OPTCON: AN ALGORITHM FOR SOLVING UNCONSTRAINED CONTROL PROBLEMS

By

SHUO LI

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2006

To my wonderful parents, Zhanwu Li and Peirong Zheng.

ACKNOWLEDGMENTS

I express my sincere gratitude to Dr. William W. Hager for his trust, encouragement, guidance, and support (without which this work could not have been completed). I would also like to thank Dr. Shari Moskow and Dr. Jayadeep Gopalakrishnan for agreeing to serve on my committee. I thank the Department of Mathematics for their financial support.

Special thanks go to my parents, who have been working very hard in their careers to enable me to study overseas. They have always given their loving support for my studies. I would like to thank my lovely husband who provided endless love and support while I was writing this thesis. I would like to thank my friends (Hongchao, Sukanya, and Beyza) for their encouragement and support.

TABLE OF CONTENTS

APPENDIX

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

OPTCON: AN ALGORITHM FOR SOLVING UNCONSTRAINED CONTROL PROBLEMS

By

Shuo Li

August 2006

Chair: William W. Hager
Major Department: Mathematics

William Hager and Hongchao Zhang developed a new optimization algorithm (CG_DESCENT). Our study shows how CG_DESCENT can be used to solve unconstrained optimal control problems. The resulting algorithm is called OPTCON.

This numerical work is meaningful, since optimal control applications appear in so many fields (such as aerospace, electronic circuits, heat conduction, and energy optimization). These problems are usually complicated, with a large number of variables, parameters, and initial values. Without a good numerical method, we barely can solve them by hand. Thus we need a fast, stable, and highly accurate method that requires less memory. We tested OPTCON and compared it with another conjugate gradient method.

CHAPTER 1
INTRODUCTION

Based on a study by Hager and Zhang [1], a new conjugate gradient method CG_DESCENT was produced. This method can obtain even higher convergence speed than the ordinary conjugate gradient method and has relatively low memory requirement during the computation. My study used the new conjugate gradient method to obtain a new method-OPTCON for solving nonlinear optimal control problems.

We first define several terms that are used throughout this thesis. Consider an unconstrained optimal control problem in Equation 1-1.

$$\text{minimize } J = \phi(x(t_f)) \tag{1-1}$$

$$\text{subject to } \dot{x}(t) = f(x(t), u(t), t),\ t \in [t_0, t_f] \tag{1-2}$$

$$x(t_0) = \boldsymbol{\alpha}, \tag{1-3}$$

where $x(t) \in \mathbf{R}^{nx}$, $\dot{x}(t)$ means $\frac{d}{dt}x$, and $u(t) \in \mathbf{R}^{nc}$, $f : \mathbf{R}^{nx} \times \mathbf{R}^{nc} \to \mathbf{R}^{nx}$, and $\phi : \mathbf{R}^{nx} \to \mathbf{R}$.

The function J evaluates the system performance or *system cost*. Equation 1-2 is called the *system dynamics*, which is a group of differential equations for $x(t) \in \mathbf{R}^{nx}$ (the *state variable*). The variable $u(t)$ is the *control variable*, which is used to optimize the *system cost*. The system has an *initial time* (t_0), and a *final time* (t_f). Our numerical work only focuses on problems for which the *initial condition* (Equation 1-3) is given. Note that some problems have a *final condition* ($x(t_f) = x_f$).

CHAPTER 2
OPTIMAL CONTROL PROBLEMS

2.1 Discrete-time Systems and Runge-Kutta Discretization

Hager [2] analyzed a Runge-Kutta discretization and its convergence for unconstrained control problems. To discretize the unconstrained optimal control problem (1.0.1)-(1.0.3), we use the uniform mesh of time interval, which has the length $h = \frac{t_f - t_0}{N}$, $N \in \mathbf{N}$. Now, if we apply Butcher's [3] s-stage Runge-Kutta integration scheme with coefficients a_{ij} and b_i, $1 \le i, j \le s$ to the system dynamics (Equation 1-2), we get Equation 2-1.

$$x'_k = \sum_{i=1}^{s} b_i f(y_i, u_{ki}), \tag{2-1}$$

where $x'_k = \frac{x_{k+1} - x_k}{h}$

$$y_i = x_k + h\sum_{j=1}^{s} a_{ij} f(y_j, u_{kj}),\ 1 \le i \le s,\ 0 \le k \le N-1. \tag{2-2}$$

Therefore, the discrete control problem is Equation 2-3.

$$\text{minimize } J = \phi(x_N) \tag{2-3}$$

$$\text{subject to } x'_k = \sum_{i=1}^{s} b_i f(y_i, u_{ki}), \text{ where } x_0 = \boldsymbol{\alpha}, \tag{2-4}$$

$$y_i = x_k + h\sum_{j=1}^{s} a_{ij} f(y_j, u_{kj}),\ 1 \le i \le s,\ 0 \le k \le N-1.$$

Since $x'_k = \frac{x_{k+1} - x_k}{h}$, x_N in Equation 2-3 is obtained by solving Equation 2-5 (the *state equation*).

$$x_{k+1} = x_k + h\sum_{i=1}^{s} b_i f(y_i, u_{ki}) \text{, where } y_i = x_k + h\sum_{j=1}^{s} a_{ij} f(y_j, u_{kj}), \tag{2-5}$$

$1 \le i \le s$, $0 \le k \le N-1$, and $x_0 = \boldsymbol{\alpha}$.

Now, we explain how to compute the gradient of the cost function $\phi(x_N)$ with respect to the discrete control. To start we introduce the associated *costate equation* (Equation 2-6).

$$\psi_k = \psi_{k+1} + h\sum_{i=1}^{s} b_i \chi_i \nabla_x f(y_i, u_{ki}), \quad \psi_N = \nabla_x \phi(x_N) \tag{2-6}$$

where $\chi_i = \psi_{k+1} + \sum_{j=1}^{s} \frac{a_{ji}}{b_i} \lambda_j$ (2-7)

$$\lambda_j = hb_j \chi_j \nabla_x f(y_j, u_{kj}). \tag{2-8}$$

As shown in Hager [2],

$$\nabla_{u_{kj}} \phi(u) = hb_j \chi_j \nabla_u f(y_j, u_{kj}). \tag{2-9}$$

We used Equations 2-1 through 2-9 in our OPTCON to evaluate the function cost and function gradient. For our numerical work, we assume the Runge-Kutta scheme is explicit. The conditions for the explicit 2nd, 3rd, and 4th order of a Runge-Kutta discretization can be found in Hager [2].

2.2 Numerical Solution Methods

Analytic solution or accurate solution of optimal control problem is hard to obtain due to the complexity of the cost function $J(u)$ and the system dynamics $f(x(t), u(t), t)$. In most practical problems, numerical optimization method must be used. Figure 2-1 shows a flow chart for solving the optimal control problem.

Many methods can be applied in the optimization process. The gradient (or steepest descent) method is one of the oldest and most obvious methods, but experience has shown that this method can be extremely slow. The conjugate gradient method (such as the feasible direction method and the gradient projection method) can be applied to our project. We compared the gradient projection method with Hager and Zhang's CG_DESCENT method in Chapter 3.

