

RICE UNIVERSITY

**The Effects of Coupling Adaptive Time-Stepping and Adjoint-State**

**Methods for Optimal Control Problems**

by

**Marco U. Enriquez**

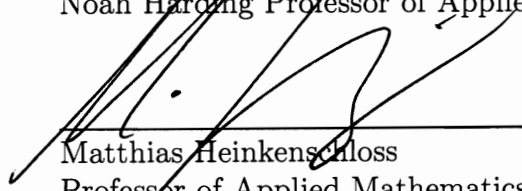
A THESIS SUBMITTED  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE

**Doctor of Philosophy**

APPROVED, THESIS COMMITTEE:



William Symes, Chair  
Noah Harding Professor of Applied Mathematics



Matthias Heinkenschloss  
Professor of Applied Mathematics



Daniel Cohan  
Assistant Professor of Environmental Engineering

HOUSTON, TEXAS

DECEMBER 2010

## **Abstract**

The Effects of Coupling Adaptive Time-Stepping and Adjoint-State Methods for  
Optimal Control Problems

by

Marco U. Enriquez

This thesis presents the implications of using adaptive time-stepping schemes with the adjoint-state method, a widely used algorithm for computing derivatives in optimal-control problems. Though we gain control over the accuracy of the time-stepping scheme, the forward and adjoint time grids become mismatched. Despite this fact, I claim using adaptive time-stepping for optimal control problems is advantageous for two reasons. First, taking variable time-steps potentially reduces the computational cost and improves accuracy of the forward and adjoint equations' numerical solution. Second, by appropriately adjusting the tolerances of the time-stepping scheme, convergence of the optimal control problem can be theoretically guaranteed via inexact Newton theory. I present proofs and computational results to support this claim.

The computational results include an extension of prior work on adaptive checkpointing schemes, enabling checkpointing when solving the reference *and* adjoint equations adaptively. The numerical results in this thesis feature an optimal control problem with a reservoir simulation constraint.

## Acknowledgments

First, and foremost, I would like to thank my advisor William Symes for being a great advisor and a great role model. His passion for research, and his general good nature, has been extremely contagious. I have sincerely enjoyed learning from him.

I am also grateful to my committee members Dan Cohan and Matthias Heinkenschloss. Dan Cohan, for being such a kind and helpful external committee member. Matthias Heinkenschloss, for his time and efforts teaching me optimization theory. Dr. Heinkenschloss has been a valuable resource, and I appreciate the sound research advice he has given me during my time at Rice.

I would like to acknowledge my Rice and Tufts professors. Their efforts have shaped the person and academic that I am today, for which I am thankful. I would like to thank these Rice University instructors: Mark Embree for his phenomenal course in Numerical Analysis, Yin Zhang for introducing me to optimization theory, Adam Singer for teaching me advanced C++ and for being a good mentor during my summer internships at ExxonMobil, Jan Hewitt for helping me become a better technical writer and a more effective speaker, Dan Sorensen for serving in my Master's committee and teaching me advanced numerical linear algebra, Richard Tapia and Steve Cox, for inspiring me to help those less fortunate than myself. I would also like to thank the following Tufts professors: Misha Kilmer, for introducing me to Applied Mathematics and getting me involved in research when I was an undergraduate and Todd Quinto, for always being a great mentor.

I would like to thank my friends, many of whom I have shared the ups and downs of graduate school with. My friends from Rice: Mili Shah, Fernando Gonzalez del Cueto, Rami Namour, Joanna Papakonstantinou, Jay Raol, Dong Sun, Xin Wang, Tony Kellems and Mona Sheikh. My good friends from Tufts: Jordan Edwards, Jeremy Scanlan, and Troy Borneman. I would also like to thank the Houston Heat Dragonboating Club, for being great friends and teammates.

The CAAM staff also deserves a lot of praise; they have made my graduate life a lot easier by providing me with help and advice when I needed it. They also made sure there was always a fresh pot of coffee brewing, which has gotten me through many early mornings and post-lunch slumps. I would, hence, like to thank the CAAM staff: Fran Moshiri, Brenda Aune, Daria Lawrence, Jennifer Treviño and (fellow food enthusiast) Ivy Gonzalez. I would also like to acknowledge the CEEE Staff: Theresa Chatman, Aaron Barelas and Linda Torres, for their help and support.

Finally, I would like to dedicate this thesis to my parents, Josefino and Sarita Enriquez, for their continued love and support. I am eternally grateful for the sacrifices they made so that I can have a better education, and a better life.

This work was partially supported by the Rice Inversion Project (TRIP), the National Science Foundation (NSF grant number: 0714193) and the NSF-Rice VIGRE Fellowship. Their support is greatly appreciated.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Literature Review</b>	<b>8</b>
2.1	Simulation-Driven Optimization Problems . . . . .	9
2.2	Adaptive Time Stepping . . . . .	15
2.3	Optimization Algorithms Using Inexact Information . . . . .	18
<b>3</b>	<b>Mathematical Background</b>	<b>25</b>
3.1	The Optimal Control Problem and The Adjoint State Method . . . . .	26
3.2	Discretization of the Optimal Control Problem . . . . .	28
3.2.1	The Adjoint State Method and Adaptive Time Stepping . . . . .	28
3.2.2	Approximate Gradient Formation . . . . .	33
3.3	Error Analysis . . . . .	33
3.3.1	Global Error Incurred in the Forward Evolution . . . . .	33
3.3.2	Error Incurred in the Adjoint Evolution and Gradient . . . . .	35
3.3.3	Objective Function Evaluation Error . . . . .	42
3.4	Adaptive Time Stepping for Optimal Control Problems . . . . .	43
3.4.1	Solving Unconstrained Optimal Control Problems with Inexact Newton . . . . .	47
3.4.2	Inexact Interior Point Methods For Constrained Optimal Con- trol Problems . . . . .	52
<b>4</b>	<b>Computational Background</b>	<b>58</b>
4.1	The Rice Vector Library (RVL) . . . . .	59
4.1.1	The Rice Vector Library (RVL) . . . . .	59
4.2	RVL and the Alg Framework . . . . .	60
4.2.1	The StateAlg Class . . . . .	61
4.2.2	The LoopAlg and terminator Classes . . . . .	62
4.2.3	The ListAlg Class . . . . .	63
4.3	The Software Framework of TSOpt . . . . .	63
4.3.1	The time Hierarchy . . . . .	64
4.3.2	The State Class . . . . .	65
4.3.3	The TimeStep Class . . . . .	66

4.3.4	The Sim Hierarchy . . . . .	67
4.3.5	The Time Terminator Hierarchy . . . . .	73
4.3.6	The jet Hierarchy . . . . .	73
4.4	TSOpt and UMin . . . . .	75
4.5	TSOpt and External Optimization Packages . . . . .	77
<b>5</b>	<b>The Black Oil Equations and the Optimal Well Rate Allocation Problem</b>	<b>78</b>
5.1	The Phase Continuity Equations . . . . .	79
5.2	Solving the Black Oil Equations . . . . .	81
5.2.1	Discretizing the Pressure Equation in Space . . . . .	82
5.2.2	Discretizing the Saturation Equation in Space . . . . .	83
5.2.3	Fixed Time Stepping for the Semi-Discretized Equations . . . . .	84
5.3	The Optimal Well-Rate Allocation Problem . . . . .	85
5.4	The Fixed Time-Step Approach for OWRA . . . . .	87
5.5	Adaptive Time-Stepping for OWRA . . . . .	88
5.5.1	Adaptive Time Stepping for the Black-Oil Equations . . . . .	88
5.5.2	Handling the Control Parameters for OWRA . . . . .	93
5.5.3	Algorithmic Development of an Adaptive Black-Oil Simulator . . . . .	94
5.6	Implementation in TSOpt . . . . .	96
5.7	Using IPOpt to Solve OWRA . . . . .	98
<b>6</b>	<b>Numerical Results</b>	<b>104</b>
6.1	Unconstrained Optimization Test . . . . .	105
6.2	Black Oil and Constrained Optimization Tests . . . . .	109
6.2.1	Tests for Fixed Time Steps . . . . .	109
6.2.2	Tests for Adaptive Time Steps . . . . .	115
<b>7</b>	<b>Future Work and Conclusion</b>	<b>119</b>
<b>A</b>	<b>Adaptive Checkpointing Algorithm</b>	<b>124</b>

# List of Figures

1.1	Numerical solution of the reference and adjoint equations corresponding to the optimal control problem (1.10), using MATLAB's <code>ode23s</code> integrator. Note that the reference and adjoint time grids are mismatched. . . . .	6
4.1	The <code>Alg</code> class and its subclasses . . . . .	61
4.2	The <code>State</code> class and its components . . . . .	65
4.3	The <code>Sim</code> class and its derived classes. . . . .	68
4.4	The <code>stackBase</code> class and its methods. . . . .	69
4.5	The <code>jet</code> class and its components. . . . .	74
5.1	Example of how interpolated values can violate the bound constraint. UB and LB represent the upper and lower bounds, respectively. The squares represent interpolation nodes, which satisfy the bound constraints. . . . .	94
6.1	Objective function plot, for both the fixed and adaptive tolerance schemes. Note the stagnation produced by the fixed-tolerance scheme. . . . .	107
6.2	Gradient-norm plot, for both the fixed and adaptive tolerance schemes. Again, note the stagnation produced by the fixed-tolerance scheme. . . . .	108
6.3	Tolerance values, for both the fixed and adaptive tolerance schemes. Note that the y-axis is on a logarithmic scale. . . . .	108
6.4	[l] Porosity and permeability plot of the SPE10 model, top layer; [r] Placement of injector (I) and producer (P) wells in the domain. . . . .	110
6.5	Plot of Aqueous Saturation at $t = 100$ days, with $dt = 25$ . . . . .	111
6.6	Plot of the difference between the computed gradient via the adjoint-state method, and the finite difference approximation. . . . .	112
6.7	Objective function for the fixed time-stepping approach to solving OWRA. Note that the iterations stop at 8, as the Black-Oil simulator crashes after the eighth iteration due to accumulation of numerical errors. . . . .	114
6.8	Progression of the control parameter for the first [l], third [m] and sixth [r] optimization iteration, taking fixed time steps. Note the well labels on the figure: "P" represents the producing wells and "I" represents the injecting wells. . . . .	115



6.9	Objective function for the fixed (blue) and adaptive (red) time-stepping approach to solving OWRA. The difference in the objective function value is about 2.5 percent. The adaptive simulation approach also did not crash. . . . .	116
6.10	Progression of the control parameter for the first [l], fourth [m] and eighth [r] optimization iteration, using adaptive simulations. Note the well labels on the figure: “P” represents the producing wells and “I” represents the injecting wells. . . . .	117
6.11	Plot of the value of the tolerances and NLP errors versus the major optimization iteration. Note that the y-axis is on a log-scale. . . . .	118

# Chapter 1

## Introduction

In its simplest form, an optimal control problem can be written as

$$\min_u \quad f(u) = \int_0^T K(w(t), t, u) dt \quad (1.1)$$

where the variables  $(w, u)$  solve the state equation:

$$\begin{aligned} \frac{d}{dt} w(t) - G(w(t), t, u) &= 0, & t \in [0, T] \\ w(0) &= 0. \end{aligned}$$

In the equations above,  $u \in \mathbb{R}^n$  is the control variable, the state trajectory  $w \in C^1([0, T], W)$ , for a state Hilbert space  $W$ ,  $K : W \times \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}$  is continuously partially differentiable, and  $G : W \times \mathbb{R} \times \mathbb{R}^n \rightarrow W$  is some nonlinear dynamic operator that is also continuously partially differentiable. Introducing an auxiliary variable  $z(t)$

satisfying the initial value problem:

$$\frac{d}{dt}z(t) = K(w(t), t, u), \quad z(0) = 0, \quad (1.2)$$

we may recast the problem (1.1) as

$$\min_u \quad f(u) = z(T) \quad (1.3)$$

where the variables  $(w, z, u)$  solve:

$$\begin{aligned} \frac{d}{dt}z(t) - K(w(t), t, u) &= 0, \quad z(0) = 0, \\ \frac{d}{dt}w(t) - G(w(t), t, u) &= 0, \quad w(0) = 0, \\ t &\in [0, T]. \end{aligned}$$

By simple substitution, the problem above fits into the framework of the following optimal control problem, which I consider for the remainder of this thesis:

$$\min_u \quad f(u) = J(y(T)) \quad (1.4)$$

where the variables  $(y, u)$  solve the state equation:

$$\frac{d}{dt}y(t) - H(y(t), t, u) = 0, \quad t \in [0, T] \quad (1.5)$$

$$y(0) = 0. \quad (1.6)$$

In the problem above, the state trajectory  $y \in C^1([0, T], Y)$ , for a state Hilbert space  $Y$ ,  $J : Y \rightarrow \mathbb{R}$  that is continuously differentiable, and  $H : Y \times \mathbb{R} \times \mathbb{R}^n \rightarrow Y$  is some nonlinear dynamic operator that is continuously partially differentiable. Throughout this thesis, numerical solution of the differential equation (1.5) will be referred to as a *forward simulation*. A forward simulation generates approximate solutions at different time-levels, called the (*forward*) *states*. The collection of all the forward states, in turn, will be referred to as the *state vector*.

An optimal control problem can also have explicit constraints, in which case the problem we consider becomes

$$\min_u \quad f(u) = J(y(T)) \quad (1.7)$$

$$\text{s.t.} \quad g_{lower} \leq g(y, u) \leq g_{upper} \quad (1.8)$$

$$u_{lower} \leq u \leq u_{upper}, \quad (1.9)$$

for a constraint function  $g \in C^1([0, T], Y) \times \mathbb{R}^n \rightarrow \mathbb{R}^k$ , bounding vectors  $g_{lower} \in (\mathbb{R} \cup \{-\infty\})^k$ ,  $u_{lower} \in (\mathbb{R} \cup \{-\infty\})^n$ ,  $g_{upper} \in (\mathbb{R} \cup \{\infty\})^k$ ,  $u_{upper} \in (\mathbb{R} \cup \{\infty\})^n$ . As in the case above, the variables  $(y, u)$  must solve the state equation (1.5) - (1.6).

In order to use derivative-based optimization algorithms to solve the problem (1.4) or (1.7), it is necessary to calculate the gradient of the objective function  $f$  with respect to the controls,  $u$ . A common method to calculate the gradient of the objective function is through the algorithm called the *adjoint-state method* [Lions, 1971]. I will motivate the use of the adjoint-state method in the third chapter.

Adjoint-state methods incur a cost roughly equivalent to the cost of numerically solving the differential equation (1.5) [Brouwer and Jansen, 2004, Sarma and Aziz, 2005]. Despite this cost, adjoint-state methods are efficient because they are not affected by the size of the control parameter. Adjoint-state methods involve solving a massive linear system, derived from linearizing the state equations over the simulation time range, then transposing the resulting matrix. For computational efficiency, instead of solving this large linear system directly, a back-substitution strategy is employed, resulting in a backward-in-time evolution. Due to the linearization step, the adjoint state method requires access to the simulation state history.

This dependence, however, poses a question for computational implementations of adjoint-state methods: what happens if we solve the state equations using an adaptive time-stepping algorithm? Adaptive time-stepping is a reasonable approach if the state equations have regions in time where the solution varies rapidly. It would be ideal to take larger time-steps over the regions where the solution varies slowly, and to restrict the time-step size over the regions where the solution varies rapidly. Taking adaptive steps in the forward and adjoint field, however, will cause the forward and adjoint time grids to mismatch. Since the forward and adjoint grids do not align, the adjoint evolution scheme will not have access to the appropriate forward state. This phenomena is easy to generate, as demonstrated by the following example:

$$\min_u \int_0^1 \frac{1}{2} y(t)^2 dt \tag{1.10}$$

where  $y(t)$  solves

$$\frac{dy}{dt} = \sum_{i=1}^N u_i \chi_i(t), \quad t \in [0, 1], \quad y(0) = 0, \quad (1.11)$$

where  $\chi_i$  is an indicator function for the interval  $[\frac{(i-1)}{N}, \frac{i}{N}]$ . The corresponding adjoint equation to the problem above is

$$\frac{dw}{dt} = -y(t), \quad t \in [0, 1], \quad w(1) = 0, \quad (1.12)$$

where  $w(t)$  is the adjoint trajectory. The example evolution seen in figure 1.1 uses the control

$$u = \begin{bmatrix} 0 & 0 & -1 & 1 & 0 & 0 & 1 & -1 & 0 & 0 \end{bmatrix}$$

placed upon equidistant nodes over the interval  $[0, 1]$  and linearly interpolated to provide access to a control value over the entire time interval. Using MATLAB's `ode23s` adaptive integrator, it is easy to see that the integration nodes between the reference and adjoint evolution do not align.

More importantly, how does this adaptive time-stepping approach affect the quality of the gradient, and the convergence to the solution of the optimal control problem (1.4)? Mismatched time-grids resulting from adaptive time stepping imply that during the adjoint evolution, an interpolation scheme must be employed to approximate the missing forward state. In turn, this implies that an interpolation error will be

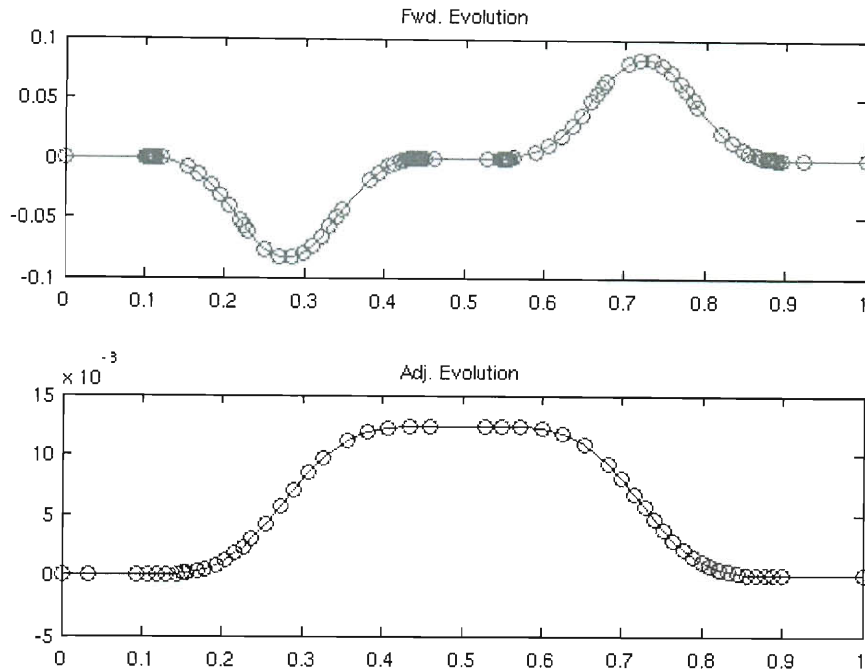


Figure 1.1: Numerical solution of the reference and adjoint equations corresponding to the optimal control problem (1.10), using MATLAB's `ode23s` integrator. Note that the reference and adjoint time grids are mismatched.

present in the adjoint state calculation. The aggregate errors from interpolation and the time-stepping algorithm manifest themselves in the gradient in a non-trivial way, and will hence affect convergence to the optimal control. However, having a controllable tolerance in the time-stepping algorithm means that the global error in the state equations' numerical solution can be changed. Is there a way to adjust time-stepping tolerances to encourage convergence to an optimal control?

In this thesis, I highlight the research I completed to answer the questions above. Chapter 2 provides a literature review of adaptive time-stepping, optimization in the presence of inexact information, and prior works to couple the two concepts. Chapter

3 provides primary analysis towards a proof of how convergence to the solution of the optimal control problem (1.4) can be guaranteed by manipulating the time-stepper’s algorithmic parameters, for the unconstrained optimal control problem. I discuss how this “adaptive tolerance” method can be applied to constrained optimal control problems as well. Chapter 4 discusses the software framework I co-developed, called `TSOpt` (“Time-Stepping for Optimization”), which is the computational tool I use to verify the theory I established. Furthermore, in chapter 4, I discuss methods to help circumvent the storage cost associated with the adjoint state method, called (Griewank) checkpointing. I also present my algorithm for “adaptive checkpointing”, which is an algorithm that can be used to checkpoint when solving the state and adjoint equations via adaptive time-stepping. Chapter 5 is dedicated to the Black-Oil equations, and how I have implemented the reference and adjoint evolution for these equations in `TSOpt`. I use this implementation to solve an explicitly-constrained optimal control problem with reservoir simulation constraints, called the “Optimal Well-Rate Allocation Problem” (OWRA). Chapter 6 presents numerical results for the Black-Oil simulator, the problem OWRA, and an unconstrained optimal control problem. I discuss future work and conclude in Chapter 7.



# Chapter 2

## Literature Review

The goal of this thesis is to explore the effect of adaptive time stepping in simulation-driven optimization problems. This chapter will review three main topics related to this goal. The first section discusses the simulation-driven optimization problem. I cover contemporary approaches to solving simulation-driven optimization problems, then introduce software packages developed to aid in solving such problems, including `TSOpt` – the software framework for the research discussed in this thesis. I then dissect the simulation-driven optimization problem into two topics: adaptive time stepping and so-called “inexact optimization methods”. In the second section, I discuss adaptive time-stepping methods, as well as software packages that implement them. In the third section, I review existing optimization methods accommodating inexact information (e.g. inexact gradients).

## 2.1 Simulation-Driven Optimization Problems

There are two main branches of strategies in solving simulation-driven optimization problems: derivative-free algorithms and derivative-based algorithms. Derivative-free algorithms are typically used for problems with an objective function that is not continuous and/or not well defined. Famous types of derivative-free algorithms are directional direct search methods (e.g., generalized pattern search, mesh-adaptive direct search) and stochastic algorithms (e.g. genetic algorithms, simulated annealing). Direct search methods use positive bases or positive spanning sets to generate descent directions in meshes or patterns [Conn et al., 2009]. Stochastic algorithms rely on the mathematical principles of randomness to update candidate solution(s). These algorithms then evaluate the objective function to gauge how “good” the candidate solutions are. Stochastic algorithms, however, suffer from the drawback of requiring many evaluations without the guarantee of monotonically decreasing objective function values. For further discussion of these strategies, see Sarma and Aziz [2005].

Derivative-based algorithms (such as Newton and its variants), as opposed to stochastic algorithms, guarantee decrease of the objective function per iteration while usually requiring fewer forward evaluations than stochastic algorithms [Renders and Flasse, 1996, Sarma and Aziz, 2005]. The major drawback of gradient-based algorithms is that for non-convex problems, convergence to the global solution is not guaranteed. In this thesis, I focus on gradient-based algorithms, since it is the only practical option for large-scale problems.

The two fundamental gradient-based strategies for solving simulation-driven optimization problems go under the names “Optimize then Discretize” (OD) and “Discretize then Optimize” (DO). OD first applies multiplier theory to the continuum problem, and then discretizes the resulting Lagrangian function. Hahn, among many others, derived explicit formulas for the continuous necessary optimality conditions for control problems [Hahn, 1996]. DO alternatively, first discretizes the continuum problem, and then solves the (discrete) optimality conditions for the resulting finite dimensional problem.

