RICE UNIVERSITY

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

by

Marco U. Enriquez

A THESIS PROPOSAL 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

Danny Sorenson
Noah Harding Professor of Applied Mathematics

Matthias Heinkenschloss
Professor of Applied Mathematics

Daniel Cohan
Assistant Professor of Environmental Engineering

Houston, Texas

December 2009

Abstract

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

by

Marco U. Enriquez

The adjoint-state method is widely used for computing gradients in simulation-driven optimization problems. The adjoint-state evolution equation requires access to the entire history of the system states. However, problems arise when adaptive time-stepping schemes are used to perform the reference and adjoint simulation. 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. This proposal highlights the work completed to justify my claim. I discuss preliminary theoretical and computational results. The computational results feature an implementation of a reservoir simulator using TSOpt, a time-stepping library for simulation-driven optimization algorithms.

Contents

List of Figures

Chapter 1

Introduction

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

$$\min_{y,u} \quad f(y,u) = \int_0^T J(y(t),u)dt \tag{1.1}$$

$$\text{s.t.} \quad \frac{d}{dt}y(t) - H(y(t),u) = 0\,, \qquad t \in [0,T] \tag{1.2}$$

$$H, y \equiv 0 \text{ for } t < 0 \tag{1.3}$$

where the control $u \in \mathbb{R}^n$, the state $y \in C^1([0,T],Y)$ for a state Hilbert space Y, J is a functional, and $H : \mathbb{R}^n \times Y \to Y$ is some nonlinear dynamic operator. The equations (1.2) - (1.3) are often referred to as the "state equation". Throughout this proposal, numerical solution of the differential equation (1.2) will be referred to as a *forward simulation.* A forward simulation generates the solution 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.*

In order to use Newton-based optimization algorithms to solve the problem (1.1), it is necessary to calculate the gradient of the objective function $J(c)$ with respect to the controls, c. A common method to calculate the gradient of the objective function (1.1) is through the algorithm called the *adjoint-state method* [Lions, 1971].

Adjoint-state methods incur a cost roughly equivalent to the cost of numerically solving the differential equation (1.2) [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 "stiff" regions. Lambert [2000] defines stiffness to be the following:

> If a numerical method with a finite region of absolute stability, applied to a system with any initial conditions, is forced to use in a certain interval of integration a steplength which is excessively small in relation to the smoothness of

the exact solution in that interval, then the system is said to be **stiff** in that interval.

It would be ideal to take larger time-steps over the non-stiff regions, and to restrict the time-step size over the stiff regions. 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.

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 (1.1)? 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 present in the adjoint state calculation. However, having a controllable tolerance in the time-stepping algorithm means that the global error in the state equations' numerical solution can be changed. 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.

In this proposal, I highlight the research I have completed to answer the questions I posed 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 (1.1) can be guaranteed by manipulating the time-stepper's algorithmic parameters. Chapter 4 discusses the software framework I helped develop, called TSOpt ("Time-Stepping for Optimization"), which is the computational tool I use to verify the theory I established. Finally, Chapter 5 discusses the Black-Oil equations, and how I have implemented the reference and adjoint evolution for these equations in TSOpt. I intend to use this implementation to solve an optimal control problem whose gradients are obtained through adaptive time-stepping and the adjoint state method. My ultimate goal is to study how adjusting time-stepping tolerances and parameters will affect convergence to an optimal control, using optimization problems implicitly constrained by the Black-Oil equations as my target example.

Chapter 2

Literature Review

The goal of this proposal is to explore the effect of adaptive time stepping in simulation-driven optimization problems. This chapter will cover 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 proposal. I then dissect the simulation-driven optimization problem into two separate topics: in the second section, I discuss simulating via adaptive time-stepping methods while in the third section, I review existing optimization methods accommodating inexact information (e.g. inexact gradients). The second section also discusses software packages for numerically solving ordinary differential equations, using both fixed and adaptive time stepping methods.

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. Famous examples of derivative-free algorithms are stochastic algorithms, such as Genetic Algorithms or Simulated Annealing. Stochastic algorithms are theoretically attractive since they are capable of finding global minima [Sarma and Aziz, 2005]. 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 proposal, I will be focusing 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). "Optimize-then-Discretize" 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]. "Discretize-then-Optimize," alternatively, first discretizes the continuum problem, and then solves the (discrete) optimality conditions for the resulting finite dimensional problem. It should be noted that the strategy "Discretize-then-Optimize" is fundamentally simpler than "Optimize-then-Discretize" because it eliminates the need to analytically calculate derivatives of the continuous Lagrangian function.

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:

$$\min_u f(u) = \int_\Omega g(y,u)dx \tag{2.1}$$

where (y,u) solves the one dimensional heat equation:

$$y_t = y_{xx}\,, \tag{2.2}$$

with boundary conditions

$$y_x(0) = 0 \qquad y(1) = 1\,. \tag{2.3}$$

Note that we can use the boundary condition $y_x(0) = 0$ along with the ghost boundary point y_0 to deduce:

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

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

$$\begin{aligned}
\dot{y_1} &= \frac{2y_2 - 2y_1}{h^2} && (2.5)\\
\dot{y_i} &= \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}\,, \qquad i = 2, 3, \ldots N-1 && (2.6)\\
\dot{y_N} &= 0\,. && (2.7)
\end{aligned}$$

Given the adjoint variable λ, the corresponding adjoint to this discretization takes the following form:

$$\begin{aligned}
-\dot{\lambda}_1 &= \frac{\lambda_2 - 2\lambda_1}{h^2} && (2.8)\\
-\dot{\lambda}_2 &= \frac{2\lambda_1 - 2\lambda_2 + \lambda_3}{h^2} && (2.9)\\
-\dot{\lambda}_i &= \frac{\lambda_{i+1} - 2\lambda_i + \lambda_{i-1}}{h^2}\,, \qquad i = 3, 4, \ldots N-2 && (2.10)\\
-\dot{\lambda}_{N-1} &= \frac{-2\lambda_{N-1} + \lambda_{N-2}}{h^2} && (2.11)\\
-\dot{\lambda}_N &= \frac{\lambda_{N-1}}{h^2}\,. && (2.12)
\end{aligned}$$

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:

$$-\lambda_t = \lambda_{xx} \tag{2.13}$$

$$\lambda_x(0) = 0 \qquad\qquad \lambda(1) = 0\,. \tag{2.14}$$

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

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

$$-\dot{\lambda}_i = \frac{\lambda_{i+1} - 2\lambda_i + \lambda_{i-1}}{h^2}\,, \qquad i = 2, 3, \ldots N-1 \tag{2.16}$$

$$-\dot{\lambda}_N = 0\,. \tag{2.17}$$

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 equations 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 "Optimize-then-Discretize" is equivalent to the strategy "Discretize-then-Optimize".

It should also noted that it is possible to couple both "Optimize-then-Discretize" and "Discretize-then-Optimize" 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 "Discretize-then-Optimize" approach around the spatial domain boundary, then use the "Optimize-then-Discretize" 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 (1.1). 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 or to verify the output of the adjoint-state evolution operator. The gradient of the objective function is then used in Newton or quasi-Newton methods, which solves 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". However, unlike `TSOpt`, `Rythmos` does not currently support gradient calculation via adjoint-state methods.

2.2 Adaptive Time Stepping

In simulation-driven optimization problems, the differential equation constraint (1.2) 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 is stiff; 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. There are, however, two alternatives to using prohibitively small time steps: implicit fixed step time stepping methods and adaptive time stepping methods. I examine both approaches in this section.

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, Suli 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].

If the differential equation's solution has both stiff and non-stiff regions on the time domain of interest, it is advantageous to use adaptive time stepping methods. Adaptive time stepping methods allow the step lengths to vary while performing the evolution. Over the stiff regions, the algorithm can restrict the step length while in non-stiff regions, the algorithm can take larger time steps [Lambert, 2000, Suli and Mayers, 2003, Kincaid and Cheney, 2002]. It should be noted that both explicit and implicit schemes can be adaptive.

Embedded explicit Runge-Kutta (RK) methods are a popular example of an adaptive time stepping algorithm [Lambert, 2000, Suli 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, Suli 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] includes 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^{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:

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

for some nonnegative 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.

Further analysis of the Newton algorithm using inexact information can be found in [Kelley and Sachs, 1999]. Kelley and Sachs examine the unconstrained optimization problem

$$\min_x f(x)\,,$$

for a function $f : \mathbb{R}^n \to \mathbb{R}$, whose objective function evaluation and gradients are given by "black-box" codes and whose absolute and relative error are controllable. Kelley and Sachs [1999] 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' CGTR algorithm also uses a finite-difference scheme to approximate Hessian-vector computation. They note that for functions $f : \mathbb{R}^n \to \mathbb{R}$, Newton iterative methods' inner iterations terminate when

$$\|\nabla^2 f(x_k)s_k + \nabla f(x_k)\| \leq \eta \|\nabla f(x_k)\|\,, \tag{2.19}$$

where η is the forcing term. Due to the finite difference approximation and the CG iteration, Kelley and Sachs find that the condition (2.19) should be altered to

$$\|\nabla^2 f(x_k)s_k + \nabla f(x_k)\| \leq \bar{\eta}_1 \|\nabla f(x_k)\| + \xi_1\,, \tag{2.20}$$

where the constant $\bar{\eta}_1 = \eta + O(\delta^q)$ and the constant $\xi_1 = O(\tau)$. Note that δ represents the finite difference increment, q represents the order of accuracy of the finite difference approximation and τ represents the error from evaluating the function value and the gradient. Using the bound (2.20), as well as analysis of the quadratic model and measurement of decrease, Kelley and Sachs [1999] propose changes to the TR algorithm that accelerate convergence.

