R ecursive
I ntegration
O ptimal
T rajectory
S olver
95

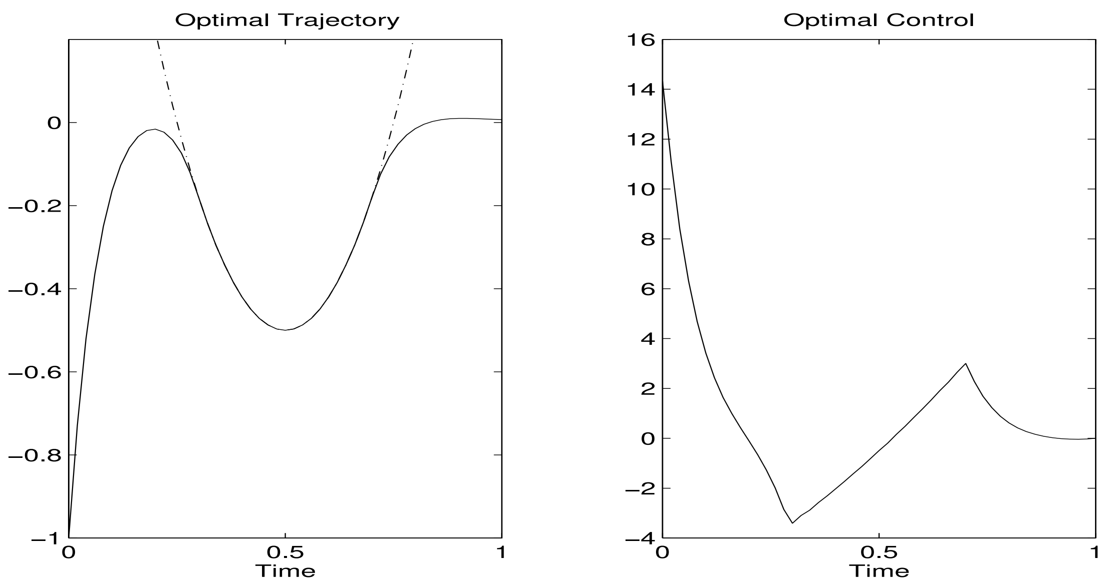
A Matlab Toolbox for Solving Optimal Control Problems

Version 1.0 for Windows

May 1997

by

A. Schwartz, E. Polak and Y. Chen



Conditions for Use of RIOTS_95™

To use any part of the RIOTS_95 toolbox the user must agree to the following conditions:

1. The RIOTS_95 toolbox for solving optimal control problem is distributed for sale according to the RIOTS_95 license agreement. Use of RIOTS_95 is limited to those users covered under the purchase agreement.
2. This software is distributed without any performance or accuracy guarantees. It is solely the reponsibility of the user to determine the accuracy and validity of the results obtained using RIOTS.
3. RIOTS_95, the RIOTS_95 user's manual, or any portion of either may not be distributed to third parties. Interested parties must obtain RIOTS_95 directly from Adam Schwartz or his associates.
4. Any modifications to the programs in RIOTS_95 must be communicated to Adam Schwartz. Modified programs will remain the sole property of Adam Schwartz.
5. Due acknowledgment must be made of the use of RIOTS_95 in any resarch reports or publications. Whenever such reports are released for public access, a copy should be forwarded to Adam Schwartz.
6. RIOTS_95, or any portion of the software in RIOTS_95, cannot be used as part of any other software without the explicit consent of Adam Schwartz.
7. RIOTS_95 has been thoroughly debugged and there are no memory leaks or memory errors in the code. However, it is possible for the user's code to create a memory error through faulty use of pointers or incorrectly allocated memory arrays.

RIOTS_95™: A Matlab Toolbox for Solving Optimal Control Problems, Version 1.0

Enquiries should be directed to:

Dr. Adam L. Schwartz

333 Quinterra Ln.
Danville, CA 94526
USA
E-mail : adams@eecs.berkeley.edu
Phone : 510-837-8248

A self-extracting RIOTS_95 educational/demonstration kit is available from the following web sites:

http://www.accesscom.com/~adam/RIOTS
http://www.shuya.home.ml.org/RIOTS_95
http://www.crosswinds.net/singapore/~yqchen/riots.html
http://www.cadcam.nus.sg/~elecyq
http://www.ee.nus.sg/~yangquan/riots.html

Abstract

RIOTS_95: A Matlab Toolbox for Solving Optimal Control Problems

by

A. L. Schwartz and E. Polak

This manual describes the use and operation of RIOTS_95. RIOTS_95 is a group of programs and utilities, written mostly in C and designed as a toolbox for Matlab, that provides an interactive environment for solving a very broad class of optimal control problems. RIOTS_95 comes pre-compiled for use with the Windows3.1, Windows95 or WindowsNT operating systems.

The numerical methods used by RIOTS_95 are supported by the theory in [1-4] which uses the approach of consistent approximations as defined by Polak [5]. In this approach, a solution is obtained as an accumulation point of the solutions to a sequence of discrete-time optimal control problems that are, in a specific sense, consistent approximations to the original continuous-time, optimal control problem. The discrete-time optimal control problems are constructed by discretizing the system dynamics with one of four fixed step-size Runge-Kutta integration methods[1] and by representing the controls as finite-dimensional B-splines. The integration proceeds on a (possibly non-uniform) mesh that specifies the spline breakpoints. The solution obtained for one such discretized problem can be used to select a new integration mesh upon which the optimal control problem can be re-discretized to produce a new discrete-time problem that more accurately approximates the original problem. In practice, only a few such re-discretizations need to be performed to achieve an acceptable solution.

RIOTS_95 provides three different programs that perform the discretization and solve the finite-dimensional discrete-time problem. The appropriate choice of optimization program depends on the type of problem being solved as well as the number of points in the integration mesh. In addition to these optimization programs, RIOTS_95 also includes other utility programs that are used to refine the discretization mesh, to compute estimates of integration errors, to compute estimates for the error between the numerically obtained solution and the optimal control and to deal with oscillations that arise in the numerical solution of singular optimal control problems.

---

[1]RIOTS_95 also includes a variable step-size integration routine and a discrete-time solver.

Table of Contents

1. PURPOSE

This chapter describes the implementation of a Matlab[2] toolbox called RIOTS_95 for solving optimal control problems. The name RIOTS stands for ‘‘Recursive[3] Integration Optimal Trajectory Solver.’’ This name highlights the fact that the function values and gradients needed to find the optimal solutions are computed by forward and backward integration of certain differential equations.

RIOTS_95 is a collection of programs that are callable from the mathematical simulation program Matlab. Most of these programs are written in either C (and linked into Matlab using Matlab’s MEX facility) or Matlab’s M-script language. All of Matlab’s functionality, including command line execution and data entry and data plotting, are available to the user. The following is a list of some of the main features of RIOTS_95.

- Solves a very large class of finite-time optimal controls problems that includes: trajectory and endpoint constraints, control bounds, variable initial conditions (free final time problems), and problems with integral and/or endpoint cost functions.
- System functions can be supplied by the user as either object code or M-files.
- System dynamics can be integrated with fixed step-size Runge-Kutta integration, a discrete-time solver or a variable step-size method. The software automatically computes gradients for all functions with respect to the controls and any free initial conditions. These gradients are computed exactly for the fixed step-size routines.
- The controls are represented as splines. This allows for a high degree of function approximation accuracy without requiring a large number of control parameters.
- The optimization routines use a coordinate transformation that creates an orthonormal basis for the spline subspace of controls. The use of an orthogonal basis can results in a significant reduction in the number of iterations required to solve a problem and an increase in the solution accuracy. It also makes the termination tests independent of the discretization level.
- There are three main optimization routines, each suited for different levels of generality of the optimal control problem. The most general is based on sequential quadratic programming methods. The most restrictive, but most efficient for large discretization levels, is based on the projected descent method. A third algorithm uses the projected descent method in conjunction with an augmented Lagrangian formulation.
- There are programs that provide estimates of the integration error for the fixed step-size Runge-Kutta methods and estimates of the error of the numerically obtained optimal control.
- The main optimization routine includes a special feature for dealing with singular optimal control problems.
- The algorithms are all founded on rigorous convergence theory.

In addition to being able to accurately and efficiently solve a broad class of optimal control problems, RIOTS_95 is designed in a modular, toolbox fashion that allows the user to experiment with the optimal control algorithms and construct new algorithms. The programs **outer** and **aug_lagrng**,

---

[2]Matlab is a registered trademark of Mathworks, Inc. Matlab version 4.2c with the Spline toolbox is required.

[3]Iterative is more accurate but would not lead to a nice acronym.

described later, are examples of this toolbox approach to constructing algorithms.

RIOTS_95 is a collection of several different programs (including a program which is, itself, called **riots**) that fall into roughly three categories: integration/simulation routines, optimization routines, and utility programs. Of these programs, the ones available to the user are listed in the following table,

Simulation Routines	Optimization Routines	Utility Programs
simulate	**riots**	**control_error**
check_deriv	**pdmin**	**distribute**
check_grad	**aug_lagrng**	**est_error**
eval_fnc	**outer**	**make_spline**
		transform

Several of the programs in RIOTS_95 require functions that are available in the Matlab Spline toolbox. In addition to these programs, the user must also supply a set of routines that describe the optimal control problem which must be solved. Several example optimal control problems come supplied with RIOTS_95. Finally, there is a Matlab script called **RIOTS_demo** which provides a demonstration of some of the main features of RIOTS_95. To use the demonstration, perform the following steps:

Step 1: Follow the directions in §8 on compiling and linking RIOTS_95. Also, compile the sample systems rayleigh.c, bang.c and goddard.c that come supplied with RIOTS_95.

Step 2: Start Matlab from within the 'RIOTS/systems' directory.

Step 3: Add the RIOTS_95 directory to Matlab's path by typing at the Matlab prompt,

```
>> path(path,'full_path_name_for_RIOTS')
>> RIOTS_demo
```

Limitations. This is the first version of RIOTS_95. As it stands, there are a few significant limitations on the type of problems which can be solved by RIOTS_95:

1. Problems with inequality state constraints that require a very high level of discretization cannot be solved by RIOTS_95. Also, the computation of gradients for trajectory constraints is not handled as efficiently as it could be.
2. Problems that have highly unstable, nonlinear dynamics may require a very good initial guess for the solution in order to be solved by RIOTS_95.
3. General constraints on the controls that do not involve state variables are not handled efficiently: adjoints are computed but not used.
4. RIOTS_95 does not allow delays in the systems dynamics (although Padé approximations can be used).
5. Numerical methods for solving optimal control problems have not reached the stage that, say, methods for solving differential equations have reached. Solving an optimal control problem can, depending on the difficulty of the problem, require significant user involvement in the solution process. This sometimes requires the user to understand the theory of optimal control, optimization and/or numerical approximation methods.

Conventions. This manual assumes familiarity with Matlab. The following conventions are used throughout this manual.

- Program names and computer commands are indicated in **bold** typeface.
- User input is indicated in `Courier` typeface.
- Optional program arguments are listed in brackets. The default value for any optional argument can be specified using `[]`.
- Optional program arguments at the end of an argument list can be omitted in which case these arguments take on their default values.
- Typing a function's name without arguments shows the calling syntax for that function. Help can be obtained for M-file programs by typing `help` followed by the function name at Matlab's prompt. Typing `help RIOTS` produces a list of the programs in RIOTS_95.
- The machine precision is denoted by $\varepsilon_{\text{mach}}$.

2. PROBLEM DESCRIPTION

RIOTS_95 is designed to solve optimal control problems of the form[4]

$$\textbf{OCP} \qquad \underset{(u,\xi)\,\in\, L_\infty^m[a,b]\times\mathbb{R}^n}{\text{minimize}} \left\{ f(u,\xi) \doteq g_o(\xi, x(b)) + \int_a^b l_o(t,x,u)dt \;\right\}$$

$$\begin{aligned}
\text{subject to:}\quad & \dot{x} = h(t,x,u)\ ,\ \ x(a) = \xi\ ,\ t \in [a,b]\ ,\\
& u^j_{\min}(t) \le u^j(t) \le u^j_{\max}(t)\ ,\ \ j = 1,\ldots,m\ ,\ \ t \in [a,b]\ ,\\
& \xi^j_{\min} \le \xi^j \le \xi^j_{\max}\ ,\ \ j = 1,\ldots,n\ ,\\
& l^\nu_{ti}(t, x(t), u(t)) \le 0\ ,\ \ \nu \in \mathbf{q}_{ti}\ ,\ t \in [a,b]\ ,\\
& g^\nu_{ei}(\xi, x(b)) \le 0\ ,\ \ \nu \in \mathbf{q}_{ei}\ ,\\
& g^\nu_{ee}(\xi, x(b)) = 0\ ,\ \ \nu \in \mathbf{q}_{ee}\ ,
\end{aligned}$$

where $x(t) \in \mathbb{R}^n$, $u(t) \in \mathbb{R}^m$, $g : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, $l : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$, $h : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ and we have used the notation $\mathbf{q} \doteq \{\, 1,\ldots,q \,\}$ and $L_\infty^m[a,b]$ is the space of Lebesgue measurable, essentially bounded functions $[a,b] \to \mathbb{R}^m$. The functions in **OCP** can also depend upon parameters which are passed from Matlab at execution time using **get_flags** (described in §4).

The subscripts o, ti, ei, and ee on the functions $g(\cdot,\cdot)$ and $l(\cdot,\cdot,\cdot)$ stand for, respectively, ''objective function'', ''trajectory constraint'', ''endpoint inequality constraint'' and ''endpoint equality constraint''. The subscripts for $g(\cdot,\cdot)$ and $l(\cdot,\cdot,\cdot)$ are omitted when all functions are being considered without regard to

[4]Not all of the optimization routines in RIOTS_95 can handle the full generality of problem **OCP**.

the subscript. The functions in the description of problem **OCP**, and the derivatives of these functions[5], must be supplied by the user as either object code or as M-files. The bounds on the components of ξ and u are specified on the Matlab command line at run-time.

The optimal control problem **OCP** allows optimization over both the control u and one or more of the initial states ξ. To be concise, we will define the variable

$$\eta = (u, \xi) \in H_2 \doteq L_\infty^m[a,b] \times \mathbb{R}^n .$$

With this notation, we can write, for example, $f(\eta)$ instead of $f(\xi, u)$. We define the inner product on H_2 as

$$\langle \eta_1, \eta_2 \rangle_{H_2} \doteq \langle u_1, u_2 \rangle_{L_2} + \langle \xi_1, \xi_2 \rangle .$$

The norm corresponding to this inner product is given by $\|\eta\|_{H_2} = \langle \eta, \eta \rangle_{H_2}^{1/2}$. Note that H_2 is a pre-Hilbert space.

Transcription for Free Final Time Problems.

Problem **OCP** is a fixed final time optimal control problem. However, free final time problems are easily incorporated into the form of **OCP** by augmenting the system dynamics with two additional states (one additional state for autonomous problems). The idea is to specify a nominal time interval, $[a, b]$, for the problem and to use a scale factor, adjustable by the optimization procedure, to scale the system dynamics and hence, in effect, scale the duration of the time interval. This scale factor, and the scaled time, are represented by the extra states. Then RIOTS_95 can minimize over the initial value of the extra states to adjust the scaling. For example, the free final time optimal control problem

$$\min_{u,T} \ \tilde{g}(T, y(T)) + \int_a^{a+T} \tilde{l}(t, y, u)dt$$

$$\text{subject to } \dot{y} = \tilde{h}(t, y, u) \ , \ y(a) = \zeta \ , \ t \in [a, a+T] \ ,$$

can, with an augmented state vector $x \doteq (y, x^{n-1}, x^n)$, be converted into the equivalent fixed final time optimal control problem

$$\min_{u,\xi^n} \ g(\xi, x(b)) + \int_a^b l(t, x, u)dt$$

$$\text{subject to } \dot{x} = h(t, x, u) \doteq \begin{pmatrix} x^n \tilde{h}(x^{n-1}, y, u) \\ x^n \\ 0 \end{pmatrix} \ , \ x(a) = \xi = \begin{pmatrix} \zeta \\ a \\ \xi^n \end{pmatrix}, \ t \in [a, b] \ ,$$

where y is the first $n-2$ components of x, $g(\xi, x(b)) \doteq \tilde{g}(a + T\xi^n, y(b))$, $l(t, x, u) \doteq \tilde{l}(x^{n-1}, y, u)$ and $b = a + T$. Endpoint and trajectory constraints can be handled in the same way. The quantity $T = b - a$ is the nominal trajectory duration. In this transcription, x^{n-1} plays the role of time and ξ^n is the *duration scale factor*, so named because $T\xi^n$ is the effective duration of the trajectories for the scaled dynamics. Thus, for any $t \in [a, b]$, $x^n(t) = \xi^n$, $x^{n-1}(t) = a + (t - a)\xi^n$ and the solution, t_f, for the final time is

[5] If the user does not supply derivatives, the problem can still be solved using **riots** with finite-difference computation of the gradients.

$t_f = x^{n-1}(b) = a + (b-a)\xi^n$. Thus, the optimal duration is $T^* = t_f - a = (b-a)\xi^n = T\xi^n$. If $a = 0$ and $b = 1$, then $t_f = T^* = \xi^n$. The main disadvantage to this transcription is that it converts linear systems into nonlinear systems.

For autonomous systems, the extra variable x^{n-1} is not needed. Note that, it is possible, even for non-autonomous systems, to transcribe minimum time problems into the form of **OCP** using only one extra state variable. However, this would require functions like $h(t, x, u) = \tilde{h}(tx^n, y, u)$. Since RIOTS_95 does not expect the user to supply derivatives with respect to the t argument it can not properly compute derivatives for such functions. Hence, in the current implementation of RIOTS_95, the extra variable x^{n-1} is needed when transcribing non-autonomous, free final time problems.

Trajectory constraints.

The definition of problem **OCP** allows trajectory constraints of the form $l_{ti}(t, x, u) \leq 0$ to be handled directly. However, constraints of this form are quite burdensome computationally. This is mainly due to the fact that a separate gradient calculation must be performed for each point at which the trajectory constraint is evaluated.

At the expense of increased constraint violation, reduced solution accuracy and an increase in the number of iterations required to obtain solutions, trajectory constraints can be converted into endpoint constraints which are computationally much easier to handle. This is accomplished as follows. The system is augmented with an extra state variable x^{n+1} with

$$\dot{x}^{n+1}(t) = \mu \max\{\, 0, l_{ti}(t, x(t), u(t)) \,\}^2 \;,\;\; x^{n+1}(a) = 0 \;,$$

where $\mu > 0$ is a positive scalar. The right-hand side is squared so that it is differentiable with respect to x and u. Then it is clear that either of the endpoint constraints

$$g_{ei}(\xi, x(b)) \doteq x^{n+1}(b) \leq 0 \qquad \text{or} \qquad g_{ee}(\xi, x(b)) \doteq x^{n+1}(b) = 0$$

is satisfied if and only if the original trajectory constraint is satisfied. In practice, the accuracy to which **OCP** can be solved with these endpoint constraints is quite limited because these endpoint constraints do not satisfy the standard constraint qualification (described in the §4). This difficulty can be circumvented by eliminating the constraints altogether and, instead, adding to the objective function the penalty term $g_o(\xi, x(b)) \doteq x^{n+1}(b)$ where now μ serves as a penalty parameter. However, in this approach, μ must now be a large positive number and this will adversely affect the conditioning of the problem. Each of these possibilities is implemented in 'obstacle.c' for problem Obstacle (see Appendix B).

Continuum Objective Functions and Minimax Problems.

Objective functions of the form

$$\min_{u} \max_{t \in [a,b]} l(t, x(t), u(t))$$

can be converted into the form used in problem **OCP** by augmenting the state vector with an additional state, w, such that

$$\dot{w} = 0 \;;\;\; w(0) = \xi^{n+1}$$

and forming the equivalent trajectory constrained problem

$$\min_{(u,\xi^{n+1})} \xi^{n+1}$$

subject to

$$l(t,x(t),u(t)) - \xi^{n+1} \le 0\ ,\ \ t \in [a,b]\ .$$

A similar transcription works for standard min-max objective functions of the form

$$\min_u \max_{\nu \in \mathbf{q}_o} g^\nu(u,\xi) + \int_a^b l^\nu(t,x(t),u(t)dt\ .$$

In this case, an equivalent endpoint constrained problem with a single objective function,

$$\min_{u,\xi^{n+1}} \xi^{n+1}$$

subject to

$$\tilde{g}^{\,\nu}(u,\xi) - \xi^{n+1} \le 0\ ,\ \ \nu \in \mathbf{q}_o$$

is formed by using the augmented state vector (x,w,z) with

$$\dot{w} = 0\ ,\ \ w(0) = \xi^{n+1}$$

$$\dot{z}^\nu = l^\nu(t,x(t),u(t))\ ,\ \ z^\nu(0) = 0\ ,\ \ \nu \in \mathbf{q}_o\ ,$$

and defining

$$\tilde{g}^{\,\nu}(u,\xi) \doteq g^\nu(u,\xi) + z^\nu(b)\ .$$

3. USING RIOTS_95

This section provides some examples of how to simulate systems and solve optimal control problems with the RIOTS_95 toolbox. Detailed descriptions of all required user-functions, simulation routines, optimization programs and utility programs are given in subsequent sections. These programs are all callable from within Matlab once Matlab's path is set to include the directory containing RIOTS_95. The Matlab command

```
>> path(path,'full_path_name_for_RIOTS')
>> RIOTS_demo
```

should be used for this purpose. Refer to the §8, ‘‘Compiling and Linking RIOTS_95’’, for details on how to install RIOTS_95.

RIOTS_95 provides approximate solutions of continuous time optimal control problems by solving discretized ‘‘approximating’’ problems. These approximating problems are obtained by *(i)* numerically integrating the continuous time system dynamics with one of four Runge-Kutta integration methods[6] and *(ii)* restricting the space of allowable controls to finite-dimensional subspaces of splines. In this way, the approximating problems can by solved using standard mathematical programming techniques to optimize over the spline coefficients and any free intial conditions. It is not important for the user of RIOTS_95 to

[6]RIOTS_95 also includes a discrete-time system solver and a variable step-size integration routine.

understand the discretization procedure or splines.

The accuracy of the solutions obtained in this manner depends on several factors which include:

(1) The accuracy of the integration scheme (which depends on the order of the integration scheme and the selection of the integration mesh).

(2) How well elements of the spline subspace can approximate solutions of the original, infinite-dimensional problem (this depends on the order and knot sequence of the splines and on the smoothness of the optimal control).

(3) How accurately the approximating problems are solved by the underlying mathematical programming algorithm.

The allowable spline orders are related to the particular integration method used (see description of **simulate** in §5). For problems that have smooth optimal controls, higher order splines will provide solutions with higher accuracy. Smoothness is not, however, typical of optimal controls for problems with control and/or trajectory constraints. In general, the spline knot sequence is constructed from the integration mesh

$$\mathbf{t}_N \doteq \{ t_k \}_{k=1}^{N+1} ,$$

which also specifies the spline breakpoints. The subscript N, referred to as the discretization level, indicates that there are N integration steps and $N+1$ spline breakpoints. Each spline is determined from the knot sequence and its coefficients. For a spline of order ρ, each control input requires $N+\rho-1$ coefficients and these coefficients are stored as *row* vectors. Thus, a system with m inputs will be stored in a ‘‘short-fat’’ matrix with m rows and $N+\rho-1$ columns. More details about splines are given in the next section.

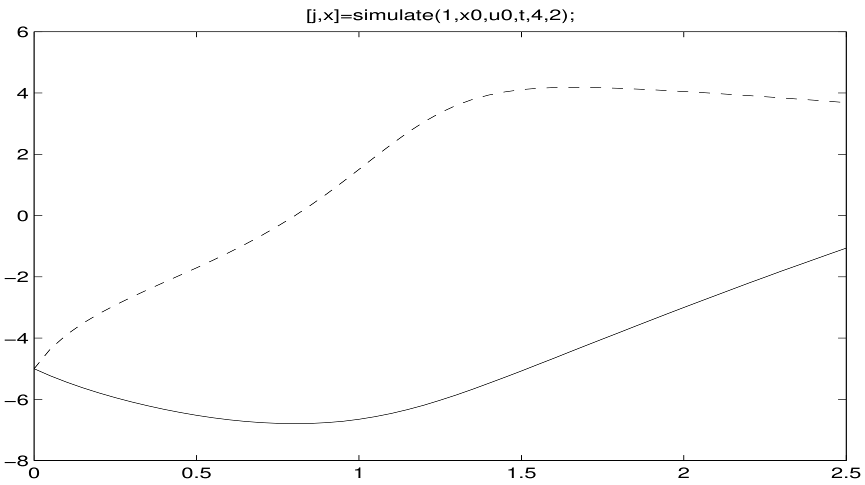
Typically, we use the Matlab variable `u` to store the spline coefficients. The system trajectories computed by integrating the system dynamics are stored in the variable `x`. Like `u`, `x` is a ‘‘short-fat’’ matrix with n rows and $N+1$ columns. Thus, for example, `x(:,k)` is the computed value of $x(t_k)$. Other quantities, such as gradients and adjoints, are also stored as ‘‘short-fat’’ matrices.

The following sample sessions with RIOTS_95 solve a few of the sample optimal control problems that are supplied with RIOTS_95 as examples. Appendix B provides a description of these problems and the C-code implementations are included in the ‘RIOTS/systems’ sub-directory.

Session 1 (unconstrained problem). In this session we compute a solution to the unconstrained nonlinear Problem Rayleigh. This system has two states and one input. We start by defining the initial conditions and a uniform integration mesh over the time interval $[0, 2.5]$ with a discretization level of $N = 50$ intervals.