Though the OD and DO approaches eventually lead to a discretized systems of equations, they are not always equivalent. Li and Petzold [2004] demonstrate this fact by considering the following problem:

$$\min_u f(u) = \int_0^T \int_{(0,1)} g(y(x, t), u) dx dt \quad (2.1)$$

where  $u$  is a control vector and  $y$  solves the one dimensional heat equation:

$$y_t = y_{xx}, \quad (2.2)$$

with boundary conditions

$$y_x(0, t) = 0 \quad y(1, t) = 1. \quad (2.3)$$

Note that in their example, Li and Petzold solely focus on the differential equation

constraint, disregarding the contribution of the objective function on the adjoint computation. Hence, I leave the objective function in its current generic form. We use the boundary condition  $y_x(0, t) = 0$  along with the ghost boundary point  $y_0$  to deduce:

$$y_x(0, t) = \frac{y_2 - y_0}{2h} = 0 \quad (2.4)$$

Using the method of lines to solve (2.2) and using (2.4), we obtain:

$$\dot{y}_1 = \frac{2y_2 - 2y_1}{h^2} \quad (2.5)$$

$$\dot{y}_i = \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}, \quad i = 2, 3, \dots, N-1 \quad (2.6)$$

$$\dot{y}_N = 0. \quad (2.7)$$

Given the state variable  $y$  and as in Li and Petzold, ignoring the contribution of the objective function, the corresponding adjoint to this discretization takes the following form:

$$-\dot{\lambda}_1 = \frac{\lambda_2 - 2\lambda_1}{h^2} \quad (2.8)$$

$$-\dot{\lambda}_2 = \frac{2\lambda_1 - 2\lambda_2 + \lambda_3}{h^2} \quad (2.9)$$

$$-\dot{\lambda}_i = \frac{\lambda_{i+1} - 2\lambda_i + \lambda_{i-1}}{h^2}, \quad i = 3, 4, \dots, N-2 \quad (2.10)$$

$$-\dot{\lambda}_{N-1} = \frac{-2\lambda_{N-1} + \lambda_{N-2}}{h^2} \quad (2.11)$$

$$-\dot{\lambda}_N = \frac{\lambda_{N-1}}{h^2}. \quad (2.12)$$

Now consider the continuous adjoint of the objective function  $f$ . The component of this adjoint that corresponds to the heat equation constraint can be written as:

$$-\lambda_t = \lambda_{xx} \quad (2.13)$$

$$\lambda_x(0, t) = 0 \quad \lambda(1, t) = 0. \quad (2.14)$$

Applying the method of lines and a central differencing scheme to (2.13) then gives:

$$-\dot{\lambda}_1 = \frac{2\lambda_2 - 2\lambda_1}{h^2} \quad (2.15)$$

$$-\dot{\lambda}_i = \frac{\lambda_{i+1} - 2\lambda_i + \lambda_{i-1}}{h^2}, \quad i = 2, 3, \dots, N-1 \quad (2.16)$$

$$-\dot{\lambda}_N = 0. \quad (2.17)$$

Note that the partly discretized adjoint of the DO strategy does not match the discretized adjoint of the OD strategy. One should notice, however, that this discrepancy can be eliminated by performing extra manipulations. In the Li and Petzold's example, consistency can be achieved by making the adjoint variable substitution  $w_1 := \lambda_1/2$  and adding a new variable  $w_N = 0$ .

Like Petzold and Li, Hager [1999] also addressed the consistency between the OD and DO strategies. Using the continuous optimality conditions, Hager established a relationship between the continuous optimal control problem and the discretized optimal control problem. By creating a transformed adjoint system, Hager established an equivalence between the Runge-Kutta discretization of the continuous adjoint equa-

tions and the first-order necessary conditions associated with the discrete control problem. Hager exploited this equivalence to derive conditions on the elements of a Runge-Kutta scheme's Butcher table and vector that guarantee a specific order of convergence to the optimal control problem. Hager accomplishes this by extending Butcher's Runge-Kutta analysis to cater to the discretization of his transformed adjoint system. Note that Hager [1999] actually established an instance where the strategy OD is equivalent to the strategy DO.

It should also be noted that it is possible to couple both OD and DO approaches to solving the simulation-driven optimization problem. In their work Li and Petzold [2004] use a "mixed" approach to derive the discrete adjoint equations for an optimal control problem; they use the DO approach around the spatial domain boundary, then use the OD approach elsewhere in the domain. Li and Petzold claim that their approach eliminates the need to formulate proper boundary conditions for the adjoint of a general PDE, while still allowing adaptive grid refinements on the interior of the domain.

A software package that accommodates the two (non-mixed) gradient-based strategies is the FDTD, or "Finite Difference Time Domain" package [Gockenbach et al., 2002]. FDTD is a C++ software package that, given a time-stepping algorithm (and related code), creates a simulator capable of generating forward, derivative (or "sensitivity"), and adjoint states. FDTD could be used to solve optimal control problems by providing necessary data structures and functions to an optimization algorithm,

such as the Quasi-Newton algorithm BFGS, provided that such algorithms are coded in conformance with a certain system of interfaces.

TSOpt – the “Time Stepping for Optimization” Package – succeeded FDTD [Symes, 2006]. TSOpt is similar to FDTD in that they both exploit C++ object-oriented programming (OOP) to solve systems of differential equations by using time stepping methods. TSOpt, however, differs from FDTD in two fundamental ways: first, TSOpt uses C++ templating so it can accommodate multiple data types. Second, and most importantly, TSOpt is based on the Rice Vector Library (RVL), while FDTD is based on the Hilbert Class Library (HCL). HCL was RVL’s predecessor; though both represented Hilbert-Space calculus objects as C++ classes, RVL improved upon HCL by fully separating “Calculus” and “Data Storage” components [Padula et al., 2009].

TSOpt is an interface for creating simulation operators which incorporated time-stepping algorithms. It supplies interfaces needed by Newton-based algorithms to solve the optimization problem (1.4). Three such interfaces define the forward evolution operator, the adjoint evolution operator and the derivative evolution operator. The forward evolution operator yields forward-simulation state vectors. These forward states are then used by the adjoint-state evolution operator to generate adjoint states, which in turn can be used to construct the objective function’s gradient. The derivative evolution operator outputs derivative states, and can be used to obtain sensitivities. The gradient of the objective function is then used in Newton or quasi-Newton methods to solve the simulation-driven optimization problem. Of course,

how well a Newton (or Newton-based) method succeeds depends on the properties of the continuum problem.

There are various other commercial and non-commercial optimal control solvers available, such as Stanford’s General Purpose Research Simulator (GPRS). GPRS is non-commercial, C++ simulation software for solving problems pertaining to reservoir engineering and management. Sarma and Aziz [2005] used GPRS to solve an oil well related optimal control problem. Of the current software packages I examined, however, the package most similar to `TSOpt` is Sandia National Laboratory’s software package `Rythmos`. `Rythmos` is a “transient integrator” of differential equations that uses time-stepping algorithms implemented in C++. `Rythmos` is similar to `TSOpt` because it also uses advanced C++ coding techniques, such as templating and class hierarchies, to create inter-operating components to solve differential equations [Coffey, 2009]. Currently, `Rythmos` is “aimed at supporting operator-split algorithms, multi-physics applications, block linear algebra and adjoint integration”. Given `Rythmos`’ current documentation, it is difficult to discuss the existence of various features, such as support for gradient calculations via the adjoint-state method or checkpointing.

## 2.2 Adaptive Time Stepping

In simulation-driven optimization problems, the differential equation constraint (1.5) is typically solved numerically by performing fixed-step time-stepping routines. This



could, however, be problematic when one or more regions of the differential equation's solution varies quickly; in order to maintain accuracy of the solution, small time steps must be used. This, in turn, leads to taking more time steps – increasing computational expense.

As an alternative to performing fixed time-steps on a fine time grid, we can instead use adaptive time-stepping algorithms. Adaptive time stepping methods allow the step lengths to vary while performing the evolution. Over the time windows where the solution is changing rapidly, the algorithm can restrict the step length while in the time windows where the solution changes slowly, the algorithm can take larger time steps [Lambert, 2000, Süli and Mayers, 2003, Kincaid and Cheney, 2002]. It should be noted that both explicit and implicit schemes can be adaptive.

Implicit methods have large stability regions, allowing bigger time steps to be taken. In exchange for the large stability region, however, an extra system of equations must be solved at every iteration. Hence, implicit methods are generally more difficult to implement [Lambert, 2000, Süli and Mayers, 2003, Kincaid and Cheney, 2002]. Despite its extra computational and implementation cost, implicit methods are preferred over explicit methods for solving stiff differential equations, since it often takes less time to simulate using an implicit method with a large, fixed time step (compared to an explicit method with an excessively small fixed time step). Some examples of implicit methods range from the common backward Euler scheme, to more complex  $k$ -step Backward Differentiation Formulae (BDF) schemes [Lambert,

2000].

Embedded explicit Runge-Kutta (RK) methods are a popular example of an adaptive time stepping algorithm [Lambert, 2000, Süli and Mayers, 2003, Kincaid and Cheney, 2002]. These methods yield a local (truncation) error estimate at every step, which can be used to alter the step length size. If the local error estimate is greater than a user defined tolerance, then the step is rejected; the step length is reduced and another forward step is attempted. This process is repeated until the local error estimate is less than the given tolerance. On the other hand, if the error estimate is significantly lower than the given tolerance, the step length can be increased [Lambert, 2000, Süli and Mayers, 2003, Kincaid and Cheney, 2002].

Multi-step algorithms (as opposed to one-step algorithms, such as Runge-Kutta) – both in explicit or implicit form – can also be used to perform adaptive time steps. Lambert [2000] describes methods referred to as *variable step, variable order* (VSVO) algorithms, such as predictor-corrector Adams methods. Popular VSVO algorithms include DIFSUB (Gear), GEAR (Hindmarsh) and EPISODE (Byrne and Hindmarsh) [Lambert, 2000, Jackson and Sacks-Davis, 1980]. Jackson and Sacks-Davis [1980] implement a variable step-size multi-step formula, which leads to efficient solution of the system of equations arising from taking an implicit time step.

Many non-commercial time stepping software packages exist. Besides the algorithms mentioned above, there are also the software packages GSL, RKSuite\_90 and ODEPACK. The GNU Scientific Library (GSL) [Galassi and Theiler, 2009] in-

cludes a time-stepping framework for solving ordinary differential equations which include adaptive time-stepping algorithms such as RKF45. Brankin et al. developed `RKSuite_90`, a collection of Runge-Kutta schemes implemented in Fortran [Brankin et al., 1993]. The Lawrence Livermore National Laboratory developed `ODEPACK`, a collection of initial value ODE solvers [Hindmarsh, 1983].

## 2.3 Optimization Algorithms Using Inexact Information

Through use of adaptive time stepping, we maintain accuracy of the numerical solution to the differential equation without resorting to excessively small, fixed time steps. However, there is a tradeoff: the time grids of the reference and adjoint simulation will no longer align, which is problematic for the adjoint state method. When performing adjoint simulation, one must interpolate the forward states in order to generate an approximation at the current time level of the adjoint simulation. This introduces an extra (interpolation) error in the adjoint states, which manifests itself into more inexactness of the numerical gradient. What can we expect from optimization algorithms when given inexact information, such as the gradient? This section reviews the previous works that attempt to answer this question.

Dembo and Steihaug [1982] used the Newton method to solve the problem  $F(x) = 0$  (with  $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ ). Newton’s method is defined by the following numerical

scheme:  $x_{k+1} = x_k + s_k$ , where  $s_k$  is the solution to the Newton linear system  $F'(x_k)s_k = -F(x_k)$ . Dembo argues that for large enough systems, performing Gaussian elimination at every iteration can be prohibitively expensive. This leads to the idea of coupling Newton with an iterative method to solve the Newton linear system, which Dembo refers to as *Newton-iterative methods*.

Dembo answers the following question in his work: how accurately must we solve the Newton linear system in order to maintain the convergence properties of Newton? Defining the residual at the  $k^{\text{th}}$  iteration as  $r_k = F'(x_k)s_k + F(x_k)$ , Dembo considers the class of Newton methods (called *inexact Newton methods*) which iteratively solve the Newton linear system while satisfying the following bound:

$$\frac{\|r_k\|}{\|F(x_k)\|} \leq \mu_k, \quad (2.18)$$

for some non-negative sequence  $\{\mu_k\}$  (called the *forcing sequence*). Dembo's main results states that if  $\mu < 1$  exists, such that  $\mu_k < \mu$  for all  $k$ , then the inexact Newton method is locally convergent. Globalization of the inexact Newton algorithm is typically accomplished via linesearch. Different strategies for the linesearch are discussed in [Eisenstat and Walker, 1994a].

Further analysis of the Newton algorithm using inexact information can be found in [Kelley and Sachs, 1999]. Motivated by optimal control problems, Kelley and Sachs

examine the unconstrained optimization problem

$$\min_x f(x),$$

for a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , whose objective function evaluation and gradients are given by “black-box” codes and whose absolute and relative error are controllable. Kelley and Sachs also use Newton-iterative methods, but they do so in the context of linear systems arising from the Conjugate-Gradient Trust Region algorithm (CGTR). Kelley and Sachs [1999] relate the controllable error parameter to how the forcing sequence should be chosen to guarantee correct behavior of the inexact Newton iteration. Further, Kelley and Sachs make algorithmic modifications so that the CGTR behaves like the error-free algorithm while  $\|\nabla f\|$  is much greater than the absolute error of gradient.

Like Kelley and Sachs [1999], Carter [1991] also uses an unconstrained optimization algorithm. Carter also considers solving  $\min f(x)$ , where  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  by using the Trust-Region (TR) algorithm, though he does not use inexact Newton methods. The TR update takes the form  $x_{k+1} = x_k + s_k$ , where  $s_k$  solves the *Trust Region* subproblem:

$$\min_s \quad \psi(x_k + s) \tag{2.19}$$

$$s.t. \quad \|D_k s\| \leq \Delta_k. \tag{2.20}$$

In the subproblem above,  $\psi_k(x_k + s) = f(x_k) + g_k^T s + \frac{1}{2} s^T H_k s$ , with  $g_k$  is the approximate evaluation of  $\nabla f$  at  $x_k$  and  $H_k$  is the approximate evaluation of  $\nabla^2 f_k$  at  $x_k$ . The matrix  $D_k$  is a positive definite preconditioning matrix, which may be taken as the identity.

Carter asserts that a suitably modified TR algorithm converges globally to a stationary point provided that

$$\frac{\|g_k - \nabla f(x_k)\|_{(D_k^T D_k)^{-1}}}{\|g_k\|_{(D_k^T D_k)^{-1}}} \leq \xi, \quad (2.21)$$

for some  $\xi \in [0, (1 - \eta)]$ , where  $0 < \eta < 1$  is a user-chosen parameter. (Here,  $\|x\|_A \equiv (x^T A x)^{\frac{1}{2}}$  for  $A \in \mathbb{R}^{n \times n}$  symmetric positive definite.) It is worth noting that, like Dembo in (2.18), Carter in (2.21) imposed a bound on the relative error from their algorithm. Carter asserts that if (2.21) is satisfied, then we have

$$\lim_{k \rightarrow \infty} \|\nabla f(x_k)\| = 0.$$

(Note that we are no longer considering a norm weighed by the matrix  $(D_k^T D_k)^{-1}$ .)

We can understand this assertion by demonstrating that, by enforcing Carter's bound, the approximated gradient will always be in the direction of the true gradient. First, note that the rate of change of  $f$  in the direction  $g_k$  at the point  $x_k$  can be expressed as  $\nabla f^T g_k$ . Hence, we must show  $\nabla f^T g_k > 0$ . We begin by introducing zeros to the

inner product, and simplifying:

$$\nabla f^T g_k = (\nabla f - g_k + g_k)^T g_k = (\nabla f - g_k)^T g_k + \|g_k\|^2.$$

Using the Cauchy-Schwarz inequality yields

$$(\nabla f - g_k)^T g_k + \|g_k\|^2 \geq -\|\nabla f - g_k\| \|g_k\| + \|g_k\|^2.$$

Then using Carter's bound, we arrive at

$$-\|\nabla f - g_k\| \|g_k\| + \|g_k\|^2 \geq -(1 - \eta) \|g_k\|^2 + \|g_k\|^2 = \eta \|g_k\|^2 > 0.$$

Carter's TR algorithm works for a subclass of problems with the following traits: first, there must be a computable error bound for each gradient approximation  $g_k$ . Second, solution accuracy must be controllable either directly (by specifying algorithm tolerances), or indirectly (for example, by refining grids). Carter's TR algorithm, however, suffers from a fundamental problem: it is usually difficult to obtain a computable error bound on the gradient error. Hence, it is hard (even impossible in some cases) to verify if (2.21) is satisfied at each optimization iteration.

Other authors have considered the effect of inexact gradients on the trust region algorithm for unconstrained optimization problems. Moré [1982] establishes convergence results for a modified trust region algorithm that uses scaling and preconditioning in solving the TR subproblem, assuming that the approximated gradient

$g_k$  satisfies the following:

$$\lim_{k \rightarrow \infty} \|g_k - \nabla f(x_k)\| = 0, \quad (2.22)$$

given a sequence  $\{x_k\}$  that converges to a stationary point. The same result can also be found in Conn et al. [2000], who give a more detailed discussion on the global convergence of the TR algorithm using approximated gradients, under various assumptions on the algorithm and the problem.

In contrast to the previous authors, Heinkenschloss and Vicente [2001] considers nonlinear, constrained optimization problems of the form,

$$\min \quad f(y, u) \quad (2.23)$$

$$\text{s.t.} \quad C(y, u) = 0, \quad (2.24)$$

for  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ ,  $C : \mathbb{R}^n \rightarrow \mathbb{R}^m$ , the state variable  $y \in \mathbb{R}^m$ , and the control variable  $u \in \mathbb{R}^{n-m}$ . Heinkenschloss and Vicente solve the problem above using a modified Trust-Region SQP method which allows for inexactness in the gradient caused by inexact linear system solves. Under the bound they propose for the gradient error, they prove the first-order global convergence of their algorithm. It is also worth noting that for the reduced unconstrained problem

$$\min_u f(y(u), u), \quad (2.25)$$



where  $(y(u), u)$  solves the constraint equation  $C(y, u) = 0$ , if the approximated gradient satisfies the gradient error bound in Heinkenschloss and Vicente [2001], global *lim inf* convergence of the Moré’s TR algorithm [Moré, 1982] can also be shown.

Similar to Heinkenschloss and Vicente [2001], Bellavia [1998] considered a problem of the form (2.23), with the addition of inequality constraints. However, in contrast to Heinkenschloss and Vicente, Bellavia considered the interior-point method to solve (2.23). Interior-point methods generate search directions for the constrained optimization problem by applying Newton’s method to the first-order necessary (KKT) conditions associated with (2.23). Instead of using the standard Newton algorithm, however, Bellavia used an *inexact* Newton algorithm – establishing the “inexact interior-point” method. Given a specific choice for the forcing sequence, and a line-search globalization scheme, Bellavia demonstrated that convergence to a stationary point can be achieved by using inexact interior-point algorithms. Like Bellavia, Wächter [2002] has also considered coupling inexact linear system solves with the interior point method. Wächter’s interior-point algorithm, `IPOpt`, is an interior-point algorithm with the option to use iterative linear system solves. (Wächter [2009] notes, however that this feature is currently in the developmental phase.) Though I do not use Bellavia’s algorithm in this thesis, I build upon the general idea of using inexact Newton to generate a search direction for the interior point subproblem. I also use inexact Newton methods for unconstrained optimal control problems.

# Chapter 3

## Mathematical Background

In this chapter I discuss the mathematical background necessary for my thesis work. I begin by introducing the adjoint-state method and the optimal control problem. I then focus on the optimal control problem's differential equation constraints and the differential equations needed for the adjoint-state method: the linearized and adjoint equations. Next, I discuss how to numerically solve reference, linearized and adjoint equations via adaptive time-stepping. I then derive global error bounds for the adaptive adjoint evolution, which I then use to establish a gradient error bound.

I couple the gradient error analysis to optimization theory in the second half of this chapter. In this dissertation, I consider optimization methods based on the inexact Newton method. This half of the chapter, hence, begins with a discussion on inexact Newton theory. I then explain how to theoretically couple adaptive-time stepping and optimal control problems by using inexact Newton methods, for problems with and without explicit constraints. In the case that the optimal control problem has no ex-

explicit constraints, there are no necessary modifications to the inexact Newton method. I show that in order to guarantee convergence to a local solution, the adaptive time-stepping tolerance must be coupled to the norm of the objective function's gradient. In the case that there are explicit constraints, I use inexact interior-point methods. Inexact interior point methods couple interior point theory to inexact Newton methods. I show that for a specific barrier subproblem, convergence to a local solution can be theoretically attained by coupling the tolerance of the adaptive time-stepping algorithm to the optimization (specifically: NLP) error. I conclude the chapter by conjecturing how this fact aids to the overall solution of the NLP.

### 3.1 The Optimal Control Problem and The Adjoint State Method

I begin by defining the optimal control problem considered in this thesis, which takes the following form:

$$\min_u \quad f(u) = J(y(T)) \tag{3.1}$$

$$\text{s.t.} \quad \frac{d}{dt}y(t) - H(y(t), t, u) = 0, \quad t \in [0, T] \tag{3.2}$$

$$y(0) = 0, \tag{3.3}$$

where the control  $u \in \mathbb{R}^n$ , the state trajectory  $y \in C^1([0, T], Y)$ , for a state Hilbert space  $Y$ ,  $J : Y \rightarrow \mathbb{R}$  is continuously differentiable, and  $H : Y \times \mathbb{R} \times \mathbb{R}^n \rightarrow Y$  is some

nonlinear dynamic operator that is continuously partially differentiable. Further, I present the following standard assumptions regarding the function  $J$  and  $H$ , which will be used for error analysis, and is required to guarantee the existence of a unique  $C^1$  solution to the differential equation constraint [Lambert, 2000, 5]:

**Assumption 3.1.0.1.**  $J, \nabla J, H, D_y H$  and  $D_u H$  are Lipschitz continuous. In other words, for all  $a, b \in Y$ , for all  $t \in [0, T]$  and for a fixed control  $\bar{u} \in \mathbb{R}^n$ , the following inequalities hold:

$$|J(a) - J(b)| \leq L_J \|a - b\| \quad (3.4)$$

$$\|\nabla J(a) - \nabla J(b)\| \leq L_{\nabla J} \|a - b\| \quad (3.5)$$

$$\|H(a, t, \bar{u}) - H(b, t, \bar{u})\| \leq L_H \|a - b\| \quad (3.6)$$

$$\|D_y H(a, t, \bar{u}) - D_y H(b, t, \bar{u})\| \leq L_{H_y} \|a - b\| \quad (3.7)$$

$$\|D_u H(a, t, \bar{u}) - D_u H(b, t, \bar{u})\| \leq L_{H_u} \|a - b\|, \quad (3.8)$$

where  $L_J, L_{\nabla J}, L_H, L_{H_y}, L_{H_u} > 0$  are the corresponding Lipschitz constants.

The formula for the gradient  $\nabla f(u) \in \mathbb{R}^n$  has been derived in [Kelley and Sachs, 1999, Hager, 1999], and can be written as the following:

$$\nabla f(u) = \int_0^T D_u H(y(t), t, u)^* \lambda(t) dt, \quad (3.9)$$

where  $\lambda \in C^1([0, T], Y)$  is called the *adjoint variable*, satisfying the final-value prob-

lem on  $[0, T]$

$$-\frac{d\lambda(t)}{dt} = D_y H(y(t), t, u)^* \lambda(t), \quad t \in [0, T] \quad (3.10)$$

$$\lambda(T) = \nabla J(y(T)). \quad (3.11)$$

This method for obtaining the gradient is called “the adjoint-state method”. Numerically, the adjoint-state method is attractive because its computational cost is often independent of the size of the control variable  $u$  [Plessix, 2006].

## 3.2 Discretization of the Optimal Control Problem

After describing the optimal control problem and presenting the continuous formulas for both the adjoint equations and the gradient, I now describe discretization. I first discuss how I solve the state and adjoint equations numerically. Then, I describe how I discretize the objective function. I end this section by outlining how I compute the approximate gradient.

### 3.2.1 The Adjoint State Method and Adaptive Time Stepping

I now introduce three differential equations that relate to the adjoint-state method: the reference (forward) equations, the linearized and the adjoint equations. I also discuss how these differential equations are solved numerically. Recall that the dif-

ferential equation constraint of the optimal control problem we consider:

$$\frac{dy(t)}{dt} = H(y(t), t, u), \quad t \in [0, T] \quad (3.12)$$

$$y(0) = 0. \quad (3.13)$$

Together, (3.12) - (3.13) are referred to as the “reference” or “forward” equations.

Using a one-step scheme (e.g., Forward Euler or Runge-Kutta) to numerically solve (3.12) - (3.13) yields the update:

$$y_{(s_{n+1})} = y_{(s_n)} + h_n^{(f)} \bar{H}(y_{(s_n)}, s_n, u), \quad n = 0, 1 \dots, (N_{(f)} - 1), \quad (3.14)$$

$$y_{(0)} = 0. \quad (3.15)$$

where  $\bar{H}$  is an operator hiding the one-step scheme being used. Further, I present the following necessary assumption on  $\bar{H}$ :

**Assumption 3.2.1.1.**  $\bar{H}$  is a Lipschitz continuous function. In other words, for all  $a, b \in Y$ , for all  $t \in [0, T]$ , for a fixed control  $\bar{u} \in \mathbb{R}^n$ :

$$\|\bar{H}(a, t, \bar{u}) - \bar{H}(b, t, \bar{u})\| \leq L_{\bar{H}} \|a - b\| \quad (3.16)$$

where  $L_{\bar{H}} > 0$  is the corresponding Lipschitz constant.

Note that in (3.14),  $s_n = \sum_{j=1}^n h_j^{(f)}$ . Also, note that it is possible to “hide” a multi-step method in the one-step scheme (3.14); it has been noted in [Kirchgraber, 1985]

that multi-step methods are essentially one-step methods. If the constraint equation's solution changes rapidly in some time regions, it would be advantageous to use adaptive time stepping to numerically compute the solution, implying that we allow  $\{h_n^{(f)}\}$  to be non-uniform.