Like Kelley and Sachs [1999], Carter [1991] also uses an unconstrained optimization algorithm. Carter also considers solving $\min f(x)$, where $f : \mathbb{R}^n \to \mathbb{R}$ 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:

$$\min_s \quad \psi(x_k + s) \tag{2.21}$$

$$s.t. \quad \|D_k s\| \leq \Delta_k \,. \tag{2.22}$$

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 ∇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

$$\frac{\|g_k - \nabla f(x_k)\|_{(D_k^T D_k)^{-1}}}{\|g_k\|_{(D_k^T D_k)^{-1}}} \leq \xi \,, \tag{2.23}$$

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 (2.23) imposed a bound on the relative error from their algorithm. Carter asserts that if (2.23) is satisfied, then we have

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

(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

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

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. Furthermore, in general, it is impossible to have *a priori* knowledge of how to control the solution accuracy so that (2.23) is satisfied.

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:

$$\lim_{k\to\infty} \|g_k - \nabla f(x_k)\| = 0\,, \tag{2.24}$$

given a sequence $\{x_k\}$ that converges to a stationary point. Note that the condition (2.24) is equivalent to Carters condition (2.23) given the sequence of iterates $\{x_k\}$ converge 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,

$$\min \quad f(y,u) \tag{2.25}$$

$$\text{s.t.} \quad C(y,u) = 0\,, \tag{2.26}$$

for $f : \mathbb{R}^n \to \mathbb{R}$, $C : \mathbb{R}^n \to \mathbb{R}^m$, 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

$$\min_u f(y(u),u)\,, \tag{2.27}$$

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.

In this proposal, I will build on the mathematical and algorithmic framework established by Heinkenschloss and Vicente. Their algorithm will be discussed in greater detail in the next chapter.

Chapter 3

Mathematical Background

In this chapter I discuss the mathematical background necessary for my thesis work. I begin by discussing the adjoint state method and the optimal control problem. Since I am interested in solving optimal control problems with inexact gradients (with incurred from adaptive time-stepping and interpolation), I then discuss Trust-Region (TR) algorithms for the unconstrained optimization problem that use inexact gradient information. Heinkenschloss and Vicente [2001] derived a bound on the gradient error that, if satisfied, guaranteed the convergence of their TR-SQP algorithm. This bound also implies convergence of Moré's unconstrained TR algorithm [Moré, 1982]; I give a full proof of this result in this chapter.

3.1 The Optimal Control Problem and The Adjoint State Method

I begin by defining the optimal control problem considered in this proposal, which is of the form

$$\min_{y,u} \quad f(y,u) = \int_0^T J(y(t),u)dt \tag{3.1}$$

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

$$H, y \equiv 0 \text{ for } t < 0 \tag{3.3}$$

where the control $u \in \mathbb{R}^n$, the state $y \in C^1([0,T],Y)$ for a state Hilbert space Y, J is a functional, and $H : Y \times \mathbb{R}^n \to Y$ is some nonlinear dynamic operator that is continuously partially differentiable. In order to simplify notation, I define the following mapping, $C : [0,T] \times \mathbb{R}^n \to Y$, as:

$$C(y(t),u) = y(t) - \int_0^t H(y(s),u)ds. \tag{3.4}$$

Using this notation, we then write (3.1) as

$$\min_{y,u} \quad f(y,u) \tag{3.5}$$

$$\text{s.t.} \quad C(y(t),u) = 0\,, \qquad t \in [0,T] \tag{3.6}$$

If the following hypotheses are satisfied:

1. $C(y,u) = 0$ has a unique solution y for all $u \in \mathbb{R}^n$
2. There is an open set $D \subset \mathbb{R}^n \times Y$ with $\{(y,u) : u \in \mathbb{R}^n, C(y,u) = 0\} \subset D$, such that f and C are twice differentiable on D
3. $C_y(y,u)$ is invertible for all pairs $(y,u) \in \{(y,u) : u \in \mathbb{R}^n, C(y,u) = 0\}$,

then we can invoke the implicit function theorem. The implicit function theorem asserts that there exists a continuously differentiable function $u \mapsto y(u)$ that is defined through the solution of $C(y,u) = 0$. Using the implicit function theorem, we reduce (3.5) to

$$\min \hat{f}(u) = f(y(u),u)\,, \tag{3.7}$$

where $(y(u),u)$ solves the constraint equation. In order to use gradient-based optimization algorithms to solve (3.7), we will need a derivative with respect to the control parameter u; the adjoint-state method is one method to obtain this derivative.

The adjoint state method can be derived by taking the total derivative of $\hat{f}$:

$$\nabla \hat{f}(u) = -C_u(y(u),u)^T C_y(y(u),u)^{-T} \nabla_y f(y(u),u) + \nabla_u f(y(u),u)\,. \tag{3.8}$$

If we introduce the adjoint variable $\lambda(u) \in \mathbb{R}^{n_y}$, which solves

$$C_y(y(u),u)^T \lambda(u) = -\nabla_y f(y(u),u)\,, \tag{3.9}$$

we can rewrite (3.8) as

$$\nabla \hat{f}(u) = \nabla_u f(y(u), u) + C_u(y(u), u)^T \lambda(u)\,. \tag{3.10}$$

Computable approximations of (3.9) - (3.10) then take the form of

$$C_y(y_k, u_k)^T \lambda \;=\; -\nabla_y f(y_k, u_k) \tag{3.11}$$

$$\nabla f(u_k) \;=\; C_u(y_k, u_k)^T \lambda + \nabla_u f(y_k, u_k)\,, \tag{3.12}$$

where the index k represents the iteration number.

The size of the problem, however, may prohibit us from solving the discrete adjoint equations (3.11) exactly; in this case, we use iterative methods to solve the linear system to a user-specified accuracy. The error in the adjoint variable then manifests itself in the gradient, causing derivative inaccuracy. This incurred gradient error, if large enough in magnitude, can be problematic for gradient-based optimization algorithms because it can lead to bad search directions. Fortunately, there are many optimization algorithms that account for this derivative inaccuracy; one such algorithm was developed by Heinkenschloss and Vicente.

3.2 Inexact Trust Region (TR) Algorithms

Heinkenschloss and Vicente developed a modified Trust-Region SQP method which allows for inexactness in the gradient, caused by inexact linear system solves. Recall that the adjoint state method requires solution to linear systems of the form

$$C_y(y_k, u_k)^T \lambda = -\nabla_y f(y_k, u_k)\,. \tag{3.13}$$

Suppose, however, that we cannot solve (3.13) exactly. In this case, we incur a residual error e, and we introduce $\hat{\lambda}$ that now satisfies

$$C_y(y_k, u_k)^T \hat{\lambda} = -\nabla_y f(y_k, u_k) - e\,. \tag{3.14}$$

This residual error then manifests itself (as another type of error) when we construct the approximate gradient g_k using $\hat{\lambda}$ via the formula

$$g_k = C_u(y_k, u_k)^T \hat{\lambda} + \nabla_u f(y_k, u_k)\,. \tag{3.15}$$

Heinkenschloss and Vicente [2001] then show if the approximate gradient g_k satisfies the following inequality:

$$\|g_k - \nabla f(x_k)\| \leq K \min\{\|g_k\|, \Delta_k\}\,, \tag{3.16}$$

where $K > 0$ is some constant and Δ_k is the Trust Region radius at the k^{th} iteration, their Trust-Region SQP algorithm exhibits first-order global convergence. Heinkenschloss and Vicente, however, note that though K does not need to be less than 1, the absolute error in the reduced gradient error must be less than Δ_k and $\|g_k\|$. In this proposal, I focus on Heinkenschloss and Vicente's result for the unconstrained TR: if the approximate gradient satisfies (3.16), then global lim inf convergence of Moré's unconstrained TR algorithm (found in [Moré, 1982]) can be established. We see that the bound (3.16) guarantees that the approximate gradient is never "too far" from the true gradient; each approximate gradient satisfies the following two facts:

1. 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 Δ and a retry of the optimization step. If the TR radius shrinks enough, it follows that $\min\{\|g_k\|, \Delta_k\} = \Delta_k$. From (3.16) we see that a small Δ_k implies that the approximated gradient is close to the true gradient, in norm.

2. In the case that $\min\{\Delta_k, \|g_k\|\} = \|g_k\|$ and $K \in [0, 1)$, we recover Carter's condition (2.23), which guarantees that the approximated gradient points in the direction of the true gradient.

The remainder of this chapter is dedicated to clearly stating Moré's algorithm, and generalizing his convergence proof (while filling in specific details).

I begin by stating the TR algorithm in Moré's paper, Algorithm 1. Moré makes the following assumptions on the TR algorithm. First, Moré places a sufficient decrease condition on the step s_k, which requires that the decrease $\psi_k(s_k)$ must be some fraction of the optimal decrease in ψ_k along the Cauchy direction in the norm $\|D_k(\cdot)\|$. Mathematically, we impose this condition by requiring the existence of constants β_1,

Algorithm 1: Trust Region Algorithm

```
Let $x_0 \in \mathbb{R}^n$ and $\Delta_0$ be given. Set $0 < \mu < \eta < 1$ and $\gamma_1 < 1 < \gamma_2$.
for $k = 0, 1, \dots$ do
    a) Compute $f(x_k)$ and the model $\psi_k$
    b) Determine solution $s_k$ to TR subproblem, satisfying (3.19):

        $\min \quad \psi_k(s_k) = g^T s_k + s_k^T B_k s_k$        (3.17)
        $\text{s.t.} \quad \|D_k s_k\| \le \Delta_k$        (3.18)

    where $g_k$ approximates $\nabla f(x_k)$, $B_k$ is the Hessian approximation, and $D_k$ is
    a scaling matrix
    c) Compute $\rho_k = \frac{f(x_k+s_k)-f(x_k)}{\psi_k(s_k)}$
    d) If $\rho_k > \mu$ then we have a *successful iteration*. Set $x_{k+1} = x_k + s_k$
       Otherwise, we have an *unsuccessful iteration*. Set $x_{k+1} = x_k$.
    e) Update model $\psi_k$ and scaling matrix $D_k$[1]
    f) Update TR radius $\Delta_k$:
        1) if $\rho_k \le \mu$ then $\Delta_{k+1} \in (0, \gamma_1 \Delta_k]$.
        2) if $\rho_k \in (\mu, \eta)$ then $\Delta_{k+1} \in [\gamma_1 \Delta_k, \Delta_k]$
        3) if $\rho_k \ge \eta$, then $\Delta_{k+1} \in [\Delta_k, \gamma_2 \Delta_k]$
end
```

β_2 and v satisfying

$$\psi_k(s_k) \le \beta_1 \min\{\psi_k(w) : D_k^T D_k w = v g_k, \|D_k w\| \le \Delta_k\}, \quad \|D_k s_k\| \le \beta_2 \Delta_k . \tag{3.19}$$

Next let us define the scaled gradient approximation and the scaled approximated Hessian as

$$\hat{g}_k = D_k^{-T} g_k , \quad \hat{B}_k = D_k^{-T} B_k D_k^{-1} . \tag{3.20}$$

[1]There are a variety of schemes that can be applied to update the scaling matrix D_k. For example, given a user-chosen parameter ϵ, one could use the update

$$D_{k+1}(i,i) = \max\{0.6 D_k(i,i), \sqrt{\max\{|H_k(i,i)|, \epsilon\}}\},$$

as proposed by Moré [Conn et al., 2000].

