop_first/bt_first;
    double ISP_first     = thrust_first/(CONSTANTS.g0*mdot_first);
    double mdot_second   = m_prop_second/bt_second;
    double ISP_second     = thrust_second/(CONSTANTS.g0*mdot_second);

    double af = 24361140.0;
    double ef = 0.7308;
    double incf = 28.5*pi/180.0;
    double Omf = 269.8*pi/180.0;
    double omf = 130.5*pi/180.0;
    double nuguess = 0;
    double cosincf = cos(incf);
    double cosOmf = cos(Omf);
    double cosomf = cos(omf);
    MatrixXd oe(6,1); oe << af, ef, incf, Omf, omf, nuguess;

    MatrixXd rout(3,1);
    MatrixXd vout(3,1);

    oe2rv(oe,CONSTANTS.mu, &rout, &vout);

    rout= rout.transpose().eval();
```

```
    vout= vout.transpose().eval();

    double m10 = m_payload+m_tot_second+m_tot_first+9*m_tot_srb;
    double m1f = m10-(6*mdot_srb+mdot_first)*t1;
    double m20 = m1f-6*m_dry_srb;
    double m2f = m20-(3*mdot_srb+mdot_first)*(t2-t1);
    double m30 = m2f-3*m_dry_srb;
    double m3f = m30-mdot_first*(t3-t2);
    double m40 = m3f-m_dry_first;
    double m4f = m_payload;

    CONSTANTS.thrust_srb    = thrust_srb;
    CONSTANTS.thrust_first  = thrust_first;
    CONSTANTS.thrust_second = thrust_second;
    CONSTANTS.ISP_srb       = ISP_srb;
    CONSTANTS.ISP_first     = ISP_first;
    CONSTANTS.ISP_second    = ISP_second;


    double rmin = -2*CONSTANTS.Re;
    double rmax = -rmin;
    double vmin = -10000.0;
    double vmax = -vmin;

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////


    int iphase;


    // Phase 1 bounds

    iphase =  1;


    problem.phases(iphase).bounds.lower.states   << rmin, rmin, rmin, vmin, vmin, vmin, m1f;
    problem.phases(iphase).bounds.upper.states   << rmax, rmax, rmax, vmax, vmax, vmax, m10;

    problem.phases(iphase).bounds.lower.controls << -1.0, -1.0, -1.0;
    problem.phases(iphase).bounds.upper.controls <<  1.0,  1.0,  1.0;

    problem.phases(iphase).bounds.lower.path     <<  1.0;
    problem.phases(iphase).bounds.upper.path     <<  1.0;


    // The following bounds fix the initial state conditions in phase 0.

    problem.phases(iphase).bounds.lower.events << r0(0), r0(1), r0(2), v0(0), v0(1), v0(2), m10;
    problem.phases(iphase).bounds.upper.events << r0(0), r0(1), r0(2), v0(0), v0(1), v0(2), m10;

    problem.phases(iphase).bounds.lower.StartTime    = 0.0;
    problem.phases(iphase).bounds.upper.StartTime    = 0.0;

    problem.phases(iphase).bounds.lower.EndTime    = 75.2;
    problem.phases(iphase).bounds.upper.EndTime    = 75.2;


    // Phase 2 bounds

    iphase =  2;

    problem.phases(iphase).bounds.lower.states   << rmin, rmin, rmin, vmin, vmin, vmin, m2f;
    problem.phases(iphase).bounds.upper.states   << rmax, rmax, rmax, vmax, vmax, vmax, m20;


    problem.phases(iphase).bounds.lower.controls << -1.0, -1.0, -1.0;
    problem.phases(iphase).bounds.upper.controls <<  1.0,  1.0,  1.0;

    problem.phases(iphase).bounds.lower.path     <<  1.0;
    problem.phases(iphase).bounds.upper.path     <<  1.0;

    problem.phases(iphase).bounds.lower.StartTime    = 75.2;
    problem.phases(iphase).bounds.upper.StartTime    = 75.2;

    problem.phases(iphase).bounds.lower.EndTime    = 150.4;
    problem.phases(iphase).bounds.upper.EndTime    = 150.4;

    // Phase 3 bounds

    iphase =  3;
```

```
    problem.phases(iphase).bounds.lower.states   << rmin, rmin, rmin, vmin, vmin, vmin, m3f;
    problem.phases(iphase).bounds.upper.states   << rmax, rmax, rmax, vmax, vmax, vmax, m30;

    problem.phases(iphase).bounds.lower.controls << -1.0, -1.0, -1.0;
    problem.phases(iphase).bounds.upper.controls <<  1.0,  1.0,  1.0;

    problem.phases(iphase).bounds.lower.path     <<  1.0;
    problem.phases(iphase).bounds.upper.path     <<  1.0;

    problem.phases(iphase).bounds.lower.StartTime  = 150.4;
    problem.phases(iphase).bounds.upper.StartTime  = 150.4;

    problem.phases(iphase).bounds.lower.EndTime    = 261.0;
    problem.phases(iphase).bounds.upper.EndTime    = 261.0;


    // Phase 4 bounds

    iphase =  4;


    problem.phases(iphase).bounds.lower.states   << rmin, rmin, rmin, vmin, vmin, vmin, m4f;
    problem.phases(iphase).bounds.upper.states   << rmax, rmax, rmax, vmax, vmax, vmax, m40;


    problem.phases(iphase).bounds.lower.controls << -1.0, -1.0, -1.0;
    problem.phases(iphase).bounds.upper.controls <<  1.0,  1.0,  1.0;

    problem.phases(iphase).bounds.lower.path     <<  1.0;
    problem.phases(iphase).bounds.upper.path     <<  1.0;

    problem.phases(iphase).bounds.lower.events   << af, ef, incf, Omf, omf;
    problem.phases(iphase).bounds.upper.events   << af, ef, incf, Omf, omf;

    problem.phases(iphase).bounds.lower.StartTime    = 261.0;
    problem.phases(iphase).bounds.upper.StartTime    = 261.0;

    problem.phases(iphase).bounds.lower.EndTime    = 261.0;
    problem.phases(iphase).bounds.upper.EndTime    = 961.0;


////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    iphase = 1;

    problem.phases(iphase).guess.states = zeros(7,5);

    problem.phases(iphase).guess.states.row(0) = linspace( r0(0), r0(0), 5);
    problem.phases(iphase).guess.states.row(1) = linspace( r0(1), r0(1), 5);
    problem.phases(iphase).guess.states.row(2) = linspace( r0(2), r0(2), 5);
    problem.phases(iphase).guess.states.row(3) = linspace( v0(0), v0(0), 5);
    problem.phases(iphase).guess.states.row(4) = linspace( v0(1), v0(1), 5);
    problem.phases(iphase).guess.states.row(5) = linspace( v0(2), v0(2), 5);
    problem.phases(iphase).guess.states.row(6) = linspace( m10  , m1f  , 5);

    problem.phases(iphase).guess.controls = zeros(3,5);

    problem.phases(iphase).guess.controls.row(0) = ones( 1, 5);
    problem.phases(iphase).guess.controls.row(1) = zeros(1, 5);
    problem.phases(iphase).guess.controls.row(2) = zeros(1, 5);


    problem.phases(iphase).guess.time = linspace(t0,t1, 5);


    iphase = 2;

    problem.phases(iphase).guess.states = zeros(7,5);

    problem.phases(iphase).guess.states.row(0) = linspace( r0(0), r0(0), 5);
    problem.phases(iphase).guess.states.row(1) = linspace( r0(1), r0(1), 5);
    problem.phases(iphase).guess.states.row(2) = linspace( r0(2), r0(2), 5);
    problem.phases(iphase).guess.states.row(3) = linspace( v0(0), v0(0), 5);
    problem.phases(iphase).guess.states.row(4) = linspace( v0(1), v0(1), 5);
    problem.phases(iphase).guess.states.row(5) = linspace( v0(2), v0(2), 5);
    problem.phases(iphase).guess.states.row(6) = linspace( m20  , m2f  , 5);

    problem.phases(iphase).guess.controls = zeros(3,5);
```

```
    problem.phases(iphase).guess.controls.row(0) = ones( 1, 5);
    problem.phases(iphase).guess.controls.row(1) = zeros(1, 5);
    problem.phases(iphase).guess.controls.row(2) = zeros(1, 5);


    problem.phases(iphase).guess.time = linspace(t1,t2, 5);

    iphase = 3;

    problem.phases(iphase).guess.states = zeros(7,5);

    problem.phases(iphase).guess.states.row(0) = linspace( r0(0), r0(0), 5);
    problem.phases(iphase).guess.states.row(1) = linspace( r0(1), r0(1), 5);
    problem.phases(iphase).guess.states.row(2) = linspace( r0(2), r0(2), 5);
    problem.phases(iphase).guess.states.row(3) = linspace( v0(0), v0(0), 5);
    problem.phases(iphase).guess.states.row(4) = linspace( v0(1), v0(1), 5);
    problem.phases(iphase).guess.states.row(5) = linspace( v0(2), v0(2), 5);
    problem.phases(iphase).guess.states.row(6) = linspace( m30  , m3f  , 5);

    problem.phases(iphase).guess.controls = zeros(3,5);

    problem.phases(iphase).guess.controls.row(0) = ones( 1, 5);
    problem.phases(iphase).guess.controls.row(1) = zeros(1, 5);
    problem.phases(iphase).guess.controls.row(2) = zeros(1, 5);


    problem.phases(iphase).guess.time = linspace(t2,t3, 5);

    iphase = 4;

    problem.phases(iphase).guess.states = zeros(7,5);

    problem.phases(iphase).guess.states.row(0) = linspace( rout(0), rout(0), 5);
    problem.phases(iphase).guess.states.row(1) = linspace( rout(1), rout(1), 5);
    problem.phases(iphase).guess.states.row(2) = linspace( rout(2), rout(2), 5);
    problem.phases(iphase).guess.states.row(3) = linspace( vout(0), vout(0), 5);
    problem.phases(iphase).guess.states.row(4) = linspace( vout(1), vout(1), 5);
    problem.phases(iphase).guess.states.row(5) = linspace( vout(2), vout(2), 5);
    problem.phases(iphase).guess.states.row(6) = linspace( m40  , m4f  , 5);

    problem.phases(iphase).guess.controls = zeros(3,5);

    problem.phases(iphase).guess.controls.row(0) = ones( 1, 5);
    problem.phases(iphase).guess.controls.row(1) = zeros(1, 5);
    problem.phases(iphase).guess.controls.row(2) = zeros(1, 5);

    problem.phases(iphase).guess.time = linspace(t3,t4, 5);


////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.nlp_iter_max                = 1000;
    algorithm.collocation_method            = "Chebyshev";
//    algorithm.mesh_refinement             = "automatic";
//    algorithm.ode_tolerance           = 1.e-5;

////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
//////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////
```