It is also necessary to define the linearized equations (also referred to as the “sensitivity equations”), which stems from the first term of the multi-parameter, first order Taylor expansion of  $H$ :

$$0 = \frac{d\delta y(t)}{dt} - D_y H(y(t), t, u)\delta y(t) - D_u H(y(t), t, u)\delta u \quad (3.17)$$

$$\delta y(0) = 0 \quad (3.18)$$

In (3.17),  $\delta y(t)$  refers to the state perturbation and  $\delta u$  refers to the control perturbation. In the case of fixed time-steps, the solution to the sensitivity equation  $\delta y$  can be used to verify the output of the adjoint evolution via the so-called “dot-product test”. The sensitivity equations can be solved discretely by performing the following update:

$$\delta y_{(r_{n+1})} = \delta y_{(r_n)} + h_n^{(d)}[\bar{H}_y(\hat{y}_{(r_n)}, r_n, u)\delta y_{(r_n)} + \bar{H}_u(\hat{y}_{(r_n)}, r_n, u)\delta u], \quad (3.19)$$

$$n = 0, \dots, (N_{(d)} - 1) \quad , \quad (3.20)$$

$$\delta y_{(0)} = 0. \quad (3.21)$$

Note that  $\bar{H}_y$  and  $\bar{H}_u$  are the one-step schemes being used for the linearized evolution,

and should be defined so that the evolution above is consistent with (3.17). Further, I assume the following:

**Assumption 3.2.1.2.**  $\bar{H}_y$  and  $\bar{H}_u$  are Lipschitz continuous functions. In other words, for all  $a, b \in Y$ , for all  $t \in [0, T]$ , for a fixed control  $\bar{u} \in \mathbb{R}^n$ :

$$\|\bar{H}_y(a, t, \bar{u}) - \bar{H}_y(b, t, \bar{u})\| \leq L_{\bar{H}_y} \|a - b\| \quad (3.22)$$

$$\|\bar{H}_u(a, t, \bar{u}) - \bar{H}_u(b, t, \bar{u})\| \leq L_{\bar{H}_u} \|a - b\| \quad (3.23)$$

where  $L_{\bar{H}_y}, L_{\bar{H}_u} > 0$  are the corresponding Lipschitz constants.

The scheme (3.19) generates the linearized state vector  $\{\delta y_{(r_n)}\}$ , defined on the time grid  $\{r_n\}$ . The reference simulation states (defined on the time grid  $\{s_j\}$ ) need to be interpolated to align with the sensitivity time grid. Hence, in (3.19), the term  $\hat{y}_{(r_n)}$  denote an interpolated reference state value at time  $r_n$ .

Having defined the linearized evolution, we may now proceed to the adjoint evolution, which yields the adjoint states needed by the adjoint state method to construct the gradient of the objective function. The “backward in time” adjoint evolution can be written as

$$0 = \frac{d\lambda(t)}{dt} + (D_y H(y(t), t, u))^* \lambda(t) \quad (3.24)$$

$$\lambda(T) = \nabla J(y(T)), \quad (3.25)$$

where the adjoint variable  $\lambda \in C^1([0, T], Y)$ . The corresponding discrete adjoint



evolution can then be written as:

$$\lambda_{(t_n)} = \lambda_{(t_{n+1})} + h_{n+1}^{(a)} ((\bar{H}_y^a[\hat{y}_{(t_n)}, t_n, u])\lambda_{(t_{n+1})}), \quad (3.26)$$

$$n = (N_{(a)} - 1), \dots, 1, 0 \quad (3.27)$$

$$\lambda_{(T)} = \nabla J(\hat{y}_{(T)}), \quad (3.28)$$

where  $\bar{H}_y^a$  is a one-step scheme, defined so that the evolution above is consistent with (3.24). I now introduce a Lipschitz assumption on two discrete operators  $\bar{H}_y^a$  and  $\bar{H}_u^a$  (the analogue of the operator  $D_u H^*$ ), which will be used for error analysis:

**Assumption 3.2.1.3.**  $\bar{H}_y^a$  and  $\bar{H}_u^a$  are Lipschitz continuous functions. In other words, for all  $a, b \in Y$ , for all  $t \in [0, T]$ , for a fixed control  $\bar{u} \in \mathbb{R}^n$ :

$$\|\bar{H}_y^a(a, t, \bar{u}) - \bar{H}_y^a(b, t, \bar{u})\| \leq L_{\bar{H}_y^a} \|a - b\| \quad (3.29)$$

$$\|\bar{H}_u^a(a, t, \bar{u}) - \bar{H}_u^a(b, t, \bar{u})\| \leq L_{\bar{H}_u^a} \|a - b\| \quad (3.30)$$

where  $L_{\bar{H}_y^a}, L_{\bar{H}_u^a} > 0$  are the corresponding Lipschitz constants.

The scheme (3.26) generates the adjoint state vector  $\{\lambda_{(t_n)}\}$ . As with the linearized evolution, note that the  $\hat{y}_{(t_n)}$  here denotes an interpolated reference simulation state, so that the reference state aligns with the adjoint time grid.

### 3.2.2 Approximate Gradient Formation

I now describe how I compute the approximate gradient  $\hat{g}$  in this thesis, which is presented in Algorithm 1. There are a few things to note. First, note step (c2) highlights two possible strategies to interpolate the reference states. We could either use polynomial interpolation (strategy i1), or use a saved simulation state as a starting point for re-simulation, making the required time  $t^-$  the simulation stopping point (strategy i2). While the former strategy is a natural choice, I also consider the latter as it grants more definitive error bounds. (Namely, the “interpolation” error becomes equivalent to the time-stepping truncation error, defined in the next section.) Regardless of the strategy employed, the interpolated reference state will be denoted as  $\hat{y}_{(t^-)}$ . Finally note that I will refer to step c3 as the “AG evolution” for the remainder of this chapter. (Notice the definition of the evolution operator  $\Phi$ .) Consequently,  $\Lambda_{(t)}$  will be referred to as an “AG state”.

## 3.3 Error Analysis

### 3.3.1 Global Error Incurred in the Forward Evolution

I now present global error bound for the forward evolution, which is a consequence of the following standard theorem regarding one-step methods [Süli and Mayers, 2003, 317-318].

**Theorem 3.3.1.1.** *Suppose Assumption 3.2.1.1 is satisfied. It then follows that the*

---

**Algorithm 1:** Algorithm for computing the approximate gradient  $\hat{g}$ 


---

Given a control  $u \in \mathbb{R}^n$ , adaptive time-stepping tolerance  $\tau$

a) Generate the forward states  $\{y_{(s_n)}\}$  via (3.14).

b) Begin adjoint evolution. Let  $t = T$ ,  $k = 0$ . Define  $h_0^{(a)}$ .

**while** ( $t > 0$ )

c1) Define  $t^- = t - h_k^{(a)}$

c2) Generate the reference state needed by the adjoint evolution  $\hat{y}_{(t^-)}$  via

i1) Polynomial interpolation, using saved reference states as nodes, or

i2) Re-simulation, the *initial* reference state as a starting point.

c3) Take adjoint and gradient accumulation step:

$$\underbrace{\begin{bmatrix} \lambda_{(t^-)} \\ \hat{g} \end{bmatrix}}_{=\Lambda_{(t^-)}} = \underbrace{\begin{bmatrix} \lambda_{(t)} \\ \hat{g} \end{bmatrix}}_{=\Lambda_{(t)}} + h_k^{(a)} \underbrace{\begin{bmatrix} \bar{H}_y^a(\hat{y}_{(t^-)}, t^-, u) \lambda_{(t)} \\ \bar{H}_u^a(\hat{y}_{(t^-)}, t^-, u) \lambda_{(t)} \end{bmatrix}}_{=\Phi(\hat{y}_{(t^-)}, t^-, u, \Lambda_{(t)})} \quad (3.31)$$

c4) Determine new steplength  $h_{k+1}^{(a)}$ , and let  $t = t^-$ .

c5) Set  $k = k + 1$ .

**end while**

---

global error  $\|e_n^{(f)}\| = \|y(t_n) - y_{(t_n)}\|$  satisfies

$$\|e_n^{(f)}\| \leq \frac{E}{L_H} (e^{L_H(t_n)} - 1), \quad (3.32)$$

where  $L_H$  is the Lipschitz constant associated with the dynamic operator  $H$  and

$$E = \max_k \|E_{(t_k)}\|,$$

where  $E_{(t_k)}$  is called the truncation error, defined as

$$E_{(t_k)} = \frac{y(t_{k+1}) - y(t_k)}{h_k} - \bar{H}(y(t_k), t_k, \bar{u}), \quad (3.33)$$

given a fixed control  $\bar{u} \in \mathbb{R}^n$ .

Since adaptive time-stepping guarantees that, given a tolerance  $\gamma$ , the truncation error made in each time step is  $O(\gamma)$  [Lambert, 2000], the following is a natural corollary to the theorem above.

**Corollary 3.3.1.1.** *Given the problem (and associated assumptions) from Theorem 3.3.1.1, the global error  $\|e_n^{(f)}\|$  incurred in the forward evolution when performing adaptive time-stepping (with a tolerance  $\gamma$ ) satisfies the following bound*

$$\|e_n^{(f)}\| \leq C^{(f)}\gamma, \quad (3.34)$$

for some constant  $C^{(f)} > 0$ .

### 3.3.2 Error Incurred in the Adjoint Evolution and Gradient

Recall that we are using adaptive time stepping schemes to solve the discretized forward, derivative and adjoint evolution problems, (3.14), (3.19) and (3.26). Using adaptive time stepping, however, presents a dilemma: the adjoint evolution requires access to the forward states, implying that the forward and AG time grids must match. If we use adaptive time stepping schemes, however, we are no longer guaranteed that the forward and AG grids will align. We must therefore interpolate the forward states to provide an approximation that aligns with the adjoint grid. (Recall from Algorithm 1 that these interpolated states are denoted as  $\{\hat{y}_{(t_k)}\}$ .) Doing so, however,

will introduce an interpolation error. I now discuss this interpolation error, and how it affects the error of the AG evolution.

**Theorem 3.3.2.1.** *Suppose the Assumption 3.1.0.1 is satisfied. Further, assume the following. First, both the forward and the adjoint problems are being solved by the same adaptive, one-step time stepping scheme. Second, whenever the adjoint evolution requires a forward state at time  $t$  that does not exist in the forward (time) grid, we “interpolate by resimulation” (Algorithm 1, strategy i2) to get an approximate reference state at time  $t$ .*

*Then, the error in the AG state  $e_k^{(a)} = \Lambda(t_k) - \Lambda_{(t_k)}$  satisfies the following error bound:*

$$\|e_k^{(a)}\| \leq C_1\gamma + C_2\xi, \quad (3.35)$$

*where the adaptive tolerance for the reference and adjoint time-steppers are  $\gamma$  and  $\xi$ , respectively, and the constants  $C_1, C_2 > 0$ .*

*Proof.* I now recall the notation introduced in the previous section. I denote the exact value of the forward states and AG states as  $\{y(s_j)\}$  and  $\{\Lambda(t_k)\}$ , respectively, for an increasing time sequence  $\{s_j\}_{j=0}^N$  and a decreasing time sequence  $\{t_k\}_{k=0}^M$  defined such that  $s_0 = t_M = T$  and  $s_N = t_0 = 0$ . ( $M, N$  are natural numbers not known a-priori, though are known to be finite.) The corresponding forward states and AG states computed via adaptive time stepping is written as  $\{y_{(s_j)}\}$  and  $\{\Lambda_{(t_k)}\}$ . The approximated forward state at time  $t_j$  (obtained via interpolation) is then written as

$\hat{y}_{(t_j)}$ . I will also define the global error for the AG evolution (and partial gradient error) as

$$e_k^{(a)} = \Lambda(t_k) - \Lambda_{(t_k)}. \quad (3.36)$$

I begin by defining the *truncation error* for an adaptive, one-step time stepping scheme as

$$E_{(t_k)} = \frac{\Lambda(t_{k+1}) - \Lambda(t_k)}{h_k} - \Phi(u, t_k, y(t_k), \Lambda(t_k)), \quad (3.37)$$

for the Lipschitz continuous function  $\Phi(u, t, y, \Lambda)$  defined in algorithm 1. I then rearrange (3.37) as

$$\Lambda(t_{k+1}) = \Lambda(t_k) + h_k \Phi(u, t_k, y(t_k), \Lambda(t_k)) + h_k E_{(t_k)}. \quad (3.38)$$

(Since we are analyzing the AG evolution, let  $h_k = h_k^{(a)}$ .) By subtracting (3.38) from the form of the AG evolution that uses interpolated forward states

$$\Lambda_{(t_{k+1})} = \Lambda_{(t_k)} + h_k \Phi(u, t_k, \hat{y}_{(t_k)}, \Lambda_{(t_k)}), \quad (3.39)$$

we arrive at:

$$\begin{aligned} \Lambda(t_{k+1}) - \Lambda(t_k) &= \\ &= \Lambda(t_k) - \Lambda(t_k) + h_k(\Phi(u, t_k, y(t_k), \Lambda(t_k))) - \Phi(u, t_k, \hat{y}(t_k), \Lambda(t_k)) + h_k E(t_k). \end{aligned}$$

Using the definition of global error (3.36) yields

$$e_{k+1}^{(a)} = e_k^{(a)} + h_k(\Phi(u, t_k, y(t_k), \Lambda(t_k))) - \Phi(u, t_k, \hat{y}(t_k), \Lambda(t_k)) + h_k E(t_k). \quad (3.40)$$

Since  $\Phi$  was assumed to be Lipschitz continuous, we can then derive the following inequalities:

$$\begin{aligned} \|e_{k+1}^{(a)}\| &= \|e_k^{(a)} + h_k(\Phi(u, t_k, y(t_k), \Lambda(t_k))) - \Phi(u, t_k, \hat{y}(t_k), \Lambda(t_k)) + h_k E(t_k)\| \\ &\leq \|e_k^{(a)}\| + |h_k| \|\Phi(u, t_k, y(t_k), \Lambda(t_k)) - \Phi(u, t_k, \hat{y}(t_k), \Lambda(t_k))\| + |h_k| \|E(t_k)\| \\ &\leq \|e_k^{(a)}\| + |h_k| L_\Phi \sqrt{\|y(t_k) - \hat{y}(t_k)\|^2 + \|\Lambda(t_k) - \Lambda(t_k)\|^2} + |h_k| \|E(t_k)\| \end{aligned} \quad (3.41)$$

$$= \|e_k^{(a)}\| + |h_k| L_\Phi \sqrt{\|y(t_k) - \hat{y}(t_k)\|^2 + \|e_k^{(a)}\|^2} + |h_k| \|E(t_k)\|, \quad (3.42)$$

for the Lipschitz constant  $L_\Phi$ . (Note that the subscript denotes this constant's dependence on the function  $\Phi$ .)

I will now introduce more notation to further simplify our error bound (3.42). First, let the adaptive time stepping scheme's tolerance for the forward evolution and AG evolution be denoted as  $\gamma$  and  $\xi$ , respectively, for positive real numbers  $\gamma$  and

$\xi$ . By definition of the tolerances, the forward and AG evolutions progress so that for all  $j, k$ , the forward truncation error  $\|E_j^{(f)}\| \leq C^{(f)}\gamma$  and the AG truncation error  $\|E_{(t_k)}^{(a)}\| \leq C^{(a)}\xi$ , for constants  $C^{(f)}, C^{(a)} > 0$ . I now define

$$C_\gamma := \max_{k=0, \dots, M} \|y(t_k) - \hat{y}_{(t_k)}\|.$$

By Corollary 3.3.1, we assert that  $C_\gamma \leq C^{(f)}\gamma$ , for some  $C^{(f)} > 0$ . We can then further bound (3.42):

$$\|e_{k+1}^{(a)}\| \leq \|e_k^{(a)}\| + h_k L_\Phi \sqrt{(C^{(f)}\gamma)^2 + \|e_k^{(a)}\|^2} + h_k C^{(a)}\xi. \quad (3.43)$$

In order to make the equation above easier to work with, we note that for all real numbers  $a, b \geq 0$ ,  $(a + b) \geq \sqrt{a^2 + b^2}$ . Applying these facts to (3.43), we arrive at

$$\|e_{k+1}^{(a)}\| \leq \|e_k^{(a)}\| + h_k L_\Phi (C^{(f)}\gamma + \|e_k^{(a)}\|) + h_k C^{(a)}\xi \quad (3.44)$$

$$\leq \|e_k^{(a)}\| (1 + h_k L_\Phi) + h_k (L_\Phi C^{(f)}\gamma + C^{(a)}\xi). \quad (3.45)$$

We can derive a more general form for (3.44) by observing the value of the error bound for increasing  $k$ .



$$\|e_0^{(a)}\| = 0 \quad (3.46)$$

$$\|e_1^{(a)}\| \leq h_0(L_{\Phi}C^{(f)}\gamma + C^{(a)}\xi) \quad (3.47)$$

$$\|e_2^{(a)}\| \leq \|e_1^{(a)}\|(1 + h_1L_{\Phi}) + h_1(L_{\Phi}C^{(f)}\gamma + C^{(a)}\xi) \quad (3.48)$$

$$= (h_0(1 + h_1L_{\Phi}) + h_1)(L_{\Phi}C^{(f)}\gamma + C^{(a)}\xi) \quad (3.49)$$

$$\vdots \quad \vdots \quad (3.50)$$

$$\|e_n^{(a)}\| \leq (L_{\Phi}C^{(f)}\gamma + C^{(a)}\xi) \sum_{m=0}^{n-1} h_m \prod_{j=m+1}^{n-1} (1 + h_jL_{\Phi}) \quad (3.51)$$

$$\leq (L_{\Phi}C^{(f)}\gamma + C^{(a)}\xi) \left( \sum_{m=0}^{n-1} h_m \right) \prod_{j=1}^{n-1} (1 + h_jL_{\Phi}) \quad (3.52)$$

We can bound (3.52) by noting that the sum of all the steplengths is equal to  $T$ :

$$\|e_n^{(a)}\| \leq (L_{\Phi}C^{(f)}\gamma + C^{(a)}\xi) T \prod_{j=1}^{n-1} (1 + h_jL_{\Phi}) \quad (3.53)$$

We can also note that

$$\prod_j (1 + h_jL_{\Phi}) = \prod_j e^{\ln(1+h_jL_{\Phi})} \quad (3.54)$$

$$\leq \prod_j e^{h_jL_{\Phi}} \quad (3.55)$$

since  $\ln(1+x) \leq x$  for  $x \geq 0$ , leaving us with the final form of our error bound:

$$\|e_n^{(a)}\| \leq Te^{\sum_j h_jL_{\Phi}} (L_{\Phi}C^{(f)}\gamma + C^{(a)}\xi) = Te^{TL_{\Phi}} (L_{\Phi}C^{(f)}\gamma + C^{(a)}\xi) \quad (3.56)$$

□

## Forward Evolution Error and Interpolation Error

Before concluding the discussion on the global error incurred in the AG evolution, it is worthwhile to compare and contrast the interpolation schemes mentioned in Algorithm 1. Strategy *i2* is desirable from the analytical standpoint, as it grants the definitive error bounds. However, a drawback of this strategy is that it incurs extra computational expense from resimulation. To minimize the cost of the resimulation, however, strategy *i2* could be coupled with a checkpointing scheme. (Checkpointing schemes are discussed in detail in the next chapter.)

Polynomial interpolation of the reference states (strategy *i1*) can incur a more reasonable computational cost than strategy *i2*. (Consider a piecewise linear interpolation scheme, for example.) Most interpolation schemes have well-known error bounds [Süli and Mayers, 2003, 183-184]. For example, for an  $n^{\text{th}}$  order interpolating polynomial  $\varphi[y(s)]$  satisfying  $\varphi[y(s)](t_k) = y(t_k)$ , the following bound exists if  $y(t) \in C^{(n+1)}[a, b]$ :

$$\|y - \varphi[y(s)]\|_{L^\infty} \leq \frac{1}{(n+1)!} \|y^{(n+1)}\|_{L^\infty} \max_{t \in [a, b]} \left| \prod_{i=0}^n (t - t_i) \right|, \quad (3.57)$$

where  $\{t_i\}$  denote the location of the interpolation nodes in time. It has also been shown that this upper bound can be minimized by choosing specific nodes corresponding to the roots of the  $n^{\text{th}}$  order Chebyshev polynomial, appropriately scaled

to the interval  $[a, b]$  [Süli and Mayers, 2003, 245-246]. Note that we cannot choose the interpolation nodes since this decision is made by the adaptive time stepping scheme, based on the properties of the differential equation. In practice, polynomial error should also reduce as the parameter  $\gamma$  and  $\xi$  are lowered, as this generates more interpolating nodes. This statement, however, is difficult to quantify and guarantee; hence, analysis of the AG global error is restricted to interpolation by re-simulation.

### 3.3.3 Objective Function Evaluation Error

I now analyze the error incurred in the evaluation of the objective function. We are interested in bounding the difference between true evaluation of the objective function  $J(y(T))$  and its computed value  $J(\hat{y}_{(T)})$ ,  $\|J(y(T)) - J(\hat{y}_{(T)})\|$ . This result is a natural corollary of Theorem 3.3.1.1 and the Lipschitz assumption on  $J$ .

**Corollary 3.3.3.1.** *Let Assumption 3.1.0.1 be satisfied. Further, suppose we use an adaptive one-step scheme to obtain the final state of the initial value problem (3.2). Then the error associated with the evaluation,*

$$\|J(y(T)) - J(\hat{y}_{(T)})\| \leq C_{obj}\gamma, \quad (3.58)$$

*where  $\gamma$  is the forward time-stepping tolerance and the constant  $C_{obj} > 0$ .*

## 3.4 Adaptive Time Stepping for Optimal Control Problems

I now discuss how to couple the gradient error analysis I presented above with convergence theory for the inexact Newton algorithm. Recall that the inexact Newton algorithm can be used to guarantee convergence to a local solution, given inexact derivative information. Hence, I begin this section by establishing the theoretical foundation of the inexact Newton method. I then explain how to couple adaptive time-stepping with the inexact Newton method to solve the optimal control problem. The coupling relies on using the measure of first-order optimality conditions to set the tolerance of the adaptive time-stepping algorithm. Of course, the optimality conditions change depending on whether we are considering unconstrained or explicitly-constrained optimal control problems. I first consider the case where we do not have explicit constraints, where the inexact Newton method can be used without alteration. I then conclude the chapter by considering the case where we have explicit constraints, which requires use of using inexact Newton algorithms to generate search directions for the interior point method (called “inexact interior-point” algorithms by Bellavia [1998]).

Before proceeding, it is important to revisit the inexact Newton method. Consider the unconstrained optimization problem

$$\min_x f(x), \tag{3.59}$$

for some twice continuously differentiable function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ . The inexact Newton scheme can be written as the following iteration:

$$x_{k+1} = x_k + s_k, \quad (3.60)$$

where  $s_k$  is obtained by solving the following linear system:

$$\nabla^2 f(x_k) s_k = -\nabla f(x_k) + r(x_k). \quad (3.61)$$

In the equation above, the term  $r(x_k)$  is called the *residual vector*. If the residual vector satisfies

$$\|r(x_k)\| \leq \eta_k \|\nabla f(x_k)\| \quad (3.62)$$

for a *forcing sequence*  $\{\eta_k\} < \mu$  for some  $\mu < 1$ , then we can guarantee convergence to a local solution given a sufficiently close starting guess. Given this overview of the inexact Newton method, I now will establish necessary notation and background required to make a precise statement and proof of its convergence.

For the remainder of this section, I will denote the set of Lipschitz continuous functions on  $D \subset \mathbb{R}^n$  as

$$Lip_L(D) = \{ h : D \rightarrow \mathbb{R}^m \mid \|h(x) - h(y)\| \leq L\|x - y\| \ \forall x, y \in D \}, \quad (3.63)$$

where  $L$  is the Lipschitz constant. Further, let  $B_\epsilon(x)$  denote the  $\epsilon$ -ball about the point  $x$ . Given the notation above, I can present the following classical estimates, which will be used to prove convergence of the inexact Newton method [Nocedal and Wright, 1999, 137].