2.3 Introduction to the CG_DESCENT Method

The CG_DESCENT method [1,4] is a conjugate gradient method for solving an unconstrained optimization problem

$\min\{f(x) : x \in R^n\}$, where $f : R^n \to R$ is continuously differentiable.

The variable x_k satisfies the recurrence: $x_{k+1} = x_k + \alpha_k d_k$. α_k is the step size and is positive. d_k is called the searching direction. It is generated by Equation 2-10.

$$d_{k+1} = -g_{k+1} + \beta_k d_k, \quad d_0 = -g_0. \tag{2-10}$$

CG_DESCENT method developed a special choice (Equation 2-11) for the parameter β_k.

$$\beta_k = \max\{B_k, \eta_k\}, \tag{2-11}$$

where $\eta_k = -\dfrac{1}{\|d_k\|\min\{\eta, \|g_k\|\}}$, and $B_k = \dfrac{1}{d_k^T y_k}(y_k - 2d_k \dfrac{\|y_k\|^2}{d_k^T y_k})^T g_{k+1}$.

η is a positive constant and α_k is updated by a line search procedure. It uses secant and bisecant steps for faster convergence rate. This procedure will stop as soon as the Wolfe's conditions (Equation 2-12) are satisfied.

$$\delta\phi'(0) \geq \frac{\phi(\alpha_k) - \phi(0)}{\alpha_k} \text{ and } \phi'(\alpha_k) \geq \sigma\phi'(0), \tag{2-12}$$

where $\phi(\alpha) = f(x_k + \alpha d_k)$. Note that δ and σ are positive constants satisfying $0 < \delta \leq \sigma < 1$.

The disadvantage of using the Wolfe's condition is that when the variable x_k is close to the local minimum, the term $\phi(\alpha_k) - \phi(0)$ becomes relatively inaccurate. Hence, in Hager and Zhang [1], Equation 2-12 is replaced by the approximate Wolfe conditions (Equation 2-13).

$$(2\delta - 1)\phi'(0) \geq \phi'(\alpha_k) \geq \sigma\phi'(0), \qquad (2\text{-}13)$$

where $0 < \delta < 1/2$ and $\delta \leq \sigma < 1$. This condition will be used only when the function value reaches some neighborhood of a local minimum.

By default, this method will apply approximate Wolfe condition. User can compute with the standard Wolfe conditions by setting AWolfe parameter to FALSE in the CG_DESCENT parameter file.

2.4 Applying CG_DESCENT in Optimal Control Problem

First we initialize the control variable $\{u(t_k)\}_{k=0}^{n}$: $u_0 = \{u_0(t_k)\}_{k=0}^{n}$. Then, we update the control by $u_{k+1} = u_k + \alpha_k d_k$, where α_k is the searching step size and d_k is the searching direction generated by Equation 2-10. The step size α_k is computed with the same fashion as in Section 2.3. The updating of the control will terminate if Inequality 2-14 holds.

$$\left\| \nabla_u \phi(x_N) \right\| \leq \textit{gradient tolerance} \qquad (2\text{-}14)$$

This method has been implemented in C. The program is called OPTCON.C.

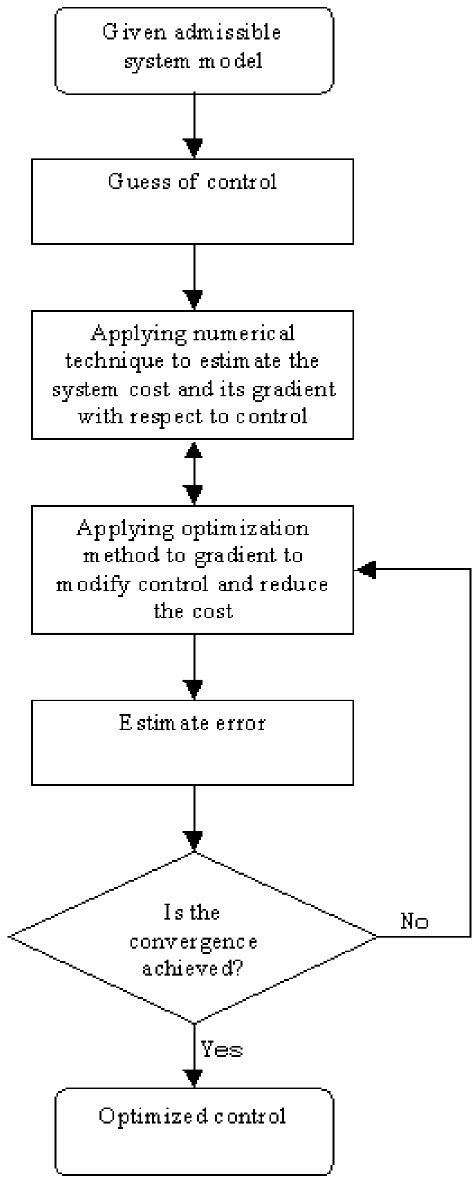

Figure 2-1. Flow chart for solving optimal control problems

CHAPTER 3
IMPLEMENTATION OF OPTCON

3.1 Introduction to OPTCON

OPTCON is a C program for solving the optimal control problems (Equation 1-1). It can be downloaded at http://www.math.ufl.edu/~lishuo/optcon.html.

In OPTCON, we provide the routine that evaluates the gradient $\nabla\tilde{J}(u)$ and the routine that evaluates the objective cost $\tilde{J}(u)$. The optimization is performed using Hager and Zhang's CG_DESCENT [1]. The main components of OPTCON are shown in Figure 3-1.

To use this program, user needs to provide a driver code, with routines that evaluate the following:

- $\phi(x(t_f))$
- $\frac{d\phi}{dx}(x(t_f))$
- $f(x,u,t) \in R^{nx}$
- $\frac{\partial f}{\partial x}(x,u,t) \in R^{nx \times nx}$
- $\frac{\partial f}{\partial u}(x,u,t) \in R^{nx \times nc}$

How to use OPTCON package and the sample problems can be found in Appendix A.

3.2 Comparison of Performance

For evaluate the performance of OPTCON, we compare with another code GP.C, which uses the gradient projection method in Hestenes [5], with an Armijo rule for step and then applies the conjugate gradient method with Polak-Ribiere update of the search direction. The source code can be found and downloaded at http://www.math.ufl.edu/~lishuo/optcon.html.

We compared performance using Problem 1 and Problem 2 (Appendix A). In these two codes, we applied the same Runge-Kutta scheme, time mesh intervals, initial guess, and gradient tolerance. Since Problem 2 is solved by the penalty approach, we also applied the same penalty factor. Data in Tables 3-1 and 3-2 shows that OPTCON with the new CG_DESCENT method runs much faster than the previous conjugate gradient method.

