NASA/TM–2014-218532

MADS Users' Guide

Daniel D. Moerder
Langley Research Center, Hampton, Virginia

October 2014

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question to help@sti.nasa.gov

- Fax your question to the NASA STI Information Desk at 443-757-5803

- Phone the NASA STI Information Desk at 443-757-5802

- Write to:
  STI Information Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/TM–2014-218532

MADS Users' Guide

Daniel D. Moerder
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

October 2014

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Abstract

MADS (Minimization Assistant for Dynamical Systems) is a trajectory optimization code in which a user-specified performance measure is directly minimized, subject to constraints placed on a low-order discretization of user-supplied plant ordinary differential equations. This document describes the mathematical formulation of the set of trajectory optimization problems for which MADS is suitable, and describes the user interface. Usage examples are provided.

Contents

1 Introduction

This document describes the user interface and provides a usage tutorial for the MADS (Minimization Agent for Dynamical Systems) trajectory optimization tool. MADS comprises a FORTRAN95 subroutine that organizes and executes trajectory optimization computations, and a set of Matlab functions that manipulate MADS input and output data. MADS casts a trajectory optimization problem as direct cost function minimization subject to a set of constraints that include a temporal discretization of the first-order ordinary differential equations that govern the user's plant dynamics, boundary conditions, and miscellaneous constraints imposed on the trajectory between boundary conditions. The resulting nonlinear programming problem (NLP) is solved by the NLP software SNOPT [1], which must be available in order to run MADS.

The user provides four problem-specific subroutines that realize the system state rates, the boundary conditions, the trajectory path constraints, and the cost function. Given the maturity and benefit of autodifferentiation (AD) software, MADS assumes the presence of autodifferentiated versions of the four user routines to supply derivatives. MADS' user subroutine interfaces assume the use of the TAPANADE [2] AD package.

The approach taken in MADS to posing and solving trajectory optimization problems is to encourage robust convergence to a solution by using a low-order discretization and a control parameterization which can gracefully accommodate the temporal discontinuities which can appear on the interior of trajectory arcs when exploring an optimal control problem. The low-order discretization and control is normally accompanied by a fairly fine, uniformly distributed, mesh of time points over which the problem is solved.

This emphasis on simplicity and accommodation of nonsmooth control behavior places MADS somewhat out of the mainstream of emerging numerical optimal control technology, much of which has been emphasizing pseudospectral techniques [3] – high-order quadrature of orthogonal polynomials, with adaptive temporal discretization mesh logic. These emerging techniques show very good performance and high accuracy on temporally smooth problems, but graceful treatment of nonsmooth behavior remains an area of active research.

Section 2 describes the family of optimal control problems that can be solved with MADS, and describes its software interface and that of the user-supplied routines. It also describes the Matlab utility functions that are used for manipulating MADS initial guesses and solutions.

Section 3 sets up and solves two simple, classic optimal control problems, both of which exhibit nonsmooth temporal behavior issues, several different ways. Simple measures are described for varying the discretization density over the trajectory, and for constraining aspects of the control trajectory's behavior.

2 Problem Formulation and Software Interface

MADS operates on trajectories of the form

$$\frac{dx_k}{dt} = f(x_k, u_k, p), \quad t \in [0,1], \quad k = 1, ..., \texttt{nph} \tag{1}$$

that is, for trajectories which may (when $\texttt{nph} > 1$) consist of multiple subarcs, or "phases." In each k^{th} phase, $x_k \in \mathcal{R}^{\texttt{nx}_k}$ are states, and $p \in \mathcal{R}^{\texttt{np}}$ is a vector of "static" free parameters; that is, parameters to be chosen by the optimization process, and which do not vary with time. Note that while x_k and u_k are specific to the k^{th} phase in (1), all of the elements of p are visible to all nph phases of the trajectory. Each $u_k(t)$ is assumed to be an integrable nu-dimensional functional defined on the interval $t \in [0,1]$. The integrability assumption is more-or-less moot, since MADS recasts the problem in discrete time. It is, however, the case that if the user attempts to solve a problem for which an integrable optimal solution does not exist, numerical difficulties will ensue. Subsection 2.2 of the Tutorials gives an example where optimal control leads to a nonintegrable "optimal" control solution, and gives approaches for compensating for the nonintegrability. It should also be noted that the unity duration of the subarcs in (1) does not restrict the user from posing variable-time problems. There are a number of easy ways to do this, and literally all of the examples in the Tutorial involve free terminal time. The very simplest is to use an element of p to scale time. Restricting to $\texttt{nph} = 1$ and p scalar for simplicity, transform t to τ via $\tau = \texttt{p}t$ so that

$$\frac{dx}{d\tau} = pf(x, u, p), \quad \tau \in [0,1] \tag{2}$$

Alternatively, an element, say u_τ, of u can be used as a time scaling parameter; that is, $\tau(t) = u_\tau(t)$, giving

$$\left.\frac{dx}{d\tau}\right|_{\tau = u_\tau(t)} = u_\tau f(x, u, p), \quad \tau \in [0, \int_0^1 u_\tau(t)dt], \qquad u_\tau \geq c_\tau > 0 \tag{3}$$

where c_τ is a user-specified constant. This latter formulation permits the flexibility of variable time steps, at the cost of some additional complexity. The Tutorial includes problems using both approaches.

Free parameters in the trajectory – state boundary values, control functionals, and the p vector are chosen to minimize a Mayer-type cost function

$$\phi(x_{10}, x_{1f}, x_{20}, x_{2f}, \ldots, x_{\texttt{nph}0}, x_{\texttt{nph}f}, p), \quad \left\{ \begin{array}{rcl} x_{k0} & = & x_k(0) \\ x_{kf} & = & x_k(1) \end{array} \right. \tag{4}$$

The cost function ϕ is minimized subject to a discretization of (1) and, optionally, boundary conditions

$$\psi_j(x_{10}, x_{1f}, \ldots, x_{\texttt{nph}0}, x_{\texttt{nph}f}, p) \left\{ \begin{array}{ll} = 0, & \texttt{iebcvec(j)} = 0 \\ \geq 0, & \texttt{iebcvec(j)} = 1 \end{array} \right. \quad \texttt{j} = 1, \ldots, \texttt{nbc} \tag{5}$$

where nbc is the dimension of ψ and iebcvec(j) is a user-specified flag that controls whether the $\texttt{j}^{th}$ element of ψ is to be treated as an equality or inequality constraint. Again optionally, constraints can be imposed on the trajectory between boundary conditions using a discretization of trajectory constraints of the form

$$c_j(x_k, u_k, p) \left\{ \begin{array}{ll} = 0, & \texttt{iecv(j)} = 0 \\ \geq 0, & \texttt{iecv(j)} = 1 \end{array} \right. \quad \texttt{k} = 1, \ldots, \texttt{nph}, \ \texttt{j} = 1, \ldots, \texttt{nc}_k \tag{6}$$

where iecv is a user-specified flag. Note that the number of trajectory constraints in each of the nph subarcs, or "phases" may vary from phase to phase.

MADS computes optimized trajectories by direct minimization of the cost function ϕ, using finite-dimensional approximations of the state and control trajectories, the former obtained via low-order collocation. Each k^{th} trajectory arc in (1) is broken into $\texttt{nd}_k$ equal time subintervals and the state trajectory across each subinterval is approximated by collocation:

$$x_{j+1} - x_j = F(x_j, x_{j+1}, u_j, p), \quad j = 1, \ldots, \texttt{nd}_k \tag{7}$$

where u_j is constant, i.e., zero-order hold (ZOH) across the discretization intervals. Note that, while each k^{th} discretized phase has one value u_j per discretization interval, for a total of $\texttt{nd}_k$ values, there are $\texttt{nd}_k + 1$ corresponding values of x_j, since each phase has an initial and terminal state.
The overall organization of constraints in MADS is

$$Z = \begin{bmatrix} z_1 \\ \vdots \\ z_{\texttt{nph}} \\ \psi \end{bmatrix}, \quad z_j = \begin{bmatrix} y_1 \\ \vdots \\ y_{\texttt{nd}_j} \end{bmatrix}, \quad y_i = \begin{bmatrix} F_{ij} \\ C_{ij} \end{bmatrix} \tag{8}$$

where F_{ij} is the discretization (7) for the i^{th} time step of the j^{th} phase, and C_{ij} is the corresponding constraint from (6).
There are currently three discretization expressions implemented, and the MADS input parameter `kode=kodev(k)` controls which of them is used to discretize the k^{th} phase of the trajectory:

1. Midpoint Euler (ME) (`kode=0`)

$$x_{j+1} - x_j = \frac{1}{\texttt{nd}_k} f\left(\frac{x_j + x_{j+1}}{2}, u_j, p\right) \tag{9}$$

   This is the most efficient discretization, since it achieves second-order accuracy with only one state derivative computation per time step.

2. Second-Order Runge-Kutta (RK2) (`kode=1`)

$$\left.\begin{array}{rcl} x_{j+1} &=& x_j + (1/4)(\eta_1 + 3\eta_2) \\ \eta_1 &=& (1/\texttt{nd}) f(x_j, u_j, p) \\ \eta_2 &=& (1/\texttt{nd}) f(x_j + (2/3)\eta_1, u_j, p) \end{array}\right\} \tag{10}$$

   This Runge-Kutta discretization requires twice as many state derivative computations as the ME, but has the property that the discretization is not dependent on x_{j+1}.

3. Fourth-Order Runge-Kutta (RK4) (`kode=2`)

$$\left.\begin{array}{rcl} x_{j+1} &=& (1/6)(\eta_1 + 2\eta_2 + 2\eta_3 + \eta_4) \\ \eta_1 &=& (1/\texttt{nd}) f(x_j, u_j, p) \\ \eta_2 &=& (1/\texttt{nd}) f(x_j + (1/2)\eta_1, u_j, p) \\ \eta_3 &=& (1/\texttt{nd}) f(x_j + (1/2)\eta_2, u_j, p) \\ \eta_4 &=& (1/\texttt{nd}) f(x_j + \eta_3, u_j, p) \end{array}\right\} \tag{11}$$

   This RK4 requires four times as many state derivative computations as the ME, but has a clear advantage in the fourth-order accuracy with which it discretizes the state trajectory.

2.1 Excursus: Custom Problem Formulations in MADS

At first glance, the user may balk at MADS' apparent crudity, most clearly exhibited in the zero-order hold (ZOH) imposed on the control variable. As the user's experience with MADS grows, however, this should mature into appreciation for MADS' flexibility. The ZOH is actually more of a "feature" of MADS, rather than a limitation. MADS is primarily intended to be used with the ME discretization and relatively small time steps. The use of a low-order discretization and small time steps encourages robust convergence to the problem solution, and the small time steps allow the ZOH to closely approximate the continuous-time optimal control trajectory.

All that being said, as stated in the Introduction, MADS is intended to provide a convenient "blank sheet" environment in which the user can formulate a wide range of trajctory optimization problems without being particularly shackled by canned dynamical model structures, or presumptions about the temporal behavior of the control trajectory.

In order to illustrate this, we consider adjustments to the standard MADS problem formulation that permit control trajectories more complicated than ZOH. Since the ME only samples the control trajectory at one point per discretization interval, why might a user want to complicate the control parameterization? Two reasons are to achieve higher numerical precision, or to obtain a control solution that accommodates known characteristics of the hardware that would actually realize the implement the control being optimized..

- The RK4 discretization may be used to check the validity of a ME-based solution by re-solving the problem on the set of time points used for the MEs solution and verifying that the RK4 and ME solutions are "sufficiently close." Since the RK4 samples the control trajectory at four points rather than one, fidelity to the continuous-time optimal control trajectory can be improved by choosing a more complicated control parameterization that varies over the discretization interval.

- Optimal control trajectories are typically unrealistic in the sense that, while actual implemented controls are the response of finite-bandwidth actuators to commands, actuation dynamics are not typically included in plant models for trajectory optimization. The control parameterization examples described below all involve the use of state variables. With these in hand, the user can easily constrain or penalize unrealistic control features, such as "instantaneous" jumps that would require very expensive actuators to implement. The first solution example includes a demonstration of this type bandwidth-limited control.

Returning to specifics, suppose, for example, that the user would prefer a piecewise linear control history, rather than the default ZOH. In that case, rewrite (1) as

$$\dot{x} = f(x, \gamma, p) \tag{12}$$

and define a state to propagate the control variable $\gamma(t)$:

$$\dot{\gamma} = u \tag{13}$$

and append it to the plant state x_{plant}

$$x = \begin{bmatrix} x \\ \gamma \end{bmatrix} \tag{14}$$

so that the value of piecewise constant u on the j^{th} discretization interval is the slope of γ over that interval. If a discontinuous first-order hold (FOH) control variation is preferred, define a state to model time variation inside the discretization interval:

$$\dot{x}_t = 1, \quad x_t(0) = 0 \tag{15}$$

and define the interval time as

$$t_j = x_t - \texttt{j}/\texttt{nd} \tag{16}$$

where both `j` and `nd` are passed to the user's plant dynamics subroutine through the calling argument. The FOH control, then, is

$$\gamma_j(t_j) = c_{0j} + c_{1j} t_j \tag{17}$$

where c_{0j}, c_{1j} appear in the problem formulation as elements appended to the vector u_{plant} that contains whatever controls, if any, are modelled as ZOH:

$$x_j = \begin{bmatrix} x_{\text{plant}} \\ x_t \end{bmatrix}, \quad u_j = \begin{bmatrix} u_{\text{plant}} \\ c_{0j} \\ c_{1j} \end{bmatrix} \tag{18}$$

If a high-order control variation with continous slope is desired, it can be simply implemented by constructing a Hermite-type spline. Again define a time state

$$\dot{x}_j = \texttt{nd}, \quad x(0) = 0, \quad t_j = x_t - \texttt{j} \tag{19}$$

so that t_j passes from 0 to 1 over each j^{th} interval. With t_j in hand, define the control function as

$$\gamma_j(t_j) = c_{0j} + c_{1j} t_j + c_{2j} t_j^2 \tag{20}$$

and require

$$\left.\begin{array}{rcccrcl} \gamma_j(1) & = & \gamma_{j+1}(0) & \rightarrow & c_{0j} + c_{1j} + c_{2j} - c_{0,j+1} & = & 0 \\ \dot{\gamma}_j(1) & = & \dot{\gamma}_{j+1} & \rightarrow & c_{1j} + 2c_{2j} - c_{1,j+1} & = & 0 \end{array}\right\} \tag{21}$$

These dynamical constraints require that c_1 and c_2 be modelled as states added to the plant state vector:

$$\begin{array}{rcl} (\dot{c}_1)_j & = & k_{1j} \\ (\dot{c}_2)_j & = & k_{2j} \end{array} \tag{22}$$

and k_{1j} and k_{2j} are appended to the control vector u_j, resulting in

$$x = \begin{bmatrix} x_{\text{plant}} \\ x_t \\ c_1 \\ c_2 \end{bmatrix}, \quad u = \begin{bmatrix} u_{\text{plant}} \\ c_0 \\ k_1 \\ k_2 \end{bmatrix} \tag{23}$$

The discussion above has been presented primarily to whet the reader's imagination for posing MADS problems, and to provide assurance that it is not difficult to set up problems that fall outside the MADS defaults.

2.2 Subroutines To Be Provided By The User

The user's problem, as expressed by (1,4,5,6) is implemented in four user-supplied subroutines. These can be divided into two groups – subroutines that operate along trajectory arcs, and subroutines that operate on boundary values. Before providing individual details of the user-supplied subroutines, we describe a SNOPT-related input common to all of them. That input is an `integer` scalar, `nstate`, which is generated by SNOPT to provide the user with signals for initialization operations – such as data initialization and memory allocation – and post-run cleanup operations such as deallocation. The three most important conditions for `nstate` are

- `nstate`=0
  normal subroutine call,

- `nstate`=1
  first call. If there are special operations to be performed on the first call, perform them now, then proceed on to the operations performed for `nstate` = 0. Note that there is only one `nstate` = 1 call to each of the user subroutines at the beginning of a MADS run.

- `nstate`=2
  final call. If the user has cleanup operations to perform, they should be done now. MADS has finished, and will not use the results of any computations performed during this call, so it would be best to simply return once user cleanup operations are complete. If the user has nothing to do for the `nstate` = 2 call in any of the four user-supplied subroutines, simply make the first executable line of each of the subroutines "`if(nstate.GE.2)return`."

There are several other values that `nstate` can take on, none of which are recognized specifically by MADS. Those values are explained in the SNOPT documentation [1]. The user is recommended not use `nstate`, but rather to perform initialization and memory operations in code units separate from those implementing (1,4,5,6)

The subroutines that operate along trajectory arcs are

`xdot`: This subroutine corresponds to (1), and provides the right-hand side of the system of first-order ordinary differential equations (ODEs) defining the trajectory for each of the `nph` trajectory phases; that is to say, MADS calls one `xdot`, and that subroutine has logic to return the state derivative for each k^{th} phase, $k = 1, \ldots,$ `nph`. The calling syntax for `xdot` is

```
subroutine xdot(nstate,kph,nk,jk,nx,x,nu,u,np,p,f)
integer,intent(in) :: nstate,kph,nk,jk,nx,nu,np
real(8),intent(in) :: x(nx),u(nu),p(np)
real(8),intent(out) :: f(nx)
```

The inputs are

`nstate`: See the discussion at the beginning of this Subsection.

`kph`: index of current trajectory phase, i.e. $1 \leq$ `kph` $\leq$ `nph`

`nk`: number of discretization intervals in this phase

`jk`: index number of current discretization interval, i.e. $1 \leq$ `jk` $\leq$ `nk`

`nx`: state dimension in current phase

`x`: `nx`-element state vector

`nu`: control dimension in current phase

`u`: current value of `nu`-element control vector

`np`: dimension of vector of static free parameter

`p`: `np`-element static free parameter vector

The output is

`f`: right-hand side of the state time derivative, for the current inputs.

`cineq`: This subroutine corresponds to (6), and provides the equality and inequality constraints that operate on the state and control trajectories in between boundary conditions. This routine differs from `xdot` in that it has access to the state at the discrete instants at the beginning and end of the current discretization interval, whereas, `xdot` is simply called with whatever state argument is provided it by the user's chosen ME, RK2, or RK4 discretization logic. This routine also differs from `xdot` in that the user needs to specify, for each element of the output vector, whether it is to be treated as an equality or inequality constraint. Recall that all inequality constraints in MADS, per (5, 6) are posed so that they are satisfied by non-negative values. The calling syntax for `cineq` is

```
 subroutine cineq(nstate,kph,nk,jk,nx,xj,xjp1,nu,u,np,p, &
&                 nc,c,iec,iecflag)
 integer,intent(in) :: nstate,kph,nk,jk,nx,nu,np,nc,iecflag
 integer,intent(out) :: iec(nc)
 real(8),intent(in) :: xj(nx),xjp1(nx),u(nu),p(np)
 real(8),intent(out) :: c(nc)
```

The inputs are

`nstate`: See the discussion at the beginning of this Subsection.