**Lemma 3.4.0.1.** *Let  $D \subset \mathbb{R}^n$  be an open set and let  $\mathcal{L} : D \rightarrow \mathbb{R}^n$  be twice continuously differentiable on  $D$  with  $\nabla^2 L \in Lip_L(D)$ . Moreover, let  $x_* \in D$  be a point at which the second order sufficient optimality conditions are satisfied. Then there exists  $\epsilon > 0$  such that  $B_\epsilon(x_*) \subset D$  and for all  $x \in B_\epsilon(x_*)$ ,*

$$\|\nabla^2 f(x)\| \leq 2\|\nabla^2 f(x_*)\|, \quad (3.64)$$

$$\|\nabla^2 f(x)^{-1}\| \leq 2\|\nabla^2 f(x_*)^{-1}\| \quad \text{and}, \quad (3.65)$$

$$\frac{1}{2\|\nabla^2 f(x_*)^{-1}\|} \|x - x_*\| \leq \|\nabla f(x)\| \leq 2\|\nabla^2 f(x_*)\| \|x - x_*\|. \quad (3.66)$$

Using the estimates above, I now present the well-known proof of local convergence of the inexact Newton algorithm (3.61), [Nocedal and Wright, 1999, 52-53]. In this proof, I will be making an extra assumption that the forcing sequence  $\eta_k = O(\|\nabla f(x_k)\|)$ , which will consequently lead to a stronger rate of convergence. There are other choices of the forcing parameter, as noted in [Dembo and Steihaug, 1982, Eisenstat and Walker, 1994b].

**Theorem 3.4.0.1.** *Let  $D \subset \mathbb{R}^n$  be an open set and let  $\mathcal{L} : D \rightarrow \mathbb{R}^n$  be twice continuously differentiable on  $D$  with  $\nabla^2 L \in Lip_L(D)$ . Moreover, let  $x_* \in D$  be a point at*

which the second order sufficient optimality conditions are satisfied.

If the residual vector in the inexact Newton scheme (3.61) satisfies

$$\|r(x_k)\| \leq \kappa \|\nabla f(x_k)\|^2, \quad (3.67)$$

for some  $\kappa > 0$ , (i.e., we choose the forcing sequence  $\eta_k = O(\|\nabla f(x_k)\|)$ ) then there exists  $\epsilon > 0$  such that the inexact Newton scheme with starting point  $x_0 \in B_\epsilon(x_*)$  generates iterates  $\{x_k\}$  which converge to  $x_*$  and which obey

$$\|x_{k+1} - x_*\| \leq C \|x_k - x_*\|^2. \quad (3.68)$$

Though I only discuss local convergence theory here, I would like to note that globalization schemes for inexact Newton methods exist, such as linesearch methods. For example, in their work, Eisenstat and Walker [1994a] require that the step  $s_k$  satisfies both the residual vector criterion (3.62) and a sufficient decrease criterion

$$\|\nabla f(x_k + s_k)\| \leq [1 - t(1 - \eta_k)] \|\nabla f(x_k)\|, \quad (3.69)$$

for some  $t \in (0, 1)$ . These schemes insure convergence to a local solution when the starting guess is not sufficiently close, and are used for all examples found in the “Numerical Results” chapter of this dissertation. The remainder of this chapter discusses how to effectively couple the inexact Newton algorithm with adaptive time-stepping to solve unconstrained and explicitly-constrained optimal control problems.

### 3.4.1 Solving Unconstrained Optimal Control Problems with Inexact Newton

In order to solve optimal control problems with adaptive time stepping and the inexact Newton method, I now relate the discretization error of the optimal control problem (3.1) with the adaptive time-stepping tolerance parameter and the residual vector of the inexact Newton method. Before proceeding, I make the following assumption regarding the tolerances for the time-stepping algorithms.

**Assumption 3.4.1.1.** *Let the forward and the adjoint time-stepping tolerance values be the same (i.e., let  $\gamma = \xi$ ). Denote this single value for the tolerance as  $\tau$ .*

Recall that the unconstrained optimal control problem takes the form

$$\min_{u \in \mathbb{R}^n} f(u) = \hat{f}(y(u), u), \quad (3.70)$$

where  $(y(u), u)$  is the solution of an implicit differential equation constraint (3.1).

We can use the Newton method to obtain a search direction for the unconstrained optimization problem above by solving the following linear system for  $p_k$ :

$$\nabla^2 f(u_k) p_k = \nabla f(u_k). \quad (3.71)$$

Assuming that we use the adjoint-state method to compute the Hessian's action on a vector and the gradient (see [Heinkenschloss, 2008]), we still incur discretization



error via adaptive time-stepping, interpolation and quadrature. Hence, the computed search direction  $p_k^c$  actually satisfies the following linear system

$$\underbrace{(\nabla^2 f(u_k) + \Delta(u_k))}_{H_k} p_k^c = \underbrace{\nabla f(u_k) - \delta(u_k)}_{g_k}, \quad (3.72)$$

where  $\Delta(u_k)$  is the discretization error in the Hessian matrix and  $\delta(u_k)$  is the discretization error in the gradient. The computed Hessian and gradient is denoted as  $H_k$  and  $g_k$ , respectively. I now present the following theorem, which describes how to update the tolerance  $\tau$  as to guarantee, via inexact Newton theory, local convergence to the unconstrained optimization problem.

**Theorem 3.4.1.1.** *Consider the problem (3.70). Suppose we obtained search directions  $p_k^c$  for the problem above by solving the perturbed Newton system (3.72). Assume that the following:*

- *Assumption 3.4.1.1 holds.*
- *The sequence of search directions  $\{p_k^c\}$  is bounded.*
- *All linear system solves used in the computation of the search direction are exact (as opposed to iterative).*
- *The norm of the Hessian discretization error,  $\|\Delta(u_k)\| = O(\tau_k)$ , where  $\tau_k$  is the value of the time-stepping tolerance at the  $k^{\text{th}}$  optimization iteration.*

- For all  $k$ , the values of the tolerance  $\tau_k$  satisfy:

$$\|[\nabla^2 f(u_k)]^{-1} \Delta(u_k)\| < 1. \quad (3.73)$$

(This ensures that the Hessian discretization error is sufficiently small, guaranteeing that the computed Hessian  $H_k$  is invertible if  $\nabla^2 f(u_k)$  is invertible.)

Then, using the following update scheme for the tolerance:

$$\tau_{k+1} = \min(\tau_k, \kappa \|g_k\|^2), \quad (3.74)$$

is enough to guarantee local convergence to the problem (3.70), where  $\kappa > 0$  is some scaling constant.

*Proof.* The next step is then to arrange equation (3.72) so that it resembles the inexact Newton equation (3.61). We now rearrange equation (3.72) as

$$\nabla^2 f(u_k) p_k^c = \nabla f(u_k) - (\Delta(u_k) p_k^c + \delta(u_k)), \quad (3.75)$$

in which case we make the distinction that

$$r(u_k) = \Delta(u_k) p_k^c + \delta(u_k) \quad (3.76)$$

by comparison. The next step is then to relate the discretization errors above to the time-stepping tolerance parameters. By assumption, the forward and adjoint

tolerances are the same, i.e.,  $\tau = \xi = \gamma$ . Since I showed in the previous section that the gradient error can be lowered by lowering  $\tau$ , it follows that the gradient discretization error can be bounded by

$$\|\delta(u_k)\| \leq C_k^{grad} \tau_k \quad (3.77)$$

for some constant  $C_k^{grad} > 0$ . By assumption, the computed steps  $p_k^c$  are bounded and the norm of the Hessian discretization error  $\|\Delta(u_k)\|$  is  $O(\tau_k)$ . Hence, for some  $C_k^{Hess} > 0$ ,

$$\|\Delta(u_k) p_k^c\| \leq \|\Delta(u_k)\| \|p_k^c\| = C_k^{Hess} \tau_k. \quad (3.78)$$

Given these bounds, we can assert that for some  $\tilde{C}_k > 0$ ,

$$\|r(u_k)\| = \|\Delta(u_k) p_k^c + \delta(u_k)\| \approx \tilde{C}_k \tau_k. \quad (3.79)$$

I now conclude this part of the chapter by relating the bound (3.79) to the bound required of the residual vector to attain local convergence (from Theorem 3.4.0.1):

$$\tilde{C}_k \tau_k \approx \|r(u_k)\| \leq \kappa \|\nabla f(u_k)\|^2, \quad (3.80)$$

for  $\kappa > 0$ . Hence, in order to enforce local convergence for the optimal control problem (3.70) by using adaptive time stepping for the reference and adjoint equations, we

can use the following update scheme for the tolerance  $\tau_k$ :

$$\tau_{k+1} = \min(\tau_k, \kappa \|g_k\|^2), \quad (3.81)$$

where  $g_k$  is the computed gradient at optimization iteration  $k$  and  $\kappa$  is some scaling factor. □

Before concluding discussion on this tolerance update method, I would like to note that the boundedness assumption on sequence of computed steps  $\{p_k^c\}$  is a corollary of Theorem 3.4.0.1, which guarantees that the iterates  $\{u_k\}$  are locally convergent as long as the forcing sequence satisfies the inexact Newton criterion (3.67). I would also like to point out that the tolerance updating scheme (3.81) has three nice properties. First, it generates a monotone decreasing sequence of tolerances. Second, as the optimization algorithm generates a control close to a local solution,  $\|g_k\|^2 \approx \|\nabla f(u_k)\|^2$ , by the tolerance update rule (3.81) and the bound (3.77). Finally, the update scheme only requires *computable* (and readily available) values.

In the next section, I discuss how to relate the inexact Newton method, the time stepping tolerance for explicitly constrained optimal control problems. I will make use of the same type of analysis as above, through use of the inexact interior-point algorithm.

### 3.4.2 Inexact Interior Point Methods For Constrained Optimal Control Problems

I now discuss constrained optimization problems – specifically, nonlinear programs (NLPs). I begin with the mathematical definition of the NLP and the associated KKT conditions. I then discuss the barrier approach for solving the NLP. Then, I describe the so-called “Inexact Interior Point” algorithm, which couples interior point methods with inexact Newton methods [Eisenstat and Walker, 1994a,b] to generate search directions. To end this section, I couple the solution approach for the barrier problem (given a fixed barrier parameter) with the Inexact Newton theory I developed in the previous section. This involves a discussion on how to theoretically accommodate inexact derivatives when the inexactness comes from using the adjoint state method with adaptive time stepping.

Supposing we have explicit constraints, then the nonlinear program we consider can be written as the following:

$$\begin{aligned}
 \min_{u \in \mathbb{R}^n} \quad & f(u) = \hat{f}(y(u), u) \\
 \text{s.t.} \quad & c_E(u) = 0 \\
 & c_I(u) \geq 0,
 \end{aligned} \tag{3.82}$$

where  $(y(u), u)$  solves the differential equation constraint of the problem (3.1),  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ ,  $c_E(u) : \mathbb{R}^n \rightarrow \mathbb{R}^{m_E}$  represent equality constraints and  $c_I(u) : \mathbb{R}^n \rightarrow \mathbb{R}^{m_I}$  denote inequality constraints and the control is denoted  $u$ . We can eliminate the

explicit inequality constraint by defining *slack variables*. Defining the slack variable vector  $s \in \mathbb{R}^{m_I}$ , we can transform the NLP as the following problem

$$\begin{aligned}
& \min_{u,s} f(u) \\
& \text{s.t. } c_E(u) = 0 \\
& \quad c_I(u) - s = 0 \\
& \quad s \geq 0
\end{aligned} \tag{3.83}$$

The associated KKT conditions with the above problem can be stated as follows:

$$\underbrace{\begin{bmatrix} \nabla f(u) - A_E^T(u)y - A_I^T(u)z \\ c_E(u) \\ c_I(u) - s \\ SZe \end{bmatrix}}_{H(u,s,y,z)} = 0. \tag{3.84}$$

where  $A_E$  and  $A_I$  are the Jacobian of the equality and inequality constraints, respectively,  $y, z$  are Lagrange multipliers,  $Z = \text{diag}(z)$  and  $S = \text{diag}(s)$ . Using the barrier approach to solving (3.83), we obtain the following NLP

$$\begin{aligned}
& \min_{u,s} f(u) - \mu \sum_{i=0}^{m_I} \ln(s_i) \\
& \text{s.t. } c_E(u) = 0 \\
& \quad c_I(u) - s = 0,
\end{aligned} \tag{3.85}$$

where  $\mu$  is called the *barrier parameter*. It is worth noting that the optimal solutions  $u_*(\mu)$  of (3.85) converge to an optimal solution of (3.82) as the barrier parameter

$\mu \rightarrow 0$  [Nocedal and Wright, 1999]. The associated KKT conditions with the problem above can be written as:

$$\begin{bmatrix} \nabla f(u) - A_E^T(u)y - A_I^T(u)z \\ c_E(u) \\ c_I(u) - s \\ SZe - \mu e \end{bmatrix} = H(u, s, y, z) - \underbrace{\mu \begin{bmatrix} 0 \\ 0 \\ 0 \\ e \end{bmatrix}}_{\bar{e}} = 0, \quad (3.86)$$

I now describe a standard approach of solving the barrier problem (3.85) given a fixed  $\mu = \bar{\mu}$ . The idea centers around using a Newton-type method to solve the nonlinear system (3.86), generating a search direction which leads to an update enforcing  $s, z \geq 0$ . Algorithm 2 describes this idea in more detail. Note that in Algorithm 2,

---

**Algorithm 2:** Solving the Primal-Dual Equations for a fixed  $\mu = \bar{\mu}$ .

---

- while** ( $\|H(u_k, s_k, y_k, z_k)\|_\infty > \bar{\mu}$ )  
 1) Given data:  $(u_k, s_k, y_k, z_k)$  with  $(s_k, z_k) > 0$ .  
 2) Solve the following:

$$H'(u_k, s_k, y_k, z_k) p_k = -H(u_k, s_k, y_k, z_k) + \bar{\mu} \bar{e}, \quad (3.87)$$

- 3) Determine new steplength  $\alpha_k$   
 4) Set  $u_{k+1} = u_k + \alpha_k p_k^u$ ,  $s_{k+1} = s_k + \alpha_k p_k^s$   
 $y_{k+1} = y_k + \alpha_k p_k^y$ ,  $z_{k+1} = z_k + \alpha_k p_k^z$

**end while**

---

the specifics of how to choose a steplength (step 4) is purposely left vague as there are many different algorithms to choose such a steplength. Generally, such a steplength is chosen to ensure that the next iterate strictly satisfies  $s, z \geq 0$  (so-called “fraction-to-boundary” rules), as well as that sufficient progress is made towards solving (3.85) [Nocedal and Wright, 1999, Wächter, 2009]. Once the optimality conditions (3.86)

are satisfied to the specified tolerance, the value of  $\mu$  is lowered, defining a new barrier problem. (Adaptive barrier updating strategies, where the value of  $\mu$  is changed per every iteration of Algorithm 2, is not considered here.) Then, the solution from the previous barrier problem is used as a starting guess for the new barrier problem.

Suppose we use the adjoint-state method, with adaptive time-stepping, to generate derivatives used in (3.87). The analysis performed in the previous section can be applied to Algorithm 2, albeit with more restrictions. I present the following corollary, which addresses local convergence for the problem (3.85) given a fixed barrier parameter  $\mu = \bar{\mu}$ . Note that, for the remainder of this chapter, I denote  $x_k = (u_k, s_k, y_k, z_k)$ .

**Corollary 3.4.2.1.** *Given the problem (3.85), with a fixed barrier parameter  $\mu = \bar{\mu}$ .*

*Suppose the following:*

1. *The assumptions from Theorem 3.4.1.1 are satisfied.*
2. *The constraints and their respective Jacobians can either be evaluated exactly, or at most incur an  $O(\tau)$  error.*

*Then, using the following update scheme for the tolerance:*

$$\tau_{k+1} = \min(\tau_k, \kappa \|H(x_k) + \delta(x_k) - \bar{\mu}\bar{e}\|^2), \quad (3.88)$$

*in conjunction with Algorithm 2 is enough to guarantee local convergence for (3.85), for a fixed barrier parameter.*



*Proof.* I begin by introducing the following terms to denote various discretization errors:

- $\Delta(x_k)$ : discretization error in the Hessian,  $H'(x_k)$
- $\delta(x_k)$ : discretization error in  $H(x_k)$

Note that, from assumption, the magnitude of the discretization errors listed above are  $O(\tau)$ . Using the notation above, the primal-dual equation takes the following form:

$$(H'(x_k) + \Delta(x_k))p_k = -(H(x_k) - \bar{\mu}\bar{e} + \delta(x_k)). \quad (3.89)$$

Also, from assumption, the sequence of search directions generated by solving (3.89),  $\{p_k\}$ , is bounded. I then rewrite (3.89) as:

$$H'(x_k)p_k = -H(x_k) - (\delta(x_k) + \Delta(x_k)p_k - \bar{\mu}\bar{e}). \quad (3.90)$$

Defining the residual vector  $r(x_k)$  as

$$r(x_k) = -(\delta(x_k) + \Delta(x_k)p_k - \bar{\mu}\bar{e}), \quad (3.91)$$

I can write the final form of the primal-dual equations as

$$H'(x_k)p_k = -H(x_k) + r(x_k), \quad (3.92)$$

which is of the same form as the inexact Newton scheme for the unconstrained problem (3.61). The proof is completed by Theorem 3.4.1.1, and the stopping criteria of the `while` loop in Algorithm 2. □

Note that for the inexact interior point method, in order to satisfy the inexact Newton criterion

$$\|r(x_k)\| \leq \kappa \|H(x_k)\|^2, \quad (3.93)$$

the tolerance update must take the following form:

$$\tau_{k+1} = \min(\tau_k, \kappa \|\bar{H}(x_k)\|^2), \quad (3.94)$$

where the computed NLP error  $\bar{H}(x) = H(x) + \delta(x) - \bar{\mu}\bar{e}$ , and  $\kappa > 0$  is some scaling factor. For each  $\mu$ , the tolerance update above helps ensure that the (3.86) is satisfied (up to user-specified tolerance) despite discretization error. Since the solutions of the barrier subproblems (3.85) converge to the solution of the NLP (3.82), I conjecture that the corollary above aids the solution of the NLP problem. In the “Numerical Results” section, I present a reservoir engineering optimal control problem that supports this claim.

# Chapter 4

## Computational Background

After discussing the theoretical background of my thesis work, I now segue to the computational tool that will verify the theory I had established. This chapter introduces the “Time Stepping Package for Optimization”, or `TSOpt`. `TSOpt` is a “middle-ware” package written in C++, designed to act as an “interface for time-stepping simulation”, providing a way for simulation software to inter-operate with optimization software [Symes, 2006, Enriquez and Symes, 2009]. `TSOpt` is capable of encapsulating the reference, linearized and adjoint simulators in a single object, and properly arrange their execution. `TSOpt` also aids in providing necessary data structures for the optimization algorithm (e.g., the gradient, formed via the adjoint-state method).

This chapter is organized as follows: the first section will introduce RVL and section two will then discuss the Alg framework developed by Tony Padula. RVL and the Alg framework provides the foundation for `TSOpt`. The most notable features of `TSOpt` include its modular code structure, due to use of the Alg framework from the

Rice Vector Library (RVL), and also accommodation of a generic data structure type through templating. The specifics of the structure of `TSOpt` and its features will be discussed in more detail in section four.

## 4.1 The Rice Vector Library (RVL)

This section introduces the RVL. Understanding the the main functionality of the Rice Vector Library is crucial to understanding the new version of `TSOpt`; `TSOpt` interfaces with RVL (and the software frameworks that stem from RVL) in order to numerically solve optimal control problems.

### 4.1.1 The Rice Vector Library (RVL)

The Rice Vector Library is a software framework consisting of C++ abstractions of Hilbert space components, making it an appropriate foundation for Newton-based optimization algorithms [Padula et al., 2009]. RVL was designed to enable expression and implementation of “coordinate-free” linear algebra and optimization algorithms. Further, RVL promotes creation of reusable algorithms, to accommodate “different application, data storage models and execution strategies” [Padula et al., 2009]. RVL’s components can be grouped into two categories: the *calculus* classes and *data management* classes. The *calculus* classes include abstractions of “a vector space, a vector, a vector-valued function and a Linear Operator.” The *data management* classes include “Data Containers and encapsulated functions”.

One of the fundamental software frameworks that stem from RVL is called the `Alg` framework, which provides a computational abstraction of all algorithms. The `Alg` framework, for example, is the base for a suite of linear algebra and optimization solvers in RVL. The `Alg` framework will also be the foundation for the `TSOpt` framework; it is imperative, hence, that we discuss the `Alg` framework in more detail.

## 4.2 RVL and the Alg Framework

Padula et al. explored what it means for a program to be an algorithm in [Padula et al., 2009]. The answer was simple: an algorithm is a program that runs in a finite amount of time (i.e., it stops). Ideally, it should also be able to relay information if its execution was successful or not. This definition easily lends itself to the following C++ implementation of a base class:

```
class Algorithm {
public:
    virtual bool run() = 0;
};
```

The class `Algorithm` became the foundation of the `Alg` framework. Using the base class `Algorithm`, a variety of subclasses can be defined as well – allowing us to abstract the functionality of different types of numerical algorithms, such as optimization and simulation algorithms [Padula et al., 2009]. This led to the insight that, since all time-stepping schemes are algorithms, `TSOpt`'s components can be implemented from

Algorithm objects. In fact, three subclasses of `Algorithm` serve as the foundation of `TSOpt`. These subclasses are called the `StateAlg`, the `LoopAlg` and the `ListAlg` classes. The UML diagram in figure 4.1 show these subclasses, along with their methods. Since it is crucial that we understand their functionality, they are discussed

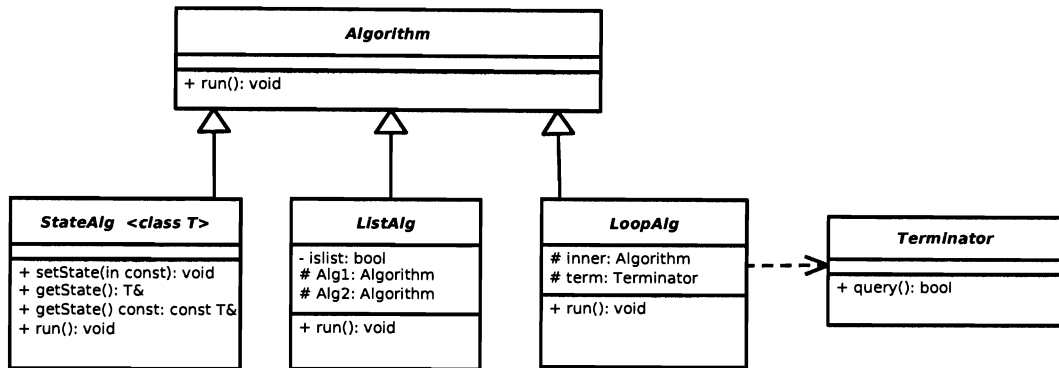


Figure 4.1: The Alg class and its subclasses

in detail below.

### 4.2.1 The StateAlg Class

A `StateAlg` is an `Algorithm` that has an explicit state variable. This abstraction is useful in a variety of mathematical algorithms, such as a Newton method where the internal state is the current value of the optimization variable. A `StateAlg` must provide methods to assign and retrieve values from its state. The following is the implementation for the `StateAlg` base class:

```

template<class T>
class StateAlg: public Algorithm {

```

```
public:
    virtual void setState(const T & x) = 0;
    virtual const T & getState() const = 0;
    virtual T & getState() = 0;
};
```

Also note that the state type is templated, meaning that this concrete subclasses of `StateAlg` can use other objects as its internal state.

## 4.2.2 The LoopAlg and terminator Classes

The Alg Framework also has a class capable of abstracting looping algorithms, such as GMRES. This class, which derives from `Algorithm` is called `LoopAlg`. A `LoopAlg` object's job is to repeat execution of an `Algorithm` object (through the `run()` method) until some criteria is met. This criteria is encapsulated in something called a `Terminator` object. The `Terminator` base class is implemented the following way:

```
class Terminator {
public:
    virtual ~Terminator() {}
    virtual bool query() = 0;
};
```

All subclasses of `Terminator` must provide a `query()` method that either returns true or false. The `LoopAlg` object will then use this `query()` function to determine whether to stop the loop or not. Given the `Algorithm` inside and the `Terminator` term, we implement `LoopAlg` class' run method as:

```

virtual bool run() {
    bool t1 = true;
    while( (!term.query()) && t1 )
        t1 = inside.run();

    return t1;
}

```

Note that the `LoopAlg` also needs to ensure that its `Algorithm` object completed its job successfully (i.e., it returned `true`).