Then, Moré second assumption states that there are constants σ_1 and σ_2 such that

$$\|\hat{B}_k\| \leq \sigma_1 \,, \quad \|\hat{D}_k^{-1}\| \leq \sigma_2 \,. \tag{3.21}$$

Using these assumptions we establish the following lemma, which will be crucial in proving convergence of the TR algorithm 1.

Lemma 1. *If s_k satisfies (3.19) and $\| \cdot \|$ is the ℓ_2 norm, then*

$$-\psi_k(s_k) \geq \frac{1}{2}\beta_1 \|\hat{g}_k\| \min\left\{\Delta_k, \frac{\|\hat{g}_k\|}{\|\hat{B}_k\|}\right\} \,,$$

where the scaled gradient approximation and the scaled approximated Hessian are defined as:

$$\hat{g}_k = D_k^{-T} g_k \,, \quad \hat{B}_k = D_k^{-T} B_k D_k^{-1} \,. \tag{3.22}$$

Proof. To begin, let us define a function $\varphi : \mathbb{R} \to \mathbb{R}$:

$$\varphi(\tau) = \psi_k \left[-\tau D_k^{-1} \left(\frac{\hat{g}_k}{\|\hat{g}_k\|} \right) \right] . \tag{3.23}$$

As a first step, let us rewrite the expression for $\varphi(\tau)$.

$$\begin{aligned}
\varphi(\tau) &= g_k^T \left[-\tau D_k^{-1}\left(\frac{\hat{g}_k}{\|\hat{g}_k\|}\right)\right] && (3.24)\\
&\quad + \frac{1}{2}\left[-\tau D_k^{-1}\left(\frac{\hat{g}_k}{\|\hat{g}_k\|}\right)\right]^T B_k \left[-\tau D_k^{-1}\left(\frac{\hat{g}_k}{\|\hat{g}_k\|}\right)\right] && (3.25)\\
&= -\tau\|\hat{g}_k\| + \frac{1}{2}\tau^2 \frac{\hat{g}_k^T \hat{B}_k \hat{g}_k}{\|\hat{g}_k\|^2}\,. && (3.26)
\end{aligned}$$

We introduce the variable $\mu_k = \frac{\hat{g}_k^T \hat{B}_k \hat{g}_k}{\|\hat{g}_k\|^2}$ to arrive at

$$\varphi(\tau) = -\tau\|\hat{g}_k\| + \frac{1}{2}\tau^2 \mu_k\,. \tag{3.27}$$

Let τ_k^* then be the minimum of φ on the interval $[0, \Delta_k]$. We now derive bounds on $\varphi(\tau_k^*)$. If $\tau_k^* \in (0, \Delta_k)$, then

$$\begin{aligned}
& \varphi'(\tau^*) = 0 && (3.28)\\
\Rightarrow\ & -\|\hat{g}_k\| + \tau^* \mu_k = 0 && (3.29)\\
\Rightarrow\ & \tau^* = \frac{\|\hat{g}_k\|}{\mu_k}\,, && (3.30)
\end{aligned}$$

and hence,

$$\varphi(\tau^*) = \frac{-\|\hat{g}_k\|^2}{2\mu_k} \le \frac{-\|\hat{g}_k\|^2}{2\|\hat{B}_k\|}\,. \tag{3.31}$$

If $\tau^* = \Delta_k$, then φ is a monotone decreasing function over $[0, \Delta_k]$. This implies that

$\mu_k \Delta_k \leq \|\hat{g}_k\|$, and we have that

$$\varphi(\tau^*) = \varphi(\Delta_k) \leq -\frac{1}{2}\Delta_k \|\hat{g}_k\| \,. \tag{3.32}$$

Since the sufficient decrease condition (3.19) implies that $\psi_k(s_k) \leq \beta_1 \varphi(\tau_k^*)$, the two bounds we just derived imply the lemma. □

We can derive a powerful result from this lemma, which gives a lower bound on the function value of the next successful TR iterate. Recall the definition of ρ_k:

$$\rho_k = \frac{f(x_k + s_x) - f(x_k)}{\psi(s_k)} \,, \tag{3.33}$$

which we can rewrite as

$$-\psi(s_k) = \frac{-f(x_k + s_x) + f(x_k)}{\rho_k} \,, \tag{3.34}$$

Plugging the above equation into Lemma 1, using the definition of successful iteration, and using assumption (3.21) yields

$$f(x_k) - f(x_{k+1}) \geq \frac{1}{2}\beta_1 \mu \|\hat{g}_k\| \min\left\{\Delta_k, \frac{\|\hat{g}_k\|}{\sigma_1}\right\} \,. \tag{3.35}$$

We will use the inequality (3.35) in proving Theorem 1.

With the assumption on the gradient behavior (3.16), algorithm behavior (3.19) - (3.21) and Lemma 1, we can now prove the following theorem (adapted from [Moré, 1982]):

Theorem 1. *If $f : \mathbb{R}^n \to \mathbb{R}$ is continuously differentiable and bounded below on $\mathbb{R}^n$, then*

$$\liminf_{k\to\infty} \|g_k\|_{(D_k^T D_k)^{-1}} = 0\,.$$

Proof. Suppose, on the contrary, that there exists $\epsilon > 0$ such that $\|g_k\| \geq \epsilon$ for sufficiently large k, where $\|\cdot\|$ is the ℓ_2 norm. Since we will eventually establish bounds that involve the trust region radius Δ_k, as a first step let us show that

$$\sum_{k=1}^{\infty} \Delta_k < \infty\,, \tag{3.36}$$

which implies that in its limit, the trust region radius approaches zero. There are two cases to consider: the case where there are a finite number of successful iterations, and the case where there are an infinite number of successful iterations. In the first case, we take a finite number of successful iterations to imply that we have an infinite number of unsuccessful iterations; hence, by the TR radius update we have that $\Delta_{k+1} \leq \gamma_1 \Delta_k$ for k sufficiently large, implying (3.36). In the second case, we consider an infinite sequence $\{k_i\}$ of successful iterations. Since f was assumed to be bounded below, and a successful step satisfies the sufficient decrease condition, it follows that

$$\{f(x_{k_i}) - f(x_{k_i+1})\}_i \to 0 \tag{3.37}$$

Due to (3.35), (3.37), and the initial assumption that there exists some $\epsilon > 0$ such that $\|g_k\| \geq \epsilon$, we see that $\{\Delta_{k_i}\} \to 0$. In turn, this implies

$$\sum_{i=1}^{\infty} \Delta_{k_i} < \infty \,. \tag{3.38}$$

The sum of the TR radii can then be decomposed into the sum of the successful iterates and the sum of the unsuccessful iterates. Using the TR radius update rule, the sum of all the unsuccessful iterates can be expressed as:

$$\sum_{i=1}^{\infty} \sum_{j=k_i+1}^{k_{i+1}-1} \Delta_j \leq \sum_{i=1}^{\infty} \gamma_2 \Delta_{k_i} \sum_{j=k_i+1}^{k_{i+1}-1} \gamma_1^{j-k_i} \leq \frac{\gamma_2}{1-\gamma_1} \sum_{i=1}^{\infty} \Delta_{k_i} \tag{3.39}$$

Combining the bounds for the sum of the successful and unsuccessful iterates yields

$$\sum_{k=1}^{\infty} \Delta_k \;<\; \left(1 + \frac{\gamma_2}{1-\gamma_1}\right) \sum_{i=1}^{\infty} \Delta_{k_i} \,, \tag{3.40}$$

in turn implying (3.36). As a next step, we show that (3.36) implies that $\{\rho_k - 1\}$ converges to zero. We begin this step by writing down the definition of ρ_k:

$$\rho_k - 1 = \frac{f(x_k + s_k) - f(x_k)}{\psi_k(s_k)} - 1 = \frac{f(x_k + s_k) - f(x_k) - \psi_k(s_k)}{\psi_k(s_k)} \,. \tag{3.41}$$