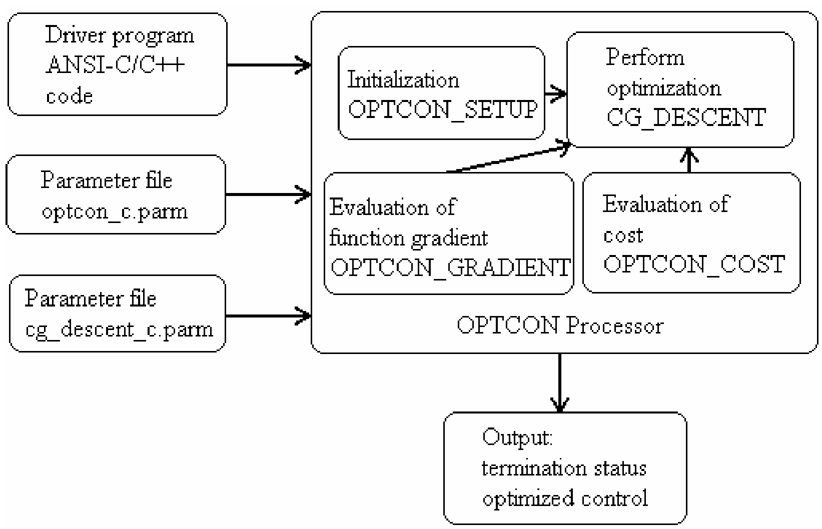

Figure 3-1. Main components of OPTCON

Table 3-1. Comparison of performance for Problem 1

Optimization method	Cost evaluation	Gradient evaluation	CG iterations	CPU time (sec)
CG_DESCENT	32 times	34 times	20	0.11
Gradient Projection	170 times	170 times	47	0.24

Table 3-2. Comparison of performance for Problem 2

Optimization method	Cost evaluation	Gradient evaluation	CG iterations	CPU time (sec)
CG_DESCENT	2616 times	3379 times	1367	18.00
Gradient Projection	25815 times	23947 times	2736	139.00

CHAPTER 4
SURVEY AND ANALYSIS

This method's performance varies with different choice of parameters and schemes. In this chapter, we will discuss performance relative to the parameters of CG_DESCENT and the Runge-Kutta schemes. Sample problems used in this Chapter are given in Appendix A.

4.1 Penalty Factor

Let us use Problem 2 (Appendix A). We solve this problem by using a penalty approach. Since the problem provides the final-time boundary conditions: $x_2(t_f) = 0$ and $x_3(t_f) = \sqrt{\frac{\mu}{x_1(t_f)}}$, we try to minimize the terms: $|x_2(t_N) - 0|$ and $\left|x_3(t_N) - \sqrt{\frac{\mu}{x_1(t_N)}}\right|$.

Hence, the target cost function becomes Equation 4-1.

$$\begin{aligned}\widetilde{J}(x_1, x_2, x_3, \theta, t) = -x_1(t_N) + P(x_2(t_N) - 0)^2 + P\left(x_3(t_N) - \sqrt{\frac{\mu}{x_1(t_N)}}\right)^2 \\ + \lambda_k^T \begin{pmatrix} x_2(t_N) \\ x_3(t_N) - \sqrt{\frac{\mu}{x_1(t_N)}} \end{pmatrix}\end{aligned}, \qquad (4\text{-}1)$$

where P is the penalty factor. Equation 4-2 updates the multiplier vector λ_k.

$$\lambda_{k+1} = \lambda_k + 2P\begin{pmatrix} x_2(t_N) \\ x_3(t_N) - \sqrt{\frac{\mu}{x_1(t_N)}} \end{pmatrix} \qquad (4\text{-}2)$$

We will recursively apply OPTCON, and update the multiplier vector λ_k in each iteration. Now, let's try different values of penalty factor. Table 4-1 shows the infinity norm of the state error and the total CPU time for different choice of penalty factors. Note: we apply time mesh size $N = 500$, Runge-Kutta scheme 2 (Table A-1) and $grad\_tol = 10^{-5}$ for all iterations. Greater penalty factor yields faster convergence rate. However, higher convergence rate is sacrificed with longer CPU time. Note that when you choose penalty factor P less than 1000, this method will not converge.

4.2 The Gradient Tolerance Factor

For Equation 4-2 we solve the discrete problem (Equation 4-1) to the level of accuracy using CG_DESCENT. The convergence criterion in CG_DESCENT is based on the $\infty - norm$ of the gradient. The code terminates when the $\infty - norm$ is less than an input parameter *grad_tol*.

Let *c* be the gradient tolerance factor. We first apply $grad\_tol = 10^{-5}$. Then, we compute $\|x(t_N) - x(t_f)\|_\infty$ and set $grad\_tol = c \cdot \|x(t_N) - x(t_f)\|_\infty$ for the sub problem. Data in Tables B-1 to B-9 result in the performance shown in Table 4-2. Based on the total CPU time listed in Table 4-2, we obtained Figure 4-1.

Comparing the performance with the different value of *c*, we can conclude the following.

- Very large *c* will not work. In this case, when $c = 100000$ the computation does not converge. CPU time $\to \infty$ as $c \geq 100000$.
- Choosing smaller *c* can take longer CPU time.
- The largest *c* such that computation converges provides the best CPU time.

4.3 Line Search Parameters

In the previous computation, we always used an approximate Wolfe line search [1]. Now, we will use a combination line search where the ordinary Wolfe conditions are used until the value of Equation 4-1 average is sufficiently small. Then, we switch to approximate Wolfe conditions. To do this, we should set "AWolfe" parameter in CG_DESCENT_C.PARM to "FALSE".

Data in Tables B-10 to B-17 result in the performance shown in Table 4-3. Comparing the performance of OPTCON using combination line search conditions with the previous one, we obtained Figure 4-2. It is clear that the performance is improved when we applying a combination line search. The best CPU time is 15 seconds, one fourth faster than the previous one.

4.4 Applying Different Runge-Kutta Schemes

OPTCON provides 6 optional explicit Runge-Kutta schemes (Table A-1). The performance varies with different schemes. For Problem 1 (Appendix A), the discrete state error in L^{∞} for different choice of Runge-Kutta schemes and different time mesh is shown in Table 4-4. Scheme 6 which has the fourth order of accuracy provides the best discretization error for this problem. To obtain the error of the order 10^{-8}, we need to take N=2000, 160, 1000, 2000, 2000, 40 for schemes 1-6 respectively.

Table 4-5 shows the performance of Runge-Kutta schemes 1-6 applying to the Problem 2 (Appendix A) with respect to $P = 10000$ and the corresponding N. It shows that scheme 6 provides the best CPU time. Among the third-order schemes (schemes 2-5), scheme 2 gives the best performance.

