RICE UNIVERSITY

A C++ Class Supporting Adjoint-State Methods

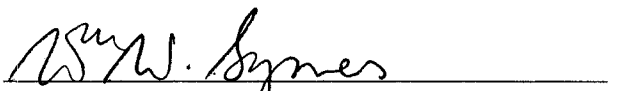
by

Marco U. Enriquez

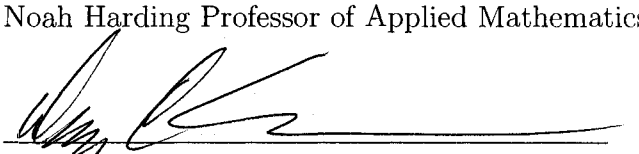
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Arts

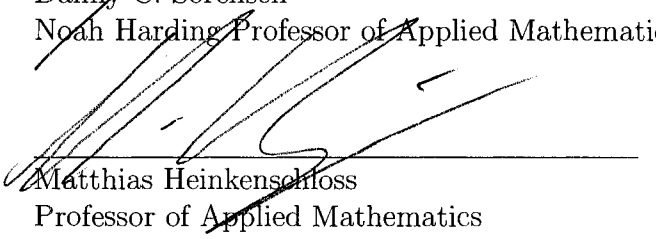
APPROVED, THESIS COMMITTEE:



William Symes, Chair
Noah Harding Professor of Applied Mathematics



Danny C. Sorensen
Noah Harding Professor of Applied Mathematics



Matthias Heinkenschloss
Professor of Applied Mathematics

HOUSTON, TEXAS

NOVEMBER 2007

Abstract

A C++ Class Supporting Adjoint-State Methods

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. There are instances, however, where the required state for the adjoint-state evolution is not readily accessible; consider large-scale problems, for example, where the entire simulation history is not saved to conserve memory. This thesis introduces a C++ state-access class, `StateHistory`, to support a myriad of solutions to this problem. Derived `StateHistory` classes implement a (simulation) time-altering function and data-access functions, which can be used in tandem to access the entire state history. This thesis also presents a derived `StateHistory` class, `GriewankStateHistory`, which uses Griewank's optimal checkpointing scheme. While only storing a small fraction of simulation states, `GriewankStateHistory` objects can reconstitute unsaved states for a small computational cost. These ideas were implemented in the context of TSOpt, a time-stepping library for simulation-driven optimization algorithms.

Acknowledgements

I would like to thank my advisor Dr. William Symes not only for defining fascinating research for me to pursue, but for his continued help and guidance. His joy and passion for scientific computing has been wildly contagious, for which I am grateful. I would also like to express my gratitude to Dr. Matthias Heinkenschloss; I have greatly benefited from, and enjoyed, discussing simulation driven optimization with him. Dr. Dan Sorensen and Dr. Jan Hewitt (along with Dr. Symes), through the Thesis Writing class, have greatly helped both my writing and presentation skills – for which I am also grateful. Their hilarious anecdotes on the "dos" and "don'ts" of technical writing will always stay will me.

I would also like to thank the people here at CAAM department that have helped me in various ways. First, I would like recognize the department staff: Daria Lawrence, Brenda Aune, Ivy Gonzalez, Theresa Chatman and Fran Moshiri. All of them have helped manage my graduate life, and they have also given me peace of mind when I needed it. Second, I would like to recognize my colleagues and friends; I certainly would not have come this far without their support. In particular, I would like to thank: Fernando González del Cueto, Leila Issa, Tony Kellems, Ryan Nong, Mili Shah, Adam Singer, Tanya Vdovina, Kirk Blazek, Mona Sheikh, Jay Raol and Joanna Papakonstantinou.

Last, but not least, I would like to thank my mom and dad not only for migrating the family to this country so I may get a better education, but for always believing in me.

This work was partially supported by The Rice Inversion Project (TRIP) and also the Rice VIGRE program. Their support is greatly appreciated.

Contents

Chapter 1

Introduction

In its simplest form, optimal control problems can be written as

$$\min_{c \in C} \quad J(c) = \hat{J}(u(c)) \tag{1.1}$$

where $u(c)$ solves the differential equation

$$\frac{du}{dt}(t) = f(u(t), c, t) \quad \text{for} \quad t \in [0, T] \tag{1.2}$$

$$u(0) = 0\,, \tag{1.3}$$

C is a Hilbert space, $u \in \mathcal{U}$ for some state Hilbert space $\mathcal{U}$, $\hat{J} : \mathcal{U} \to \mathbb{R}$ and f is a function that models a dynamical system. In order to use Newton-based optimization algorithms to solve this problem, it is necessary to calculate the gradient of the objective function $J(c)$. The gradient $\nabla J(c)$ is the linear map such that

$$\langle \nabla J(c), \delta c \rangle_C = \lim_{h \to 0} \frac{1}{h} \left(J(c + h\delta c) - J(c) \right) \quad \text{for all } \delta c. \tag{1.4}$$

A common method to calculate the gradient (1.4) of the objective function (1.1) is through the iterative algorithm called the *adjoint-state method* [12].