```
    MatrixXd x_ph1, x_ph2, x_ph3, x_ph4, u_ph1, u_ph2, u_ph3, u_ph4;
    MatrixXd t_ph1, t_ph2, t_ph3, t_ph4;

    x_ph1 = solution.get_states_in_phase(1);
    x_ph2 = solution.get_states_in_phase(2);
    x_ph3 = solution.get_states_in_phase(3);
    x_ph4 = solution.get_states_in_phase(4);

    u_ph1 = solution.get_controls_in_phase(1);
    u_ph2 = solution.get_controls_in_phase(2);
    u_ph3 = solution.get_controls_in_phase(3);
    u_ph4 = solution.get_controls_in_phase(4);

    t_ph1 = solution.get_time_in_phase(1);
    t_ph2 = solution.get_time_in_phase(2);
    t_ph3 = solution.get_time_in_phase(3);
    t_ph4 = solution.get_time_in_phase(4);



    x.resize(7, x_ph1.cols()+ x_ph2.cols()+ x_ph3.cols()+ x_ph4.cols() );
    u.resize(3, u_ph1.cols()+ u_ph2.cols()+ u_ph3.cols()+ u_ph4.cols() );
    t.resize(1, t_ph1.cols()+ t_ph2.cols()+ t_ph3.cols()+ t_ph4.cols() );

    x << x_ph1, x_ph2, x_ph3, x_ph4;
    u << u_ph1, u_ph2, u_ph3, u_ph4;
    t << t_ph1, t_ph2, t_ph3, t_ph4;


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

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

////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd r, v, altitude, speed;

    r = x.block(0,0,3,x.cols());

    v = x.block(3,0,3,x.cols());

    altitude = (sum_columns(elemProduct(r,r)).cwiseSqrt())/1000.0;

    speed = sum_columns(elemProduct(v,v)).cwiseSqrt();

    plot(t,altitude,problem.name, "time (s)", "Altitude (km)");


    plot(t,speed,problem.name, "time (s)", "speed (m/s)");

    plot(t,u,problem.name,"time (s)", "u");


    plot(t,altitude,problem.name, "time (s)", "Altitude (km)", "alt",
                                  "pdf", "launch_altitude.pdf");

    plot(t,speed,problem.name, "time (s)", "speed (m/s)", "speed",
                               "pdf", "launch_speed.pdf");

    plot(t,u,problem.name,"time (s)", "u (dimensionless)", "u1 u2 u3",
                               "pdf", "launch_control.pdf");



}

////////////////////////////////////////////////////////////////////////////
////////////////// Define auxiliary functions //////////////////////////////
////////////////////////////////////////////////////////////////////////////


void rv2oe(adouble* rv, adouble* vv, double mu, adouble* oe)
{
       int j;

       adouble K[3]; K[0] = 0.0; K[1]=0.0; K[2]=1.0;
```

```
       adouble hv[3];
       cross(rv,vv, hv);

       adouble nv[3];
       cross(K, hv, nv);

       adouble n  = sqrt(  dot(nv,nv,3) );

       adouble h2 = dot(hv,hv,3);

       adouble v2 = dot(vv,vv,3);

       adouble r         = sqrt(dot(rv,rv,3));

       adouble ev[3];
       for(j=0;j<3;j++)  ev[j] = 1/mu *( (v2-mu/r)*rv[j] - dot(rv,vv,3)*vv[j] );

       adouble p         = h2/mu;

adouble e  = sqrt(dot(ev,ev,3)); // eccentricity
adouble a  = p/(1-e*e);    // semimajor axis
adouble i  = acos(hv[2]/sqrt(h2)); // inclination



#define USE_SMOOTH_HEAVISIDE
        double a_eps = 0.1;

#ifndef USE_SMOOTH_HEAVISIDE
  adouble Om = acos(nv[0]/n); // RAAN
  if ( nv[1] < -PSOPT_extras::GetEPS() ){ // fix quadrant
  Om = 2*pi-Om;
  }
#endif

#ifdef USE_SMOOTH_HEAVISIDE

        adouble Om =  smooth_heaviside( (nv[1]+PSOPT_extras::GetEPS()), a_eps )*acos(nv[0]/n)
                     +smooth_heaviside( -(nv[1]+PSOPT_extras::GetEPS()), a_eps )*(2*pi-acos(nv[0]/n));
#endif

#ifndef USE_SMOOTH_HEAVISIDE
  adouble om = acos(dot(nv,ev,3)/n/e); // arg of periapsis
  if ( ev[2] < 0 ) { // fix quadrant
  om = 2*pi-om;
  }
#endif

#ifdef USE_SMOOTH_HEAVISIDE
        adouble om =  smooth_heaviside( (ev[2]), a_eps )*acos(dot(nv,ev,3)/n/e)
                     +smooth_heaviside( -(ev[2]), a_eps )*(2*pi-acos(dot(nv,ev,3)/n/e));
#endif

#ifndef USE_SMOOTH_HEAVISIDE
  adouble nu = acos(dot(ev,rv,3)/e/r); // true anomaly
  if ( dot(rv,vv,3) < 0 ) { // fix quadrant
  nu = 2*pi-nu;
  }
#endif

#ifdef USE_SMOOTH_HEAVISIDE
       adouble nu =  smooth_heaviside( dot(rv,vv,3), a_eps )*acos(dot(ev,rv,3)/e/r)
                     +smooth_heaviside( -dot(rv,vv,3), a_eps )*(2*pi-acos(dot(ev,rv,3)/e/r));
#endif

        oe[0] = a;
        oe[1] = e;
        oe[2] = i;
        oe[3] = Om;
        oe[4] = om;
        oe[5] = nu;

        return;

}


void oe2rv(MatrixXd& oe, double mu, MatrixXd* ri, MatrixXd* vi)
{
double a=oe(0), e=oe(1), i=oe(2), Om=oe(3), om=oe(4), nu=oe(5);
double p = a*(1-e*e);
double r = p/(1+e*cos(nu));
```

```
MatrixXd rv(3,1);
        rv(0) = r*cos(nu);
        rv(1) = r*sin(nu);
        rv(2) = 0.0;

MatrixXd vv(3,1);

        vv(0) = -sin(nu);
        vv(1) = e+cos(nu);
        vv(2) = 0.0;
        vv    *= sqrt(mu/p);

double cO = cos(Om),  sO = sin(Om);
double co = cos(om),  so = sin(om);
double ci = cos(i),   si = sin(i);

MatrixXd R(3,3);
   R(0,0)=  cO*co-sO*so*ci;  R(0,1)=  -cO*so-sO*co*ci; R(0,2)=  sO*si;
R(1,0)=  sO*co+cO*so*ci; R(1,1)=  -sO*so+cO*co*ci; R(1,2)=-cO*si;
R(2,0)=   so*si;         R(2,1)=    co*si;         R(2,2)=  ci;

*ri = R*rv;
*vi = R*vv;

        return;
}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.112, 3.113 and 3.114, which contain the trajectories of the altitude, speed and the elements of the control vector, respectively.

```
PSOPT results summary
=====================

Problem:  Multiphase vehicle launch
CPU time (seconds): 3.801232e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:22:22 2020

Optimal (unscaled) cost function value:  -7.529661e+03
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 7.520000e+01
Phase 1 maximum relative local error: 5.841054e-07
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  0.000000e+00
Phase 2 initial time: 7.520000e+01
Phase 2 final time: 1.504000e+02
Phase 2 maximum relative local error: 3.326159e-07
```

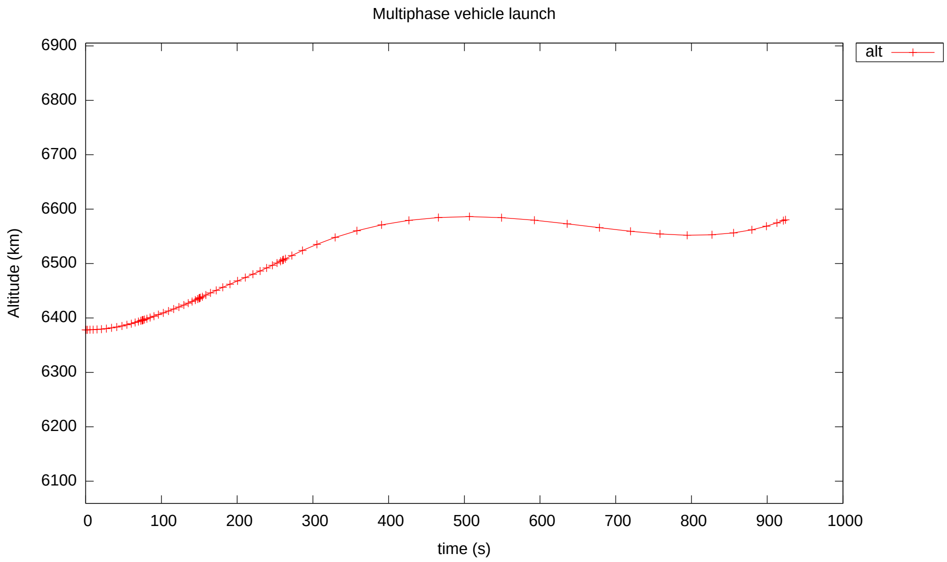

Figure 3.112: Altitude for the vehicle launch problem

```
Phase 3 endpoint cost function value: 0.000000e+00
Phase 3 integrated part of the cost:  0.000000e+00
Phase 3 initial time: 1.504000e+02
Phase 3 final time: 2.610000e+02
Phase 3 maximum relative local error: 3.131853e-07
Phase 4 endpoint cost function value: -7.529661e+03
Phase 4 integrated part of the cost:  0.000000e+00
Phase 4 initial time: 2.610000e+02
Phase 4 final time: 9.241413e+02
Phase 4 maximum relative local error: 1.479027e-06
NLP solver reports:  The problem has been solved!
```

3.44 Zero propellant maneouvre of the International Space Station

This problem illustrates the use of $\mathcal{PSOPT}$ for solving an optimal control problem associated with the design of a zero propellant maneouvre for the international space station by means of control moment gyroscopes (CMGs). The example is based on the results presented in the thesis by Bhatt [5] and also reported by Bedrossian and co-workers [1]. The original 90 and 180 degree maneouvres were computed using DIDO, and they were actually implemented on the International Space Station on 5 November 2006 and 2