We then examine bounds for the numerator and denominator. We then perform the

Taylor expansion of $f(x_k + s_k)$:

$$|f(x_k + s_k) - f(x_k) - \psi_k(s_k)| = |f(x_k) + \nabla f(x_k)^T s_k + O(\|s_k\|^2) - f(x_k) - \psi_k(s_k)| \tag{3.42}$$

$$\leq |\nabla f(x_k)^T s_k - \psi_k(s_k)| + O(\|s_k\|^2) \tag{3.43}$$

Taking the scaling matrix $D_k = I$ for simplicity, we then bound the first term above in the following manner:

$$\begin{aligned}
|\psi_k(s_k) - \nabla f(x_k)^T s_k| &= |g_k^T s_k - \frac{1}{2} s_k^T B_k s_k - \nabla f(x_k)^T s_k| && (3.44)\\
&\leq |g_k^T s_k - \nabla f(x_k)^T s_k| + \frac{1}{2}|s_k^T B_k s_k| && (3.45)\\
&\leq \|g_k - \nabla f(x_k)\| \|s_k\| + \frac{1}{2}\|B_k\| \|s_k\|^2 && (3.46)\\
&\leq \|g_k - \nabla f(x_k)\| \|s_k\| + \frac{1}{2}\sigma_1 \|s_k\|^2 . && (3.47)
\end{aligned}$$

We then use Heinkenschloss et al.'s error bound on the inexact gradient:

$$\begin{aligned}
|\psi_k(s_k) - \nabla f(x_k)^T s_k| &\leq \|g_k - \nabla f(x_k)\| \|s_k\| + \frac{1}{2}\sigma_1 \|s_k\|^2 && (3.48)\\
&\leq K \min\{\|g_k\|, \Delta_k\} \beta_2 \Delta_k + \frac{1}{2}\sigma_1 \beta_2^2 \Delta_k^2 && (3.49)\\
&\leq \left(K\beta_2 + \frac{1}{2}\sigma_1 \beta_2^2\right) \Delta_k^2 . && (3.50)
\end{aligned}$$

We now bound the denominator using Lemma 1 and the original assumption that there exists $\epsilon > 0$ such that $\|g_k\| \geq \epsilon$ for sufficiently large k.

$$\begin{aligned} -\psi_k(s_k) &\geq \frac{1}{2}\beta_1 \|g_k\| \min\left\{\Delta_k, \frac{\|g_k\|}{\|B_k\|}\right\} && (3.51) \\ &\geq \frac{1}{2}\beta_1 \epsilon \min\left\{\Delta_k, \frac{\epsilon}{\sigma_1}\right\} && (3.52) \\ &\geq \frac{1}{2}\beta_1 \epsilon \Delta_k \, . && (3.53) \end{aligned}$$

Since we established that $\Delta_k \to 0$, (3.43), (3.50) and (3.53) imply that $\{\rho_k - 1\} \to 0$. However, the trust region radius update rules show that Δ_k is not decreased if $\rho_k \geq \eta$. This implies that Δ_k cannot converge to zero, which contradicts our original assumption, hence proving the theorem. □

We have just shown the convergence properties of the inexact TR algorithm, using Heinkenschloss et al.'s gradient error bound. In the next section, we proceed to examine the differential equation constraint hiding in the function $C(y(t, u), u)$. I will also discuss the various evolutions that is of interest in this proposal: the reference, linearized and adjoint evolutions.

3.3 The Adjoint State Method and Adaptive Time Stepping

Recall that the constraint (state) equation represents a differential equation:

$$C(y(t),u) = y(t) - \int_0^t H(y(s,u),u)ds = 0\,, \tag{3.54}$$

from which we recover the differential equation constraint (3.2), by using the fundamental theorem of Calculus:

$$\begin{aligned} \frac{dy(t,u)}{dt} &= H(y(t,u),u)\,, \quad t \in [0,T] && (3.55)\\ y(0,\cdot) &= 0\,. && (3.56) \end{aligned}$$

Together, (3.55) - (3.56) are referred to as the "reference" or "forward" equations. We assume that the solution to the forward problem $y \in C^1([0,T],Y)$, where Y is some Hilbert space.

Naturally, if we wish to numerically solve this system of differential equations, we can use One-step or Multi-step Methods such as Forward Euler, Runge-Kutta, or Leap-Frog. In its general form, a multi-step method can be written as

$$\sum_{j=0}^{N} \alpha_j \bar{y}_{n+j} = h_n^{(f)} \sum_{j=0}^{N} \beta_j \Phi^j(\bar{y}_{n+j},c)\,, \tag{3.57}$$

where $\{\hat{y}_n\}$ are the solution approximations, $\{h_n^{(f)}\}$ are the time-steps (the superscript (f) denotes the forward grid), $\{\alpha_i\}$ and $\{\beta_j\}$ are scaling parameters, and $\{\Phi^j\}$ is a family of functions (which are typically sums of the function f evaluated at different points). To simplify notation in (3.57), we introduce the discrete dynamic operator H and note that multi-step methods can be written as a one-step method [Kirchgraber, 1985]. Then (3.57) can be written in the form

$$y_{n+1} = y_n + h_n^{(f)} \bar{H}(y_n, c_n, \Delta t_n^{(f)}), \qquad n = 0, 1 \ldots, (N_{(f)} - 1)\,, \tag{3.58}$$

$$y_0 = 0\,. \tag{3.59}$$

Note that the transformed states here are denoted y, whereas in (3.57) they were denoted $\hat{y}$. Also note that in (3.58), $\bar{H}$ is the discrete analogue of the dynamic operator H and $h_n^{(f)} = \sum_{j=1}^{n} h_j^{(f)}$. If the constraint equation (which is also the reference evolution) is stiff in some 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:

$$0 = \frac{d\delta y}{dt} - D_y H(y(t,u),u)\delta y - D_u H(y(t,u),u)\delta u \tag{3.60}$$

$$\delta u(0) \equiv 0 \text{ for } t < 0\,. \tag{3.61}$$

In (3.60), $\delta y(t)$ refers to the state perturbation and δu refers to the control perturbation. The solution to the sensitivity equation δy may be needed for the adjoint evolution. Also, another approach of obtaining the gradient of the objective function (3.1) with respect to the control parameter is the "sensitivity method approach" [Li and Petzold, 2004]. The sensitivity equations can then be solved discretely by performing the following update:

$$\begin{aligned}
\delta y_{n+1} &= \delta y_n + h_n^{(d)}[D_u\bar{H}(y_n, c_n, h_{(d)}^n)\delta y_n + D_c\bar{H}(y_n, c_n, h_n^{(d)})\delta c_n]\,, && (3.62)\\
&\quad n = 0, \ldots, (N_{(d)} - 1) \quad , && (3.63)\\
\delta y_0 &\equiv 0. && (3.64)
\end{aligned}$$

The corresponding time levels, $\{h_n^{(d)}\}$ is referred to as the *derivative grid*.

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. Given the adjoint state field $\lambda \in C^1([0,T], Y)$, the "backward in time" adjoint evolution can be written as

$$\begin{aligned}
0 &= \frac{d\lambda}{dt} + (D_u H(y(t,u), u))^* \lambda && (3.65)\\
\lambda &\equiv 0\,, \quad t > T\,, && (3.66)
\end{aligned}$$

where the operator * denotes the adjoint. The corresponding discrete adjoint evolution can then be written as:

$$w_n = w_{n+1} + h_{(a)}^{n+1} (D_u \bar{H}[u_n, c_n, h_n^{(a)}])^* w_{n+1} \,, \tag{3.67}$$

$$n = (N-1), \ldots, 0 \,, \tag{3.68}$$

$$w_n = 0 \text{ for } n > (N_{(a)} - 1) \,. \tag{3.69}$$

Recall that we are using adaptive time stepping schemes to solve the discretized forward, derivative and adjoint evolution problems, (3.58), (3.62) and (3.67). Using adaptive time stepping, however, presents a dilemma: the adjoint evolution requires access to the forward states, implying that the forward and adjoint time grids must match. If we use adaptive time stepping schemes, however, we are no longer guaranteed that the forward and adjoint grids will align. We must therefore interpolate the forward states to provide an approximation that aligns with the adjoint grid. Doing so, however, will introduce an interpolation error. I will discuss this interpolation error in more detail in the "Future Work" chapter, as well as discuss my plans to tie the interpolation error, truncation error into Heinkenschloss and Vicente's TR algorithm.

Chapter 4

TSOpt

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 an "interface for time-stepping simulation" written in C++ [Symes, 2006]; it encapsulates reference, linearized and adjoint simulators in a single object.

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) and `LocalDataContainers`

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".

RVL data management classes to provide workspace for its simulations. An example of a data management class is the concrete data storage class called `LocalDataContainer`. In an algorithmic context, it is useful to think of a `LocalDataContainer` object as a specialized array that has functions to relay information about itself, such as size and data. LocalDataContainer objects also have the ability to manipulate the data they store through FunctionObjects, which are interfaces "behind which to hide data manipulations of all sorts" [Padula et al., 2009]. The code below shows all essential functions in the LocalDataContainer class. The comments above each function declaration explain what each function does. Note the template argument at the top of the class, which dictates the type of data the LocalDataContainer holds.

```
template<class DataType>
class LocalDataContainer: public DataContainer {

  public:

    /** return size of local data container */
    virtual int getSize() const = 0;

    /** return address of writable data array */
    virtual DataType * getData() = 0;

    /** return address of read-only data array */
    virtual DataType const * getData() const = 0;

    /** local evaluation: defined at this level so that subtypes do not
    need to re-implement.
    */
    void eval(FunctionObject & f,
          vector<DataContainer const *> & x) { ... }

    /** Similar evaluation method for FORs. */
    void eval(FunctionObjectRedn & f,
          vector<DataContainer const *> & x) const { ... }
};
```

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 as `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. Since it is crucial that we understand their functionality, they are discussed in detail in the following subsections.

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. The following implements the `LoopAlg` class.

```
class LoopAlg: public Algorithm {
public:
  LoopAlg(Algorithm & alg, Terminator & stop) : inside(alg), term(stop) {}

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

    return t1;
  }

protected:
  Algorithm & inside;
  Terminator & term;
};
```

Note that the `LoopAlg` also needs to ensure that its `Algorithm` object completed it's 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. The following is the implementation of the `ListAlg` class:

```
class ListAlg: public Algorithm {
public:
  ListAlg(Algorithm & first): one(first), islist(false), two(*this) {}
  ListAlg(Algorithm & first, Algorithm & next)
    : one(first), islist(true), two(next) {}

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

    return (t1 && t2);
  }


protected:
  bool islist;
  Algorithm & one;
  Algorithm & two;

};
```

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 (4.1). All `State` classes must implement methods to get and set the `time` object, and methods to access and alter its internal data structure.