We can take a look at the solution trajectories by simulating this system with some initial control. We will specify an arbitrary piecewise linear (order $\rho = 2$) spline by using $N + \rho - 1 = N + 1$ coefficients and perform a simulation by calling **simulate**.

```
>> N=50;
>> x0=[-5;-5];                          % Initial conditions
>> t=[0:2.5/50:2.5];                    % Uniform integration mesh
>> u0=zeros(1,N+1);                     % Spline with all coeff's zero.
>> [j,x]=simulate(1,x0,u0,t,4,2);
>> plot(t,x)
```



Next, we find an approximate solution to the Problem Rayleigh, which will be the same type of spline as `u0`, by using either **riots** or **pdmin**.

```
>> [u1,x1,f1]=riots(x0,u0,t,[],[],[],100,4);
>> [u1,x1,f1]=pdmin(x0,u0,t,[],[],[],100,4);
```

The first three input arguments are the initial conditions, initial guess for the optimal control, and the integration mesh. The next three inputs are empty brackets indicating default values which, in this case, specify that there are no control lower bounds, no control upper bounds, and no systems parameters. The last two inputs specify that a maximum of 100 iterations are to be allowed and that integration routine 4 (which is a fourth order Runge-Kutta method) should be used. The outputs are the control solution, the trajectory solution, and the value of the objective function.

The displayed output for **pdmin** is shown below. The displayed output for **riots** depends on the mathematical programming algorithm with which it is linked (see description of **riots** in §6).

```
This is a nonlinear system with 2 states, 1 inputs and 0 parameters,
     1 objective function,
     0 nonlinear and 0 linear trajectory constraints,
     0 nonlinear and 0 linear endpoint inequality constraints,
     0 nonlinear and 0 linear endpoint equality constraints.
Initial Scale factor = 0.02937
Method = L-BFGS.
Quadratic fitting off.
Completed 1    pdmin iter ; step = +1.67e+00 (k= -1), ||free_grad|| = 1.47e-01, FFF, cost = 34.40807327949193
Completed 2    pdmin iters; step = +4.63e+00 (k= -3), ||free_grad|| = 1.01e-01, FFF, cost = 31.33402612711411
Completed 3    pdmin iters; step = +2.78e+00 (k= -2), ||free_grad|| = 5.26e-02, FFF, cost = 29.78609937166251
Completed 4    pdmin iters; step = +1.67e+00 (k= -1), ||free_grad|| = 2.25e-02, FFF, cost = 29.30022802876513
Completed 5    pdmin iters; step = +1.00e+00 (k= +0), ||free_grad|| = 9.03e-03, FFF, cost = 29.22362561134763
Completed 6    pdmin iters; step = +1.67e+00 (k= -1), ||free_grad|| = 2.61e-03, FFF, cost = 29.20263210973429
Completed 7    pdmin iters; step = +1.00e+00 (k= +0), ||free_grad|| = 5.06e-04, FFF, cost = 29.20066785222028
Completed 8    pdmin iters; step = +1.00e+00 (k= +0), ||free_grad|| = 1.80e-04, FFF, cost = 29.20060360626269
Completed 9    pdmin iters; step = +1.00e+00 (k= +0), ||free_grad|| = 1.86e-05, FFF, cost = 29.20059986273411
Completed 10   pdmin iters; step = +1.00e+00 (k= +0), ||free_grad|| = 5.94e-06, FFF, cost = 29.20059981048738
Completed 11   pdmin iters; step = +1.67e+00 (k= -1), ||free_grad|| = 2.07e-06, FFF, cost = 29.20059980021174
Completed 12   pdmin iters; step = +1.00e+00 (k= +0), ||free_grad|| = 1.57e-07, FFF, cost = 29.20059979946436
Completed 13   pdmin iters; step = +1.00e+00 (k= +0), ||free_grad|| = 5.18e-08, FFF, cost = 29.20059979945842
Completed 14   pdmin iters; step = +1.00e+00 (k= +0), ||free_grad|| = 1.16e-08, FFF, cost = 29.20059979945757
Completed 15   pdmin iters; step = +1.00e+00 (k= +0), ||free_grad|| = 3.20e-10, TTF, cost = 29.20059979945753
Completed 16   pdmin iters; step = +6.00e-01 (k= +1), ||free_grad|| = 1.66e-10, TTT, cost = 29.20059979945752

Finished pdmin loop on the 16-th iteration.
Normal termination test satisfied.
```

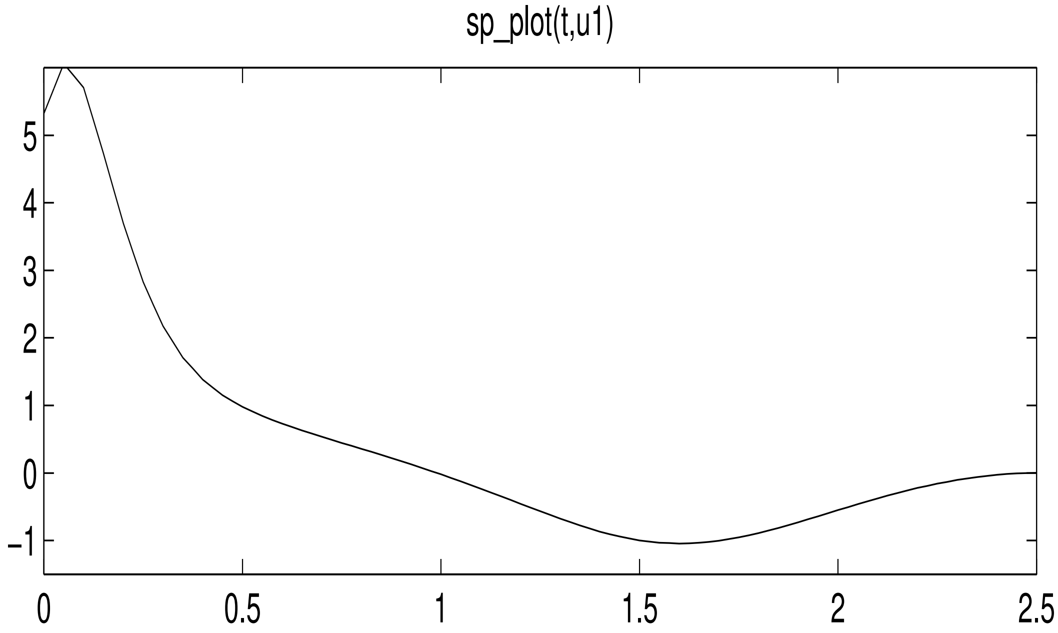
The column labeled `||free_grad||` gives the value of $\|\nabla f(\eta)\|_{H_2}$, the norm of the gradient of the objective function. For problems with bounds on the free initial conditions and/or controls, this norm is restricted to the subspace where the bounds are not active. For problems without state constraints, $\|\nabla f(\eta)\|_{H_2}$ goes to zero as a local minimizer is approached. The column with three letters, each a `T` or `F`, indicates which of the three normal termination criterion (see description of **pdmin** in §6) are satisfied. For problems with control or initial condition bounds there are four termination criteria.

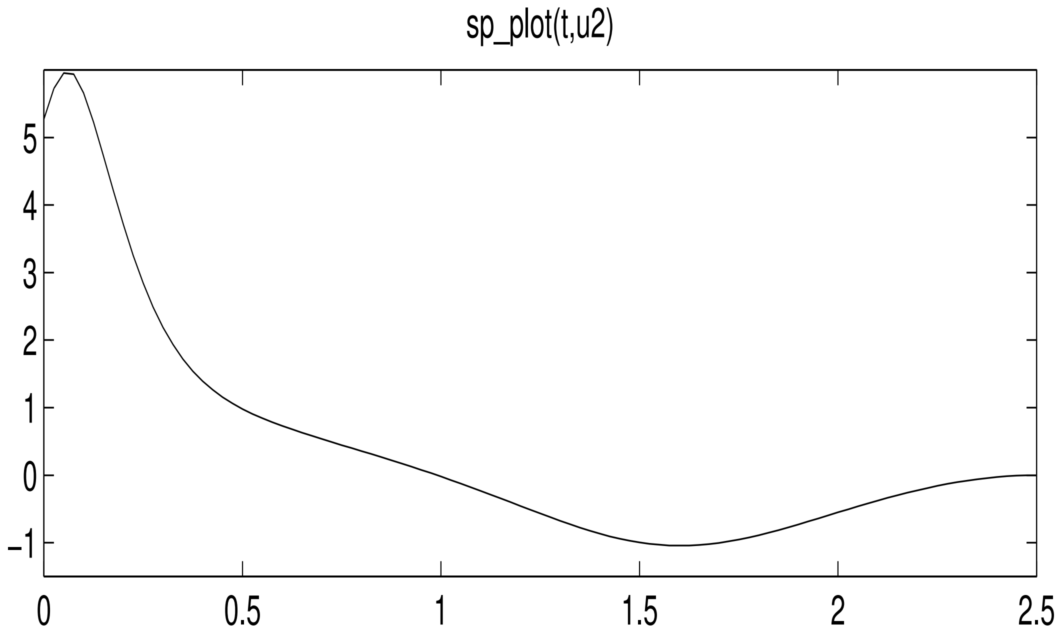
We can also solve this problem with quadratic splines ($\rho = 3$) by using $N + \rho - 1 = N + 2$ spline coefficients.

```
>> u0=zeros(1,N+2);
>> [u2,x2,f2]=pdmin(x0,u0,t,[],[],[],100,4);
```

We can view the control solutions using **sp_plot** which plots spline functions. The trajectory solutions can be viewed using **plot** or **sp_plot**.

```
>> sp_plot(t,u1)                    % Plot linear spline solution
>> sp_plot(t,u2)                    % Plot quadratic spline solution
```


sp_plot(t,u1)
5
4
3
2
1
0
−1
0
0.5
1
1.5
2
2.5


sp_plot(t,u2)
5
4
3
2
1
0
−1
0
0.5
1
1.5
2
2.5

Session 2 (problem with endpoint constraint). The user-defined functions for Problem Rayleigh, solved in session 1, are written so that it will include the endpoint constraint $x_1(2.5) = 0$ if there is a global Matlab variable called FLAGS set to the value of 1 (see **get_flags** in §4). To solve this problem with the endpoint constraint we can use either **riots** or **aug_lagrng**. We must clear **simulate** before re-solving so that the variable FLAGS gets read.

```
>> global FLAGS
>> FLAGS = 1;
>> clear simulate             % Reset simulate so the it will check for FLAGS
>> simulate(0,[]);            % Initialize
Loaded 1 flag.
Rayleigh
This is a nonlinear system with 2 states, 1 inputs and 0 parameters,
     1 objective function,
     0 nonlinear and 0 linear trajectory constraints,
     0 nonlinear and 0 linear endpoint inequality constraints,
     0 nonlinear and 1 linear endpoint equality constraints.
```

The output displayed above shows that one flag has been read from the Matlab workspace. The next two lines are messages produced by the user-supplied routines. The last set of data shows the value of the system information (see discussion of neq[] in the description of **init**, §4, and also **simulate**, §5). Since this problem has a state constraint, we can use either **aug_lagrng** or **riots** to solve it.

```
>> x0=[-5;-5];
>> u0=zeros(1,51);
>> t=[0:2.5/50:2.5];
>> u=aug_lagrng(x0,u0,t,[],[],[],100,5,4);

Finished pdmin loop on the 2-nd iteration.
Step size too small.

Completed 1  Outer loop iterations.
Multipliers          : -2.81973
Penalties            : 10
Constraint Violations: 1.90255
Norm of unconstrained portion of Lagrangian gradient = 0.00646352
Rayleigh

Finished pdmin loop on the 15-th iteration.
Normal termination test satisfied.

Completed 2  Outer loop iterations.
Multipliers          : -0.658243
Penalties            : 10
Constraint Violations: 0.000483281
Norm of unconstrained portion of Lagrangian gradient = 0.000206008
Rayleigh

Finished pdmin loop on the 8-th iteration.
Normal termination test satisfied.
```

```
Completed 3  Outer loop iterations.
Multipliers          : -0.653453
Penalties            : 10
Constraint Violations: -7.91394e-06
Norm of unconstrained portion of Lagrangian gradient = 1.37231e-06
Rayleigh

Finished pdmin loop on the 7-th iteration.
Normal termination test satisfied.

Completed 4  Outer loop iterations.
Multipliers          : -0.653431
Penalties            : 10
Constraint Violations: -8.6292e-07
Norm of unconstrained portion of Lagrangian gradient = 2.19012e-07
Objective Value      : 29.8635
Normal termination of outer loop.
```

The displayed output reports that, at the current solution, the objective value is 29.8635 and the endpoint constraint is being violated by -8.63×10^{-6}. There is some error in these values due to the integration error of the fixed step-size integration routines. We can get a more accurate measure by using the variable step-size integration routine to simulate the system with the control solution `u`:

```
>> simulate(1,x0,u,t,5,0);          % Simulate system using LSODA
>> simulate(2,1,1)                  % Evaluate the objective function

ans =

   29.8648

>> simulate(2,2,1)                  % Evaluate the endpoint constraint

ans =

   5.3852e-06
```

The integration was performed with the default value of $1e-8$ for both the relative and absolute local integration error tolerances. So the reported values are fairly accurate.

Session 3 (Problem with control bounds and free final time). This session demonstrates the transcription, explained in §2, of a free final time problem into a fixed final time problem. The transcribed problem has bounds on the control and free initial states. Also, **distribute** (see §7) is used to improve integration mesh after an initial solution is found. A more accurate solution will then be computed by re-solving the problem on the new mesh.

The original problem, Problem Bang, is a minimum-time problem with three states and one input. This problem is converted into a fixed final time problem using the transcription described in §2. Only one extra state variable was needed since the problem has time-independent (autonomous) dynamics. The augmented problem is implemented in the file 'bang.c'. First we will define the integration mesh and then the initial conditions.

```
>> N = 20;                          % Discretization level
>> T = 10;                          % Nominal final time
>> t=[0:T/N:T];                     % Nominal time interval for maneuver
```

The nominal time interval is of duration `T`. Next, we specify a value for ξ^3, the duration scale factor, which is the initial condition for the augmented state. The quantity $T\xi^3$ represents our guess for the optimal duration of the maneuver.

```
>> x0=[0 0 1]';                     % Initial conditions for augmented system
>> fixed=[1 1 0]';                  % Which initial conditions are fixed
>> x0_lower=[0 0 0.1]';             % Lower bound for free initial condition
>> x0_upper=[0 0 10]';              % Upper bound for free initial condition

>> X0=[x0,fixed,x0_lower,x0_upper]

X0 =

         0    1.0000         0         0
         0    1.0000         0         0
    1.0000         0    0.1000   10.0000
```

The first column of `X0` is the initial conditions for the problem; there are three states including the augmented state. The initial conditions for the original problem were $x(0) = (0, 0)^T$. The initial condition for the augmented state is set to `x0(3)` $= \xi^3 = 1$ to indicate that our initial guess for the optimal final time is one times the nominal final time of $T = 10$, *i.e.*, $\xi^3 T$. The second column of `X0` indicates which initial conditions are to be considered fixed and which are to be treated as free variables for the optimization program to adjust. A one indicates fixed and a zero indicates free. The third and fourth columns provide lower an upper bound for the free initial conditions.

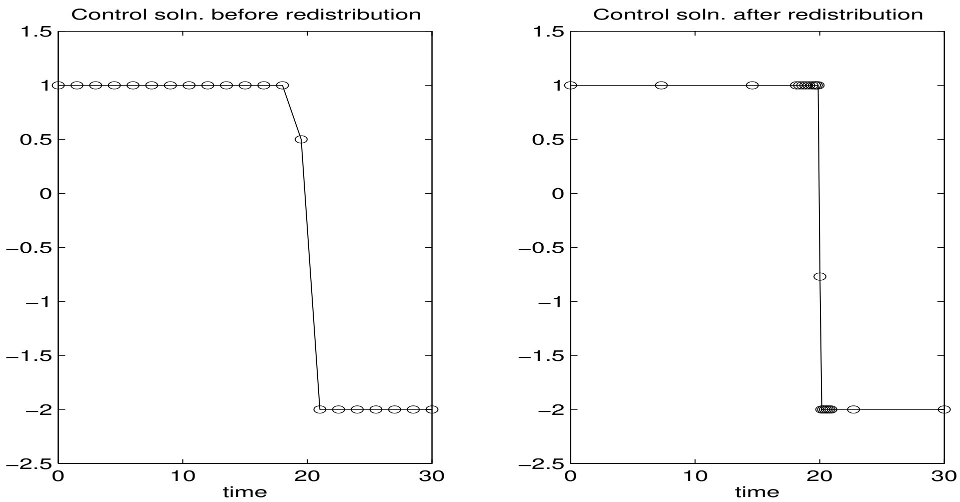
```
>> u0=zeros(1,N+1);
>> [u,x,f]=riots(X0,u0,t,-2,1,[],100,2);         % Solve problem; f=x(3,1)=x0(3)
>> f*T                                           % Show the final time.

ans =

   29.9813
```

In this call to **riots**, we have also specified a lower bound of -2 and an upper bound of 1 for all of the control spline coefficients. Since we are using second order splines, this is equivalent to specifying bounds on the value of the control at the spline breakpoints, *i.e.* bounds on $u(t_k)$. We also specify that the second order Runge-Kutta integration routine should be used. The objective value `f` $= \xi^3$ is the duration scale factor. The final time is given by $a + (b-a)\xi^3 = T\xi^3 = 10$`f`. Here we see that the final time is 29.9813. A plot of the control solution indicates a fairly broad transition region whereas we expect a bang-bang solution. We can try to improve the solution by redistributing the integration mesh. We can then re-solve the problem using the new mesh and starting from the previous solution interpolated onto the new mesh. This new mesh is stored in `new_t`, and `new_u` contains the control solution interpolated onto this new mesh.

```
>> [new_t,new_u]=distribute(t,u,x,2,[],1,1);            % Re-distribute mesh

redistribute_factor = 7.0711

Redistributing mesh.
>> X0(:,1) = x(:,1);
>> [u,x,f]=riots(X0,new_u,new_t,-2,1,[],100,2);
>> f*10

ans =

   30.0000
```

Notice that before calling **riots** the second time, we set the initial conditions (the first column of `X0`) to `x(:,1)`, the first column of the trajectory solution returned from the preceding call to **riots**. Because ξ^3 is a free variable in the optimization, `x(3,1)` is different than what was initially specified for `x0(3)`. Since `x(3,1)` is likely to be closer to the optimal value for ξ^3 than our original guess we set the current guess for `X0(3,1)` to `x(3,1)`.

We can see the improvement in the control solution and the solution for the final time. The reported final time solution is 30 and this happens to be the exact answer. The plot of the control solution before and after the mesh redistribution is shown below. The circles indicate where the mesh points are located. The improved solution does appear to be a bang-bang solution.



Session 4 (Example using outer). This example demonstrates the experimental program **outer** which repeatedly adjusts the integration mesh between calls to **riots** in order to achieve a desired solution accuracy. We use **outer** to solve the Goddard rocket ascent problem implemented in the file 'goddard.c'. The Goddard rocket problem is a free-time problem whose objective is to maximize the rocket's altitude subject to having a fixed amount of fuel. This problem is particularly difficult because its solution contains a singular sub-arc. We use an initial guess of $u(t) = 1$ for all t so that the rocket starts out climbing and does not fall into the ground. We will use a second order spline representation and start with a discretization level of $N = 50$. Also, since this is a minimum-time problem, we augmented the system dynamics with a fourth state that represents the duration scale factor. We start by guessing a duration scale factor of 0.1 by setting $\xi^4 = 0.1$ and we specify $[0, 1]$ for the nominal time interval. Thus the nominal final time is $T\xi^4 = 0.1$.

```
>> x0=[0 1 1 0.1]';
>> fixed=[1 1 1 0]';
>> t=[0:1/50:1];
>> u0=ones(1,51);
```

Now **outer** is called with lower and upper control bounds of 0 and 3.5, respectively; no systems parameters; a maximum of 300 iterations for each inner loop; a maximum of 10 outer loop iteration with a maximum discretization level of $N = 500$; default termination tolerances; integration algorithm 4 (RK4); and mesh redistribution strategy 2.

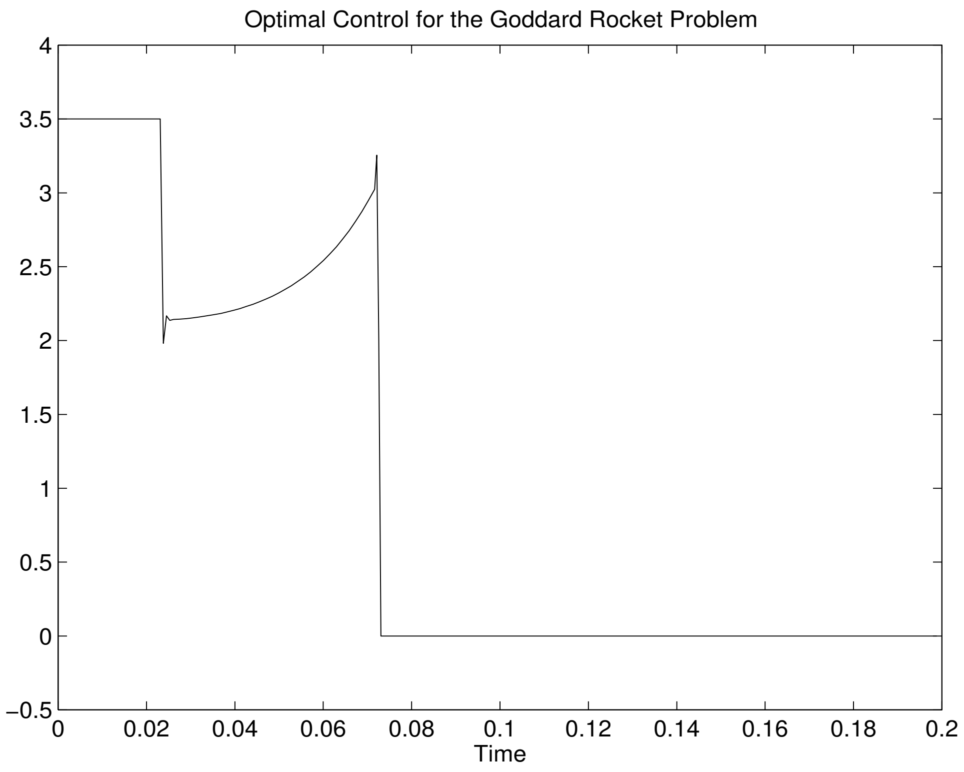
```
>> [new_t,u,x]=outer([x0,fixed],u0,t,0,3.5,[],500,[10;500],4,[],2);
Goddard

Completed 70  riots  iterations.  Normal Termination.
Doubling mesh.
========Completed  1  OUTER  iteration=========
Norm of Lagrangian gradient = 3.43882e-05
Sum of constraint errors    = 4.57119e-09
Objective function value    = -1.01284
Integration error           = 1.49993e-06
===============================================
Goddard

Completed 114  riots  iterations.  Kuhn-Tucker conditions satisfied but sequence did not converge.

========Completed   2 OUTER  iterations========
Norm of Lagrangian gradient = 4.64618e-06
Sum of constraint errors    = 4.41294e-10
Objective function value    = -1.01284
Integration error           = 2.01538e-07
Change in solutions         = 0.128447
Control error estimate      = 0.0200655
===============================================

Redistribution factor = 2.07904
Redistributing mesh.
New mesh contains 146 intervals.  Old mesh contained 100 intervals.
Goddard
```

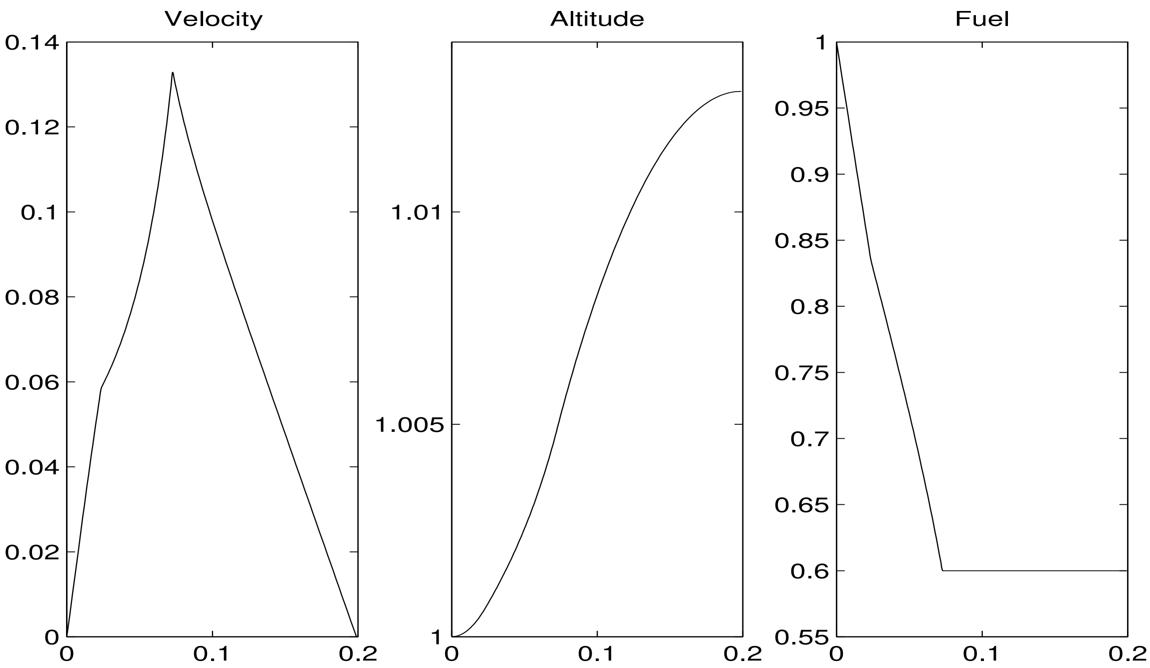
```
Completed 206  riots  iterations.  Kuhn-Tucker conditions satisfied but sequence did not converge.

========Completed   3 OUTER  iterations========
Norm of Lagrangian gradient = 2.38445e-08
Sum of constraint errors    = 8.49733e-11
Objective function value    = -1.01284
Integration error           = 4.67382e-09
Change in solutions         = 0.0878133
Control error estimate      = 0.000452989
==============================================
Normal Termination.
CPU time = 26.9167 seconds.
```

The message stating that the Kuhn-Tucker conditions are satisfied but that the sequence did not converge is a message from NPSOL which is the nonlinear programming algorithm linked with **riots** in this example. This message indicates that, although first order optimality conditions for optimality are satisfied (the norm of the gradient of the Lagrangian is sufficiently small), the control functions from one iteration of **riots** to the next have not stopped changing completely. The sources of this problem are *(i)* the Goddard problem is a singular optimal control problem; this means that small changes in the controls over some portions of the time interval have very little effect on the objective function and *(ii)* **outer** calls **riots** with very tight convergence tolerances. Because of this, the calls to **riots** probably performed many more iterations than were useful for the level of accuracy achieved. Choosing better convergence tolerances is a subject for future research.