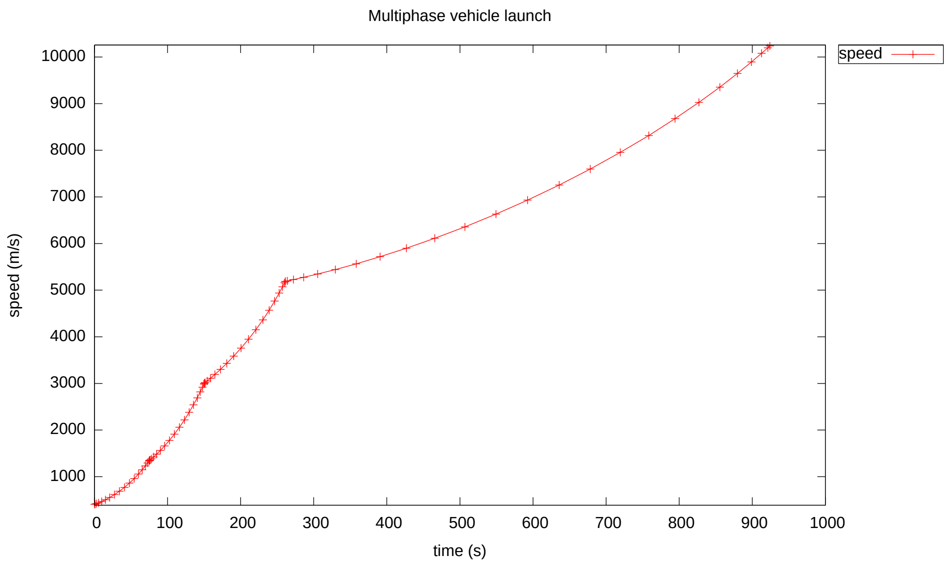

Figure 3.113: Speed for the vehicle launch problem

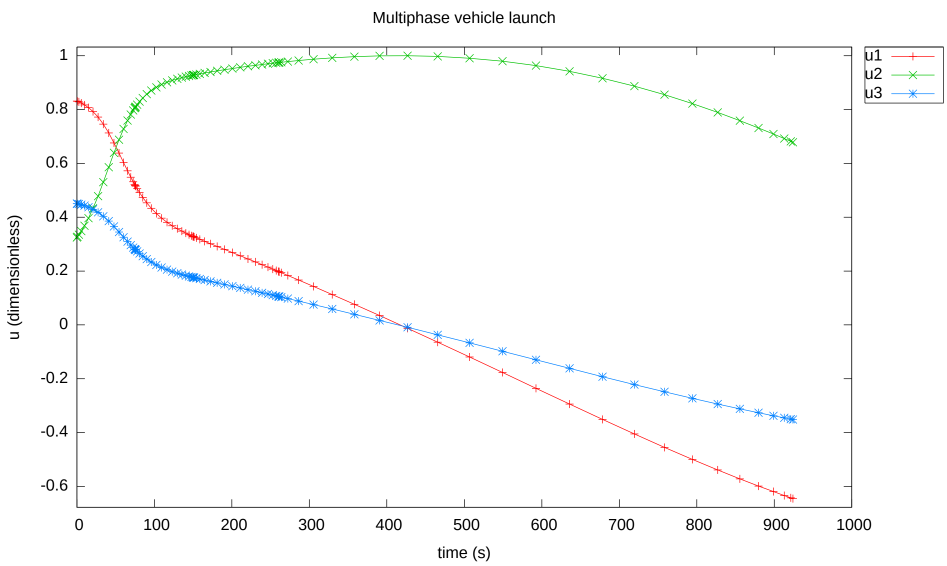

Figure 3.114: Controls for the vehicle launch problem

January 2007, respectively, resulting in savings for NASA of around US$1.5m in propellant costs. The dynamic model employed here does not account for atmospheric drag as the atmosphere model used in the original study is not available. Otherwise, the equations and parameters are the same as those reported by Bhatt in his thesis. The effects of atmospheric drag are, however, small, and the results obtained are comparable with those given in Bhatt's thesis. The implemented case corresponds with a maneovure lasting 7200 seconds and using 3 CMG's.

The problem is formulated as follows. Find $\mathbf{q_c}(t) = [q_{c,1}(t)\, q_{c,2}(t)\, q_{c,3}(t)\, q_{c,4}]^T$, $t \in [t_0, t_f]$ and the scalar parameter γ to minimise,

$$J = 0.1\gamma + \int_{t_0}^{t_f} ||u(t)||^2 \mathrm{d}t \tag{3.186}$$

subject to the dynamical equations:

$$\begin{aligned}
\dot{\mathbf{q}}(t) &= \frac{1}{2}\mathbf{T}(\mathbf{q})(\omega(t) - \omega_o(\mathbf{q})) \\
\dot{\omega}(t) &= \mathbf{J}^{-1}\left(\tau_d(\mathbf{q}) - \omega(t) \times (\mathbf{J}\omega(t)) - \mathbf{u}(t)\right) \\
\dot{\mathbf{h}}(t) &= \mathbf{u}(t) - \omega(t) \times \mathbf{h}(t)
\end{aligned} \tag{3.187}$$

the path constraints:

$$\begin{aligned}
||\mathbf{q}(t)||_2^2 &= 1 \\
||\mathbf{q}_c(t)||_2^2 &= 1 \\
||\mathbf{h}(t)||_2^2 &\leq \gamma \\
||\dot{\mathbf{h}}(t)||_2^2 &= \dot{h}_{\max}^2
\end{aligned} \tag{3.188}$$

the parameter bounds

$$0 \leq \gamma \leq h_{\max}^2 \tag{3.189}$$

and the boundary conditions:

$$\begin{array}{lll}
\mathbf{q}(t_0) = \bar{\mathbf{q}}_0 & \omega(t_0) = \omega_o(\bar{\mathbf{q}}_0) & \mathbf{h}(t_0) = \bar{\mathbf{h}}_0 \\
\mathbf{q}(t_f) = \bar{\mathbf{q}}_\mathbf{f} & \omega(t_f) = \omega_o(\bar{\mathbf{q}}_f) & \mathbf{h}(t_f) = \bar{\mathbf{h}}_f
\end{array} \tag{3.190}$$

where $\mathbf{J}$ is a 3×3 inertia matrix, $\mathbf{q} = [q_1, q_2, q_3, q_4]^T$ is the quarternion vector, ω is the spacecraft angular rate relative to an inertial reference frame and expressed in the body frame, $\mathbf{h}$ is the momentum, $\mathbf{T}(\mathbf{q})$ is given by:

$$\mathbf{T}(\mathbf{q}) = \begin{bmatrix} -q_2 & -q_3 & -q_4 \\ q_1 & -q_4 & q_3 \\ q_4 & q_1 & -q_2 \\ -q_3 & q_2 & q_1 \end{bmatrix} \tag{3.191}$$

$\mathbf{u}$ is the control force, which is given by:

$$\mathbf{u}(t) = \mathbf{J}\left(K_P \tilde{\varepsilon}(q, q_c) + K_D \tilde{\omega}(\omega, q_c)\right) \tag{3.192}$$

where

$$\begin{aligned}\tilde{\varepsilon}(\mathbf{q},\mathbf{q}_c) &= 2\mathbf{T}(\mathbf{q}_c)^T\mathbf{q} \\ \tilde{\omega}(\omega,\omega_c) &= \omega-\omega_c\end{aligned} \tag{3.193}$$

ω_o is given by:

$$\omega_o(\mathbf{q}) = n\mathbf{C}_2(\mathbf{q}) \tag{3.194}$$

where n is the orbital rotation rate, $\mathbf{C}_j$ is the j column of the rotation matrix:

$$\mathbf{C}(\mathbf{q}) = \begin{bmatrix} 1-2(q_3^2+q_4^2) & 2(q_2q_3+q_1q_4) & 2(q_2q_4-q_1q_3) \\ 2(q_2q_3-q_1q_4) & 1-2(q_2^2+q_4^2) & 2(q_3q_4+q_1q_2) \\ 2(q_2q_4+q_1q_3) & 2(q_3q_4-q_1q_2) & 1-2(q_2^2+q_3^2) \end{bmatrix} \tag{3.195}$$

τ_d is the disturbance torque, which in this case only incorporates the gravity gradient torque τ_{gg} (the disturbance torque also incorporates the atmospheric drag torque in the original study):

$$\tau_d = \tau_{gg} = 3n^2\mathbf{C}_3(\mathbf{q}) \times (\mathbf{J}\mathbf{C}_3(\mathbf{q})) \tag{3.196}$$

The constant parameter values used were: $n = 1.1461 \times 10^{-3}$ rad/s, $h_{\max} = 3 \times 3600.0$ ft-lbf-sec, $\dot{h}_{\max} = 200.0$ ft-lbf, $t_0 = 0$ s, $t_f = 7200$ s, and

$$\mathbf{J} = \begin{bmatrix} 18836544.0 & 3666370.0 & 2965301.0 \\ 3666370.0 & 27984088.0 & -1129004.0 \\ 2965301.0 & -1129004.0 & 39442649.0 \end{bmatrix} \text{slug} - \text{ft}^2 \tag{3.197}$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
////////////////                zpm.cxx               /////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////              PSOPT Example           /////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:  Zero propellant maneouvre problem  //////////////
//////// Last modified: 09 November 2009               ////////////////
//////// Reference:     S.A. Bhatt (2007), Masters Thesis,////////////////
////////                Rice University, Houston TX       ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                     ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"



// Set CASE below to 1: 6000 s maneouvre, or to 2: 7200 maneouvre
#define CASE 2



struct Constants {
MatrixXd J;
double n;
double Kp;
double Kd;
double hmax;
};
```

```
typedef struct Constants Constants_;

static Constants_ CONSTANTS;


void Tfun( adouble T[][3], adouble *q )
{

adouble q1 = q[0];
adouble q2 = q[1];
adouble q3 = q[2];
adouble q4 = q[3];

T[0][0] = -q2 ; T[0][1] = -q3; T[0][2] = -q4;

T[1][0] =  q1 ; T[1][1] = -q4; T[1][2] =  q3;

T[2][0] =  q4 ; T[2][1] =  q1; T[2][2] = -q2;

T[3][0] = -q3;  T[3][1] =  q2; T[3][2] =  q1;

}


void compute_omega0(adouble* omega0, adouble* q)
{
// This function computes the angular speed in the rotating LVLH reference frame
int i;
double n = CONSTANTS.n;
adouble C2[3];

   adouble q1 = q[0];
   adouble q2 = q[1];
   adouble q3 = q[2];
   adouble q4 = q[3];

C2[ 0 ] = 2*(q2*q3 + q1*q4);
C2[ 1 ] = 1.0-2.0*(q2*q2+q4*q4);
C2[ 2 ] = 2*(q3*q4-q1*q2);

for (i=0;i<3;i++)   omega0[i] = -n*C2[i];

}

void compute_control_torque(adouble* u, adouble* q, adouble* qc, adouble* omega )
{
// This function computes the control torque
//

double Kp = CONSTANTS.Kp; // Proportional gain
double Kd = CONSTANTS.Kd; // Derivative gain
double n  = CONSTANTS.n;  // Orbital rotation rate [rad/s]


int i, j;

MatrixXd& J = CONSTANTS.J;


adouble T[4][3];

Tfun( T, q );

adouble Tc[4][3];

Tfun( Tc, qc );

adouble epsilon_tilde[3];

for(i=0;i<3;i++) {
epsilon_tilde[i] = 0.0;
for(j=0;j<4;j++) {
epsilon_tilde[i] += 2*Tc[j][i]*q[j];
}
}

adouble omega_c[3];

compute_omega0( omega_c, qc );

adouble omega_tilde[3];
```

```
for(i=0;i<3;i++) {
omega_tilde[i] = omega[i]-omega_c[i];
}

adouble uaux[3];

for(i=0;i<3;i++) {
uaux[i]= Kp*epsilon_tilde[i]+Kd*omega_tilde[i];
}

product_ad( J, uaux, 3, u );