Throughout this thesis, 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.*

Adjoint-state methods incur a cost roughly equivalent to the cost of numerically solving the differential equation (1.2) [1], [16]. Despite this cost, adjoint-state methods are efficient because they are not affected by the size of the control parameter. Adjoint-state methods are defined by a backward-in-time evolution equation that depends on all of the forward states. This dependence, however, poses a problem for computational implementations of adjoint-state methods: what happens in the case where all the forward states are not immediately accessible? This is a reasonable concern for large-scale problems.

The problem of accessing unsaved simulation states for use in adjoint evolutions has been solved in various ways. Such established solutions include storing the whole state history (despite its memory cost), resimulating from the initial time until the time of interest t and storing intermediate states and resimulating from the nearest stored state until the time of interest t (called "Checkpointing", which will be discussed in a later chapter) [6]. All these solutions, however, have a commonality that can be abstracted and exploited; all aforementioned approaches provide the adjoint-

state algorithm some method of generating (and accessing) all simulation states, even if only a fraction of the simulation history was saved. This observation leads to the main focus of this thesis.

This thesis describes an abstract software type, a C++ class called `StateHistory`, which contains generic (simulation) time-setting and state-accessing methods. The various strategies for accessing (and possibly reconstituting) forward states are then implemented in the subtypes' concrete inherited methods, hiding implementation details from the user. In turn, these subtypes' time-setting and state-accessing methods will enable the execution of the adjoint-state method – even if the entire simulation history is not saved.

Derived `StateHistory` classes implement five functions. The first function is called `getTime()`, which returns the current simulation time. The second function is called `getTol()`, which returns a tolerance that is the maximum allowable difference between two times so that they can be (numerically) considered equal. The third function is called `setTime()`, which changes the current simulation time to equal its scalar input. The next two functions are both called `getState()`. Depending on whether the function is declared `const` or not, `getState()` will return either a const reference or a mutable reference, respectively, to the state corresponding to the current simulation time. Using `setTime()` and `getState()` in tandem will provide access to a (forward) state at any time.

This thesis presents two derived `StateHistory` classes: `SmartStateHistory` and

`GriewankStateHistory`. `SmartStateHistory` is the simplest implementation of a derived `StateHistory` class; it simply stores all the forward states, which can be trivially accessed for use in the adjoint-state method. `GriewankStateHistory`, on the other hand, is a nontrivial derived class; it combines the functionality of the `StateHistory` class with Andreas Griewank's optimal checkpointing scheme to create a memory-efficient method for computing the adjoint-state method [6].

The `StateHistory` class hierarchy is implemented as a part of the software package TSOpt, "Time-Stepping for Optimization", which encapsulates simulators, its derivatives and adjoint derivatives in a single object [18]. TSOpt also provides an interface to access this data, for use in constructing necessary components needed by optimization algorithms, such as the objective function's gradient. TSOpt will be introduced and discussed in more detail in the subsequent chapters. Of course, adding the `StateHistory` class to TSOpt's framework comes with repercussions. In order to add the `StateHistory` class (and its derived classes) to TSOpt's framework, components that either accessed or stored the state vector needed to be changed. Due to abstraction and the innate dependencies in TSOpt, this implied updating many of the components to reflect the new locations and new data structures of certain TSOpt components.

The addition of the `StateHistory` class hierarchy also establishes the foundation for adding adaptive time steps, allowing time-stepping algorithms to take variable and error-controlled time-steps. This, in turn, will lead to quicker solution of large-scale

problems. Taking adaptive time-steps, however, imply that the forward, derivative and adjoint states will likely not exist at the same time levels – violating a crucial assumption of the adjoint-state method. Though this thesis only considers fixed time-steps, in the "Future Work" section, I will discuss how derived `StateHistory` classes' `setTime()` and `getState()` functions can be implemented to compensate for the discrepancy between the forward time-levels and the adjoint time-levels.

This thesis is structured as follows: Chapter 2 discusses the various approaches to solving simulation-driven optimization problems. It also describes TSOpt and its predecessor FDTD, and how they help solve simulation-driven optimization problems. Chapters 3 and 4 discuss background material, namely the mathematical foundation of TSOpt and its software structure. Chapter 5 presents the new `StateHistory` class, as well as describes two classes that derive from it: `SmartStateHistory` and `GriewankStateHistory`. Chapters 6 presents numerical results and Chapter 7 discusses future work and conclusions.

Chapter 2

Literature Review

To better understand the adjoint-state methods, it is necessary to discuss the mathematical problem that it stemmed from: the simulation-driven optimization problem. This chapter will first discuss contemporary approaches to solving simulation-driven optimization problems. Then, it will introduce software packages developed to aid in solving such problems, including TSOpt – the software framework for the research discussed in this thesis.

There are two main branches of strategies in solving simulation-driven optimization problems: derivative-free algorithms and gradient-based algorithms. Two prominent types of derivative-free algorithms are direct-search algorithms and stochastic algorithms. Direct-search algorithms' are often used when it is difficult or impossible to obtain derivative information. Though Kolda et al. in [11] acknowledge that though it is typically impossible to prove second-order convergence for direct search methods, direct search methods reliably find local minimizers in practice. Stochastic algorithms, such as genetic algorithms or simulated annealing, are attractive in practice since they often converge to global minima given a large number of iterations [16].