`kph`: index of current trajectory phase, i.e. $1 \leq$ `kph` $\leq$ `nph`

`nk`: number of discretization intervals in this phase

`jk`: index number of current discretization interval, i.e. $1 \leq$ `jk` $\leq$ `nk`

`nx`: state dimension in current phase

`xj`: `nx`-element state vector at the beginning of discretization interval `jk`

`xjp1`: `nx`-element state vector at the end of discretization interval `jk`

`nu`: control dimension in current phase

`u`: current value of `nu`-element control vector

`np`: dimension of vector of static free parameter

`p`: `np`-element static free parameter vector

`c`: the dimension of the constraint vector c that is to be output by `cineq`

`iecflag`: integer scalar that signals `cineq` to output the `iec` vector. The values of `iecflag` that MADS inputs are

- `0`: a normal call, in which `cineq` is to compute its constraint quantities.
- `1`: `cineq` is required to output `iec` and then exit. No other operations are wanted from `cineq` in this case. If the user does perform other operations, they will be ignored.

The outputs are

`c`: `nc`-dimensional constraint vector from (6)

`iec`: `nc`-dimensional integer vector of ones and zeros whose i^{th} component signals MADS whether $c(i)$ is an equality or inequality constraint. The values of `iec` are

`iec(i)=0`: $c(i) = 0$ is required for solution

`iec(i)=1`: $c(i) \geq 0$ is required for solutions

The next two user subroutines operate on state boundary values and the p vector to implement (4) and 5). The boundary values are available to the routines as a vector, `xbc`, that stacks the initial and terminal values, along with integer vectors `k0` and `kf` such that `k0(k)` + 1 points to the first element of the initial state for the k^{th} trajectory phase and `kf(k)` + 1 points to the first element of its terminal state, in turn. This is displayed in the following equation, in which $x_{0,k}$ refers to the first (initial boundary value) state vector in the k^{th} phase and $x_{f,k}$ refers to the last (terminal boundary value) state vector in the k^{th} phase. For a trajectory with `nph` phases, there will be `nph` state dimensions, which can be collected in an integer vector `nxv(k)`, $k = 1, \ldots,$ `nph`:

$$\texttt{xbc} = \begin{bmatrix} x_{0,1} \\ x_{f,1} \\ x_{0,2} \\ x_{f,2} \\ \vdots \\ x_{0,\texttt{nph}} \\ x_{f,\texttt{nph}} \end{bmatrix} \begin{matrix} \ldots \texttt{k0(1)} + 1 \\ \ldots \texttt{kf(1)} + 1 \\ \ldots \texttt{k0(2)} + 1 \\ \ldots \texttt{kf(2)} + 1 \\ \vdots \\ \ldots \texttt{k0(nph)} + 1 \\ \ldots \texttt{kf(nph)} + 1 \end{matrix} \qquad x_{0,k} = \begin{bmatrix} (x_{0,k})_1 \\ \vdots \\ (x_{0,k})_{\texttt{nxv(k)}} \end{bmatrix} \qquad x_{f,k} = \begin{bmatrix} (x_{\texttt{nd}_k+1,k})_1 \\ \vdots \\ (x_{\texttt{nd}_k+1,k})_{\texttt{nxv(k)}} \end{bmatrix} \tag{24}$$

The two user subroutines that operate on `xbc` and `p` are `psibc` for (5), and `phiobj` for the cost (4):

`psibc` As noted above, this subroutine implements (5), operating on the trajectory's boundary values and the `p` vector to compute boundary conditions as equality or inequality constraints, as specified by the user. The calling syntax is:

```
subroutine psibc(nstate,nph,k0,kf,nxv,nxbc,xbc,np,p, &
&                npsi,psi,iebc,iebcflag)
integer,intent(in) :: nstate,nph,k0(nph),kf(nph),nxv(nph), &
&                     nxbc,np,npsi,iebcflag
integer,intent(out) :: iebc(npsi)
real(8),intent(in) :: xbc(nxbc),p(np)
real(8),intent(out) :: psi(npsi)
```

The inputs are:

`nstate`: See the discussion at the beginning of this Subsection.

`nph`: the number of phases in the trajectory

`k0`: explained in (24)

`kf`: explained in (24)

`nxv`: `nph`-element integer containing state dimension for each phase

`nxbc`: dimension of `xbc`, that is `2*sum(nxv)`

`np`: dimension of `p` vector

`npsi`: dimension of `psi` vector – number of boundary conditions

`iebcflag` integer scalar that signals `psibc` to output the `iecbvec` vector. The values of `iebcflag` that MADS inputs are

- `0`: a normal call, in which `psibc` is to compute its constraint quantities.
- `1`: `psibc` is required to output `iecbvec` and then exit. No other operations are wanted from `psibc` in this case. If the user does perform other operations, they will be ignored.

The outputs are:

`psi:` the `npsi`-element vector of boundary condition constraints

`iebc:` `npsi`-dimensional integer vector of ones and zeros whose i^{th} component signals MADS whether $\psi(i)$ is an equality or inequality constraint. The values of `iebc` are

`iebc(i)=0:` $\psi(i) = 0$ is required for solution

`iebc(i)=1:` $\psi(i) \geq 0$ is required for solutions

`phiobj:` This routine implements (4), evaluating the cost function ϕ. The calling syntax is

```
subroutine phiobj(nstate,nph,k0,kf,nxv,nxbc,xbc,np,p,phi)
integer,intent(in) :: nstate,nph,k0(nph),kf(nph),nxv(nph), &
&                     nxbc,np
real(8),intent(in) :: xbc(nxbc),p(np)
real(8),intent(out) :: phi
```

The, by now, familiar inputs are:

`nstate:` See the discussion at the beginning of this Subsection.

`nph:` the number of phases in the trajectory

`k0:` explained in (24)

`kf:` explained in (24)

`nxv:` `nph`-element integer containing state dimension for each phase

`nxbc:` dimension of `xbc`, that is `2*sum(nxv)`

`np:` dimension of `p` vector

and the subroutine outputs the cost:

`phi:` the scalar cost function

2.3 Producing and Operating On A MADS Solution

This Subsection describes the organization of data in a MADS solution and describes software for operating on that data. The software is a mix of FORTRAN95 for the actual optimization calculations, and Matlab functions for data manipulation. This Subsection does not go into detail in formulating and coding trajectory optimization problems; that is provided via solved examples in Section 2.
The steps in producing a MADS solution are as follows:

1. Formulate a reasonable problem.
2. Code the four user-supplied subroutines, per Subsection 1.2.
3. Code routines to provide the derivatives of the four subroutines referred to in Step 2. This is to be done using autodifferentiation software; automatic, widely available, reliable, and (currently) free for noncommercial use.
4. Assemble an initial guess.
5. Run MADS.

6. Review the solution. Accept it, or modify the problem formulation and go to Step 2.

Steps 1 and 2 are up to the user, with help and insight from this Tutorial. Step 3 can be accomplished using scripts provided with the MADS package, provided that the user installs TAPANADE [2]. This Section will also provide generic comments on autodifferentiation for MADS, should the user prefer to use a different package. Before discussing autodifferentiation, though, we will concentrate on Steps 4 through 6.

2.3.1 Formats for MADS Solution Data

We first consider the organization of data in MADS, and software for manipulating it. The problem variables for optimization are stacked together in a vector:

$$v_{MADS} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{\texttt{nph}} \\ p \end{bmatrix} \quad v_k = \begin{bmatrix} x_{1,k} \\ u_{1,k} \\ \vdots \\ x_{\texttt{nd}_k,k} \\ u_{\texttt{nd}_k,k} \\ x_{\texttt{nd}_k+1,k} \end{bmatrix} \tag{25}$$

In order for MADS to operate on v_{MADS}, it requires dimensional and program option data. This is supplied by a text file with integers as follows:

$$\begin{array}{lllll} \texttt{nph}, & \texttt{npsi}, & \texttt{np} & & \\ \texttt{nxv(1)}, & \texttt{nuv(1)}, & \texttt{ncv(1)}, & \texttt{ndv(1)}, & \texttt{kodev(1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \texttt{nxv(nph)}, & \texttt{nuv(nph)}, & \texttt{ncv(nph)}, & \texttt{ndv(nph)}, & \texttt{kodev(nph)} \end{array} \tag{26}$$

This will be referred to in the sequel as a "`premads`" file.

Because (25) and (26) comprise a miserably inconvenient format for operating on or visualizing a MADS solution, utility functions are provided for converting between v_{MADS}/`premads` and a Matlab structure – call it "S" – with the following fields:

`S.nph`: number of phases

`S.np`: dimension of `p`

`S.nxv`: `nph`-element array of state dimensions

`S.nuv`: `nph`-element array of control dimensions

`S.ndv`: `nph`-element array whose k^{th} element is the number of discretization intervals in the k^{th} phase.

`S.x`: `nph`-element cell array containing whose k^{th} element contains that phase's state trajectory.. Each `x{k}` is

$$\texttt{x\{k\}} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,\texttt{nxv(k)}} \\ x_{2,1} & x_{2,2} & & x_{2,\texttt{nxv(k)}} \\ \vdots & \vdots & \ddots & \vdots \\ x_{\texttt{(ndv(k)+1)},1} & x_{\texttt{(ndv(k)+1)},2} & \cdots & x_{\texttt{(ndv(k)+1)},\texttt{nxv(k)}} \end{bmatrix} \tag{27}$$

`S.tx`: `nph`-element cell array containing time (independent variable) instants for plotting the trajectory in `S.x`. These assume the use of (2) for time scaling, and that the first `nph` elements of the `p` vector are used for this purpose. The organization of `S.tx{k}` is

$$\texttt{S.tx\{k\}} = \begin{bmatrix} t_0(k) \\ t_0(k) + \texttt{p}(k)/\texttt{ndv(k)} \\ \vdots \\ t_0(k) + \texttt{p}(k) \end{bmatrix}$$
$$t_0(k) = \begin{cases} 0, & k = 1 \\ t_0(k-1) + \texttt{p}(k-1), & k > 1 \end{cases} \tag{28}$$

`S.u`: `nph`-element cell array whose k^{th} element contains that phase's control trajectory. Similarly to `S.x`, `S.uk` has the structure

$$\texttt{u\{k\}} = \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,\texttt{nuv(k)}} \\ u_{2,1} & u_{2,2} & & u_{2,\texttt{nuv(k)}} \\ \vdots & \vdots & \ddots & \vdots \\ u_{\texttt{ndv(k)},1} & u_{\texttt{ndv(k)},2} & \cdots & u_{\texttt{ndv(k)},\texttt{nuv(k)}} \end{bmatrix} \tag{29}$$

Note that there are `ndv(k)` instants in this array, rather than `ndv(k)+1`.

`S.tu`: `nph`-element cell array containing time instants for plotting the control trajectory in `S.u`. Again, the assumption is made that (2) is used; but the control instants are indexed to the midpoints of the discretization intervals:

$$\texttt{S.tu\{k\}} = \begin{bmatrix} t_0(k) + \frac{1}{2}\texttt{p(k)}/\texttt{ndv(k)} \\ t_0(k) + (1 + \frac{1}{2})\texttt{p(k)}/\texttt{ndv(k)} \\ \vdots \\ t_0(k) + (\texttt{ndv(k)} - \frac{1}{2})\texttt{p(k)}/\texttt{ndv(k)} \end{bmatrix} \tag{30}$$

and $t_0(k)$ is defined in (28).

A nice feature of this structure is, of course, that the user can add additional fields to S.

2.3.2 Matlab Functions for Operating On MADS Data

The MADS package includes several Matlab functions for

- importing of v_{MADS} data into the Matlab environment, and supporting its visualization,
- modifying MADS solution data for a given problem formulation, in support of constructing an initial guess for an alternate, but related, problem formulation,
- exporting S-format data to MADS input data.

The functions that import MADS data and support visualization are, primarily, `plotmadsMADS`, and `dtplotMADS`. Two additional functions, `getMADS`, `readpremadsMADS` are included to provide the user with a little more flexibility in programming style. The functions are

`importMADS`: This is the main import function for MADS, and is called as

```
S=importMADS(premads,fdata,flag)
```

The inputs are:

`premads`: character string, the name of the `premads` file for the MADS data to be imported.

`fdata`: another character string, this the name of the MADS output data file, whose data is in v_{MADS} format.

`flag`: scalar flag to indicate whether or not the lagrange multipliers computed by SNOPT for the MADS problem should also be imported. The admissible values are

0 . . . don't import

1 . . . do

The `importMADS` output is S, which contains all of the data in Subsection 1.3.1 and, additionally,

`S.up`: `nph`-element cell array containing the control trajectory, formatted to plot as a sequence of zero-order hold values.

`S.tup`: `nph`-element cell array containing the corresponding time values, again, assuming that the first `S.np` element of the `S.p` vector are used in (2) for time scaling.

`S.lamx`: cell array containing lagrange multiplier histories for the discretization constraints, output if `flag=1`.

`S.lama`: cell array containing lagrange multiplier histories for the constraints in (6), output if `flag=1`. `lamak=[]` for those phases where there are no (6) constraints.

`S.lampsi`: lagrange multiplier vector for (5), output if `flag=1`. Note that `lampsi=[]` if there are no boundary conditions.

It should also be noted that, if `S.np` $= 0$, then `S.p` is returned as `nan`.

`dtplotMADS`: This function produces time vectors for use in plotting S state and control trajectories for the case where (3) is used for variable time steps. It is called as

```
[tu,ux]=dtplotMADS(dtin)
```

with input

`dtin`: `nph`-element cell array, the k^{th} element of which contains the vector of u_τ from the MADS solution for that phase.

The outputs are

`tu`: cell array containing the time values for plotting `S.u`.

`tx`: cell array containing the time values for plotting `S.x`.

The user who has a solution with variable time steps should get the solution with `importMADS`, then discard that `S.tx`, `S.tu`, and run `[S.tu,S.tx]=dtplotMADS(dtin)`, having pulled `dtin` together from the S.u data.

`getMADS`: This function, called as

```
S=getMADS(fdata,nxv,nuv,ndv,np)
```

outputs an S data structure containing only `x`, `u`, `tx`, `tu`, `p`, `nxv`, `nuv`, `ndv`, and `np`.

`readpreMADS`: This function, called as

```
S=readpreMADS(name)
```

reads a `premads` file with name `name` and outputs a partially populated S structure, which contains the dimensional and discretization option information contained in the `premads` file.

The Matlab functions that support modifying and exporting MADS data are `adduxMADS`, `breaktrajMADS`, `dtaunphMADS`, and `writepreMADS`.

`adduxMADS`: This function, called as

```
vout=adduxMADS(x,u,nxout,nuout,randmag,(optional) outind)
```

pads or removes columns from S-format `x` and `u`, and collects them into a v_{MADS}-format vector. The states and control may optionally be reordered before concatenating into v_{MADS}. The inputs are

- `x`: S-format state trajectory `x(nd+1,nx)`
- `u`: S-format control trajectory `u(nd,nu)`. This may be empty if `nu` $= 0$.
- `nxout`: desired output state dimension. If `nxout` $>$ `nx`, then the additional states are indexed as `x(nx` $+ 1) \ldots$ `x(nxout)`.
- `nuout`: desired output control dimension. Padding for `u` follows the same pattern as with `x`.
- `randmag`: If `nxout` $>$ `nx` or `nuout` $>$ `nu`, the padded extra states or controls are populated with `r=randmag*rand`, where `rand` is the Matlab uniform random number generator.
- `outind`: (optional argument) structure containing desired ordering of states or controls in the output. The `x` indices are in `outind.xind` and the `u` indices are in `outind.uind`. If either `xind` or `uind` are not to rearrange state or control indices, they may be set empty; otherwise, the dimension of each must be `nxout` and `nuout`, respectively. As a concrete example, suppose that `nx` $= 3$, `nxout` $= 5$ and `outind.xind=[4 1:3 5]`. In this case, the S-format organization of each row of the expanded state is $\mathtt{x}_k = [\mathtt{x(4)}\ \mathtt{x(1)}_k\ \mathtt{x(2)}_k\ \mathtt{x(3)}_k\ \mathtt{x(5)}],\ k = 1, \ldots,$ `nd`.

The output is

- `vout`: `x` and `u` in v_{MADS} format.

Note that, for multiphase problems, `adduxMADS` is simply called for each phase, and the output vectors are stacked to make the full v_{MADS}. For example,

```
randmag=0;
v=[];
for k=1:n
  v=[v;adduxMADS(S.x{k},S.u{k},nxout(k),nuout(k),randmag)];
end
v=[v;S.p];
```

`breaktrajMADS`: This function is used to break a single `xk`, `uk` phase into subphases and, optionally, to resample one or more of the subphases with a different number of discretization intervals. The function also assumes that the time scaling approach of (2) is used, and produces a vector of phase durations for the output subphases. The function is called as

```
bout=breaktrajMADS(x,u,bv,tau,(optional) ndin)
```

The inputs are

- `x`: S-format state trajectory `x(nd+1,nx)`
- `u`: S-format control trajectory `u(nd,nu)`. This may be empty if `nu` $= 0$.
- `bv`: vector of breakpoints – time-wise indices comprising the initial points of the output subphases. Note, `bv` does not include the inital point of the input single-phase trajectory, i.e. `bv` $\neq 1$. The breakpoints `bv` are defined in terms of the state discretization mesh points, with `bv` $= k$ breaking the trajectory at mesh point k. Suppose, for simplicity, we have `x` scalar, `nph=1`, `ndv=20`, and `bv=[3]`. This breaks the trajectory into two output phases, with `ndv=[2 18]`, and $\mathtt{x}_{bout}(\mathtt{kf}(1)+1) = \mathtt{x}(3)$ and $\mathtt{x}_{bout}(\mathtt{k0}(2)+1) = \mathtt{x}(3)$, where $\mathtt{x}_{bout}$ is informal notation for the multiphase output of `breaktrajMADS`. This, and the fact that controls are defined on the interval between the k^{th} and $(k+1)^{th}$ instants means that if the user uses the control trajectory to choose a breakpoint, the chosen control instant will appear in the phase to the right of the breakpoint. Note that it is admissible to input `bv=[]`. This would correspond to the case of resampling a single trajectory phase. In this case, `ndin`, detailed below, would have a single element.
- `tau`: scalar – the duration of the input phase.
- `ndin`: This is an optional input, but must be present if `bv=[]`. `ndin` is the number of integration intervals desired for each of the output subphases. The resampling is done using piecewise linear interpolation of input state and control trajectories.

and the output is

- `bout`: structure contining the split trajectory and associated information:
  - `ndv`: `nbv`-element array containing the number of discretization intervals in each subphase.
  - `tau`: `nbv`-element array containing the duration of each subphase, based on the input value of `tau`.
  - `x`: `nbv`-element cell array containing state subphases. Assuming that `ndin` is not input, we have (referring to `bout.x` as `x` and `bout.nph` as `nph`,

$$\mathtt{x}_{in} = \begin{bmatrix} \mathtt{x}_1 \\ \vdots \\ \mathtt{x}_{nd+1} \end{bmatrix} \rightarrow \begin{cases} \mathtt{x\{1\}} = \begin{bmatrix} \mathtt{x}_1 \\ \vdots \\ \mathtt{x}_{\mathtt{bv(1)}} \end{bmatrix} \\ \mathtt{x\{k\}} = \begin{bmatrix} \mathtt{x}_{\mathtt{bv(k-1)}} \\ \vdots \\ \mathtt{x}_{\mathtt{bv(k)}} \end{bmatrix}, \quad k = 2, \ldots, \mathtt{nph}-1 \\ \mathtt{x\{nph\}} = \begin{bmatrix} x_{\mathtt{bv(nph)-1}} \\ \vdots \\ \mathtt{x}_{\mathtt{nd+1}} \end{bmatrix} \end{cases} \tag{31}$$

`u:` `nbv`-element cell array of control subphases. Here,

$$\mathtt{u}_{in} = \begin{bmatrix} \mathtt{u}_1 \\ \vdots \\ \mathtt{u}_{nd} \end{bmatrix} \rightarrow \begin{cases} \mathtt{u\{1\}} = \begin{bmatrix} \mathtt{u}_1 \\ \vdots \\ \mathtt{u}_{\mathtt{bv(1)}} \end{bmatrix} \\ \mathtt{u\{k\}} = \begin{bmatrix} \mathtt{u}_{\mathtt{bv(k-1)}} \\ \vdots \\ \mathtt{u}_{\mathtt{bv(k)}} \end{bmatrix}, \quad k = 2, \ldots, \mathtt{nph}-1 \\ \mathtt{u\{nph\}} = \begin{bmatrix} u_{\mathtt{bv(nph)-1}} \\ \vdots \\ \mathtt{u}_{\mathtt{nd}} \end{bmatrix} \end{cases} \tag{32}$$

`dtaunphMADS:` This function, called as

```
dtout=dtaunphMADS(bv,dtin)
```

breaks the u_τ trajectory from (3) into multiple subphases. It needs to be used in order to correctly scale the subphases' u_τs. The inputs are

`bv:` vector of breakpoints, identical to that for `breaktrajMADS`, above.