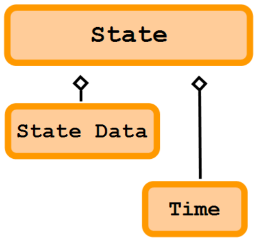

Figure 4.1: The State class and its components

There are two example of `State` subclasses that have been implemented in `TSOpt`, 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 is actually a wrapper class for the `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 `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` setruct, defined as:

```
  typedef struct {
    /** current time */
    double time;
    /** state dim */
    int nu;
    /** control dim */
    int nc;
    /** state samples */
    double * u;
    /** control samples */
    double * c;

  } realrn;
```

There are two differences worth noting between the RnState and RealRnState classes. First, note that RealRnState's internal data type 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.

For example, 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 in the adjoint computation). This `Sim` subclass does not require the storage of the simulation state history. The `StdSim` class is implemented in the following manner:

```
template<typename State>
class StdSim: public Sim<State> {

private:
  StdSim();

public:
  /** main constructor */
  StdSim(TimeStep<State> & step,
         TimeTerm & term,
         Algorithm &initstep)
    : Sim<State>(step, term, initstep) {}
  /** use arg struct - for generic policy */
  StdSim(StdSimData<State> const & d)
    : Sim<State>(d.getStep(),d.getTerm(),d.getInit()) {}

  StdSim(StdSim<State> const & s)
    : Sim<State>(s) {}

  virtual ~StdSim() {}

  bool run() {
    try {
      LoopAlg a(this->step, this->term);
      ListAlg aa(this->initstep, a);
      aa.run();
    }
    catch (RVLException & e) {
      e<<"\ncalled from StdSim::run\n";
      throw e;
    }
    return true;
  }

};
```

Note that there is 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.

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. The user-defined stack is a template argument to `RASim`, as well as other `Sim` subclasses that need to store parts of the simulation state history. The concrete stack `stdVector` highlights the necessary interfaces the stack classes require in order to operate with TSOpt:

```
  /** StdVector -- a wrapper class around the STD vector class */
  template<typename State, typename Alloc>
  class StdVector {

  private:
    std::vector<State*> _ldclist;

  public:
    StdVector() : _ldclist() {}

    StdVector(StdVector<State, Alloc> const & s): _ldclist(s._ldclist)
    {}

    ~StdVector() {
     for (int i=0; i<_ldclist.size(); ++i) {
delete _ldclist.at(i);
      }

     _ldclist.clear();
    }


    /** Place state element at the back of the stack */
    void push_back(State const & t ) {
      State * tmp;
      tmp = new State(t);

      _ldclist.push_back(tmp);
    }

    /** Pop the state element at the top of the stack */
    void pop_back() {
      delete _ldclist.back();
      _ldclist.pop_back();
    }

    /** Returns stack size */
    int size() { return _ldclist.size(); }


    /** Allows access to a specific state at index idx */
    State & at(int idx) { return *(_ldclist.at(idx));  }
    State const & at(int idx) const { return *(_ldclist.at(idx)); }

    /** Returns reference at the head of the stack */
    State & front() { return *(_ldclist.front()); }
    State const & front() const { return *(_ldclist.front()); }

    /** Returns reference at the tail of the stack */
    State & back() { return *(_ldclist.back()); }
    State const & back() const { return *(_ldclist.back()); }

  };
```

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 "forgetful" simulator and a "remember-all' simulator; it allows access to the forward simulation state history for a logarithmic forward recomputation and storage cost (with respect to the total number of time steps taken). Two types of checkpointing exist in TSOpt: offline mode for fixed time step simulations, and online mode for adaptive simulations. A more thorough discussion of optimal checkpointing (both online and offline modes) can be found in [Griewank and Walther, 2000] and [Hinze and Sternberg, 2005].

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 reference, 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.

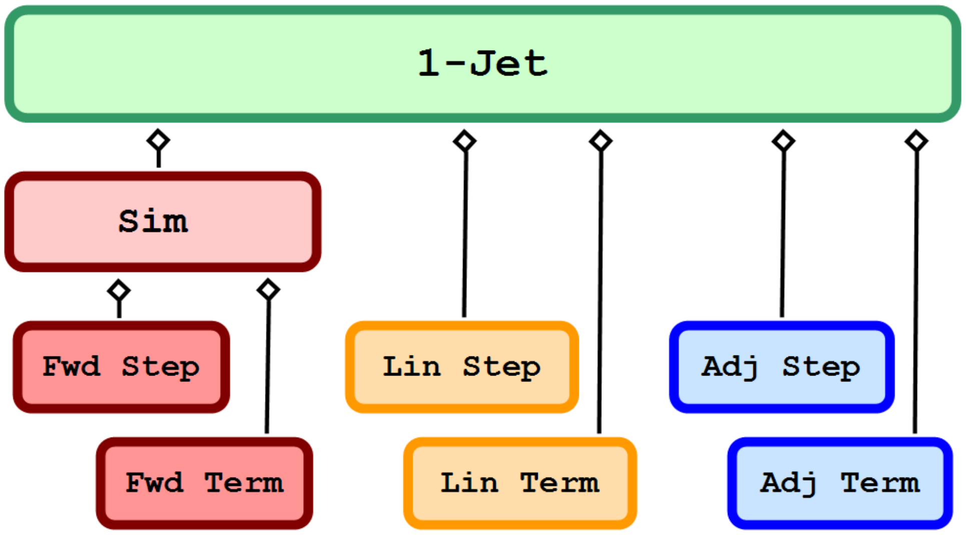

Figure 4.2: The jet class and its components.

The jet objects provide three very important functions that return the forward evolution Sim object or create a linearized and adjoint evolutionSim 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

// Create jet object
jet j(...);

// Run forward sim
j.getFwd().run();

// Run lin. sim
j.getLin().run();