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 [16].

Gradient-based algorithms (such as Newton and its variants), on the other hand, guarantee decrease of the objective function per iteration while usually requiring fewer forward evaluations than derivative-free algorithms [11], [15], [16]. There are, however, two major drawbacks to gradient-based algorithms. First, gradient-based algorithms are more prone to being "stuck" at local minima. Second, computing the gradient may be very difficult or computationally costly. In this thesis, I will be focusing on gradient-based algorithms.

The two fundamental gradient-based strategies for solving simulation-driven optimization problems go under the names "Optimize-then-Discretize" and "Discretize-then-Optimize". Though these two approaches eventually lead to a discretized systems of equations, they are not always equivalent. I will now compare and contrast the two strategies.

The strategy "Optimize-then-Discretize" implies the optimality conditions are first derived for the continuum problem, and then the optimality conditions are discretized via time-stepping methods. Jahn, among many others, derived explicit formulas for the continuous necessary optimality conditions for control problems [9]. Hager discussed this strategy in [7]. Using the continuous optimality conditions, Hager established a relationship between the continuous optimal control problem and the

discretized optimal control problem. He then 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 which guarantee that the Runge-Kutta discretization of the optimal control problem will have the same order of convergence as the Runge-Kutta numerical integrator.

The strategy "Discretize-then-Optimize," alternatively, implies that the continuum problem should be discretized first, and then subjected to (discrete) optimality conditions. Note that in [7], Hager actually established many instances where the strategy "Optimize-then-Discretize" is equivalent to the strategy "Discretize-then-Optimize". Brouwer and Jansen employed the "Discretize-then-Optimize" strategy, along with the Steepest Descent optimization method, to determine optimal valve setting for smart wells [1]. 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.

A software package that accommodates the two gradient-based strategies is the FDTD, or "Finite Difference Time Domain" package [4]; 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 can accommodate the two gradient-based strategies because each approach eventually

yields discrete equations. In turn, these discrete equations can be solved by FDTD. FDTD can 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.

TSOpt – the "Time Stepping for Optimization" Package – succeeded FDTD [18]. TSOpt is similar to FDTS 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 FDTS 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 [14].

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.

Currently, there are not many software packages that integrate forward, adjoint and derivative simulations. It is even less common to find software packages that use the adjoint-state method to obtain gradients for optimal control problems, such as TSOpt. Of all the software packages I examined, Sandia National Laboratory's Rythmos is the most similar to TSOpt; Rythmos is a "transient integrator" of differential equations that uses time-stepping algorithms implemented in C++. Like TSOpt, Rythmos uses advanced C++ coding techniques, such as templating and class hierarchies, to create inter-operating components to solve differential equations [2]. Currently, Rythmos is "aimed at supporting operator-split algorithms, multiphysics applications, block linear algebra and adjoint integration" [2]. However, unlike TSOpt, Rythmos does not currently support gradient calculation via adjoint-state methods, which can be a drawback when solving large-scale simulation-driven optimization problems.

Chapter 3

TSOpt's Mathematical Framework

Before discussing the `StateHistory` class, it is crucial to describe the underlying mathematical framework that motivates TSOpt's existence. Recall that the mathematical formulation of the problem that TSOpt (in conjunction with an optimization algorithm) solves can be expressed as the following optimization problem with respect to the control parameter, c:

$$\min_{c \in C} \quad J(c) = \hat{J}(u(c)) \tag{3.1}$$

where $u(c)$ solves the differential equation

$$\frac{du}{dt}(t) = f(u(t), c, t) \quad \text{for} \quad t \in [0, T] \tag{3.2}$$

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

C is a Hilbert space, $u \in \mathcal{U}$ for some state Hilbert space $\mathcal{U}$, $\hat{J}$ is a differentiable mapping such that $\hat{J} : \mathcal{U} \to \mathbb{R}$ and f is a function that models a dynamical system under consideration. To provide the optimization algorithm with the information needed

to solve the problem (3.1), TSOpt defines three abstract solution algorithms: the forward evolution (which enforces the differential equation constraint), the adjoint evolution (through which the gradient of the objective function is obtained), and the derivative evolution (which is used for checking the adjoint states and computing linearized terms). These abstract algorithms, however, must be provided with concrete implementations before they can be used.

3.0.1 The Forward Grid

Let us first focus on the system of differential equations

$$\frac{du}{dt}(t) \;=\; f(u(t), c, t) \tag{3.4}$$

that constrains our optimization problem (3.1), and the mathematics behind enforcing this constraint.

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

$$\sum_{j=0}^{N-1} \alpha_j v_{n+j} = \Delta t \sum_{j=0}^{N-1} \beta_j \Phi^j(v_{n+j}, c)\,, \tag{3.5}$$

where $\{v_n\}$ are the solution approximations, $\{\Delta t\}$ are the time-steps, $\{\alpha_i\}$ 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). By noting that multi-step methods can be written as one-step methods [10] and by introducing the discrete dynamic operator H, we can write (3.5) as

$$u_{n+1} = u_n + \Delta t H^n(u_n, c, \Delta t), \quad n = 0, 1 \ldots, (N-2)\,, \tag{3.6}$$

$$u_0 = 0\,. \tag{3.7}$$

The states $\{u_n\}_{n=0}^{N-1}$ can represent one or many time levels of the approximate dynamical system solution, depending on the problem and time-stepping scheme being considered. For example, in order to solve a second-order wave equation for the displacement field with a three-level leap-frog scheme, each state u_n represents two successive time levels of the discrete displacement.