`dtin:` vector of u_τ values taken from a MADS solution.

and the output is

`dtout:` cell array of with `length(bv)+1` elements. Each k^{th} cell contains the u_τ trajectory for the subphase of `dtin` that began with its element `bv(k)`.

`writepreMADS:` This function is the twin of `readpreMADS`. It inputs an S structure and writes the corresponding `premads` file onto a file with name `name`. It is called as

```
writepreMADS(S,name)
```

Function `writepreMADS` opens and closes file "`name`."

There are several other MADS support functions with very specialized applicability, and they will be described in their contexts. Before leaving this topic, though, there is one additional function to describe.

`findIMADS:` This function can be used to assist in debugging MADS problems. The function is called as

```
theI=findIMADS(S,row)
```

with S structure `S` and "`row`" as inputs. This function is used to help diagnose misbehavior in MADS solutions. The diagnostic output from SNOPT includes a check of the correctness of the overall problem jacobian at the beginning of the run, comparing the user-supplied analytic jacobian to one using numerical differentiation. If there's an error at a particular constraint element – "`row`" the comment "`bad`" is appended to the right of the output for that row. At the end of the run, a summary of the final constraint activity is provided, labelled "`Section 1 - Rows`." If there is an infeasibility for a particular constraint (`row`), it will be flagged with an "`I`" in the column labelled "`State`." To use `findIMADS`, the user supplies dimensional data in S, gotten either by using `importMADS` or `readpreMADS`, and the offending `row` number. The output is

`theI:` structure containing

- `kph:` the phase number. `theI.kph=[]` if the constraint is in `psibc`.
- `jk:` the discretization interval. `theI.jk=[]` if the constraint is in `psibc`.
- `xcp:` the "type" of the constraint: 'c' if in `cineq`, 'x' if in `xdot`, or 'p' if in `psibc`.
- `ind:` the position of the constraint in the in the user-supplied routine's output. For example, if `theI.xcp='p'` and `theI.ind=2`, the problem is with `psi(2)`.

We hope that the user will never want to use this function, but we know better.

2.4 Setting Up and Executing a MADS Run

Thus far, we have described the general MADS problem formulation, the user-supplied subroutines for realizing a given problem in MADS, and utilitiies for operating on MADS' input and output data. We now turn to the mechanics of actually making a MADS run.

There are three principal considerations to be kept in mind when setting up a problem for MADS:

1. Because the problem formulation resides in the four separate subroutines of Subsection 1.2, it is generally in the user's best interest to treat those data that are common to two or more problem routines as global variables. We recommend use of the FORTRAN "MODULE" program unit as a means of safely sharing data across multiple local program units. Consider the following code fragment:

```
module myprobMOD
implicit none
integer, parameter ::        &
& UsedByCineq = 1,           &
& UsedByAll = UsedByCineq + 2
real(8) ::                   &
& xdotParam
end module myprobMOD

subroutine cineq(nstate, kph, nk, jk, nx, xj, xjp1, &
&            nu, u, np, p, nc, c, iec, iecflag)
use myprobMOD, only : UsedByCineq, UsedByAll
implicit none
        ⋮

subroutine xdot(nstate, kph, nk, jk, nx, x, nu, u, np, p, f)
use myprobMOD, only : UsedByAll, xdotParam
implicit none
        ⋮

program myprob
use myprobMOD, only xdotParam
implicit none
        ⋮
xdotParam = some number
        ⋮
```

2. Typically, for a given physical plant model, several different trajectory optimziation problems may be posed for it. These may simply be different missions that the plant is being called upon to perform, or it may be different formulations of superficially similar optimization problems that the user explores while seeking results that are not only optimal but desirable. Each of the examples in the Tutorial Section 2 will contain examples of this type of exploration. Because of this, it is highly desirable that the user strictly separate plant model software from problem formulation software. For example, suppose that a problem is posed with free terminal time using the (2) formulation. In that case, `xdot` should look like

```
module FreeTimeMOD
        ⋮
integer, parameter ::          &
& loctau = 1        ! location of terminal time in p
end module FreeTimeMOD

subroutine xdot(nstate, kph, nk, jk, nx, x, nu, u, np, p, f)
use FreeTimeMOD, only : loctau
        ⋮
!    MyPlant is a separate subroutine for plant ODEs
call MyPlant(x, u, f)
f = p(loctau) * f  ! Here is the time − scaling
end subroutine xdot
```

3. The instantiation of a given MADS problem will typically involve some 15-16 unique files, not counting those pertaining to the plant model. These include "plain" and autodifferentiated `xdot`, `cineq`, `psibc` and `phiobj` routines, main and MODULE program units, input and output data, and scripts for visualization and for assembling the initial guess. Experience teaches that, unless the problem-unique files for each MADS problem are kept separate from those of other MADS problems, confusion will ensue. The authors strongly recommend devoting a directory or "folder" to each such problem and, further, to affix a common character string to each of the names of each of the files that associates them with their particular problem. This practice will be illustrated in the Tutorial.

MADS execution is performed by the subroutine `batchMADS`, calling SNOPT. This subroutine is called by a main routine, and needs to be linked to the MADS subroutine library, the problem subroutines `cineq`, `xdot`, `psibc`, `phiobj`, their derivatives `cineq_dv`, `xdot_dv`, `psibc_dv`, `phiobj_dv`, and any model-specific subroutines. Generation of the derivative routines' source code is described in the next Subsection, and the syntax of `batchMADS` is given below:

`batchMADS`: This subroutine,

```
subroutine batchMADS(findata,fpremads,foutdata,inform)
integer,intent(in) :: findata,fpremads,foutdata
integer,intent(out):: inform
```

has the following arguments:

`findata`: FORTRAN "unit number" for an opened file containing the input guess.

`fpremads`: Unit number for an opened file containing the `premads` file for the given problem.

`foutdata`: Unit number for an opened file into which batchMADS will write the solution data at the conclusion of `batchMADS` execution. This file contains a concatenation of the output value of v_{MADS} from (25) and the SNOPT-computed lagrange multipliers for the constraints in the Z vector, defined in (8). Recall that extracting and organizing the lagrange multipliers from MADS output is handled by `importMADS` as an option. If SNOPT fails to converge, the output data in `foutdata` corresponds to the state of the iterations at the end of the run.

`info`: This is a `batchMADS` output, and is a pass-through from SNOPT, in whose documentation [1] `info` is fully documented. The most typical values that will be encountered by the MADS user are

- `1`: finished successfully
- `3`: couldn't achieve desired accuracy
- `13`: infeasible constraint(s)
- `41`: current point cannot be improved
- `52`: incorrect constraint derivatives

In `batchMADS`, several SNOPT parameters are set. These are `Infinite Bound` set to 10^{20}, `Verify Level`, set to 3, `Derivative Level`, set to 3, and `Linesearch Tolerance`, set to 0.99. These values can be overridden, or other SNOPT options set by writing and linking a subroutine `userset`:

```
subroutine userset(fprt,fsumm,info,cw,lcw,iw,liw,rw,lrw)
integer,intent(in) :: fprt,fsumm,lcw,liw,lrw
integer,intent(out) :: info,iw(liw)
real(8),intent(out) :: rw(lrw)
character(8),intent(out) cw(lcw)
```

In the body of this subroutine, the user calls SNOPT routines `snseti`, or `snsetr` to set parameters as preferred. The user is not required to provide a `userset`, as the MADS library includes a dummy for linking.

2.4.1 Autodifferentiation for MADS

This Subsection provides a short discussion of algorithmic differentiation (AD) in MADS. Algorithmic, or "automatic" differentiation [4] is the use of software techniques to numerically evaluate the derivative of a function calculated by a computer code. This is – very superficially speaking – done by analytically differentiating the most primitive computations in the code, e.g., exponentiation, transcendental functions, and so on, and building up the full evaluation of the derivative by repeatedly applying the chain rule. This technique has a substantial advantage over the use of finite differences in that the computed derivative is accurate to machine precision, with no degradation due to the truncation that is associated with difference-based differentiation.

MADS requires the derivatives of f (1), c (6), ψ (5), and ϕ (4). If the user has TAPANADE [2] installed, the subroutines that provide these derivatives – `xdot_dv`, `cineq_dv`, `psibc_dv`, `phiobj_dv` – are simply obtained by executing the following Matlab script invocations from the command line:

```
tapxdot('routines called by xdot, MODULES referenced by xdot')
tapcineq('routines called by cineq, MODULES referenced by cineq')
tappsibc('routines called by psibc, MODULES referenced by psibc')
```

```
tapphiobj('routines called by phiobj, MODULES referenced by phiobj')
tapscript
```

and then the user need not think any further about AD. In the code fragments above, incidentally, the arguments to each of the `tapxxx` functions are to be input as string variables. For example, if `xdot` calls `MyPlant.f90` and references variables from `MyProbMOD.f90`, then the detailed call to `tapxdot` would be

```
tapxdot('MyProbMOD.f90 MyPlant.f90')
```

The order of files in the argument does not matter. Again, TAPANADE users can skip the rest of this section.

If the user wishes to use a different AD package to generate the `XXX_dv` subroutines, it will need to be able to provide "multidimensional" differentiation, and to do so in "forward," or synonymously, "linear tangent" mode. The first of these options will assure that the routines are differentiated over the entire span of their arguments: For a code that computes $g(x)$, $x \in \mathcal{R}^n$, rather than generating a `_dv` subroutine to compute $v^T g_x$, it will compute

$$(g_x)_{jk} = \frac{\partial g_k}{\partial x_j} \tag{33}$$

Regarding "forward" versus the alternative "adjoint" or, synonymously, "backward" modes of differentiation, "forward" is a differentiation mode that follows the order of execution in the subroutine(s) being differentiated. This mode produces subroutines with the interface syntax

```
subroutine xdot_dv(nstate,kph,nk,jk,nx,x,xd,nu,u,ud,np,p,pd,f,fd,nb)
integer,intent(in) :: nstate,kph,nk,jk,nx,nu,np,nb
real(8),intent(in) :: x(nx),xd(nb,nx),u(nu),ud(nb,nu),p(np),pd(nb,np)
real(8),intent(out) :: f(nx),fd(nb,nx)
! Note: nb=nx+nu+np

subroutine cineq_dv(nstate,kph,nk,jk,nx,xj,xjd,xjp1,xjp1d,      &
&                   nu,u,ud,np,p,pd,nc,c,cd,iec,iecflag,nb)
integer,intent(in) :: nstate,kph,nk,jk,nx,nu,np,nc,iecflag,nb
integer,intent(out) :: iec(nc)
real(8),intent(in) :: xj(nx),xjd(nb,nx),xjp1(nx),xjp1d(nb,nx),  &
&                     u(nu),ud(nb,nu),p(np),pd(nb,np)
real(8),intent(out) :: c(nc),cd(nb,nc)
! Note: nb=2*nx+nu+np

subroutine psibc_dv(nstate,nph,k0,kf,nxv,nxbc,xbc,xbcd,np,p,pd, &
&                   npsi,psi,psid,iebc,iebcflag,nb)
integer,intent(in) :: nstate,nph,k0(nph),kf(nph),nxv(nph),np,   &
&                     npsi,iebcflag,nb
integer,intent(out) :: iebc(npsi)
real(8),intent(in) :: xbc(nxbc),xbcd(nb,nxbc),p(np),pd(nb,np)
real(8),intent(out) :: psi(npsi),psid(nb,npsi)
! Note: nb=nxbc+np

subroutine phiobj_dv(nstate,nph,k0,kf,nxv,nxbc,xbc,xbcd,np,p,pd, &
&                    phi,phid,nb)
integer,intent(in) :: nstate,nph,k0(nph),kf(nph),nxv(nph),np,nb
real(8),intent(in) :: xbc(nxbc),xbcd(nb,nxbc),p(np),pd(nb,np)
real(8),intent(out) :: phi,phid(nb)
! Note: nb=nxbc+np
```

The _dv subroutines above are autodifferentiated with the following lists of dependent and independent variables:

	dependent	independent
xdot :	f	x u p
cineq :	c	xj xjp1 u p
psibc :	psi	xbc p
phiobj :	phi	xbc p

TAPANADE, and other AD programs within the author's experience, concatenates lists of multiple independent variables, conceptually, into a column vector "b":

$$\begin{array}{rll} \texttt{xdot}: & \texttt{b} = [\texttt{x}^T\ \texttt{u}^T\ \texttt{p}], & \texttt{nb} = \texttt{nx} + \texttt{nu} + \texttt{np} \\ \texttt{cineq}: & \texttt{b} = [\texttt{xj}^T\ \texttt{xjp1}^T\ \texttt{u}^T\ \texttt{p}^T], & \texttt{nb} = 2\texttt{nx} + \texttt{nu} + \texttt{np} \\ \texttt{psibc}: & \texttt{b} = [\texttt{xbc}^T\ \texttt{p}], & \texttt{nb} = \texttt{nxbc} + \texttt{np} \\ \texttt{phiobj}: & \texttt{b} = [\texttt{xbc}^T\ \texttt{p}], & \texttt{nb} = \texttt{nxbc} + \texttt{np} \end{array} \tag{34}$$

The "d"-suffix variables in the subroutine input arguments, e.g., xd, ud, pd in the case of xdot_dv, are column partitions of $\partial \texttt{b}/\partial \texttt{b}^T = I$. Note the following:

1. The orders of variables in (34) is not important. What is illustrated here is merely the order in which the variables are stacked in the MADS code.

2. It is **critical** that the AD user autodifferentiates using the entire list of independent variables at once. For example, if a user autodifferentiates xdot first with dependent variable x, then with u, then with p, he or she might get a differentiated subroutine whose argument list looks superficially similar to that displayed above; but attempting its use would almost certainly crash the program's execution or, worse yet, only provide erroneous results.

3. It is not unusual for an AD code to presume that the dimensionality of b is actually larger than that declared via the list of dependent variables declared to the AD software. In order to accomodate this presumption, they do things such as introducing a global variable into the _dv code that contains the "real" row dimension of the d-suffix variables. For example, TAPANADE inserts the line USE DIFFSIZES into the subroutine header, where DIFFSIZES is to be a user-supplied MODULE file containing a declaration of the variable NBDIRSMAX. This NBDIRSMAX is then used as the row dimension for all of the d-suffix variables and certain intermediate variables. This behavior is unsuitable for MADS. The row dimensions of these variables should be nb, not NBDIRSMAX. This, and a few other annoyances are corrected for TAPANADE-based MADS users by executing the script tapscript.

4. Some AD packages, by default, second-guess the user's declaration of dependent variables and fail to provide a d-suffix arguments in the _dv routine's argument list for a given dependent variable if the declared dependency doesn't actually exist for the given subroutine. For example, suppose that xdot didn't actually include any element of the p vector in its computation of f. In this case, the AD output would have first line

```
subroutine xdot_dv(nstate,kph,nk,jk,nx,x,xd,nu,u,ud, &
&                  np,p,f,fd,nb)
```

   This example is missing "pd," and attempting its use in a MADS execution would result in, at best, a program crash, since MADS requires a particular, fixed, interface

to its `_dv` subroutines. The user should be wary about this. This `can` happen; for example, it is TAPANADE's default behavior, though that can be corrected by using the command line option "`-fixinterface`."

3 Tutorial Examples

This section is a tutorial for the use of MADS in solving numerical optimal control problems. The exposition is cast in the form of a sequence of example problems. This sequence progresses from very simple to fairly complicated, and each problem in the sequence features one or more tricks that are generically useful in other problems that the user may face. Two examples are considered:

1. **Linear System Minimum Time to Origin**
   The optimal solution of this classic problem, described in Chapter 3.9 of [5] in continuous time is a "bang-bang" control trajectory in which the control jumps discontinously between maximum and minimum constraint limits. Because MADS employs a fixed time step in its discretization, it will be seen that the solution to the most straightforward MADS formulation differs from the continuous time solution by having an "excrescence" in its control trajectory – a single time interval in which the control takes on an intermediate value. Two approaches for capturing the structure of the continuous time problem are described:

   - Break the problem into two phases.
   - Introduce variable discretization step size.

   Alternatively, recognizing that true "bang-bang" control trajectories are not physically realizable, we also demonstrate a technique that exploits the structure of the midpoint euler discretization to limit the bandwidth of the control solution by penalizing or constraining its rate and acceleration. An alternative to this technique would be to introoduce a low-pass filter for the control into the dynamics. In optimizing the trajectory with such a filter in place, however, there will be a tendency for the control solution to attempt to cancel out the filter dynamics. The approach given here has the advantage of operating directly on the control.

2. **Goddard Problem**
   In the Goddard problem [8], the goal is to maximize the final altitude of a sounding rocket's ascent. In the problem treated here, it flies through an exponentially decaying atmosphere in an inverse-square gravitational field, and is subject to an inequality constraint on dynamic pressure.

   This problem is distinguished by having a "singular arc" as described in Chapter 8 of [5]. A singular arc, in a variational optimal control problem, is a finite portion of the optimal trajectory in which the control necessary condition for optimality vanishes identically; in other words, to first order, the optimal performance of the system is oblivious to the value of the control while it traverses the singular arc. It is straightforward to obtain converged solutions to such problems using MADS, but this insensitivity of performance to the control along singular arcs can affect the quality of the converged solution in two ways:

   (a) The user is likely to get a messy-looking control trajectory where there is no numerically unambiguous optimum.

   (b) The ambiguity in the control solution may also imply that the state trajectory drifts some from the exact optimum.

In this example, a straightforward direct solution for the case without an active dynamic pressure constraint displays the "messy-looking" control trajectory warned of above, and a penalty function is applied to the jitter. The example concludes with imposition of a an active dynamic pressure inequality constraint.