The optimal control and optimal state trajectories are shown on the next page. Notice that to plot the optimal control we multiply the time vector `new_t` by `x(4,1)` which contains the duration scale factor. The optimal final time for this problem, since $a = 0$ and $b = 1$, is just `x(4,1)=0.1989`. Note that the final mass of the rocket is 0.6. This is the weight of the rocket without any fuel. The maximum height is the negative of the objective function, $h^*(t) \approx 1.01284$.

```
>> sp_plot(new_t*x(4,1),u)
>> plot(new_t*x(4,1),x(1,:))
>> plot(new_t*x(4,1),x(2,:))
>> plot(new_t*x(4,1),x(3,:))
```


Optimal Control for the Goddard Rocket Problem
4
3.5
3
2.5
2
1.5
1
0.5
0
−0.5
0
0.02
0.04
0.06
0.08
0.1
0.12
0.14
0.16
0.18
0.2
Time


Velocity
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
0.1
0.2
Altitude
1.01
1.005
1
0
0.1
0.2
Fuel
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0
0.1
0.2

4. USER SUPPLIED SYSTEM SUBROUTINES

All of the functions in the description of **OCP** in §2 are computed from the user functions **h**, **l** and **g**; the derivatives of these functions are computed from the user functions **Dh**, **Dl** and **Dg**. Two other user functions, **activate** and **init**, are required for the purpose of passing information to and from RIOTS_95.

Smoothness Requirements. The user-supplied functions must have a certain degree of smoothness. The smoothness requirement comes about for three reasons. First, the theory of differential equations requires, in general, that $h(t, x, u)$ be piecewise continuous with respect to t, Lipschitz continuous with respect to x and u and that $u(\cdot)$ be continuous, in order to ensure the existence and uniqueness of a solution satisfying the system of differential equations. A finite number of discontinuities in $h(\cdot, x, u)$ and $u(\cdot)$ are allowable. Second, the optimization routines needs at least one continuous derivative of the objective and constraint functions $g(\cdot, \cdot)$ and $l(t, \cdot, \cdot)$. Two continuous derivatives are needed in order for there to be a chance of superlinear convergence. The third reason is that the accuracy of numerical integration of differential equations depends on the smoothness of $h(\cdot, \cdot, \cdot)$ and $l(\cdot, \cdot, \cdot)$. For a fixed step-size methods with order s, $\partial^{(s)} h(t, x, u)/\partial x^s$ and $\partial^{(s)} h(t, x, u)/\partial u^s$ should be continuous (or the $(r-1)$-th partial should be Lipschitz continuous). Furthermore, any discontinuities in $h(\cdot, x, u(\cdot))$ or its derivatives should occur only at integration breakpoints[7]. Conversely, the user should place integration breakpoints wherever such discontinuities occur. The same considerations also hold for the function $l(t, x, u)$. For variable step-size integration, $h(t, x, u)$ and $l(t, x, u)$ should have at least continuous partial derivatives of order one with respect to x and u. Again, any discontinuities in $h(\cdot, x, u(\cdot))$ and $l(\cdot, x, u(\cdot))$ or its derivatives should only occur at integration break points.

Constraint Qualifications. A common requirement of mathematical programming algorithms is linear independence of the active constraints gradients at a solution. It is easy to mathematically specify a valid constraint in such a way that this condition is violated. For example, consider a scalar constraint of the form $g(u) = 0$. This constraint can be specified as

$$g(u)^2 = 0 \ .$$

However, $\frac{d}{du}(g(u)^2) = 2g(u)\frac{dg}{du}$. Thus, if this constraint is active at the solution u^*, *i.e.*, $g(u^*) = 0$, then the gradient of this constraint is zero. So this specification for the constraint violates the constraint qualification. However, if the constraint is specified simply as

$$g(u) = 0 \ ,$$

then the constraint qualification is not violated.

The user functions can be supplied as object code or as M-files. The C-syntax and M-file syntax for these functions are given below. Because all arguments to the object code versions of the functions are passed by reference, the object code format is compatible with Fortran. A template for these functions can be found in the file `systems/template.c`. There are also several example problems in the `systems` directory. In addition to the user-supplied routines, this section also describes two other functions, **get_flags** and **time_fnc**, that are callable by user object code.

[7]Note that discontinuities in $u(t)$ can only occur at the spline breakpoints, t_k.

There are three main differences between object code and M-file versions of the user functions:

- The programs in RIOTS_95 execute much faster when object code is used.
- Object code versions of the user functions do not need to assign zero values to array components which are always zero. M-file versions must set all array values (with the exception of **sys_init**).
- There must be a separate M-file for each function with the same name as that function. The names begin with `sys_` followed by the name of the function. For example, sys_Dh.m is the M-file for the user function **sys_Dh**. The directory in which these M-files are located must be in Matlab's search path.
- **Important:** Arrays in Matlab are indexed starting from 1 whereas in C arrays are indexed starting from 0. For example, `neq[4]` in C code has an M-file equivalent of `neq(5)`.

activate, sys_activate

Purpose

This function is always called once before any of the other user-supplied functions. It allows the user to perform any preliminary setup needed, for example, loading a data array from a file.

C Syntax

```
void activate(message)
     char **message;
{
  *message = "";
  /* Any setup routines go here. */
}
```

M-file Syntax

```
function message = sys_activate

message = '';
```

Description

If the message string is set, that string will be printed out whenever **simulate** (form 0) or an optimization routine is called. It is useful to include the name of the optimal control problem as the message.

See Also: **get_flags**.

init, sys_init

Purpose

This function serves two purposes. First, it provides information about the optimal control problem to RIOTS_95. Second, it allows system parameters to be passed from Matlab to the user-defined functions at run-time. These system parameters can be used, for instance, to specify constraint levels. Unlike **activate**, **init** may be called multiple times. The array `neq[]` is explained after the syntax.

C Syntax

```
void init(neq,params)
int neq[];
double *params;
{
  if ( params == NULL ) {
    /* Set values in the neq[] array. */
  }
  else {
    /* Read in runtime system parameters. */
  }
}
```

M-file Syntax

```
function neq = sys_init(params)

% if params is NULL then setup neq.  Otherwise read system
% parameters in params.  In Matlab, arrays are indexed
% starting from 1, so neq(i) corresponds to the C statement
% neq[i-1].
if params == [],
   % Each row of neq consists of two columns.  The value in
   % the first column specifies which piece of system
   % information to set.  The value in the second column is
   % the information.  For example, to indicate that the
   % system has 5 system parameters, one row in neq should be
   % [3 5] since neq(3) stores the number of system
   % parameters.

   % Here we set nstates = 2; ninputs = 1; 1 nonlinear
   % endpoint constr.
   neq = [1 2 ; 2 1 ; 12 1 ];
else
   % Read in systems parameters from params and store them in
   % the global variable sys_params which will be accessible
   % to other systems M-files.
   global sys_params
   sys_params = params;
end
```

Description

When this functions is called, the variable `params` will be set to 0 (`NULL`) if `init()` is expected to return information about the optimal control problem via the `neq[]` array. Otherwise, `params` is a vector of system parameters being passed from Matlab to the user's program. When `params==0`, the values in `neq[]` should be set to indicate the following:

```
neq[0]  ---  Number of state variables.
neq[1]  ---  Number of inputs.
neq[2]  ---  Number of system parameters.
neq[3]  ---  Not used on calls to init().  Contains time index.
neq[4]  ---  Not used on calls to init().  Used to indicate which function to evaluate.
neq[5]  ---  Number of objective functions (must equal 1).
neq[6]  ---  Number of general nonlinear trajectory inequality constraints.
neq[7]  ---  Number of general linear trajectory inequality constraints.
neq[8]  ---  Number of general nonlinear endpoint inequality constraints.
neq[9]  ---  Number of general linear endpoint inequality constraints.
neq[10] ---  Number of general nonlinear endpoint equality constraints.
neq[11] ---  Number of general linear endpoint equality constraints.
neq[12] ---  Indicates type of system dynamics and cost functions:
                0 --> nonlinear system and cost,
                1 --> linear system,
                2 --> linear and time-invariant system,
                3 --> linear system with quadratic cost,
                4 --> linear and time-invariant with quadratic cost.
```

Remember that, for M-files, `neq(i)` is equivalent to the C-code statement `neq[i-1]`. The values of `neq[]` all default to zero except `neq[5]` which defaults to 1. The relationship between the values in `neq[]` and the general problem description of **OCP** given in §2 is as follows: n = `neq[0]`, m = `neq[1]`, p = `neq[2]`, q_{ti} = `neq[6]+neq[7]`, q_{ei} = `neq[8]+neq[9]` and q_{ee} = `neq[10]+neq[11]`. The locations `neq[3]` and `neq[4]` are used in calls to the other user-defined functions.

If **init** sets `neq[2]>0`, then **init** will be called again with `params` pointing to an array of system parameters which are provided by the user at run-time. These parameters can be stored in global variables for use at other times by any of the other user-defined functions. Some examples of useful system parameters include physical coefficients and penalty function parameters. These parameters are fixed and will not be adjusted during optimization. Parameters that are to be used as decision variables must be specified as initial conditions to augmented states π with $\dot{\pi} = 0$.

Notes

1. Control bounds should be indicated separately when calling the optimization routines. Do not include any simple bound constraints in the general constraints. Similarly, simple bounds on free initial conditions should be specified on the command line.

2. For nonlinear systems, all constraints involving a state variable are nonlinear functions of the control. Thus, the constraint $g(\xi, x(b)) = x(b) = 0$, while linear in its arguments, is nonlinear with respect to u. The user does not need to account for this situation, however, and should indicate that g is a linear constraint. RIOTS_95 automatically treats all general constraints for nonlinear systems as nonlinear.

h, sys_h

Purpose

This function serves only one purpose, to compute $h(t, x, u)$, the right hand side of the differential equations describing the system dynamics.

C Syntax

```
void h(neq,t,x,u,xdot)
     int neq[];
     double *t,x[NSTATES],u[NINPUTS],xdot[NSTATES];
{
  /* Compute xdot(t) = h(t,x(t),u(t)). */
}
```

M-file Syntax

```
function xdot = sys_h(neq,t,x,u)

global sys_params

% xdot must be a column vector with n rows.
```

Description

On entrance, `t` is the current time, `x` is the current state vector and `u` is the current control vector. Also, `neq[3]` is set to the current discrete-time index, $k-1$, such that $t_k \leq t < t_{k+1}$[8].

On exit, the array `xdot[]` should contain the computed value of $h(t, x, u)$. The values of `xdot[]` default to zero for the object code version. Note that for free final time problems the variable `t` should not be used because derivatives of the system functions with respect to `t` are not computed. In the case of non-autonomous systems, the user should augment the state variable with an extra state representing time (see transcription for free final time problems in §2).

See Also: **time_fnc**.

[8]The index is $k-1$ since indexing for C code starts at zero. For M-files, `neq(4) = k`.

l, sys_l

Purpose

This function serves two purposes. It is used to compute values for the integrands of cost functions, $l_o(t, x, u)$, and the values of state trajectory constraints, $l_{ti}(t, x, u)$.

C Syntax

```
double l(neq,t,x,u)
     int neq[];
     double *t,x[NSTATES],u[NINPUTS];
{
  int F_num, constraint_num;
  double z;

  F_num = neq[4];
  if ( F_num == 1 ) {
    /* Compute z = l(t,x(t),u(t)) for the integrand.        */
    /* If this integrand is identically zero,               */
    /* set z = 0 and neq[3] = -1.                           */
  }
  else {
    constraint_num = F_num - 1;
    /* Compute z = l(t,x(t),u(t) for the     */
    /* constraint_num trajectory constraint. */
  }
  return z;
}
```

M-file Syntax

```
function z = sys_l(neq,t,x,u)
% z is a scalar.

global sys_params
F_NUM = neq(5);

if F_NUM == 1
  % Compute z = l(t,x(t),u(t)) for the objective integrand.
else
  constraint_num = F_num - 1;
  % Compute z = l(t,x(t),u(t)) for the constraint_num
  % traj. constraint.
end
```

Description

On entrance, `t` is the current time, `x` is the current state vector and `u` is the current control vector. Also, `neq[3]` is set to the current discrete-time index $k-1$ such that $t_k \le t < t_{k+1}$ (see footnote for **h**) and `neq[4]` is used to indicate which integrand or trajectory constraint is to be evaluated. Note that, for free final time problems, the variable `t` should not be used because derivatives of the system functions with respect to `t` are not computed. In this case, the user should augment the state variable with an extra time state and an extra final-time state as described in §2.

If $neq[4] = 1$, then `z` should be set to $l_o^{neq[4]}(t, x, u)$. If $l_o^{neq[4]}(\cdot, \cdot, \cdot) = 0$ then, besides returning 0, **l** (in object code versions) can set $neq[3] = -1$ to indicate that the function is identically zero. The latter increases efficiency because it tells RIOTS_95 that there is no integral cost. Only the function **l** is allowed to modify $neq[3]$. Regardless of how $neq[3]$ is set, **l** *must* always return a value even if the returned value is zero.

If $neq[4] > 1$, then z should be set to $l_{ti}^{neq[4]-1}(t, x, u)$. If there are both linear and nonlinear trajectory constraints, the nonlinear constraints must precede those that are linear. The ordering of the functions computed by **l** is summarized in the following table:

	ν	function to compute
$neq[4] = 1$	$neq[4]$	$l_o^{\nu}(t, x, u)$
$neq[4] > 1$	$neq[4] - 1$	$l_{ti}^{\nu}(t, x, u)$, nonlinear $l_{ti}^{\nu}(t, x, u)$, linear

g, sys_g

Purpose

This function serves two purposes. It is used to compute the endpoint cost function $g_o(\xi, x(b))$ and the endpoint inequality and equality constraints $g_{ei}(\xi, x(b))$ and $g_{ee}(\xi, x(b))$. The syntax for this function includes an input for the time variable t for consideration of future implementations and should not be used. Problems involving a cost on the final time T should use the transcription for free final time problems described in §2.

C Syntax

```
double g(neq,t,x0,xf)
     int neq[];
     double *t,x0[NSTATES],xf[NSTATES];
{
  int F_num, constraint_num;
  double value;

  F_num = neq[4];
  if ( F_num <= 1 ) {
    /* Compute value of g(t,x0,xf) for the */
    /* F_num cost function.                */
  }
  else {
    constraint_num = F_num - 1;
    /* Compute value g(t,x0,xf) for the    */
    /* constraint_num endpoint constraint. */
  }
  return value;
}
```

M-file Syntax

```
function J = g(neq,t,x0,xf)
% J is a scalar.

global sys_params
F_NUM = neq(5);
if F_NUM <= sys_params(6)
   % Compute g(t,x0,xf) for cost function.
elseif F_NUM == 2
   % Compute g(t,x0,xf) for endpoint constraints.
end
```

Description

On entrance, `x0` is the initial state vector and `xf` is the final state vector. The value `neq[4]` is used to indicate which cost function or endpoint constraint is to be evaluated. Nonlinear constraints must precede linear constraints. The order of functions to be computed is summarized in the following table:

	ν	function to compute
$neq[4] = 1$	1	$g_o(\xi, x(b))$
$1 < neq[4] \le 1 + q_{ei}$	$neq[4] - 1$	$g_{ie}^{\nu}(\xi, x(b))$, nonlinear $g_{ie}^{\nu}(\xi, x(b))$, linear
$1 + q_{ei} < neq[4] \le 1 + q_{ei} + q_{ee}$	$neq[4] - 1 - q_{ei}$	$g_{ee}^{\nu}(\xi, x(b))$, nonlinear $g_{ee}^{\nu}(\xi, x(b))$, linear

See Also: **time_fnc**.

Dh, sys_Dh
Dl, sys_Dl
Dg, sys_Dg

Purpose

These functions provide the derivatives of the user-supplied function with respect to the arguments x and u. The programs **riots** (see §6) can be used without providing these derivatives by selecting the finite-difference option. In this case, dummy functions must be supplied for **Dh**, **Dl** and **Dg**.

C Syntax

```
void Dh(neq,t,x,u,A,B)
     int neq[];
     double *t,x[NSTATES],u[NINPUTS];
     double A[NSTATES][NSTATES],B[NSTATES][NINPUTS];
{
  /* The A matrix should contain dh(t,x,u)/dx. */
  /* The B matrix should contain dh(t,x,u)/du. */
}


double Dl(neq,t,x,u,l_x,l_u)
  int neq[];
  double *t,x[NSTATES],u[NINPUTS],l_x[NSTATES],l_u[NINPUTS];
{
  /* l_x[] should contain dl(t,x,u)/dx        */
  /* l_u[] should contain dl(t,x,u)/du        */
  /* according to the value of neq[4].        */
  /* The return value is dl(t,x0,xf)/dt which */
  /*   is not currently used by RIOTS.        */
  return 0.0;
}


double Dg(neq,t,x0,xf,g_x0,g_xf)
     int neq[];
     double *t,x0[NSTATES],xf[NSTATES],J_xf[NSTATES];
{
  /* g_x0[] should contain dg(t,x0,xf)/dx0.   */
  /* g_xf[] should contain dg(t,x0,xf)/dxf.   */
  /*   according to the value of neq[4].      */
  /* The return value is dg(t,x0,xf)/dt which */
  /*   is not currently used by RIOTS.        */
  return 0.0;
}
```

M-file Syntax

```
function [A,B] = sys_Dh(neq,t,x,u)
global sys_params
% A must be an n by n matrix.
% B must be an n by m matrix.


function [l_x,l_u,l_t] = sys_Dl(neq,t,x,u)
global sys_params
% l_x should be a row vector of length n.
% l_u should be a row vector of length m.
% l_t is a scalar---not currently used.


function [g_x0,g_xf,g_t] = sys_cost(neq,t,x0,xf)
global sys_params
% g_x0 and g_xf are row vectors of length n.
% g_t is a scalar---not currently used.
```

Description

The input variables and the ordering of objectives and constraints are the same for these derivative functions as they are for the corresponding functions **h, l,** and **g**. The derivatives with respect to t are not used in the current implementation of RIOTS_95 and can be set to zero. The derivatives should be stored in the arrays as follows:

Function	First output	index range	Second output	index range
Dh	$A[i][j] = \left[\dfrac{dh(t,x,u)}{dx}\right]_{i+1,j+1}$	$i = 0{:}\,n-1$ $j = 0{:}\,n-1$	$B[i][j] = \left[\dfrac{dh(t,x,u)}{du}\right]_{i+1,j+1}$	$i = 0{:}\,n-1$ $j = 0{:}\,m-1$
Dl	$l\_x[i] = \left[\dfrac{dl(t,x,u)}{dx}\right]_{i+1}$	$i = 0{:}\,n-1$	$l\_u[i] = \left[\dfrac{dl(t,x,u)}{du}\right]_{i+1}$	$i = 0{:}\,m-1$
Dg	$g\_x0[i] = \left[\dfrac{dg(t,x0,xf)}{dx0}\right]_{i+1}$	$i = 0{:}\,n-1$	$g\_xf[i] = \left[\dfrac{dg(t,x0,xf)}{dxf}\right]_{i+1}$	$i = 0{:}\,m-1$
sys_Dh	$A(i,j) = \left[\dfrac{dh(t,x,u)}{dx}\right]_{i,j}$	$i = 1{:}\,n$ $j = 1{:}\,n$	$B(i,j) = \left[\dfrac{dh(t,x,u)}{du}\right]_{i,j}$	$i = 1{:}\,n$ $j = 1{:}\,m$
sys_Dl	$l\_x(i) = \left[\dfrac{dl(t,x,u)}{dx}\right]_{i}$	$i = 1{:}\,n$	$l\_u(i) = \left[\dfrac{dl(t,x,u)}{du}\right]_{i}$	$i = 1{:}\,m$
sys_Dg	$g\_x0(i) = \left[\dfrac{dg(t,x0,xf)}{dx0}\right]_{i}$	$i = 1{:}\,n$	$g\_xf(i) = \left[\dfrac{dg(t,x0,xf)}{dxf}\right]_{i}$	$i = 1{:}\,m$

Note that, for **sys_Dh**, RIOTS_95 automatically accounts for the fact that Matlab stores matrices transposed relative to how they are stored in C.

get_flags

Purpose

This function allows user-supplied object code to read a vector of integers from Matlab's workspace.

C Syntax

```
int get_flags(flags,n)
     int flags[],*n;
```

Description

A call to **get_flags** causes `flags[]` to be loaded with up to `n` integers from the array `FLAGS` if `FLAGS` exists in Matlab's workspace. It is the user's responsibility to allocate enough memory in `flags[]` to store `n` integers. The value returned by **get_flags** indicates the number of integers read into `flags[]`.

The main purpose of **get_flags** is to allow a single system program to be able to represent more than one problem configuration. The call to **get_flags** usually takes place within the user-function **activate**. In the example below, **get_flags** reads in the number of constraints to use for the optimal control problem.

Example

```
extern int get_flags();
static int Constraints;

void activate(message)
     char **message;
{
  int n,flags[1];

  *message = "Use FLAGS to specify number of constraints.";
  n = 1;
  if ( get_flags(flags,&n) > 0 );
    Constraints = flags[0];
  else
    Constraints = 0;
}
```

Notes

1. It is best to define `FLAGS` as a global variable in case **simulate** gets called from within an M-file. This is accomplished by typing

```
>> global FLAGS
```

At the Matlab prompt. To clear `FLAGS` use the Matlab command

```
>> clear global FLAGS
```

2. Since **activate** is called once only, you must clear **simulate** if you want to re-read the values in `FLAGS`. To clear **simulate**, at the Matlab prompt type

```
>> clear simulate
```

3. For M-files, any global variable can be read directly from Matlab's workspace so an M-file version of **get_flags** is not needed.

time_fnc

Purpose

This function allows user-supplied object code to make calls back to a user-supplied Matlab m-function called **sys_time_fnc.m** which can be used to compute a function of time. Call-backs to Matlab are very slow. Since this function can be called thousand of times during the course of a single system simulation it is best to provide the time function as part of the object code if possible.

C Syntax

```
void time_fnc(t,index,flag,result)
     int *index,*flag;
     double *t,result[];
```

Syntax of sys_time_fnc.m

```
function f = sys_time_fnc(tvec)

% tvec = [time;index;flag]
% Compute f(time,index,flag).
```

Description

If **time_fnc** is to called by one of the user-functions, then the user must supply an m-function named **sys_time_fnc**. The inputs `tvec(1)=time` and `tvec(2)=index` to **sys_time_fnc** are related by $t_{index} \leq \texttt{time} \leq t_{index+1}$. The value of `index` passed to **sys_time_fnc** is one greater than the value passed from **time_fnc** to compensate for the fact the Matlab indices start from 1 whereas C indices start from 0. The input `flag` is an integer that can be used to select from among different time functions. Even if `flag` is not used, it *must* be set to some integer value.

The values in the vector `f` returned from **sys_time_fnc** are stored in `result` which must have enough memory allocated for it to store these values.

Example

Suppose we want **l** to compute $f(t)x^1(t)$ where $f(t) = sin(t) + y_d(t)$ with $y_d(t)$ is some pre-computed global variable in the Matlab workspace. Then we can use **time_fnc** to compute $f(t)$ and use this value to multiply `x[0]`:

```
extern void time_fnc();
double l(neq,t,x,u)
     int neq[];
     double *t,x[NSTATES],u[NINPUTS];
{
  int i,zero;
  double result;

  i = neq[3];                         /* Discrete-time index. */
  zero = 0;
  time_fnc(t,&i,&zero,&result);       /* Call time_fnc with flag=0. */
  return result*x[0];                 /* Return f(t)*x1(t).   */
}
```

Here is the function that computes $f(t)$. It computes different functions depending on the value of `flag=t(3)`. In our example, it is only called with `flag=0`.

```
function f = sys_time_fnc(t)

global yd             % Suppose yd is a pre-computed, global variable.
time = t(1);
index = t(2);
flag = t(3);

if flag == 0
  f = yd(time) + sin(time);
else
  f = another_fnc(time);
end
```

5. SIMULATION ROUTINES

This section describes the central program in RIOTS_95, **simulate**. All of the optimization programs in RIOTS_95 are built around **simulate** which is responsible for computing all function values and gradients and serves as an interface between the user's routines and Matlab.