In TSOpt, taking a forward step means taking one time step (i.e., performing the update (3.5)) in solving the equations (3.2). The term *forward grid* then refers to the collection of time levels $\{n\,\Delta t\}_{n=0}^{N-1}$.

3.0.2 Forming the Gradient for a Least-Squares Objective Function

Ultimately, TSOpt's goal is to create abstract operators that generate necessary data structures for Newton-based optimization algorithms, such as the gradient (or Hessian) of the function we are trying to minimize, J. Since this thesis' focus is the adjoint-state method, this section will focus on formation of the objective function's (3.1) gradient, ∇J.

In order to obtain the gradient of J, we first need to define the function. It is common for J to be posed as the least-squares function

$$J(c) = \frac{1}{2} \|SU(c) - d\|_R^2 \tag{3.8}$$

where S is a sampling operator (which is assumed to be linear), $U(c)$ is a vector composed of N states, with each state $u_i \in \mathbb{R}^M$

$$U(c) = (u_0\, u_1\, u_2 \ldots u_{N-1})^T \in \mathbb{R}^{MN},$$

and $d(t)$ is the observed data. (Note the subscript R on the norm, which represents the range of observed data from the simulator. In any finite dimensional space, any inner product $\langle x, y \rangle$ can be represented as $x^T R y$, where R is a symmetric positive definite matrix that acts as a weight.) If the error function J is written as the least squares function (3.8), it is possible to derive a closed-form equation for the gradient of J.

First define a discrete dynamic operator $A[U, c]$, as follows:

$$A[U,c]_n = \begin{cases} u_0 - u_0^{given} & \text{if } n = 0\ , \\ u_n - (u_{n-1} + \Delta t H^{n-1}(u_{n-1}, c, \Delta t)) & \text{if } n = 1, \ldots, (N-1)\ . \end{cases}$$

We can derive an expression for the gradient by taking the first-order Taylor expansion

of J:

$$\begin{aligned}\langle \nabla J(c), \delta c \rangle &= DJ(c)\,\delta c \\ &= \langle SU(c) - d\,,\, SD_cU(c)\,\delta c \rangle \\ &= \langle S^T\,[SU(c) - d]\,,\, D_cU(c)\,\delta c \rangle\,.\end{aligned}$$

By considering the derivative of $A[U,c]$ with respect to c, we obtain an expression for $D_cU(c)\delta c$:

$$0 = D_c(A[U,c])\,\delta c\,, \tag{3.9}$$

$$0 = D_cA[U,c]\,\delta c + D_UA[U,c]D_cU(c)\,\delta c\,, \tag{3.10}$$

$$\Rightarrow D_cU(c)\,\delta c = -D_UA[U,c]^{-1}D_cA[U,c]\,\delta c\,. \tag{3.11}$$

Substituting $D_cU(c)\delta c$ in the derivation of the $\nabla J(c)$ yields:

$$\begin{aligned}\langle \nabla J(c), \delta c \rangle &= -\langle S^T\,[SU(c) - d]\,,\, D_UA[U,c]^{-1}D_cA[U,c]\,\delta c \rangle \\ &= -\langle D_cA[U,c]^T(D_UA[U,c]^T)^{-1}S^T\,[SU(c) - d]\,,\, \delta c \rangle\,.\end{aligned}$$

Finally, by comparing inner products on both sides of the equality, we can conclude that the formula for the gradient of J evaluated at c can be neatly written as:

$$\nabla J(c) = D_cA[U,c]^T(D_UA[U,c]^T)^{-1}S^T\,[SU(c) - d]\,.$$

Since we now have expressions for both the error function J and its gradient, we may choose to use a variety of Newton-based optimization routines to minimize J.

Efficiently Calculating $(D_U A[U,c]^T)^{-1}$:

Though an inverse of a general matrix in $\mathbb{R}^{MN\times MN}$ has a computational cost of $O(M^3N^3)$, the inverse of $(D_U A[U,c]^T)$ can be computed at a cost of $O(MN)$ due to the way the discrete dynamic operator $A[U,c]$ is defined. Recall that the n^{th} element of the vector $A[U,c]$ is defined as:

$$A[U,c]_n = \begin{cases} u_0 - u_0^{given} & \text{if } n = 0 \ , \\ u_n - (u_{n-1} + \Delta t H^{n-1}(u_{n-1}, c, \Delta t)) & \text{if } n = 1, \ldots, (N-1) \ . \end{cases}$$

We can then see that $D_U A[U,c]$ will be the following lower triangular (more specifically: *lower bidiagonal*) matrix:

$$\begin{bmatrix} I & 0 & 0 & \ldots & 0 \\ -I - \Delta t D_{u_0} H^0(u_0, c, \Delta t) & I & 0 & \ldots & 0 \\ 0 & -I - \Delta t D_{u_1} H^1(u_1, c, \Delta t) & I & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & I \end{bmatrix}$$

In turn, this implies that $(D_U A[U,c]^T)$ is an *upper* bidiagonal matrix. Since calculating an inverse is equivalent to solving a linear system, we can obtain $(D_U A[U,c]^T)^{-1}$

by solving a bidiagonal system with backward substitution. For a dense matrix, backward substitution has the computational cost of $O(M^2N^2)$. The bidiagonal structure of this matrix, however, allows us to compute the inverse for the cost of $O(MN)$.