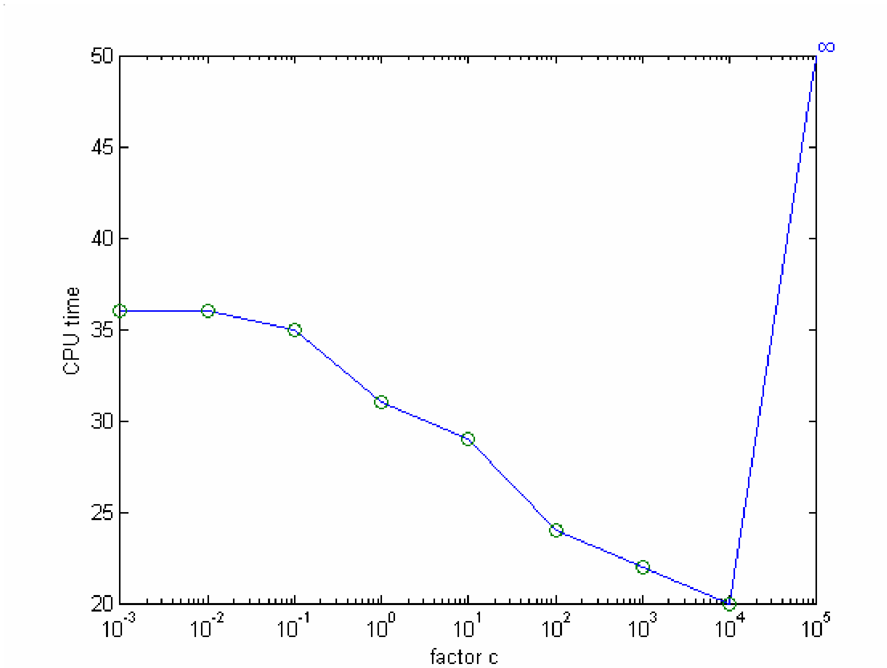

Figure 4-1. Relationship of gradient tolerance factor *c* and CPU time

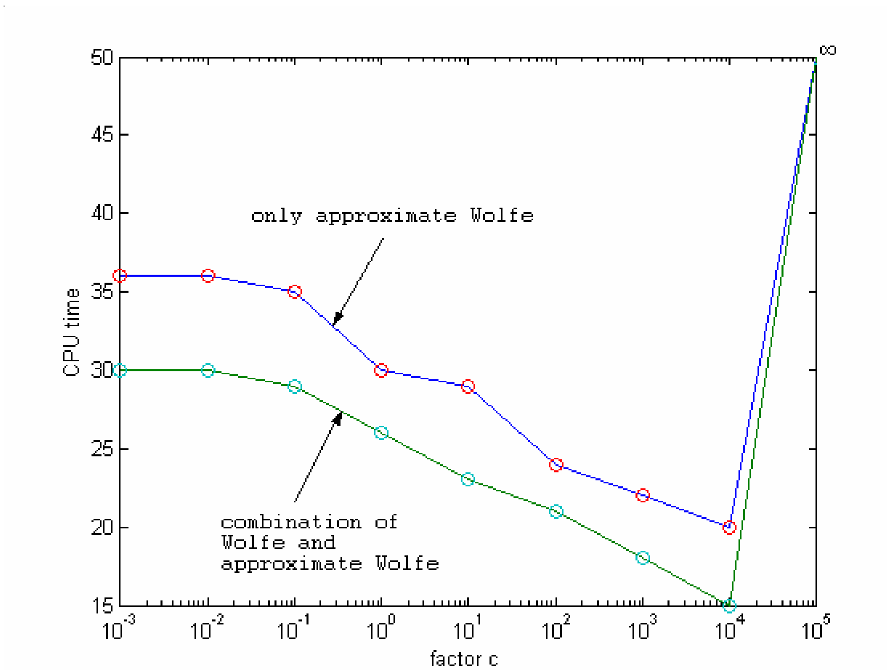

Figure 4-2. Comparison of CPU times when applying a combination of Wolfe/approximate Wolfe

Table 4-1. Discrete state error in L^∞ and CPU time with different penalty factors

Iterations	P = 1000	P = 5000	P = 10000	P = 50000	P = 500000
1	5.87e-003	1.43e-003	7.38e-004	1.52e-004	1.53e-005
2	1.52e-003	1.35e-004	3.84e-005	1.71e-006	1.76e-008
3	6.30e-004	1.66e-005	2.51e-006	2.37e-008	2.21e-011
4	2.59e-004	2.04e-006	1.64e-007	3.39e-010	3.61e-011
5	1.06e-004	2.49e-007	1.07e-008	1.78e-011	CPU time: 36 sec
6	4.37e-005	3.05e-008	6.56e-010	3.49e-011	
7	1.79e-005	3.66e-009	9.49e-011	CPU time: 33 sec	
8	7.37e-006	4.97e-010	2.33e-011		
9	3.03e-006	1.13e-010	2.33e-011		
10	1.24e-006	2.16e-011	CPU time: 27 sec		
11	5.11e-007	1.78e-011			
12	2.10e-007	1.78e-011			
13	8.62e-008	CPU time: 20 sec			
14	3.53e-008				
15	1.46e-008				
16	5.95e-009				
17	2.41e-009				
18	1.34e-009				
19	4.10e-010				
20	3.71e-010				
21	1.65e-010				
22	1.70e-010				
	CPU time: 14 sec				

Table 4-2. Performance of OPTCON using different value of gradient tolerance factor

Factor *c*	CPU time	$\|x(t_N)-x(t_f)\|_\infty$	CG iterations
100,000	Unbounded		
10,000	20.00 sec.	4.47e-011	1693
1,000	22.00 sec.	6.76e-012	1917
100	24.00 sec.	2.36e-011	2119
10	29.00 sec.	2.53e-011	2578
1	31.00 sec.	2.63e-011	2548
0.1	35.00 sec.	9.18e-012	2842
0.01	36.00 sec.	3.17e-011	3053
0.001	36.00 sec.	3.17e-011	3053

Table 4-3. Performance of OPTCON using a combination line search conditions

Factor c	CPU time	$\|x(t_N) - x(t_f)\|_\infty$	CG iterations
100,000	Unbounded		
10,000	15.00 sec.	5.17e-011	1279
1,000	18.00 sec.	2.33e-011	1453
100	21.00 sec.	2.11e-011	1758
10	23.00 sec.	2.27e-011	1843
1	26.00 sec.	4.08e-011	2224
0.1	29.00 sec.	2.76e-011	2401
0.01	30.00 sec.	4.25e-011	2406
0.001	30.00 sec.	4.25e-011	2406

Table 4-4. Discrete state error in L^∞ for Problem 1 and schemes 1-6

time mesh	Scheme 1	Scheme 2	Scheme 3	Scheme 4	Scheme 5	Scheme 6
N=2000	7.0e-008	8.0e-012	1.5e-008	3.5e-008	6.0e-008	1.4e-013
N=1000	2.8e-007	6.4e-011	6.1e-008	1.4e-007	2.4e-007	1.6e-013
N=600	7.8e-007	3.0e-010	1.7e-007	3.9e-007	6.7e-007	5.4e-013
N=320	2.7e-006	2.0e-009	5.9e-007	1.4e-006	2.4e-006	5.9e-012
N=160	1.1e-005	1.6e-008	2.4e-006	5.5e-006	9.5e-006	9.6e-011
N=80	4.4e-005	1.3e-007	9.7e-006	2.2e-005	3.8e-005	1.5e-009
N=40	1.8e-004	1.1e-006	3.9e-005	8.9e-005	1.6e-004	2.4e-008