}

void quarternion2Euler( MatrixXd& phi, MatrixXd& theta, MatrixXd& psi, MatrixXd& q)
{
// This function finds the Euler angles given the quarternion vector
//
long N = q.cols();
MatrixXd q0; q0 = q.row(0);
  MatrixXd q1; q1 = q.row(1);
  MatrixXd q2; q2 = q.row(2);
MatrixXd q3; q3 = q.row(3);

phi.resize(1,N);
theta.resize(1,N);
psi.resize(1,N);

    for(int i=0;i<N;i++) {
          phi(i)=atan2( 2*(q0(i)*q1(i) + q2(i)*q3(i)), 1.0-2.0*(q1(i)*q1(i)+q2(i)*q2(i)) );
            theta(i)=asin( 2*(q0(i)*q2(i)-q3(i)*q1(i)) );
            psi(i) = atan2( 2*(q0(i)*q3(i)+q1(i)*q2(i)), 1.0-2.0*(q2(i)*q2(i)+q3(i)*q3(i)) );
   }
}



void compute_aerodynamic_torque(adouble* tau_aero, adouble& time )
{
// This function approximates the aerodynamic torque by using the model and
// parameters given in the following reference:
// A. Chun Lee (2003) "Robust Momemtum Manager Controller for Space Station Applications".
// Master of Arts Thesis, Rice University.
//

double alpha1[3] = {1.0, 4.0, 1.0};
   double alpha2[3] = {1.0, 2.0, 1.0};
   double alpha3[3] = {0.5, 0.5, 0.5};
adouble t = time;

double phi1 = 0.0;
double phi2 = 0.0;

double n = CONSTANTS.n;

  for(int i=0;i<3;i++) {
                // Aerodynamic torque in [lb-ft]
tau_aero[i] = alpha1[i] + alpha2[i]*sin( n*t + phi1 ) + alpha3[i]*sin( 2*n*t + phi2);
}
}


///////////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
///////////////////////////////////////////////////////////////////////////


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

double end_point_weight = 0.1;

   adouble gamma     = parameters[ 0 ];

return (end_point_weight*gamma);

}
```

```
//////////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
double running_cost_weight = 1.0;

adouble q[4]; // quarternion vector
adouble u[3]; // control torque

q[0] = states[ 0 ];
q[1] = states[ 1 ];
q[2] = states[ 2 ];
q[3] = states[ 3 ];

adouble omega[3]; // angular rate vector

omega[0] = states[ 4 ];
omega[1] = states[ 5 ];
omega[2] = states[ 6 ];


adouble qc[4]; // control vector

qc[0] = controls[ 0 ];
qc[1] = controls[ 1 ];
qc[2] = controls[ 2 ];
qc[3] = controls[ 3 ];

compute_control_torque(u,q,qc,omega);

return running_cost_weight*dot(u,u,3);

}

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ///////////////////////////////////
//////////////////////////////////////////////////////////////////////////
void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

int i,j;


double n = CONSTANTS.n; // Orbital rotation rate [rad/s]

adouble q[4]; // quarternion vector

q[0] = states[ 0 ];
q[1] = states[ 1 ];
q[2] = states[ 2 ];
q[3] = states[ 3 ];

adouble omega[3]; // angular rate vector

omega[0] = states[ 4 ];
omega[1] = states[ 5 ];
omega[2] = states[ 6 ];

adouble h[3]; // momentum vector

h[0] = states[ 7 ];
h[1] = states[ 8 ];
h[2] = states[ 9 ];

adouble qc[4]; // control vector

qc[0] = controls[ 0 ];
qc[1] = controls[ 1 ];
qc[2] = controls[ 2 ];
qc[3] = controls[ 3 ];

adouble C2[3], C3[3];

adouble u[3];

adouble gamma;
```

```
gamma = parameters[ 0 ];


// Inertia matrix in slug-ft^2

MatrixXd& J = CONSTANTS.J;

MatrixXd Jinv; Jinv = J.inverse();

adouble q1 = q[0];
adouble q2 = q[1];
adouble q3 = q[2];
adouble q4 = q[3];

C3[ 0 ] = 2*(q2*q4 - q1*q3);
C3[ 1 ] = 2*(q3*q4 + q1*q2);
C3[ 2 ] = 1.0-2.0*(q2*q2 + q3*q3);



adouble T[4][3];

Tfun( T, q );

adouble qdot[4];

adouble omega0[3];

compute_omega0( omega0, q);

// Quarternion attitude kinematics

for(j=0;j<4;j++) {

qdot[j]=0;

for(i=0;i<3;i++) {

   qdot[j] += 0.5*T[j][i]*(omega[i]-omega0[i]);

}

}


adouble Jomega[3];

product_ad( J, omega, 3, Jomega );

adouble omegaCrossJomega[3];

cross(omega,Jomega, omegaCrossJomega);

adouble F[3];

// Compute the torque disturbances:

adouble tau_grav[3], tau_aero[3];

adouble v1[3];

for(i=0;i<3;i++) {
  v1[i] = 3*pow(n,2)*C3[i];
}

adouble JC3[3];

product_ad( J, C3, 3, JC3 );

//gravity gradient torque

cross( v1, JC3, tau_grav );

//Aerodynamic torque
compute_aerodynamic_torque(tau_aero, time );

for(i=0;i<3;i++) {
  // Uncomment this section to ignore the aerodynamic disturbance torque
  tau_aero[i] = 0.0;
}
adouble tau_d[3];
```

```
for (i=0;i<3;i++) {

   tau_d[i] =  tau_grav[i] + tau_aero[i];

}

compute_control_torque(u, q, qc, omega );


for (i=0;i<3;i++) {
   F[i]     =  tau_d[i] - omegaCrossJomega[i] - u[i];
}

adouble omega_dot[3];

// Rotational dynamics

product_ad( Jinv, F, 3, omega_dot );

adouble OmegaCrossH[3];

cross( omega, h , OmegaCrossH );

adouble hdot[3];


//Momemtum derivative

for(i=0; i<3; i++) {
   hdot[i] = u[i] - OmegaCrossH[i];
}



derivatives[0] =       qdot[ 0 ];
derivatives[1] =       qdot[ 1 ];
derivatives[2] =       qdot[ 2 ];
derivatives[3] =       qdot[ 3 ];
derivatives[4] =       omega_dot[ 0 ];
derivatives[5] =       omega_dot[ 1 ];
derivatives[6] =       omega_dot[ 2 ];
derivatives[7] =       hdot[ 0 ];
derivatives[8] =       hdot[ 1 ];
derivatives[9]=        hdot[ 2 ];


path[ 0 ] =   dot( q,  q,  4);
path[ 1 ] =   dot( qc, qc, 4);
path[ 2 ] =   dot( h, h, 3 ) - gamma;   //  <= 0
path[ 3 ] =   dot( hdot, hdot,3); // <= hdotmax^2,

}






////////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

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

{

adouble q1_i        = initial_states[0];
adouble q2_i        = initial_states[1];
adouble q3_i        = initial_states[2];
adouble q4_i        = initial_states[3];
adouble omega1_i    = initial_states[4];
adouble omega2_i    = initial_states[5];
adouble omega3_i    = initial_states[6];
adouble h1_i        = initial_states[7];
adouble h2_i        = initial_states[8];
adouble h3_i        = initial_states[9];

adouble q1_f        = final_states[0];
adouble q2_f        = final_states[1];
```

```
adouble q3_f        = final_states[2];
adouble q4_f        = final_states[3];
adouble omega1_f    = final_states[4];
adouble omega2_f    = final_states[5];
adouble omega3_f    = final_states[6];
adouble h1_f        = final_states[7];
adouble h2_f        = final_states[8];
adouble h3_f        = final_states[9];

// Initial conditions

e[ 0 ] = q1_i;
e[ 1 ] = q2_i;
e[ 2 ] = q3_i;
e[ 3 ] = q4_i;
e[ 4 ] = omega1_i;
e[ 5 ] = omega2_i;
e[ 6 ] = omega3_i;
e[ 7 ] = h1_i;
e[ 8 ] = h2_i;
e[ 9 ] = h3_i;

// Final conditions

 e[ 10 ] = q1_f;
 e[ 11 ] = q2_f;
 e[ 12 ] = q3_f;
 e[ 13 ] = q4_f;
 e[ 14 ] = omega1_f;
 e[ 15 ] = omega2_f;
 e[ 16 ] = omega3_f;
 e[ 17 ] = h1_f;
 e[ 18 ] = h2_f;
 e[ 19 ] = h3_f;


}



///////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
    // Single phase
}



////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;



   CONSTANTS.Kp = 0.000128; // Proportional gain
   CONSTANTS.Kd = 0.015846; // Derivative gain

   double hmax; // maximum momentum magnitude  in [ft-lbf-sec]


   if (CASE==1) {
CONSTANTS.n = 1.1461E-3; // Orbital rotation rate [rad/s]
hmax = 4*3600.0; // 4 CMG's
   }
   else if (CASE==2) {
CONSTANTS.n = 1.1475E-3;
hmax = 3*3600.0; // 3 CMG's
   }
```

```
CONSTANTS.hmax = hmax;

   MatrixXd& J = CONSTANTS.J;

   J.resize(3,3);

   // Inertia matrix in slug-ft^2

   if (CASE==1) {
J(0,0) = 17834580.0 ; J(0,1)= 2787992.0; J(0,2)= 2873636.0;
J(1,0) = 2787992.0  ; J(1,1)= 2773815.0; J(1,2)= -863810.0;
J(2,0) = 28736361.0 ; J(2,1)= -863810.0; J(2,2)=  38030467.0;
   }

   else if (CASE==2) {
J(0,0) = 18836544.0 ; J(0,1)= 3666370.0;  J(0,2)= 2965301.0;
J(1,0) = 3666370.0  ; J(1,1)= 27984088.0; J(1,2)= -1129004.0;
J(2,0) = 2965301.0  ; J(2,1)= -1129004.0; J(2,2)=  39442649.0;
   }



////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name         = "Zero Propellant Maneouvre of the ISS";
    problem.outfilename                 = "zpm.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

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

    psopt_level1_setup(problem);

/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
/////////////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates    = 10;
    problem.phases(1).ncontrols  = 4;
    problem.phases(1).nevents             = 20;
    problem.phases(1).npath      = 4;
    problem.phases(1).nodes               = (RowVectorXi(5) << 20, 30, 40, 50, 60).finished();   // <<  20, 30, 40, 50, 60;


    problem.phases(1).nparameters         = 1;


    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////

   // Control bounds

    problem.phases(1).bounds.lower.controls(0) =  -1.0;
    problem.phases(1).bounds.lower.controls(1) =  -1.0;
    problem.phases(1).bounds.lower.controls(2) =  -1.0;
    problem.phases(1).bounds.lower.controls(3) =  -1.0;


    problem.phases(1).bounds.upper.controls(0) =   1.0;
    problem.phases(1).bounds.upper.controls(1) =   1.0;
    problem.phases(1).bounds.upper.controls(2) =   1.0;
    problem.phases(1).bounds.upper.controls(3) =   1.0;