3.0.3 The Adjoint Grid

It is possible that explicitly forming the gradient of the objective function (3.1) could be very expensive to compute, or a closed form for the gradient cannot be obtained. TSOpt is designed to handle these cases by using the adjoint-state method. Recall that the adjoint-state method is an iterative algorithm that computes the gradient of the objective function (3.1).

The adjoint method differs from the forward evolution equation because it requires solving the following problem backwards in time. Given an *adjoint state* field w in the adjoint state space $\mathcal{W}$, one must solve:

$$\frac{dw}{dt}(t) = -D_u f(u(t), c, t)^* w(t) - S^*(Su(t) - d(t)) \tag{3.12}$$

$$w(T) = 0\,. \tag{3.13}$$

Recall that S is the sampling operator, $d(t)$ is the data and t lies in the interval $[0, T]$.

Using the discrete dynamic operator H mentioned in (3.6), the corresponding

evolution equation can be written as:

$$w_n = w_{n+1} + \Delta t(D_u H^n[u_n, c, \Delta t])^* w_{n+1} + S^*(SU(c) - d_n)\,, \tag{3.14}$$

$$n = (N-2), \ldots, 0\,, \tag{3.15}$$

$$w_{N-1} = 0. \tag{3.16}$$

The adjoint is then obtained by performing the accumulation

$$\sum_n D_c H^n[u_n, c, \Delta t]^* w_n\,, \tag{3.17}$$

which can be performed within the same loop as the backward evolution (3.14). The full derivation of this can be found in [17] and [18].

Note that in order to perform the backward evolution (3.14), we need access all the forward simulation states $U = (u_0\, u_1\, u_2\, ...\, u_{N-1})^T \in \mathbb{R}^{MN}$, where each u_i for $i = 0, 1, ..., (N-1)$ is obtained from performing the updates (3.5).

3.0.4 The Derivative Grid

The derivative grid is used in TSOpt to obtain "sensitivities", if needed by the problem model. This collection of derivative states $\{\delta u\}$ can be obtained by solving another

problem, often referred to as the *sensitivity equations*

$$\frac{d\delta u}{dt}(t) = D_u f(u(t), c, t)\delta u(t) + D_c f(u(t), c, t)\delta c\,, \tag{3.18}$$

$$\delta u(0) = 0\,. \tag{3.19}$$

Note that in (3.18) that δc is the perturbation in controls. The sensitivity equations can be solved discretely by performing the following update

$$\delta u_{n+1} = \delta u_n + \Delta t[D_u H^n(u_n, c, \Delta t)\delta u_n + D_c H^n(u_n, c, \Delta t)\delta c]\,, \tag{3.20}$$

$$n = 0, \ldots, (N-2) \quad , \tag{3.21}$$

$$\delta u_0 = 0. \tag{3.22}$$

The corresponding time levels, $\{n\Delta t\}_{n=0}^{N-1}$ is referred to as the *derivative grid.* Note that (3.18) is derived from the variation of (3.4) with respect to the state u and control parameter c.

Chapter 4

TSOpt's Software Framework

After explaning the mathematical framework of TSOpt, I can now discuss its C++ components and how these components function. Before examining TSOpt's components, however, it is necessary to introduce the software framework for TSOpt, the Rice Vector Library or RVL.

4.1 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 [14]. 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".