Table 4-5. Discrete state error in L^∞ and CPU time for Problem 2 and schemes 1-6

	Scheme 1 N=2000	Scheme 2 N=160	Scheme 3 N=1000	Scheme 4 N=2000	Scheme 5 N=2000	Scheme 6 N=40
$\|x(t_N) - x(t_f)\|_\infty$	4.2e-011	7.4e-011	9.3e-012	2.6e-011	2.6e-011	7.6e-012
CPU time	27.00 sec	6.00 sec	69.00 sec	43.00 sec	65.00 sec	1.00 sec

APPENDIX A
HOW TO USE OPTCON

System Requirements

OPTCON uses GNU **gcc** complier, which is available on most UNIX systems. Availability of memories for running the program depends on the complexity of the problem. Once you run OPTCON, it will first check the availability of your computer memory.

Parameter File and Default Values

The OPTCON package has two parameter files. One file is cg_descent_c.parm, which is used in CG_DESCENT. The meaning of the parameters and their default values can be found in [6]. Another parameter file optcon_c.parm is used in OPTCON subroutine. The parameters are as follows:

- PrintLevel (to print the result for each iteration)
- PrintFinal (to print the final result)
- scheme (choice of explicit Runge-Kutta schemes)

Table A-1 gives the options of schemes. Table A-2 gives the default values of OPTCON parameters.

Running OPTCON

To run the OPTCON, the user needs to create a driver program, which should be placed in the same directory where the OPTCON package stored. I will demonstrate how to use OPTCON with the following example.

Problem 1:

$$\text{minimize } \frac{1}{2}\int_0^1 u(t)^2 + 2x(t)^2 dt \tag{A-1}$$

$$\text{subject to } x'(t) = .5x(t) + u(t), x(0) = 1 \tag{A-2}$$

Problem 1 [2] can be solved with the analytic optimal solution:

$$x^*(t) = \frac{2e^{3t} + e^3}{e^{3t/2}(2+e^3)}, \quad u^*(t) = \frac{2(e^{3t} - e^3)}{e^{3t/2}(2+e^3)}.$$

Now, let $x(t)$ be denoted by $x_1(t)$ and $x_2'(t) = 2x_1(t)^2 + u(t)^2$, by fundamental theorem of calculus, the original cost function (Equation A-1) equals to Equation A-3.

$$\text{minimize } \frac{1}{2}[x_2(1) - x_2(0)]. \tag{A-3}$$

Let $x_2(0) = 0$, we transformed Problem 1 into Equations A-4 through A-6.

$$\text{minimize } \phi = \frac{1}{2}x_2(1) \tag{A-4}$$

$$\text{subject to } x_1'(t) = .5x_1(t) + u(t), x_1(0) = 1 \tag{A-5}$$

$$x_2'(t) = 2x_1(t)^2 + u(t)^2, x_2(0) = 0. \tag{A-6}$$

Now, we create the driver program-driver1.c as shown in Figure A-1 for the transformed system (Equations A-4 to A-6). Note that we use the default values of the OPTCON parameters. Lines 9 to 12 of Figure A-1 defined the number of time mesh intervals-*n*, number of state variables-*nx*, number of controls-*nc* and number of stage(s)-*ns* in the Runge-Kutta scheme. Note that the size of the state array is $n \times ns \times nx + nx$ (see line 29 of Figure A-1), in which the last nx elements store the final state. Let $x_{i,j}^k$ be the discrete state, where *k* is the time level $0 \le k \le n-1$, *i* is the component of state $1 \le i \le nx$, and *j* is the stage in Runge-Kutta scheme $1 \le j \le ns$. We store $x_{i,j}^k$ in the array state as the following:

For each fixed *j* and *k*, we first increment *i* from 1 to *nx*, next we increment *j* from 1 to *ns*, and finally we increment *k* from 0 to $n-1$. The discrete control $u^k_{i',j}$ ($1 \le i' \le nc$) is stored in the control array (defined in line 30 of Figure A-1) in the same fashion. The user-provided routines include (lines 49 to 73 in Figure A-1)

- *double my_phi(double *x_f)* – the routine that evaluates the cost function $\phi(x(t_f))$. The input is "*double *x_f*"-a pointer points to the first element of the array *x_f* which contains the final states. The output is a double precision value of cost $\phi(x(t_f))$.

- *void my_dphi(double *dphi, double *x_f)* – the routine that evaluates $\frac{d\phi}{dx}(x(t_f))$. The input is "*double *x_f*". The output is the array *dphi* that contains the value of $\frac{d\phi}{dx}(x(t_f)) \in R^{nx}$.

- *void my_f(double *f, double *x, double *u, double time)* – the routine that evaluates the system dynamic function $f(x,u,t) \in R^{nx}$. Inputs are: "*double *x*"-an array contains the states *x*, "*double *u*"-an array contains the controls *u*, and "*double time*"-time *t*. The output is "*double *f*"-an array contains the value of $f(x,u,t) \in R^{nx}$.

- *void my_fx(double *fx, double *x, double *u, double time)* – the routine that evaluates $\frac{\partial f}{\partial x}(x,u,t) \in R^{nx \times nx}$. "*Double *x*", "*double *u*", and "*double time*" are inputs. The output is "double **fx*"-an array contains the value of $\frac{\partial f}{\partial x}(x,u,t) \in R^{nx \times nx}$. Note that we store the matrix by rows (see lines 63-66 of Figure A-1).

- void my_fu(double *fu, double *x, double *u, double time) – the routine that evaluates $\frac{\partial f}{\partial u}(x,u,t) \in R^{nx \times nc}$. "*Double *x*", "*double *u*", and "*double time*" are inputs. The output is "*double *fu*"-an array contains the value of $\frac{\partial f}{\partial u}(x,u,t) \in R^{nx \times nc}$. Note that we store the matrix by rows (see lines 70 and 71 of Figure A-1).

We applied the provided Runge-Kutta scheme in this example. To provide your own Runge-Kutta scheme, you should set the "scheme" parameter to 0 and initialize the *a* and b arrays to contain the coefficients of the scheme (as shown in Figure A-2, lines 20 and 21). Note that we store the coefficients of Runge-Kutta matrix A by rows.

After calling OPTCON (line 42 in Figure A-1 or line 43 in Figure A-2), we obtain the final state (the last nx elements in the state array) and the discrete optimal control for Problem 1.

Now, let's consider another example, which has both initial conditions and final conditions.