3.1 Linear System Minimum Time to Origin

3.1.1 Baseline Problem

In this classic problem [5], a system of the form

$$\ddot{x} = u \qquad u \in [-1, 1] \tag{35}$$

is driven from

$$x(0) = a \qquad \dot{x}(0) = b \tag{36}$$

to the origin in minimum time, τ^*, resulting in an optimal trajectory in which the control, $u(t)$, rides the saturation boundaries cited in (35), switching a maximum of one time from -1 to 1 or vice versa, depending on initial conditions. The problem is expressed for MADS by converting the expression $\ddot{x}$ to first order ODEs:

$$\dot{x}_1 = x_2 \qquad \dot{x}_2 = u \tag{37}$$

and time-scaling the the equations of motion, per (2), as

$$\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \tau \begin{bmatrix} x_2 \\ u \end{bmatrix} \tag{38}$$

The cost function is

$$\phi = \tau \tag{39}$$

the control inequality constraints are

$$c(x, u) = \begin{bmatrix} 1 - u \\ 1 + u \end{bmatrix} \geq 0 \tag{40}$$

and the boundary conditions are

$$\psi = \begin{bmatrix} x_1(0) - a \\ x_2(0) - b \\ x_1(1) \\ x_2(1) \\ \tau - \epsilon_\tau \end{bmatrix} \qquad \texttt{iebc} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{41}$$

In (41), recall that boundary conditions expressed with $\texttt{iebc} = 0$ are treated as equality constraints. In the case where $\texttt{iebc} = 1$, above, the constraint is nonnegative:

$$\tau - \epsilon_\tau \geq 0$$

where ϵ_τ is a user-selected small positive number. A constraint of this kind is inexpensive to include, and can be quite valuable, since duration parameters in collocation-based trajectory optimization computations – at least in our experience – frequently exhibit unproductive behavior during the iterative search for a solution, taking on zero or negative values unless restrained. The parameter τ is placed in the `pv` partition of the vector of free variables:

$$\texttt{vin} = [\bar{x}_0,\ u_0,\ \bar{x}_1,\ u_1,\ \cdots\ \bar{x}_{\texttt{nd}},\ u_{\texttt{nd}},\ \bar{x}_{\texttt{nd}+1},\ \tau]$$

where $\bar{x}_k = [(x_1)_k,\ (x_2)_k]$ and `nd` is the number of discretization intervals selected. For this problem, we choose $\texttt{nd} = 50$.

The subroutines below are reliably parsed by the version of TAPANADE [2] available at the time of writing.

```
subroutine xdot(nstate,kph,inst,xv,nx,uv,nu,pv,np,fv)
implicit none
integer,intent(in) :: nstate,kph,inst,nx,nu,np
real(8),intent(in) :: xv(nx),uv(nu),pv(np)
real(8),intent(out) :: fv(nx)
real(8) :: x2,u,p
if(nstate.GT.1)return  ! This is the last ‘‘cleanup’’ call from SNOPT
x2=xv(2)  ! Calling out scalar variables aids in clarifying
u=uv(1)   ! complicated nonlinear expressions.  Pointers could be
p=pv(1)   ! substituted here if they were reliably supported.
fv(1)=x2
fv(2)=u
fv=fv*p
return
end subroutine xdot

subroutine cineq(nstate,kph,nx,xj,xjp1,nu,uv,np,pv,nc,cv)
implicit none
integer,intent(in) :: nstate,kph,nx,nu,np,nc
real*8,intent(in) :: xj(nx),xjp1(nx),uv(nu),pv(np)
real*8,intent(out) :: cv(nc)
if(nstate.GT.1)return
cv(1)=1+uv(1)
cv(2)=1-uv(1)
return
end subroutine cineq

subroutine psibc(nstate,nph,k0,kf,nxv,nxbc,xbc,np,pv,npsi, &
              &  psi,iebc,iebcflag)
implicit none
integer,intent(in) :: nstate,nph,k0(nph),kf(nph),nxv(nph), &
                    & nxbc,np,npsi,iebcflag
integer,intent(out) :: iebc(npsi)
real(8),intent(in) :: xbc(nxbc),pv(np)
real(8),intent(out) :: psi(npsi)
real(8) :: x10,x20,x1f,x2f,tau
if(nstate.GT.1)return
if(iebcflag.EQ.1)then  ! At the beginning of the run, tell MADS
  iebc=0               ! which boundary conditions are
  iebc(5)=1            ! equalities and which are inequalities,
  return               ! and RETURN!
endif
x10=xbc(k0(1)+1)  ! x1(0)
x20=xbc(k0(1)+2)  ! x2(0)
x1f=xbc(kf(1)+1)  ! x1(t_f)
x2f=xbc(kf(1)+2)  ! x2(t_f)
tf=pv(1)
psi(1)=1-x10
psi(2)=1-x20
psi(3)=x1f
psi(4)=x2f
psi(5)=tau-1.D-6  !terminal time is positive
```

```
return
end

subroutine phiobj(nstate,nph,k0,kf,nxv,nxbc,xbc,np,pv,phi)
implicit none
integer,intent(in) :: nstate,nph,k0(nph),kf(nph),nxv(nph),nxbc,np
real(8),intent(in) :: xbc(nxbc),pv(np)
real(8),intent(out) :: phi
if(nstate.GT.1)return
phi=pv(1)
return
end subroutine phiobj
```

and, finally, the `premads` file is

```
1,5,1
2,1,50,2,0
```

where the last element of the second line, recall, indicates that the midpoint Euler discretization is specified.

Having assembled software for the cost function and modelling constraints, and AD-processed the `xdot`, `cineq`, `psibc`, and `phiobj` routines, it remains to compute a converged solution. The first two attempts to compute a solution for this very simple, nearly linear, problem were unsuccessful. The initial guesses for `vin` were, respectively, all ones, and all random numbers generated by Matlab's `rand`() function. These attempts were both unsuccessful – SNOPT was unable to compute a feasible iterate.

The next attempt to generate an initial guess for this problem was to solve a similar, but less stringent, problem. The changes were to eliminate the state boundary conditions, moving them to the cost function. In other words,

$$\begin{aligned} \psi &= \tau - \epsilon, \quad \texttt{iebc} = 1 \\ \phi &= \tau + \alpha * \left[(x_1(0) - a)^2 + (x_2(0) - b)^2 + (x_1(1))^2 + (x_2(1))^2 \right] \end{aligned}$$

This was also an unsuccessful problem, in that it did no better in leading to a feasible solution. A successful initial guess was generated, however, by fixing τ, rather than letting it vary freely. This was done by setting setting `iebc` to zero for the `psibc` constraint $\tau - 1 = 0$, and using all ones as an initial guess.

This latter guess was passed on to the original problem formulation laid out above, and used successfully as an initial guess to obtain the time-optimal solution that satisfied the state boundary conditions. It will be seen that this pattern plays out generically in using MADS for solving OCPs. Initial guesses are most easily generated by relaxing path constraints, such as boundary conditions, instead getting a solution time history that satisfies the discretization constraints, say (9). Final time should be treated warily in the search for the initial guess.

Figure 1 displays the solution for a minimum-time trajectory from initial conditions $x(0) = 1$, $\dot{x}(0) = 1$. The plot on the right displays the actual control time history (piecewise constant) as dark red, and a continuous line drawn through the midpoints of the control increments in lighter red. As mentioned in the tutorial's introduction, the control history is not perfectly "bang-bang," having a little excrescence near the time $t = 2.25s$. Two approaches for fixing this are given below:

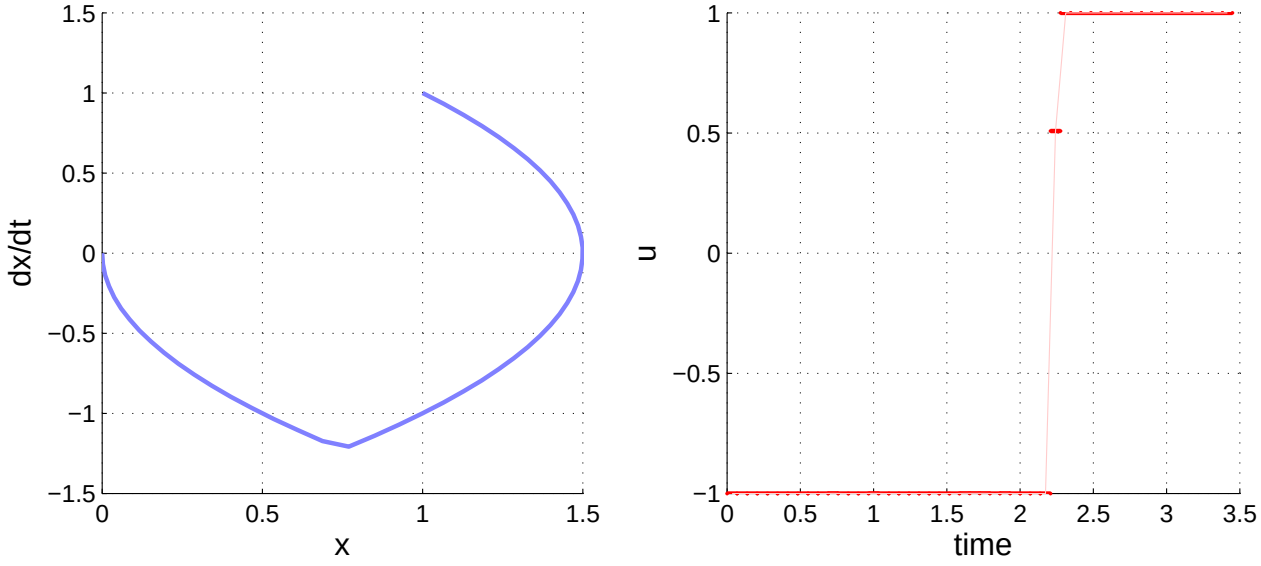

Figure 1. Two-State Bang-Bang Problem with Constant Time Step

3.1.2 Break the problem into two phases

A cleaner control discontinuity can be obtained by breaking the problem into two phases at the point at which the solution "bangs" from its maximum to its minimum value. This is done by introducing continuity boundary conditions on the states, including separate duration parameters for each phase, and setting the control to its appropriate constrained value during each phase. Denoting the states as $x_{11},\ x_{21}$ during the first phase and $x_{12},\ x_{22}$ during the second, the revised formulation is

$$\begin{array}{ll} \dot{x}_{11} = x_{21}, & \dot{x}_{21} = u \\ \dot{x}_{12} = x_{22}, & \dot{x}_{22} = u \end{array} \qquad \begin{array}{l} \text{PHASE 1} \\ \text{PHASE 2} \end{array} \qquad (\texttt{xdot}) \tag{42}$$

$$\phi = \tau_1 + \tau_2 \tag{43}$$

$$c = \left.\begin{array}{rcl} u - 1 & \geq & 0 \\ -u + 1 & \geq & 0 \end{array}\right\} \text{PHASE 1} \qquad \left.\begin{array}{rcl} u + 1 & \geq & 0 \\ -u - 1 & \geq & 0 \end{array}\right\} \text{PHASE 2} \qquad (\texttt{cineq}) \tag{44}$$

$$\psi = \begin{bmatrix} x_{11}(0) - a \\ x_{21}(0) - b \\ x_{11}(1) - x_{12}(0) \\ x_{21}(1) - x_{22}(0) \\ x_{12}(1) \\ x_{22}(1) \\ \tau_1 - \epsilon \\ \tau_2 - \epsilon \end{bmatrix} \qquad \texttt{iebc} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \qquad (\texttt{psibc}) \tag{45}$$

Note that the inequalities in (44) enforce $u = 1$ in phase 1 and $u = 1$ in phase 2. A `psibc` code fragment implementing the boundary conditions in (45) is

```
x110=xbc(k0(1)+1)
x210=xbc(k0(1)+2)
x11f=xbc(kf(1)+1)
x21f=xbc(kf(1)+2)
x120=xbc(k0(2)+1)
x220=xbc(k0(2)+2)
x12f=xbc(kf(2)+1)
x22f=xbc(kf(2)+2)
tau1=pv(1)
tau2=pv(2)

psi(1)=x110-a
psi(2)=x210-b
psi(3)=x11f-x120  ! continuity across phases
psi(4)=x21f-x220  ! continuity across phases
psi(5)=x12f
psi(6)=x22f
psi(7)=tau1-epsilon
psi(8)=tau2-epsilon
```

The initial guess for this problem is assembled from the solution of the single-phase problem by using the MADS Matlab data utilities. Assume that the single-phase output data file is "`bangbang1out.dat`" and that the corresponding `premads` file is "`bangbang1premads.dat`," and that the trajectory's control discontinuity occurs near the 27^{th} discretization interval. The code fragment for generating the input data for the two-phase problem is

```
S=importMADS('bangbang1premads.dat','bangbang1out.dat',0);
              % tau -- the duration of the single phase -- is needed to
              % compute durations of the two new phases.  In this case, the
              % only element of S.p is tau.  Note that, when there are
              % other static parameters than ``tau,'' it is convenient to
              % place the trajectory duration(s) at one end of S.p or the
              % other.
tau=p(length(S.p));
bout=breaktrajMADS(S.x,S.u,27,tau);
              % Overwrite various fields of S with two-phase data and
              % write a new premads file.
S.p=bout.tau;
S.x=bout.x;
S.ndv=bout.ndv;
S.u=bout.u;
              % The user is required to supply the following information for
              % constructing the premads file for the new problem. Note that
              % the new state, control, and trajectory constraint  dimensions
              % are identical with the old ones.
S.nph=2;
S.nxv=S.nxv*ones(2,1);
S.nuv=S.nuv*ones(2,1);
S.ncv=S.ncv*ones(2,1);
S.kodev=[0;0];
```

```
S.npsi=8;
writepreMADS(S,'bangbang2premads.dat');
              % Generate and write the input guess file for the two-phase
              % problem using adduxMADS, but without adding any additional
              % states or controls.
v=[];
for k=1:S.nph
  v=[v;adduxMADS(S.x{k},S.u{k},S.nxv(k),S.nuv(k),0)];
end
v=[v;S.p];
save bangbang2in.dat v -ascii -double
```

3.1.3 Introduce variable discretization step size

In order for the bang-bang control to be perfectly realized in the foregoing, it was necessary to break the problem into two phases, requiring an estimate of the point along the trajectory at which to position the break, and a substantial increase in the complexity of the `psibc` subroutine.

This can be avoided by allowing the integration intervals in the discretization to vary along the trajectory so that the duration of the subarc with max control will naturally be "τ_1" and that with minimum control will be "τ_2". Despite the fact that MADS nominally uses a fixed-step discretization, this can be achieved posing the time step as a control variable, rather than a scalar parameter:

$$x' = u_\tau(s)\dot{x}(s), \quad 0 \le s \le 1 \tag{46}$$

so that

$$\tau = \int_0^1 u_\tau ds \tag{47}$$

The increments u_τ are prevented from behaving irresponsibly by penalizing deviation of u_τ from its average value,

$$\int_0^1 (u_\tau - \tau)^2 ds \tag{48}$$

Additionally, in `cineq`, impose

$$u_\tau \ge c_\tau > 0 \tag{49}$$

where c_τ is a user-selected constant. There are three things to be noted in this case:

1. Although the plant dynamics are scaled by u_τ, in this case the elapsed time and penalty computations must not be.

2. The duration τ is the terminal boundary value of the differential equation for (47). This can be made to appear in the integral by imposing a boundary condition on τ and a free static parameter, say, p_τ, in `psibc`:

   $$p_\tau = \tau \tag{50}$$

   so that (48) would be expressed

   $$\dot{\phi_\tau} = (u_\tau - p_\tau)^2 \tag{51}$$

for a total performance index of

$$\phi = \tau + k_\tau \phi_\tau, \qquad k_\tau > 0 \text{ user} - \text{selected} \tag{52}$$

3. We generically deplore the use of penalty functions in trajectory optimization as being an unacceptably vague way of expressing performance goals and constraints. In this case, a penalty is introduced only to prevent a nonunique solution for the additional $(u_\tau)_k$ and p_τ degrees of freedom, rather than to compete with the principal goal of minimizing τ.

With the introduction of two additional states and one additional control, the problem elevates from being completely trivial to being fairly simple. Since these additional variables are being manipulated in four different user routines, it behooves the user to take measures to avoid mistakenly picking off the wrong element of the `xv` or `uv` vectors in different routines. Define a FORTRAN Module:

```
module bangMOD
implicit none          ! enforce strong typing.  Highly Recommended!
integer,parameter :: & ! ‘‘parameters’’ are compile-time fixed constants
& locx2=2            & ! elements of xv
& locxtau=3          &
& loctauint=4        &
& locu=1             & ! elements of uv
& locutau=2          &
& locptau=1            ! element of p
real(8),parameter :: &
& a=1                & ! initial value of x1
& b=1                & ! initial value of x2
& ctau=1d-6          & ! value of c_tau in u_tau constraint
& ktau=1d-2            ! penalty weight on penalty integral
end module bangMOD
```

The relevant code fragments are shown below. Only the fragment for subroutine `xdot` shows the use of the `module` in the subroutine header, but similar "`use`" statements appear in `cineq`, `phiobj`, and `psibc`.

xdot
```
     use bangMOD,only : locx2,locxtau,loctauint,locu,locutau,locptau
     implicit none
     integer,intent(in) :: nstate,kph,nx,nu,np
     real(8),intent(in) :: xv(nx),uv(nu),pv(np)
     real(8),intent((out) :: fv(nx)
     real(8) :: x2,xtau,tauint,u,utau,avetau
     x2=xv(locx2)
     xtau=xv(locxtau)
     tauint=xv(loctauint)
     u=uv(locu)
     utau=uv(locutau)
     avetau=pv(locptau)

     fv(1)=utau*x2
     fv(locx2)=utau*u
     fv(locxtau)=utau
     fv(loctauint)=(utau-avetau)
```

```
cineq !use bangMOD,only : locu,locutau,ctau  -- info only; belongs in declarations
      u=uv(locu)
      utau=uv(locutau)
      c(1)=u+1
      c(2)=1-u
      c(3)=utau-ctau

phiobj !use bangMOD,only : loctauint,locxtau,ktau -- info only; goes in declarations
      tauint=xbc(kf(1)+loctauint)
      taufinal=xbc(kf(1)+locxtau)
      phi=taufinal+ktau*tauint

psibc !use bangMOD,only : locx2,loctau,loctauint,locptau -- info only
      iebc=0
      psi(1)=xbc(k0(1)+1)-a      ! plant boundary conditions
      psi(2)=xbc(k0(1)+locx2)-b
      psi(3)=xbc(kf(1)+1)
      psi(4)=xbc(kf(1)+locx2)

      psi(5)=xbc(k0(1)+locxtau)              ! zero IC for utau
      psi(6)=xbc(k0(1)+loctauint)            ! zero IC for penalty integral
      psi(7)=pv(locptau)-xbc(kf(1)+locxtau)  ! place terminal value of utau in ptau
```

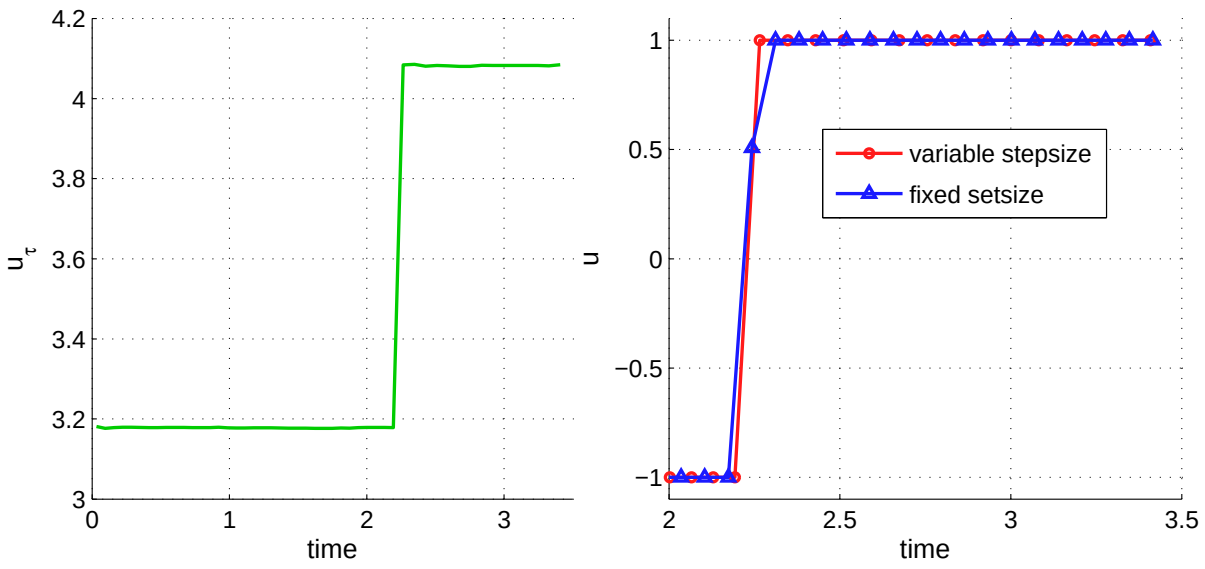

Figure 2. Two-State Bang-Bang Problem with Variable Time Step

This problem was solved, as in the case of uniform time step (UTS), with 50 discretization intervals, i.e. `nd` $= 50$, and the penalty weight on variation in u_τ was set to $k_\tau = 1/100$.