TSOpt uses a concrete data storage class called `LocalDataContainer` from RVL. 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" [14]. 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 { ... }
};
```

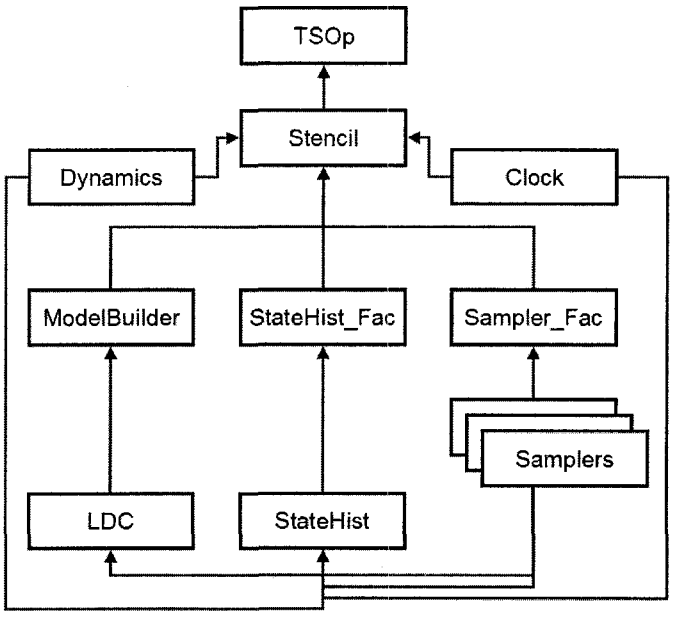


Figure 4.1: This figure shows TSOpt's components and the primary dependencies between these components. In this diagram, $A \rightarrow B$ implies that the class B has a dependence on the class A. This image will be useful to refer to throughout this section.

4.2 TSOpt's Components

TSOpt is comprised of many inter-operating components, which consist of the following classes: `ModelBuilder`, `Samplers`, `SamplerFactory`, `Statics`, `Dynamics`, `Clock`, `StateHistory`, `StateHistoryFactory`, `Stencil` and `TSOp`. I will now give an overview of each class' purpose, how it operates, and how it fits into TSOpt's framework. For a more detailed discussion of each of these components, see [18].

As mentioned in the previous section, workspace in the form of `LocalDataContainers` must be provided to TSOpt (from RVL) in order to store the data structures needed by a simulation-driven optimization algorithm; this is what the `ModelBuilder`

mixin is for. Classes that derive from `ModelBuilder` must implement a method to generate `LocalDataContainers` for the simulation's state and control.

The `Statics` class of TSOpt is responsible for knowing the internal and external representations of the state and control (with respect to the simulation), as well as having the functionality to convert an internal representation to an external representation and vice-versa. This translation between internal and external representations of state and control is accomplished through sampling and interpolation; this is the what `Sampler` classes are for. The `Sampler` classes provide methods to sample the control, the forward states, the derivative states, or adjoint states. The class `SamplerFactory`, as its name implies, is responsible for generating `Sampler` objects – following the "Factory" design pattern discussed by the "Gang of Four" in [3]. The `Statics` class makes use of this `SamplerFactory` to create `Sampler` objects.

The `Dynamics` base class holds pure virtual declarations of the forward, derivative and adjoint time-stepping methods needed to run the simulation. These pure virtual declarations will have to be defined in derived classes of `Dynamics`, to reflect the time-stepping algorithm that the user chooses to implement.

There needs to be a component that actually keeps track of time in order to accompany the methods defined in `Dynamics`; this is exactly what the `Clock` class does. A `Clock` object allows access to the current simulation time, as well as the simulation start time and end time. Furthermore, the `Clock` object also knows the time-step length Δt.

Classes that derive from `StateHistory` are then responsible for allowing access to any (one) state in the simulation history. Derived `StateHistory` classes own an instance of a `ModelBuilder` object so that it has access to (state and control) workspace-building functions. Derived `StateHistory` classes also have extra functionality related to time-stepping; hence, derived `StateHistory` classes will also need copies of a `Dynamics` object, a `Sampler` object and a `Clock` object. The justification for this functionality, along with the explanation of `StateHistory`'s role in TSOpt, will be throughly discussed in the next chapter. The class `StateHistoryFactory` is then responsible for making `StateHistory` objects.

The `Stencil` class then uses a `ModelBuilder` object, multiple `SamplerFactory` objects, a `Clock` object, a `Dynamics` object, and a `StateHistoryFactory` object to simulate a single time-step of a discretized dynamical system. The `Stencil` class follows the "Facade" design pattern [3]; abstractly, it provides a *unified* interface that orchestrates a set of other interfaces to perform one time-step.

The `TSOp` class is at the highest level of the TSOpt hierarchy. The `TSOp` class makes use of `TS` functions, which are capable of executing the time-step methods defined in `Stencil` (through the `Dynamics` object) multiple times, until a `Clock` object declares that the simulation should stop. (This occurs when the final simulation time has been reached.) The `TSOp` class is capable of running the full time-stepping simulation in order to obtain the state vector, the derivative state vector, or gradient of the objective function (via the adjoint state method). The output of `TSOp` functions is then used

to define the objective function (3.1), its gradient, etc. In turn, this information can be accessed by optimization algorithms through interfaces – thus helping to solve the simulation-driven optimization problem.

Chapter 5

The StateHistory Class Hierarchy

This chapter describes `StateHistory` class, how it fits into the TSOpt framework and how it is utilized by other classes. Further, this chapter highlights two derived `StateHistory` classes: the `SmartStateHistory` class and the `GriewankStateHistory` class, which uses an optimal checkpointing scheme – allowing execution of the adjoint-state method even if the entire state history is not stored in memory.

5.1 The StateHistory Class

From the highest level of abstraction, `StateHistory`'s task is simple: to define abstract functions whose concrete implementations can be used to access an arbitrary state in the simulation history. The following code shows the essential methods behind the `StateHistory` base class, along with explanations of what each function does:

```
template<class Scalar>

  class StateHistory{
  public:
```

```
  /** sets current time to t */
  virtual void setTime(Scalar t) = 0;

  /** query for time */
  virtual Scalar getTime() const = 0;

  /** query for tolerance */
  NormType getTol() const { return tol; }

  /** returns non-const reference to current state */
  virtual LocalDataContainer<Scalar> & getState() = 0;