The computation of function values and gradients is performed on the integration mesh

$$\mathbf{t}_N \doteq \{ t_{N,k} \}_{k=1}^{N+1} .$$

This mesh also specifies the breakpoints of the control splines. For any mesh $\mathbf{t}_N$ we define

$$\Delta_{N,k} \doteq t_{N,k+1} - t_{N,k} .$$

The values of the trajectories computed by **simulate** are given at the times $t_{N,k}$ and are denoted $x_{N,k}$, $k = 1,..,N+1$. Thus, $x_{N,k}$ represents the computed approximation to the true solution $x(t_{N,k})$ of the differential equation $\dot{x} = h(t,x,u)$, $x(a) = \xi$. The subscript N is often omitted when its its presence is clear from context.

Spline Representation of controls. The controls u are represented as splines. These splines are given by

$$u(t) = \sum_{k=1}^{N+\rho-1} \alpha_k \phi_{\mathbf{t}_N,\rho,k}(t)$$

where $\alpha_k \in \mathbb{R}^m$ and $\phi_{\mathbf{t}_N,\rho,k}(\cdot)$ is the k-th B-spline basis element of order ρ, defined on the knot sequence formed from $\mathbf{t}_N$ by repeating its endpoints ρ times. Currently, RIOTS_95 does not allow repeated interior knots. We will denote the collection of spline coefficients by

$$\alpha \doteq \{ \alpha_k \}_{k=1}^{N+\rho-1} .$$

For single input systems, α is a row vector. Those interested in more details about splines are referred to the excellent reference [6]. The times t_k, $k = 1,\ldots,N$, define the spline breakpoints. On each interval $[t_k, t_{k+1}]$, the spline coincides with an ρ-th order polynomial. Thus, fourth order splines are made up of piecewise cubic polynomials and are called cubic splines. Similarly, third order splines are piecewise quadratic, second order splines are piecewise linear and first order splines are piecewise constant. For first and second order splines, $\alpha_k = u(t_k)$. For higher-order splines, the B-spline basis elements are evaluated using the recursion formula in (A2.2a).

The following pages describe **simulate**. First, the syntax and functionality of **simulate** is presented. This is followed by a description of the methods used by the integration routines in **simulate** to compute function values and gradients. Finally, two functions, **check_deriv** and **check_grad**, for checking user-supplied derivative information, and the function **eval_fnc** are described.

simulate

Purpose

This is the central program in RIOTS_95. The primary purpose of **simulate** is to provide function values and gradients of the objectives and constraints using one of six integration algorithms. The optimization routines in RIOTS_95 are built around **simulate**. This program also serves as a general interface to the user-supplied functions and provides some statistical information.

There are currently seven different forms in which **simulate** can be called. Form 1 and form 2 (which is more conveniently accessed using **eval_fnc**) are the most useful for the user. The other forms are used primarily by other programs in RIOTS_95. The form is indicated by the first argument to simulate. A separate description for each form is given below.

Form 0

```
[info,simed] = simulate(0,{params})
```

Form 1

```
[f,x,du,dz,p] = simulate(1,x0,u,t,ialg,action)
```

Form 2

```
f=simulate(2,f_number,1)
[du,dz,p] = simulate(2,f_number,action)
```

Form 3

```
[xdot,zdot] = simulate(3,x,u,t,{f_num,{k}})
[xdot,zdot,pdot] = simulate(3,x,u,t,p,{k})
```

Form 4

```
[h_x,h_u,l_x,l_u] = simulate(4,x,u,t,{f_num,{k}})
```

Form 5

```
[g,g_x0,g_xf] = simulate(5,x0,xf,tf,{f_num})
```

Form 6

```
stats = simulate(6)
```

Form 7

```
lte = simulate(7)
```

Description of Inputs and Outputs

The following table describes the inputs that are required by the various forms of **simulate**.

Table S1

Input	number of rows	number of columns	description
`x0`	n	1[9]	initial state
`xf`	n	1	final state
`u`	m	$N+\rho-1$	control vector
`t`	1	$N+1$	time vector
`tf`	1 to 4	1	final time
`ialg`	1	1	integration algorithm
`action`	1	1	(see below)
`f_num`	1	1	(see below)
`params`	(see below)	(see below)	system parameters

The following table describes the outputs that are returned by the various forms of **simulate**.

Table S2

Output	number of rows	number of columns	description
`f`	1	1	objective or constraint value
`x`	n	$N+1$	state trajectory
`p`	n	$N+1$	adjoint trajectory
`du`	m	$N+\rho-1$	control gradient
`dx0`	n	1	gradient of initial conditions
`lte`	$n+1$	$N+1$	local integration error
`xdot`	n	$N+1$	$h(t,x,u)$
`zdot`	1	$N+1$	$l(t,x,u)$
`h_x`	n	n	$\partial h/\partial x$
`h_u`	n	m	$\partial h/\partial u$
`l_x`	1	n	$\partial l/\partial x$
`l_u`	1	m	$\partial l/\partial u$
`g_x0`	1	n	$\partial g/\partial x_0$
`g_xf`	1	n	$\partial g/\partial x_f$

If a division by zero occurs during a simulation, **simulate** returns the Matlab variable `NaN`, which stands for "Not a Number", in the first component of each output. This can be detected, if desired, using the Matlab function `isnan()`.

Note: The length of the control vector depends on the control representation. Currently, all of the integration routines are setup to work with splines of order ρ defined on the knot sequence constructed from $\mathbf{t}_N$. The current implementation of RIOTS_95 does not allow repeated interior knots. The length (number of columns) of `u` and `du` is equal to $N+\rho-1$ where `N=length(t)-1` is the number of intervals in

[9] `x0` can be a matrix but only the first column is used.

the integration mesh. The allowable spline orders depends on the integration algorithm, `ialg`, according to the following table:

Table S3

IALG	Order of spline representation
0 (discrete)	discrete-time controls
1 (Euler)	$\rho = 1$
2 (RK2)	$\rho = 2$
3 (RK3)	$\rho = 2$
4 (RK4)	$\rho = 2, 3, 4$
5 (LSODA)	$\rho = 1, 2, 3, 4$[10]
6 (LSODA w/0 Jacobians)	$\rho = 1, 2, 3, 4$[10]

When more than one spline order is possible, the integration determines the order of the spline representation by comparing the length of the control input `u` to the length of the time input `t`. If LSODA is called with `ialg=5`, the user must supply $\frac{dh}{dx}$ and $\frac{dl}{dx}$ in the user-functions **Dh** and **Dl**. If the user has not programmed these Jacobians, LSODA must be called with `ialg=6` so that, if needed, these Jacobians will be computed by finite-differences. The different integration methods are discussed in detail following the description of the various forms in which **simulate** can be called.

Bugs

1. There may be a problem with computation of gradients for the variable step-size integration algorithm (`ialg=5,6`) if the number of interior knots n_{knots} is different than one (see description of form 1 and gradient computations for LSODA below).

See Also: **eval_fnc**

Description of Different Forms

```
[info,simed] = simulate(0,{params})
```

This form is used to load system parameters and to return system information. If `params` is supplied, **simulate** will make a call to **init** so that the user's code can read in these parameters. Normally `params` is a vector. It can be a matrix in which case the user should keep in mind that Matlab stores matrices column-wise (Fortran style). If the system has no parameters then either omit `params` or set `params=[]`. If no output variables are present in this call to **simulate** the system message loaded on the call to **activate** and other information about the system will be displayed.

[10] The maximum spline order allowed by **simulate** when using LSODA can be increased by changing the pre-compiler define symbol `MAX_ORDER` in adams.c.

The following is a list of the different values in `info` returned by **simulate**:

`info(1)`	number of states
`info(2)`	number of inputs
`info(3)`	number of system parameters
`info(4)`	(reserved)
`info(5)`	(reserved)
`info(6)`	number of objective functions
`info(7)`	number of nonlinear trajectory inequality constraints
`info(8)`	number of linear trajectory inequality constraints
`info(9)`	number of nonlinear endpoint inequality constraints
`info(10)`	number of linear endpoint inequality constraints
`info(11)`	number of nonlinear endpoint equality constraints
`info(12)`	number of linear endpoint equality constraints
`info(13)`	type of system (0 through 4) 0: nonlinear dynamics and objective 1: linear dynamics; nonlinear objective 2: linear, time-invariant dynamics; nonlinear objective 3: linear dynamics; quadratic objective 4: linear, time-invariant dynamics; quadratic objective
`info(14)`	number of mesh points used in the most recent simulation

The scalar output `simed` is used to indicate whether a call to **simulate** (form 1) has been made. If `simed=1` then a simulation of the system has occurred. Otherwise `simed=0`.

`[f,x,du,dx0,p] = simulate(1,x0,u,t,ialg,action)`

This form causes the system dynamics, $\dot{x} = h(t, u, x)$ with $x(a) = x0$, to be integrated using the integration method specified by `ialg` (*cf.* Table S3). Also, the value `f` of the first objective function, and possibly its gradients, `du` and `dx0` and the adjoint `p`, can be evaluated. Only the first column of `x0` is read. The strictly increasing time vector `t` of length $N + 1$ specifies the integration mesh on $[a, b]$ with `t`$(1) = a$ and `t`$(N + 1) = b$. The control `u` is composed of m rows of spline coefficients.

The calculations performed by **simulate** (form 2) depend on the value of `action`. These actions are listed in the following table:

Table S4

Action	Return Values
0	no return values
1	function value f
2	f and system trajectory x
3	f, x and control and initial condition gradients du and dz
4	f, x, du, dz and the adjoint trajectory p.

When using the variable step-size method LSODA (`ialg = 5,6`), the argument `ialg` can include three additional pieces of data:

	Setting	Default Value
`ialg(2)`	Number of internal knots used during gradient computation.	1
`ialg(3)`	Relative integration tolerance.	1e-8
`ialg(4)`	Absolute integration tolerance.	1e-8

The meaning of "internal knots" is discussed below in the description of gradient computation with LSODA.

Example

The following commands, typed at the Matlab prompt, will simulate a three state system with two inputs using integration algorithm RK4 and quadratic splines. The simulation time is from $a = 0$ until $b = 2.5$ and there are $N = 100$ intervals in the integration mesh.

```
>> N=100;
>> t = [0:2.5/N:2.5];
>> x0 = [1;0;3.5];
>> u0 = ones(2,N+2);                      % u0(t)=[1;1];
>> [j,x] = simulate(1,x0,u0,t,4,2);
```

`j = simulate(2,f_number,1)`
`[du,dx0,p] = simulate(2,f_number,action)`

This form allows function values and gradients to be computed without re-simulating the system. A call to this form must be proceeded by a call to **simulate** (form 1). The results are computed from the most recent inputs (`x0,u,t,ialg`) for the call to **simulate**, form 1. The following table shows the relationship between the value of `f_number`, and the function value or gradient which is computed.

Table S5

`f_number` range	Function	Function to be evaluated
$1 \le$ `f_number` $\le n_1$	$g_o^\nu(\xi, x(b)) + \int_a^b l_o^\nu(t, x, u)dt$	$\nu =$ `f_number`
$n_1 <$ `f_number` $\le n_2$	$l_{ti}^\nu(t, x(t), u(t))$	$\nu = n\%(N+1)+1$, $t = t_k$ where $n =$ `f_number` $- n_1 - 1$ and $k =$ `f_number` $- n_1 - \nu(N+1)$.
$n_2 <$ `f_number` $\le n_3$	$g_{ei}^\nu(\xi, x(b))$	$\nu =$ `f_number` $- n_2$
$n_3 <$ `f_number` $\le n_4$	$g_{ee}^\nu(\xi, x(b))$	$\nu =$ `f_number` $- n_3$

where $n_1 = q_o$ is the number of objective functions, $n_2 = n_1 + q_{ti}(N+1)$ with q_{ti} the number of trajectory constraints, $n_3 = n_2 + q_{ei}$ with q_{ei} the number of endpoint inequality constraints, and $n_4 = n_3 + q_{ee}$ with q_{ee} the number of endpoint equality constraints. The notation $n\%m$ means the remainder after division of n by m (n modulo m). Thus, for trajectory constraints, the ν-th constraint (with $\nu = n\%(N+1)+1$) is evaluated at time t_k.

If `action=2`, only `du` and `dx0` are returned. If `action=3`, `du`, `dx0` and `p` are returned. The function, **eval_fnc**, provides a convenient interface to this form.

[xdot,zdot] = simulate(3,x,u,t,{f_num,{k}})
[xdot,zdot,pdot] = simulate(3,x,u,t,p,{k})

This form evaluates (as opposed to integrates) the following quantities: $\dot{x} = h(t,x,u)$, $\dot{z} = l_o^\nu(t,x,u)$, and $\dot{p} = -(\frac{\partial h(t,x,u)^T}{\partial x} p + \frac{\partial l(t,x,u)^T}{\partial x})$ at the times specified by `t`. These functions are evaluated at the points in `t`. If `f_num` is specified, ν = `f_num`, otherwise $\nu = 1$. The function $l^\nu(\cdot,\cdot,\cdot)$ is evaluated according to Table S5 above. The last input, `k`, can only be supplied if `t` is a single time point. It is used to indicate the discrete-time interval containing `t`. That is, `k` is such that $t_k \le$ `t` $< t_{k+1}$. If `k` is given, **l** is called with `neq[3]` $= k - 1$. In this call, the values in `u` represent pointwise values of $u(t)$, not its spline coefficients. The inputs `x` and `u` must have the same number of columns as `t`.

[h_x,h_u,l_x,l_u] = simulate(4,x,u,t,{f_num,{k}})

This form evaluates $\frac{\partial h(t,x,u)}{\partial x}$, $\frac{\partial h(t,x,u)}{\partial u}$, $\frac{\partial l^\nu(t,x,u)}{\partial x}$, and $\frac{\partial l^\nu(t,x,u)}{\partial x}$. In this call, `t` must be a single time point. If `f_num` is specified, ν = `f_num`, otherwise $\nu = 1$. The function $l^\nu(\cdot,\cdot,\cdot)$ is evaluated according to Table S5 above. The last input, `k`, indicates the discrete-time interval containing `t`. That is, `k` is such that $t_k \le$ `t` $< t_{k+1}$. If `k` is given, **l** is called with `neq[3]` $= k - 1$. In this call, the values in `u` represent pointwise values of $u(t)$, not its spline coefficients.

[g,g_x0,g_xf] = simulate(5,x0,xf,tf,{f_num})

This form evaluates $g^\nu(\mathtt{x0},\mathtt{xf})$, $\frac{\partial g^\nu(\mathtt{x0},\mathtt{xf})}{\partial \mathtt{x0}}$, and $\frac{\partial g^\nu(\mathtt{x0},\mathtt{xf})}{\partial \mathtt{xf}}$. If `f_num` is specified, ν = `f_num`. Otherwise $\nu = 1$. The input `tf` gets passed to the user functions **g** and **Dg** (see descriptions in §2) for compatibility with future releases of RIOTS_95.

stats = simulate(6)

This form provides statistics on how many times the functions **h** and **Dh** have been evaluated, how many times the system has been simulated to produce the trajectory x, and how many times functions or the gradients of $f^\nu(\cdot,\cdot)$, $g^\nu(\cdot,\cdot)$ or $l_{ti}^\nu(\cdot,\cdot,\cdot)$ have been computed. The following table indicates what the components of `stats` represent:

Table S6

Component	Meaning
`stats(1)`	Number of calls to **h**.
`stats(2)`	Number of calls to **Dh**.
`stats(3)`	Number of simulations.
`stats(4)`	Number of function evaluations.
`stats(5)`	Number of gradient evaluations.

`lte = simulate(7)`

This form, which must be preceded by a call to **simulate** (form 1) with `ialg=1,2,3,4`, returns estimates of the local truncation error for the fixed step-size Runge-Kutta integration routines. The local truncation error is given, for $k = 1, \ldots, N$, by

$$lte_k = \begin{pmatrix} x_k(t_{k+1}) - x_{N,k+1} \\ z_k(t_{k+1}) - z_{N,k+1} \end{pmatrix},$$

where $x_k(t_{k+1})$ and $z_k(t_{k+1})$ are the solutions of

$$\begin{pmatrix} \dot{x} \\ \dot{z} \end{pmatrix} = \begin{pmatrix} h(x,u) \\ l_o^1(t,x,u) \end{pmatrix}, \quad \begin{matrix} x(t_k) = x_{N,k} \\ z(t_k) = 0 \end{matrix} \quad , \quad t \in [t_k, t_{k+1}] \, .$$

and $x_{N,k+1}$ and $z_{N,k+1}$ are the quantities computed by one Runge-Kutta step from $x_{N,k}$ and 0, respectively. These local truncations errors are estimated using double integration steps as described in [4, Sec. 4.3.1]. The local truncation error estimates are used by **distribute** (see description in §7) to redistribute the integration mesh points in order to increase integration accuracy.

IMPLEMENTATION OF THE INTEGRATION ROUTINES

Here we discuss some of the implementation details of the different integration routines built into **simulate**.

System Simulation

System simulation is accomplished by forward integration of the differential equations used to describe the system. There are four fixed step-size Runge-Kutta integrators, one variable step-size integrator (LSODA), and one discrete-time solver. The RK integrators and LSODA produce approximate solutions to the system of differential equation

$$\dot{x} = h(t, x, u)\ ,\ \ x(a) = \xi$$

$$\dot{z} = l(t, x, u)\ ,\ \ z(a) = 0$$

on the interval $t \in [a, b]$. The four Runge-Kutta integrators are Euler's method, improved Euler, Kutta's formula and the classical Runge-Kutta method (see 7 or [4, Sec. 4.2]) and are of order 1, 2, 3 and 4 respectively. The discrete-time integrator solves

$$x_{k+1} = h(t_k, x_k, u_k)\ ,\ \ x_0 = \xi$$

$$z_{k+1} = l(t_k, x_k, u_k)\ ,\ \ z_0 = 0$$

for $k = 1, \ldots, N$.

The variable step-size integrator is a program called LSODA [8,9]. LSODA can solve both stiff and non-stiff differential equations. In the non-stiff mode, LSODA operates as an Adams-Moulton linear, multi-step method. If LSODA detects stiffness, it switches to backwards difference formulae. When operating in stiff mode, LSODA requires the system Jacobians $\frac{\partial h(t,x,u)}{\partial x}$ and $\frac{\partial l(t,x,u)}{\partial x}$. If the user has not supplied these functions, LSODA must be called using `ialg=6` so that these quantities will be computed using finite-difference approximations. Otherwise, LSODA should be called using `ialg=5` so that the analytic expressions for these quantities will be used.

The integration precision of LSODA is controlled by a relative tolerance and an absolute tolerance. These both default to $1e-8$ but can be specified in `ialg(3:4)` respectively (see description of **simulate**, form 1). The only non-standard aspect of the operation of LSODA by **simulate** is that the integration is restarted at every mesh point t_k due to discontinuities in the control spline $u(\cdot)$, or its derivatives, at these points.

Gradient Evaluation

In this section we discuss the computation of the gradients of the objective and constraint functions of problem **OCP** with respect to the controls and free initial conditions. These gradients are computed via backwards integration of the adjoint equations associated with each function.

Discrete-time Integrator. For the discrete-time integrator, the adjoint equations and gradients are given by the following equations. For the objective functions, $\nu \in \mathbf{q}_o$, $k = N, \ldots, 1$,

$$p_k = h_x(t_k, x_k, u_k)^T p_{k+1} + l_x^\nu(t_k, x_k, u_k)^T\ \ ;\ \ p_{N+1} = \frac{\partial g^\nu(\xi, x_{N+1})^T}{\partial x_{N+1}}$$

$$\left[\frac{df^\nu(\xi,u)}{du}\right]_k^T = h_u(t_k,x_k,u_k)^T p_{k+1} + l_u^\nu(t_k,x_k,u_k)^T$$

$$\frac{df^\nu(\xi,u)^T}{d\xi} = \frac{\partial g^\nu(\xi,x_{N+1})^T}{\partial\xi} + p_0 \ .$$

For the endpoint constraints, $\nu \in \mathbf{q}_{ei} \cap \mathbf{q}_{ee}$, $k = N,\ldots,1$,

$$p_k = h_x(t_k,x_k,u_k)^T p_{k+1} \ \ ; \ \ p_{N+1} = \frac{\partial g^\nu(\xi,x_{N+1})^T}{\partial x_{N+1}}$$

$$\left[\frac{dg^\nu(\xi,x_{N+1})}{du}\right]_k^T = h_u(t_k,x_k,u_k)^T p_{k+1}$$

$$\frac{dg^\nu(\xi,x_{N+1})^T}{d\xi} = \frac{\partial g^\nu(\xi,x_{N+1})^T}{\partial\xi} + p_1 \ .$$

For the trajectory constraints, $\nu \in \mathbf{q}_{ti}$, evaluated at the discrete-time index $l \in \{ 1,\ldots,N+1 \}$,

$$p_k = h_x(t_k,x_k,u_k)^T p_{k+1} \ , \ \ k = l-1,\ldots,1 \ \ ; \ \ p_l = l_x^\nu(t_l,x_l,u_l)^T$$

$$\left[\frac{dl^\nu(t_k,x_k,u_k)}{du}\right]_k^T = \begin{cases} h_u(t_k,x_k,u_k)^T p_{k+1} & k = 1,\ldots,l-1 \\ l_u^\nu(t_k,x_k,u_k)^T & k = l \\ 0 & k = l+1,\ldots,N \end{cases}$$

$$\frac{dl^\nu(t_l,x_l,u_l)^T}{d\xi} = p_1 \ .$$

Runge-Kutta Integrators. For convenience, we introduce the notation

$$u_{k,j} = u(\tau_{k,i_j}) \ , \ \ k = 1,\ldots,N \ , \ \ j = 1,..,r \ ,$$

where

$$\tau_{k,i} \doteq t_k + c_i\Delta_k \ ,$$

and $c_i \in [0,1]$ are parameters of the Runge-Kutta integration method. For RK1, RK2 and RK3, $r = s$ where $s = 1,2,3$ repespectively and $i_j = j$. However, RK4 has a repeated control sample, *cf.* [4, Sec. 2.4], we have $r = 3$, $i_1 = 1$, $i_2 = 2$ and $i_3 = 4$.

The computation of the control gradients is a two-step process. First, the gradient of $f(\xi,u)$ with respect to the control samples $u_{k,j}$, $k = 1,\ldots,N$, $j = 1,\ldots,r$, where r is the number of control samples per integration interval, and with respect to ξ is computed. Second, the gradient with respect to the spline coefficients, α_k, of $u(t)$ is computed using the chain-rule as follows,

$$\frac{df(\xi,u)}{d\alpha_k} = \sum_{i=1}^{N}\sum_{j=1}^{r} \frac{df(\xi,u)}{du_{i,j}}\frac{du_{i,j}}{d\alpha_k} \ , \ \ k = 1,\ldots,N+\rho-1 \ ,$$

where ρ is the order of the spline representation. Most of the terms in the outer summation are zero because the spline basis elements have local support. The quantity

$$\frac{du_{i,j}}{d\alpha_k} = \phi_{\mathbf{t}_N,r,k}(\tau_{i,j})$$

is easily determined from a recurrence relation for the B-spline basis [6].

Due to the special structure of the specific RK methods used by **simulate** there is a very efficient formula, discovered by Hager [10] for computing $df/du_{i,j}$. We have extended Hager's formula to deal with the various constraints and the possibility of repeated control samples (see Chapter 2.4). To describe this formula, we use the notation for $k = 1, \ldots, N-1$ and $j = 1, \ldots, s$,

$$A_{k,j} \doteq h_x(\tau_{k,j}, y_{k,j}, u(\tau_{k,j}))^T \ ,$$

$$B_{k,j} \doteq h_u(\tau_{k,j}, y_{k,j}, u(\tau_{k,j}))^T \ ,$$

$$lx^{\nu}_{k,j} \doteq l^{\nu}_x(\tau_{k,j}, y_{k,j}, u(\tau_{k,j}))^T \ ,$$

and

$$lu^{\nu}_{k,j} \doteq l^{\nu}_u(\tau_{k,j}, y_{k,j}, u(\tau_{k,j}))^T \ .$$

where, with $a_{k,j}$ parameters of the Runge-Kutta method,

$$y_{k,1} \doteq x_k \ , \quad y_{k,j} \doteq x_k + \Delta_k \sum_{m=1}^{j-1} a_{j,m} h(y_{k,m}, u(\tau_{k,m})) \ , \ j = 2, \ldots, s \ .$$

The quantities $y_{k,j}$ are estimates of $x(\tau_{k,j})$.