### 4.2.3 The ListAlg Class

The `ListAlg` class is just an `Algorithm` that is composed of two other `Algorithms`. This particular `Algorithm`'s `run()` command executes the two `Algorithms` in order, one after another. Given two `Algorithm` objects `one` and `two`, we implement `ListAlg` class' `run` method as:

```

virtual bool run() {
    bool t1 = true, t2 = true;
    t1 = one.run();
    if( islist )
        t2 = two.run();

    return (t1 && t2);
}

```

## 4.3 The Software Framework of TSOpt

After discussing RVL and the Alg framework, we can now discuss TSOpt. TSOpt is a software package that encapsulates reference, linearized and adjoint simulations in a



single object. As mentioned in earlier sections, `TSOpt` uses `RVL` and the `Alg` package as the foundation of its framework. This section presents the main components of the `TSOpt` framework, which consist of the `time`, `state`, `timestep`, `sim`, `terminator` and `jet` classes.

### 4.3.1 The time Hierarchy

The `time` class is perhaps the most fundamental class in `TSOpt`. This base class `Time` is an abstraction of the simulation times. A `time` object only knows the current simulation time; it does not know extra information about the simulation, such as the final simulation time or the step length. All subclasses of `time` must provide methods for assignment of simulation time, as well as the comparison operators for “less than” (`<`) and “greater than” (`>`). There are two current concrete subclasses of `time`: the `DiscreteTime` object and the `RealTime` object.

The `DiscreteTime` object is used for simulations of fixed time steps; it uses a time index (in the form of an `int`) to keep track of the simulation time. Hence, by altering this time index, we can change the simulation time. The `RealTime` object, on the other hand, allows for variable time steps. It does not have an internal time index; it only holds a `double` to represent the current simulation time, which can be accessed and altered directly.

### 4.3.2 The State Class

The `State` class is not, strictly speaking, a part of `TSOpt` – though a couple of different concrete `State` classes have been implemented in `TSOpt`. Users of `TSOpt` can implement their own `State` class to act as an interface between their preferred simulator data structure and `TSOpt`. A `State` object is composed of two objects: a data structure to hold data (e.g., an array) and a `time` object, which holds the current simulation time associated with the data. This relationship can be seen in the UML diagram, figure (4.2). All `State` classes must implement methods to get and set the `time` object, and methods to access and alter its internal data structure. There are two examples of `State` subclasses that have been implemented in `TSOpt`,

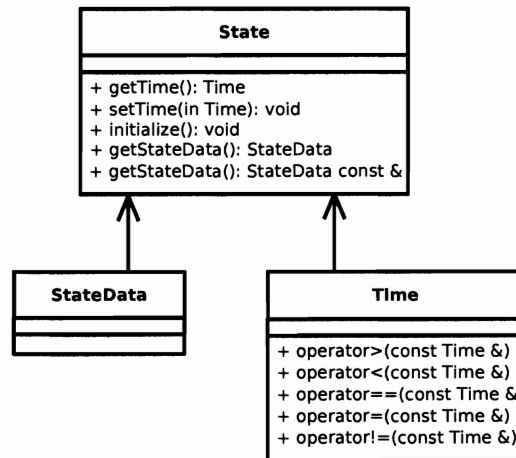


Figure 4.2: The `State` class and its components

to accompany the two different time types: `RnState` and `RealRnState`. The `RnState` class contains a `DiscreteTime` object, and is used for fixed time step simulations. (The “Rn” refers to the vector space  $\mathbb{R}^n$ .) The `RnState` class internally contains an

rn struct, defined with the following components:

```
typedef struct {
    /** time index */
    int it;
    /** state dim */
    int nu;
    /** control dim */
    int nc;
    /** state samples */
    float * u;
    /** control samples */
    float * c;
} rn;
```

The class `RnState` then provides methods to access and initialize the components of the `rn` struct.

The `RealRnState`, in turn, contains a `RealTime` object and is used for adaptive time step simulations. Like `RnState`, `RealRnState` is a wrapper class for the `realrn` struct. There are two differences worth noting between the `RnState` and `RealRnState` classes, however. First, `RealRnState`'s internal data type is `double`, while `RnState`'s inner data type is `float`. Also, since it is not relevant in adaptive time stepping, the `realrn` struct does not contain a time index component.

### 4.3.3 The TimeStep Class

The `TimeStep` class is the base class for all time stepping methods in `TSOpt`. The `TimeStep` class is implemented as follows:

```

class TimeStep: public StateAlg<TimeState>, public Writeable {
public:
    virtual ~TimeStep() {}
    void setTime(Time const & t) { (this->getState()).setTime(t); }
    Time const & getTime() const { return (this->getState()).getTime(); }
    virtual Time const & getNextTime() const = 0;
};

```

Note that the `TimeStep` class derives from `StateAlg`. On top of `StateAlg`'s functionality, however, `TimeStep` adds the functions `setTime()` and `getTime()` for reading and changing the simulation time. Furthermore, `TimeStep` subclasses must provide a read-only method to get the next simulation time, which will be suitable for adaptive time-stepping schemes. `TSOpt` requires that the user define a *single* forward, linearized and adjoint step as (inherited) `TimeStep` objects.

#### 4.3.4 The Sim Hierarchy

The `Sim` class, as its name implies, is a simulator class. It orchestrates a `StateAlg` object, a `Terminator` object and a `Time` object in order to perform the simulation. Concrete subclasses of `Sim` also implement different simulation/memory managing schemes for use in either the linearized or adjoint computations. The UML diagram 4.3 show the subclasses of the `Sim` class. These subclasses, the `StdSim`, `RASim` and `CPSim` classes, will be explained in more detail below.

The subclass `StdSim` is a “forgetful” simulator; to provide the appropriate reference state during the adjoint evolution, the `StdSim` will run the reference simulator from the initial time until the desired time (which is taken to be the next time level

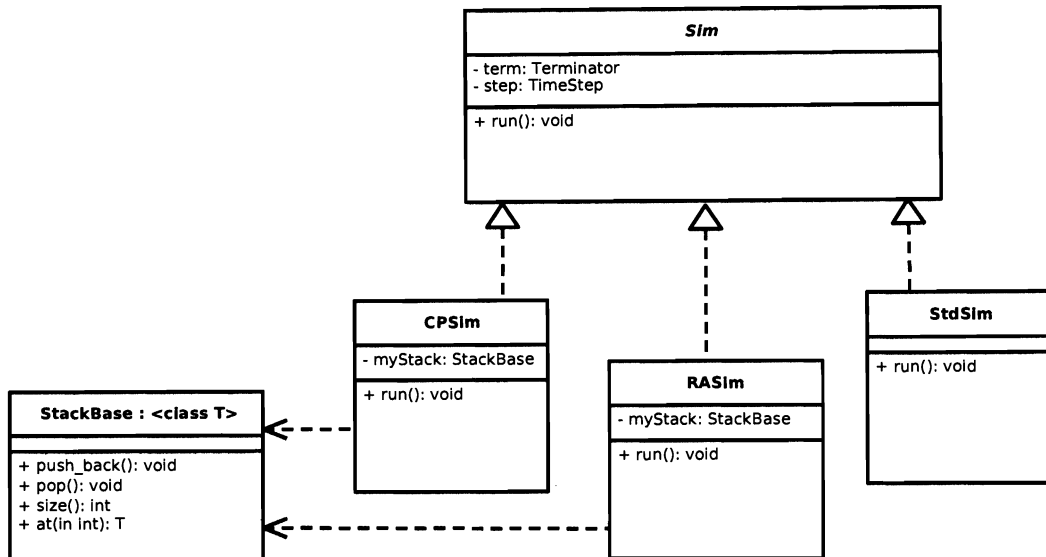


Figure 4.3: The `Sim` class and its derived classes.

in the adjoint computation). This `Sim` subclass does not require the storage of the simulation state history. Further, an Algorithm called `initstep` that is required for the construction of the `StdSim` object; this allows users to write custom initialization schemes for their simulator. One example use of the `initstep` class is to reset the simulation state to its initial values. Given a `TimeStep` object `step` and a corresponding `Terminator` `term`, the `StdSim`'s `run` method is implemented in the following manner:

```

void run() {
  try {
    LoopAlg a(this->step, this->term);
    ListAlg aa(this->initstep, a);
    aa.run();
  }
  catch (RVLEException & e) {... }
}

```

In contrast, the subclass `RASim` is a “remember-all” simulator. As it runs the reference simulation, it saves all the simulation states into a user-defined stack – eliminating the need to run the reference simulation more than once. The values in the stack are then appropriately accessed during the adjoint evolution.

In order to create a stack in `TSOpt`, users must implement a concrete subclass of the `stackBase` class, which is shown in the UML diagram 4.4 All `Sim` subclasses

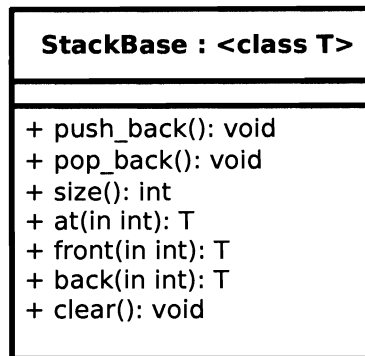


Figure 4.4: The `stackBase` class and its methods.

whose functionality necessitates storage of simulation states need to provide a concrete `stackBase` class to the constructor. For example, the following objects are needed in order to construct an `RASim` object: a `TimeStep`, `Terminator` and `stackBase`. One concrete `stackBase` subclass available in `TSOpt` is the `stdVector` class, which acts as a wrapper to the standard library’s `vector` class.

Other `Sim` subclasses exist in `TSOpt`; of note is the `CPSim` class, which uses Griewank’s optimal checkpointing scheme [Griewank and Walther, 2000]. Checkpointing is the “middle ground” between the two aforementioned strategies of a “for-

getful” simulator and a “remember-all” simulator. Two types of checkpointing exist in TSOpt: offline mode for fixed time step simulations, and online mode for adaptive simulations. I discuss the notion behind checkpointing in more detail below.

## Checkpointing

Recall that using adjoint method to obtain the gradient of the objective function necessitates access to the values of the state vector in *reverse*. This, however, can be problematic because the state vector can be large. Repeatedly recalculating the state vector, as is done by the `StdSim` class, comes at a computational cost of  $\frac{N^2}{2}$  (where  $N$  is the number of time-steps) and is generally prohibitive for large problems. Alternatively, storing the whole state vector like the `RASim` class can be costly in terms of memory. For example, for a typical 2D Reverse Time Migration problem, storing the full state vector requires  $O(10^6)$  Gigawords) in space and  $O(10^4)$  Gigawords) time steps. This could lead the program to use disk-swapped memory, which adversely affects the program execution time.

To avoid the steep computational and storage costs associated with the “forgetful” and “Remember-All” strategies, Griewank proposed an algorithm called checkpointing [Griewank and Walther, 2000]. The idea behind checkpointing is actually an intelligent combination of the two previously mentioned strategies: save a few states in some buffer (called checkpoints), and then forward-simulate from the nearest saved state until the time of interest. As the backward traversal continues, the checkpoints are updated such that none have been passed (and rendered useless) by the traver-

sal. Through this process, checkpointing eliminates the need to store the whole state vector while minimizing the recomputation of states. Given some assumptions of the costs of memory access and recomputation, Griewank also proved the optimality of his checkpointing algorithm in Griewank and Walther [2000]; given  $N_B$  buffers and  $N_S$  states such that  $N_B \ll N_S$ , his checkpointing scheme only adds logarithmic (i.e.,  $O(\log(N_S))$ ) recomputation cost.

Griewank implemented his optimal checkpointing algorithm in a package called *Revolve* [Griewank and Walther, 2000]. *Revolve* has two main phases in its execution. Given the number of time steps to be taken, the *scheduling phase* of *Revolve* determines the optimal checkpoint placement. Then, the *backward traversal phase* dictates what should be done to complete the backward traversal of states; this explicitly states if the saved checkpoints should be used, updated, or if a forward simulation (starting from a previously saved state) needs to be performed. Generally, *Revolve* is used such that the *scheduling phase* is immediately followed by the *backward traversal phase*. It was shown in [Enriquez, 2008], however, that separating execution of the scheduling phase and the backward traversal phase leads to a more efficient checkpointing algorithm. The implementation of `CPSim` in `TSOpt` follows the algorithm found in [Enriquez, 2008].

### **Adaptive Checkpointing**

In `ARevolve`, adaptive checkpointing works like fixed-step checkpointing algorithms, with the exception of not requiring an input of the number of time-steps to be taken.



In exchange, however, the user must set an algorithmic flag to denote that the forward evolution is finished, and the simulations are ready for the adjoint simulation. The biggest limitation of Hinze and Sternberg [2005]’s checkpointing algorithm, however, is that it does not cater to taking adaptive simulation in the adjoint field. `AREvolve` makes the assumption that the time levels in the adjoint and reference field align, implying that the adjoint time grid will be dictated by the reference simulation. This assumption is often incorrect, as the adjoint dynamics may have very little similarities with the reference dynamics (e.g., adaptive quadrature).

I hence create the adaptive checkpointing algorithm to cater to adaptive simulations in *both* the reference and adjoint fields. The idea is to use `AREvolve` to fill (and supply nodes to) an interpolation buffer, which moves along with the adjoint simulation. Ideally, the interpolation buffer should have size  $n + 1$ , where  $n$  is the order of the time-stepping scheme. The extra algorithmic work then comes from managing the interpolation buffer, as well as managing the calls made to the `AREvolve`. Algorithm 6 in the Appendix shows though pseudo-code how this adaptive checkpointing algorithm was structured.

Similar to the checkpointing algorithm in [Enriquez, 2008], the adaptive checkpointing algorithm consists of a forward mode and a backward mode – ensuring that the full forward evolution runs only once before the adjoint evolution takes place. The key difference here is the incorporation of the interpolation buffer, which itself is a deque that is being managed by a class. (The deque is a good choice for such an

algorithm since push and pop operations are supported at both ends of the buffer, for  $O(1)$  computational complexity.) Every time we “update” the interpolation buffer, it simply means that one slot in the buffer is replaced with a new interpolation node, such that the interpolation nodes are in order (in time).

### 4.3.5 The Time Terminator Hierarchy

Recall that the `Sim` subclasses requires a `Terminator` class, which it queries when the simulation should stop. The main criterion for when the simulation should stop is when the simulation time has reached its intended target time. To this end, `TSOpt` has a `Terminator` subclass, `TimeTerminator`, that is aware of the the simulation time. Like all `Terminator` objects, it has a `query()` function; this particular base class just allows the `query()`’s output to rely on the simulation time.

The `TimeTerminator` class has a variety of useful subclasses: a `FwdTimeTerminator` (a time terminator for forward time-marching schemes), a `BwdTimeTerminator` (a time terminator for backward time marching schemes), an `AndTerminator` and an `OrTerminator`. The `AndTerminator` and `OrTerminator` have `query()` functions that output the result of the logical operation of two terminators’ `query()` function.

### 4.3.6 The jet Hierarchy

The term “jet”, in applied mathematics, refers to a collection of a function, its derivative and its adjoint. True to this definition, the `jet` class is meant to hold the refer-

ence, linearized and adjoint simulators, and is at the highest level of TSOpt hierarchy. The `jet` subclasses require a `Sim` object for the forward evolution, and two triples of `timestep`, `stateAlg` and `timeTerminator` objects for both the linearized and adjoint evolution. This class assumes that the collection of objects pertaining to the forward, linearized and adjoint evolution are related in the appropriate sense. The following figure is a UML diagram showing the relationship between the `jet` class and its components.

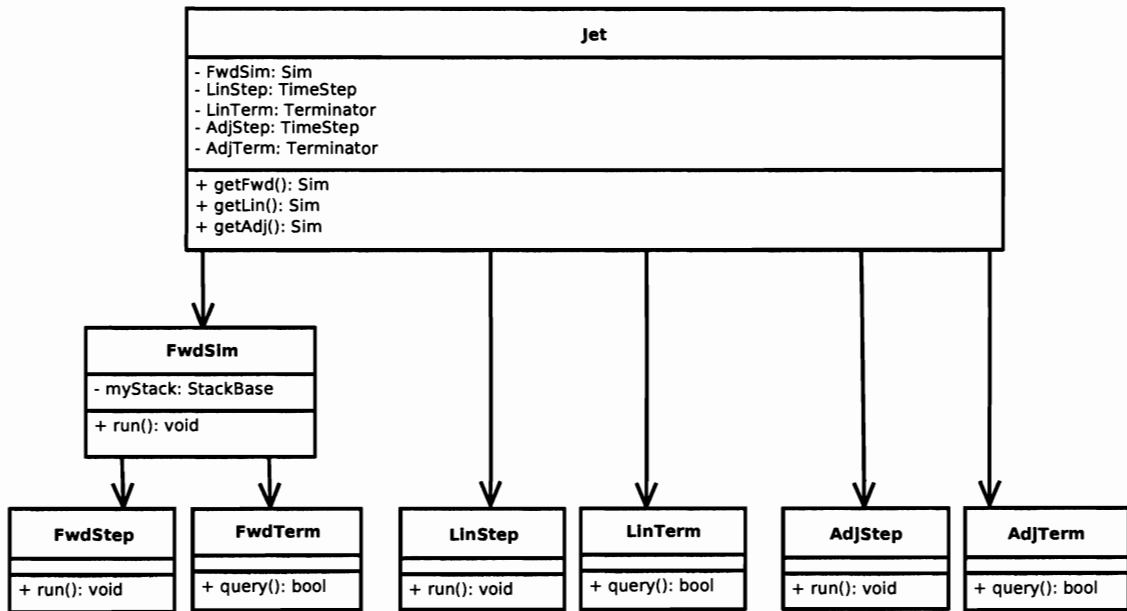


Figure 4.5: The `jet` class and its components.

The `jet` objects provide three very important functions that return the forward evolution `Sim` object or create a linearized and adjoint evolution `Sim` objects, respectively called `getFwd()`, `getLin()`, and `getAdj()`. It is worth noting how this simplifies coding at the top (user) level; in order to run the forward, linearized and adjoint

simulations, one would only need to code the following lines in `main()`:

```
...           // Construct various objects that jet needs
jet j(...); // Create jet object
j.getFwd().run(); // Run forward sim
j.getLin().run(); // Run lin. sim
j.getAdj().run(); // Run adj. sim
```

## 4.4 TSOpt and UMin

Recall that TSOpt provides various simulation operators whose output can be used in conjunction with optimization algorithms. If we are considering a purely unconstrained optimization problem, we can use RVL's UMin (“unconstrained minimization”) package. Similar to TSOpt, UMin was created by subclassing Alg components. Currently, the LBFGS and Conjugate-Gradient Trust-Region (CGTR) algorithms are available in UMin.

To use the UMin package, the user must create three `RVL::FunctionObjects`. In RVL, `FunctionObjects` act on RVL data containers, mimicking a (mathematic) operator acting on a variable. `FunctionObjects` are based on the “Acyclic Visitor” design pattern [Gamma et al., 1998], which “allows new functions to be added to existing class hierarchies without affecting those hierarchies, and without creating the dependency cycles.” In order to use the UMin package, a `FunctionObject` must be created to supply the following by using the Jet object: objective function evaluation, the gradient vector evaluation and the Hessian matrix evaluation.

The collection of `FunctionObjects` will be used to construct an `RVL::Functional` object, which is the interface for scalar-valued vector functions. `Functional` objects must provide first and second derivatives (gradient and Hessian), by using the `FunctionObjects` mentioned above. For example, the code below shows generic code that uses the `Jet` object to form a gradient `FunctionObjects` and `Functionals`.

```
class GradFunctionObject{
private:
    jet j;

public:
    void operator()(LocalDataContainer<Scalar> & y,
                  LocalDataContainer<Scalar> const & x) {

        jet.setControl(x);           // set control for fwd/adj sim.
        jet.getAdj().run();          // run adjoint simulation
        jet.getAdj().getGrad(y);     // get gradient via reference
    }
};

class ExampleFunctional {
protected:
    virtual void applyGradient(const Vector<Scalar> & x,
                              Vector<Scalar> & g) const {

        GradFunctionObject<Scalar> f(...); // make GradFunctionObject
        g.eval(f,x);                        // eval uses overloaded ()
                                           // operator defined in
                                           // GradFunctionObject
    }
};
```

The `Functional` object is used to make a `FunctionalEvaluation` object, which in

turn, can then be passed to the `UMin` framework to perform the optimization. For a more thorough discussion of this process, and the associated classes, see [Padula et al., 2009].

## 4.5 TSOpt and External Optimization Packages

Sometimes, it is necessary to consider explicit constraints for the optimal control problem. For example, my target application is an optimal control problem with (oil) reservoir simulation constraints. This problem features equality and bound constraints, representing physical limitations of a reservoir model and its wells. To deal with such problems, it is necessary to turn to external optimization packages that can handle explicit constraints. Fortunately, `TSOpt`'s modular design allows easy linkage with external optimization packages via the `Jet` object. Chapter 5 provides a specific example of how the `Jet` object links `TSOpt` to the optimization software `IPOpt` (“Interior-Point Optimizer”). `IPOpt` [Wächter, 2002] is open-source software designed to solve large-scale nonlinear optimization problems, and is capable of handling nonlinear equality and inequality constraints. `IPOpt` uses an interior-point method to generate search directions for the nonlinear optimization problem, then applies a filter linesearch globalization scheme.

## Chapter 5

# The Black Oil Equations and the Optimal Well Rate Allocation Problem

In this chapter, I introduce the *Black-Oil Equations*, which are equations used to model fluid flow in reservoirs. The Black-Oil Equations stem from the *phase continuity* equations, which capture simultaneous, physical fluid flow behavior of up to three immiscible phases (namely: water, oil and gas). The Black-Oil Equations assumes that no mass transfer behavior between the water phase and the other phases occur, and is often used to model low-volatility oil systems [Peaceman, 1977]. As part of my dissertation, I implement a Black-Oil reservoir simulator in the `TSOpt` framework. Given a finite-volume discretization in space, I use time-stepping algorithms

to numerically solve the semi-discretized equations in time. I begin with the fixed time-step formulation, then move to the adaptive time-step algorithm.

The latter part of this chapter discusses the “Optimal Well Rate Allocation Problem” (OWRA), a reservoir engineering problem, which I formulate here as an optimization problem implicitly constrained by the Black-Oil equations. OWRA deals with profit maximization of a reservoir, by adjusting well rates over time. I will present a more precise mathematical statement in the second part of this chapter. Solving OWRA is the main target application of my thesis, and I intend to show that using adaptive time stepping techniques for optimal control problems is advantageous over the fixed time-step approaches. Before highlighting formulations and algorithms for adaptive time stepping for OWRA, however, I will present algorithms for the fixed time-step formulation.

## 5.1 The Phase Continuity Equations

I begin this chapter by introducing the phase continuity equations, and by explaining the physical significance of its components. Let  $\Omega \in \mathbb{R}^2$  be an open set, let  $x \in \Omega$  and let  $t \in [0, T]$ . Considering aqueous and liquid (oil with possible solution gas) phases the phase continuity equations which the Black Oil Equations stem from can



be written as:

$$\nabla \cdot \left[ \frac{\rho_l(t,x)K(x)k_{rl}(t,x)}{\mu_l(t,x)} (\nabla p_l(t,x)) \right] - q_l(t,x) - \frac{\partial(\phi(t,x)\rho_l(t,x)S_l(t,x))}{\partial t} = 0 \quad (5.1)$$

$$\nabla \cdot \left[ \frac{\rho_a(t,x)K(x)k_{ra}(t,x)}{\mu_a(t,x)} (\nabla p_a(t,x)) \right] - q_a(t,x) - \frac{\partial(\phi(t,x)\rho_a(t,x)S_a(t,x))}{\partial t} = 0, \quad (5.2)$$

where the subscripts  $a$  and  $l$  respectively refer to the aqueous and liquid phase,  $\rho$  is the fluid density,  $K$  is the absolute permeability of the medium,  $k_r$  is the relative permeability,  $\mu$  is the fluid viscosity,  $p$  is the pressure,  $q$  is taken to be the *mass rate* of production (if it is negative) or injection (if it is positive) per unit volume of the reservoir,  $\phi$  is the rock porosity, and  $S$  denotes the saturation (on a scale from 0 to 1). Since we consider two phase flow, the liquid and aqueous saturation must together fill the reservoir, hence implying:

$$S_l + S_a = 1. \quad (5.3)$$

We can further simplify the phase continuity equations (5.1) using Darcy's velocity approximation, which is an empirical law describing low to moderate flow of fluids through porous media. Darcy's law can be written as:

$$v_\theta(t,x) = -K(x) \frac{k_{r\theta}(t,x)}{\mu_\theta(t,x)} \nabla p_\theta(t,x) = -K(t,x) \lambda_\theta(t,x) \nabla p_\theta(t,x), \quad (5.4)$$

where  $\theta$  denotes a fluid phase and  $\lambda$  denotes the phase mobility. Substituting (5.4) into the phase continuity equations (5.1) yields:

$$\nabla \cdot v_l(t, x) - q_l(t, x) - \frac{\partial(\phi(t, x)\rho_l(t, x)S_l(t, x))}{\partial t} = 0 \quad (5.5)$$

$$\nabla \cdot v_a(t, x) - q_a(t, x) - \frac{\partial(\phi(t, x)\rho_a(t, x)S_a(t, x))}{\partial t} = 0. \quad (5.6)$$

Further, assuming the rock porosity  $\phi$  and the density  $\rho$  is time-invariant (i.e., the rock and fluid are incompressible), and normalizing the phase density yields:

$$\nabla \cdot v_l(t, x) - q_l(t, x) - \phi \frac{\partial S_l(t, x)}{\partial t} = 0 \quad (5.7)$$

$$\nabla \cdot v_a(t, x) - q_a(t, x) - \phi \frac{\partial S_a(t, x)}{\partial t} = 0, \quad (5.8)$$

which we consider as the incompressible two-phase Black-Oil equations.