    // state bounds

    problem.phases(1).bounds.lower.states(0) =  -1.0;
    problem.phases(1).bounds.lower.states(1) =  -0.2;
    problem.phases(1).bounds.lower.states(2) =  -0.2;
    problem.phases(1).bounds.lower.states(3) =  -1.0;
    problem.phases(1).bounds.lower.states(4) =  -1.E-2;
    problem.phases(1).bounds.lower.states(5) =  -1.E-2;
    problem.phases(1).bounds.lower.states(6) =  -1.E-2;
    problem.phases(1).bounds.lower.states(7) =  -8000.0;
```

```
    problem.phases(1).bounds.lower.states(8) =  -8000.0;
    problem.phases(1).bounds.lower.states(9) =  -8000.0;

    problem.phases(1).bounds.upper.states(0) =  1.0;
    problem.phases(1).bounds.upper.states(1) =  0.2;
    problem.phases(1).bounds.upper.states(2) =  0.2;
    problem.phases(1).bounds.upper.states(3) =  1.0;
    problem.phases(1).bounds.upper.states(4) =  1.E-2;
    problem.phases(1).bounds.upper.states(5) =  1.E-2;
    problem.phases(1).bounds.upper.states(6) =  1.E-2;
    problem.phases(1).bounds.upper.states(7) =  8000.0;
    problem.phases(1).bounds.upper.states(8) =  8000.0;
    problem.phases(1).bounds.upper.states(9) =  8000.0;

   // Parameter bound


    problem.phases(1).bounds.lower.parameters(0) =  0.0;
    problem.phases(1).bounds.upper.parameters(0) =  hmax*hmax;


    // Event bounds

    // Initial / Final  condition values:
    MatrixXd q_i(4,1),  q_f(4,1), omega_i(3,1), omega_f(3,1), h_i(3,1), h_f(3,1);

    adouble q_ad[4], omega_ad[3];

// Initial conditions
    if (CASE==1) {
     q_i(0) =  0.98966;
     q_i(1) =  0.02690;
     q_i(2) = -0.08246;
     q_i(3) =  0.11425;

        q_ad[ 0 ]=q_i(0);
        q_ad[ 1 ]=q_i(1);
        q_ad[ 2 ]=q_i(2);
        q_ad[ 3 ]=q_i(3);
        compute_omega0( omega_ad, q_ad);
        omega_i(0) = omega_ad[ 0 ].value();
        omega_i(1) = omega_ad[ 1 ].value();
        omega_i(2) = omega_ad[ 2 ].value();

 //    omega_i(1) = -2.5410E-4;
 //    omega_i(2) = -1.1145E-3;
 //    omega_i(3) =  8.2609E-5;

     h_i(0) =  -496.0;
     h_i(1) =  -175.0;
     h_i(2) = -3892.0;
    }

    else if (CASE==2) {
   q_i(0) =  0.98996;
     q_i(1) =  0.02650;
     q_i(2) = -0.07891;
     q_i(3) =  0.11422;
        q_ad[ 0 ]=q_i(0);
        q_ad[ 1 ]=q_i(1);
        q_ad[ 2 ]=q_i(2);
        q_ad[ 3 ]=q_i(3);
        compute_omega0( omega_ad, q_ad);
        omega_i(0) = omega_ad[ 0 ].value();
        omega_i(1) = omega_ad[ 1 ].value();
        omega_i(2) = omega_ad[ 2 ].value();

     h_i(0) =  1000.0;
     h_i(1) =  -500.0;
     h_i(2) = -4200.0;
    }

    // Final conditions
    q_f(0) =  0.70531;
    q_f(1) = -0.06201;
    q_f(2) = -0.03518;
    q_f(3) = -0.70531;

    q_ad[ 0 ]=q_f(0);
    q_ad[ 1 ]=q_f(1);
    q_ad[ 2 ]=q_f(2);
    q_ad[ 3 ]=q_f(3);
```

```
    compute_omega0( omega_ad, q_ad);
    omega_f(0) = omega_ad[ 0 ].value();
    omega_f(1) = omega_ad[ 1 ].value();
    omega_f(2) = omega_ad[ 2 ].value();

//    omega_f(1) = 1.1353E-3;
//    omega_f(2) = 3.0062E-6;
//    omega_f(3) = -1.5713E-4;

    h_f(0) = -9.0;
    h_f(1) = -3557.0;
    h_f(2) =  -135.0;

    double DQ = 0.0001;
    double DWF = 0.0;
    double DHF = 0.0;

    problem.phases(1).bounds.lower.events(0)  = q_i(0)-DQ;
    problem.phases(1).bounds.lower.events(1)  = q_i(1)-DQ;
    problem.phases(1).bounds.lower.events(2)  = q_i(2)-DQ;
    problem.phases(1).bounds.lower.events(3)  = q_i(3)-DQ;
    problem.phases(1).bounds.lower.events(4)  = omega_i(0);
    problem.phases(1).bounds.lower.events(5)  = omega_i(1);
    problem.phases(1).bounds.lower.events(6)  = omega_i(2);
    problem.phases(1).bounds.lower.events(7)  = h_i(0);
    problem.phases(1).bounds.lower.events(8)  = h_i(1);
    problem.phases(1).bounds.lower.events(9)  = h_i(2);
    problem.phases(1).bounds.lower.events(10) = q_f(0)-DQ;
    problem.phases(1).bounds.lower.events(11) = q_f(1)-DQ;
    problem.phases(1).bounds.lower.events(12) = q_f(2)-DQ;
    problem.phases(1).bounds.lower.events(13) = q_f(3)-DQ;
    problem.phases(1).bounds.lower.events(14) = omega_f(0)-DWF;
    problem.phases(1).bounds.lower.events(15) = omega_f(1)-DWF;
    problem.phases(1).bounds.lower.events(16) = omega_f(2)-DWF;
    problem.phases(1).bounds.lower.events(17) = h_f(0)-DHF;
    problem.phases(1).bounds.lower.events(18) = h_f(1)-DHF;
    problem.phases(1).bounds.lower.events(19) = h_f(2)-DHF;


    problem.phases(1).bounds.upper.events(0)  = q_i(0)+DQ;
    problem.phases(1).bounds.upper.events(1)  = q_i(1)+DQ;
    problem.phases(1).bounds.upper.events(2)  = q_i(2)+DQ;
    problem.phases(1).bounds.upper.events(3)  = q_i(3)+DQ;
    problem.phases(1).bounds.upper.events(4)  = omega_i(0);
    problem.phases(1).bounds.upper.events(5)  = omega_i(1);
    problem.phases(1).bounds.upper.events(6)  = omega_i(2);
    problem.phases(1).bounds.upper.events(7)  = h_i(0);
    problem.phases(1).bounds.upper.events(8)  = h_i(1);
    problem.phases(1).bounds.upper.events(9)  = h_i(2);
    problem.phases(1).bounds.upper.events(10) = q_f(0)+DQ;
    problem.phases(1).bounds.upper.events(11) = q_f(1)+DQ;
    problem.phases(1).bounds.upper.events(12) = q_f(2)+DQ;
    problem.phases(1).bounds.upper.events(13) = q_f(3)+DQ;
    problem.phases(1).bounds.upper.events(14) = omega_f(0)+DWF;
    problem.phases(1).bounds.upper.events(15) = omega_f(1)+DWF;
    problem.phases(1).bounds.upper.events(16) = omega_f(2)+DWF;
    problem.phases(1).bounds.upper.events(17) = h_f(0)+DHF;
    problem.phases(1).bounds.upper.events(18) = h_f(1)+DHF;
    problem.phases(1).bounds.upper.events(19) = h_f(2)+DHF;

// Path bounds

    double hdotmax = 200.0; // [ ft-lbf ]

    double EQ_TOL = 0.0002;

    problem.phases(1).bounds.lower.path(0) = 1.0-EQ_TOL;
    problem.phases(1).bounds.upper.path(0) = 1.0+EQ_TOL;

    problem.phases(1).bounds.lower.path(1) = 1.0-EQ_TOL;
    problem.phases(1).bounds.upper.path(1) = 1.0+EQ_TOL;


    problem.phases(1).bounds.lower.path(2) = -hmax*hmax;
    problem.phases(1).bounds.upper.path(2) =  0.0;


    problem.phases(1).bounds.lower.path(3) = 0.0;
    problem.phases(1).bounds.upper.path(3) = hdotmax*hdotmax;

    // Time bounds
```

```
    double TFINAL;

    if (CASE==1) {
TFINAL = 6000.0;
    }
    else {
TFINAL = 7200.0;
    }

    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = TFINAL;
    problem.phases(1).bounds.upper.EndTime      = TFINAL;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost      = &integrand_cost;
    problem.endpoint_cost       = &endpoint_cost;
    problem.dae                = &dae;
    problem.events          = &events;
    problem.linkages         = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

   MatrixXd time_guess;      time_guess      = linspace(0.0, TFINAL, 50 );
   MatrixXd state_guess;     state_guess     = zeros(10,50);
   MatrixXd control_guess;   control_guess   = zeros(4,50);
   MatrixXd parameter_guess; parameter_guess = hmax*hmax*ones(1,1);

   control_guess.row(0) = linspace( q_i(0), q_i(0), 50 );
   control_guess.row(1) = linspace( q_i(1), q_i(1), 50 );
   control_guess.row(2) = linspace( q_i(2), q_i(2), 50 );
   control_guess.row(3) = linspace( q_i(3), q_i(3), 50 );

   state_guess.row(0) = linspace( q_i(0), q_i(0), 50);
   state_guess.row(1) = linspace( q_i(1), q_i(1), 50);
   state_guess.row(2) = linspace( q_i(2), q_i(2), 50);
   state_guess.row(3) = linspace( q_i(3), q_i(3), 50);

   state_guess.row(4) = linspace( omega_i(0), omega_f(0), 50);
   state_guess.row(5) = linspace( omega_i(1), omega_f(1), 50);
   state_guess.row(6) = linspace( omega_i(2), omega_f(2), 50);

   state_guess.row(7) = linspace( h_i(0), h_f(0), 50);
   state_guess.row(8) = linspace( h_i(1), h_f(1), 50);
   state_guess.row(9) = linspace( h_i(2), h_f(2), 50);



   problem.phases(1).guess.controls    = control_guess;
   problem.phases(1).guess.states      = state_guess;
   problem.phases(1).guess.time        = time_guess;
   problem.phases(1).guess.parameters  = parameter_guess;



////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-5;
    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.defect_scaling              = "jacobian-based";
    algorithm.jac_sparsity_ratio          = 0.104;