The gradients of the objective and constraint functions with respect to the controls $u_{k,j}$ and the initial conditions ξ are given as follows. In what follows $a_{s+1,j} \doteq b_j$, $q_{k,s} = q_{k+1,0}$ and the standard adjoint variables, p_k, are given by $p_k = q_{k,0}$. For the objective functions, we have for $\nu \in \mathbf{q}_o$, $k = N, \ldots, 1$,

$$q_{N+1,0} = \frac{\partial g^{\nu}(\xi, x_{N+1})^T}{\partial x_{N+1}} \ ,$$

$$q_{k,j} = q_{k+1,0} + \Delta_k \sum_{m=j+1}^{s} a_{s-j+1,s-m+1}(A_{k,m} q_{k,m} + lx^{\nu}_{k,m+1}) \ , \ \ j = s-1, s-2, \ldots, 0$$

$$\left[\frac{df^{\nu}(\xi, u)}{du}\right]^T_{k,j} = b_j \Delta_k \left(B_{k,j} q_{k,j} + l^{\nu}_{k,j} \right), \ \ j = 1, \ldots, s \ ,$$

$$\frac{df^{\nu}(\xi, u)^T}{d\xi} = \frac{\partial g^{\nu}(\xi, x_N)^T}{\partial \xi} + q^0_1 \ ,$$

For the endpoint constraints, we have for $\nu \in \mathbf{q}_{ei} \cap \mathbf{q}_{ee}$, $k = N, \ldots, 1$,

$$q_{k,j} = q_{k+1,0} + \Delta_k \sum_{m=j+1}^{s} a_{s-j+1,s-m+1} A_{k,m} q_{k,m} \ , \ \ j = s-1, s-2, \ldots, 0 \ ; \ \ q_{N+1,0} = \frac{\partial g^{\nu}(\xi, x_{N+1})^T}{\partial x_{N+1}} \ ,$$

$$\left[\frac{dg^{\nu}(\xi, x_{N+1})}{du}\right]^T_{k,j} = b_j \Delta_k B_{k,j} q_{k,j} \ , \ \ j = 1, \ldots, s$$

$$\frac{dg^\nu(\xi, x_{N+1})^T}{d\xi} = \frac{\partial g^\nu(\xi, x_{N+1})^T}{\partial \xi} + q_1^0 \ ,$$

For the trajectory constraints, $\nu \in \mathbf{q}_{ti}$, evaluated at the the discrete-time index $l \in \{ 1, \ldots, N+1 \}$,

$$q_l^0 = l_x^\nu(t_l, x_l, u(\tau_{l,s}))^T$$

$$q_{k,j} = q_{k+1,0} + \Delta_k \sum_{m=j+1}^{s} a_{s-j+1,s-m+1} A_{k,m} q_{k,m} \ , \ k = l-1, .., 1 \ , \ j = s-1, s-2, \ldots, 0 \ ;$$

$$\left[\frac{dl^\nu(t_k, x_k, u(t_k))}{du} \right]_{k,j}^T = \begin{cases} b_j \Delta_k B_{k,j} q_{k,j} & k = 1, \ldots, l-1 \ , \ j = 1, \ldots, s \\ l_u^\nu(t_k, x_k, u(\tau_{k,j})) & k = l \ ; \ j = 0 \text{ if } l \le N, \text{ else } j = s \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{dl^\nu(t_l, x_l, u(t_l))^T}{d\xi} = q_1^0 \ .$$

For method RK4, we have the special situation that $\tau_{k,2} = \tau_{k,3}$ for all k because $c_2 = c_3 = 1/2$. Hence, there is a repeated control sample: $u_{k,2} = u(\tau_{k,2}) = u(\tau_{k,3})$. Thus, for any function f, the derivatives with respect to $u_{k,1}$, $u_{k,2}$ and $u_{k,3}$ are given by the expressions,

$$\frac{df}{du_{k,1}} = \left[\frac{df}{du} \right]_{k,1} \ , \quad \frac{df}{du_{k,2}} = \left[\frac{df}{du} \right]_{k,2} + \left[\frac{df}{du} \right]_{k,3} \ , \quad \frac{df}{du_{k,3}} = \left[\frac{df}{du} \right]_{k,4} \ .$$

Variable Step-Size Integrator (LSODA). For the variable step-size integrator, LSODA, the adjoint equations and gradients are given by the equations below which require knowledge of $x(t)$ for all $t \in [a, b]$. As in [11], $x(t)$ is stored at the internal knots $\{ t_k + \frac{i}{n_{knots}+1} \Delta_k \}_{i=0,k=1}^{n_{knots}+1,N+1}$ during the forward system integration. By default, $n_{knots} = 1$, but the user can specify $n_{knots} \ge 1$ by setting `ialg(2)` $= n_{\text{knots}}$ (see description of **simulate**, form 1). Then, during the computation of the adjoints and gradients, $x(t)$ is determined by evaluating the quintic[11] Hermite polynomial which interpolates $(t, x(t), \dot{x}(t))$ at the nearest three internal knots within the current time interval $[t_k, t_{k+1}]$. Usually $n_{\text{knots}} = 1$ is quite sufficient.

We now give the formulae for the adjoints and the gradients. It is important to note that, unlike the fixed step-size integrators, the gradients produced by LSODA are not exact. Rather, they are numerical approximations to the continuous-time gradients for the original optimal control problem. The accuracy of the gradients is affected by the integration tolerance and the number of internal knots used to store values of $x(t)$. Under normal circumstances, the gradients will be less accurate than the integration tolerance. For the objective functions, $\nu \in \mathbf{q}_o$,

[11]The order of the Hermite polynomial can be changed by setting the define'd symbol ORDER in the code adams.c. If the trajectories are not at least five time differentiable between breakpoints, then it may be helpful to reduce the ORDER of the Hermite polynomials and increase n_{knots}.

$$\dot{p} = -\,(h_x(t,x,u)^T p + l_x^\nu(t,x,u)^T)\ ,\ \ t \in [a,b]\ \ ;\ \ p(b) = \frac{\partial g^\nu(\xi,x(b))^T}{\partial x(b)}$$

$$\left[\frac{df^\nu(\xi,u)}{d\alpha_k}\right]^T = \int_a^b (h_u(t,x,u)^T p(t) + l_u^\nu(t,x,u)^T)\phi_{\mathbf{t}_N,\rho,k}(t)dt\ ,\ \ k = 1,\ldots,N+\rho-1$$

$$\frac{df^\nu(\xi,u)^T}{d\xi} = \frac{\partial g^\nu(\xi,x(b))^T}{\partial \xi} + p(a)\ .$$

For the endpoint constraints, $\nu \in \mathbf{q}_{ei} \cap \mathbf{q}_{ee}$,

$$\dot{p} = -\,h_x(t,x,u)^T p\ ,\ \ t \in [a,b]\ \ ;\ \ p(b) = \frac{\partial g^\nu(\xi,x(b))^T}{\partial x(b)}$$

$$\left[\frac{dg^\nu(\xi,u)}{d\alpha_k}\right]^T = \int_a^b h_u(t,x,u)^T p(t)\,\phi_{\mathbf{t}_N,\rho,k}(t)dt\ ,\ \ k = 1,\ldots,N+\rho-1$$

$$\frac{dg^\nu(\xi,u)^T}{d\xi} = \frac{\partial g^\nu(\xi,x(b))^T}{\partial \xi} + p(a)\ .$$

For the trajectory constraints, $\nu \in \mathbf{q}_{ti}$, evaluated at time $t = t_l$, $l \in \{\,1,\ldots,N+1\,\}$,

$$\dot{p} = -\,h_x(t,x,u)^T p\ ,\ \ t \in [a,t_l]\ \ ;\ \ p(t_l) = l_x^\nu(t_l,x(t_l),u(t_l))^T$$

$$\left[\frac{dl^\nu(t_l,x_l,u(t_l))}{d\alpha_k}\right]^T = \int_a^{t_l} h_u(t,x,u)^T p(t)\,\phi_{\mathbf{t}_N,\rho,k}(t)dt\ ,\ \ k = 1,\ldots,N+\rho-1$$

$$\frac{dl^\nu(t_l,x_l,u(t_l))^T}{d\xi} = p(a)\ .$$

The numerical evaluation of the integrals in these expressions is organized in such a way that they are computed during the backwards integration of $\dot{p}(t)$. Also, the computation takes advantage of the fact that the integrands are zero outside the local support of the spline basis elements $\phi_{\mathbf{t}_N,\rho,k}(t)$.

check_deriv

Purpose

This function provides a check for the accuracy of the user-supplied derivatives **Dh**, **Dl** and **Dg** by comparing these functions to derivative approximations obtained by applying forward or central finite-differences to the corresponding user-supplied function **h**, **l** and **g**.

Calling Syntax

```
[errorA,errorB,max_error] = check_deriv(x,u,t,{params},{index},
                                         {central},{DISP})
```

Description

The inputs $x \in \mathbb{R}^n$, $u \in \mathbb{R}^m$ and $t \in \mathbb{R}$ give the nominal point about which to evaluate the derivatives $h_x(t,x,u)$, $h_u(t,x,u)$, $l_x^\nu(t,x,u)$, $l_u^\nu(t,x,u)$, $g_x^\nu(t,x,u)$ and $g_u^\nu(t,x,u)$. If there are system parameters (see description of **init** in §3), they must supplied by the input `params`. If specified, `index` indicates the discrete-time index for which `t(index)` $\le$ `t` $\le$ `t(index+1)`. This is only needed if one of the user-supplied system functions uses the discrete-time index passed in `neq[3]`.

The error in each derivative is estimated as the difference between the user-supplied derivative and its finite-difference approximation. For a generic function $f(x)$, this error is computed, with e_i the i-th unit vector and δ_i a scalar, as

$$E = \frac{f(x) - f(x+\delta_i e_i)}{\delta_i} - \frac{df(x)}{dx} e_i \ ,$$

if forward differences are used, or

$$E = \frac{f(x-\delta_i e_i) - f(x+\delta_i e_i)}{2\delta_i} - \frac{df(x)}{dx} e_i \ ,$$

if central differences are used. The perturbation size is $\delta_i = \varepsilon_{\mathrm{mach}}^{1/3} \max\{\,1, |x_i|\,\}$. Central difference approximations are selected by setting the optional argument `central` to a non-zero value. Otherwise, forward difference approximations will be used.

The first term in the Taylor expansion of E with respect to δ_i is of order is $O(\delta_i^2)$ for central differences and $O(\delta_i)$ for forward differences. More details can be found in [12, Sec. 4.1.1]. Thus, it is sometimes useful to perform both forward and central difference approximations to decide whether a large difference between the derivative and its finite-difference approximations is due merely a result of scaling or if it is actually due to an error in the implementation of the user-supplied derivative. If the derivative is correct then E should decrease substantially when central differences are used.

If `DISP=0`, only the maximum error is displayed.

The outputs `errorA` and `errorB` return the errors for $h_x(t,x,u)$ and $h_u(t,x,u)$ respectively. The output `max_error` is the maximum error detected for all of the derivatives.

Example

The following example compares the output from **check_deriv** using forward and central finite-differences. The derivatives appear to be correct since the errors are much smaller when central differences are used. First forward differences are used, then central differences.

```
>> check_deriv([-5;-5],0,0);
========================================================================
              System matrices:
Error in h_x =
   1.0e-04 *
         0   -0.0000
   -0.0000   -0.6358

Error in h_u =
   1.0e-10 *
         0
    0.9421
              For function 1:
Error in l_x =
   1.0e-04 *

   -0.3028         0

Error in l_u =    6.0553e-06

              For function 1:
Error in g_x0 =      0     0

Error in g_xf =      0     0

Maximum error reported is 6.35823e-05.
========================================================================

>> check_deriv([-5;-5],0,0,[],0,1);
========================================================================
              System matrices:
Error in h_x =
   1.0e-10 *
         0   -0.0578
   -0.2355   -0.3833

Error in h_u =
   1.0e-10 *
         0
    0.9421
              For function 1:
Error in l_x =
    1.0e-10 *

    0.5782         0

Error in l_u =      0

              For function 1:
Error in g_x0 =      0     0

Error in g_xf =      0     0

Maximum error reported is 9.42135e-11.
========================================================================
```

check_grad

Purpose

This function checks the accuracy of gradients of the objective and constraint functions, with respect to controls and initial conditions, as computed by **simulate**, forms 1 and 2. It also provides a means to indirectly check the validity of the user-supplied derivative **Dh**, **Dl** and **Dg**.

Calling Syntax

```
max_error = check_grad(i,j,k,x0,u,t,ialg,{params},{central},
                            {DISP})
```

Description

The input `x0`, `u`, `t` and `ialg` specify the inputs to the nominal simulation **simulate(1,x0,u,t,ialg,0)** prior to the computation of the gradients. The gradients are tested at the discrete-time indices as specified in the following table:

Index	Purpose
i	Spline coefficient control u that will be perturbed. If `i=0`, the gradients with respect to u will not be checked.
j	Index of initial state vector, ξ, that will be perturbed. If `j=0`, the gradients with respect to the ξ will not be checked.
k	For each trajectory constraints, `k` indicates the discrete-time index, starting with `k=1`, at which the trajectory constraints will be evaluated. If `k=0`, the trajectory constraint gradients will not be checked.

The finite-difference computations are the same as described for **check_deriv**.

If there are system parameters (see description of **init**, § 3), these must be given by the input `params`. Central difference approximations will be used if a non-zero value for `central` is specified; otherwise forward differences will be used. If `DISP=0`, only the maximum error is displayed. This is particularly useful if **check_deriv** is used in a loop on any of the indices `i,j,k`. The output `max_error` is the maximum error detected in the gradients.

Example

The following example checks the tenth component of the control gradient and the second component of initial condition gradient as computed by RK2 using central differences. The integration is performed on the time interval $t \in [0, 2.5]$ with $N = 50$ intervals. The gradients are evaluated for the second order spline control $u(t) = 1$ for all t (*i.e.*, $\alpha_k = 1$, $k = 1, \ldots, N+1$).

```
>> t = [0:2.5/50:2.5];
>> u = ones(1,51);
>> x0 = [-5;-5];
>> check_grad(10,2,0,x0,u,t,2,[],1);

============================================================================
Using perturbation size of 6.05545e-06

                    Evaluating function 1.
error_u  = 1.84329e-09
error_x0 = -4.88427e-11
Relative error in control gradient = 2.52821e-07%
Gradient OK

Relative error in x0 gradient = 1.14842e-09%
Gradient OK

                    Evaluating endpoint constraint 1.
error_u  = -5.46737e-11
error_x0 = -5.98271e-12
Relative error in control gradient = 6.04337e-08%
Gradient OK

Relative error in x0 gradient = 1.87846e-09%
Gradient OK
Maximum error reported is 1.84329e-09.
============================================================================
```

eval_fnc

Purpose

This function provides a convenient interface to **simulate** (form 2), for computing function and gradient values. A system simulation must already have been performed for this function to work.

Calling Syntax

```
[f,du,dx0,p] = eval_fnc(type,num,k)
```

Description of Inputs

`type` A string that specifies the type of function to be evaluated. The choices are

`'obj'`	Objective function
`'ei'`	Endpoint inequality constraint
`'ee'`	Endpoint equality constraint
`'traj'`	Trajectory constraint

`num` Specifies ν for the function of the type specified by `type` is to be evaluated.

`k` For trajectory constraints only. Specifies the index for the time, t_k, in the current integration mesh at which to evaluate the trajectory constraint. If `k` is a vector, the trajectory constraint will be evaluated at the times specified by each mesh point index in `k`.

Description of Outputs

`f` The function value.

`du` The gradient with repect to u. Not computed for trajectory constraints if `index` is a vector.

`dx0` The derivative of the function with respect to initial conditions, ξ. Not computed for trajectory constraints if `index` is a vector.

`p` The adjoint trajectory. Not computed for trajectory constraints if `index` is a vector.

Examples

The following examples assume that a simulation has already been performed on a system that has at least two endpoint equality constraints and a trajectory constraint. The first call to **eval_fnc** evaluates the second endpoint equality constraint.

```
>> f=eval_fnc('ee',2)

f =

    0.2424
```

Since equality constraints should evaluate to zero, this constraint is violated. This next call evaluates the first trajectory constraint at the times t_k, $k = 5, \ldots, 15$, in the current integration mesh.

```
>> eval_fnc('traj',1,5:15)

ans =

  Columns 1 through 7

   -1.0182   -1.0222   -1.0258   -1.0288   -1.0311   -1.0327   -1.0338

  Columns 8 through 11

   -1.0335   -1.0318   -1.0295   -1.0265
```

Since inequality constraints are satisfied if less than or equal to zero, this trajectory constraint is satisfied at these specified points.

6. OPTIMIZATION PROGRAMS

This section describes the suite of optimization programs that can be used to solve various cases of the optimal control problem **OCP**. These programs seek local minimizers to the discretized problem. The most general program is **riots** which converts **OCP** into a mathematical program which is solved using standard nonlinear programming techniques. Besides being able to solve the largest class of optimal control problems, **riots** is also the most robust algorithm amongst the optimization programs available in RIOTS_95. However, it can only handle medium size problems. The size of a problem, the number of decision variables, is primarily determined by the number of control inputs and the discretization level. What is meant by medium size problems is discussed in the description of **riots**.

The most restrictive program is **pdmin** which can solve optimal control problems with constraints consisting of only simple bounds on ξ and u. State constraints are not allowed. The algorithm used by **pdmin** is the projected descent method described in [4, Chap. 3]. Because of the efficiency of the projected descent method, **pdmin** can solve large problems.

Problems that have, in addition to simple bounds on u and ξ, endpoint equality constraints can be solved by **aug_lagrng**. The algorithm is a multiplier method which relies upon **pdmin** to solve a sequence of problems with only simple bound constraints. This program provides a good example of how the toolbox style of RIOTS_95 can be used to create a complex algorithm from a simpler one. Currently, the implementation of **aug_lagrng** is fairly naive and has a great deal of room left for improvement. Also, it would be relatively straightforward to add an active set strategy to **aug_lagrng** in order to allow it to handle inequality constraints.

Finally, the program **outer** is an experimental outer loop which repeatedly calls **riots** to solve a sequence of increasingly accurate discretizations (obtained by calls to **distribute**) of **OCP** in order to efficiently compute the optimal control to a specified accuracy.

Choice of Integration and Spline Orders.

Each of these optimization programs requires the user to select an integration routine and the order of the spline representation for the controls. There are several factors involved in these selections. Some of these factors are discussed below and summarized in the Table O2 that follows. Consult [4, Chap 4.2] for a more in-depth discussion.

Fixed step-size integration. The first consideration is that, for each of the fixed step-size Runge-Kutta methods, there is a limit to how much accuracy can be obtained in the control solutions at certain discrete time points. The accuracy, $\|u_N{}^* - u^*\|$, of the control splines can not be greater than the solutions at these time points. The order of the accuracy of spline solutions with respect to the discretization level for *unconstrained* problems is given in the following table. The quantity Δ used in this table is defined as $\Delta \doteq \max_k t_{N,k+1} - t_{N,k}$. The third column is a reminder of the spline orders that are allowed by **simulate** for each RK method.

Table O1

RK Method	Order of Accuracy	Allowable Spline Orders
1	$O(\Delta^1)$	1
2	$O(\Delta^2)$	2
3	$O(\Delta^2)$	2
4	$O(\Delta^3)$	2, 3, 4

While it is possible with some optimal control problems to achieve higher order accuracies, this is a non-generic situation. The order of spline representation should therefore not exceed the accuracies listed in the second column of this table. Thus, for RK4, even though cubic splines are allowed there is usually no reason to use higher than quadratic splines ($\rho = 3$).

The orders listed in the above table are usually only achieved for unconstrained problems. For problems with control constraints it is typically impossible to achieve better than first order accuracy. This is even true if the discontinuities in the optimal control are known *a priori* since the locations of these discontinuities will not coincide with the discontinuities of the discretized problems. For problems with state constraints, the issue is more complicated. In general, we recommend using second order splines (except for Euler's method) for problems with control and/or trajectory constraints. Even if first order accuracy is all that can be achieved, there is almost no extra work involved in using second order splines. Furthermore, second order splines will often give somewhat better results than first order splines even if the accuracy is asymptotically limited to first order.

A second consideration is that the overall solution error is due to both the integration error and the error caused by approximating an infinite dimensional function, the optimal control, with a finite dimensional spline. Because of the interaction of these two sources of error and the fact that the accuracy of the spline representations is limited to the above table, improving the integration accuracy by using a higher order method does not necessarily imply that the accuracy of the solution to the approximating problem will improve. However, even if the spline accuracy is limited to first order, it is often the case that the integration error, which is of order $O(\Delta^s)$, where s is the order of the RK method, still has a significantly greater effect on the overall error than the spline error (especially at low discretization levels). This is partly due to the fact that errors in the control are integrated out by the system dynamics. Thus, it is often advantageous to use higher-order integration methods even though the solution error is asymptotically limited to first order by the spline approximation error.

The importance of the RK order, in terms of reducing the overall amount of computational work required to achieve a certain accuracy, depends on the optimization program being used. Each iterations of **riots** requires the solution of one or more dense quadratic program. The dimension of these quadratic programs is equal to the number of decision parameters (which is $m(N + \rho - 1)$ plus the number of free initial conditions). Because the work required to solve a dense quadratic program goes up at least cubically with the number of decision variables, at a certain discretization level most of the work at each iteration will be spent solving the quadratic program. Thus, it is usually best to use the fourth order RK method to achieve as much accuracy as possible for a given discretization level. An exception to this rule occurs when problem **OCP** includes trajectory constraints. Because a separate gradient calculation is performed at each mesh point for each trajectory constraint, the amount of work increases significantly as the integration order is increased. Thus, it may be beneficial to use a RK3 or even RK2 depending on the problem.

On the other hand, for the optimization programs **pdmin** and **aug_lagrng** (which is based on **pdmin**) the amount of work required to solve the discretized problem is roughly linear in the number of decision variables which is basically proportional to the discretization level N. The amount of work required to integrate the differential equations is linearly proportional to Ns where s the order of the Runge-Kutta method. Since the integration error is proportional to $1/N^s$, if not for the error for the spline approximation it would always be best to use RK4. However, because there is error from the finite dimensional spline representation, it does not always pay to use the highest order RK method. If, roughly speaking, the error from the control representation contributes to the overall error in the numerical solution to larger extent than the integration error (note that the spline error and the integration error are in different units) then it is wasteful to use a higher order RK method. This usually happens only at high discretization levels.

The relative effect of the spline error versus the integration error depends on the nature of the system dynamics and the smoothness of the optimal control. Unfortunately, this is hard to predict in advance. But a sense of the balance of these errors can be obtained by solving, if possible, the problem at a low discretization level and viewing the solution using **sp_plot** and using **simulate** (form 7) or **est_errors** to obtain an estimate of the integration error.

There is a third consideration for selecting the integration order. For some problems with particularly nonlinear dynamics, in may not be possible integrate the differential equation if the discretization level is too small. In these cases, the minimum discretization level needed to produce a solution is smallest when using RK4. For some problems, it may not be possible to achieve an accurate solution of the differential equation at any reasonable discretization level. For these problems, the variable step-size integration method, discussed next, will have to be used.

Regardless of the integration method used, higher order splines ($\rho > 2$) should not be used unless the optimal control is sufficiently smooth. Of course, the optimal control is not known in advanced. Generally, though, when solving control and/or trajectory constrained problems, second order splines should be used (except with Euler's method which can only use first order splines) as per the discussion above. For other problems being integrated with RK4, it may be advantageous to use quadratic splines.

The following table provides a set of basic guidelines for the selection of the integration method and the spline order for solving different classes of problems. These choices may not be ideal for any specific problem but they are generally acceptable for most problems.

Table O2

type of problem	optimization program	RK order (`ialg`)	spline order (ρ)
no control nor trajectory constraints	**pdmin/aug_lagrng**	4 (N small) 2 (N large)	3 (N small) 2 (N large)
	riots	4	3
control constraints	**pdmin/aug_lagrng**	4 (N small) 2 (N large)	2
	riots	4	
trajectory constraints	**riots**	2[12]	2

Variable step-size integration. From the point of view of integrating differential equations, it is much more efficient to use a variable step-size integration routine than a fixed step-size method. However, this is usually not the case when solving optimal control problems. There are three basic reasons for this. First, the overall solution accuracy cannot exceed the accuracy with which splines can approximate the optimal control. Thus, it is quite conceivable that a great deal of work will be spent to achieve a very accurate integration, which is often necessary to ensure successful line searches, but this effort will be wasted on a relatively inaccurate solution. Second, the solution of the discretized problem can easily involve hundreds of simulations. The integration accuracy during most of the simulations will have very little affect on the accuracy of the final solution. Therefore, it is usually much more efficient to solve a sequence of discretized problems, each with a more accurate integration mesh, using a fast, fixed step-size integration method. Third, the gradients produced for the variable step-size method are approximations to the actual, continuous-time gradients for the original problem **OCP**; they are not exact gradients for the discretized problems. Thus, the solution of the discretized problem will usually require more iterations and will be less accurate (relative to the actual solution of the discretized problem) when using the variable step-size method than when using one of the fixed step-size integration routines. Again, otherwise useless integration accuracy is required to produce sufficiently accurate gradient approximations.

There are, however, situations in which it is best to use the variable step-size integration method. The first situation is when the system dynamics are very difficult to integrate. In this case, or any other case in which the the integration error greatly exceeds the spline approximation error, it is more efficient to use the variable step-size method. In some cases, the integration has to be performed using the variable step-size method. This can occur if the system is described by stiff differential equations or if the system contains highly unstable dynamics.

Another situation in which it can be advantageous to use the variable step-size integration method is if the location of discontinuities in the optimal control, or discontinuities in the derivatives of the optimal control, are known *a priori*. In this case, it may be possible to increase the solution accuracy by placing breakpoints in the discretization mesh where these discontinuities occur and then using a spline of order one greater than the overall smoothness of the optimal control[13]. The location of the discontinuity for the discretized problem will be very close to the discontinuity in the optimal control if the integration tolerance is small and the optimal control is well-approximated by the spline away from the discontinuity. Hence, the overall accuracy will not be limited by the discontinuity.