## 5.2 Solving the Black Oil Equations

Wiegand [2010] solve equations (5.7) - (5.8) using the finite volume method. Using finite volume analysis, they derive two equations: the *pressure equation* and the *saturation equation*. Denoting the disjoint, compact subdomains of  $\Omega$  as  $\Omega_i$ , each with its own boundary  $\partial\Omega_i$ , we can express the pressure equation as:

$$- \int_{\partial\Omega_i} K(\lambda_l + \lambda_a) \nabla p \cdot \mathbf{n} dS = \int_{\Omega_i} q_l + q_a dv. \quad (5.9)$$

The saturation equation can be written as:

$$\left(\phi \frac{\partial s_a}{\partial t}\right)_i \cdot |\Omega_i| - \int_{\partial\Omega_i} K \lambda_a \nabla p \cdot \mathbf{n} dS = \int_{\Omega_i} q_a dV. \quad (5.10)$$

The next two sections reveal the discretization of the pressure and saturation equations in space, and in time. Wiegand [2010] give a thorough treatment of the derivation, associated Neumann boundary conditions, as well as a discussion of the solution properties of the pressure and saturation equations. They are presented here to clarify design decisions I make in implementing a Black-Oil simulator in TSOpt.

### 5.2.1 Discretizing the Pressure Equation in Space

The discrete form of the pressure residual equation takes following form:

$$\sum_{j \in \text{neighbor}(i)} K_{i,j} \lambda_{t,i,j} \frac{\Delta p_{i,j}}{l_{i,j}} A_{i,j} = \int_{\Omega_i} q_t dv = q_i, \quad (5.11)$$

where  $j$  being a neighbor of  $i$  implies that the volumes  $\Omega_j$  are adjacent to the volume  $\Omega_i$ , the total phase mobility  $\lambda_t = \lambda_a + \lambda_l$ , the change in pressure  $\Delta p_{i,j} = p_i - p_j$ , the length between the barycenter of the cells  $i$  and  $j$  are denoted as  $l_{i,j}$  and the area of the face between two cells are denoted as  $A_{i,j}$ . Defining the transmissibility as

$$T_{i,j} = \frac{K_{i,j} A_{i,j}}{l_{i,j}}, \quad (5.12)$$

we may simplify the discretized pressure residual equation as

$$g(t, s_a(t), p(t), q(t))_i = \sum_{j \in \text{neighbor}(i)} (T_{i,j} \lambda_{t_{i,j}} \Delta p_{i,j}) - q_i, \quad (5.13)$$

which we put into matrix form as

$$g(t, s_a(t), p(t), q(t)) = q - Ap. \quad (5.14)$$

In (5.14), the matrix  $A$  is constructed in the following manner:

$$A_{i,j} = -T_{i,j} \lambda_{t_{i,j}} \quad A_{i,i} = \sum_j T_{i,j} \lambda_{t_{i,j}}. \quad (5.15)$$

## 5.2.2 Discretizing the Saturation Equation in Space

The discrete form of the saturation equation can be written as the following:

$$\frac{1}{\phi_i \cdot |\Omega_i|} \left( q_{a_i} - \sum_{j \in \text{neighbor}(i)} T_{i,j} \lambda_{t_{i,j}} \Delta p_{i,j} \right) \approx \left( \frac{\partial s_a}{\partial t} \right)_i. \quad (5.16)$$

We can express the equation above as:

$$f(t, s_a(t), p(t), q(t)) = D^{-1}(q - \tilde{A}p) = \frac{\partial s_a}{\partial t}, \quad (5.17)$$

where the matrices  $D$  and  $\tilde{A}$  are defined in the following manner:

$$D_{i,i} = \phi_i \cdot |\Omega_i| \quad (5.18)$$

$$\tilde{A}_{i,j} = -T_{i,j} \lambda_{a_{i,j}} \quad \tilde{A}_{i,i} = \sum_j T_{i,j} \lambda_{a_{i,j}}. \quad (5.19)$$

Note that (5.17) is an ordinary differential equation, and we may choose a variety of schemes to solve it. However, it is most common in industry to use the backward Euler scheme – an implicit one-step scheme – due to its stability properties and its low computational cost. Also, the ordinary differential equation above is often referred to as “semi-discretized”, as the finite volume approach has discretized the equation in space, but not time.

### 5.2.3 Fixed Time Stepping for the Semi-Discretized Equations

There are also many possible approaches to solving the discretized pressure and saturation equations. Peaceman [1977] offers a more detailed survey of solution strategies for the saturation and pressure equations. In this thesis, I only focus on the so-called *coupled-implicit* approach, which implies solving (5.14) and (5.17) simultaneously. This approach, though incurring a larger computational cost, is preferred due to its numerical stability.

Using the coupled-implicit approach manifests itself as a nonlinear system of equa-

tions. Assuming fixed time-steps, and setting the primary variables as the pressure  $p^{k+1}$  and the aqueous saturation  $s_a^{k+1}$  yields:

$$\begin{bmatrix} q^{k+1} - Ap^{k+1} \\ D^{-1}(q^{k+1} - \tilde{A}p^{k+1}) \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{s_a^{k+1} - s_a^k}{\Delta t} \end{bmatrix}. \quad (5.20)$$

We must solve (5.20) at every time step (i.e. for  $k = 0, 1, \dots, N$ , where  $N = T/\Delta t$ ).

### 5.3 The Optimal Well-Rate Allocation Problem

There are various optimal control problems that can be posed using the Black-Oil equations, such as history matching and optimal well-rate allocation (OWRA). History matching entails attempting to verify and improve a model by comparing the model's output with historical field data, while optimal well-rate allocation attempts to find the best pumping and injecting rates for reservoir wells over a certain time window. In this thesis, I focus on OWRA. Solving optimal well-rate allocation via optimal control is not a new topic; previous attempts have been made by Ramirez [1987], Brouwer and Jansen [2004] and Sarma and Aziz [2005], for example. I solve the problem posed by Wiegand [2010], that finds the optimal well rate that will maximize revenue from oil production, while penalizing water injection and production:

$$\min_{q_i} J(q) = \int_0^T dt \left( \sum_{i \in P} \alpha(1 - s_a)q_i(t) + \sum_{i \in P} \frac{\beta}{2} s_a q_i^2(t) + \sum_{i \in I} \gamma q_i(t) \right), \quad (5.21)$$

where  $q_i$  are the well rate at the  $i$  is an index representation a location in the domain,  $I$  is a set of indices that correspond to injecting wells,  $P$  is a set of indices that correspond to producing wells,  $\alpha, \beta$  and  $\gamma$  are scalar variables and the aqueous pressure  $p$  and aqueous saturation  $s_a$  solve:

$$-\nabla \cdot (K(x)\lambda_{tot}(s_w(x, t))\nabla p(x, t)) = \sum_{i \in P} (1 - s_a)q_i(t)\delta(x - x_i) \quad (5.22)$$

$$+ \sum_{i \in PUI} s_a q_i(t)\delta(x - x_i) \quad (5.23)$$

$$\phi(x)\frac{\partial}{\partial t}s_a(x, t) - \nabla \cdot (K(x)\lambda_a(s_a(x, t))\nabla p(x, t)) = \sum_{i \in PUI} s_a q_i(t)\delta(x - x_i) \quad (5.24)$$

$$+ \text{B.C.s.} \quad (5.25)$$

Recall that in the equation above,  $K$  represents permeability,  $\lambda$  represents phase mobility and  $\phi$  represents rock porosity. We incorporate explicit equality and inequality constraints on the well rates to model the physical limitation of the wells.

## 5.4 The Fixed Time-Step Approach for OWRA

Considering a fixed-time stepping scheme for the saturation equations, the fully discretized optimal control problem then takes the form of:

$$\min \quad J_{\Delta t}(q) = \Delta t \sum_{k=1}^N l(t^k, s^k, q^k) \quad (5.26)$$

$$s.t. \quad e^T q^k = 0 \quad (5.27)$$

$$q_{min} \leq q^k \leq q_{max}, \quad (5.28)$$

where  $s^{k+1}$  and  $p^{k+1}$  solve:

$$\begin{bmatrix} g(t^{k+1}, s_a^{k+1}, p^{k+1}, q^{k+1}) \\ f(t^{k+1}, s_a^{k+1}, p^{k+1}, q^{k+1}) \end{bmatrix} = \begin{bmatrix} q - Ap^{k+1} \\ D^{-1}(q_a - \tilde{A}p^{k+1}) \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{s_a^{k+1} - s_a^k}{\Delta t} \end{bmatrix}. \quad (5.29)$$

Note that here the steplengths and state variables have superscript indices, to be consistent with the notation in [Wiegand, 2010]. Wiegand [2010] derive the adjoint equations from the optimality conditions using the ‘‘Discretize-then-Optimize’’ approach, and arrive at the following adjoint evolution scheme. For  $k = N - 1, \dots, 1$ , simultaneously solve for the adjoint variables  $\lambda_s^k$  and  $\lambda_p^k$  in the following equation:

$$-\frac{\lambda_s^{k+1} - \lambda_s^k}{\Delta t} = D_s f(\dots^k)^T \lambda_s^k - D_s g(\dots^k)^T \lambda_p^k - \nabla_s l(\dots^k) \quad (5.30)$$

$$0 = -D_p f(\dots^k)^T \lambda_s^k + D_p g(\dots^k)^T \lambda_p^k. \quad (5.31)$$



The directional derivative can then be obtained from the following expression:

$$\nabla J(q)\delta q = \sum_{k=1}^N \Delta t [\nabla_q l(\cdot^k) - D_{q^k} f(\dots^k)^T \lambda_s^k + D_{q^k} g(\dots^k)^T \lambda_p^k]^T \delta q^k. \quad (5.32)$$

## 5.5 Adaptive Time-Stepping for OWRA

In this part of the chapter, I discuss the derivation and algorithmic developments needed to solve OWRA with adaptive time stepping. First, I review some possible adaptive time-stepping schemes used for the Black-Oil equations and I discuss the adaptive time-stepping scheme I use to solve OWRA. Then, I address how to handle the control parameters via interpolation for OWRA.

### 5.5.1 Adaptive Time Stepping for the Black-Oil Equations

Typically, adaptive time-stepping algorithm have two phases: the “trial-step” phase and the “correction” phase. In the trial-step phase, some a-posteriori error estimate is established. If this error is greater than the user-specified tolerance, we restrict the size of the time-step and reject the step. In the correction phase, the step is tried again at the smaller step length. If the error estimate, on the other hand, is much less than the user-specified tolerance, we accept the step and increase the size of the step length. One of the popular adaptive time-stepping algorithms is based on embedded Runge-Kutta schemes.

Reservoir engineers, however, have adopted an alternative strategy for changing

time steps in the Black-Oil simulation. M.R. Todd [1972a,b] first proposed using the change in pressure and aqueous saturation (between two consecutive time steps) as a criterion for changing the step length. Before describing Todd's time-step selection logic, we introduce the following terms:

$p_{lim}$  = Maximum pressure changes desired

$s_{lim}$  = Maximum saturation changes desired

$p_{max}$  = Maximum pressure change calculated during previous time-step

$s_{max}$  = Maximum saturation change calculated during previous time-step

The scheme can be described as the following:

$$\Delta t_p = \Delta t^n \frac{p_{lim}}{p_{max}} \quad (5.33)$$

$$\Delta t_s = \Delta t^n \frac{s_{lim}}{s_{max}} \quad (5.34)$$

$$\Delta t^{n+1} = \min(\Delta t_p, \Delta t_s). \quad (5.35)$$

It is clear that controlling  $p_{lim}$  and  $s_{lim}$  affects the truncation error of the time stepping scheme, since  $p_{lim} \rightarrow 0$  and/or  $s_{lim} \rightarrow 0$  implies  $\Delta t \rightarrow 0$ . From experience, however, I found two problems with this alternate approach for adaptive reservoir simulation. First, straight-forward implementation of the scheme above leads to erratic changes in timestep length and large timestep values. A second problem is that, without further reservoir engineering expertise, it is difficult to determine what a good value

for  $p_{lim}$  and  $s_{lim}$  should be. It would be more advantageous to specify the desired error tolerance. For these two reasons, I decided to use the classical method of adaptive time-stepping for the Black-Oil equations.

In choosing an adaptive time-stepping scheme for the Black-Oil simulator, however, it would be ideal if we found a scheme that:

1. Is absolutely stable to avoid excessively small timesteps
2. Does not require making many structural changes to the simulator software.

This exactly was the topic of the work by Kavetski et al. [2002]. The authors consider solving an ODE system of the form

$$M \frac{d\theta}{dt} + K\theta = F, \quad (5.36)$$

motivated by finite element analysis. (In fact, in their work  $M, K$  and  $F$  are global finite element matrices, where  $M$  is the mass matrix,  $K$  is the conductivity matrix and  $F$  is a forcing term.) The authors established an adaptive backward Euler scheme that is based upon the weighed Euler difference family for the ODE system (5.36):

$$[M + \phi \Delta t K^{n+\phi}] V^{n+\phi} = -K^{n+\phi} \theta^n + F^{n+\phi} \quad (5.37)$$

$$\theta^{n+1} = \theta^n + \Delta t V^{n+\phi}, \quad (5.38)$$

where  $V^{n+\phi}$  is an  $O(\Delta t)$  approximation of  $(\frac{d\theta}{dt})^{n+\phi}$ ,  $K^{n+\phi} = K(\theta^n + \phi \Delta t V^{n+\phi})$  and  $\phi \in [0, 1]$ . Note that when  $\phi = 1$ , (5.37 – 5.38) yields the backward Euler scheme.

Kavetski et al. then compared (5.37 - 5.38) to variable-step variable-order (VSVO) scheme proposed by Thomas and Gladwell, which can be written as the following:

$$[\varphi_2 M + \varphi_3 \Delta t K] \dot{\theta}^{n+1} = [-(1 - \varphi_2) M \dot{\theta}^n - (\varphi_1 - \varphi_3) \Delta t K \dot{\theta}^n] - K \theta^n + F^{n+\varphi_1} \quad (5.39)$$

$$\theta^{n+1} = \theta^n + \frac{1}{2} \Delta t (\dot{\theta}^n + \dot{\theta}^{n+1}), \quad (5.40)$$

for  $\varphi_1, \varphi_2, \varphi_3 \in [0, 1]$ . The Thomas-Gladwell scheme is unconditionally stable for  $2\varphi_3 \geq \varphi_1 > \frac{1}{2}$  and  $\varphi_2 \geq \frac{1}{2}$ . Further, the Thomas-Gladwell scheme is  $O(\Delta t^2)$  convergent when  $\varphi_1 = \varphi_2$ .

Kavetski et al. noted that with  $\varphi_1 = \varphi_2 = \varphi_3 = 1$ , equations (5.39 - 5.40) are identical to equations (5.37 - 5.38) with  $\phi = 1$ , which yields the following scheme.

$$[M + \Delta t K^{n+1}] \dot{\theta}^{n+1} = -K^{n+1} \theta^n + F^{n+1} \quad (5.41)$$

$$\theta_{(1)}^{n+1} = \theta^n + \Delta t \dot{\theta}^{n+1} \quad (5.42)$$

$$\theta_{(2)}^{n+1} = \theta^n + \frac{1}{2} \Delta t (\dot{\theta}^n + \dot{\theta}^{n+1}). \quad (5.43)$$

We note that using the update (5.42) transforms (5.41 - 5.43) to the backward Euler scheme, while using the update (5.43) turns (5.41 - 5.43) into a (second order) Adams-Moulton method. We then obtain a measure of the local error by subtracting the first order step from the second order step, which can in turn be used as a criteria to adapt the steplengths.

Since the Black-oil saturation equation is of the form

$$\frac{dSa}{dt} = f(\dots), \quad (5.44)$$

we can take  $M = I$  and  $K = 0$  in (5.41), implying:

$$\theta_{(1)}^{n+1} = \theta^n + \Delta t F^{n+1} \quad (5.45)$$

$$\theta_{(2)}^{n+1} = \theta^n + \frac{1}{2} \Delta t (F^n + F^{n+1}). \quad (5.46)$$

(Note that we are left with the backward Euler scheme and the trapezoid rule, both of which are A-stable schemes.) Kavetski et al.'s scheme can be applied to the current Black-Oil simulator with little alteration. The extra work would go towards computing the local error estimate, for which we need the value of  $\theta^n = F^n$ . Further, local extrapolation is possible by adding the error estimate to the first order solution, providing a second order approximation to the solution. One potential drawback of Kavetski et al.'s scheme, however, is that the time-step sizes chosen by the algorithm are small in comparison to the time step sizes found in M.R. Todd [1972a,b]. Although more accurate, the scheme found in [Kavetski et al., 2002] leads to longer simulation times, and hence long inversion times.

## 5.5.2 Handling the Control Parameters for OWRA

Recall that, for the class of optimal control problems I consider, the control parameter is not time dependent. This implies that we require a mapping that takes the non-time-dependent controls  $q \in \mathbb{R}^n$  to  $\tilde{q}$ , which I define to be the control at time  $t$ . This may or may not necessitate use of interpolation schemes, depending on how the control parameter is interpreted. In my approach to solving OWRA, I define the controls on a fixed (time) grid, and use an interpolation scheme to provide a control over the entire time interval. To get accurate interpolation results over the entire simulation timespan, however, every time level of the simulation timespan must correspond to a control. The tradeoff for this requirement is that, in adding an extra control parameter, we add to the size of the numerical gradient and the amount of parameters we must optimize over. For example, for a 100-day simulation with 20-day timesteps, the controls will be defined on the time levels  $\{0, 20, 40, 60, 80, 100\}$ , instead of  $\{20, 40, 60, 80, 100\}$ . The algorithm I use to interpolate the controls is a piecewise-linear scheme. I justify this in two ways:

1. Since we use a first-order method to solve the ODE (recall we use a Backward-Euler scheme), it is natural to match the order of interpolation with that of the time-stepping algorithm's.
2. Interpolating with higher order polynomials (of order 2 and above) can lead to interpolated controls that violate bound constraints, even though the interpolation nodes themselves satisfy such constraints. This is particularly problematic

since such violations cause problems for the simulation (e.g., injecting wells turning into producing wells) and the constrained optimization (e.g., measurement of constraint violations, convergence, etc.).

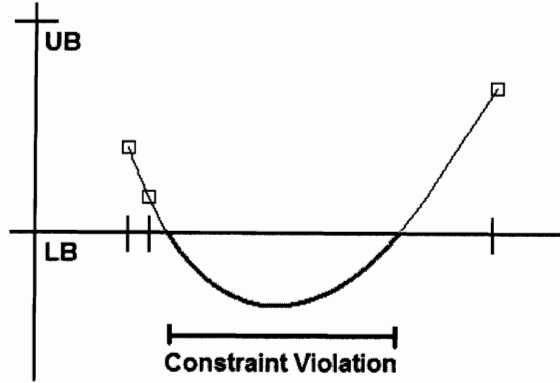


Figure 5.1: Example of how interpolated values can violate the bound constraint. UB and LB represent the upper and lower bounds, respectively. The squares represent interpolation nodes, which satisfy the bound constraints.

In fact, the second item enforces the first item: since the highest order interpolating scheme we can use for the optimal well-rate allocation problem is of the first order, the highest order time-stepping scheme we can use must also be a first order scheme.

### 5.5.3 Algorithmic Development of an Adaptive Black-Oil Simulator

After discussing various issues regarding adaptive time-stepping schemes and interpolation, I may now present the algorithm for the adaptive Black-Oil simulator. I begin by presenting algorithm 3, which runs the reference Black-Oil simulation with adaptive time-stepping. Note the safeguards placed in the algorithm to prevent large

changes in the time-step length, and how a failed step leads to a different strategy for how the next steplength is determined.

Next, I present the algorithm for the adaptive adjoint evolution for the Black-Oil equations. Recall that the adjoint evolution requires a reference state that is defined at the same adjoint time level. Due to adaptive time stepping, however, it is likely that the reference and adjoint time grids become mismatched. Hence, we must be able to interpolate the reference states. The interpolation scheme we use, however, depends on how the reference simulation states were stored. I now present the three strategies for handling the reference states during the forward simulation, which are adaptations of the strategies presented for the fixed time-stepping.

The first strategy is to save none of the reference states during the forward simulation. Hence, we rely solely on evolution to access the proper simulation state for the adjoint evolution. In this case, the forward evolution must always simulate to the next time level in the adjoint simulation. This removes the need for interpolating the reference states, though this incurs a large computational cost.

The second strategy is to save all of the reference states, and use them as necessary during the adjoint evolution. If we use this approach, we must choose all (or a subset) of the reference states as interpolation nodes. The resulting interpolating function is then evaluated at the time needed by the adjoint evolution. This approach incurs a huge storage cost for large problems, and it also introduces an interpolation error in the computation of the reference states.



The third strategy is to use adaptive checkpointing, which requires saving a subset of reference states. This was discussed in the previous chapter. The performance of this approach matches that of the “save-all” strategy and only requires  $n + 1$  extra state buffers and  $\log(N)$  recomputation, where  $n$  is the order of the time-stepping scheme and  $N$  is the total number of time steps to be taken. The adjoint Black-Oil algorithm 4 is compatible with the reference state storage strategies I discussed above.

## 5.6 Implementation in TSOpt

Solving the Black-Oil equations using TSOpt requires four things:

- a state type that is capable of holding the primary variables (aqueous pressure and saturation), that uses the `RealTime` class to store the time
- a “stack” class that handles storage of the state history, if we choose to use a checkpointing scheme for the adjoint computation
- Step classes, capable of internally changing its steplength parameter, that define *one step* of the forward and the adjoint evolution
- A software package for interpolation. Currently, TSOpt uses the `Spline` package, a collection of C++ functions that implement various approximation algorithms – such as divided differences and various splines [Burkardt, 2007].

Hence, I have implemented a state class called `RealBOState` that holds a pressure field, a saturation field (both as vectors from the standard library), and a `RealTime`

object to keep track of the time. There is also a stack class called `BOStack` that saves and accesses the pressure and saturation histories to file. I also created the step classes `Adapt_Fwd_BO_Dyn` and `Adapt_Adj_BO_Dyn` to execute one step of the algorithms 3 and 4, respectively (i.e., the part of the algorithm inside the `while` loop). Appropriate `LoopAlgs` keep the `Step` classes iterating, while `Terminators` check to see that the current time  $t$  is less than or equal to the desired final simulation time,  $T$ .

I then created a `Sim` object, which is composed of an appropriate `Terminator` object and an `Adapt_Fwd_BO_Dyn` object. In turn, this `Sim` object, along with an `Adapt_Adj_BO_Dyn` class object, was used to create a `jet` object. After construction, we may test the forward evolution and the adjoint evolution by issuing the following commands:

```
jet<...> myJet(...);  
myJet.getFwd().run();  
myJet.getAdj().run();
```

Currently, the three types of `Sim` classes I mentioned, which handled storage strategy of the simulation states (the “forgetful”, “remember-all” and checkpointing `Sim`), work with the adaptive Black-Oil simulator.

## 5.7 Using IPOpt to Solve OWRA

IPOpt is a software package designed to deal with large-scale nonlinear optimization problems of the form

$$\min_{x \in \mathbb{R}^n} f(x) \tag{5.54}$$

$$\text{s.t.} \quad g_L \leq g(x) \leq g_U \tag{5.55}$$

$$x_L \leq x \leq x_U, \tag{5.56}$$

where  $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$  is the objective function, and  $g(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$  are the constraint functions. The vectors  $g_L$  and  $g_U$  denote the lower and upper bounds on the constraints, and the vectors  $x_L$  and  $x_U$  are the bounds on the variables  $x$ . Wächter [2009] notes that the functions  $f(x)$  and  $g(x)$  can be nonlinear and non-convex, but should be twice continuously differentiable. IPOpt implements an interior-point line-search filter method in order to solve the problem above. This problem formulation is appropriate when solving OWRA, since not only does OWRA feature bound constraints on the optimization variables (i.e., the well rate constraints), it also has the pressure condition.