// Run adj. sim
j.getAdj().run();
```

4.4 TSOpt and Optimization

Recall that TSOpt provides various simulation operators whose output can be used in conjunction with optimization algorithms. Since TSOpt was created from subclassed Alg components, it is natural that we use an optimization package also created from the Alg framework. Such an optimization package exists for unconstrained optimization, called the umin package [Padula et al., 2009]. We choose the unconstrained optimization package umin since we are considering the "black-box" approach to solving the simulation driven optimization problem. Currently, only the LBFGS and Newton optimization algorithms are available in `umin`.

Chapter 5

The Black Oil Equations

In this chapter, I discuss 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 proposal, I implement a Black-Oil reservoir simulator in the TSOpt framework. This establishes the necessary code for my dissertation, in which I will use adaptive time-stepping algorithms to solve these partial differential equations in time.

I begin by introducing the mathematical equation, 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:

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

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

where the subscripts a and l respectively refer to the aqueous and liquid phase, ρ is the fluid density, K is the absolute permeability of the medium, k_r is the relative permeability, μ 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, ϕ 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:

$$S_l + S_a = 1\,. \tag{5.3}$$

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:

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

where θ denotes a fluid phase and λ denotes the phase mobility. Substituting (5.4) into the phase continuity equations (5.1) yields:

$$\begin{aligned} \nabla \cdot v_l(t,x) \quad & -q_l(t,x) - \tfrac{\partial(\phi(t,x)\rho_l(t,x)S_l(t,x))}{\partial t} & = 0 \qquad (5.5) \\ \nabla \cdot v_a(t,x) \quad & -q_a(t,x) - \tfrac{\partial(\phi(t,x)\rho_a(t,x)S_a(t,x))}{\partial t} & = 0\,. \qquad (5.6) \end{aligned}$$

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

$$\begin{aligned} \nabla \cdot v_l(t,x) \quad & -q_l(t,x) - \phi\tfrac{\partial S_l(t,x)}{\partial t} & = 0 \qquad (5.7) \\ \nabla \cdot v_a(t,x) \quad & -q_a(t,x) - \phi\tfrac{\partial S_a(t,x)}{\partial t} & = 0\,, \qquad (5.8) \end{aligned}$$

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

5.1 Solving the Black Oil Equations

Wiegand et al. [2008] 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 Ω as Ω_i, each with its own boundary $\partial\Omega_i$, we can express the pressure equation as:

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

The saturation equation can be written as:

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

The next two sections are reveal the discretization of the pressure and saturation equations in space, and in time. Wiegand et al. [2008] give a thorough treatment of the derivation, 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.

5.1.1 Discretizing the Pressure Equation

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

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

where j being a neighbor of i implies that the volumes Ω_j are adjacent to the volume Ω_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

$$T_{i,j} = \frac{K_{i,j}A_{i,j}}{l_{i,j}} \, , \tag{5.12}$$

we may simplify the discretized pressure residual equation as

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

which we put into matrix form as

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

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

$$A_{i,j} = -T_{i,j} \lambda_{t_{i,j}} \qquad A_{i,i} = \sum_j T_{i,j} \lambda_{t_{i,j}} \,. \tag{5.15}$$

5.1.2 Discretizing the Saturation Equation

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

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

We can express the equation above as:

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

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

$$
\begin{aligned}
D_{i,i} &= \phi_i \cdot |\Omega_i| && (5.18)\\
\tilde{A}_{i,j} &= -T_{i,j}\lambda_{a_{i,j}} \qquad \tilde{A}_{i,i} = \sum_j T_{i,j}\lambda_{a_{i,j}}\,. && (5.19)
\end{aligned}
$$

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.

5.2 Solving the Discretized Pressure and Saturation Equations

There are also many possible approaches to solving the discretized pressure and saturation equations. Peaceman in [Peaceman, 1977] offers a more detailed survey of solution strategies for the saturation and pressure equations. In this proposal, we 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 equations, with primary variables as the pressure p^{k+1} and the aqueous saturation s_a^{k+1}:

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

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

5.3 Implementation in TSOpt

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

- a state type that is capable of holding the primary variables (aqueous pressure and saturation)
- a "stack" class that handles storage of the state history, if we choose to use a checkpointing scheme for the adjoint computation
- `Step` classes that define *one step* of the forward and the adjoint evolution .

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

There is a `Fwd_BO_Dyn` class which encapsulates the forward evolution. Since we consider the coupled implicit formulation only, calling the `run()` method of a

`Fwd_BO_Dyn` objects solves the discretized pressure residual equation and the saturation equation using the backward Euler time-stepping scheme simultaneously. Further, since backward Euler is an implicit scheme, we use the Newton algorithm to solve the nonlinear equations. We must, hence, solve multiple linear systems (until convergence) at every time step in the forward evolution. Similarly, there is a `Adj_BO_Dyn` class whose `run()` function runs one step of the adjoint evolution.

I then created a `Sim` object, which is composed of an appropriate `Terminator` object and an `Fwd_BO_Dyn` object. In turn, this `Sim` object, along with an `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 Black-Oil simulator.

5.4 Numerical Results

5.4.1 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.

The first set up results is for a 100-day simulation, with 25-day time steps. I use the porosity and permeability data from the top layer of the SPE10 model in the simulation.

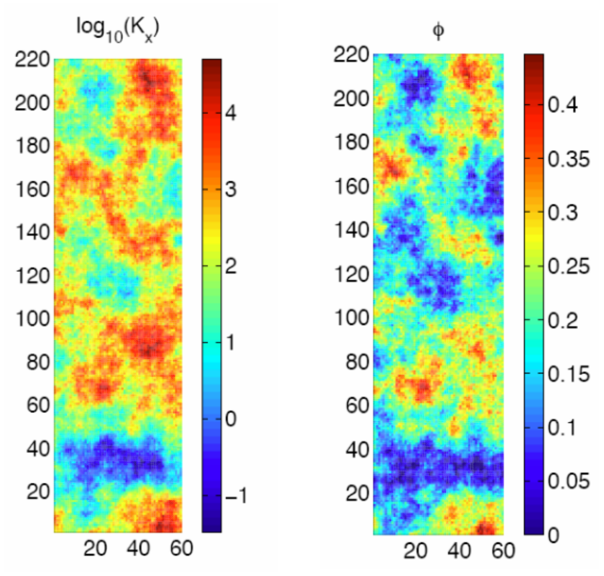

Figure 5.1: Porosity and permeability plot of the SPE10 model, top layer.

The source and sink terms (which correspond to injecting and producing wells in this example) are configured in the following manner in the domain:

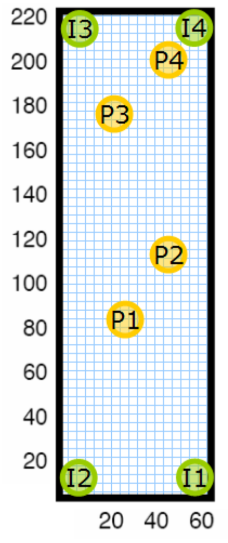

Figure 5.2: Placement of injector (I) and producer (P) wells in the domain.

Given the porosity, permeability, and source/sink data, the results of the 100-day simulation can be seen in figure (5.3). Note how the water saturation is high where

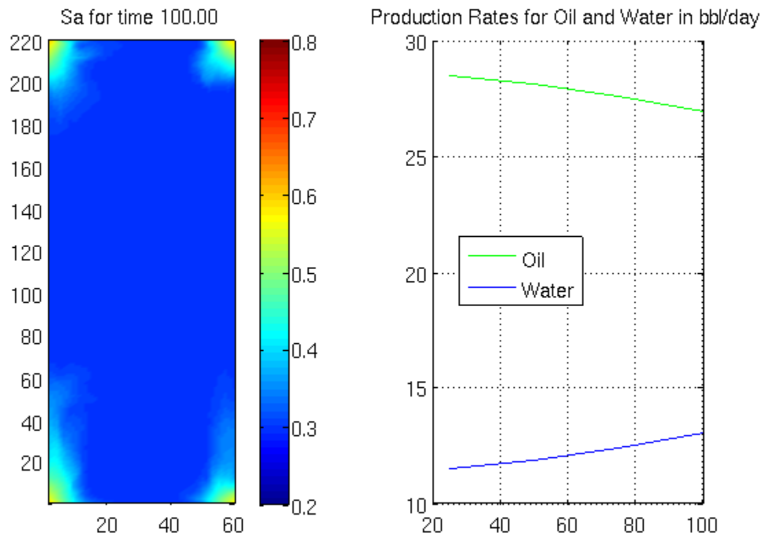

Figure 5.3: Plot of Aqueous Saturation at t = 100 days, with dt = 25.

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.

5.4.2 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 first define an optimization problem with the reservoir simulation constraints. Consider the following optimization problem, posed by Wiegand et al. [2008], that finds the optimal well rate that will maximize revenue from oil production, while penalizing water injection and production:

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

where α, β and γ are scalar variables and the aqueous pressure p and aqueous saturation s_a solve:

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

$$+ \sum_{i\in P\cup I} s_a q_i(t)\delta(x-x_i) \tag{5.23}$$

$$\phi(x)\frac{\partial}{\partial t}s_a(x,t) - \nabla\cdot(K(x)\lambda_a(s_a(x,t))\nabla p(x,t)) = \sum_{i\in P\cup I} s_a q_i(t)\delta(x-x_i)\,. \tag{5.24}$$

We use the "discretize-then-optimize" approach to obtaining the derivative of the optimization problem (5.21). Further, we incorporate explicit equality and inequality constraints on the well rates to model the physical limitation of the wells. The fully discretized optimal control problem then takes the form of:

$$\min \qquad J_{\Delta t}(q) = \Delta t \sum_{k=1}^{N} l(t^k, s^k, q^k) \tag{5.25}$$

$$s.t. \qquad e^T q^k = 0 \tag{5.26}$$

$$q_{min} \le q^k \le q_{max}\,, \tag{5.27}$$

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

$$\begin{bmatrix} f(t^{k+1}, s_a^{k+1}, p^{k+1}, q^{k+1}) \\ \\ g(t^{k+1}, s_a^{k+1}, p^{k+1}, q^{k+1}) \end{bmatrix} = \begin{bmatrix} q - Ap^{k+1} \\ \\ D^{-1}(q_a - \tilde{A}p^{k+1}) \end{bmatrix} = \begin{bmatrix} 0 \\ \\ \frac{s_a^{k+1} - s_a^k}{\Delta t} \end{bmatrix} . \tag{5.28}$$

Wiegand et al. [2008] derive the adjoint equations from the optimality conditions, and arrive at the following adjoint evolution scheme. For $k = N-1, \ldots, 1$, simultaneously solve for the adjoint variables λ_s^k and λ_p^k in the following equation:

$$-\frac{\lambda_s^{k+1} - \lambda_s^k}{\Delta t} = D_s f(\ldots^k)^T \lambda_s^k - D_s g(\ldots^k)^T \lambda_p^k - \nabla_s l(\ldots^k) \tag{5.29}$$

$$0 = -D_p f(\ldots^k)^T \lambda_s^k + D_p g(\ldots^k)^T \lambda_p^k . \tag{5.30}$$

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

$$\nabla J(q)\delta q = \sum_{k=1}^{N} \Delta t[\nabla_q l(\cdot^k) - D_{q^k} f(\ldots^k)^T \lambda_s^k + D_{q^k} g(\ldots^k)^T \lambda_p^k]^T \delta q^k) . \tag{5.31}$$

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

$$\lim_{h \to 0} \frac{f(x + h\delta x) - f(x) - hf'(x)[\delta x]}{h} = 0 . \tag{5.32}$$

Suppose rewrite (5.32) as:

$$\lim_{h \to 0} \frac{f(x + h\delta x) - f(x)}{h} - f'(x)[\delta x] = 0\,. \tag{5.33}$$

We note that the first component resembles 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 different 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:

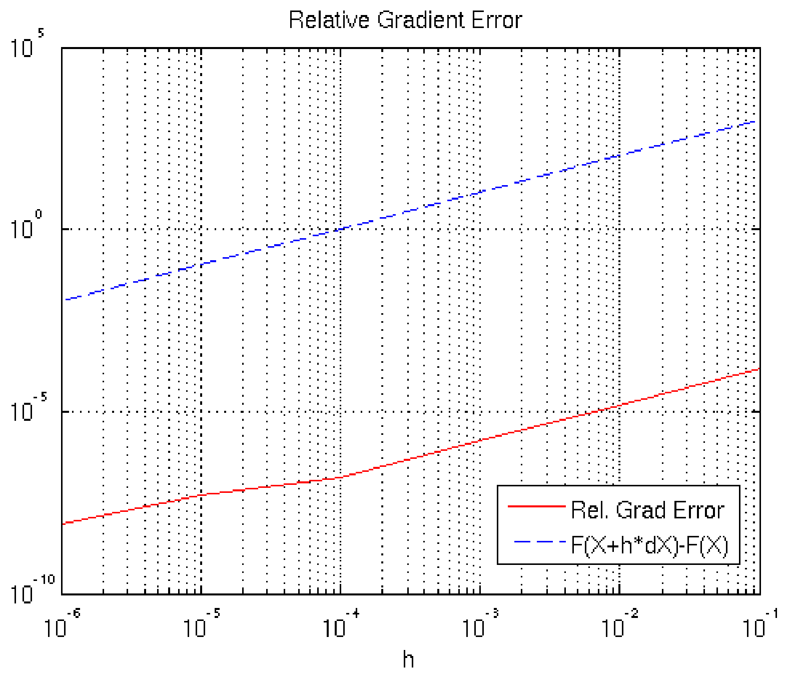

Figure 5.4: Plot of the difference between the computed gradient via the adjoint-state method, and the finite difference approximation.

We see that the difference between the computed gradient and the finite difference 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.

This was actually the topic of one of my internships at ExxonMobil, for which I got great results. Though the specific data associated with these results are proprietary, there was a very intuitive logic behind the results. If one looks at the plot of producers and injectors, figure (5.2), 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. Since we still need injector 4 to push the oil out of the reservoir, however, we expect to see the optimizer close producing well 4. My initial goal would then to match this result using Moré’s unconstrained TR algorithm, enforcing the explicit constraints with a barrier method. This will be discussed in more detail in next chapter, “Future Work”.

Chapter 6

Proposed Future Work

This chapter discusses the work I have, and plan to, complete for my dissertation. The first section discusses what I need to do in order to establish a convergence proof for an optimal control problem whose simulation constraint is solved by an adaptive time-stepping algorithm. The second section highlights my plans to add adaptive time-stepping logic to the fixed-step Black-Oil simulator I presented in the previous chapter. This second section also discuses the research and implementation work I have completed towards an adaptive Black-Oil simulator. In the third and sections, I talk about software I intend to implement for my dissertation; in the third section, I list algorithms I will add to the optimization framework in `RVL`, while in the fourth section, I discuss my plans to incorporate a testing framework within TSOpt. The final section of this chapter then explains my plans for the adaptive Black-Oil simulator, once it is completed: namely, to study the quality of gradients generated via the adjoint-state method and adaptive time-stepping, and its effects on the convergence of optimization algorithms.

6.1 Convergence Theory

First and foremost, I wish to establish a convergence theorem for the optimal control problem:

$$\begin{aligned} \min \quad & f(y(t),u) = \int_0^T J(y(t),u)dt && (6.1)\\ \text{s.t.} \quad & \frac{d}{dt}y(t) - H(y(t),u) = 0\,, \qquad t \in [0,T] && (6.2)\\ & H, y \equiv 0 \text{ for } t < 0\,, && (6.3) \end{aligned}$$

when the differential equation constraint is solved via adaptive time-stepping methods, and the gradient of the objective function with respect to the controls is obtained via the adjoint state method. Since I intend to use the TR optimization algorithm with inexact gradients, I must figure out how to compute (and enforce) Heinkenschloss and Vicente's bound on the approximated gradient error

$$\|g_k - \nabla f(x_k)\| \leq K \min\{\|g_k\|, \Delta_k\}\,. \tag{6.4}$$

The hardest quantity to compute in the bound above is the gradient error $\|g_k - \nabla f(x_k)\|$, since in general, we do not have access to the true gradient ∇f. We can, however, use an approximation of the gradient error, which can be obtained through an *a-posteriori* error estimate. There are numerous ways to estimate the global error *a-posteriori*, as noted by Skeel [1986]. In his survey paper, Skeel had mentioned a simple, yet popular, method of error estimation that involves computing solutions at different tolerances – something that is directly applicable to the adaptive time-stepping schemes I consider for this proposal. Let g^{τ} be an approximate gradient computed with tolerance τ, and let $g^{R\tau}$ be the approximate gradient computed at a cruder tolerance $R\tau$. We can then extrapolate and use the following as an error estimate for y:

$$\frac{g^{\tau} - g^{R\tau}}{R-1}\,. \tag{6.5}$$

Skeel, however, noted that this error estimate is not recommended. Skeel [1986] suggested that the error estimate $|g^{\tau} - g^{R\tau}|$ is safer to use.

Further, we must ensure that computing the approximate gradient with a lower tolerance results in a "better" approximate gradient. Since the key components of gradient construction via the adjoint state method are the adjoint states, I must ensure that lowering algorithmic tolerances imply a decrease in the global error in the adjoint evolution. My initial analysis highly suggests that the global error of the adjoint-state method (when using adaptive time stepping, coupled with interpolation) is not only bounded, but also controllable through algorithmic parameters. The main ideas that justify this claim are the following: first, by definition, adaptive time stepping algorithm's tolerance is the maximum allowable truncation error per time step. It follows that the truncation error can be lowered by adjusting the adaptive time stepper's tolerance. Second, we know that interpolation error is bounded (for both polynomials and splines), given that the function whose points we sample has $(n+1)$ bounded derivatives, where n is the order of interpolation. The interpolation error can be lowered in two ways. First, we indirectly lower the interpolation error by lowering the time-stepping tolerance, since this means we have more nodes to use as interpolation data points. Second, we can directly lower the interpolation error by adjusting the maximum allowable time-step of the adaptive time-stepper. Finally, I would like to note that careful observation of these errors should dictate how one should choose the constant K in Heinkenschloss and Vicente's error bound.

As a next step, I plan to extend Heinkenschloss and Vicente's theoretical framework, so that it may accommodate solving the state equations with adaptive time stepping. What mathematical space should the state y belong to if adaptivity is allowed? How does this affect what kind of functions space the dynamic operator H belongs to? These are some of the questions I need to research and answer, in order to come up with a rigorous convergence theory for this problem.

With these components in place, we can enforce Heinkenschloss and Vicente's bound on the approximated gradient error

$$\|g_k - \nabla f(x_k)\| \leq K \min\{\|g_k\|, \Delta_k\}\,, \tag{6.6}$$

by modifying the TR algorithm in the following manner:

Algorithm 2: Modified TR, With Added Control Logic