The initial guess for this problem was taken from the solution of the baseline case in Section 2.1.1, using the following script to add the additional two states and one control to the input data:

```
S=importMADS('bangbang1premads.dat','bangbang1out.dat',0);
tau=p(length(S.p));
S.nuv=S.nuv+1;     % more controls
S.nxv=S.nxv+2;     % more states
S.npsi=7;          % more boundary conditions
S.ncv=3;           % more trajectory constraints
writepreMADS(S,'varsteppremads.dat');
vout=adduxMADS(S.x,S.u,S.nxv,S.nuv,0)
vout=[vout;tau];
save varstepin.dat vout -ascii -double
```

Figure 2 displays details of the solution. The plot on the left displays u_τ as a function of time. The plot on the right compares details of the control trajectory for the UTS case and the variable time step (VTS) case. The VTS does provide a clean "bang" and, in fact, returns a very slight improvement in terminal time – $3.4495s$ versus $3.4502s$ for the UTS case. It's also interesting, but not surprising that the bang is initiated a little later for the VTS case.

3.1.4 Eliminate the Bangs

As was pointed out in Section 2.1, various functions of state and control variables can be formulated to take the place of "u" in $f(x, u, p)$, and these can be manipulated to control their temporal behavior; for example, limiting bandwidth. In this subsection, an alternate approach to enforcing limits on control accelerations is described — in order to illustrate a useful trick available to the user with the ME discretization.

The ME discretization can be adapted to provide a multistep delay buffer. Generally, this can be used to implement a sliding temporal window along the trajectory on which the problem formulation can impose conditions. For the current problem, we require three time steps for building up a numerical estimate for $\ddot{u}$, and implement the following differential equations in `xdot` **without time scaling**:

$$\left.\begin{array}{rcl} \dot{v}_1 & = & \mathtt{2nd}(-v_1 + u) \\ \dot{v}_2 & = & \mathtt{2nd}(-v_2 + 2v_1 - u) \end{array}\right\} \tag{53}$$

where `nd` is the number of discretization intervals. These ODEs, when ME-discretized give, at the k^{th} instant,

$$(v_1)_k = u_{k-1}, \quad (v_2)_k = u_{k-2} \tag{54}$$

Assuming a constant time step, $\ddot{u}$ at the $(k-1)^{th}$ instant is approximately

$$\ddot{u}_{k-1} \approx \left(\frac{\mathtt{nd}}{\tau}\right)^2 (u - 2v_1 + v_2)_k \tag{55}$$

where `nd` is the number of discretization intervals, and the expression for $\dot{u}$ is

$$\dot{u}_k \approx \left(\frac{\mathtt{nd}}{\tau}\right) (u - v_1)_k \tag{56}$$

Obviously, this trick can easily be adapted to the case of variable time steps by appropriately modifying the numerical differentiation formulae.

With these control derivatives in hand, the bandwidth of the optimal control solution can be limited either through a slovenly penalty function approach, such as adding a term like

$$\phi_{\ddot{u}} = \int_0^\tau (u - 2v_1 + v_2)^2 d\tau \tag{57}$$

to the cost function, or by imposing a constraint on some function of the acceleration.

As an illustration of the latter approach, we consider the trade between $|\ddot{u}|_{\max}$ – rather than the rms-like acceleration measure in (57) – and system performance. This trade can be explored either by fixing τ and minimizing $|\ddot{u}|_{\max}$ or vice versa. Pretend that our double-integrator is actually a physical system, and that the value of $|\ddot{u}|_{\max}$ plays an important role in its cost of manufacture. Pretend, further, that there is a range of potentially acceptable τ and that we want to find the "knee in the curve" in the relationship between $|\ddot{u}|_{\max}$ and τ. Since we know what τ we can tolerate, constrain τ and minimize $|\ddot{u}|_{\max}$ by solving the problem of minimizing

$$\phi = p_{\mathrm{ddmax}} \qquad (\texttt{phiobj}) \tag{58}$$

subject to

$$\left.\begin{array}{ll} -p_{\mathrm{ddmax}} \le \left(\frac{\mathtt{nd}}{\tau}\right)^2 (u - 2v_1 + v_2)_k \le p_{\mathrm{ddmax}} & (\texttt{cineq}) \\ p_{\mathrm{ddmax}} \ge 0 & (\texttt{psibc}) \end{array}\right\} \tag{59}$$

$$0 = \psi_\tau = \tau - \tau_{\mathrm{given}} \qquad (\texttt{psibc}) \tag{60}$$

and, implemented in `xdot` and `psibc`,

$$\left.\begin{array}{llll} \dot{x}_1 = \tau x_2, & x_1(0) = a, & x_1(1) = 0 \\ \dot{x}_2 = \tau u, & x_2(0) = b, & x_2(1) = 0 \\ \dot{v}_1 = 2(-v_1 + u), & \multicolumn{2}{l}{\text{boundary values free}} \\ \dot{v}_2 = 2(-v_2 + 2v_1 - u), & \multicolumn{2}{l}{\text{boundary values free}} \end{array}\right\} \tag{61}$$

Note that the acceleration constraints in (59) are indexed on the k^{th} instant rather than the $(k+1)^{th}$. This is necessary in order to have u in its proper place in the $u,\ v_1,\ v_2$ sequence of points.

Finally, why specify τ via the constraint (60) rather than by merely using the constant τ_{given} as the duration parameter? The lagrange multiplier associated with $\psi_{\tau_{\mathrm{given}}}$ in (60) is available from SNOPT (and hence, MADS) upon successfully converging a solution, and will supply useful information in computing the trade between τ_{given} and $|\ddot{u}|_{\max}$. In particular, as is well-known, express a hypothetical cosntrained minimization problem as

$$\left.\begin{array}{rcl} x^* & = & \arg\ \min_{c(x)=0} \mathcal{L}(x) \\ \mathcal{L} & = & \phi(x) - \lambda(c(x) - \delta)|_{\delta=0} \end{array}\right\} \tag{62}$$

where δ is a hypothetical small variation in the constraint setting. By differentating $\mathcal{L}$ by δ it's easy to see that

$$\mathcal{L}^*_\delta = \lambda^* \tag{63}$$

Thus, for our trade, the sensitivity of minimum peak control acceleration with respect to fixed terminal time constraint setting is $\lambda^*_{\tau\ \mathrm{given}}$.

Figure 3 displays solutions to this problem for several values of τ_{given} corresponding to 0.5%, 1%, 2%, and 5% degradation in τ from the perfect "bang-bang" value of $\tau = 3.4495s$. The plot on the right displays the resulting control histories for each τ_{given}. The plot on the left displays a hermite spline that uses the p_{ddmax} solution data and the associated lagrange

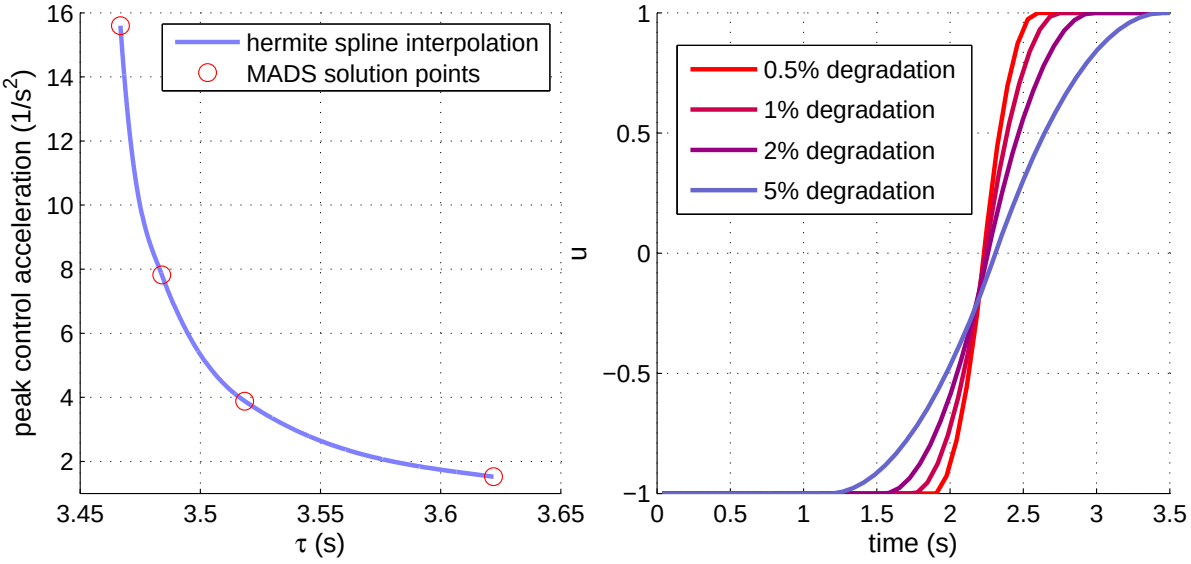

Figure 3. Two-State Bang-Bang Problem with Variable Time Step

multipliers for (60) to provide an estimate of peak $|\ddot{u}|$ away from the MADS solutions. The hermite spline used here is a cubic polynomial $y(s)$ defined on the interval $0 \leq s \leq x$ that satisfies the boundary conditions

$$\left.\begin{array}{rclrcl} y(0) & = & \bar{y}(0) & y'(0) & = & \bar{y}'(x) \\ y(x) & = & \bar{y}(x) & y'(x) & = & \bar{y}'(x) \end{array}\right\} \tag{64}$$

where the barred quantities are data. Thus, the lagrange multpliers provide "free" data that permit a cubic fit between pairs of solutions points, rather than the quartuples of solution points that are needed with cubic splines that operate only on point values. In this case, the $\bar{y}$ are the constrained system performance, i.e. the peak acceleration magnitudes, and the $\bar{y}'$ are the lagrange multipliers for the constraints (60). Table 3.1.4 displays the values of τ_{given}, p_{ddmax}, and the lagrange multiplier λ_{ψ_τ}.

Table 1. Data From MADS Solutions for Various Values of τ_{given}

τ_{given}	p_{ddmax}	$\lambda_{\tau\ \text{given}}$
3.4668	15.598	$-1.071(10^3)$
3.4840	7.8212	$-2.021(10^2)$
3.5185	3.8812	$-5.121(10^1)$
3.6220	1.5223	-9.8350

3.1.5 Problem Summary

This example has provided several things useful to the aspiring MADS user. They are listed below, associated with the subsection in which they appear:

1.1

1. Code fragments shown for casting a simple problem in MADS format. In particular, treatment of boundary conditions and cost is demonstrated, and an informal discussion of measures to be taken in using an autodifferentiation code for preparing the model for MADS is given.
2. Description of obtaining an initial guess, including a warning that the user needs to be wary of allowing free terminal time problems to freely vary terminal time.
3. The impact of fixed discretization stepsize on the fidelity with which discontinuous phenomena are preserved from the continuous time problem is demonstrated.

1.2

1. Brief description of breaking the original problem into two subarcs in order to recover the "bang-bang" control behavior is provided. Multiple-subarc problems will be dealt with in more detail in the next example

1.3

1. Technique for allowing variable discretization time step in the context of a single arc.
2. Demonstrated use of `psibc` to make state boundary values available to the system differential equations.

1.4

1. Demonstration of using the structure of the midpoint euler discretization (`kode` = 0) to model time derivatives of the optimal control solution. This permits direct manipulation of the temporal characteristics of the control without resorting to ineffective artifices such as placing filters into the plant dynamics to bandlimit the control.
2. Code for changing the structure of a single-phase MADS run's output data to provide an input guess for a MADS problem with that different structure. A subsequent problem will demonstrate breaking a run into multiple phases and changing the discretization density.
3. Demonstration of minimizing a variable's maximum value. The same approach can be used for constraining the same.
4. Demonstration and discussion of using lagrange multipliers that are output by SNOPT to provide better continuous realization of the variation of constrained system performance with constraint settings. This is of particular importance for getting the best possible performnace surveys when it is not feasible to perform a large number of MADS runs.

3.2 Goddard Problem

The Goddard problem [8] is a nonlinear optimal control problem in which a sounding rocket travels vertically to maximize its peak altitude. For the version of the problem considered here, the rocket flies through an atmosphere whose density decays expontially with altitude, the trajectory is terminated at its peak altitude, and an inequality constraint is imposed on the maximum dynamic pressure to which the rocket is exposed. The dynamics, taken from [8], have nondimensionalized states r – radius from Earth's center, V – speed, and m – mass. The state equations are

$$\dot{r} = V \tag{65}$$

$$\dot{V} = \frac{T - D}{m} - \frac{1}{r^2} \tag{66}$$

$$\dot{m} = -T/c, \quad c = 1/2 \tag{67}$$

where T is the control, thrust, satisfying

$$0 \leq T \leq 3.5 \tag{68}$$

and $D = K_D \bar{q}(r, V)$ is drag, with $K_D = 620$, and

$$\bar{q} = \frac{V^2}{2} \exp\left(\beta(1-r)\right), \quad \beta = 500 \tag{69}$$

is dynamic pressure. The dynamic pressure constraint is simply

$$\bar{q}_{\max} \geq \bar{q}(r, V) \tag{70}$$

Note that this is a state inequality constraint; that is, the control, T, does not appear in (70). The boundary conditions for the trajectory are

$$\left.\begin{array}{lllll} r(0) & = & 1 & V(0) & = & 0 & m(0) = 1 \\ & & & V(t_f) & = & 0 & m(t_f) = 0.6 \end{array}\right\} \tag{71}$$

The problem is solved by determining the thrust history that maximizes the terminal altitude when the rocket runs out of fuel; that is,

$$\phi = -r(t_f) \tag{72}$$

Aside from the nonlinearity of the plant and the presence of a state inequality constraint, the problem introduces us to an additional complication in the form of a "singular arc," described in Chapter 8 of [5]. Loosely speaking, a singular arc is a finite-length subinterval of a continuous-time optimal trajectory during which the trajectory cost function's sensitivity to control vanishes to first order. In other words, to first order, the optimal control problem simply doesn't care what the control does during a singular arc. There is, indeed, an optimal solution for the control, but its computation typically requires recourse to the calculus of variations (COV) to obtain higher-order necessary conditions for optimality, which are solved as a system of equations. The MADS user, on the other hand, uses first derivative information to directly minimize the cost function via NLP. In this case, it will be seen that the singular arc exhibits itself by a tendency toward ugly, jittery, behavior in the control during the singular arc.

This example will carry the user through solving the Goddard problem several different ways, and at several different levels of complexity. Although the basic problem is nearly trivial to solve, the user does have choices available in how to organize plant and inequality constraints, whether to suppress numerical artifacts in the control during singular arcs, or whether to optimze the distribution of discretization intervals to improve accuracy.

Before beginning the Goddard solutions, we propose guidelines for coding them – and the user's own problems. There are three main considerations in setting up problems in MADS.

- **Separate Plant Code:**
  There are typically more states and controls in the optimization problem than in the plant model. This was seen in the min-time linear problem of Subsection 3.1, and will be particularly seen in the Goddard problem. Because of this, the plant-specific modelling – particularly the plant dynamics – should be coded separately, to facilitate reliable re-use.

- **Named Scalars from Vectors:**
  Nonlinear models are characterized by scalar computations. Therefore, readability of code will be enhanced by separating these variables out from MADS state, control, and parameter vectors. For example, if `xv` $= [r,\ V,\ m]$ is the state vector, then the plant dynamics will be more readable using scalars than vector references.

```
r=xv(1)
V=xv(2)
m=xv(3)
T=uv(1)
D=620*exp(500*(1-r))*V**2/2
fv(1)=r
fv(2)=(T-D)/m-1/r**2
fv(3)=2*T
```

This is still seriously flawed, though. Consistent naming is needed throughout the problem code, and hardwiring this variable name indexing scheme in each MADS user routine would be horribly fragile; moreover, hardwiring plant dynamical parameters reduces code clarity and makes it difficult to update the model.

- **Use a MODULE to Set Indices and Parameters:**
  MADS uses four user-supplied routines to instantiate any trajectory problem, and plant information is used by each of them, so that having a single location in the user code for defining shared indices and parameters is essential. The FORTRAN MODULE provides an easy means to have such sharing. For our example,

```
module rocketMOD
implicit none ! enforce strong typing (highly recommended!)
integer,parameter :: & ! ‘‘parameters’’ are compile-time fixed constants
& locr=1,            & ! radius
& locV=2,            & ! speed
& Locm=3,            & ! mass
& locT=1,            & ! thrust (control)
& nxp=3,             & ! plant state dimension
& nup=1,             & ! plant control dimension
& kxp=1,             & ! index beginning plant state
& kup=1                ! index beginning plant control

real(8),parameter :: &
& KD=620,            & ! drag coefficient
& beta=500,          & ! atmospheric density lapse rate
& ISP=0.5D0,         & ! specific impulse
& mEmpty=0.6D0,      & ! rocket empty mass
& Tmin=0,            & ! min thrust
& Tmax=3.5D0           ! max thrust
end module rocketMOD
```

This permits us to rewrite the plant dynamics as

```
subroutine rocket(xp,up,fp)
                              ! note that ‘‘only’’ keyword prevents ‘‘rocket’’
                              ! from seeing nxp, nup, mEmpty, Tmin, Tmax
use rocketMOD,only : locr,locV,locm,locT,KD,beta,ISP
implicit none
real(8),intent(in) :: xp(3),up(1) ! state and control
real(8),intent(out) :: fp(3)      ! state rate
real(8) :: r,V,m,T,D
r=xp(locr)
```

```
V=xp(locV)
m=xp(locm)
T=up(locT)
D=KD*exp(beta*(1-r))*V**2/2
fp(locr)=V
fp(locV)=(T-D)/m-1/r**2
fp(locm)=T/ISP
end subroutine rocket
```

This approach is used throughout the examples, described below. Note that the source code for each example is located in Appendix GoddardAppx.