Problem 2: orbit transfer problem (see Bryson and Ho [7] page 66-68)

For a constant-thrust rocket with thrust T, operating from time 0 to time t_f, we want to find an optimal thrust angle history $\theta(t)$ that transfer the rocket from an initial given orbit to the largest possible circular orbit. Now, let's transform it into a mathematical model. First, the variables and parameters are given by: x_1-the radius of the orbit with an attracting center; x_2-the radial velocity; x_3-the tangential velocity; m_0-the mass of the rocket; $\dot{m}$-the fuel consumption rate; $\theta(t)$-the history of thrust angle; μ-the gravity constant of the attracting center. Then, the model is formulated by Equations A-7 to A-15.

$$\text{maximize } x_1(t_f), \tag{A-7}$$

System dynamics:

$$x_1' = x_2, \tag{A-8}$$

$$x_2' = \frac{x_3^{\ 2}}{x_1} - \frac{\mu}{x_1^{\ 2}} + \frac{T\sin\theta}{m_0 - |\dot{m}|t}, \tag{A-9}$$

$$x_3' = -\frac{x_2 x_3}{x_1} + \frac{T\cos\theta}{m_0 - |\dot{m}|t}. \tag{A-10}$$

Initial conditions:

$$x_1(0) = a, \tag{A-11}$$
$$x_2(0) = 0, \tag{A-12}$$
$$x_3(0) = \sqrt{\frac{\mu}{a}}. \tag{A-13}$$

Final conditions:

$$x_2(t_f) = 0, \tag{A-14}$$
$$x_3(t_f) = \sqrt{\frac{\mu}{x_1(t_f)}}. \tag{A-15}$$

Where, $m_0 = 10{,}000kg, \dot{m} = 12.9kg/day, a = 149.6 \times 10^9 m, T = 8.336N,$

$\mu = 1.3273310^{20} m^3/s^2$ and $t_f = 193 days$.

Since the final conditions are given, we solve this problem using a penalty approach. At step *k*, the cost function is given by Equation A-16,

$$\begin{aligned}\phi(x_1(t_N), x_2(t_N), x_3(t_N)) = -x_1(t_N) + Px_2(t_N)^2 + P\left(x_3(t_N) - \sqrt{\frac{\mu}{x_1(t_N)}}\right)^2 \\ + \lambda_{1,k} x_2(t_N) + \lambda_{2,k}\left(x_3(t_N) - \sqrt{\frac{\mu}{x_1(t_N)}}\right)\end{aligned} \tag{A-16}$$

with Lagrange multipliers λ_1 and λ_2, and a constant penalty factor P. At step $k+1$, λ_1 and λ_2 are updated by Equations A-17 and A-18.

$$\lambda_{1,k+1} = \lambda_{1,k} + 2Px_2(t_N) \tag{A-17}$$
$$\lambda_{2,k+1} = \lambda_{2,k} + 2P\left(x_3(t_N) - \sqrt{\frac{\mu}{x_1(t_N)}}\right) \tag{A-18}$$

Note that $\lambda_{1,0}$ and $\lambda_{2,0}$ are given. By penalty approach, we solve this problem by recursively calling OPTCON, each iteration involved in updating the value of λ_1 and λ_2.

The program will stop when the value $|x_2(t_N)| + \left|x_3(t_N) - \sqrt{\frac{\mu}{x_1(t_N)}}\right|$ is no longer decreasing. Please refer to the driver program-driver2.c, which can be found in the OPTCON package.

```c
#include "optcon.h"

double my_phi(double *x_f);
void my_dphi(double *dphi, double *x_f);
void my_f(double *f, double *x, double *u, double time);
void my_fx(double *fx, double *x, double *u, double time);
void my_fu(double *fu, double *x, double *u, double time);

int n = 500; // number of time mesh intervals
int nx = 2;  // number of state variables
int nc = 1;  // number of controls
int ns = 3;  // number of stage(s) in Runge-Kutta scheme

int main(void) {
       double t0, tf;
       double grad_tol = 1.0e-15;
       double *state0;
       double *state;
       double *control;
       double *a, *b; // Runge-Kutta matrix a and vector b
       int i;

       /* define initial time and final time */
       t0 = 0;
       tf = 1;

       /* allocate computer memory */
       state0 = (double*) malloc(nx*sizeof(double));
       state = (double*) malloc((n*ns*nx+nx)*sizeof(double));
       control = (double*) malloc(n*ns*nc*sizeof(double));

       /* guess of control */
       for ( i = 0; i < n*ns*nc; i ++ ) {
           control[i] = .5;
       }

       /* initial states */
       state0[0] = 1;
       state0[1] = 0;

       /* call OPTCON */
       optcon(grad_tol, n, nx, nc, ns, t0, tf, control, state0,
              state, a, b, my_phi, my_dphi, my_f, my_fx, my_fu);

       free(state0)
       free(state);
       free(control);
}
double my_phi(double *x_f) {
     return .5*x_f[1];
}
void my_dphi(double *dphi, double *x_f) {
     dphi[0] = 0;
     dphi[1] = .5;
     return;
}
void my_f(double *f, double *x, double *u, double time) {
     f[0] = .5*x[0] + u[0];
     f[1] = u[0]*u[0] + 2*x[0]*x[0];
     return;
}
void my_fx(double *fx, double *x, double *u, double time) {
     fx[0] = 0.5;
     fx[2] = 4*x[0];
     fx[1] = 0;
     fx[3] = 0;
     return;
}
void my_fu(double *fu, double *x, double *u, double time) {
     fu[0] = 1;
     fu[1] = 2*u[0];
     return;
}
```

Figure A-1. Driver1.c

Suppose the user-provide Runge-Kutta matrix A and vector b are: $A=\begin{pmatrix} 0 & 0 & 0 \\ .5 & 0 & 0 \\ .5 & .5 & 0 \end{pmatrix}$, $b=\begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix}$

```
int main(void) {
        double t0, tf;
        double grad_tol = 1.0e-15;
        double *state0;
        double *state;
        double *control;
        double a[9] = {0, 0, 0, .5, 0, 0, .5, .5, 0}; // Runge-Kutta matrix a
        double b[3] = {1./3., 1./3., 1./3.}; // Runge-Kutta vector b
        int i;

        /* define initial time and final time */
        t0 = 0;
        tf = 1;

        /* allocate computer memory */
        state0 = (double*) malloc(nx*sizeof(double));
        state = (double*) malloc((n*ns*nx+nx)*sizeof(double));
        control = (double*) malloc(n*ns*nc*sizeof(double));

        /* guess of control */
        for ( i = 0; i < n*ns*nc; i ++ ) {
            control[i] = .5;
        }

        /* initial states */
        state0[0] = 1;
        state0[1] = 0;

        /* call OPTCON */
        optcon(grad_tol, n, nx, nc, ns, t0, tf, control, state0,
                state, a, b, my_phi, my_dphi, my_f, my_fx, my_fu);

        free(state0)
        free(state);
        free(control);
}
```