```
A. Define initial tolerances for adaptive time stepper. Also assign initial
algorithmic parameters to TR algorithm, including $\Delta_0$.
B. Compute approximate gradient given initial tolerances
C. Set $k = 0$
while TR != CONVERGED do
    1. Set $flag = false$
    while flag == false do
        2. Calculate a-posteriori error estimate
        if error estimate $\leq \min\{\|g_k\|, \Delta_k\}$ then
            3a. Set $flag = true$;
        end
        else
            3b. Halve the tolerances
            3c. Compute approximate gradient with new tolerances
        end
    end
    4. Carry on with TR step
    5. Equate the global error bound (which is a function of the algorithmic
    parameters) to $\min\{\|g_k\|, \Delta_k\}$, and find tolerances that satisfy such an
    equation
    6. Calculate the approximate gradient with the tolerances obtained in (5)
    7. Set $k = k + 1$
end
```

This algorithm ensures that Heinkenschloss and Vicente's error bound is always satisfied, hence guaranteeing global lim inf convergence per Moré's theorem on the convergence of the TR algorithm with inexact gradient information.

6.2 Adding Adaptive Logic to the Black-Oil Simulator

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 are based on embedded Runge-Kutta schemes.

Reservoir engineers, however, have adopted a different way 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:

$$\Delta t_p = \Delta t^n \frac{p_{lim}}{p_{max}} \tag{6.7}$$

$$\Delta t_s = \Delta t^n \frac{s_{lim}}{s_{max}} \tag{6.8}$$

$$\Delta t^{n+1} = \min(\Delta t_p, \Delta t_s)\,. \tag{6.9}$$

It is clear that controlling p_{lim} and s_{lim} affects the truncation error of the time stepping scheme, since $p_{lim} \to 0$ and/or $s_{lim} \to 0$ implies $\Delta t \to 0$. Given this plan for adaptive time-stepping for the Black-Oil equations, I now segue to its implementation in TSOpt.

6.2.1 Implementation of an Adaptive Black-Oil Simulator

As a preliminary step to adding adaptive time-stepping logic to TSOpt's Black-Oil Simulator, let us examine and analyze the algorithm for the fixed-step simulation. Recall that we solve the discretized Black-Oil equations using the IMPSAT approach: for $k = 0, 1, \ldots N-1$ we solve

$$\begin{bmatrix} q^{k+1} - Ap^{k+1} \\ \Delta t D^{-1}(q^{k+1} - \tilde{A}p^{k+1}) - (s_a^{k+1} - s_a^k) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{6.10}$$

using a Newton scheme.

If we consider performing adaptive time-steps for the scheme (6.11), we need not change much. In the problem formulation I consider, the control parameter is time-independent. 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. The algorithm for the reference simulation is presented below:

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

```
Let $\Delta t^0$, $p_{lim}$ and $s_{lim}$ be given.
Also, let the controls $q \in \mathbb{R}^n$ be defined
Set $k = 0, t = 0.0$.
while $t < T$ do
    a) Use function $\varphi(t, q)$, which takes a the control $q$ and a time $t$,
    to create a control $\tilde{q}$ that corresponds to time $t$
    b) Define $\tilde{q} = \varphi(t + \Delta t^k, q)$
    c) Obtain $p^{k+1}$ and $s^{k+1}$ by solving the following, using Newton's
    Algorithm:
        $$\begin{bmatrix} \tilde{q} - Ap^{k+1} \\ \Delta t^k D^{-1}(\tilde{q} - \tilde{A}p^{k+1}) - (s_a^{k+1} - s_a^k) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad (6.11)$$
    d) Set $t = t + \Delta t^k$
    e) Compute $p_{max} = \|p^{k+1} - p^k\|_\infty$ and $s_{max} = \|s^{k+1} - s^k\|_\infty$
    f) Calculate $\Delta t_p$ and $\Delta t_s$:
        $$\Delta t_p = \Delta t^n \frac{p_{lim}}{p_{max}} \qquad (6.12)$$
        $$\Delta t_s = \Delta t^n \frac{s_{lim}}{s_{max}} \qquad (6.13)$$
        $$\qquad (6.14)$$
    g) Set $\Delta t^{k+1} = \min(\Delta t_p, \Delta t_s)$
    h) Set $k = k + 1$
end
```

Next, I present the algorithm for the adaptive adjoint evolution for the Black-Oil equations. We now face a dilemma: 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. Recall there were three strategies for handling the reference states during the forward simulation.

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 checkpointing, which requires saving a subset of reference states. (A variant of checkpointing that handles adaptive simulations was already developed by Hinze et al. in [Hinze and Sternberg, 2005].) It may even be possible to combine the algorithm for adaptive checkpointing with the first strategy. We use the saved subset of the reference states as starting points for evolution, but we only evolve up to the time level needed by the adjoint evolution. This will have the benefit of balancing computational and storage cost, while not incurring interpolation error.

I present the adjoint Black-Oil algorithm below, which is compatible with the reference state storage strategies I discussed above:

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