The variable step-size integration routine can use first, second, third, or fourth order splines. For unconstrained problems, or problem with endpoint constraints, it is best to use fourth order splines so that the spline approximation error is as small as possible. For problems with control and/or trajectory constraints, first or second order splines are recommended.

Coordinate Transformation

All of the optimization programs in RIOTS_95 solve finite-dimensional approximations to **OCP** obtained by the discretization procedure described in the introduction of §5. Additionally, a change of basis is performed for the spline control subspaces. The new basis is orthonormal. This change of basis is accomplished by computing the matrix $\mathbf{M}_\alpha$ with the property that for any two splines $u(\cdot)$ and $v(\cdot)$ with coefficients α and β,

---

[12]Sometimes a higher-order method must be used to provide a reasonable solution to the system differential equations.

[13]A spline of higher order would be too smooth since RIOTS_95 currently does not allow splines with repeated interior knots.

$$\langle u, v \rangle_{L_2} = \langle \alpha \mathbf{M}_\alpha, \beta \rangle ,$$

(recall that α and β are row vectors. The splines coefficients in the transformed basis are given by $\tilde{\alpha} = \alpha \mathbf{M}_\alpha^{1/2}$ and $\tilde{\beta} = \beta \mathbf{M}_\alpha^{1/2}$. In the new coordinates,

$$\langle u, v \rangle_{L_2} = \langle \tilde{\alpha} , \tilde{\beta} \rangle .$$

In words, the L_2-inner product of any two splines is equal to the Euclidean inner product of their coefficients in the new basis. The matrix $\mathbf{M}_\alpha$ is referred to as the *transform matrix* and the change of basis is referred to as the *coordinate transformation*.

By performing this transformation, the standard inner-product of decision variables (spline coefficients) used by off-the-shelf programs that solve mathematical programs is equal to the function space inner product of the corresponding splines. Also, because of the orthonormality of the new basis, the conditioning of the discretized problems is no worse than the conditioning of the original optimal control problem **OCP**. In practice, this leads to solutions of the discretized problems that are more accurate and that are obtained in fewer iterations than without the coordinate transformation. Also, any termination criteria specified with an inner product become independent of the discretization level in the new basis.

In effect, the coordinate transformation provides a natural column scaling for each row of control coefficients. It is recommended that, if possible, the user attempt to specify units for the control inputs so that the control solutions have magnitude of order one. Choosing the control units in this way is, in effect, a row-wise scaling of the control inputs.

One drawback to this coordinate transformation is that for splines of order two and higher the matrix $\mathbf{M}_\alpha^{-1/2}$ is dense. A diagonal matrix would be preferable for two reasons. First, computing $\mathbf{M}_\alpha^{-1/2}$ is computationally intensive for large N. Second, there would be much less work involved in transforming between bases: each time a new iterate is produced by the mathematical programming software, it has to be un-transformed to the original basis. Also, every gradient computation involves an inverse transformation. Third, simple control bounds are converted into general linear constraints by the coordinate transformation. This point is discussed next.

Control bounds under the coordinate transformation. Simple bounds on the spline coefficients takes the form $a_k \le \alpha_k \le b_k$, $k = 1, .., N + \rho - 1$. If a_k and b_k are in fact constants, a and b, then for all t, $a \le u(t) \le b$. Now, under the coordinate transformation, simple bounds of this form become

$$(a_1 , \ldots , \alpha_{N+\rho-1}) \le \tilde{\alpha}\, \mathbf{M}_\alpha^{-1/2} \le (b_1 , \ldots , b_{N+\rho-1}) .$$

Thus, because of the coordinate transformation, the simple bounds are converted into general linear bounds. Since this is undesirable from an efficiency point of view, RIOTS_95 instead replaces the bounds with

$$(a_1 , \ldots , \alpha_{N+\rho-1}) \mathbf{M}_\alpha^{1/2} \le \tilde{\alpha} \le (b_1 , \ldots , b_{N+\rho-1}) \mathbf{M}_\alpha^{1/2} .$$

For first order splines (piecewise constant), these bounds are equivalent to the actual bounds since $\mathbf{M}_\alpha^{1/2}$ is diagonal. For higher order splines, these bounds are not equivalent. They are, however, approximately correct since the entries of the matrix $\mathbf{M}_\alpha$ fall off rapidly to zero away from the diagonal.

It turns out that the problems enumerated above can be avoided when using second order splines (piecewise linear) which are, in any case, the recommended splines for solving problems with control bounds. Instead of using $\mathbf{M}_\alpha$ in the coordinate transformation, the diagonal matrix

$$\mathbf{M} = \begin{bmatrix} \Delta_1 & & & & & \\ & \frac{\Delta_1 + \Delta_2}{2} & & & & \\ & & \frac{\Delta_2 + \Delta_3}{2} & & & \\ & & & \ddots & & \\ & & & & \frac{\Delta_{N-1} + \Delta_N}{2} & \\ & & & & & \Delta_N \end{bmatrix},$$

with $\Delta_k \doteq t_{N,k+1} - t_{N,k}$, is used. This transformation matrix is derived in [4, Sec. 2.7] and retains the important attributes of the transformation given by $\mathbf{M}_\alpha$. In **riots** and **pdmin**, $\mathbf{M}$ is used for the coordinate transformation, instead of $\mathbf{M}_\alpha$, when second order splines are used if *(i)* problem **OCP** has control bounds, *(ii)* RK2 is being used as the integration method, or *(iii)* $N > 300$. The latter case is employed because the time it takes to compute the transform becomes excessive when N is large. When $\rho > 2$, the transformation is skipped altogether if *(i)* $N > 300$ or *(ii)* LSODA is being used on a problem with control bounds[14].

[14]Recall from Table S3 (p. 36) that splines of order greater than 2 can only be used with RK4 and LSODA.

Description of the Optimization Programs

The first six inputs are the same for all of the optimization programs; they are listed in the following table. Default values for vectors apply to each component of that vector. Specifying `[]` for an input causes that input to be set to its default value. In the following, N is the discretization level and ρ is the order of the control splines.

Table O3

Input	Rows	Columns	Description
`x0`	n	1, 2 or 4	`x0=[x0,{fixed,{x0min,x0max}}]` where `x0` is the nominal value of the initial state ξ. `fixed` For each i such that `fixed(i)=0`, the corresponding initial state value ξ^i is treated as a free decision variable. Default: 1 `x0min` Specifies lower bound for each free initial condition ξ^i. Default: $-\infty$ `x0max` Specifies upper bound for each free initial condition ξ^i. Default: ∞
`u0`	m	$N+\rho-1$	Initial guess for the spline coefficients of the control u.
`t`	1	$N+1$	The integration mesh points/spline breakpoints.
`Umin`	m	$N+\rho-1$ or 1	Lower bounds on the spline coefficients for u. If `Umin` is specified as a single column, its values will apply as a lower bound on all of the spline coefficients. Default: $-\infty$
`Umax`	m	$N+\rho-1$ or 1	Upper bounds on the spline coefficients for u. If `Umax` is specified as a single column, its values will apply as an upper bound on all of the spline coefficients. Default: ∞
`params`	p	1	Provides the system parameters if required.

The first two outputs are the same for all of the optimization programs; they are listed in the following table:

Table O4

Output	Rows	Columns	Description
`u`	m	$N+\rho-1$	The optimal control solution.
`x`	n	$N+1$	The optimal state trajectory solution.

aug_lagrng

Purpose

This function uses **pdmin** as an inner loop for an augmented Lagrangian algorithm that solves optimal control problem with, in addition to simple bounds on ξ and u, endpoint equality constraints. Only one objective function is allowed.

The user is urged to check the validity of the user-supplied derivatives with the utility program **check_deriv** before attempting to use **pdmin**.

Calling Syntax

```
[u,x,f,lambda,I_i] = aug_lagrng([x0,{fixed,{x0min,x0max}}],u0,t,
                                  Umin,Umax,params,N_inner,N_outer,
                                  ialg,{method},{[tol1,tol2]},{Disp})
```

Description of the Inputs

The first six inputs are described in Table O3.

`N_inner` Maximum number of iterations for each inner loop call to **pdmin**.

`N_outer` Maximum number of outer iterations.

`ialg` Specifies the integration algorithm used by **simulate**.

`method` Specifies the method for computing descent directions in the unconstrained subspace. The choices are explained in the description of **pdmin**. Default: `'vm'`.

`tol1,tol2` Optimality tolerances. Default: [$\varepsilon_{\text{mach}}^{1/2}$, $\varepsilon_{\text{mach}}^{1/3}$]. The outer loop terminates if

$$\|\nabla f(\eta) - \sum_{\nu=1}^{q_{ee}} \lambda_\nu \nabla g_{ee}^{\nu}(\eta)\| \le \texttt{tol1}(1 + |f(\eta)|)$$

and

$$\max_{\nu \in \mathbf{q}_{ee}} |g_{ee}^{\nu}(\eta)| \le \texttt{tol2}\,.$$

`Disp` Passed on to **pdmin** to control amount of displayed output. Default: 0.

Description of the Outputs

The first two outputs are described in Table O4.

`f` The objective value at the obtained solution.

`I_i` Index set of elements of $[u(:)\,;\,\xi]$ that are not at their bounds.

`lambda` Vector of Lagrange multipliers associated with the endpoint equality constraints.

Description of the Algorithm

This program calls **pdmin** to minimize a sequence of augmented Lagrangian functions of the form

$$L_{c,\lambda}(\eta) = f(\eta) - \sum_{\nu=1}^{q_{ee}} \lambda_\nu g_{ee}^\nu(\eta) + \frac{1}{2}\sum_{\nu=1}^{q_{ee}} c_\nu g_{ee}^\nu(\eta)^2$$

subject to simple bounds on ξ and u. The value of the augmented Lagrangian and its gradient are supplied to **pdmin** by **a_lagrng_fnc** via extension 1 (see description of **pdmin**).

The values of the Lagrange multiplier estimates λ_ν, $\nu = 1, \ldots, q_{ee}$, are determined in one of two ways depending on the setting of the internal variable `METHOD` in aug_lagrng.m. Initially $\lambda_\nu = 0$.

Multiplier Update Method 1. This method adjusts the multipliers at the end of each iteration of **pdmin** by solving the least-squares problem

$$\lambda = \min_{\lambda \in \mathbb{R}^{q_{ee}}} \|\nabla f(\eta) - \sum_{\nu=1}^{q_{ee}} \lambda_\nu \nabla g_{ee}^\nu(\eta)\|_{I\_i}^2 ,$$

where the norm is taken only on the uncstrained subspace of decision variables which is indicated by the index set `I_i`. This update is performed by **multiplier_update** which is called by **pdmin** via extension 2. If update method 1 is used, the tolerance requested for the inner loop is decreased by a factor of ten on each outer iteration starting from $10^{\min\{6,\texttt{N\_outer}\}}\varepsilon_{\mathrm{mach}}^{1/2}$ until the tolerance is $\varepsilon_{\mathrm{mach}}^{1/2}$.

Multiplier Update Method 2. This method is the standard "method of multipliers" which solves the inner loop completely and then uses the first order multiplier update

$$\lambda_\nu \leftarrow \lambda_\nu - c_\nu g_{ee}^\nu(\eta) \quad , \forall\, \nu \in I_\nu$$

where

$$I_\nu \doteq \{\, \nu \in \mathbf{q}_{ee} \mid |g_{ee}^\nu(\eta)| \le \tfrac{1}{4}\, |g_{ee}^\nu(\eta_{\mathrm{previous}})| \text{ or } |g_{ee}^\nu(\eta)| \le \texttt{tol2} \,\} \ .$$

If update method 2 is used, the tolerance requested for the inner loop is fixed at $\varepsilon_{\mathrm{mach}}^{1/2}$.

Penalty Update. The initial values for the constraint violation penalties are $c_\nu = 1$, $\nu = 1, \ldots, q_{ee}$. It may be helpful to use larger initial values for highly nonlinear problems. The penalties are updated at the end of each outer iteration according to the rule

$$c_\nu \leftarrow 10 c_\nu \quad , \forall\, \nu \notin I_\nu \ ,$$

where I_ν is as defined above.

Note that this algorithm is implemented mainly to demonstrate the extensible features of **pdmin** and is missing features like, *(i)* constraint scaling, *(ii)* an active set method for handling inequality endpoint constraints, *(iii)* a mechanism for decreasing constraint violation penalties when possible and, most importantly, *(iv)* an automatic mechanism for setting the termination tolerance for each call to **pdmin**.

Notes:

1. On return from a call to **aug_lagrng**, the variable `opt_program` will be defined in the Matlab workspace. It will contain the string `'aug_lagrng'`.

See Also: **pdmin**, a_lagrng_fnc.m, multiplier_update.m.

outer

Purpose

This program calls **riots** to solve problems defined on a sequence of different integration meshes, each of which result in a more accurate approximation to **OCP** than the previous mesh. The solution obtained for one mesh is used as the starting guess for the next mesh.

The user is urged to check the validity of the user-supplied derivatives with the utility program **check_deriv** before attempting to use **pdmin**.

Calling Syntax

```
[new_t,u,x,J,G,E] = outer([x0,{fixed,{x0min,x0max}}],u0,t,
                          Umin,Umax,params,N_inner,[N_outer,{max_N}]
                          ialg,{[tol1,tol2,tol3]},{strategy},{Disp})
```

Description of the Inputs

The first six inputs are described in Table O3.

`N_inner` Maximum number of iterations for each inner loop of **riots**.

`N_outer` Maximum number of outer iterations.

`max_N` The maximum discretization level; **outer** will terminate if the discretization level exceeds `max_N`. Default: ∞

`ialg` Specifies the integration algorithm used by **simulate**.

`tol1,tol2,tol3` Optimality tolerances. The outer loop terminates if

$$\|\nabla L(\eta)\|_I \ \le\ \texttt{tol1}(1 + |f(\eta)|)\ ,$$

where $\|\nabla L(\eta)\|_I$ is the H_2-norm of the free portion of $\nabla L(\eta)$,

$$\max_{\nu\ \in\ \mathbf{q}_{ee}} |g^{\nu}_{ee}(\eta)| \ \le\ \texttt{tol2}\ ,$$

and

$$\|u_N - u^*\| \le \texttt{tol3}(1 + \|u_N\|_\infty)b\ ,$$

where b is the nominal final time. The default values for these tolerances factors are $[\varepsilon_{\text{mach}}^{1/3}, \varepsilon_{\text{mach}}^{1/4}, \varepsilon_{\text{mach}}^{1/6}]$.

`strategy` Passed on to **distribute** to select the mesh redistribution strategy. Default = 3.

`Disp` Passed on to **riots** to control amount of displayed output. Default = 1.

Description of the Outputs

The first two outputs are described in Table O4.

`new_t` The final integration mesh obtained from the final mesh redistribution.

`u` The optimal control solution defined on the final mesh `new_t`.

`x` The optimal trajectory solution.

`J` A row vector whose i-th component is the value of the objective function, computed using LSODA, after the i-th call to **riots**.

`G` A row vector whose i-th component is the sum of the constraint violations, computed using LSODA, after the i-th call to **riots**.

`E` A row vector whose i-th component is an estimate of $\|\eta_N - \eta^*\|_{H_2}$ after the $(i+1)$-th iteration. With $\eta = (u, \xi)$, $\|\eta\|_{H_2}$ is defined by

$$\|\eta\|_{H_2} \doteq \left[\|\xi\|_2 + \int_a^b \|u(t)\|_2^2 dt \right]^{1/2} .$$

Description of Algorithm

outer is an outer loop for **riots**. During each iteration, **riots** is called to solve the discretized problem on the current mesh starting from the solution of the previous call to **riots** interpolated onto the new mesh. After **riots** returns a solution, **est_errors** and **control_error** are called to provide estimates of certain quantities that are used to determine whether **outer** should terminate or if it should refine the mesh. If necessary, the mesh is refined by **distribute**, with `FAC=10`, according to `strategy` except following the first iteration. After the first iteration, the mesh is always doubled.

After each iteration, the following information is displayed: the H_2-norm of the free portion of the gradient of the Lagrangian, the sum of constraint errors, objective function value, and integration error of the integration algorithm `ialg` at the current solution. All of these quantities are computed by **est_errors**. The first three values are estimates obtained using LSODA with a tolerance set to about one thousandth of the integration error estimate. The control solution is plotted after each iteration (although the time axis is not scaled correctly for free final time problems).

Additionally, following all but the first iteration, the change in the control solution from the previous iteration and an estimate of the current solution error, $\|\eta_N^* - \eta^*\|_{H_2}$, are display.

Notes:

1. If solutions exhibit rapid oscillations it may be helpful to add a penalty on the piecewise derivative variation of the control by setting the variable `VAR` in outer.m to a small positive value.

2. The factor by which **distribute** is requested to increase the integration accuracy after each iteration can be changed by setting the variable `FAC` in outer.m.

3. An example using **outer** is given in Session 4 (§3).

See Also: **riots**, **distribute**, **est_errors**, **control_error**.

pdmin

Purpose

This is an optimization method based on the projected descent method [3]. It is highly efficient but does not solve problems with general constraints or more than one objective function.

The user is urged to check the validity of the user-supplied derivatives with the utility program **check_deriv** before attempting to use **pdmin**.

Calling Syntax

```
[u,x,J,inform,I_a,I_i,M] = pdmin([x0,{fixed,{x0min,x0max}}],u0,t,
                                Umin,Umax,params,[miter,{tol}],
                                ialg,{method},{[k;{scale}]},{Disp})
```

Description of Inputs

The first six inputs are described in Table O3. The remainder are described here.

`miter` The maximum number of iterations allowed.

`tol` Specifies the tolerance for the following stopping criteria

$$\|g_k\|_{I_k} / |I_k| < \mathtt{tol}^{2/3}(1 + |\mathrm{f}(\eta_\mathrm{k})|) \ ,$$

$$f(u_k) - f(u_{k-1}) < 100\mathtt{tol}(1 + |\mathrm{f}(\mathrm{u_k})|) \ ,$$

$$\|u_k - u_{k-1}\|_\infty < \mathtt{tol}^{1/2}(1 + \|u_k\|_\infty) \ ,$$

$$x_k^i = 0 \ \ , \forall\, i \in A_k \ ,$$

where g_k is the k-th component of the derivative of $f(\cdot)$ in transformed coordinates, I_k is set of inactive bound indices and A_k is set of active bound indices. Default: $\varepsilon_{\mathrm{mach}}^{1/2}$.

`ialg` Specifies the integration algorithm used by **simulate**.

`method` A string that specifies the method for computing descent directions in the unconstrained subspace. The choices are:

`''`	limited memory quasi-Newton (L-BFGS)
`'steepest'`	steepest descent
`'conjgr'`	Polak-Ribière conjugate gradient method
`'vm'`	limited memory quasi-Newton (L-BFGS)

The default method is the L-BFGS method.

`k` This value is used to determine a perturbation with which to compute an initial scaling for the objective function. Typically, `k` is supplied from a previous call to **pdmin** or not at all.

`scale` This value is used to determine a perturbation with which to compute an initial function scaling. Typically, `scale` is supplied from a previous call to **pdmin** or not at all.

`Disp` — `Disp = 0,1,2` controls the amount of displayed output with 0 being minimal output and 2 being full output. Default: 2.

Description of Outputs

The first two outputs are described in Table O4.

`J` — A row vector whose $(i+1)$-th component is the value of the objective function at the end of the i-th iteration. The last component of `J` is the value of the objective function at the obtained solution.

`I_a` — Index set of elements of $[u(:)\,;\,\xi]$ that are actively constrained by bounds.

`I_i` — Index set of elements of $[u(:)\,;\,\xi]$ that are not constrained by bounds.

`inform` — This is a vector with four components:

- `inform(1)` Reason for termination (see next table).
- `inform(2)` Function space norm of the free portion of $\nabla f(\eta)$, $\eta = (u, \xi)$.
- `inform(3)` Final step-size $k = \log\lambda / \log\beta$ where λ is the Armijo step-length and $\beta = 3/5$.
- `inform(4)` The value of the objective function scaling.

The possible termination reasons are:

`inform(1)`	Cause of Termination.
-1	Simulation produced NaN or Inf.
0	Normal termination tests satisfied.
1	Completed maximum number of iterations.
2	Search direction vector too small.
3	All variables at their bounds and going to stay that way.
4	Gradient too small.
5	Step-size too small.
6	User test satisified (user test returned 2).

Description of Displayed Output

Depending on the setting of `Disp`, **pdmin** displays a certain amount of information at each iteration. This information is displayed in columns. In the first column is the number of iterations completed; next is the step-size, $\lambda = \beta^k$, with k shown in parenthesis; next is $\|\nabla f(\eta)\|_{I_k}$ which is the norm of the gradient with respect to those decision variables that are not at their bounds; next is a four (three if there are no upper or lower bounds) letter sequence of `T`'s and `F`'s where a `T` indicates that the corresponding termination test, described above, is satisfied; next is the value of the objective function; and in the last column, an asterix appears if the set of indices corresponding to constrained variables changed from the previous iteration.

Extensible Features

Because **pdmin** is designed to be callable by other optimization programs, it includes three extensions that allow the user to customize its behavior. These extensions are function calls that are made to user supplied subroutines at certain points during each iteration. They allow the user to *(i)* construct the objective function and its gradients, *(ii)* specify termination criteria and perform computations at the end of each **pdmin** iteration, and *(iii)* add additional tests to the step-size selection procedure. The use of the first two of these extensions is demonstrated in the program **aug_lagrng**.

Extension 1. If the global variable `USER_FUNCTION_NAME` is defined in Matlab's workspace and is a string containing the name of a valid m-file, **pdmin** will call that m-file, instead of **simulate**, to evaluate the system functions and gradients. This can be used to construct a composite function from several different calls to **simulate**. For instance, a penalty function can be formed to convert a constrained problem into an unconstrained problem. The syntax for the user function is

```
[f0,x,grad_u,grad_x0] = USER_FUNCTION_NAME(x0,u,t,ialg,action)
```

where the input and output variables are the same as for calls to **simulate**. See a_lagrng_fnc.m for an example.

Extension 2. If the global variable `USER_TEST_NAME` is defined in Matlab's workspace and is a string containing the name of a valid m-file, **pdmin** will call that m-file at the end of each iteration. The syntax for the user function is

```
user_terminate = USER_TEST_NAME(f0,x,u,grad_u,grad_x0,I_i,free_x0)
```

where `I_i` is a column vector indexing all elements of $[u(:);\xi]$ that are not actively constrained by bounds and `free_x0` is the index set of free initial conditions. If the user test returns `user_terminate=1` and the other termination conditions are satisfied, then **pdmin** will terminate. If `user_terminate=2`, then **pdmin** will terminate without regard to the other termination tests. This function can be used solely for the purpose of performing some operations at the end of each iteration by always returning 1. See multiplier_update.m for an example.

Extension 3. If the global variable `ARMIJO_USER_TEST` is defined in Matlab's workspace and is a string containing the name of a valid m-file, the function **armijo**, which is called by **pdmin** to compute the Armijo step-length, will call that m-file in order to guarantee that the step-length satisfies

```
ARMIJO_USER_TEST(j,x,x0,u,t,ialg,I_i,free_x0) <= 0
```

where x and u are evaluated at the current trial step-length and `I_i` and `free_x0` have the same meaning as for Extension 2. This extension can be used, for instance, in a barrier function algorithm to prevent trial step-lengths that are outside the region of definition of the barrier function.

Notes:

The following features are used in the current implementation of **pdmin**.

1. A scaling for the objective function is computed using the objective scaling 2 described for **riots**. The primary purpose of this scaling is to prevent an excessive number of function evaluations during the first line search.

2. The Armijo step-length adjustment mechanism will stop increasing the step-length if $k \leq 0$ and and the next increase in step-length results in an increase in the objective function.

3. A quadratic fit is performed at the end of each step-length calculation when the selected search direction method is `conjgr`. This fit takes one extra system simulation.

4. If **simulate** returns `NaN`, the step-length will be decreased until **simulate** returns a valid result.

5. Because of the coordinate transformation, the inner products in the termination tests are inner-products in $L_2[a, b]$. Thus the tests are independent of the discretization level.

6. Control bounds can be violated if using splines of order $\rho > 2$ if the spline coordinate transformation is in effect. This is only possible with RK4 because splines of order $\rho > 2$ are only allowed for RK4 and LSODA and the transform is turned off for LSODA if bounds are used.

riots

Purpose

This is the main optimization program in RIOTS_95. The algorithm used by **riots** is a sequential quadratic programming (SQP) routine called NPSOL. Multiple objective functions can be handled indirectly using the transcription describe in §2.3.

The user is urged to check the validity of the user-supplied derivatives with the utility program **check_deriv** before attempting to use **riots**.

Calling Syntax

```
[u,x,f,g,lambda2] = riots([x0,{fixed,{x0min,x0max}}],u0,t,Umin,Umax,
                          params,[miter,{var,{fd}}],ialg,
                          {[eps,epsneq,objrep,bigbnd]},{scaling},
                          {disp},{lambda1});
```

Description of Inputs

The first six inputs are described in Table O3. The remainder are described here.