Figure A-2. Segment of Driver1.c if user provides Runge-Kutta scheme

Table A-1. Options for explicit Runge-Kutta schemes

scheme 1	scheme 2
$A = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, b = \begin{pmatrix} 1/2 \\ 1/2 \end{pmatrix}$	$A = \begin{pmatrix} 0 & 0 & 0 \\ 1/2 & 0 & 0 \\ -1 & 2 & 0 \end{pmatrix}, b = \begin{pmatrix} 1/6 \\ 2/3 \\ 1/6 \end{pmatrix}$
scheme 3	scheme 4
$A = \begin{pmatrix} 0 & 0 & 0 \\ 1/2 & 0 & 0 \\ 0 & 3/4 & 0 \end{pmatrix}, b = \begin{pmatrix} 2/9 \\ 1/3 \\ 4/9 \end{pmatrix}$	$A = \begin{pmatrix} 0 & 0 & 0 \\ 1/2 & 0 & 0 \\ 1/2 & 1/2 & 0 \end{pmatrix}, b = \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix}$
scheme 5	scheme 6
$A = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1/4 & 1/4 & 0 \end{pmatrix}, b = \begin{pmatrix} 1/6 \\ 1/6 \\ 2/3 \end{pmatrix}$	$A = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, b = \begin{pmatrix} 1/6 \\ 1/3 \\ 1/3 \\ 1/6 \end{pmatrix}$

Table A-2. Parameters' default values in optcon_c.parm

Parameter's name	Default value
PrintLevel	0
PrintFinal	1
scheme	2

APPENDIX B
TESTING DATA FOR GRADIENT TOLERANCE FACTOR

We test different value of gradient tolerance factor c for Problem 2 in Appendix A. Let's apply time mesh size $N = 500$, scheme 2 (Table A-1), penalty factor $P = 10000$, and initial $grad\_tol = 10^{-5}$.

Using only approximate Wolfe line search conditions in CG_DESCENT, we obtain Tables B-1 to Table B-9. Using a combination of Wolfe and approximate Wolfe line search conditions in CG_DESCENT, we obtain Tables B-10 to Table B-17.

Table B-1. Approximate Wolfe line search, c = 100000

Iteration #	$\lVert x(t_N) - x(t_f) \rVert_\infty$	CPU time	CG iterations
1	4.54e-004	11.00 sec	1128
2	7.44e-005	0.00 sec	2
3	3.80e-005	0.00 sec	6
4	5.48e-006	1.00 sec	12
5	9.31e-006	0.00 sec	39
		Not convergent	

Table B-2. Approximate Wolfe line search, c = 10000

Iteration #	$\lVert x(t_N) - x(t_f) \rVert_\infty$	CPU time	CG iterations
1	4.54e-004	12.00 sec	1128
2	4.24e-005	0.00 sec	22
3	1.02e-005	0.00 sec	28
4	5.21e-007	0.00 sec	26
5	6.42e-008	1.00 sec	66
6	3.20e-009	1.00 sec	35
7	4.98e-010	1.00 sec	136
8	1.06e-010	1.00 sec	60
9	7.25e-011	1.00 sec	49
10	2.58e-011	2.00 sec	120
11	1.50e-011	0.00 sec	4
12	3.91e-012	1.00 sec	18
13	4.47e-011	0.00 sec	1
		Total: 20.00 sec	

Table B-3. Approximate Wolfe line search, c = 1000

Iteration #	$\|x(t_N) - x(t_f)\|_\infty$	CPU time	CG iterations
1	4.54e-004	12.00 sec	1128
2	2.89e-005	0.00 sec	80
3	2.58e-006	2.00 sec	91
4	2.42e-007	0.00 sec	82
5	1.88e-008	2.00 sec	114
6	1.00e-009	1.00 sec	109
7	8.39e-011	1.00 sec	97
8	4.60e-011	3.00 sec	184
9	1.27e-011	1.00 sec	9
10	5.80e-012	0.00 sec	7
11	6.76e-012	0.00 sec	16
		Total: 22.00 sec	

Table B-4. Approximate Wolfe line search, c = 100

Iteration #	$\|x(t_N) - x(t_f)\|_\infty$	CPU time	CG iterations
1	4.54e-004	11.00 sec	1128
2	2.95e-005	2.00 sec	133
3	2.04e-006	2.00 sec	138
4	1.35e-007	1.00 sec	132
5	8.95e-009	3.00 sec	288
6	5.95e-010	3.00 sec	231
7	7.19e-011	1.00 sec	33
8	1.10e-011	1.00 sec	21
9	2.36e-011	0.00 sec	15
		Total: 24.00 sec	

Table B-5. Approximate Wolfe line search, c = 10

Iteration #	$\|x(t_N) - x(t_f)\|_\infty$	CPU time	CG iterations
1	4.54e-004	12.00 sec	1128
2	2.96e-005	2.00 sec	172
3	1.92e-006	2.00 sec	208
4	1.26e-007	3.00 sec	294
5	8.19e-009	5.00 sec	410
6	5.51e-010	2.00 sec	171
7	8.01e-011	0.00 sec	27
8	2.03e-011	1.00 sec	51
9	7.67e-012	1.00 sec	87
10	2.53e-011	1.00 sec	30
		Total: 29.00 sec	

Table B-6. Approximate Wolfe line search, c = 1

Iteration #	$\lVert x(t_N) - x(t_f) \rVert_\infty$	CPU time	CG iterations
1	4.54e-004	12.00 sec	1128
2	2.96e-005	3.00 sec	246
3	1.93e-006	4.00 sec	342
4	1.26e-007	5.00 sec	355
5	8.25e-009	2.00 sec	175
6	5.22e-010	2.00 sec	114
7	3.65e-011	1.00 sec	88
8	1.09e-011	1.00 sec	97
9	4.12e-012	1.00 sec	1
10	2.63e-011	0.00 sec	2
		Total: 31.00 sec	

Table B-7. Approximate Wolfe line search, c = 0.1

Iteration #	$\lVert x(t_N) - x(t_f) \rVert_\infty$	CPU time	CG iterations
1	4.54e-004	12.00 sec	1128
2	2.96e-005	4.00 sec	350
3	1.93e-006	5.00 sec	377
4	1.26e-007	4.00 sec	363
5	8.17e-009	4.00 sec	258
6	5.52e-010	2.00 sec	150
7	9.22e-011	0.00 sec	52
8	4.66e-011	1.00 sec	37
9	3.65e-011	2.00 sec	76
10	1.54e-011	0.00 sec	2
11	1.32e-011	0.00 sec	10
12	6.47e-012	1.00 sec	19
13	9.18e-012	0.00 sec	20
		Total: 35.00 sec	

Table B-8. Approximate Wolfe line search, c = 0.01

Iteration #	$\lVert x(t_N) - x(t_f) \rVert_\infty$	CPU time	CG iterations
1	4.54e-004	11.00 sec	1128
2	2.96e-005	6.00 sec	441
3	1.93e-006	4.00 sec	404
4	1.26e-007	4.00 sec	327
5	8.21e-009	5.00 sec	378
6	5.16e-010	3.00 sec	200
7	7.05e-011	0.00 sec	48
8	2.31e-011	2.00 sec	104
9	3.17e-011	1.00 sec	23
		Total: 36.00 sec	