////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////////
```

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

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////


    MatrixXd states, controls, t;

    states      = solution.get_states_in_phase(1);
    controls    = solution.get_controls_in_phase(1);
    t           = solution.get_time_in_phase(1);


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(states,"states.dat");
    Save(controls,"controls.dat");
    Save(t,"t.dat");

    MatrixXd omega, h, q, phi, theta, psi, qc, euler_angles;

    q      = states.block(0,0,4,length(t));
    omega  = states.block(4,0,3,length(t));
    h      = states.block(7,0,3,length(t));
    qc     = controls;

    quarternion2Euler(phi, theta, psi,  q);


    euler_angles.resize(3,length(t));

    euler_angles << phi ,
                    theta ,
                    psi;

    adouble  qc_ad[4], u_ad[3];

    MatrixXd u(3,length(t));
    MatrixXd hnorm(1,length(t));
    MatrixXd hi;
    MatrixXd hm = hmax*ones(1,length(t));


    int i,j;

  for (i=0; i< length(t); i++ ) {
for(j=0;j<3;j++) {
   omega_ad[j] = omega(j,i);
        }
for(j=0;j<4;j++) {
   q_ad[j] = q(j,i);
           qc_ad[j]= qc(j,i);
        }

compute_control_torque(u_ad, q_ad, qc_ad, omega_ad );

for(j=0;j<3;j++) {
   u(j,i) = u_ad[j].value();
   }

   hi = h.col(i);

hnorm(0,i) = hi.norm();

    }

    omega = omega*(180.0/pi)*1000; // convert to mdeg/s

    phi = phi*180.0/pi; theta=theta*180.0/pi; psi=psi*180.0/pi;


    Save(u,"u.dat");

    Save(euler_angles,"euler_angles.dat");



///////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
```

```
//////////////////////////////////////////////////////////////////////////////


    plot(t,q,problem.name+" quarternion elements: q",  "time (s)", "q", "q");

    plot(t,qc,problem.name+" Control variables: qc",  "time (s)", "qc", "qc");

    plot(t,phi,problem.name+" Euler angles: phi",     "time (s)", "angles (deg)", "phi");

    plot(t,theta,problem.name+" Euler angles: theta", "time (s)", "angles (deg)", "theta");

    plot(t,psi,problem.name+" Euler angle: psi",       "time (s)", "psi (deg)", "psi");

    plot(t,omega.row(0),problem.name+": omega 1","time (s)", "omega1", "omega1");

    plot(t,omega.row(1),problem.name+": omega 2","time (s)", "omega2", "omega2");

    plot(t,omega.row(2),problem.name+": omega 3","time (s)", "omega3", "omega3");

    plot(t,h.row(0),problem.name+": momentum 1","time (s)", "h1", "h1");

    plot(t,h.row(1),problem.name+": momentum 2","time (s)", "h2", "h2");

    plot(t,h.row(2),problem.name+": momentum 3","time (s)", "h3", "h3");

    plot(t,u.row(0),problem.name+": control torque 1","time (s)", "u1", "u1");

    plot(t,u.row(1),problem.name+": control torque 2","time (s)", "u2", "u2");

    plot(t,u.row(2),problem.name+": control torque 3","time (s)", "u3", "u3");

    plot(t,hnorm,t,hm,problem.name+": momentum norm", "time (s)", "h", "h hmax");


    plot(t,phi,problem.name+" Euler angles: phi",     "time (s)", "angles (deg)", "phi",
       "pdf", "zpm_phi.pdf" );

    plot(t,theta,problem.name+" Euler angles: theta", "time (s)", "angles (deg)", "theta",
       "pdf", "zpm_theta.pdf");

    plot(t,psi,problem.name+" Euler angle: psi",       "time (s)", "psi (deg)", "psi",
       "pdf", "zpm_psi.pdf");

    plot(t,omega.row(0),problem.name+": omega 1","time (s)", "omega1", "omega1",
       "pdf", "zpm_omega1.pdf");

    plot(t,omega.row(1),problem.name+": omega 2","time (s)", "omega2", "omega2",
       "pdf", "zpm_omega2.pdf");

    plot(t,omega.row(2),problem.name+": omega 3","time (s)", "omega3", "omega3",
       "pdf", "zpm_omega3.pdf");

    plot(t,h.row(0),problem.name+": momentum 1","time (s)", "h1", "h1",
              "pdf", "zpm_h1.pdf");

    plot(t,h.row(1),problem.name+": momentum 2","time (s)", "h2", "h2",
      "pdf", "zpm_h2.pdf");

    plot(t,h.row(2),problem.name+": momentum 3","time (s)", "h3", "h3",
      "pdf", "zpm_h3.pdf");

    plot(t,u.row(0),problem.name+": control torque 1","time (s)", "u1", "u1",
      "pdf", "zpm_u1.pdf");

    plot(t,u.row(1),problem.name+": control torque 2","time (s)", "u2", "u2",
      "pdf", "zpm_u2.pdf");

    plot(t,u.row(2),problem.name+": control torque 3","time (s)", "u3", "u3",
      "pdf", "zpm_u3.pdf");

    plot(t,hnorm,t,hm,problem.name+": momentum norm", "time (s)", "h (ft-lbf-sec)", "h hmax",
      "pdf", "zpm_hnorm.pdf");

    plot(t,u,problem.name+": control torques","time (s)", "u (ft-lbf)", "u1 u2 u3",
      "pdf", "zpm_controls.pdf");

}

//////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
```

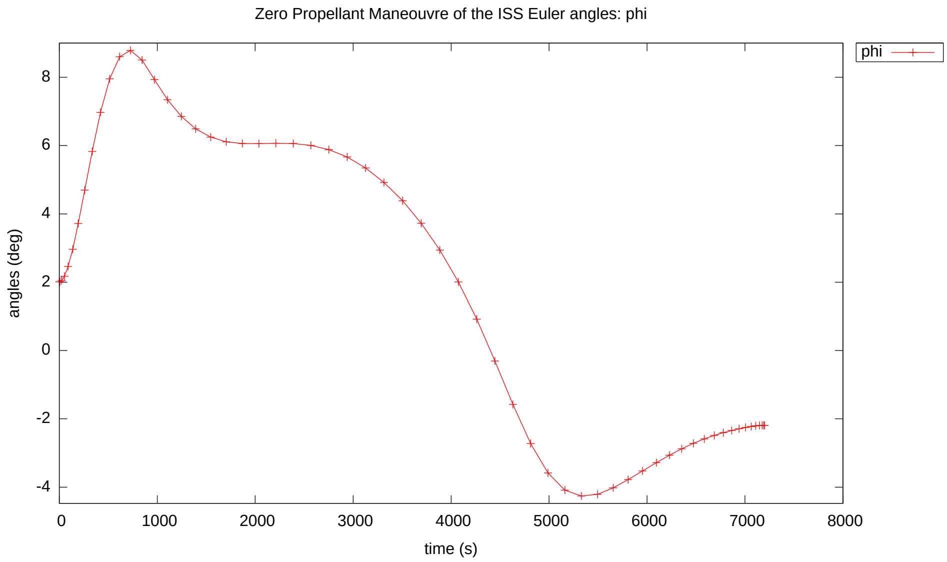

Figure 3.115: Euler angle ϕ (roll)

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 3.115 to 3.127..

```
PSOPT results summary
=====================

Problem:  Zero Propellant Maneouvre of the ISS
CPU time (seconds): 1.073687e+02
NLP solver used:  IPOPT
PSOPT release number:  5.0
Date and time of this run:  Wed Sep 23 12:30:18 2020

Optimal (unscaled) cost function value:  6.680106e+06
Phase 1 endpoint cost function value: 5.239629e+06
Phase 1 integrated part of the cost:  1.440477e+06
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 7.200000e+03
Phase 1 maximum relative local error: 1.019370e-03
NLP solver reports:  The problem has been solved!
```

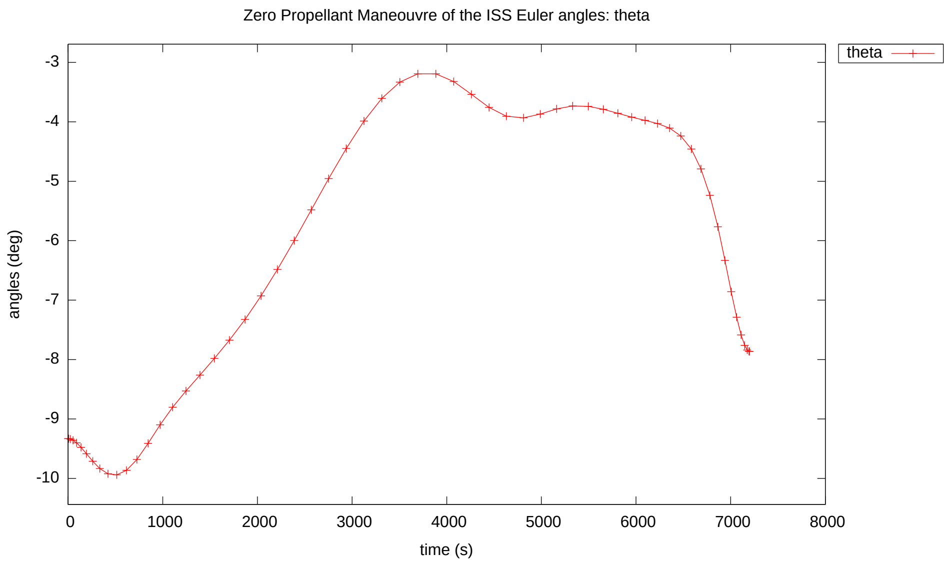

Figure 3.116: Euler angle θ (pitch)

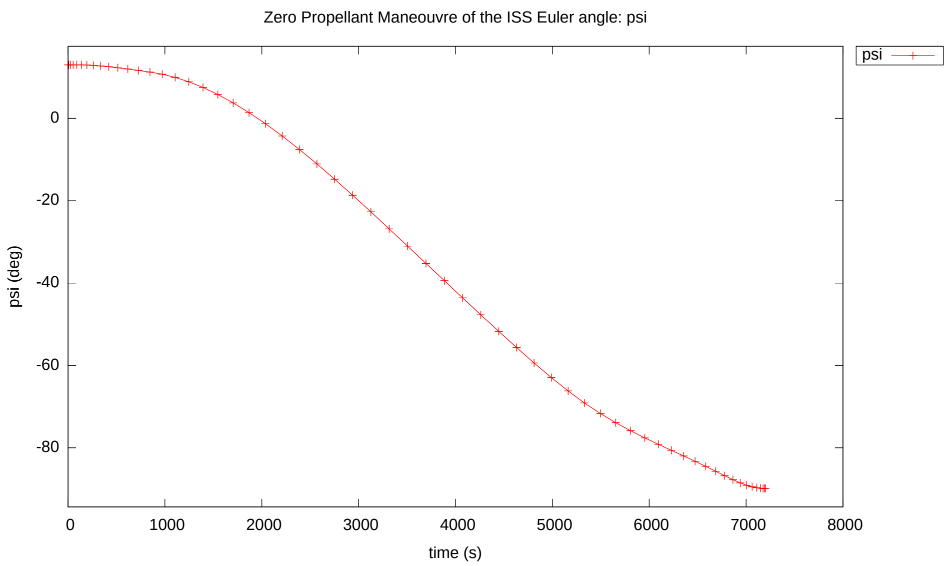

Figure 3.117: Euler angle ψ (yaw)