`miter`	The maximum number of iterations allowed.
`var`	Specifies a penalty on the piecewise derivative variation[4, Sec. 4.5] [15] of the control to be added to the objective function. Can only be used with first and second order splines. Adding a penalty on the piecewise derivative variation of the control is useful if rapid oscillations are observed in the numerical solution. This problem often occurs for singular problems [13,14] in which trajectory constraints are active along singular arcs. The penalty should be ten to ten thousand times smaller than the value of the objective function at a solution.
`fd`	If a non-zero value is specified, the gradients for all functions will be computed by finite-difference approximations. In this case **Dh**, **Dg**, and **Dl** will not be called. Default: 0.
`ialg`	Specifies the integration algorithm used by **simulate**.
`eps`	Overall optimization tolerance. For NPSOL, `eps` is squared before calling NPSOL. See the SQP user's manual for more details. Default: 10^{-6}.
`epsneq`	Nonlinear constraint tolerance. Default: 10^{-4}.
`objrep`	Indicates function precision. A value of 0 causes this features to be ignored. Default: 0.
`bigbnd`	A number large than the largest magnitude expected for the decision variables. Default: 10^6.
`scaling`	Allowable values are 00, 01, 10, 11, 12, 21, 22. Default: 00. See description below.
`disp`	Specify zero for minimal displayed output. Default: 1.

[15] The piecewise derivative variation is smoothed to make it differentiable by squaring the terms in the summation.

`lambda1` Only applies to NPSOL. Controls warm starts. Default: 0. See description below.

Description of Outputs

The first two outputs are described in Table O4.

`f` The objective value at the obtained solution.

`g` Vector of constraint violations in the following order (*N.B.* linear constraints are treated as nonlinear constraint for systems with nonlinear dynamics):

Table O5

linear endpoint inequality
linear trajectory inequality
linear endpoint equality
nonlinear endpoint inequality
nonlinear trajectory inequality
nonlinear endpoint equality

`lambda2` Vector of Lagrange multipliers. This output has two columns if NPSOL is used. The first column contains the Lagrange multipliers. The first $m(N + \rho - 1)$ components are the multipliers associated with the simple bounds on u. These are followed by the multipliers associated with the bounds on any free initial conditions. Next are the multipliers associated with the general constraint, given in the same order as the constraint violations in the output `g`. The second column of `lambda2` contains information about the constraints which is used by **riots** if a warm start using `lambda1` is initiated (as described below).

Scaling

There are several heuristic scaling options available in **riots** for use with badly scaled problems. There are two scaling methods for objective functions and two scaling methods for constraints. These are selected by setting `scaling` to one of the two-digit number given in the following table:

Table O6

`scaling`	Objective Scaling Method	Constraint Scaling Method
00	no scaling	no scaling
01	no function scaling	constraint scaling 1
10	function scaling 1	no constraint scaling
11	function scaling 1	constraint scaling 1
12	function scaling 1	constraint scaling 2
21	function scaling 2	constraint scaling 1
22	function scaling 2	constraint scaling 2

In the following, $FACTOR = 20$. Also, $\eta_0 = (u_0, \xi_0)$.

Objective Scaling 1: For each $\nu \in \mathbf{q}_o$, the ν-th objective function is scaled by

$$\gamma_o^\nu = \frac{1}{1 + |f^\nu(\eta_0)|} FACTOR \,.$$

Objective Scaling 2: For each $\nu \in \mathbf{q}_o$, let

$$S = (1 + \|\eta_0\|_\infty) \,/\, (100 \|\nabla f^\nu(\eta_0)\|_\infty)$$

$$\delta\eta = [\eta_0 - S \nabla f^\nu(\eta_0)]_\# \,,$$

$$\gamma = \frac{1}{2} \left| \frac{\langle \delta\eta, \delta\eta \rangle_I}{f^\nu(\eta_0 + \delta\eta_0) - f^\nu(\eta_0) - \langle \nabla f^\nu(\eta_0), \delta\eta_0 \rangle_I} \right| ,$$

where $[\cdot]_\#$ is the projection operator that projects its argument into the region feasible with respect to the simple bounds on u and ξ, and I is the set of indices of η_0 corresponding to components which are in the interior of this feasible region (γ is the distance along the projected steepest descent direction, $\delta\eta$, to the minimum of a quadratic fit to $f(\cdot)$). If $\gamma \geq 10^{-4}$, scale the ν-th objective function by $\gamma_o^\nu = FACTOR\,\gamma$. Otherwise, compute $\gamma = \|\nabla f^\nu(\eta_0)\|$. If $\gamma \geq 10^{-3}$, set $\gamma_o^\nu = FACTOR\,\gamma$. Otherwise, use function scaling 1.

Constraint Scaling 1: For each $\nu \in \mathbf{q}_{ei}$, the endpoint inequality constraints are scaled by

$$\gamma_{ei}^\nu = \frac{1}{\max\{\, 1, |g_{ei}^\nu(\eta_0)| \,\}} FACTOR \,,$$

for each $\nu \in \mathbf{q}_{ee}$, the endpoint equality constraints are scaled by

$$\gamma_{ee}^\nu = \frac{1}{\max\{\, 1, |g_{ee}^\nu(\eta_0)| \,\}} FACTOR \,,$$

and, for each $\nu \in \mathbf{q}_{ti}$, the trajectory inequality constraints are scaled by

$$\gamma_{ti}^\nu = \frac{1}{\max\{\, 1, \max\limits_{k \,\in\, \{1,\ldots,N+1\}} |l_{ti}^\nu(t_k, x_k, u_k)| \,\}} FACTOR \,.$$

Constraint Scaling 2: The trajectory constraint scalings are computed in the same way as for constraint scaling method 1. For each $\nu \in \mathbf{q}_{ei}$, the endpoint inequality constraints are scaled by $\gamma_{ei}^\nu = \gamma$ and, for each $\nu \in \mathbf{q}_{ee}$, the endpoint equality constraints are scaled by $\gamma_{ee}^\nu = \gamma$ where γ is determined as follows. If $|g(\eta_0))| \geq 10^{-3}$, let

$$\gamma = \frac{1}{|g(\eta_0)|} FACTOR \,,$$

otherwise, if $\|\nabla g(\eta_0)\| \geq 10^{-3}$, let

$$\gamma = \frac{1}{\|\nabla g(\eta_0)\|} FACTOR \,,$$

otherwise do not scale.

Scaling will not always reduce the amount of work required to solve a specific problem. In fact, it can be detrimental. In the following table, we show the number of iterations required to solve some problems (described in the appendix) with and without function scaling. All of these problems were solved using second order splines on a uniform mesh with a discretization level of $N = 50$. The problems were solved using `scaling` set to 0, 10, and 20. It should be noted that none of these problems is seriously ill-conditioned.

Table O7

Problem	ialg	0	10	20
LQR	2	5	7	7
Rayleigh w/o endpoint constraint	2	18	17	14
Rayleigh with endpoint constraint	2	24	29	19
Goddard w/o trajectory constraint	4	69	29	45
Goddard with trajectory constraint	4	22	17	19

For the last row, **riots** was called with `var` $= 10^{-4}$. Constraint scaling did not have any affect on the number of iterations for these problems. Discussion of scaling issues can be found in [12,15,16].

Warm Starts

The input `lambda1` controls the warm-starting feature available with **riots** if it is linked with NPSOL. There are two types of warm starts.

The first type of warm start is activated by setting `lambda1=1`. If this warm start is used, the Lagrange multiplier estimates and Hessian estimate from the previous run will automatically be used as the starting estimates for the current run. This is useful if **riots** terminates because the maximum number of iterations has been reached and you wish to continue optimizing from where **riots** left off. This type of warm start can only be used if the previous call to **riots** specified `lambda1=-1` or `lambda1=1`. Setting `lambda1=-1` does not cause a warm-start, it just prepares for a warm start by the next call to **riots**.

The second type of warm start allows warm starting from the previous solution from **riots** but interpolated onto a new mesh and is only implemented for first and second order splines. It is activated by providing estimates of the Lagrange multipliers in the first column of input `lambda1` and the status of the constraints in the second column of `lambda1`. Typically, `lambda1` is produced by the program distribute which appropriately interpolates the `lambda2` output from the previous run of **riots** onto the new mesh. When `lambda1` is supplied in this way, **riots** estimates $H(\eta)$, the Hessian of the Lagrangian at the current solution point, by applying finite-differences to the gradients of all objective and constraint functions weighted by their Lagrange multipliers (and scalings if a scaling option has been specified).

The estimate $H(\eta)$ of the Hessian of the Lagrangian is computed by the program **comp_hess**. This computation requires $N + \rho + n_{\text{free x0}}$ system simulations (where $n_{\text{free x0}}$ is the number of free initial conditions) and twice as many gradient computations as there are objective functions and constraints with non-zero Lagrange multipliers. Also, if a non-zero value for `var` is specified, the second derivative of the penalty term on the piecewise derivative variation of the control is added to the Hessian estimate. When $\rho \leq 2$, the computation takes advantage of the symmetry of the Hessian by stopping the simulations and gradient computations once the calculations start filling the Hessian above its diagonal. After H is computed, it is converted into transformed coordinates using the formula $\widetilde{H} = (\mathbf{M}_\alpha^{-1/2})^T H \mathbf{M}_\alpha^{-1/2}$, unless the transformation mechanism has been disabled.

Because NPSOL expects the Cholesky factorization of a positive definite Hessian estimate, the following additional steps are taken. First, a Cholesky factorization is attempted on $\tilde{H}$. If this fails (because $\tilde{H}$ is not positive definite) the computation continues with the following procedure. A singular value decomposition is performed to obtain the factorization $\tilde{H} = USV^T$, where S is the diagonal matrix of singular values of $\tilde{H}$. Next, each diagonal element, σ_i, of S is set to $\sigma_i = \max\{\sigma_i, \varepsilon_{\text{mach}}^{1/3}\}$. Then, we set $\tilde{H} = USU^T$, which, because $\tilde{H} = \tilde{H}^T$, makes all negative eigenvalues of $\tilde{H}$ positive while preserving the eigenstructure of $\tilde{H}$. Finally, the Cholesky factorization of $\tilde{H}$ is computed.

Notes:

1. Since NPSOL is not a feasible point algorithm, it is likely that intermediate iterates will violate some nonlinear constraints.

2. Because of the coordinate transformation, the inner products in the termination tests correspond to inner-products in $L_2[a, b]$. Thus the tests are independent of the discretization level.

3. Control bounds can be violated if using splines of order $\rho > 2$ if the spline coordinate transformation is in effect. This is only possible with RK4 because splines of order $\rho > 2$ are only allowed for RK4 and LSODA and the transform is turned off for LSODA if bounds are used.

Bugs:

1. **riots** uses the Matlab MEX function **mexCallMATLAB** to make calls to **simulate**. There is a bug in this function that interferes with the operation of `ctrl-C`. This problem can be circumvented by compiling **simulate** directly into **riots** (see instructions on compiling **riots**).

2. The full warm-start feature, which requires the computation of the Hessian using finite-differencing of the gradients, is not allowed if the input `fd` is set to a non-zero value.

7. UTILITY ROUTINES

There are several utility programs, some are used by the optimization programs and some are callable by the user. Those utility programs of interest to the user are described in this section. These are:

control_error Computes an estimate of the norm of the error of the computed solution. If η_N^* is the computed solution and η^* is a local minimizer for problem **OCP**, the solution error is $\|\eta_N^* - \eta^*\|_{H_2}$.

distribute Redistributes the integration mesh according to one of several mesh refinement strategies including one which simply doubles the mesh. The control spline defined on the previous mesh will be interpolated onto the mesh. The order of the spline is allowed to change.

est_errors Returns an estimate of the global integration error for the fixed step-size Runge-Kutta methods and uses the variable step-size integration algorithm to obtain accurate measures of the objective functions, constraint violations and trajectories. It also returns the function space norm the free portion of the gradient of the augmented Lagrangian which is needed by **control_error**.

sp_plot Plots spline functions.

transform Computes a matrix which allows the L_2 inner product of two splines to be computed by taking the inner product of their coefficients.

control_error

Purpose

This function uses values computed by **est_errors** for solutions of **OCP** on different integration meshes to estimate $\|\eta_N - \eta^*\|_{H_2}$ for the current solution $\eta_N = (u_N, \xi_N)$ using results from [4, Sec. 4.4.].

Calling Syntax

```
[error,norm_zd]=control_error(x01,u1,t1,ze1,x02,u2,t2,ze2,{Tf})
```

Description

This program compares the two solutions $\eta_{N_1} = (\texttt{u1}, \texttt{x01})$ and $\eta_{N_2} = (\texttt{u2}, \texttt{x02})$, corresponding to the mesh sequences `t1` and `t2` to produce an estimate of $\|\eta_{N_2} - \eta^*\|_{H_2}$ where $\eta^* = (u^*, \xi^*)$ is a solution for **OCP**. For free final time problems, `Tf` should be set to the duration scale factor (see transcription for free final time problems in §2). Only the first columns of `x01` and `x02` are used. The inputs `ze1` and `ze2` are the norms of the free gradients of the augmented Lagrangians evaluated at η_{N_1} and η_{N_2}, respectively, which can be obtained from calls to **est_errors**.

The output `error` is the estimate of $\|\eta_{N_2} - \eta^*\|_{H_2}$ where

$$\|\eta_{N_2} - \eta^*\|_{H_2}^2 \doteq \|\texttt{x02} - \xi^*\|_2^2 + \int_a^{a+(b-a)\texttt{Tf}} \|u_2(t) - u^*(t)\|_2^2 dt \ ,$$

with $u_2(\cdot)$ the spline determined by the coefficients `u2`. The output `norm_zd` is $\|\eta_{N_2} - \eta_{N_1}\|_{H_2}$ where

$$\|\eta_{N_2} - \eta_{N_1}\|_{H_2}^2 \doteq \|\texttt{x02} - \texttt{x01}\|_2^2 + \int_a^{a+(b-a)\texttt{Tf}} \|u_2(t) - u_1(t)\|_2^2 dt \ ,$$

with $u_1(\cdot)$ and $u_2(\cdot)$ the splines determined by the coefficients `u1` and `u2`, respectively.

Example

Let u_1 be the coefficients of the spline solution for the mesh `t1` and let u_2 be the coefficients of the spline solution for the mesh `t2`. Let λ_1 and λ_2 be the Lagrange multipliers (if the problem has state constraints) and let I_1 and I_2 be the index set of inactive control bounds returned by one of the optimization programs (if the problem has control bounds). The Lagrange multipliers and the inactive control bound index sets are also returned by the optimization routines. Then we can compute the errors, $e_1 = \|\eta_{N_1} - \eta^*\|_{H_2}$ and $e_2 = \|\eta_{N_2} - \eta^*\|_{H_2}$ as follows:

```
>> [int_error1,norm_gLa1] = est_errors(x0,u1,t1,1,ialg1,lambda1,I1);
>> [int_error2,norm_gLa2] = est_errors(x0,u2,t1,1,ialg2,lambda2,I2);
>> error1 = control_error(x0,u2,t2,norm_gLa2,x0,u1,t1,norm_gLa1,1);
>> error2 = control_error(x0,u1,t1,norm_gLa1,x0,u2,t2,norm_gLa2,1);
```

See Also: **est_errors**.

distribute

Purpose

This function executes various strategies for redistributing and refining the current integration mesh. It also interpolates the current control and Lagrange multipliers corresponding to trajectory constraints onto this new mesh.

Calling Syntax

```
[new_t,new_u,new_lambda,sum_lte]=distribute(t,u,x,ialg,lambda,
                                            n_free_x0,strategy,
                                            {FAC},{new_K},{norm})
```

Description of Inputs

`t` Row vector containing the sequence of breakpoints for the current mesh.

`u` The coefficients of the spline defined on the current mesh.

`x` Current state trajectory solution.

`ialg` Integration algorithm to be used during next simulation or optimization.

`lambda` Current Lagrange multiplier estimates from **riots**. Specify `lambda=[]` if you do not need new multipliers for a warm start of **riots**.

`n_free_x0` Number of free initial conditions. This value only affects the extension of Lagrange multipliers needed for a warm start of **riots**.

`strategy` Selects the redistribution strategy according to the following table:

`strategy`	Type of Redistribution
1	Movable knots, absolute local truncation error.
2	Fixed knots absolute local truncation error.
3	Double the mesh by halving each interval.
4	Just change spline order to `new_K`.
11	Movable knots, relative local truncation error.
12	Fixed knots, relative local truncation error.

For more information on these strategies, see Chapter 4.3.2 in 4. The quasi-uniformity constant in equations (4.3.13) and (4.3.24) of that reference is set to $\bar{\delta} = 50$. In *Step 2* of Strategy 2 (and 12), $\sigma = 1/4$.

`FAC` For use with strategies 1,2,11 and 12. If specified, the number of intervals in the new mesh is chosen to achieve an integration accuracy approximately equal to the current integration accuracy divided by `FAC`. If `FAC=[]` or `FAC=0`, the number of intervals in the new mesh will be the same as the previous mesh for strategies 1 and 11. For strategies 2 and 12, the relative errors $\bar{e}_k$ will be used without being pre-weighted by `FAC`.

`new_K` Specifies the order of the output spline with coefficients `new_u`. By default, `new_K` is the same as the order of the input spline with coefficients `u`.

`norm` Specifies the norm used to measure the integration error on each interval. If `norm=0`, then

$$e_k = \|\mathrm{lte}_k\|_2 \ , \ \ k = 1, \ldots, N \ .$$

If `norm=1`, then

$$e_k = \|\mathrm{lte}_k\|_\infty \ , \ \ k = 1, \ldots, N \ .$$

The quantity lte_k is an estimate of the local truncation error produced by the k-th integration (see description of **simulate**, form 7). Default: 0.

Description of Outputs

`new_t` Contains the sequence of breakpoints for the new mesh.

`new_u` Contains the coefficients of the spline of order `new_K` (if specified) interpolated from `u` onto the new mesh.

`new_lambda` Two column matrix of Lagrange multiplier estimates and associate constraint status indicators. Those multipliers (and indicators) corresponding to control bounds and trajectory constraints are extended to the new mesh. This is for use with the warm start facility of **riots**.

`sum_lte` An $(n+1)$-column vector of the accumulated local truncation errors produced by the integration:

$$\texttt{sum\_lte(i)} = \sum_{k=1}^{N} e_k^i \ , \ \ i = 1, .., n+1 \ ,$$

where e_k^i is as computed above. The $(n+1)$-th component represents the accumulation of local truncation errors for the integrand of the first objective function.

Notes:

1. The algorithm used in strategies 1 and 2 does not take into account the presence, if any, of trajectory constraints. Strategies 2 and 12 include a mechanism that tends to add mesh points at times, or near times, where trajectory constraints are active. The input `lambda` must be supplied for this mechanism to be used.

est_errors

Purpose

This function performs a high accuracy integration with LSODA to produce estimates of various quantities. One of these quantities is used by **control_error** to produce an estimate of $\|\eta_N - \eta^*\|_{H_2}$.

Calling Syntax

```
[int_error,norm_gLa,J,G,x,Ii] = est_errors([x0,{fixed}],u,t,Tf,
                                            ialg,lambda,{I_i})
```

Description of Inputs

`x0`	Initial conditions of the *current solution*. When one or more initial conditions are free variables, set `x0=x(:,1)` where `x` is the trajectory solution returned by one of the optimization programs.
`fixed`	An n-vector that indicates which components of `x0` are free variables. If `fixed(i)=0` then `x0(i)` is a free variable. Default: all ones.
`u`	Current control solution.
`t`	Sequence of breakpoints for the current integration mesh on the (nominal) time interval $[a, b]$.
`Tf`	The duration scale factor. For fixed final time problems, set `Tf=1`.
`ialg`	Integration algorithm used to produce the current solution.
`lambda`	Vector of Lagrange multiplier estimates (one or two columns depending on which optimization program produced `lambda`).
`I_i`	Index set of controls and free initial conditions that are not at their bounds (returned by one of the optimization program).

Description of Outputs

`int_error` — `int_error(i)`, $i = 1, \ldots, n+1$, is an estimate of the global integration error, $|x^i_{N,N+1} - x^i(b)|$, of the current solution computed by summing the local truncation errors produced by the integration method specified by `ialg`. The local truncation errors are obtained by a call to **simulate** (form 7). If the discrete solver or the variable stepsize integration routine is being used, `int_error` is set to a vectors of zeros. If this is the only output requested, the rest of the calculations are skipped.

`norm_gLa` — This is an estimate of the H_2 norm of the free gradient of the augmented Lagrangian $L_{c,\lambda}$ evaluated at the current solution $\eta = (u, \xi)$. The H_2 norm of the free gradient of the augmented Lagrangian is the norm restricted to the subspace of controls and initial conditions that are not constrained by their bounds. Let `grad_Lu` be the gradient of the augmented Lagrangian with respect to controls, `grad_Lx0` be the gradient of the augmented

Lagrangian with respect to initial conditions and $\mathbf{M}_\alpha$ be the spline transformation matrix computed by **transform**. If `Ii` is the index set estimating the free portion of η = `[u(:);xi(free_x0)]` (see below), then the free norm if computed as follows:

$$\|\nabla_{\text{free}} L_{c,\lambda}(\eta)\|_{H_2} = \texttt{gLM(Ii)'*gL(Ii)}\ ,$$

where

$$\texttt{gLM} = [\texttt{grad\_Lu(:)}\,\mathbf{M}_\alpha^{-1}\,;\,\texttt{grad\_Lx0(free\_x0)}]$$

and

$$\texttt{gL} = [\texttt{grad\_Lu(:)}\ ;\ \texttt{grad\_Lx0(free\_x0)}]\ .$$

In forming the augmented Lagrangian, λ = `lambda(:,1)` and $c_i = |\lambda_i|$. The quantity $\|\nabla_{\text{free}} L_{c,\lambda}(\eta)\|_{H_2}$ is used by **control_error** to estimate the error $\|\eta_N - \eta^*\|_{H_2}$.

`J` An estimate of the objective function at the current solution. This estimate is produced using LSODA.

`G` An estimate of the sum of constraint violations. This estimate is produced using LSODA.

`x` The solution trajectory as produced using LSODA.

`Ii` Set of indices that specify those time points in the mesh `t` that are contained in the estimate I of subintervals in $[a,b]$ on which the control solution is not constrained by a control bound followed by the indices of any free initial conditions that are not constrained by a bound. This index set is used by **control_error**. For the purpose of demonstration, consider a single input systems ($m = 1$) with no free initial conditions. Let

$$\hat{I} \doteq \bigcup_{k\,\in\,\texttt{I\_i}} [t_{k-1}, t_{k+1}]\ ,$$

where $t_0 \doteq t_1$ and $t_{N+2} \doteq t_{N+1}$. $\hat{I}$ is an estimate of the time intervals on which the control bounds are inactive. From $\hat{I}$, the index set `Ii` is set to

$$\texttt{Ii} \doteq \{\, k \mid t_k \in \hat{I}\,\}\ .$$

When there are multiple inputs, this procedure is repeated for each input. When there are free initial conditions, the indices of the unconstrained components of `x0(free_x0)` are added to the end of `Ii`.

Notes:

1. If the user does not supply the derivative functions **Dh** and **Dl** then it will be necessary to change the statement `IALG=5` to `IALG=6` in the file est_errors.m.

See Also: **control_error**.

sp_plot

Purpose

This program allows the user to easily plot controls which are represented as splines.

Calling Syntax

```
val = sp_plot(t,u,{tau})
```

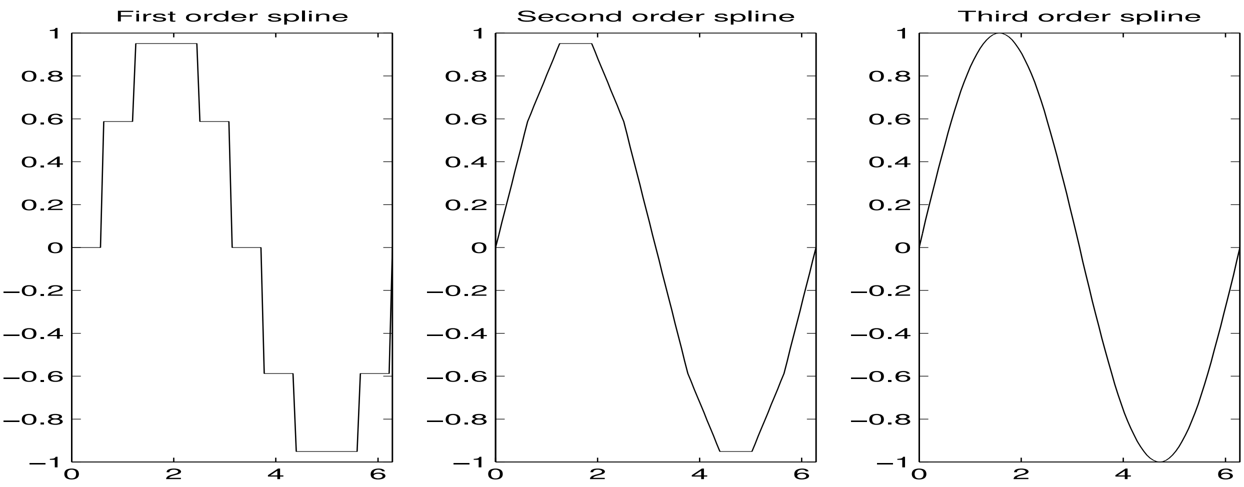
Description

Produces a plot of the spline with coefficients u defined on the knot sequence constructed from the integration mesh t. The order, ρ, of the spline is presumed equal to length(u) $- N + 1$. If tau is specified, u is not plotted, just evaluated at the times tau. Otherwise, u is plotted at 100 points with the same relative spacing as the breakpoints in t. Second order splines can also be plotted using the Matlab command **plot** instead of **sp_plot**.

If the input tau is not given, then the output is val=[t;uval] where t are the data points and uval are the data values; uval has the same number of rows as the input u. If the input tau is given, then the output is just val=uval.