The organization of the remainder of this Section is as follows: In Subection 3.2.1, an initial guess for the problem is generated. Next, (Subsection 3.2.2,) the problem is solved directly, with and without a dynamic pressure constraint. Subsection 3.2.3 introduces an adhoc but effective penalty function which can be used to correct singlarity-induced freaks in the direct solution.

3.2.1 Obtaining an Initial Guess

Because the problem is substantially nonlinear, it will not do to simply choose random numbers as the initial guess for the Goddard problem. Yet, at the same time, we would prefer to keep the initial guess workload as low as possible. Experience indicates that boundary conditions are frequently the most difficult constraints to satisfy in generating a feasible trajectory in MADS. If an optimal MADS trajectory can be obtained for boundary conditions that have been "relaxed" in some way by starting from a random initial guess, perhaps such a solution will be an adequate initial guess for the problem with boundary conditions enforced.

It is reasonable to initially try solving a trajectory problem with the rocket dynamics as per (65-67) and cost function (72), but with the boundary conditions relaxed. This is done by moving them over to cost as penalty terms:

$$\phi = -r(t_f) + K * (r(0) - 1)^2 + V^2(0) + V^2(t_f) + (m(0) - 1)^2 + (m(tf) - 0.6)^2 \qquad (73)$$

where $K > 0$.

The highlights of the code are summarized below. Since this is a problem with free terminal time, the trajectory duration is introduced as an element of the MADS `pv` vector, and used to scale the RHS of the plant differential equations. The code fragment in `subroutine xdot` is

```
call rocket(xv(kxp),uv(kup),fv(kxp))
tau=pv(loctau)
fv=tau*fv
```

The code implementing (68) in `subroutine cineq` is

```
if(iecinflag.EQ.1)then
  iecinv=1   ! all cineq constraints are inequalities
  return
endif
T=uv(locT)
cv(1)=Tmax-T
cv(2)=T-Tmin
```

and the implementation of the cost function (73) in `subroutine phiobj` is

```
r0=xbc(k0(1)+locr)
rf=xbc(kf(1)+locr)
V0=xbc(k0(1)+locV)
Vf=xbc(kf(1)+locV)
m0=xbc(k0(1)+locm)
mf=xbc(kf(1)+locm)
phi=-rf+K*((r0-1)**2+V0**2+Vf**2+(m0-1)**2+(mf-mEmpty)**2)
```

For this problem, `psibc` implements only a constraint ensuring that optimal duration τ^* is not zero or negative:

$$\psi = \tau - \tau_{\min} > 0 \tag{74}$$

The parameters $\tau_{\min}$ and `mEmpty` pertain to the problem definition, rather than the plant description. As such, they are placed in a separate module file:

```
subroutine GGUESSMOD
implicit none
real(8) ::  &
& taumin,   &
& K
end GGUESSMOD
```

An initial guess for this problem is generated from random numbers in `makeGGUESS.m`, given in the Appendix, is essentially

```
randmag=0.01
P.nxv=3;
P.nuv=1;
P.nav=2;
P.ndv=200;
P.np=1;
P.nph=1;
P.kodev=0;
P.nbc=1;
vin=randguessMADS(P.nxv,P.nuv,P.ndv,P.np,randmag)+1;
save GGUESS_vin.dat vin -ascii -double
name='GGUESS_premads.dat';
writepremadsMADS(P,name)
```

The files generated by this code correspond to the main routine code

```
program GGUESS
use GGUESSMOD,only : taumin,K
taumin=1D-4
K=100
Finput=13
open(13,file='GGUESS_vin.dat',status='old')
Fpremads=14
open(14,file='GGUESS_premads.dat',status='old')
Foutput=15
open(15,file='GGUESS_vout.dat',status='unknown')
call batchMADS(Finput,Fpremads,Foutput,info)
end program GGUESS
```

Finally, the solution is organized for plotting in Matlab by executing

```
S=importMADS('GGUESS_premadst.dat','GGUESS_vout.dat',0)
```

where `S` contains the solution data in a form suitable for graphics. The details of `importMADS`, recall, are given in Section 1.3.2.

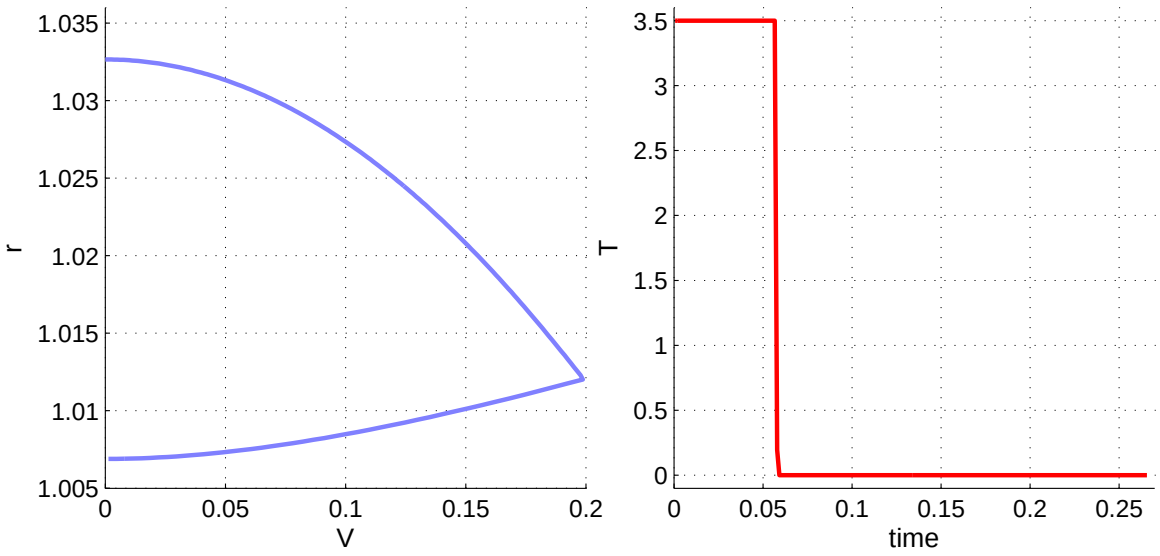

Figure 4. Initial Guess for Goddard Problem

Figure 4 displays the solution to this problem. Note that the altitude boundary conditions are seriously violated, and that the thrust profile is substantially different from that in Figure 5, lacking the intermediate thrust arc that appears in the optimal solution. Effectively, the solution has moved the starting point for the trajectory to a higher altitude, where the atmosphere is thinner, reducing drag, so that the "bang-bang" thrust solution produces optimal altitude gain.
This section has provided the following:

- It has provided and illustrated advice on organizing a MADS problem. The key points are –
  1. Separate plant-related code from the rest of the MADS problem code, because the MADS formulation may introduce additional states, control, and parameters.
  2. Provide a consistent, globally available, index to identify each individual element of state, control, and parameter vectors.
  3. Place these in `MODULE`s, separating the `MODULE` or `MODULE`s dedicated to the plant dynamics from those that will carry values related only to the particular problem formulation.
- It has discussed a philosphy for starting MADS solutions from initial guesses that consist only of random numbers. This was demonstrated by posing a problem with boundary conditions eliminated and replaced by a penalty function. While we do not claim that it will always work, it's a good place to start.

3.2.2 Simple Solution with Dynamic Pressure Constraint

This subsection provides the direct, unconstrained solution for the Goddard problem, and demonstrates two approaches for imposing constraints that involve states in `cineq`. The unconstrained problem is posed by retaining `xdot` and `cineq` from Subsection 3.2.1, and moving the boundary conditions from the penalty function (73) to `psibc` from `phiobj`. In `psibc`,

```
if(iebcflag.EQ.1)then
  iebcvec=0         ! The boundary conditions are equalities
  iebcvec(6)=1      ! The constraint on tau versus taumin is inequality
  return
endif
psi(1)=xbc(k0(1)+locr)-1
psi(2)=xbc(k0(1)+locV)
psi(3)=xbc(k0(1)+locm)-1
psi(4)=xbc(kf(1)+locV) ! Not theoretically necessary, but sharpens the problem
psi(5)=xbc(kf(1)+locm)-mEmpty
psi(6)=pv(loctau)-taumin
```

and, in `phiobj`,

```
phi=-xbc(kf(1)+locr)
```

Note that for the very simple and unrepetitive expressions in `psibc` and `phiobj`, here, we've not bothered to break out individually named scalar variables. The code, as written, is perfectly readable.

Assuming that the initial guess has been stored in GGUESS.mat as a `importMADS` structure "`S`," the script for setting up the inputs for the MADS run will be

```
load GGUESS   % Assume this .mat file contains the structure S from using
              % importMADS on GGUESS_vout.dat and GGUESS_premads.dat
randmag=0.0;
P=S;          % Mostly, this problem is structured the same as GGUESS.
P.nbc=6;      % This problem has a different number of boundary conditions
vin=adduxMADS(S.x{1},S.u{1},P.nxv,P.nuv,randmag); % This reorganizes S.x and
                                                  % S.u back into a vector...
vin=[vin;S.p];            % and the duration parameter is tacked onto the end
save G1unc_vin.dat vin -ascii -double
name='G1unc_premads.dat';
writepremads(P,name)
```

We now solve the problem, operate on the output and "premads" files with `importMADS`, and save the resulting structure in `G1unc.mat`. Figure 5 displays the resulting altitude/speed and thrust plots. The most noticeable feature of the displayed solution is the noisy appearance of the thrust history. Why is this? The optimal trajectory for this case is "singular", and can be explained by informal appeal to variational optimal control theory.

In variational optimal control, the Minimum Principle [6], [7], states that the optimal trajectory minimizes the problem's hamiltonian function, $\mathcal{H}$, where, for $\dot{x} = f(x, u)$,

$$\mathcal{H} = \lambda^T f(x, u), \qquad \dot{\lambda} = -f_x^T \lambda \tag{75}$$

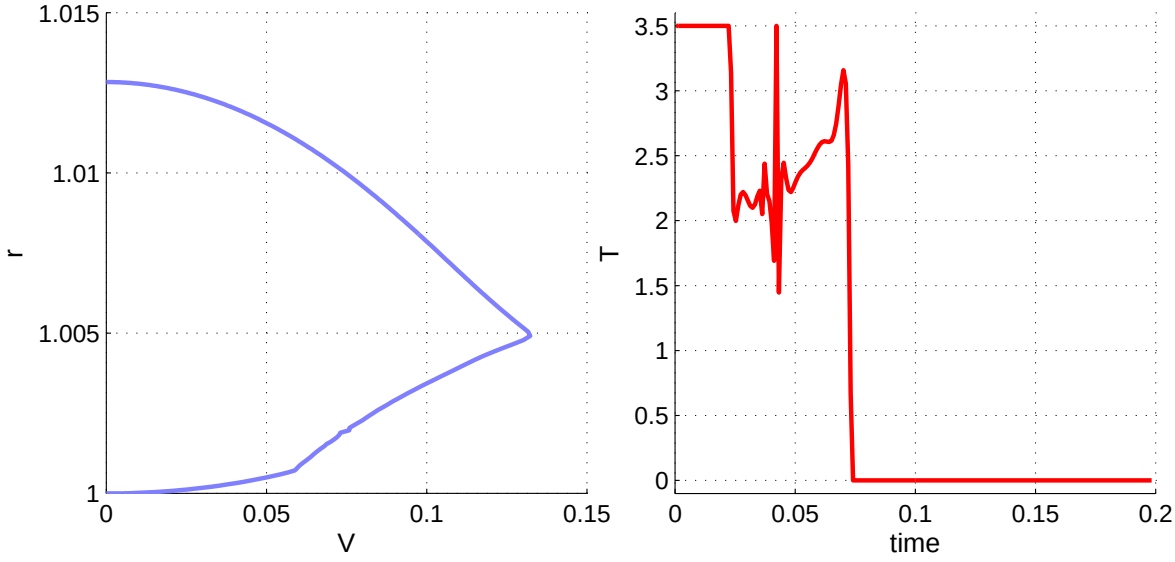

Figure 5. Goddard Solution Without Dynamic Pressure Constraint

and λ is a vector roughly analogous to the instantaneous value of trajectory of lagrange multipliers. Because $\mathcal{H}$ is to be minimized by thrust $T(t)$, its partial derivative trajectory $\mathcal{H}_T$ provides a useful diagnostic. When $\mathcal{H}_T \neq 0$, optimality requires

$$T^* = \begin{cases} T_{\min} & \text{for} \quad \mathcal{H}_T > 0 \\ T_{\max} & \text{for} \quad \mathcal{H}_T < 0 \\ \text{some intermediate value} & \text{for} \quad \mathcal{H}_T = 0 \end{cases} \tag{76}$$

The "singularity" in this problem arises from the fact that, since T enters the dynamics linearly in (66), T is absent from $\mathcal{H}_T$, so that it cannot be used to determine optimal T^*. Additionally, $\mathcal{H}_{TT}$ is identically zero (hence singularity.) In practical terms, up to second order, the optimal trajectory simply "doesn't care" about the particular value of T when $\mathcal{H} = 0$. This, in turn, means that for our direct minimization solution process, the absence of performance impact due to thrust deprives the solution iterations information necessary to settle on a locally unique thrust trajctory, even though those iterations satisfy the solution algorithm's convergence criteria.

Figure 6 displays the optimal trajectory of $\mathcal{H}_T$ for this problem, and the sequence of $T_{\max}$, followed by some intermediate value, followed by $T_{\min}$ is clear. This Figure, incidentally, was generated by using MADS to emulate a variational solution to the Goddard problem, posing a discretization of the variational necessary conditions for optimal control as a penalty function to be minimized by meeting the boundary conditions and constraints.

This singular behavior is not a mere curiosity. It most frequently occurs in problems in which the dynamics are linear in one or more control variables – a category that includes aerospace systems that involve propulsion. An extremely coarse cure for this issue would be to introduce a nonlinearity in T into the problem dynamics, such as an integral thrust

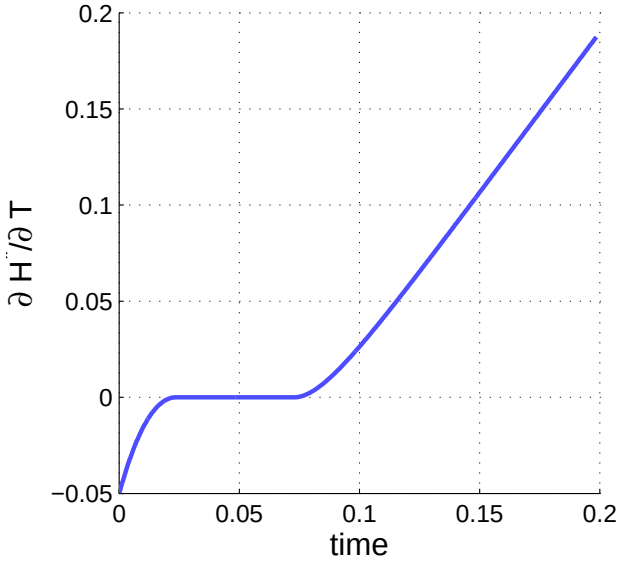

Figure 6. Hamiltonian Thrust Partial Derivative for Goddard Problem

penalty,

$$\phi_{\text{penalty}} = -r(\tau) + \int_0^\tau T^2 dt \tag{77}$$

Such a penalty would cure the thrust jitters, since the additional state for the penalty integral is nonlinear in T. The cure comes at the cost, however, of moving the performance goal away from maximizing the terminal altitude, and toward minimizing thrust usage. We are no longer solving the original problem. Short of adopting a full variational solution, it is best to introduce the needed nonlinearity in a way that only neglibly affects optimal performance, particularly during the nonsingular problem phases.

Let's introduce a dynamic pressure constraint:

$$\left.\begin{array}{c} \bar{q}_{\max} - \bar{q}(r,V) \geq 0 \\ \bar{q} = \frac{V^2}{2}\exp(\beta(1-r)) \end{array}\right\} \tag{78}$$

This is the first instance, in this tutorial, of a state inequality constraint. Since, in MADS, control is held constant across each discretization interval, expressing a control-only inequality is straightforward. What of the case where states are involved in the constraint expression, as in (78)? States appear at different instants in `cineq`, and in the discretization logic that calls `xdot`. In `cineq` the state is available at the beginning (`xj`) and end (`xjp1`) of each discretization interval. Depending on the value of `kode`, `xdot` may be called with the midpoint average (`xj` + `xjp1`)/2, for `kode` = 0, or with several extrapolated values, when using the Runge Kutta (RK) discretizations, `kode` = 1, 2.

In the case.of the RK discretizations, there is no choice but to compute inequality constraint quantities in `cineq` using state values from xj or $xjp1$. In the mindpoint Euler case, however, the user has the option of computing a constraint quantity, say "`y`," in `xdot`, assigning a state, say `xv(locy)`, to it and using an expression like

$$\texttt{fv(locy)} = 2 * \texttt{nk} * (\texttt{y} - \texttt{xv(locy)})$$

to propagate it, so that the current value of `y` is available to `cineq` at `xjp1(locy)`. Recall that we introduced this trick in the Minimum-Time Double Integrator problem in Subsection 2.1.4.

Why would one chose to compute `y` this way? The most direct motivation for doing this would be when computation of `y` is tightly integrated with the software called in `xdot` for computing the RHS of the plant ODEs. Extracting `y` from the existing plant dynamics computations might be preferable, from a software point of view, to redundantly coding the logic for computing `y`. Furthermore, if the computation of `y` is not only tightly integrated with the plant dynamics, but also expensive, the user has further incentive to compute `y` in `xdot`.

If the user chooses to set the problem up with `cineq`-related quantities computed in `xdot`, it must be remembered that they're being computed at the midpoint of the integration interval, i.e. at `(xj+xjp1)/2` in the notation of the `cineq` argument list. The practical implications of this are explored in the remainder of this example.

The dynamic pressure constraint (78) is implemented in subroutine `calcqbar`:

```
subroutine calcqbar(xv,qbar)
use rocketMOD,only : c,kD,beta,locr,locV,nxp
implicit none
real(8),intent(in) :: xv(nxp)
real(8),intent(out) :: qbar
real(8) :: dexp
real(8),parameter :: ONE=1.D0

real(8) :: r,V
r=xv(locr)
V=xv(locV)
qbar=dexp(beta*(ONE-r))*V**2/2
end subroutine calcqbar
```

For the formulation in which `qbar` is computed in `xdot` with `kodev=0`, the relevant lines of `xdot` and `cineq` are

```
real(8) :: tau,qbar
call rocket(xv(kxp),uv(kup),fv(kxp))
tau=pv(loctau)
fv(kMain)=tau*fv(kMain)
call calcqbar(xv(kxp),qbar)
fv(locqbarm)=2*nk*(qbar-xv(locqbarm))
```

and

```
if(iecflg.EQ.1)then
  iecin=1
  return
endif
cv(1)=Tmax-uv(locT)
cv(2)=uv(locT)-Tmin
cv(3)=qbarmax-xjp1(locqbarm)
```

respectively. With `qbar` computed in `cineq`, the code in `xdot` is

```
call rocket(xv(kxp),uv(kup),fv(kxp))
tau=pv(loctau)
fv=tau*fv
```

and

```
if(iecflg.EQ.1)then
  iecin=1
  return
endif
cv(1)=Tmax-uv(locT)
cv(2)=uv(locT)-Tmin
call calcqbar(xj(kxp),qbar)
cv(3)=qbarmax-qbar
```

Note the differences in `xdot`. For the `kodev=0` case, (the former,) the time scaling is restricted to `fv(kMain)`, rather than to all of `fv`. The index vector is set in `G1MOD.f90`, given below:

```
module G1MOD
use rocketMOD,only : nxp
implicit none

integer,parameter :: &
& locqbarm=nxp+1        ! state for carrying qbar to cineq

integer,parameter ::  &
& loctau=1              ! time-scaling parameter in pv

real(8),parameter ::  &
& mfinal=0.6D0          ! empty mass

integer :: kmod
integer,parameter :: &   ! indices identifying states to be time-scaled.  The
                         ! lag state carrying qbar is *not* time-scaled.
& kMain(nxp)=(/(kmod,kmod=1,nxp)/)

real(8) ::            &
&  taumin,            & !
&  qbarmax              !
end module G1MOD\
```

The index vector `kMain` is initialized to `kMain=[1,2,3]`, above, and the state variable for passing `qbar` is given the index `locqbarm=nxp+1`, appending it the plant state vector as a fourth state.