Table B-9. Approximate Wolfe line search, c = 0.001

Iteration #	$\|x(t_N) - x(t_f)\|_\infty$	CPU time	CG iterations
1	4.54e-004	11.00 sec	1128
2	2.96e-005	5.00 sec	441
3	1.93e-006	5.00 sec	404
4	1.26e-007	4.00 sec	327
5	8.21e-009	5.00 sec	378
6	5.16e-010	3.00 sec	200
7	7.05e-011	0.00 sec	48
8	2.31e-011	2.00 sec	104
9	3.17e-011	1.00 sec	23
		Total: 36.00 sec	

Table B-10. Combination line search, c = 10000

Iteration #	$\|x(t_N) - x(t_f)\|_\infty$	CPU time	CG iterations
1	4.54e-004	8.00 sec	807
2	4.25e-005	0.00 sec	22
3	1.00e-005	1.00 sec	31
4	8.34e-007	0.00 sec	26
5	5.21e-008	1.00 sec	48
6	1.58e-008	1.00 sec	59
7	1.99e-010	0.00 sec	51
8	6.55e-011	2.00 sec	138
9	4.86e-011	1.00 sec	74
10	4.79e-011	0.00 sec	3
11	5.17e-011	1.00 sec	20
		Total: 15.00 sec	

Table B-11. Combination line search, c = 1000

Iteration #	$\|x(t_N) - x(t_f)\|_\infty$	CPU time	CG iterations
1	4.54e-004	8.00 sec	807
2	2.82e-005	1.00 sec	71
3	3.36e-006	1.00 sec	79
4	1.15e-007	1.00 sec	60
5	9.48e-009	2.00 sec	132
6	8.46e-010	1.00 sec	95
7	4.25e-011	2.00 sec	114
8	2.35e-011	0.00 sec	34
9	1.70e-011	1.00 sec	15
10	2.33e-011	1.00 sec	46
		Total: 18.00 sec	

Table B-12. Combination line search, c = 100

Iteration #	$\|x(t_N) - x(t_f)\|_\infty$	CPU time	CG iterations
1	4.54e-004	8.00 sec	807
2	2.97e-005	1.00 sec	138
3	1.83e-006	2.00 sec	140
4	1.26e-007	1.00 sec	129
5	7.74e-009	4.00 sec	314
6	4.82e-010	1.00 sec	82
7	6.55e-011	1.00 sec	50
8	9.31e-012	1.00 sec	83
9	2.11e-011	1.00 sec	15
		Total: 21.00 sec	

Table B-13. Combination line search, c = 10

Iteration #	$\|x(t_N) - x(t_f)\|_\infty$	CPU time	CG iterations
1	4.54e-004	8.00 sec	807
2	2.96e-005	2.00 sec	181
3	1.92e-006	2.00 sec	184
4	1.26e-007	3.00 sec	234
5	8.23e-009	3.00 sec	250
6	5.04e-010	2.00 sec	139
7	6.21e-011	1.00 sec	28
8	6.92e-012	0.00 sec	17
9	2.27e-011	1.00 sec	3
		Total: 23.00 sec	

Table B-14. Combination line search, c = 1

Iteration #	$\|x(t_N) - x(t_f)\|_\infty$	CPU time	CG iterations
1	4.54e-004	8.00 sec	807
2	2.96e-005	2.00 sec	233
3	1.93e-006	4.00 sec	306
4	1.26e-007	4.00 sec	373
5	8.19e-009	3.00 sec	240
6	5.62e-010	3.00 sec	180
7	3.74e-011	1.00 sec	80
8	4.08e-011	1.00 sec	5
		Total: 26.00 sec	

Table B-15. Combination line search, c = 0.1

Iteration #	$\lVert x(t_N) - x(t_f) \rVert_\infty$	CPU time	CG iterations
1	4.54e-004	8.00 sec	807
2	2.96e-005	4.00 sec	358
3	1.93e-006	5.00 sec	368
4	1.26e-007	3.00 sec	275
5	8.23e-009	3.00 sec	237
6	5.32e-010	3.00 sec	165
7	4.86e-011	0.00 sec	44
8	1.68e-011	2.00 sec	97
9	1.52e-011	0.00 sec	5
10	2.76e-011	1.00 sec	45
		Total: 29.00 sec	

Table B-16. Combination line search, c = 0.01

Iteration #	$\lVert x(t_N) - x(t_f) \rVert_\infty$	CPU time	CG iterations
1	4.54e-004	8.00 sec	807
2	2.96e-005	6.00 sec	432
3	1.93e-006	5.00 sec	354
4	1.26e-007	3.00 sec	260
5	8.20e-009	4.00 sec	320
6	5.40e-010	2.00 sec	132
7	3.97e-011	1.00 sec	55
8	2.51e-011	1.00 sec	20
9	4.25e-011	0.00 sec	26
		Total: 30.00 sec	

Table B-17. Combination line search, c = 0.001

Iteration #	$\lVert x(t_N) - x(t_f) \rVert_\infty$	CPU time	CG iterations
1	4.54e-004	8.00 sec	807
2	2.96e-005	6.00 sec	432
3	1.93e-006	4.00 sec	354
4	1.26e-007	4.00 sec	260
5	8.20e-009	4.00 sec	320
6	5.40e-010	2.00 sec	132
7	3.97e-011	1.00 sec	55
8	2.51e-011	1.00 sec	20
9	4.25e-011	0.00 sec	26
		Total: 30.00 sec	

LIST OF REFERENCES

1. W. W. Hager and H. Zhang, “A new conjugate gradient method with guaranteed descent and an efficient line search”, *SIAM Journal on Optimization*, 16, 2005, pp. 170-192.

2. W. W. Hager, “Runge-Kutta methods in optimal control and the transformed adjoint system”, *Numerische Mathematik*, 87, 2000, pp. 247-282.

3. J. C. Butcher, *The numerical analysis of ordinary differential equations*, John Wiley, New York, 1987.

4. W. W. Hager and H. Zhang, *Source code C version 1.2*, November 14, 2005, University of Florida, Gainesville, FL, URL: http://www.math.ufl.edu/~hager/papers/CG/cg_descent-C-1.2.tar.gz, last accessed: May 14, 2006.

5. M. R. Hestenes, *Conjugate direction methods in optimization*, Springer-Verlag, New York, 1980.

6. W. W. Hager and H. Zhang, *CG_DESCENT version 1.4, user's guide*, November 14, 2005, University of Florida, Gainesville, FL, URL: http://www.math.ufl.edu/~hager/papers/CG/cg_manual-1.4.ps, last accessed: May 14, 2006.

7. A. E. Bryson and Jr., Y.-C. Ho, *Applied optimal control*, Blaisdell, Waltham, MA, 1969.

BIOGRAPHICAL SKETCH

Shuo Li was born in Beijing, China, and completed her bachelor's degree in applied mathematics from Beijing Polytechnic University in July 2000. She obtained her M.S. in applied mathematics, specializing in numerical optimization (under the supervision of Professor William W. Hager) from the University of Florida, August 2006.