In order to use IPOpt, users have to provide a concrete implementations of the functions of the class TNL below. (A call to an `optimize` function will then run the optimization algorithm.) Note that, for brevity, I exclude the inputs to the functions. More information regarding the functions below can be found in [Wächter, 2009].

```

class TNLP {
public:
    /** Method to return some info about the nlp */
    virtual bool get_nlp_info( ... );

    /** Method to return the bounds for my problem */
    virtual bool get_bounds_info( ... );

    /** Method to return the starting point for the algorithm */
    virtual bool get_starting_point( ... );

    /** Method to return the objective value */
    virtual bool eval_f( ... );

    /** Method to return the gradient of the objective */
    virtual bool eval_grad_f( ... );

    /** Method to return the constraint residuals */
    virtual bool eval_g( ... );

    /** Method to return:
     * 1) The structure of the jacobian (if "values" is NULL)
     * 2) The values of the jacobian (if "values" is not NULL)
     */
    virtual bool eval_jac_g( ... );

    /** Method to return:
     * 1) The structure of the hessian of the lagrangian (if "values" is NULL)
     * 2) The values of the hessian of the lagrangian (if "values" is not NULL)
     */
    virtual bool eval_h( ... );
};

```

Interfacing with IPOpt is quite simple given TSOpt's jet class. For example, by allowing the subclassed TNLP class to own a jet object j, the implementation of the eval\_grad\_f function is written as the following:

```

void eval_grad_f( ..., std::vector<double> & grad ) {
    j.getFwd().run();
    j.getAdj().run();
    j.getAdj().getGrad(grad); // getGrad is an extra method endowed
                               // to the B0 Simulator, passes values
                               // by reference
}

```

IPOpt can also be used to perform the numerical experiments to validate the theory established in this dissertation. Recall that when considering adaptive time-stepping, the theory discussed in chapter 3 states that the time-stepping tolerances must be set to the KKT error of the current optimization iteration, in order to achieve convergence. IPOpt also provides an interface called `intermediate_callback` that allows access to such information in between major optimization iterations. Given this extra interface, I establish the algorithm 5 to perform optimization with adaptive time-stepping.

---

**Algorithm 3:** Adaptive Reference Black-Oil Equation Simulation (IMPSAT Formulation)

---

Let  $h^0$ , and the tolerance  $\tau$  be given.

Also, let the controls  $q \in \mathbb{R}^n$  be defined

Set  $k = 0, t = 0.0$ .

**while**  $t < T$  **do**

a) Define the function  $\hat{q}(t)$  to extend the control  $q$  to the interval  $[0, T]$ :

$$\hat{q}(t) = \sum_{i=1}^n q_i \chi_i(t), \quad (5.47)$$

where  $\chi_i$  is an basis function for the interval  $[\frac{T(i-1)}{n}, \frac{Ti}{n}]$ .

b) Define  $\tilde{q} = \hat{q}(t + h^k)$

c) Obtain  $p^{k+1}$  and  $s^{k+1}$  by solving the following, using Newton's Algorithm:

$$\begin{bmatrix} \tilde{q} - Ap^{k+1} \\ h^k D^{-1}(\tilde{q} - \tilde{A}p^{k+1}) - (s_a^{k+1} - s_a^k) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (5.48)$$

d) Set  $t = t + h^k$

e) Compute truncation error estimate  $e^k$  by taking the difference of (5.45) and (5.46). Take the relative norm  $\|e^k\|_r$ .

f) **if**  $\|e^k\| < \tau$  **then**

Set

$$h^k = h^k \times \min \left( 0.9 \sqrt{\frac{\tau}{\max(\|e^k\|_r, \text{EPS})}}, 4.0 \right) \quad (5.49)$$

**end**

**else**

Set

$$h^k = h^k \times \max \left( 0.9 \sqrt{\frac{\tau}{\max(\|e^k\|_r, \text{EPS})}}, 0.1 \right) \quad (5.50)$$

Go to step (c)

**end**

g) Set  $k = k + 1$

**end**

---

---

**Algorithm 4:** Adaptive Adjoint Black-Oil Equation Simulation (IMPSAT Formulation)

---

Let  $h^0$  and tolerance  $\tau$  be given.

Also, let the controls  $q \in \mathbb{R}^n$  be defined

Set  $k = 0, t = T$ .

**while**  $t > 0$  **do**

a) Define the function  $\hat{q}(t)$  to extend the control  $q$  to the interval  $[0, T]$ :

$$\hat{q}(t) = \sum_{i=1}^n q_i \chi_i(t), \quad (5.51)$$

where  $\chi_i$  is an basis function for the interval  $[\frac{T(i-1)}{n}, \frac{Ti}{n}]$ .

b) Define  $\tilde{q} = \hat{q}(t + h^k)$

c) Compute  $p^*$  and  $s^*$ , which approximate the pressure and saturation at time  $t - h^k$

d) Obtain  $\lambda_p^{k+1}$  and  $\lambda_s^{k+1}$  by solving the following linear system:

$$\begin{bmatrix} D_s f(\tilde{q}, p^*, s^*)^T & -D_s g(\tilde{q}, p^*, s^*)^T \\ -D_p f(\tilde{q}, p^*, s^*)^T & D_p g(\tilde{q}, p^*, s^*)^T \end{bmatrix} \begin{bmatrix} \lambda_s^{k+1} \\ \lambda_p^{k+1} \end{bmatrix} = \begin{bmatrix} \frac{\lambda_s^k - \lambda_s^{k+1}}{h^k} + \nabla_s l(\tilde{q}, p^*, s^*) \\ 0 \end{bmatrix}$$

e) Set  $t = t - h^k$

f) Compute truncation error estimate  $e^k$  by taking the difference of (5.45) and (5.46). Take the relative norm  $\|e^k\|_r$ .

g) **if**  $\|e^k\| < \tau$  **then**

Set

$$h^k = h^k \times \min \left( 0.9 \sqrt{\frac{\tau}{\max(\|e^k\|_r, \text{EPS})}}, 4.0 \right) \quad (5.52)$$

**end**

**else**

Set

$$h^k = h^k \times \max \left( 0.9 \sqrt{\frac{\tau}{\max(\|e^k\|_r, \text{EPS})}}, 0.1 \right) \quad (5.53)$$

Go to step (d)

**end**

h) Set  $k = k + 1$

**end**

---

---

**Algorithm 5:** Interior-Point Optimal Control Solver, with Adaptive Time Stepping

---

```
A. Set initial (optimization) tolerance  $tol$ 
B. Set spacing for (uniform) control grid,  $\Delta q$ .
C. Set initial time-stepping tolerance  $\tau_0$ 
D. Set initial control vector  $q^0$ , defined on the grid  $\Delta q$ .
for  $n = 0, 1, 2, \dots$  do
    1. Solve reference equations adaptively with tolerance  $\tau$ , using algorithm 3
    2. Solve adjoint equations with tolerance  $\tau$  using algorithm 4
    3. Form numerical gradient, as to align with the control grid  $\Delta q$ 
    4. Pass function evaluation and numerical gradient to IPOpt
    5. Obtain current NLP error (estimate)  $\epsilon_{eNLP}$  from IPOpt
    6. Set  $\tau_{k+1} = \min\{\tau_k, \epsilon_{eNLP}^2\}$ 
    7. if  $\epsilon_{eNLP} < tol$  then
        | Exit Algorithm
    end
    else
        | Continue, and set  $k = k + 1$ .
    end
end
```

---



# Chapter 6

## Numerical Results

This chapter presents the numerical results that verify the theory established in this thesis. I first present unconstrained optimization inversion results. I solve a control problem taken from Kelley and Sachs [1999] using the inexact Newton algorithm discussed in Chapter 3, with the modifications I discussed to adjust the adaptive time-stepping tolerance parameter. The second part of this chapter presents the tests for the Black-Oil simulator, and the constrained optimization inversion results. I solve the optimal well-rate allocation problem using the inexact interior point methods. Before proceeding, I would also like to note that for these numerical tests, contrary to the mathematical background, the evaluation of the objective function was obtained via adaptive quadrature, whose node placements were solely determined by the reference equations. In other words, I follow the problem formulation (1.1), instead of the transformed problem (1.4). Further, I perform the gradient accumulation separate from the adjoint evolution, by using regular (i.e., non-adaptive) quadrature. Due to

how the control variable was interpreted (see Algorithm 3, for example), gradient accumulation needed to be performed over a regular time grid. All the numerical results in this chapter were obtained using a 2.16 GHz Intel Core 2 duo machine with 3 gigabytes of RAM.

## 6.1 Unconstrained Optimization Test

I consider the following control problem:

$$\min_u \int_0^1 (y - 3)^2 + 0.01u^2, \quad (6.1)$$

where  $u \in \mathcal{L}^\infty[0, 1]$  and  $y(t)$  solves the following initial value problem

$$\frac{dy}{dt} = yu + t^2 \quad y(0) = 0. \quad (6.2)$$

The adjoint equations corresponding the the objective function and state equation above is

$$\frac{d\lambda}{dt} = -(\lambda u + 2(y - 3)) \quad \lambda(1) = 0. \quad (6.3)$$

Given the adjoint variable  $\lambda(t)$ , the derivative of the objective function  $\nabla f(u)(t)$  with respect to the  $\mathcal{L}^2$  inner product can be expressed as

$$\nabla f(u)(t) = \lambda y + 0.02u. \quad (6.4)$$

Similar to the approach used by Kelley and Sachs [1999], I approximate the objective function using a simple quadrature rule, and the discretized control  $u$  is treated as a piecewise linear spline with 10 equidistant nodes, and the unknown are the values at the nodes. To interpolate the state variables (for the adjoint evolution), I also use a piecewise linear spline.

To solve this control problem, I couple `TSOpt` with the time-stepping software package in the GNU Scientific Library [Galassi and Theiler, 2009]. To further test `TSOpt`'s functionality, I use the “forgetful” `Sim` (meaning that none of the reference states were saved) coupled with `GSL`'s implementation of the implicit Gear's method. I then interface the time-stepping code with the LBFSGS algorithm implemented in the `UMin` package in `RVL`.

The results I show in this chapter highlight the advantage of updating the algorithmic tolerances adaptively. For the problem (6.1), I compare the optimization results for two cases. The first case only has a fixed algorithmic tolerance,  $\tau = 0.01$ . The second case feature tolerance updating as described in the mathematical theory

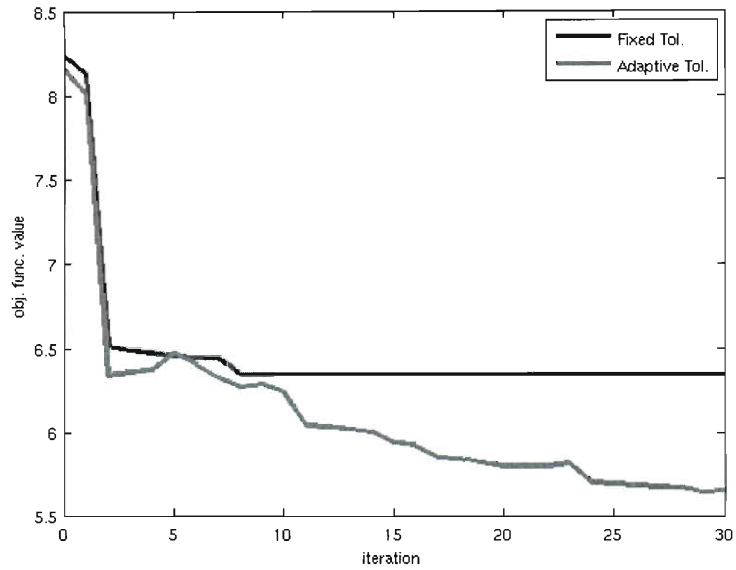


Figure 6.1: Objective function plot, for both the fixed and adaptive tolerance schemes. Note the stagnation produced by the fixed-tolerance scheme.

section of this thesis:

$$\tau_{k+1} = \min(\tau_k, \kappa \|g_k\|^2), \quad (6.5)$$

with  $\tau_0 = 0.5$ . Figure 6.1 plots the objective function and figure 6.2 plots the norm of the scaled gradients. Note that the adaptive tolerance scheme produces lower objective function values and gradient-norms. The values of the tolerances (in a  $\log -y$  plot) can be viewed in figure 6.3. Note that the fixed-tolerance algorithm stagnates, as that version of the algorithm no longer produces a descent direction after the eighth optimization iteration.

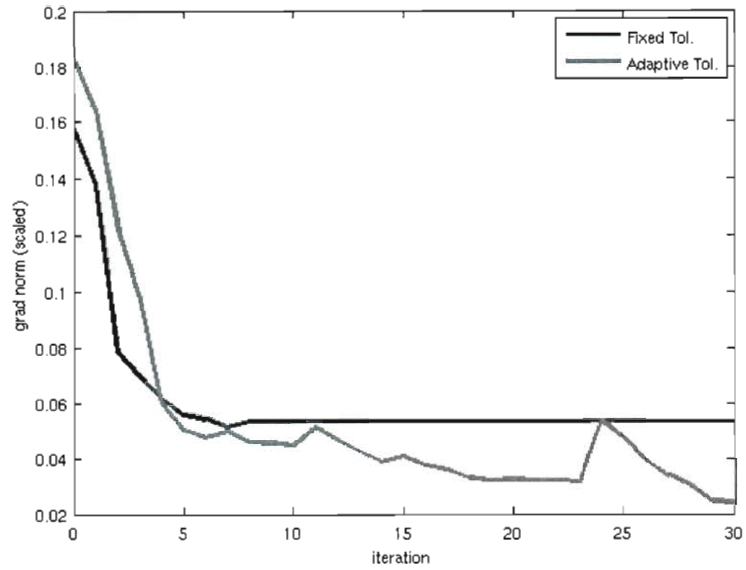


Figure 6.2: Gradient-norm plot, for both the fixed and adaptive tolerance schemes. Again, note the stagnation produced by the fixed-tolerance scheme.

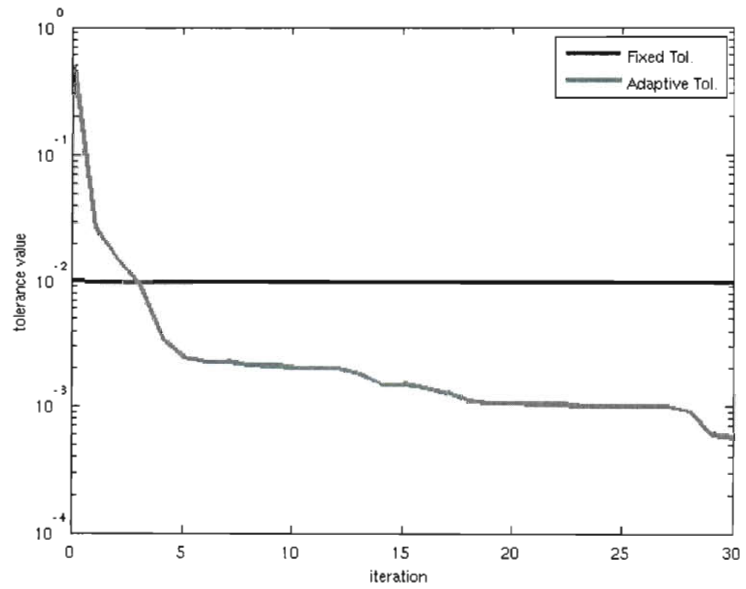


Figure 6.3: Tolerance values, for both the fixed and adaptive tolerance schemes. Note that the y-axis is on a logarithmic scale.

## 6.2 Black Oil and Constrained Optimization Tests

I now present numerical results for the Black-Oil simulator. The numerical results here are split into two parts: fixed time-step results, and adaptive time-step results. When presenting the fixed time-step results, I show the output of the forward simulation. I also check the quality of the gradient produced from the adjoint-state method by comparing to the finite difference approximation. Though not the focus of my thesis, I show fixed time-stepping results to verify that the simulator in its fixed-step form functions correctly. I also present inversion results for OWRA using fixed time-steps to highlight the fundamental problem with fixed time-stepping for inversions; this allows me to segue to the presentation of adaptive time-stepping results. I highlight differences between the fixed-step simulations for both the forward and adjoint evolution. I also present inversion results, and show how the inversion benefits from adaptive time-stepping.

### 6.2.1 Tests for Fixed Time Steps

#### Checking the Forward Simulation

This section provides numerical results from the Black Oil simulator implemented in `TSOpt`. Namely, in this section I show results for the forward simulator, and I highlight gradient convergence for a sample objective function constrained by the Black Oil equations.

For the simulations considered in this thesis, we use a 2D regular grid of size

220 × 60. Porosity and permeability data come from the top layer of the SPE10 model. The SPE10 model dimensions are 1200 × 2200 (ft). The fine scale cell size is 20 × 10 (ft). The source and sink terms (which correspond to injecting and producing wells in this example) are configured according to figure 6.4 (on the right).

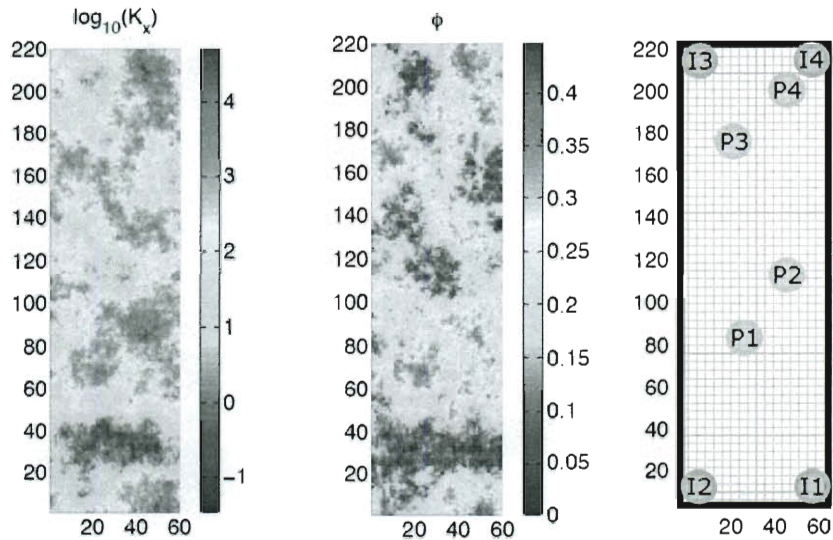


Figure 6.4: [l] Porosity and permeability plot of the SPE10 model, top layer; [r] Placement of injector (I) and producer (P) wells in the domain.

Given the porosity, permeability, and source/sink data, the results of the 100-day simulation can be seen in figure (6.5). Note how the water saturation is high where the injectors are located, hence the higher water saturation around the corners of the above figure. It should be noted that these figures were generated using MATLAB, using the simulation data obtained from the C++ simulation.

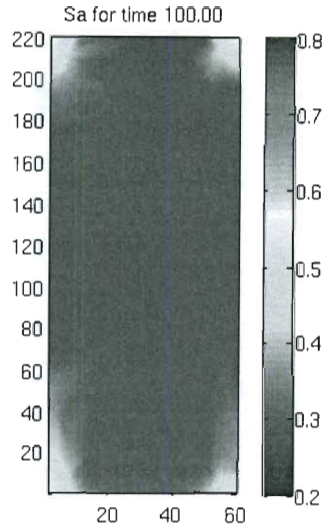


Figure 6.5: Plot of Aqueous Saturation at  $t = 100$  days, with  $dt = 25$ .

### Checking the Adjoint States and Gradient Formulation

We can test the quality of the adjoint states by considering the quality of the derivative of an objective function with respect to its controls.

We check the quality of  $\nabla J(q)$  by using the mathematical definition of a directional derivative: for  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  and a direction  $\delta x$ , the directional derivative  $f'(x)[\cdot]$  must satisfy

$$\lim_{h \rightarrow 0} \frac{f(x + h\delta x) - f(x) - hf'(x)[\delta x]}{h} = 0. \quad (6.6)$$

Suppose rewrite (6.6) as:

$$\lim_{h \rightarrow 0} \frac{f(x + h\delta x) - f(x)}{h} - f'(x)[\delta x] = 0. \quad (6.7)$$



We note that the first component is a finite difference approximation to the derivative. From this observation, we can test the gradient from the adjoint-state method by subtracting it from the finite difference approximation, for decreasing values of  $h$ . Wiegand et. al divides this difference by the value of the objective function, to produce the relative gradient error. The following graph shows the results of this test: We see that the difference between the computed gradient and the finite difference

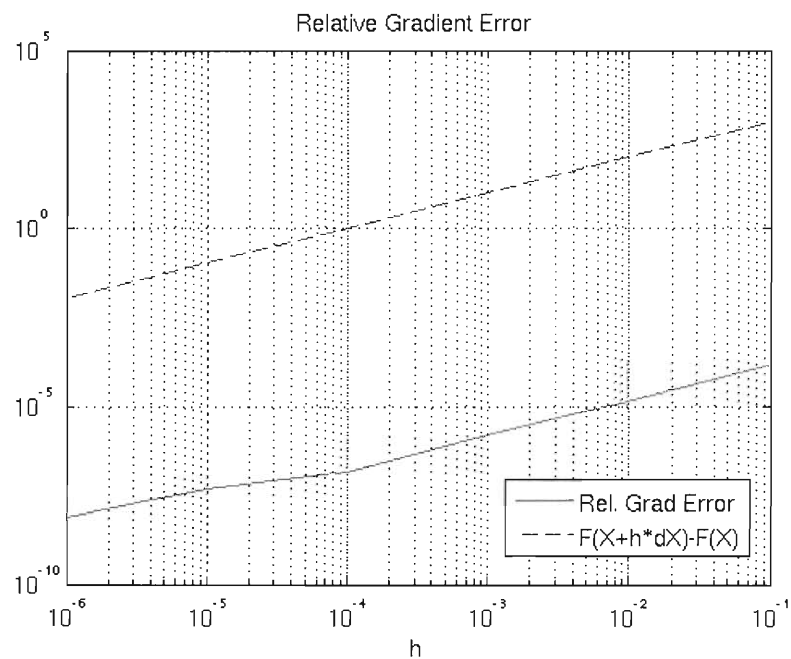


Figure 6.6: Plot of the difference between the computed gradient via the adjoint-state method, and the finite difference approximation.

approximation gets closer as  $h$  gets smaller, which is the behavior we expect to see. A natural next step, since we have a verified gradient, is to couple the simulation results with an optimization framework.

## Inversion Results

In this experiment, I attempt to solve OWRA in a 200-day window, with a fixed time-step  $h = 20$  days, using the software package IPOpt. The initial guess passed to the optimization algorithm is a constant rate of injection and production, all set to 10 bbl/day. Each well rate must stay within the range  $[0, 20]$  bbl/day. Production rates are assigned a negative value and an injection rate is assigned a positive value. The stopping tolerance is set to  $5e - 2$ , or a 5% NLP error. Looking at the plot of producers and injectors, figure (6.4), we see that producing well 4 and injecting well 4 are deemed “too close” to one another. After some time, the water that placed into the reservoir by injector 4 will immediately be ejected by producer 4, implying a waste of resources. Hence, we expect to see the optimizer to throttle the rates for either producing well 4 or injecting well 4. Given this foresight into the problem, I now present the results for OWRA using fixed time-steps in figures (6.8) and (6.7).

Iter.	Objective Function	Constr. Violation	$g - A_E^T(q)y - A_I^T(q)z$	LS Calls
0	-4.2662536e+05	0.00e+00	4.40e-01	0
1	-4.3471903e+05	3.55e-15	3.65e-01	1
2	-5.7138241e+05	5.33e-15	2.60e-01	1
3	-5.9096253e+05	8.88e-15	1.49e-01	1
4	-6.0666657e+05	7.11e-15	1.13e-01	1
5	-6.1676339e+05	5.33e-15	1.85e-01	1
6	-6.1945346e+05	5.33e-15	1.84e-01	1

Table 6.1: Optimization report generated by IPOpt, using fixed time-stepping.