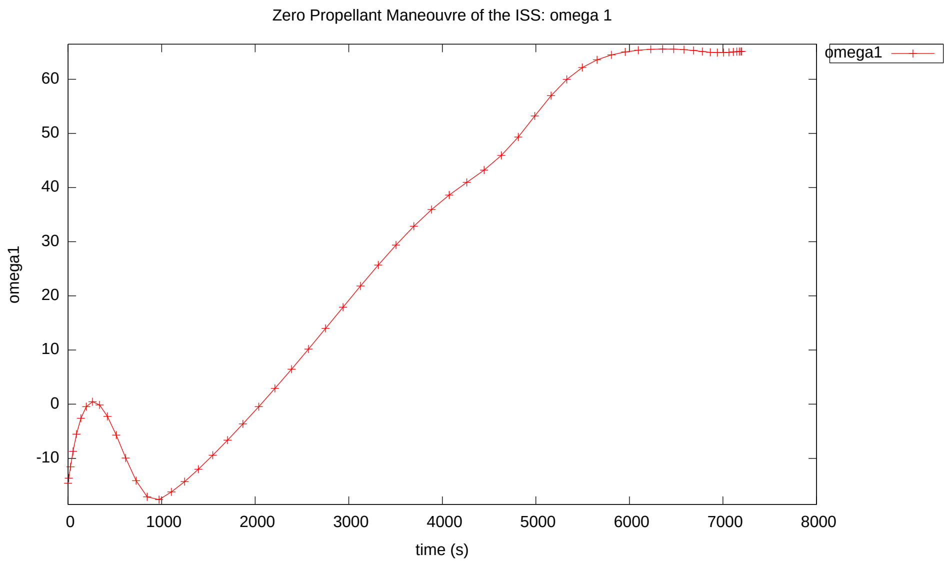

Figure 3.118: Angular speed ω_1 (roll)

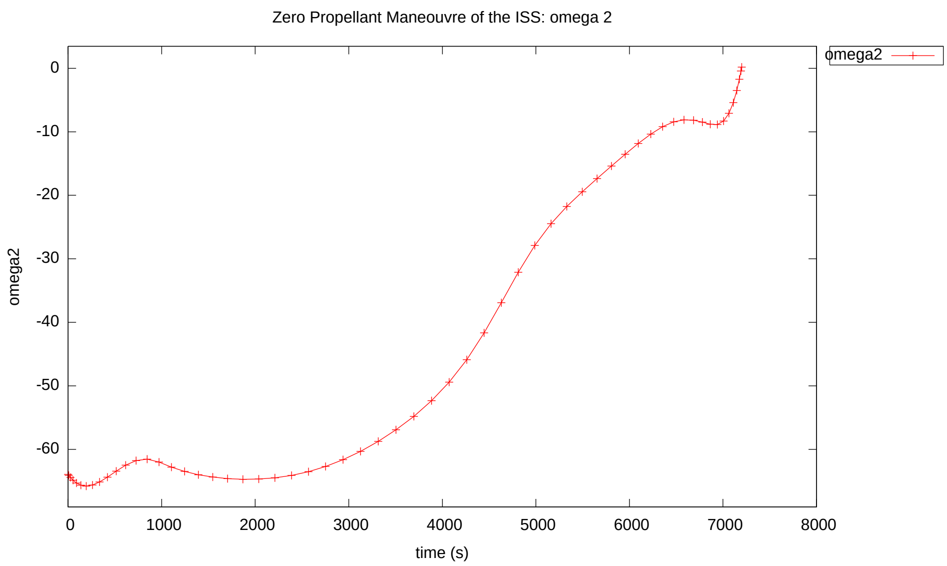

Figure 3.119: Angular speed ω_2 (pitch)

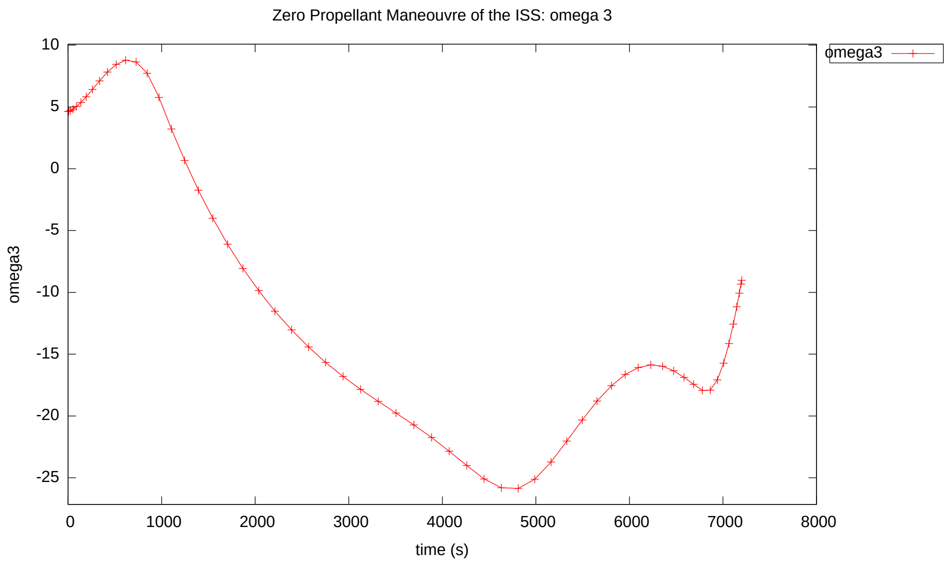

Figure 3.120: Angular speed ω_3 (yaw)

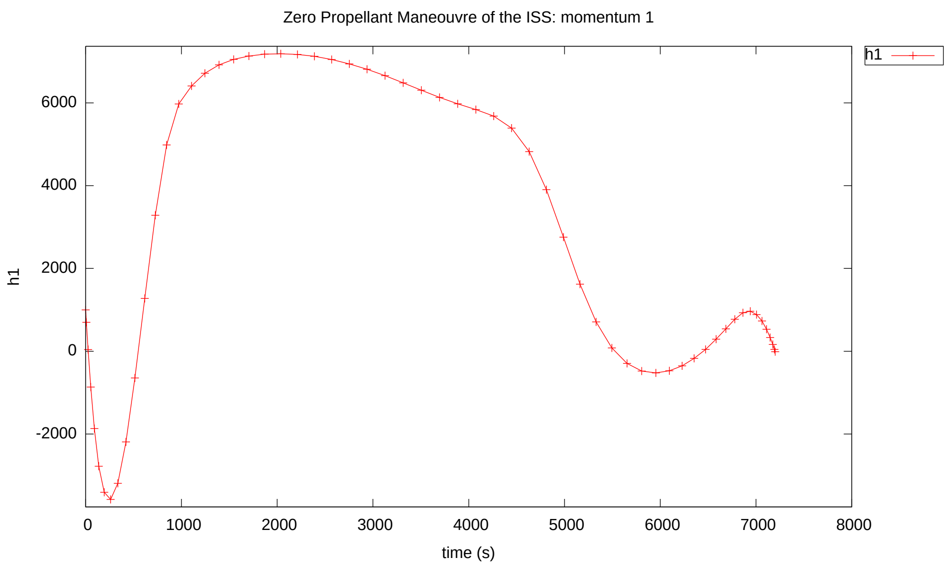

Figure 3.121: Momentum h_1 (roll)

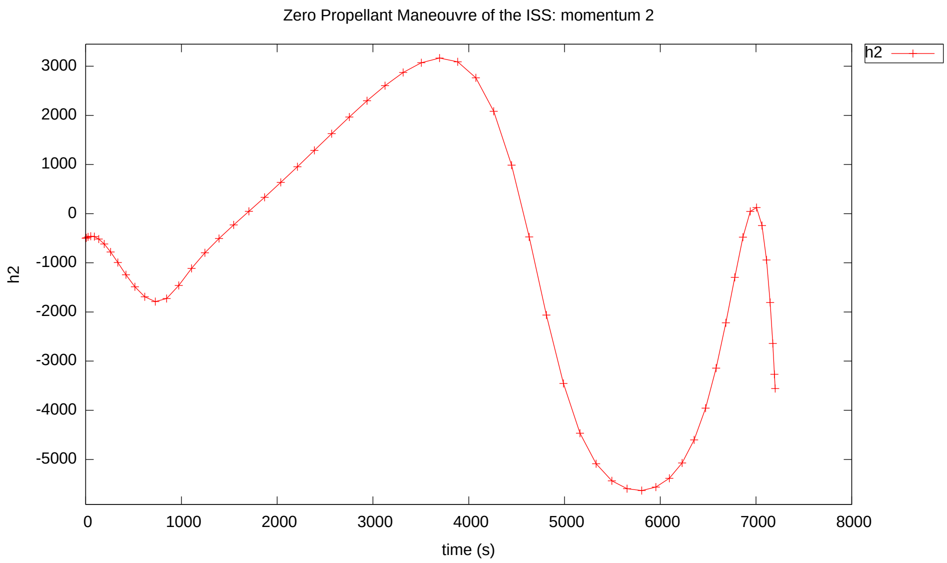

Figure 3.122: Momentum h_2 (pitch)

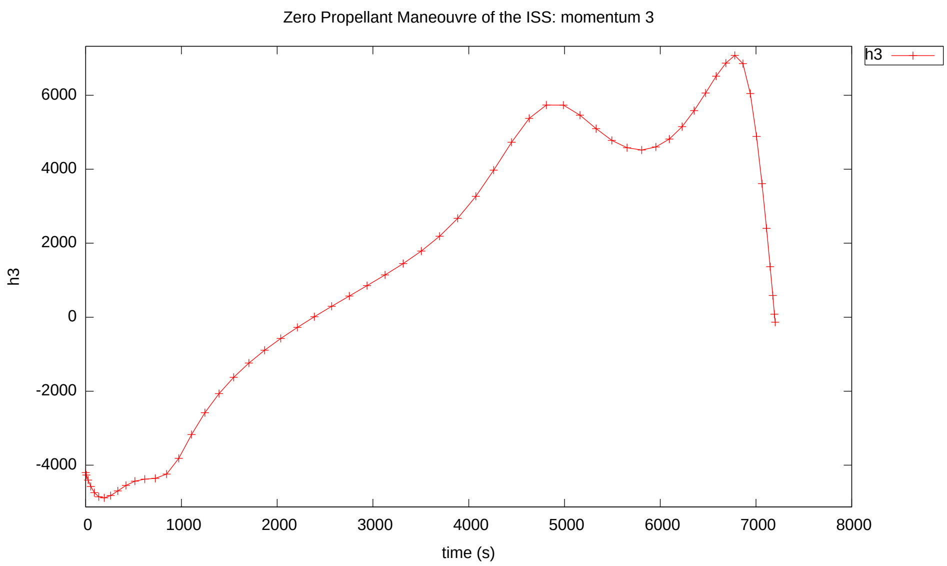

Figure 3.123: Momentum h_3 (yaw)