Both of the above versions of the `qbar`-constrained problem were started using the unconstrained solution as an initial guess. For the former case, the additional state `xv(locqbarm)` was added to the input guess vector using the script

```
S=importMADS('G1unc_premads.dat','G1unc_vout.dat',0);
randmag=0.0;
P.nxv=4;         % note additional state
P.nuv=S.nuv;
```

```
P.nav=3;          % additional inequality constraint
P.ndv=S.ndv;
P.np=S.np;
P.nph=S.nph;
P.kodev=S.kodev;
P.nbc=S.nbc;
vin=adduxMADS(S.x{1},S.u{1},P.nxv,P.nuv,randmag); % This will add a state
vin=[vin;S.p];
save G1_vin.dat vin -ascii -double
name='G1_premads.dat';
writepremads(P,name)
```

The script converts the "`G1unc`" output from a column vector to a `importMADS` structure, and then uses "`adduxMADS`" to create a new vector with the fourth state appended.

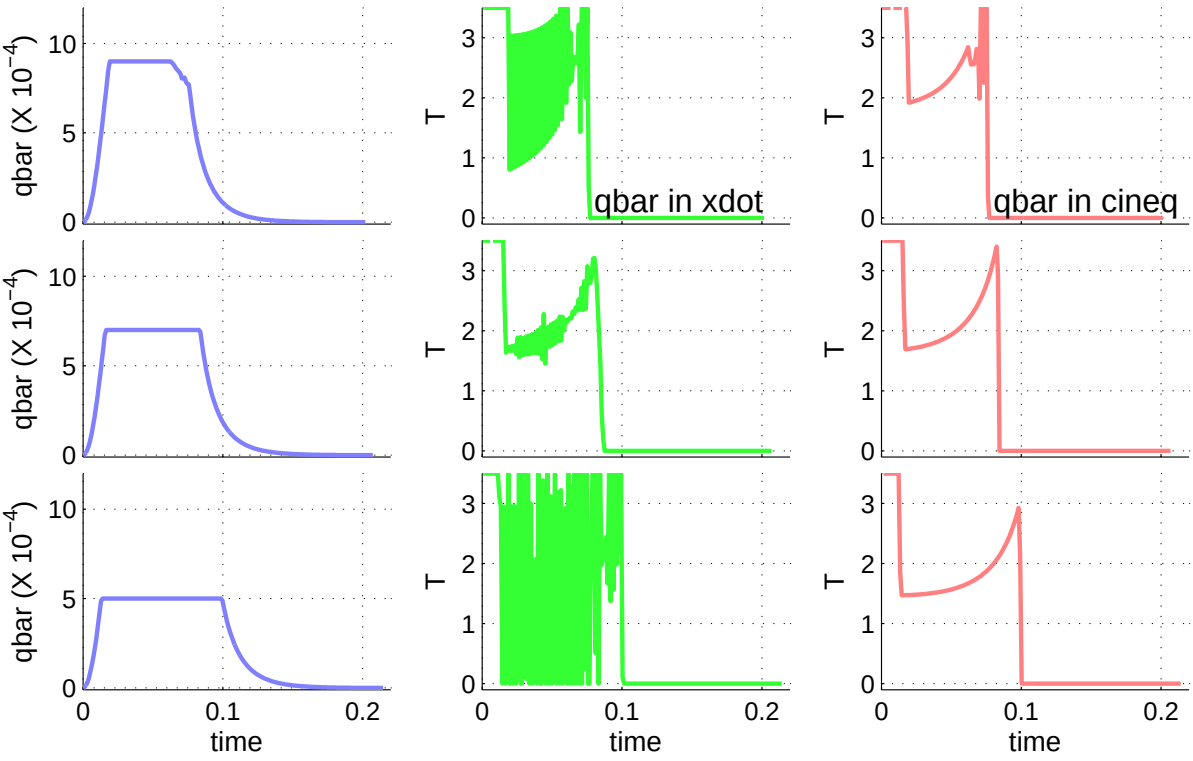

Figure 7. Thrust Solution Behavior for Several Values of `qbarmax`

Figure 7 compares the behavior of the optimized thrust for several values of `qbarmax` (unconstrained maximum $\bar{q}$ is roughly $1.2(10^{-3})$, and for $\bar{q}$ computed in `xdot` versus $\bar{q}$ computed in `cineq`. Scanning from the left, the first column displays the $\bar{q}$ histories (taken from the cases where $\bar{q}$ was computed in `cineq`). The middle column displays the corresponding optimized thrust histories where $\bar{q}$ was computed in `xdot`, and the third column shows the corresponding thrust histories with $\bar{q}$ computed in `cineq`. All of these runs successfully converged.

Obviously, Figure 7 demonstrates that computing the state constraint quantity in `xdot` leaves the user vulnerable to misbehavior in the control solution. The reason for this is that satisfying the constraint at the midpoint of each discretization interval does not guarantee that the constraint rate is zeroed. At this point, Murphy's Law takes charge. This does not necessarily mean that it never makes sense to compute the constraint quantity in `xdot`, but it certainly indicates that additional measures need to be taken in this case, before the user can be confident of acceptable results.

There is more to be seen in this Figure. Observe how the $\bar{q}$ profile for `qbarmax` $= 7(10^{-4})$ falls off at the end of the active constraint arc, signalling a probable singular arc. This tentative diagnosis is supported by the choppy behavior of the thrust solution in column three. Alternatively, for `qbarmax` $= 5(10^{-4})$, the active constraint arc completely eliminates the singular arc, giving a very crisp $T_{\max}-$constrained$-T_{\min}$ thrust profile. For `qbarmax` $= 7(10^{-4})$, however, close examination of the thrust profile shows a little rounding of the thrust history near the end of the active constraint arc. Is this a very short singular arc? The only way to tell for sure is by solving the corrsponding variational optimal control problem, which can be quite intricate, per Chapter 8 of [5].

The user may very likely not care about the forensic details that are opened up by a full variational analysis, but would just like to tame bad control behavior on singular arcs. One way of doing this is assign a penalty function designed to suppress jitter without interfering with large control movements. This approach is described in the sequel. Before proceeding, quickly review what has been introduced in this Subsection:

- Two different approaches for imposing a state inequality constraint were described and demonstrated, and pros and cons of each were discussed.
- One approach (computing constraint quantities in `xdot`) requires an additional state. Logic for adding states to an input guess was demonstrated.
- Logic for restricting time scaling of the state derivative to a subset of the states was shown.

3.2.3 A Penalty Function to Smooth Out Singular Jitter

The easiest, and least elegant, way to suppress singularity-induced control jitter is to penalize it. Looking at Figure 7, the reader will agree that control jitter is characterized by persistent, large, acceleration; so, it would make sense to penalize control acceleration. At the same time, however, it would be best if the penalty did not suppress or distort the rapid "bang-bang" control shifts associated with leaving $T_{\max}$ and moving to $T_{\min}$; they are, after all, optimal. A compromise penalty would penalize control acceleration more heavily when control rates are low, with the expectation that superfluous acceleration (jitter) would disappear, leaving necessary accleration (step changes) relatively intact. Such a penalty would take the form

$$\mathcal{J}_{\text{pen}} = \int_0^1 \frac{\ddot{u}^2}{\epsilon_{\text{pen}} + \dot{u}^2} dt \tag{79}$$

where it should be noted that we recommend that the penalty state be integrated from 0 to 1, i.e., without time scaling. This could be implemented directly by redefining the control variable to be thrust acceleration, i.e., $u = \dot{T}$ and appropriately introducing additional states to realize (79) and T. MADS approximates (79) in `subroutine ddpenMADS`, which differences across time steps to model acceleration and rates:

$$\frac{d}{dt}\mathcal{J}_{\text{ddpen}} = \frac{a^2}{\epsilon_{\text{pen}} + v_-^2 + v_+^2}, \quad \left\{ \begin{array}{rcl} a &=& T_k - 2T_{k-1} + T_{k-2} \\ v_- &=& T_{k-1} - T_{k-2} \\ v_+ &=& T_k - T_{k-1} \end{array} \right. \tag{80}$$

where the $(\cdot)_k$ subscripts denote the time index in MADS. Implementing the time differencing in `ddpenMADS` requires three states, and the penalty integral introduces a fourth state.

This penalty is applied to the problem of Subsection 3.2.2 by adding the states needed for `ddpenMADS` and $\mathcal{J}_{\text{ddpen}}$, and including the penalty in the performance index:

$$\phi = -r(\tau) + K_{\text{pen}}\mathcal{J}_{\text{ddpen}}(1) \tag{81}$$

where $K_{\text{pen}} \geq 0$ is user-specified. The variables are organized in

```
module SLO1MOD
use rocketMOD,only : nxp
implicit none

integer,parameter ::    &
& locqbarm=nxp+1,       & ! carry qbar to cineq       (4)
& locddpv=locqbarm+1,   & ! 3-element state for ddpen (5)
& locddpint=locddpv+3     ! ddpen penalty integral    (8)

integer,parameter ::  &
& loctau=1              ! time-scaling parameter in pv

integer :: kmod
integer,parameter :: &   ! indices identifying states to be time-scaled.  The
                         ! ddpen states not time-scaled.
& kMain(nxp)=(/(kmod,kmod=1,nxp)/)

real(8) ::            &
&  taumin,            & !
&  qbarmax,           & !
&  epsvdd,            & ! denominator bias term in ddpen
&  Kddp                 ! weighting term for ddpen penalty
end module SLO1MOD
```

and the logic in `xdot` becomes

```
call rocket(xv(kxp),uv(kup),fv(kxp))
call calcqbar(xv(kxp),qbar)
fv(locqbarm)=2*nk*(qbar-xv(locqbarm))
   ! Note that 4th argument of ddpenMADS is ‘‘3’’  That is the number of
   ! states that the user needs to make available to ddpenMADS, and the
   ! subroutine checks to see that the user has at least declared that he has
   ! provided the right number of states.
call ddpenMADS(nk,uv(kxp),xv(locddpv),3,epsvdd,fv(locddpv),fv(locddpint))

tau=pv(loctau)
fv(kMain)=tau*fv(kMain)
```

there is no change in `cineq`, but the integral state $\mathcal{J}_{\text{ddpen}}$ does require a zero initial condition, so that the logic becomes

```
psi(1)=xbc(k0(1)+locr)-1
psi(2)=xbc(k0(1)+locV)
```

```
psi(3)=xbc(k0(1)+locm)-1
psi(4)=xbc(k0(1)+locddpint)     ! zero I.C. for penalty integral
psi(5)=xbc(kf(1)+locV)
psi(6)=xbc(kf(1)+locm)-mfinal
psi(7)=pv(loctau)-taumin
```

The cost function is implemented in `phiobj` as

```
rf=xbc(kf(1)+locr)
phi=-rf + Kddp*xbc(kf(1)+locddpint)
```

Unconstrained solutions for $K_{\rm pen} = \{10^{-3}, 10^{-2}, 10^{-1}\}$ and $\epsilon_{\rm pen} = 10^{-2}$ were obtained using the solution from Figure 5 as an initial guess, and the results are displayed in Figure 8. The left panel dispays the falloff in altitude performance as percentages for the several values of $K_{\rm pen}$. The right panel displays the T time histories for the $K_{\rm pen} = 10^{-3}$ and 10^{-1} cases. If one enlarges the thrust profile for $K_{\rm pen} = 10^{-3}$, small jitters are visible, but there is a substantial improvement from the behavior shown in Figure 5. The profile for $K_{\rm pen} = 10^{-1}$ is smooth, but shows very little similarity to the optimal solution. Nonetheless, referring to the Figure's left panel, there is only 0.03% degradation in altitude performance. This is a consequence of the presence of the singular arc – recall that the optimal performance is almost entirely insensitive to the particular thrust profile over a significant portion of the trajectory.

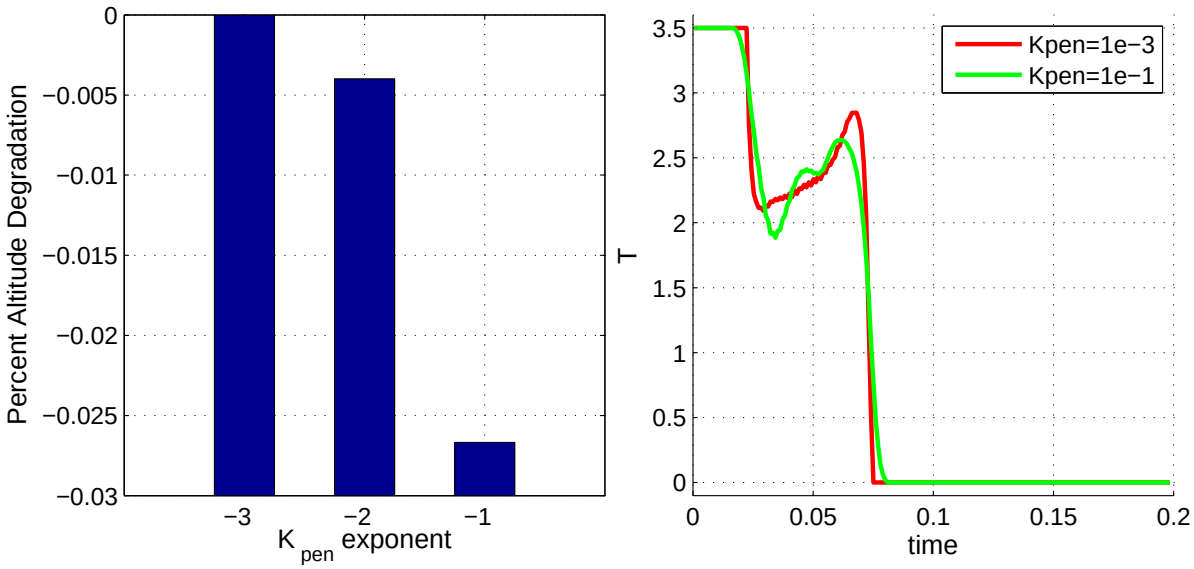

Figure 8. Variation of Altitude Performance and Thrust Behavior with $K_{\rm pen}$

Solutions were next generated using the jitter penalty, for the same values of `qbarmax` as those displayed in Figure 7. Figure 9 displays the resulting thrust profiles. In the left column of the Figure, all runs used the solution for $K_{\rm pen} = 10^{-3}$ from Figure 8 as the initial guess, and $K_{\rm pen}$ and $\epsilon_{\rm pen}$ were selected as displayed on each panel. The Figure's right column display thrust for the case where the solutions were "walked in," i.e., the solution for

`qbarmax` $= 9(10^{-4})$ is started from the unconstrained case, that for `qbarmax` $= 7(10^{-4})$ is started from the solution for `qbarmax` $= 9(10^{-4})$, and so on. We see that the latter approach provides cleaner-looking solutions for less intrusive K_{pen}, ϵ_{pen} settings.

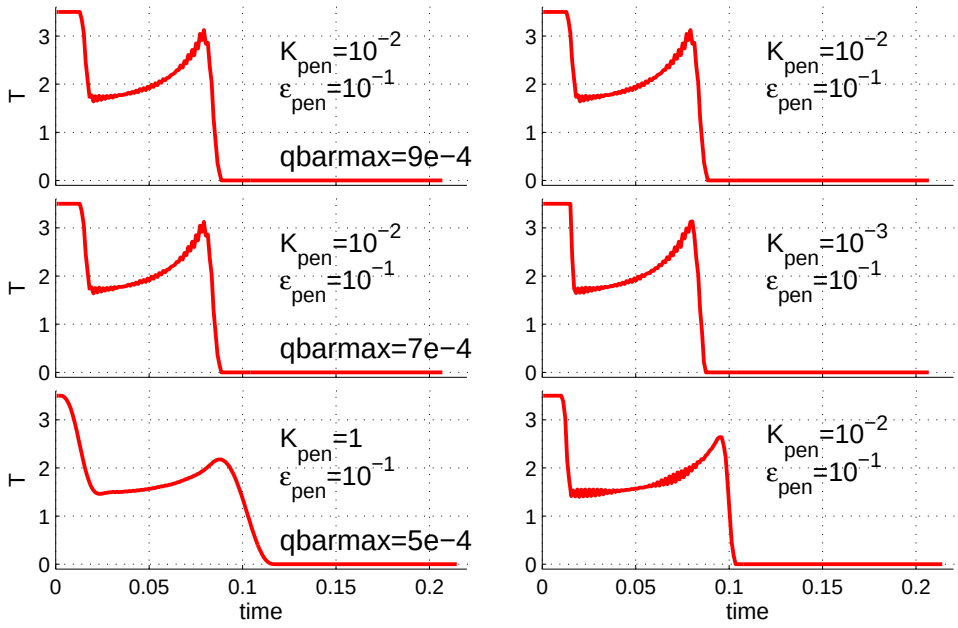

Figure 9. Comparison of $\bar{q}$-constrained Thrust Profiles with Jitter Penalty

We most particularly note, however, that all of these $\bar{q}$-constrained runs required heavier jitter penalties than the unconstrained case from Figure 8. Recall that, in the unconstrained case, the singular arc jitter exists because the optimization is largely oblivious to the value of T; therefore, a very small penalty suffices to correct the jitter. In the actively $\bar{q}$-constrained cases, the optimization is taking advantage of the midpoint Euler discretization , settling on a choppy T solution to maximize terminal altitude. A heavier penalty is required to compete with this, and the overall T solution suffers additional distortion.

Before proceeding further, let's review where we are in solving the Goddard problem. We saw in Figure 8 that, using `ddpenMADS`, we could obtain a fairly clean, fairly undistorted unconstrained solution. We also saw that, even if we did penalize singularity-induced jitter so heavily that the thrust profile changed significantly from the true optimal solution, it wouldn't matter have much impact on performance. In imposing an inequality constraint on $\bar{q}$, we saw, from Figure 7, that we had our best results computing the constraint in `cineq` rather than in `xdot`. The trajectory for `qbarmax` $= 9(10^{-4})$, though, is marred by an untidy singular arc that follows the active constraint arc.

What can be done about this latter case? The simplest thing would be to apply a `ddpenMADS` penalty to the problem, while using `cineq` to compute $\bar{q}$. The result, for $K_{\text{pen}} = 10^{-4}$, $\epsilon_{\text{pen}} = 10^{-2}$ is shown in Figure 10. Like the lightly penalized unconstrained case in Figure 8, it provides a "pretty good" solution, probably good enough for any practical application.