Though these results are promising, there is one major problem: due to the inaccu-

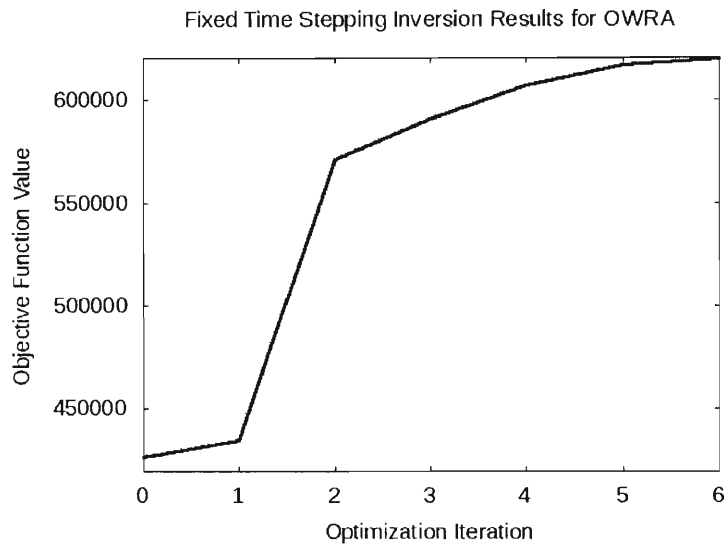


Figure 6.7: Objective function for the fixed time-stepping approach to solving OWRA. Note that the iterations stop at 8, as the Black-Oil simulator crashes after the eighth iteration due to accumulation of numerical errors.

racies accumulating from the process of (fixed) time-stepping, the simulator actually crashes after the sixth major optimization iteration. The inaccurate simulations led to computed pressure and saturation variables that violated physical bounds (e.g., negative pressures). This, in turn, led to a singular Jacobian matrix. (Recall that the Jacobian matrix was needed for the backward-Euler step in the reference simulation.) It is true that the simulator breakdown might be avoided by choosing a smaller time-step; however, it is impossible to determine *a-priori* how small the time-step needs to be. An excessively big time-step will lead to simulator breakdown, while an excessively small time-step will lead to longer simulations, and hence longer inversions. (Since inversion is a time-consuming process, the “trial-and-error” approach to selecting time-step lengths is not desirable.) As I will show in the next section, such

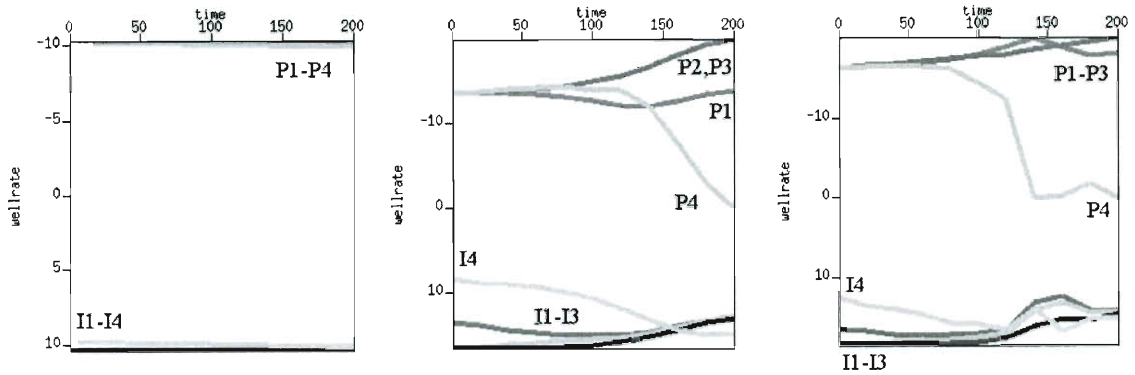


Figure 6.8: Progression of the control parameter for the first [l], third [m] and sixth [r] optimization iteration, taking fixed time steps. Note the well labels on the figure: “P” represents the producing wells and “I” represents the injecting wells.

problems do not occur when using adaptive time stepping to solve OWRA.

## 6.2.2 Tests for Adaptive Time Steps

I now discuss the results for the adaptive time-stepping approach to solving OWRA. I employ the various algorithms and strategies discussed in this chapter (e.g., which scheme to perform the adaptation, how to handle controls, etc.). All inversions were completed using the IPOpt optimization package.

### Inversion Results

Similar to the fixed time-step case, I consider optimizing the well-rates for OWRA over a 200-day window. Controls are defined on a uniform grid, with  $\Delta q = 20$  days. The error tolerance for the optimization algorithm is set to  $5e-2$ . The initial tolerances for the forward and the adjoint evolution was set to  $\tau = 0.5$ . The initial guess passed

to the optimization algorithm is a constant rate of injection and production, all set to 10 bbl/day. Each well rate must stay within the range  $[0, 20]$  bbl/day.

Figure 6.9 shows the objective function for both the adaptive and fixed-step simulations for OWRA. Note that the objective function for the adaptive simulations are higher (by roughly 2.5 percent), and actually converge according to the tolerance specified. The succession of controls seen in 6.10 show that adaptive simulations used for inversions not only lead to shutting the fourth producer, but *also* throttling the fourth injector during the first few days. Figure 6.11 plots the tolerances chosen per each optimization iteration, based on the scheme discussed in the mathematical theory chapter. Note that using the adaptive tolerance scheme, we were able to bring the KKT error to 5 percent, versus the 18.4 percent error for the fixed-step solution.

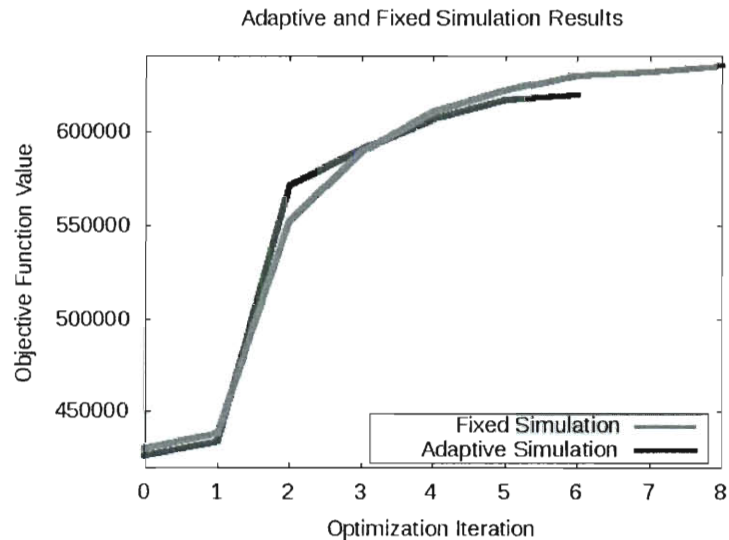


Figure 6.9: Objective function for the fixed (blue) and adaptive (red) time-stepping approach to solving OWRA. The difference in the objective function value is about 2.5 percent. The adaptive simulation approach also did not crash.

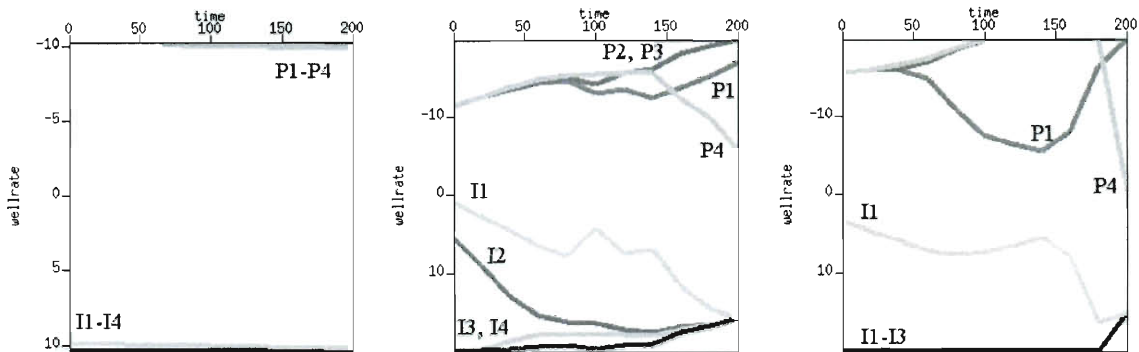


Figure 6.10: Progression of the control parameter for the first [l], fourth [m] and eighth [r] optimization iteration, using adaptive simulations. Note the well labels on the figure: “P” represents the producing wells and “I” represents the injecting wells.

Iter.	Objective Function	Constr. Violation	$g - A_E^T(q)y - A_I^T(q)z$	LS Calls
0	-4.3098630e+05	0.00e+00	4.29e-01	0
1	-4.3890821e+05	3.55e-15	3.69e-01	1
2	-5.5241292e+05	3.55e-15	2.36e-01	1
3	-5.8961775e+05	4.44e-15	2.70e-01	1
4	-6.1046177e+05	5.33e-15	2.09e-01	1
5	-6.2190984e+05	3.55e-15	1.55e-01	1
6	-6.3005662e+05	6.22e-15	1.13e-01	1
7	-6.3212067e+05	5.33e-15	6.98e-02	1
8	-6.3474929e+05	3.55e-15	4.56e-02	1

Table 6.2: Optimization report generated by IPOpt, using adaptive time stepping.

It is clear for the figures above that the adaptive time-stepping approach to solving OWRA is superior. Higher objective function values were attained (recall that we are maximizing) and the simulator did not crash. Notice also the extra insight gained from the adaptive solution regarding how to manage the fourth injector and producer. Finally, I present comparative results of how long it takes the fixed and adaptive

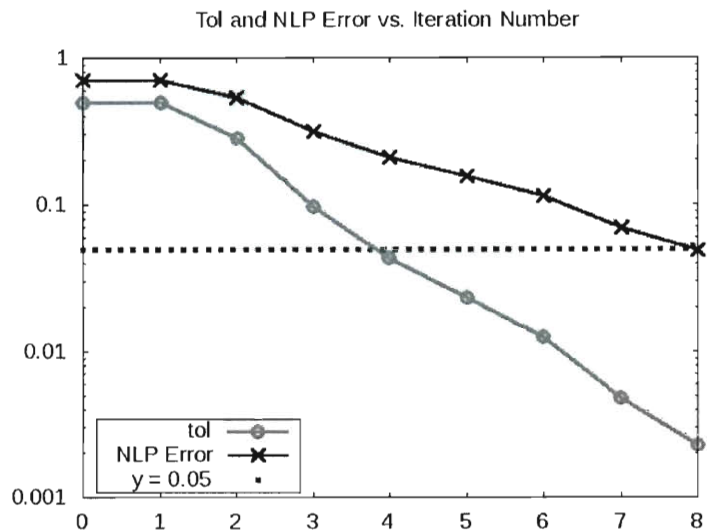


Figure 6.11: Plot of the value of the tolerances and NLP errors versus the major optimization iteration. Note that the y-axis is on a log-scale.

NLP Stop Tolerance	Fixed-Step Alg.	Adaptive Alg
1.1e-1	9+ hours, $\Delta t = 0.25$ days	3 hours
5.0e-2	[Failed]	4.5 hours

Table 6.3: Time comparisons for the fixed time-step approach and the adaptive approach, to reach a specified NLP error tolerance.

simulations to reduce the NLP error to a specified value. Note that the adaptive algorithm runs faster than its fixed-step analogue when the NLP stopping criteria was set to  $1.1e-1$ . The fixed-step approach failed when the NLP stopping criteria was set to  $5.0e-2$ , due to a memory error from IPOpt. (Further refinement of the control grid led to a huge number of optimization variables, which strained IPOpt.)

# Chapter 7

## Future Work and Conclusion

In this chapter, I discuss some future directions for this research, as well as summarize my doctoral work. The three topics I believe should be examined in the future include changing the tolerance-updating scheme, considering inexact Trust-Region (TR) methods and consideration of discontinuous ODE constraints for the optimal control problem.

To begin discussion of the future work, I would first like to address different tolerance-updating schemes. Recall that the tolerance updating scheme I use in this thesis is of the form

$$\tau_{k+1} = \min(\tau_k, \kappa \|\nabla g_k\|^2), \quad (7.1)$$

where  $\tau_k$  and  $g_k$  denote the tolerance and the computed gradient at the  $k^{\text{th}}$  iteration, respectively, and  $\kappa$  is some scaling constant. It is possible that this strategy aggres-



sively decreases the tolerance, leading to longer simulation times than necessary. The problem, however, is that there are no obviously superior alternative strategies to automate the tolerance-updating procedure. One option is to consider a different power on the gradient norm, e.g. consider the tolerance-updating scheme

$$\tau_{k+1} = \min(\tau_k, \kappa \|\nabla g_k\|^p), \quad (7.2)$$

for some  $p \in (1, 2)$ . This approach, however, will change the convergence rate of the algorithm. Another option is to implement some sort of “watch-dog” that monitors optimization progress, and decides when to decrease the tolerance depending on certain criteria. Such criteria can include objective function decrease, number of linesearch calls, etc.

One possible way to pursue less aggressive tolerance updating schemes is to consider the inexact Trust Region (TR) method. Recall that globalization schemes for the Newton method involve either the linesearch or the Trust-Region method. (I used the latter in this dissertation.) Hence, it is not surprising that the TR analogue of the inexact Newton methods exist, and are referred to as “inexact TR algorithms”. The most promising inexact TR algorithm is that of Heinkenschloss and Vicente [2001], which requires the following condition on the computed gradient to attain *lim-inf* convergence to a stationary point:

$$\|\nabla f(x_k) - g_k\| \leq C \min(\|g_k\|, \Delta_k). \quad (7.3)$$

In (7.3),  $\Delta_k$  is the trust-region radius and  $C$  is some positive constant. Given the assumptions on the gradient error in this thesis, namely,

$$\|\nabla f(x_k) - g_k\| \approx K\tau \quad (7.4)$$

for some constant  $K > 0$ , then we can use the tolerance update scheme

$$\tau_{k+1} = \min(\|g_k\|, \Delta_k) \quad (7.5)$$

to satisfy Heinkenschloss and Vicente's requirement (7.3). What is promising about this approach is that the tolerance update scheme incorporates the globalization parameter in a nice way. A poor approximation of the true gradient will lead to an inaccurate TR model function. In turn, this will lead to poor predicted model decrease, which triggers a shrinking of the TR radius  $\Delta$  and a retry of the optimization step. If the TR radius shrinks enough, it follows that  $\min\{\|g_k\|, \Delta_k\} = \Delta_k$ . From (7.3) we see that a small  $\Delta_k$  implies that the approximated gradient is close to the true gradient, in norm. Also, should we consider explicitly-constrained optimal control problems, a TR-SQP version of Heinkenschloss and Vicente's algorithm can be used. Heinkenschloss and Vicente [2001] also requires their TR-SQP to satisfy the bound (7.3) in order to achieve convergence to a stationary point.

The last topic I would like to examine is how to handle optimal control problems with discontinuous ODE constraints. There was a particular technique I employed

in my research that could be effective when dealing with discontinuous ODEs: the act of forcing the simulation to align on a certain grid. If the location (in time) of the discontinuity is known, we can force the time-stepping algorithm to align to that location instead of integrating over it, preserving the accuracy of the computed solution. This gain in accuracy should translate to more accurate derivatives for solving the optimal control problem. More theoretical and computational research needs to be performed in order to verify this claim.

In conclusion, I showed how beneficial it is to solve optimal control problems with adaptive time stepping in this dissertation. A huge body of literature only consider adaptive time-stepping for the reference equations, and using the reference time grid for the adjoint evolution. This is oft an erroneous assumption, as there is no guarantee that the adjoint dynamics will behave like the reference dynamics. Though adaptive time-stepping for both reference and adjoint fields lead to mismatched time grids, the gained accuracy in the solution of the differential equations makes a big difference in the optimization results. I proved this claim theoretically by relating the discretization error of my approach to the residual vector in the inexact Newton method. I verified the theory I established by using the software framework `TSOpt`. `TSOpt` simplifies the process of attempting various numerical approaches for solving optimal control problems. `TSOpt`'s modular structure aided the development of my "Adaptive checkpointing" algorithm, an extension of `AREvolve` [Hinze and Sternberg, 2005] that is capable of handling adaptive time steps in the reference and adjoint

fields. Further, by using `TSOpt`, I was able to solve complex problems such as the “Optimal Well-Rate Allocation Problem”, and highlight the improvement of adaptive time-stepping approach, over fixed time-step methods, for optimal control problems.

# Appendix A

## Adaptive Checkpointing Algorithm

The algorithm below shows how to manage `AREvolve` and an interpolation buffer such that the checkpointing algorithm will work for cases where the adjoint equations are solved adaptively in time. This algorithm features a “lock” on the function call to `AREvolve`, which opens and closes depending on certain conditions that depend on the evolution and the interpolation buffer. As seen in [Enriquez, 2008], this algorithm features a “Forward Mode” and a “Backward Mode” for efficiency. The extra cost of using the adaptive checkpointing algorithm solely comes from the interpolation buffer, which requires  $n + 1$  buffers, where  $n$  is the order of the time-stepping scheme being used for evolution. The interpolation buffer is taken to be a wrapper to a `deque` object, so that pushing and popping from the front and back of the buffer only incurs an  $O(1)$  cost.

---

**Algorithm 6:** Adaptive Checkpointing Algorithm (Using ARevolve and Interpolation Buffer)

---

```
if forwardMode then
  while action != youturn do
    action = revolve.run()
    if action == advance then
      | run reference step
      | update interpolation buffer
    end
    if action == takeshot then
      | update checkpointing buffer
    end
  end
  forwardMode = false
  revolveLock = false
end
else
  Let  $t$  = time requested by adjoint simulation
  while  $t > t_0$  do
    if revolveLock = false then
      | action = revolve.run()
    end
    if action == advance then
      | run reference step
    end
    if action == restore then
      | load proper state from checkpoint buffer
    end
    if action == takeshot then
      | save state into a checkpoint buffer slot
    end
    if action == youturn then
      if  $t$  is in the time-span of interpolation buffer then
        | use interpolation buffer to approximate state at time  $t$ 
        | revolveLock = true
      end
      else
        | revolveLock = false
        | update interpolation buffer
      end
    end
  end
  forwardMode = true
end
```

---

# Bibliography

- S. Bellavia. Inexact interior-point method. *Journal of Optimization Theory and Applications*, 26:109–121, 1998.
- R. W. Brankin, I. Gladwell, and L. F. Shampine. RKSUITE: A suite of explicit Runge-Kutta codes. In *Contributions in Numerical Mathematics*, pages 41–53. World, 1993.
- D.R. Brouwer and J.-D. Jansen. Dynamic optimization of waterflooding with smart wells using optimal control theory. *SPE Journal*, pages 391–402, 2004.
- J. Burkardt. Spline: Interpolation and approximation of data, 2007. URL <http://people.sc.fsu.edu/~burkardt>. Data Accessed: 9.19.09.
- R. Carter. On the global convergence of trust region algorithms using inexact gradient information. *SIAM J. Numerical Analysis*, 28:251–25, 1991.
- T. Coffey. Rythmos: Transient integration of differential equations. <http://software.sandia.gov/trilinos/packages/docs/dev...>

- </packages/rythmos/doc/html/index.html>, 2009. Date Accessed: October 1, 2009.
- A. Conn, N. Gould, and P. Toint. *Trust-Region Methods*. SIAM, Philadelphia, PA, USA, 2000.
- A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-Free Optimization*. SIAM, Philadelphia, 2009.
- S. Dembo and T. Steihaug. Inexact newton methods. *SIAM Journal on Numerical Analysis*, 19:400–408, 1982.
- S. Eisenstat and H. Walker. Globally convergent inexact newton methods. *SIAM J. Optimization*, 4:393–422, 1994a.
- S. Eisenstat and H. Walker. Choosing the forcing terms in an inexact newton method. *SIAM Journal on Scientific Computing*, 1994b.
- M. Enriquez. A C++ class supporting adjoint state methods. Master’s thesis, Rice University, 2008.
- M. Enriquez and W. Symes. A user’s guide to TSOpt (“time-stepping for optimization”) software. Technical report, Rice University, 2009.
- M. Galassi and J. Theiler. Gnu scientific library. <http://www.gnu.org/software/gsl/>, 2009. Date accessed: 3/25/09.



- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1998.
- M. Gockenbach, W. Symes, D. Reynolds, and P. Shen. Efficient and automatic implementation of the adjoint state method. *ACM TOMS*, 28(1):22–24, 2002.
- A. Griewank and A. Walther. Revolve: An implementation of checkpointing of the reverse or adjoint mode of computational differentiation. *ACM TOMS*, 26:19–45, 2000.
- W. Hager. Runge-Kutta methods in optimal control and the transformed adjoint system. *Numerische Mathematik*, 87:247–282, 1999.
- J. Hahn. *Introduction to the Theory of Nonlinear Optimization*. Springer-Verlag, 2nd edition, 1996.
- M. Heinkenschloss. Numerical solution of implicitly constrained optimization problems. Technical Report TR08–05, Department of Computational and Applied Mathematics, Rice University, Houston, TX 77005–1892, 2008.
- M. Heinkenschloss and L. Vicente. Analysis of inexact trust-region SQP algorithms. *SIAM J. Optimization*, 12:283–302, 2001.
- A.C. Hindmarsh. ODEPACK, a systematized collection of ODE solvers. *Scientific Computing*, 1:55–65, 1983.
- M. Hinze and J. Sternberg. A-Revolve: An adaptive memory-reduced procedure

- for calculating adjoints; with an application to computing adjoints of instationary navier-stokes system. *Optimization Methods and Software*, 20:645–663, 2005.
- K.R. Jackson and R. Sacks-Davis. An alternative implementation of variable step-size multistep formulas for stiff ODEs. *ACM TOMS*, 6:295–318, 1980.
- D. Kavetski, P. Binning, and S. Sloan. Adaptive backward euler time stepping with truncation error control for numerical modlling of saturated fluid flow. *International Journal for Numerical Methods in Engineering*, 53:1301–1322, 2002.
- C. T. Kelley and E.W. Sachs. Truncated newton methods for optimization with inaccurate functions and gradients. *SIAM Journal on Optimization*, 10:43–55, 1999.
- D. Kincaid and W. Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. Brooks/Cole, 2002.
- U. Kirchgraber. Multi-step methods are essentially one-step methods. *Numerishe Mathematik*, 48:85–90, 1985.
- J.D. Lambert. *Numerical Methods for Ordinary Differential Equations: The Initial Value Problem*. Wiley & Sons, 2000.
- S. Li and L. Petzold. Adjoint sensitivity analysis for time-dependent partial differential equations with adaptive mesh refinement. *Journal of Computational Physics*, 198:310–325, 2004.

- J. L. Lions. *Optimal Control Of Systems Governed By Partial Differential Equations*. Springer Verlag, 1971.
- J.J. Moré. Recent developments in algorithms and software for trust region methods. In *Mathematical Programming State of The Art*. Springer-Verlag, 1982.
- G.J. Hiirasaki M.R. Todd, P.M. O'Dell. Methods for increased accuracy in numerical reservoir simulators. *SPE*, 253:515–530, 1972a.
- W.J. Longstaff M.R. Todd. The development, testing, and application of a numerical simulator for predicting miscible flood performance. *Journal of Petroleum Technology*, 3484:874–882, 1972b.
- J. Nocedal and S. Wright. *Numerical Optimization*. Springer, 1999.
- A. Padula, S. Scott, and W. Symes. A software framework for abstract expression of coordinate-free linear algebra and optimization algorithms. *ACM Trans. Math. Softw.*, 36(2):1–36, 2009. ISSN 0098-3500. doi: <http://doi.acm.org/10.1145/1499096.1499097>.
- D. Peaceman. *Fundamentals of Numerical Reservoir Simulation*. Elsevier, 1977.
- R.-E. Plessix. A review of the adjoint-state method for computing the gradient of a functional with geophysical applications. *Geophysical Journal International*, 167: 495–503, 2006.

- W. F. Ramirez. *Application of optimal control theory to enhanced oil recovery*. Elsevier Science Inc., 1987.
- J. M. Renders and S. Flasse. Hybrid methods using genetic algorithms for global optimization. *IEEE Transactions on Systems*, 26:243–258, 1996.
- P. Sarma and K. Aziz. Implementation of adjoint solution for optimal control of smart wells. *SPE*, 2005. Jan. 31.
- E. Süli and D. Mayers. *An Introduction to Numerical Analysis*. Cambridge University Press, 2003.
- W. Symes. A time-stepping library for simulation-driven optimization. Technical report, Rice University, TRIP, 2006.
- A. Wächter. *An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering*. PhD thesis, Carnegie Mellon University, 2002.
- A. Wächter. Short tutorial: Getting started with ipopt in 90 minutes. Technical report, IBM T.J. Watson Research Center, 2009.
- K. Wiegand. A numerical study of an adjoint based method for reservoir optimization. Master’s thesis, Department of Computational and Applied Mathematics, Rice University, Houston, TX, May 2010.