```
Let $\Delta t^0$, $\lambda_{p_{lim}}$ and $\lambda_{s_{lim}}$ be given.
Also, let the controls $q \in \mathbb{R}^n$ be defined
Set $k = 0, t = T$.
while $t > 0$ do
    a) Use function $\varphi(t, q)$, which takes a the control $q$ and a time $t$,
    to create a control $\tilde{q}$ that corresponds to time $t$
    b) Define $\tilde{q} = \varphi(t - \Delta t^k, q)$
    c) Compute $p^*$ and $s^*$, which approximate the pressure and saturation at
    time $t - \Delta t^k$
    d) Obtain $\lambda_p^{k+1}$ and $\lambda_s^{k+1}$ by solving the following linear system:
    $$\begin{bmatrix} D_s f(\tilde{q}, p^*, s^*)^T & -D_s g(\tilde{q}, p^*, s^*)^T \\ -D_p f(\tilde{q}, p^*, s^*)^T & D_p g(\tilde{q}, p^*, s^*)^T \end{bmatrix} \begin{bmatrix} \lambda_s^{k+1} \\ \lambda_p^{k+1} \end{bmatrix} = \begin{bmatrix} \frac{\lambda_s^k - \lambda_s^{k+1}}{\Delta t} + \nabla_s l(\tilde{q}, p^*, s^*) \\ 0 \end{bmatrix}$$
    e) Set $t = t - \Delta t^k$
    f) Compute $\lambda_{p_{max}} = \|\lambda_p^{k+1} - \lambda_p^k\|_\infty$ and $\lambda_{s_{max}} = \|\lambda_s^{k+1} - \lambda_s^k\|_\infty$
    g) Calculate $\Delta t_{\lambda_p}$ and $\Delta t_{\lambda_s}$:
    $$\begin{aligned} \Delta t_{\lambda_p} &= \Delta t^n \frac{\lambda_{p_{lim}}}{\lambda_{p_{max}}} \\ \Delta t_{\lambda_s} &= \Delta t^n \frac{\lambda_{s_{lim}}}{\lambda_{s_{max}}} \end{aligned}$$
    h) Set $\Delta t^{k+1} = \min(\Delta t_{\lambda_p}, \Delta t_{\lambda_s})$
    i) Set $k = k + 1$
end
```

TSOpt Implementation

As I discussed in the fourth chapter, TSOpt has the components to accommodate adaptive time-stepping. Since this framework for adaptive time-stepping exists, I was able to implement a significant portion of algorithms 2 and 3. This required the following components:

- 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].

Though the basic foundation of the adaptive simulators exists, there are still components I need to implement or integrate with one another. Currently, I need to integrate the `Spline` package with the state storage class, to allow sampling of the reference states at any time, as needed by my proposed algorithms above. Also, I intend to examine the feasibility of creating an algorithm that allows adaptive checkpointing's forward evolution component to stop at the time level dictated by the adjoint evolution.

6.3 A Testing Framework for TSOpt

The convergence theory I had established in this proposal would not mean much if the forward, the derivative and the adjoint simulators used to numerically solve the optimal control problem exhibited undesirable properties or incorrect behavior.

We are primarily concerned with the following: will the solutions of the discretized (reference, linearized and adjoint) equations converge to the solution of the continuous equations? By the equivalence theorem, consistency and stability is equivalent to convergence for finite difference schemes. One type of incorrect behavior, hence, could mean inconsistency with the continuum differential equation. If we refine the time grid enough, using a uniform mesh, could we reduce the local truncation error effectively? Also, how could we ensure the stability of the finite difference scheme used to solve the evolution equations?

Another type of incorrect behavior could mean an incorrect relationship between the different simulators (e.g., the derivative evolution does not stem from linearization of the forward evolution, etc.). If these relationships are not upheld, it does not make sense to attempt solving the optimal control problem since the adjoint evolution would yield incorrect adjoint states, which in turn yields an incorrect derivative of the objective function. We hence require a test to verify these relationships, and this test should be computationally efficient. It would suffice to apply this verification to one step of the forward, adjoint and sensitivity evolution. Luckily, utilities to verify adjoint and derivative relationships have been built in to `RVL` [Padula et al., 2009]. The remaining challenge then is to create a wrapper that makes a user-specified simulation compatible with `RVL` – creating a mapping between the user's data containers to elements living in mathematical spaces, represented in `RVL`.

Part of my proposal would be to incorporate a testing framework for `TSOpt` that performs these checks on the simulators. The idea would be that, if the user-supplied simulators pass these tests, we can guarantee that we have a convergent Jet. (Recall that a "Jet" is a collection of the reference, linearized and adjoint simulations.) This would imply that the base classes for the user-supplied simulator must enforce some attributes, so that executing the test is always possible. Which attributes should be incorporated into TSOpt, and which attributes should be expected from the user-specified simulators, deserves more thought.

6.4 Towards an Optimization Framework

All this proposed work, of course, assumes that the unconstrained and constrained optimization algorithms are available to TSOpt. (Recall that the convergence proof presented in this proposal concerned Moré's unconstrained TR algorithm.) My future work, hence, will involve contributing to (and restructuring) `RVL`'s software packages for unconstrained and constrained optimization, respectively called `umin` and `cmin` [Padula et al., 2009].

Currently in the `umin` package, we have a robust implementation of the LBFGS algorithm. I plan to implement Moré's TR algorithm to add to `umin`'s framework. Once this is accomplished, I can attempt to solve the optimal control problem constrained by the Black-Oil equations (5.21), using a barrier method to enforce the explicit constraints. I intend to match the overall behavior I observed when I had solved the same problem at ExxonMobil. To obtain a more robust solver for the optimal control problem, however, I cannot solely depend on penalty and barrier methods as a means of handling explicit optimization constraints. It is also necessary to consider building a framework for constrained optimization. I intend to, hence, create a SQP optimization framework in `RVL`, using the `Alg` framework as the base. (This will be implemented in the `cmin` framework.) Some possible algorithms of interest are the linesearch SQP algorithm, and possibly Heinkenschloss and Vicente's TR-SQP algorithm [Heinkenschloss and Vicente, 2001].

6.5 Putting it All Together

After I create the adaptive forward and adjoint Black-Oil simulator, implement Moré's TR algorithm (in `umin`) and create the SQP framework in `cmin`, there are a variety of interesting problems that I wish to explore. First, I wish to study the relationship between the adaptive time stepper's tolerance parameter and the quality of the gradient adaptive evolution schemes produce, via the adjoint-state method. From the initial analysis I have showed, we expect that lowering the tolerance will increase the quality of the gradient. Also, suppose we used an iterative linear solver instead of a sparse-direct solver. How would the associated parameters in the iterative linear solver affect the quality of the gradient? This is another question I wish to answer using the code I will establish in TSOpt.

Ultimately, however, I intend to address the following question with my dissertation: can we guarantee convergence of the optimal control problem by tuning various algorithmic tolerances? Again, my preliminary analysis shows that the theoretical answer is yes. In my dissertation, I wish to show numerical results that support my claim.

Bibliography

R. W. Brankin, I. Gladwell, and L. F. Shampine. RKSUITE: A suite of explicit Runge-Kutta codes. In *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*, 2004.

John Burkardt. Spline: Interpolation and approximation of data, 2007. URL `http://people.sc.fsu.edu/ burkardt`. Data Accessed: 9.19.09.

Richard Carter. On the global convergence of trust region algorithms using inexact gradient information. *SIAM J. Numerical Analysis*, 28:251–25, 1991.

Todd 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.

Andrew Conn, Nicholas Gould, and Philippe Toint. *Trust-Region Methods*. SIAM, 2000.

Stanley Eisenstat Dembo and Trond Steihaug. Inexact newton methods. *SIAM Journal on Numerical Analysis*, 19:400–408, 1982.

M. Galassi and J. Theiler. Gnu scientific library. `http://www.gnu.org/software/gsl/`, 2009. Date accessed: 3/25/09.

Mark Gockenbach, William Symes, Daniel Reynolds, and Peng Shen. Efficient and automatic implementation of the adjoint state method. *ACM TOMS*, 28(1):22–24, 2002.

Andreas Griewank and Andrea 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. *Numerishe Mathematik*, 87:247–282, 1999.

J. Hahn. *Introduction to the Theory of Nonlinear Optimization*. Springer-Verlag, 2nd edition, 1996.

Matthias Heinkenschloss and Luis 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.

Michael Hinze and Julia 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.

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.

David Kincaid and Ward Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. Brooks/Cole, 2002.

Urs Kirchgraber. Multi-step methods are essentially one-step methods. *Numerishe Mathematik*, 48:85–90, 1985.

John Denholm Lambert. *Numerical Methods for Ordinary Differential Equations: The Initial Value Problem*. Wiley & Sons, 2000.

Shentai Li and Linda 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.

Anthony D. Padula, Shannon D. Scott, and William 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.

Donald Peaceman. *Fundamentals of Numerical Reservoir Simulation.* Elsevier, 1977.

Jean-Michel Renders and Stephane 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.

Robert Skeel. Thirteen ways to estimate global error. *Numerishe Mathematik*, 48: 1–20, 1986.

Endre Suli and David Mayers. *An introduction to Numerical Analysis*. Cambridge University Press, 2003.

William Symes. A time-stepping library for simulation-driven optimization. Technical report, Rice University, TRIP, 2006.

Wiegand, El-Bakry, and Matthias Heinkenschloss. Adjoint calculations for a reservoir management problem. In *SIAM Annual Meeting 2008*, 2008.