Example. This example plots a first, second and third order spline approximation to one period of a sinusoid using ten data points. The splines are produced using the commands in the Spline Toolbox.

```
>> t=[0:2*pi/10:2*pi];
>> sp1 = spapi(t,t(1:10),sin(t(1:10)));
>> [dummy,u1] = spbrk(sp1);
>> knots2 = augknt(t,2); knots3 = augknt(t,3);
>> sp2 = spapi(knots2,t,sin(t));
>> [dummy,u2] = spbrk(sp2);
>> tau = aveknt(knots3,3);
>> sp3 = spapi(knots3,tau,sin(tau));
>> [dummy,u3] = spbrk(sp3);
>> sp_plot(t,u1); sp_plot(t,u2); sp_plot(t,u3);
```



transform

Purpose

This function produces the transformation matrix $\mathbf{M}_\alpha$. It is called by **riots** and **pdmin** to generate the spline coordinate transformation for the controls.

Calling Syntax

```
Malpha = transform(t,order)
```

Description

Given two splines u_1 and u_2 of order ρ = `order` with coefficient α_1 and α_2 defined on the knot sequence with breakpoints given by `t`, $\langle u_1, u_2 \rangle_{L_2} = trace(\alpha_1 \mathbf{M}_\alpha \alpha_1^T)$. This function works with non-uniform meshes and with repeated interior knot points.

The output, `Malpha` is given in sparse matrix format. The transform matrix for $\rho = 1, 2, 3,$ or 4 has been pre-computed for uniformly spaced mesh points. Also, if the inputs to the preceding call to **transform**, if there was a preceding call, were the same as the values of the current inputs, then the previously computed transform matrix is returned.

Example

This example generates two second order splines and computes their L_2 inner-product by integrating their product with the trapezoidal rule on a very fine mesh and by using $\mathbf{M}_\alpha$.

```
>> t = [0:.1:1];
>> knots = augknt(t,2);
>> coef1 = rand(1,11); coef2 = rand(1,11);
>> sp1 = spmak(knots,coef1);
>> sp2 = spmak(knots,coef2);
>> tau = [0:.0001:1];
>> u1 = fnval(sp1,tau);
>> u2 = fnval(sp2,tau);
>> inner_prod1 = trapz(tau,u1.*u2)

inner_prod1 = 0.2800

>> Malpha = transform(t,2);
>> inner_prod2 = coef1*Malpha*coef2'

inner_prod2 = 0.2800

>> inner_prod1-inner_prod2

ans = 1.9307e-09
```

8. INSTALLING, COMPILING AND LINKING RIOTS Most of the programs supplied with RIOTS_95 are pre-compiled and ready to run as-is. By default, RIOTS_95 is configured to run user problems supplied as 'sys_*.m' m-files. The m-file form is described in Section 4 of this manual. If the user wishes to run RIOTS_95 in this manner, no compilation and/or linking is required. However, a significant increase in performance is possible if the user supplies his problem descprition in C code. In this case, the user must compile his C code and link the resulting object code with the simulation program. This is a fairly straightforward endeavor and is explained below.

Note: If you have the RIOTS_95 demo package but have not yet purchased RIOTS_95, you will not be able to solve your own optimal control problems. Please refer to "license.doc" supplied with the demonstration for further details on the RIOTS_95 purchase agreement.

Compiling User-Supplied System Code

What you need:

1. Windows 3.x/95/NT
2. A RIOTS_95 distribution package available from the RIOTS homepage
   http://robotics.eecs.berkeley.edu/~adams/riots.html
   http://www.cadcam.nus.sg/~elecyq/riots.html
   or send email to Adam Schwartz (adams@eecs.berkeley.edu) or
   Yangquan Chen (yangquan@ee.nus.sg).
3. Watcom C/C++ compiler[16] version 10 or up
   (http://www.powersoft.com/products/languages/watccpl.html).
4. Matlab 4.2c1 or Matlab 4.0
5. Spline Toolbox versions 1.1a, 1993.11.25.

Important: If you want to use a math function such as *sin()* in your optimal control problem, you must include the pre-compiler directive

```
#include <math.h>
```

in your code.

It is recommended that you make a copy of the "simulate.mex" that comes supplied with RIOTS_95 before creating your own "simulate.mex" with the steps outlined here. Then, if you want to use the m-file interface for some reason you can copy back the original version of "simulate.mex".

Step 1: Write the user-supplied C routines (refer to §4 for details) required for you optimal control problem. Several sample problems are supplied with RIOTS_95 in the "systems" directory. Additionally, there is a file called "template.c" which you can use as a starting point for writing your own problem.

[16] If you are using Matlab v. 4.0, only version 9.0 or up of the Watcom C compiler is required.

Step 2: In the following, assume you have created a C code problem, located in your RIOTS_95/systems directory, called "my_problem.c". Before executing these commands, save the version of "simulate.mex" that comes distributed with RIOTS_95 to another file, say, "m_sim.mex". Then, if you want to use the m-file interface later (in which case you can move "m_sim.mex" back to "simulate.mex"). Open a DOS box in Windows and execute the following sequence of commands:

- `'command /e:4096'` (to increase the size of the environment space.)
- `'cd \riots_95'`
- change relevant disk/directory setings in "compile.bat" and "cmex.bat" with a file editor.
- `'cd systems'`
- 'compile my_problem.c'
  'linksimu my_problem.o'

These sequence of commands will generate a file called "simulate.mex" which is used by RIOTS_95 to solve your problem.

Step 3: To use RIOTS_95 to solve your optimal control problem,

- Run Matlab and at the Matlab prompt, type:
  ```
  >> path(path,'\riots_95')
  >> cd systems
  ```

Now you are ready to use RIOTS_95 to solve your problem.

The M-file interface.

As mentioned above, RIOTS_95 comes distributed to run user m-file programs. This allows users that do not have the Watcom C compiler to use RIOTS_95. The m-file interface for RIOTS_95 can be produced with the Watcom C compiler with the following steps executed in a DOS box:

- Compile "msyslink.c"
- Run "linksimu.bat"

With the m-file interface, the user only needs to provide "sys_*.m" m-files, but the solution time is much longer than with C code.

9. PLANNED FUTURE IMPROVEMENTS

This version of RIOTS was developed over a period of two years. Many desirable features that could have been included were omitted because of time constraints. Moreover, there are many extensions and improvements that we have envisioned for future versions. We provide here a synopsis of some of the improvements currently being planned for hopefully, upcoming versions of RIOTS.

- **Automatic Differentiation of user-supplied functions.** This would provide automatic generation of the derivative functions **Dh**, **Dl** and **Dl** using techniques of automatic differentiation [17,18].
- **Extension to Large-Scale Problems.** The size of the mathematical programming problem created by discretizing an optimal control problem (the way it is done in RIOTS) depends primarily on the discretization level N. The work done by the projected descent algorithm, **pdmin**, grows only linearly with N and hence **pdmin** (and **aug_lagrng**) can solve very large problems. However, these programs cannot handle trajectory constraints or endpoint equality constraints[17] The main program in, **riots**, is based on dense sequential quadratic programming (SQP). Hence, **riots** is not well-suited for high discretization levels. There are many alternate strategies for extending SQP algorithms to large-scale problems as discussed in [4, Chap. 6]. The best approach is not known at this time and a great deal of work, such as the work in [19-22] as well as our on investigations, is being done in this area.
- **Trajectory constraints.** Our current method of computing functions gradients with respect to the control is based on adjoint equations. There is one adjoint equation for each function. This is quite inefficient when there are trajectory constraints because for each trajectory constraint there is, in effect, one constraint function per mesh point. Thus, for an integration mesh with $N+1$ breakpoints, roughly N adjoint equations have to be solved to compute the gradients at each point of a trajectory constraint. An alternate strategy based on the state-transition (sensitivity) matrix may prove to be much more efficient. Also, it is really only necessary to compute gradients at points, t_k, where the trajectory constraints are active or near-active. The other mesh points should be ignored. Algorithms for selecting the active or almost active constraint are present in [23,24] along with convergence proofs.
- **Stabilization of Iterates.** One of the main limitations of the current implementation of RIOTS is that it is not well-equipped to deal with problems whose dynamics are highly unstable. For such problems, the iterates produced by the optimization routines in RIOTS can easily move into regions where the system dynamics "blow-up" if the initial control guess is not close to a solution. For instance, a very difficult optimal control problem is the Apollo re-entry problem [25]. This problem involves finding the optimum re-entry trajectory for the Apollo space capsule as it enters the Earth's atmosphere. Because of the physics of this problem, slight deviations of the capsules trajectory can cause the capsule to skip off the Earth's atmosphere or to burn up in the atmosphere. Either way, once an iterate is a control that drives the system into such a region of the state-space, there is no way for the optimization routine to recover. Moreover, in this situation, there is no way to avoid these regions of the state-space using control constraints.

  This problem could be avoided using constraints on the system trajectories. However, this is a very expensive approach for our method (not for collocation-based methods), especially at high discretization levels. Also, for optimization methods that are not feasible point algorithms, this approach still might not work. An intermediate solution is possible because it is really only necessary to check the trajectory constraints at a few points, called nodes, in the integration mesh. This can be accomplished as follows. Let t_k be one such node. Then define the decision variable $\tilde{x}_{k,0}$ which will be

[17] Endpoint inequality constraints can be handled effectively with **aug_lagrng** by incorporating a suitable active constraint set strategy.

taken as the initial condition for integrating the differential equations starting at time t_k. This $\tilde{x}_{k,0}$ is allowed to be different than the value $\bar{x}_k$ of the state integrated up to time t_k. However, to ensure that these values do, in fact, coincide at a solution, a constraint of the form $g_k(u) \doteq \tilde{x}_{k,0} - \bar{x}_k = 0$ must be added at each node. Note that, for nonlinear systems, $g_k(u)$ is a nonlinear constraint. The addition of these node variables allows bounds on that states to be applied at each node point. This procedure is closely related to the multiple shooting method for solving boundary value problems and is an intermediate approach between using a pure control variable parameterization and a control/state parameterization (as in collocation methods). See [26] for a discussion of node placement for multiple shooting methods.

Other Issues and Extensions. Some other useful features for RIOTS would include:

- A graphical user interface. This would allow much easier access to the optimization programs and selection of options. Also, important information about the progress of the optimization such as error messages and warnings, condition estimates, step-sizes, constraint violations and optimality conditions could be displayed in a much more accessible manner.

- Dynamic linking. Currently, the user of RIOTS must re-link **simulate** for each new optimal control problem. It would be very convenient to be able to dynamically link in the object code for the optimal control problem directly from Matlab (without having to re-link **simulate**). There are dynamic linkers available but they do not work with Matlab's MEX facility.

- For problems with dynamics that are difficult to integrate, the main source of error in the solution to the approximating problems is due to the integration error. In this case, it would be useful to use an integration mesh that is finer than the control mesh. Thus, several integration steps would be taken between control breakpoints. By doing this, the error from the integration is reduced without increasing the size (the number of decision variables) of the approximating problem.

- The variable transformation needed to allow the use of a standard inner product on the coefficient space for the approximating problems adds extra computation to each function and gradient evaluation. Also, if the transformation is not diagonal, simple bound constraints on the controls are converted into general linear constraints. Both of these deficits can be removed for optimization methods that use Hessian information to obtain search directions. If the Hessian is computed analytically, then the transformation is not needed at all. If the Hessian is estimated using a quasi-Newton update, it may be sufficient to use the transformation matrix $\mathbf{M}_N$ or $\mathbf{M}_\alpha$ as the initial Hessian estimate (rather than the identity matrix) and dispense with the variable transformation. We have not performed this experiment; it may not work because the the updates will be constructed from gradients computed in non-transformed coordinates[18].

- It may be useful to allow the user to specify bounds on the control derivatives. This would be a simple matter for piecewise linear control representations.

- Currently the only way to specify general constraints on the controls is using mixed state-control trajectory constraints. This is quite inefficient since adjoint variables are computed but not needed for pure control constraints.

- Currently there is no mechanism in RIOTS for to directly handle systems with time-delays or, more generally, integro-differential equations [29]. This would be a non-trivial extension.

[18] With appropriate choice of H_0, quasi-Newton methods are invariant with respect to objective function scalings[27,28], but not coordinate transformations (which is variable scaling).

- Add support for other nonlinear programming routines in **riots**.
- There have been very few attempts to make quantitative comparisons between different algorithms for solving optimal control problems. The few reports comparing algorithms [30,31], involve a small number of example problems, are inconclusive and are out of date. Therefore, it would be of great use to have an extensive comparison of some of the current implementations of algorithms for solving optimal control problems.
- Make it easy for the user to smoothly interpolate from data tables.

APPENDIX

This appendix describes several optimal control problem examples that are supplied with RIOTS_95 in the 'systems' directory. Control bounds can be included on the command line at run-time. See the file 'systems/README' for a description of the code for these problems.

Problem: LQR [10].

$$\min_u J(u) \doteq \int_0^1 0.625x^2 + 0.5xu + 0.5u^2\, dt$$

subject to:

$$\dot{x} = \tfrac{1}{2}x + u \quad ; \quad x(0) = 1 \ .$$

This problem has an analytic solution given by

$$u^*(t) = -(\tanh(1-t) + 0.5)\cosh(1-t)/\cosh(1)\ , \quad t \in [0,1]\ ,$$

with optimal cost $J^* = e^2 \sinh(2)/(1+e^2)^2 \approx 0.380797$.

Problem: Bang [13, p. 112].

$$\min_{u,T} J(u,T) \doteq T$$

subject to:

$$\dot{x}_1 = x_2 \quad ; \quad x_1(0) = 0 \ , \quad x_1(T) = 300$$

$$\dot{x}_2 = u \quad ; \quad x_2(0) = 0 \ , \quad x_2(T) = 0 \ ,$$

and

$$-2 \le u(t) \le 1 \ , \ \forall\, t \in [0,T]\ .$$

This problem has an analytic solution which is given by $T^* = 30$ and

	$0 \le t < 20$	$20 \le t \le 30$
$u^*(t)$	1	-2
$x_1^*(t)$	$t^2/2$	$-t^2 + 60t - 600$
$x_2^*(t)$	t	$60 - 2t$

Problem: Switch [13(pp. 120-123),32].

$$\min_u J(u) \doteq \int_0^1 \tfrac{1}{2}\, u^2 \, dt$$

subject to:

$$\dot{x} = v \quad ; \quad x(0) = 0 \ , \ \ x(1) = 0$$

$$\dot{v} = u \quad ; \quad v(0) = 1 \ , \ \ v(1) = -1$$

$$x(t) - L \le 0 \ , \ \ \forall\, t \in [0, 1] \ ,$$

with $L = 1/9$. This problem has an analytic solution. For any L such that $0 < L \le 1/6$, the solution is $J^* = \frac{4}{9L}$ with

	$0 \le t < 3L$	$3L \le t < 1 - 3L$	$1 - 3L \le t \le 1$
$u^*(t)$	$-\frac{2}{3L}\,(1 - \frac{t}{3L})$	0	$-\frac{2}{3L}\,(1 - \frac{1-t}{3L})$
$v^*(t)$	$(1 - \frac{t}{3L})^2$	0	$(1 - \frac{1-t}{3L})^2$
$x^*(t)$	$L(1 - (1 - \frac{t}{3L})^3)$	L	$L(1 - (1 - \frac{1-t}{3L})^3)$

Problem: Rayleigh [33,34].

$$\min_u J(u) \doteq \int_0^{2.5} x_1^2 + u^2 \, dt$$

subject to:

$$\begin{aligned} \dot{x}_1(t) &= x_2(t) & x_1(0) &= -5 \\ \dot{x}_2(t) &= -x_1(t) + [1.4 - 0.14 x_2^2(t)]x_2(t) + 4u(t) & x_2(0) &= -5 \end{aligned}$$

A constrained version of this problem is formed by including the state constraint

$$x_1(2.5) = 0 \ .$$

Problem: VanDerPol [33].

$$\min_u J(u) \doteq \tfrac{1}{2} \int_0^5 x_1^2 + x_2^2 + u^2 \, dt$$

subject to:

$$\begin{aligned} \dot{x}_1(t) &= x_2(t) & x_1(0) &= 1 \\ \dot{x}_2(t) &= -x_1(t) + (1 - x_2^2)x_2(t) + u(t) & x_2(0) &= 0 \end{aligned}$$

$$-x_1(5) + x_2(5) - 1 = 0 \ .$$

Problem: Parabola [35].

$$\min_u J(u) \doteq \int_0^1 x_1^2 + x_2^2 + 0.005u^2 \, dt$$

subject to:

$$\dot{x}_1 = x_2 \quad ; \quad x_1(0) = 0$$

$$\dot{x}_2 = -x_2 + u \quad ; \quad x_2(0) = -1$$

and

$$x_{2(t)} - 8(t - 0.5)^2 + 0.5 \le 0 \ , \ \forall\, t \in [0, T] \ .$$

Problem: Obstacle [36].

$$\min_u J(u) \doteq 5x_1(2.9)^2 + x_2(2.9)^2$$

subject to:

$$\begin{aligned} \dot{x}_1 &= x_2 & x_1(0) &= 1 \\ \dot{x}_2 &= u - 0.1(1 + 2x_1^2)x_2 & x_2(0) &= 1 \end{aligned}$$

$$-1 \le u(t) \le 1 \ , \ \forall\, t \in [0, 2.9]$$

$$1 - 9(x_1(t) - 1)^2 - \left(\frac{x_2(t) - 0.4}{0.3}\right)^2 \le 0 \ , \ \forall\, t \in [0, 2.9]$$

$$-0.8 - x_2(t) \le 0 \ , \ \forall\, t \in [0, 2.9] \ .$$

Problem: Goddard Rocket, Maximum Ascent [37].

$$\max_{u,T} J(u,T) \doteq h(T)$$

subject to:

$$\begin{array}{lll}
\dot{v} = \dfrac{1}{m}(u - D(h,v)) - \dfrac{1}{h^2}\ , \quad D(h,v) = \frac{1}{2} C_D A \rho_0 v^2 e^{\beta(1-h)} & & v(0) = 0 \\
\dot{h} = v & & h(0) = 1 \\
\dot{m} = -\dfrac{1}{c}\, u & & m(0) = 1 \quad ; \quad m(T) = 0.6
\end{array}$$

$$0 \le u(t) \le 3.5\ , \ \ \forall\, t \in [0,T]\ .$$

where $\beta = 500$, $C_D = 0.05$ and $A\rho_0 = 12,400$. The variables used above have the following meanings:

v	vertical velocity
h	radial altitude above earth ($h = 1$ is earth's surface)
m	mass of vehicle
u	thrust
c	specific impulse (impulse per unit mass of fuel burned, $c = 0.5$)
ρ	air density ($\rho = \rho_0 e^{\beta(1-h)}$)
q	dynamic pressure ($q = \frac{1}{2}\rho v^2$)
D	drag

The endpoint constraint $m(T) = 0.6$ means that there is no more fuel left in the rocket. Another version of this problem includes the trajectory constraint

$$Aq(t) \le 10\ , \ \ \forall\, t \in [0,T]\ .$$

This is a upper bound on the dynamic pressure experienced by the rocket during ascent.

REFERENCES

1. A. Schwartz and E. Polak, “Consistent approximations for optimal control problems based on Runge-Kutta integration,” *SIAM J. Control Optim.* **34**(4)(1996).
2. A. Schwartz and E. Polak, “Runge-Kutta discretization of optimal control problems,” in *Proceedings of the 10th IFAC Workshop on Control Applications of Optimization*, (1996).
3. A. Schwartz and E. Polak, “A family of projected descent methods for optimization problems with simple bounds,” *J. Optim. Theory and Appl.* **91**(1)(1997).
4. A. Schwartz, “Theory and Implementation of Numerical Methods Based on Runge-Kutta Integration for Solving Optimal Control Problems,” Ph.D. Dissertation, Dept. of Electrical Engineering, University of California, Berkeley (1996). Avaiable from `http://robotics.eecs.berkeley.edu/~adams`
5. E. Polak, “On the use of consistent approximations in the solution of semi-infinite optimization and optimal control problems,” *Math. Prog.* **62** pp. 385-415 (1993).
6. Carl de Boor, *A Practical Guide to Splines,* Springer-Verlag, New York (1978).
7. J. D. Lambert, *Numerical Methods for Ordinary Differential Systems,* John Wiley and Sons, England (1991).
8. K. Radhakrishnan and A. C. Hindmarsh, “Description and use of LSODE, the Livermore Solver for Ordinary Differential Equations,” NASA Reference Publ. 1327 (1993).
9. L. R. Petzold, “Automatic selection of methods for solving stiff and nonstiff systems of differential equations,” *SIAM J. Sci. Stat. Comput.* **4** pp. 136-148 (1983).
10. W.W. Hager, “Rates of convergence for discrete approximations to unconstrained control problems,” *SIAM J. Numer. Anal.* **13**(4) pp. 449-472 (1976).
11. L. S. Jennings, M. E. Fisher, K. L. Teo, and C. J. Goh, “MISER3: Solving optimal control problems---an update,” *Advances in Engineering software* **14**(13) pp. 190-196 (1991).
12. P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization,* Academic Press, London (1981).
13. A. E. Bryson and Y. Ho, *Applied Optimal Control,* Hemisphere Publishing Corp. (1975). (revised printing)
14. D. J. Bell and D. H. Jacobson, *Singular Optimal Control Problems,* Academic Press, London (1975).
15. L. T. Biegler and J. E. Cuthrell, “Improved infeasible path optimization for sequential modular simulators--II: the optimization algorithm,” *Computers & Chemical Engineering* **9**(3) pp. 257-267 (1985).
16. O. Stryk, “Numerische Losung optimaler Steuerungsprobleme: Diskretisierung, Parameteroptimierung und erechnung der adjungierten Variablen,” Diploma-Math., Munchen University of Technology, VDI Verlag, Germany (1995).
17. A. Griewank, D. Juedes, and J. Utke, *ADOL-C: A package for the automatic differentiation of algorithms written in C/C++,* Argonne National Laboratory, ftp://info.mcs.anl.gov/pub/ADOLC (December 1993).
18. A. Griewank, “On automatic differentiation,” Preprint MCS-P10-1088, Argonne National

Laboratory, ftp://info.mcs.anl.gov/pub/tech_reports/reports (October 1988).

19. J. T. Betts and P. D. Frank, “A sparse nonlinear optimization algorithm,” *J. Optim. Theory and Appl.* **82**(3) pp. 519-541 (1994).

20. J. T. Betts and W. P. Huffman, “Path-constrained trajectory optimization using sparse sequential quadratic programming,” *J. Guidance, Control, and Dynamics* **16**(1) pp. 59-68 (1993).

21. Henrik Jonson, “Newton Method for Solving Non-linear Optimal Control Problems with Genereal constraints,” Ph.D. Dissertation, Linkoping Studies in Science and Technology (1983).

22. J. C. Dunn and D. P. Bertsekas, “Efficient dynamic programming implementations of Newton’s method for unconstrained optimal control problems,” *J. Optim. Theory and Appl.* **63**(1) pp. 23-38 (1989).

23. J. E. Higgins and E. Polak, “An ε-active barrier-function method for solving minimax problems,” *Appl. Math. Optim.* **23** pp. 275-297 (1991).

24. J. L. Zhou and A. L. Tits, “An SQP algorithm for finely discretized continuous minimax problems and other minimax problems with many objective functions,” *to appear in SIAM J. Optimization*, ().

25. O. Stryk and R. Bulirsch, “Direct and indirect methods for trajectory optimization,” *Annals of Operations Research* **37** pp. 357-373 (1992).

26. U. Ascher, R. Mattheij, and R. Russell, *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations,* Prentice Hall, Englewood Cliffs, NJ (1988).

27. D. F. Shanno and K. H. Phua, “Matrix conditioning and nonlinear optimization,” *Math. Prog.* **14** pp. 149-160 (1978).

28. S. S. Oren, “Perspectives on self-scaling variable metric algorithms,” *J. Optim. Theory and Appl.* **37**(2) pp. 137-147 (1982).

29. F.H. Mathis and G.W. Reddien, “Difference approximations to control problems with functional arguments,” *SIAM J. Control and Optim.* **16**(3) pp. 436-449 (1978).

30. D. I. Jones and J. W. Finch, “Comparison of optimization algortihms,” *Int. J. Control* **40** pp. 747-761 (1984).

31. S. Strand and J. G. Balchen, “A Comparison of Constrained Optimal Control Algorithms,” pp. 439-447 in *IFAC 11th Triennial World Congress*, , Estonia, USSR (1990).

32. O. Stryk, “Numerical solution of optimal control problems by direct collocation,” *International Series of Numerical Methematics* **111** pp. 129-143 (1993).

33. N. B. Nedeljković, “New algorithms for unconstrained nonlinear optimal control problems,” *IEEE Trans. Autom. Cntrl.* **26**(4) pp. 868-884 (1981).

34. D. Talwar and R. Sivan, “An Efficient Numerical Algorithm for the Solution of a Class of Optimal Control Problems,” *IEEE Trans. Autom. Cntrl.* **34**(12) pp. 1308-1311 (1989).

35. D. H. Jacobson and M. M. Lele, “A transformation technique for optimal control problems with a state variable inequality constraint,” *IEEE Trans. Optim. Cntrl.* **14**(5) pp. 457-564 (1969).

36. V. H. Quintana and E. J. Davison, “Clipping-off gradient algorithms to compute optimal controls with constrained magnitude,” *Int. J. Control* **20**(2) pp. 243-255 (1974).

37. H. Seywald and E. M. Cliff, “Goddard Problem in Presence of a Dynamic Pressure Limit,” *J. Guidance, Control and Dynamics* **16**(4) pp. 776-781 (1993).