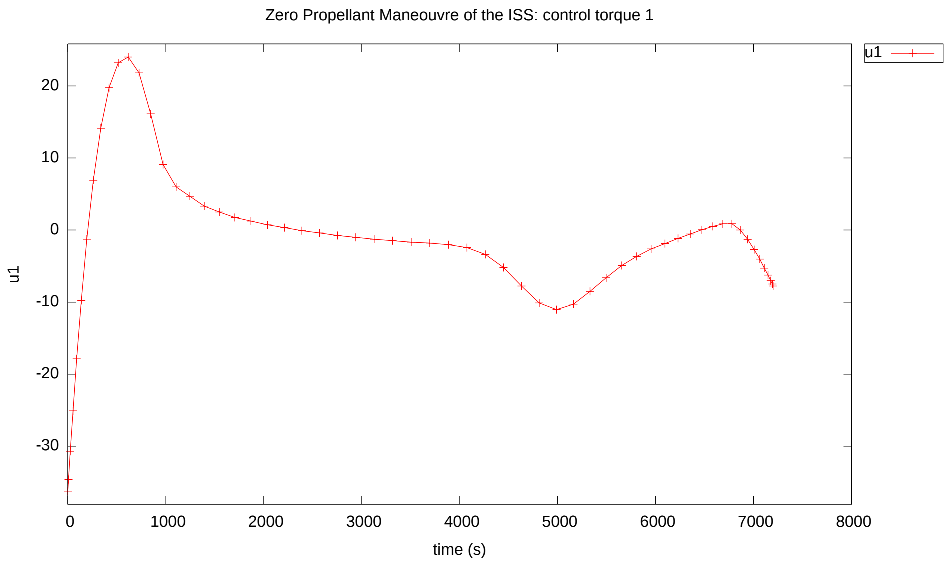

Figure 3.124: Control torque u_1 (roll)

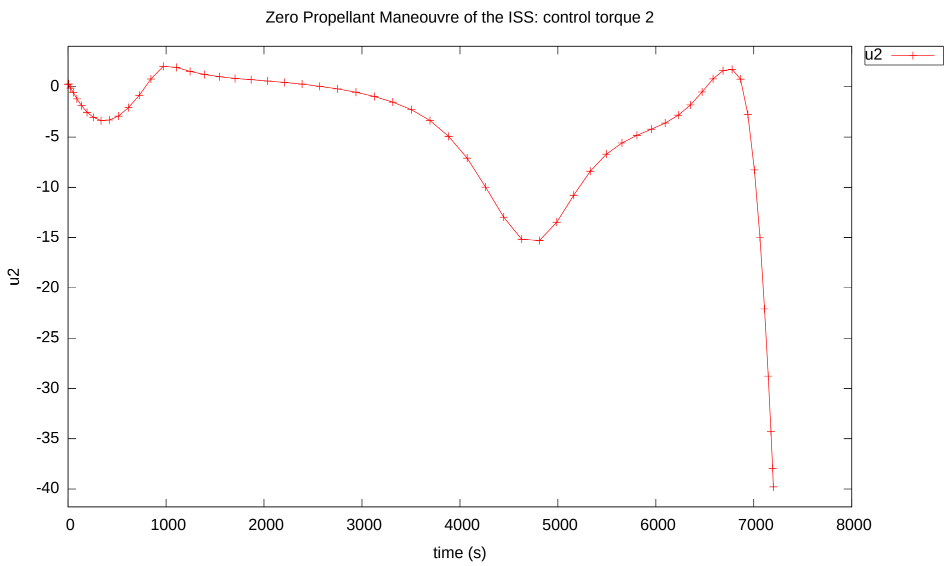

Figure 3.125: Control torque u_2 (pitch)

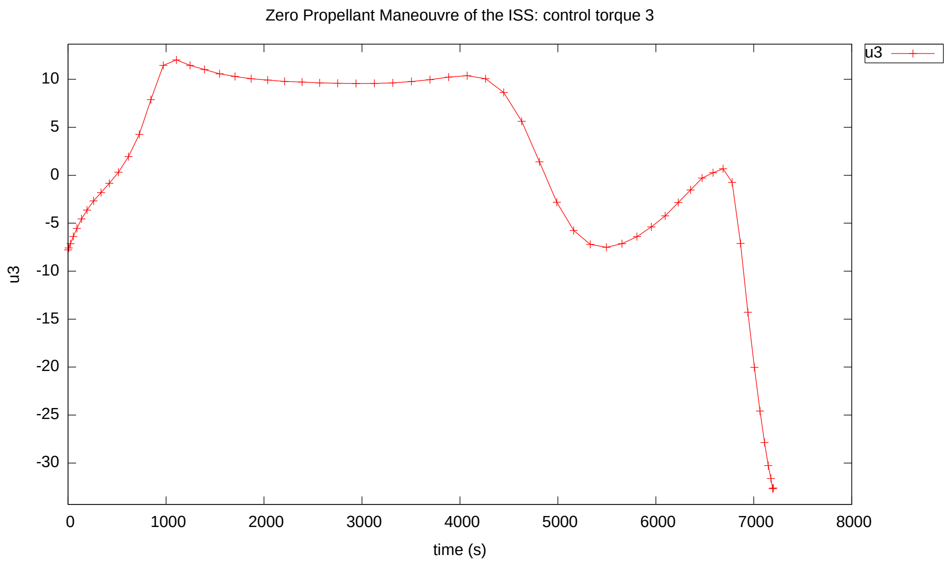

Figure 3.126: Control torque u_3 (yaw)

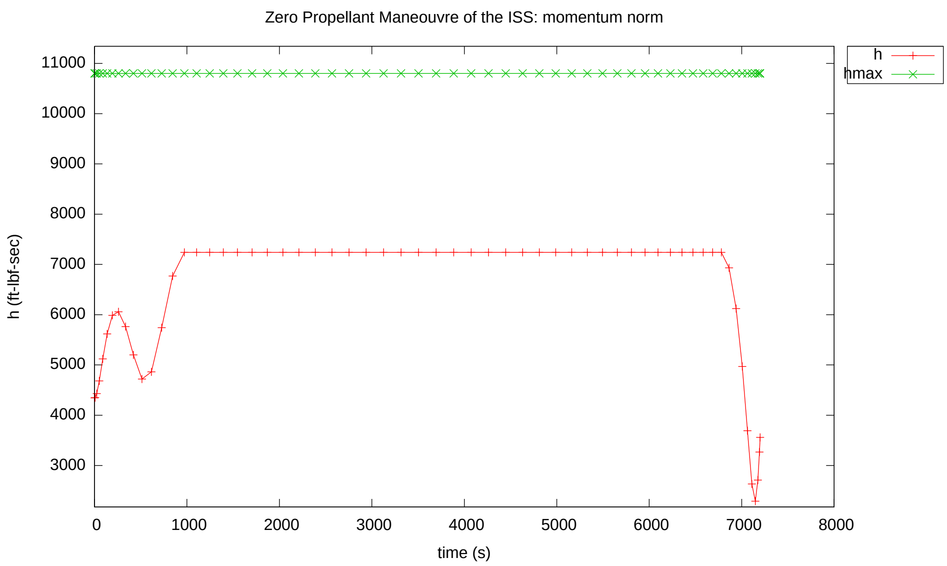

Figure 3.127: Momentum norm $||\mathbf{h}(t)||$

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] N.S. Bedrossian, S. Bhatt, W. Kang, and I.M. Ross. Zero Propellant Maneuver Guidance. *IEEE Control Systems Magazine*, 29:53–73, 2009.

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

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

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

[5] S.A. Bhatt. Optimal reorientation of spacecraft using only control moment gyroscopes. Master's thesis, Rice University, Houston, Texas, 2007.

[6] A.E. Bryson. *Dynamic Optimization*. Addison-Wesley, 1999.

[7] A.E. Bryson, M.N. Desai, and W.C. Hoffman. Energy-State Approximation in Performance Optimization of Supersonic Aircraft. *Journal of Aircraft*, 6:481–488, 1969.

[8] A.E. Bryson and Yu-Chi Ho. *Applied Optimal Control*. Halsted Press, 1975.

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

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

[11] C. Canuto, M.Y. Hussaini, A. Quarteroni, and T.A. Zang. *Spectral Methods: Evolution to Complex Geometries and Applications to Fluid Dynamics*. Springer, Heidelberg, Germany, 2007.

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

[13] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the Estimation of Sparse Jacobian Matrices. *Journal of the Institute of Mathematics and Applications*, 13:117–120, 1974.

[14] E. D. Dolan and J. J. More. *Benchmarking optimization software with COPS 3.0*. Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439, 2004.

[15] 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.

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

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

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

[19] J. Franke and M. Otter. The manutec r3 benchmark models for the dynamic simulation of robots. Technical report, Institute for Robotics and System Dynamics, DLR Oberpfaffenhofen, 1993.

[20] Q. Gong, F. Fahroo, and I.M. Ross. Spectral algorithms for pseudospectral methods in optimal control. *Journal of Guidance Control and Dynamics*, 31:460–471, 2008.

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

[22] L.S. Jennings, M.E. Fisher, K.L. Teo, and C.J. Goh. *MISER3 Optimal Control Software Version 2.0 Theory and User Manual*. Department of Mathematics, The University of Western Australia, 2002.

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

[24] 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.

[25] 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.

[26] Z. Li, M. R. Osborne, and T. Prvan. Parameter estimation of ordinary differential equations. *IMA Journal of Numerical Analysis*, 25:264–285, 2005.

[27] R. Luus. *Iterative Dynamic Programming*. Chapman and Hall / CRC, 2002.

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

[29] M. Otter and S. Turk. The dfvrl models 1 and 2 of the manutec r3 robot. Technical report, Institute for Robotics and System Dynamics, DLR Oberpfaffenhofen, Germany, 1987.

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

[31] A.V. Rao, D. Benson, G. Huntington, and C. Francolin. *User's manual for GPOPS version 1.3: A Matlab package for dynamic optimization using the Gauss pseudospectral method*. 2008.

[32] A.V. Rao and K.D. Mease. Eigenvector Approximate Dichotomic Basis Method for Solving Hyper- sensitive Optimal Control Problems. *Optimal Control Applications and Methods*, 21:1–19, 2000.

[33] 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.

[34] P. E. Rutquist and M. M. Edvall. *PROPT Matlab Optimal Control Software*. TOMLAB Optimization, 2009.

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

[36] O. Von Stryk. *User's guide for DIRCOL (Version 2.1): A direct collocation method for the numerical solution of optimal control problems*. Technical Report, Technische Universitat Munchen, 1999.

[37] S. Subchan and R. Zbikowski. *Computational optimal control: tools and practice*. Wiley, 2009.

[38] K.L. Teo, C.J. Goh, and K.H. Wong. *A Unified Computational Approach to Optimal Control Problems*. Wiley, New York, 1991.

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

[40] D.A. Vallado. *Fundamentals of Astrodynamics and Applications*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.

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

[42] A. Wächter and L. T. Biegler. On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming. *Mathematical Programming*, 106:25–57, 2006.