What – in all of these singular cases – if we want cleaner results? Consider, again, the `qbarmax` $= 9(10^{-4})$ case from Figure 7. It is easy to see that the optimal trajectory goes through four distinct phases: max thrust – max $\bar{q}$ – singular arc – min thrust. Conceptually, we could partition the trajectory and apply each phase's appropriate constraints, one at a

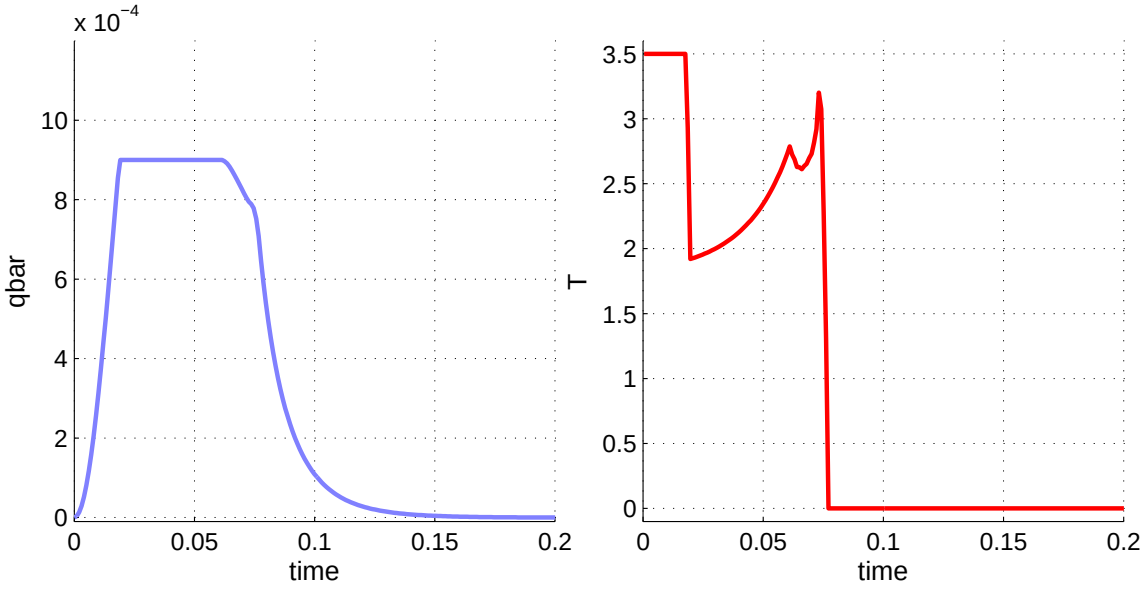

Figure 10. Comparison of $\bar{q}$-constrained Thrust Profiles with Jitter Penalty

time: Require:

$$\begin{array}{lccl} \text{Phase } 1\ldots & T_{\max} = T, & \texttt{qbarmax} \geq \bar{q} & \\ \text{Phase } 2\ldots & T_{\max} \geq T \geq T_{\min}, & \texttt{qbarmax} = \bar{q} & \\ \text{Phase } 3\ldots & T_{\max} \geq T \geq T_{\min}, & \texttt{qbarmax} \geq \bar{q}, & \texttt{ddpenMADS} \left\{ \begin{array}{l} K_{\text{pen}} = 10^{-4} \\ \epsilon_{\text{pen}} = 10^{-1} \end{array} \right. \\ \text{Phase } 4\ldots & T \geq T_{\min}, & \texttt{qbarmax} \geq \bar{q} & \end{array}$$

The result is shown in the upper plots in Figure 11. The lower pair of plots in the Figure are the data from Figure 10. There are clearly some differences between the two problem formulations, but we will defer their discussion until after showing code fragments for setting up the four-phase problem.

Multi-phase problems have several key differences from single-phase problems, that must be kept in mind:

1. Each phase will have its own duration. For this problem, the relevant lines of `xdot` are

```
call rocket(xv(kxp),uv(kup),fv(kxp))
tau=pv(loctau0+kph)
fv(kMain)=tau*fv(kMain)      ! time-scale only the rocket plant states

if(kph.EQ.3)then
  call ddpenMADS(nk,uv(locT),xv(locddpv),3,epsvdd,fv(locddpv),fv(locddpint))
endif
```

   where `loctau0` is set in the problem's `MODULE` file as

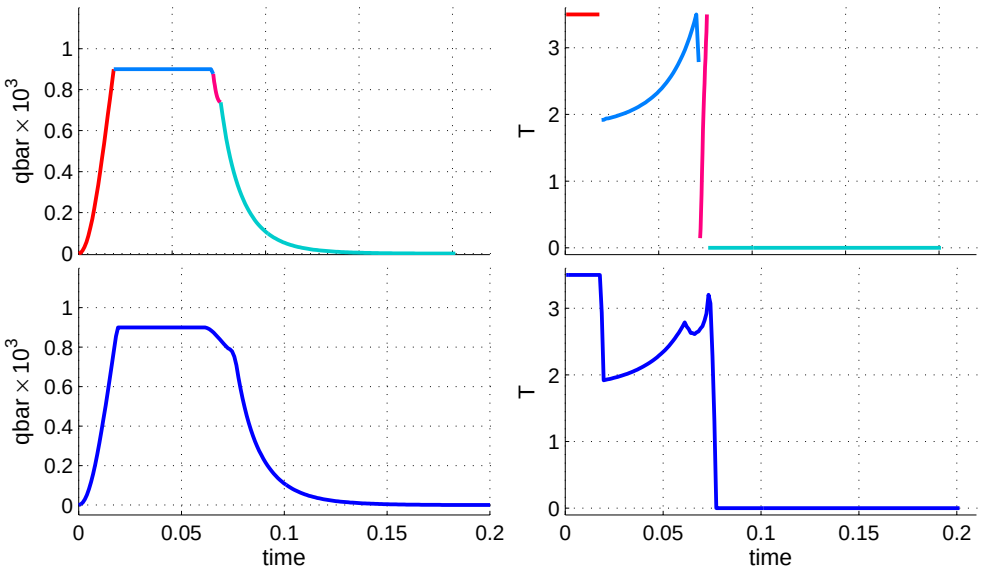

Figure 11. Comparison of One- and Four-Phase $\bar{q}$-constrained Cases with Jitter Penalty

```
integer,parameter ::  &
& loctau0=0              ! zero-base pointer for the taus for four phases
```

2. Each phase may have a different pattern of equality and inequality constraints in `cineq`. Here,

```
if(iecflg.EQ.1)then
  select case(kph)
    case(1)
      iecin=(/0,1/)      ! Tmax, q bounded
    case(2)
      iecin=(/1,1,0/)    ! T bounded, qmax
    case(3)
      iecin=(/1,1,1/)    ! T and q bounded (singular)
    case(4)
      iecin=(/0,1/)       ! Tmin, q bounded
    end select
  return
endif

call calcqbar(xj(kxp),qbar)
select case(kph)
  case(1)
    cv(1)=Tmax-uv(locT)    ! T=Tmax
    cv(2)=qbarmax-qbar     ! qbarmax >= qbar
  case(2)
    cv(1)=Tmax-uv(locT)    ! Tmax>=T
```

```
    cv(2)=uv(locT)-Tmin   ! T>=Tmin
    cv(3)=qbarmax-qbar    ! qbarmax=qbar
  case(3)
    cv(1)=Tmax-uv(locT)   ! Tmax>=T
    cv(2)=uv(locT)-Tmin   ! T>=Tmin
    cv(3)=qbarmax-qbar    ! qbarmax>=qbar
  case(4)
    cv(1)=Tmin-uv(locT)   ! T=Tmin
    cv(2)=qbarmax-qbar    ! qbarmax>=qbar
end select
```

3. If the user is constructing the multi-phase problem from a single-phase one, it is critical not to forget that terminal boundary conditions no longer occur for `xbc(kf(1)+...)`. This is very easy to forget. The best policy is to express terminal values in terms of `xbc(kf(nph)+...)`. This is always correct, regardless of changes to the problem code. For `phiobj`, the code becomes

   ```
   rf=xbc(kf(nph)+locr)
   phi=-rf + Kddp*xbc(kf(3)+locddpint) ! ddpenMADS only on phase 3
   ```

   Note that because the `ddpenMADS` states only exist during phase three, the terminal value of the penalty integral is hard-wired to that phase.

4. A multi-phase problem introduces boundary conditions at the intermediate phase boundaries. In this case, we require that the plant states be continuous, and we need to initialize the penalty integral – `xv(locddpint)`, referred to in `phiobj`, above – to zero at the beginning of the third phase. The easiest way to handle continuity boundary conditions is to set up an index vector. Suppose that states 1, 3, and 4 are to be continuous across the phase 1/2 boundary. Define

   ```
   integer,parameter :: k134(3)=(/1,3,4/)
   ```

   and use it in `psibc` as follows:

   ```
   psi(kpsi+k134)=xbc(kf(1)+k134)-xbc(k0(2)+k134)
   kpsi=kpsi+3
   ```

   where `kpsi` at the beginning of the code fragment was the number of `psi` elements defined thus far. For our problem, the relevant lines of `psibc` are

   ```
   if(iebcflag.EQ.1)then
     iebc=0
     iebc(16:19)=1
     return
   endif
             ! initial conditions
   psi(1)=xbc(k0(1)+locr)-1
   psi(2)=xbc(k0(1)+locV)
   psi(3)=xbc(k0(1)+locm)-1
   kpsi=3                      ! 3
              ! continuity of plant traj from Tmax to qmax arcs
   ```

```
psi(kpsi+kMain)=xbc(kf(1)+kMain)-xbc(k0(2)+kMain)
kpsi=kpsi+nxp              ! 6
           ! continuity from qmax to singular
psi(kpsi+kMain)=xbc(kf(2)+kMain)-xbc(k0(3)+kMain)
kpsi=kpsi+nxp              ! 9
           ! initial condition for ddpenMADS integral
psi(kpsi+1)=xbc(k0(3)+locddpint)
kpsi=kpsi+1                ! 10
           ! continuity from singular to Tmin
psi(kpsi+kMain)=xbc(kf(3)+kMain)-xbc(k0(4)+kMain)
kpsi=kpsi+nxp              ! 13
           ! terminal conditions
psi(kpsi+1)=xbc(kf(nph)+locV)
psi(kpsi+2)=xbc(kf(nph)+locm)-mEmpty
kpsi=kpsi+2                ! 15
           ! bounds on taus
do k=1,nph
  psi(kpsi+k)=pv(loctau0+k)-taumin
enddo
kpsi=kpsi+nph              ! 19
```

Note, in the fragment above, the initialization of the penalty integral as `psi(10)` and imposition of $\tau_j \geq$ `taumin`, $j = 1,$`nph`. The index vector `kMain` has already been defined for use in time-scaling the plant equations of motion.

Although this multi-phase problem has a substantially more complicated temporal structure than the corresponding single phase one, it is easy to create an initial guess, starting from a compatible single phase solution. Assume that we are starting from the solution for `qbarmax` $= 9(10^{-4})$, from Figure 7, in the upper right corner. The first step in preparing the initial guess for for the multi-phase problem is to plot whatever variable (frequently a control) most clearly displays the switching structure, simply against index, e.g.,

```
S=importMADS('SinglePhase_premads.dat','SinglePhase_vout.dat',0);
plot(S.u{1});grid on;
```

The user then observes the index number(s) where the structure changes, and records them. For this problem, we have `bv=[19 62 77]` for four phases. The "`S`" structure and `bv` are saved together:

```
save SinglePhase S bv
```

and that `.mat` file is made available to a script like

```
load SinglePhase  % provide S and bv
tau=S.p;
bout=breaktrajMADS(S.x{1},S.u{1},bv,tau);  % Note that
          % construct dimensions for four-phase initial guess
P.nxv=[3 3 7 3];   % extra states for ddpenMADS in singular arc
P.nuv=[1 1 1 1];
P.nav=[2 3 3 2];   % see cineq...
P.kodev=[0 0 0 0]; % kodev only *needs* to be 0 during phase 3
P.nbc=19;
P.nph=4;
```

```
P.ndv=bout.ndv;
P.np=4;   % p vector will hold four taus.

randmag=0;
vin=[];
for k=1:P.nph   % Note adding extra states in phase 3
  vin=[vin;adduxMADS(bout.x{k},bout.u{k},P.nxv(k),P.nuv(k),randmag)];
end
vin=[vin;bout.tau];

save FourPhase_vin.dat vin -ascii -double
name='FourPhase_premads.dat';
writepremads(P,name)
```

Subsection 2.3.2 describes how to use `breaktrajMADS`, and`adduxMADS`, and we have already used the latter in setting up the initial guess for the single-phase problem with `ddpenMADS`. The function `breaktrajMADS` is used to partition the trajectory into subintervals delimited at the the integration intervals given in `bv`. The output of `breaktrajMADS`,"`bout`," includes the "`ndv`" vector, `bout.ndv`, and the vector of subinterval durations, `bout.tau`, in addition to the state and control trajectories for each subinterval. Note that the input guess has three states, so additional states are added only in the third phase, where `P.nxv` = 7.

We now return to Figure 11, to compare the single-phase and four-phase solutions. Before that, though, a housekeeping comment is needed regarding the falloff in T in the four-phase case (upper right plot) at the end of the max $-\bar{q}$ arc. The falloff in T lasts for one integration interval, and induces a corresponding violation of the $\bar{q} =$ `qbarmax` equality constraint for that phase. Why did this happen? It is the result of computing $\bar{q}$ in `cineq` as a function of `xj`, the "current" value of the state when `cineq` is called. Recall that for a trajectory with `nd` integration intervals, the state appears at `nd` + 1 instants. By imposing the $\bar{q}$ constraint on `xj` values, the terminal value of $\bar{q}$ was unconstrained. This situation can be treated in several ways:

- **Brute Force**
  The constraint could be doubled, imposed at both `xj` and at `xjp1`. This works, but is wasteful of computation, since it involves 2(`nd` − 1) redundant evaluations of the constraint.

- **Additional Boundary Condition**
  The missing constraint could be imposed in `psibc` via

  ```
  call calcqbar(xbc(kf(2)+kxp),qbar)
  psi(kpsi+1)=qbarmax-qbar  ! This element has iebcvec(kpsi+1)=0
  ```

  The is computationally efficient, but has a downside in that it requires that $\bar{q}$ be computed separately in `psibc`.

- **Compute the Constraint in xdot**
  This issue does not come up when $\bar{q}$ is computed in `xdot`, because it results in the constraint being imposed at the midpoints of the integration intervals: All state instants participate in the constraint. The downside, here, is that there is a vulnerability to control jitter on the active constraint arc. The jitter, as we've seen, can be treated using a penalty, as seen in this subsection. It is actually more effective, though, to constrain the constraint rate – in this case $d\bar{q}/dt$ to zero. This will be demonstrated in the next subsection.

- **Set the Problem Up Cleverly**
  We observed the temporal pattern of active constraints in the single-phase case, before breaking it up into the four-phase problem, so we knew that the first phase, for which `qbarmax` $\geq \bar{q}$, terminates with that constraint active, and that the state trajectory is continuous. Because of this, the first `qbarmax` $= \bar{q}$ constraint in the second phase is redundant, when $\bar{q}$ is computed using `xj`. We would do better in this case to use `xjp1`. The problem in Figure 11 can always be avoided by laying out the problem thoughtfully.

Having described how we set up the four-phase problem, and how we should have set it up, we return to Figure 11 to compare the single-phase and four-phase solutions. The most obvious differences appear in the thrust profiles in the Figure's right column. The jitter-suppressed "singular" arc in the four-phase case is nothing like that in the single-phase one. Not only is it, to say the least, difficult to assign it a plausible physical interpretation, but it is very short. It follows an elongated max $-\bar{q}$ arc that carrys T all the way back to its $T_{\max}$ boundary. This latter difference has a serious impact on the trajectory, increasing the time spent on the max $-\bar{q}$ constraint boundary by roughly 20%. This is visually evident in the Figure's left column, in which the four-phase $\bar{q}$ history, at the top, essentially lacks the $\bar{q}$ dropoff during the singular arc that was seen in the bottom, single-phase, plot.

Which of these trajectories is the (most) optimal one? It happens that the altitude gain for the single-phase problem is slightly less than .01% better than that for the four-phase one, but that's certainly not very significant. If it is important to know what the optimal solution is, without the distortions of jitter penalties, misbehaving singular arcs, or other numerical artifacts, then it is necessary to formulate and solve the problem as a variational optimal control problem. While this can be quite difficult – indeed, practically impractical for practical problems - the next Subsection will use MADS COV support routines to lay out an approximate variational solution.

Before leaving this Subsection, review what has been introduced here. This Subsection has focused on dealing with singular arcs using an heuristically motivated penalty function, implemented in `ddpenMADS` that targets large control accelerations accompanied by relatively small control rates. New material introduced in this Subsection included

1. application of `ddpenMADS` to problems with an active state constraint arc implemented by computing the constraint function in `xdot`. It was seen that, while the measure nowhere produces perfect results, it benefits from gradually "walking the solution in;" that is, solving a sequence of problems with increasingly stringent constraint settings, using each solution as the initial guess for the next.

2. The details of setting up a multiple-phase problem from a single-phase one were demonstrated, and various subtleties were discussed for posing state constraints in a multi-phase setting that uses `cineq` to compute the constraint function.

References

1. Gill, P. E., Murray, W. M., and Saunders, M. A., "User's Manual for SNOPT Version 7: Software for Large-Scale Nonlinear Programming," University of California, San Diego, April 2007.

2. Hascoët, L. and Pascual, V., "The Tapenade Automatic Differentiation tool: Principles, Model, and Specification," ACM Transactions On Mathematical Software, Vol. 39, No. 3, 2013.

3. Huntington, G. T., Benson, D. A., and Rao, A. V., "A Comparison of Accuracy and Computational Efficiency of Three Pseudospectral Methods," AIAA Paper 2007-6405, 2007 AIAA Guidance, Navigation, and Control Conference, Hilton Head, SC, August 2007.

4. Neidinger, R., "Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming," SIAM Review, Vol. 52, No. 3, 2010.

5. Bryson, A. E., and Ho, Y.-C., Applied Optimal Control, Hemisphere Publishing, New York, 1975.

6. McShane, E., "On Multipliers for Lagrange Problems," American J. Math., 1939, Vol. 61.

7. Pontryagin, L., The Mathematical Theory of Optimal Processes, Wiley, New York, 1962.

8. Seywald, H. and Cliff, E. M., "Goddard Problem in Presence of a Dynamic Pressure Limit," AIAA Journal of Guidance, Control, and Dynamics, Vol. 16, No. 4, 1993, pp. 776-781.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 10 - 2014	Technical Memorandum	

4. TITLE AND SUBTITLE

MADS Users' Guide

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Moerder, Daniel D.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

473452.02.07.03.02.01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

L-20256

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)

NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

NASA/TM-2014-218532

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified
Subject Category 08
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES

14. ABSTRACT

MADS (Minimization Assistant for Dynamical Systems) is a trajectory optimization code in which a user-specified performance measure is directly minimized, subject to constraints placed on a low-order discretization of user-supplied plant ordinary differential equations. This document describes the mathematical formulation of the set of trajectory optimization problems for which MADS is suitable, and describes the user interface. Usage examples are provided.

15. SUBJECT TERMS

Guide; MADS; Users

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	63	(443) 757-5802