  /** returns const reference to current state */
  virtual LocalDataContainer<Scalar> const & getState() const = 0;
};
```

The function `setTime()` changes the simulation time to equal its C++ `<Scalar>` input. The `getState()` functions return a (const or mutable) reference to a simulation state corresponding to the current simulation time. Hence, first calling `setTime()` with a scalar input `t`, then calling `getState()` will return a const or mutable reference to the state corresponding to the time `t`. It should be noted that, `setTime()` must be called at least once before `getState()` is called; this ensures that the initial simulation state and time data are initialized. Otherwise, calling `getState()` will result in an error being thrown (thus halting program execution) since a non-existent simulation state is being accessed. Other error-checking mechanisms regarding the `setTime()` and `getState()` functions are discussed in subsequent sections of this chapter.

In the case where all the forward states are saved, the implementations of `setTime()` and `getState()` are straightforward; `setTime()` can be used to run the forward simulation, storing every forward state as it simulates, up to some time (dictated by its

<Scalar> input). The getState() function will then return the element of the state vector corresponding to the current simulation time. However, in the case where all states are *not* stored – which is not an unreasonable assumption – the implementations of setTime() and getState() become more complex; getState() must still somehow provide a reference to the state corresponding to setTime()'s input, even if that state was not stored. To help recover these unsaved states, StateHistory objects must have the ability to perform forward time-steps.

Since StateHistory objects encapsulate the forward simulation, derived StateHistory classes must own a copy of a ModelBuilder object (in order to generate workspace for the simulation), a Dynamics object (to access time-stepping routines), a Clock object (to keep track of simulation time during the time-stepping routine), as well as a (adjoint) Sampler object (to allow access to the externally-represented control).

Due to the abstract definitions of the setTime() and getState() functions, the implementation of recovering unsaved states can be encapsulated in *either* the setTime() function or the getState() function. I decided to give the function setTime() the task of changing the simulation time *and* procuring (if necessary) the state to correspond to that time. Hence, in both derived StateHistory classes discussed in this thesis, getState() is purely an accessor or "getter" function – it merely returns a reference to the last state simulated by setTime().

Since all the implementation and algorithmic work was placed inside the setTime()

function, for the remainder of this chapter, I will only discuss `StateHistory`'s derived classes and how they implement the `setTime()` function.

5.2 The Derived `StateHistory` Classes

5.2.1 The `SmartStateHistory` Class

The first derived `StateHistory` class is called `SmartStateHistory`. Despite its name, `SmartStateHistory` is not a very efficient class; `SmartStateHistory` simply stores *all* the simulation states into a vector of `LocalDataContainers` as it simulates.

Once a time request is received by `SmartStateHistory`'s `setTime()` member function, there are one of four things that could happen. First, if the time requested has already been stored, nothing needs to be done; the desired state can be accessed by `getState()`. Second, if the time requested falls outside the simulation time window, an error is returned. Third, note that for the derived `StateHistory` classes discussed in this thesis, only *fixed* Δt are being considered. Hence if the requested time does not align itself with the grid, an error is also thrown. Though it is possible to use interpolation to approximate states corresponding to a time not aligned with the grid, I chose to throw errors to enforce fixed time steps; since adaptive time steps are not being considered in this thesis, the requested time requested should always align with the grid. Exceptions to this should be regarded as potential simulation errors. Fourth, if the time requested is greater than the maximum time stored (but still falls in the simulation time window), a forward simulation must be performed up to the

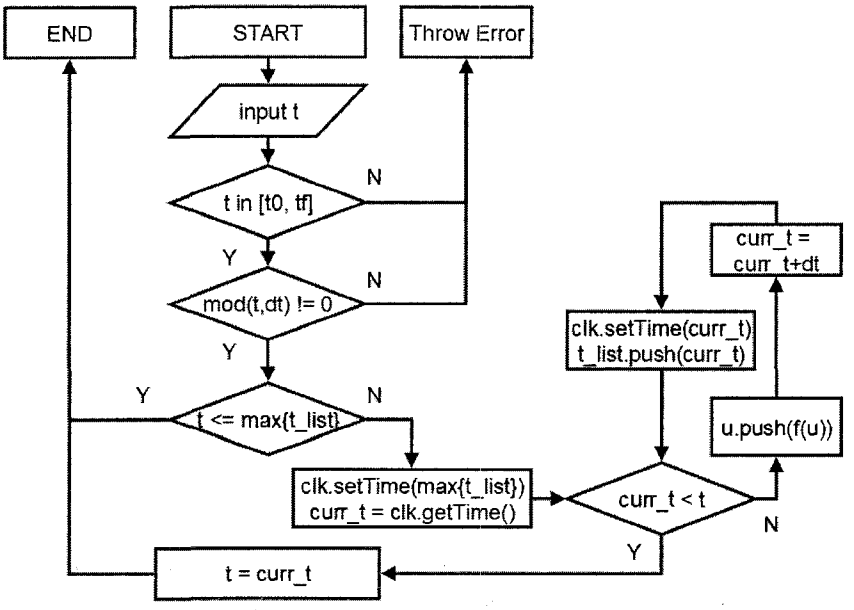


Figure 5.1: Flowchart of the `setTime()` function defined in `SmartStateHistory`. Note the time-stepping loop located in the lower-left corner of the flowchart.

time requested and every (new) simulated state must be saved. Figure 5.1 shows the flow chart for the `setTime()` algorithm.

One should note the following in Figure 5.1. First, notice that the initial two queries in the algorithm are error checking mechanisms; they ensure that the time requested is in the simulation time-frame and also that the time requested aligns itself with the grid.

Second, note the third query: if the time requested is less than than the maximum element in the time list, it implies that the time requested (as well as the corresponding state) is stored in memory and can be accessed. Otherwise, a forward simulation needs to take place; this forward simulation is initialized by setting the simulation's current

time to the maximum time list element.

Finally, note the time-stepping loop on the bottom-right corner of figure (5.1), which corresponds to the following pseudocode:

```
While (curr_t < t) {
u.push(f(u));     // Note u is the state vector, and f is some
                  //    multi-step (or one-step) time-stepping
                  //    algorithm. This line appends the next
                  //    simulated state to the state list
curr_t = curr_t + dt;     // Update current time
clk.setTime(curr_t);     // make curr_t the current simulation time
t_list.push(curr_t);     // append curr_t to the time list
}
```

At the end of the time stepping, the current simulation time (`curr_t`) should equal the time requested (`t`), since it is assumed that the time-step length dt is fixed. The full implementation of the `SmartStateHistory` class can be found in Appendix B.

5.2.2 Griewank Checkpointing

This section describes how checkpointing schemes work and why they are needed; it is crucial that this topic is introduced before discussing the `GriewankStateHistory` class.

In order to use the adjoint-state method (3.12) to obtain the objective function's

(3.1) gradient, the values of the state vector needs to be accessed in *reverse*. (Recall that this is due to the definition of the discrete adjoint evolution (3.14).) This, however, can be problematic when the state vector is very large; repeatedly recalculating the state vector is computationally expensive, and storing the whole state vector can be costly in terms of memory. I will illuminate this point by discussing two naïve strategies for the backward access of the simulation states.

The first strategy is relatively simple: one could just simply record the full state vector into a buffer, then access this buffer to obtain state vector values. It should be noted, however, that recording the full vector could incur a prohibitive memory cost. This, in turn, could slow the execution of TSOpt. For example, consider Reverse Time Migration (RTM) – a geophysical imaging technique that uses the time-reversibility property of the wave equation. RTM techniques apply the adjoint state method to the wave equation in order to find sources or scatterers. For a typical 3D RTM problem, storing the entire simulation history requires $O(10^{13})$ bytes. This memory cost could cause use of disk-swapped memory, adversely affecting the program execution time.

The second strategy is also simple: one can keep re-simulating (i.e., performing the forward evolution) from $n = 0$ until the desired level. Since the adjoint method runs backward in time, the first iteration requires a forward simulation up to the N^{th} state. Then, at the second iteration, it simulates until the $(N-1)^{th}$ state, and so on. At a computational cost of $\frac{N^2}{2}$, this strategy is generally prohibitive for large problems. In a typical 3D RTM problem, this strategy could incur a cost of $O(10^{38})$

flops.

To reduce the costs associated with reconstituting states for use in adjoint-state methods, Griewank used an algorithm called checkpointing [5]. The idea behind checkpointing is an intelligent combination of the two previously mentioned strategies: make a list of forward states to be saved (called checkpoints), choose a subset of these states and store them in an allotted buffer, and then forward-simulate from the nearest saved state until the time of interest. As the backward traversal progresses, the checkpoint list is consulted to update the saved states (in the buffer) such that they do not become obsolete. In effect, checkpointing eliminates the need to store the whole state vector while reducing the recomputation of states.

The computational cost of checkpointing algorithms vary with how one chooses the location of the checkpoints. For example, in [5] Griewank notes that recovering the entire simulation history using a uniform distribution of checkpoints will incur a cost of $O(\sqrt{N})$ flops. More sophisticated schemes will yield lower flop costs; Griewank used recursive checkpoint placement schemes to improve upon the aforementioned $O(\sqrt{N})$ cost. In fact, by using these recursive schemes, Griewank established the *optimal* checkpoint placement – allowing the entire simulation state history to be recovered at a cost of $O(log(N))$ flops. Before discussing Griewank's optimal checkpointing algorithm, however, more terminology needs to be introduced.

Griewank's algorithm uses three different types of simulation functions: a regular forward evolution, a "recording" forward evolution and an adjoint evolution. The

regular forward evolution will be denoted as

$$s_{i+1} \leftarrow f(s_i) \quad \text{for } i = 0, 1, \ldots N-1\,,$$

where the function call f is dependent on the forward states $\{s_i\}$ and s_0 is some fixed state in the forward state space S. If the forward evolution must record its generated state to the allotted buffer, it will be written similarly, except that the forward evolution function will be denoted as $\hat{f}$. The adjoint evolution is represented by:

$$\bar{s}_i \leftarrow \bar{f}(\bar{s}_{i+1}) \quad \text{for } i = N-1, N-2, \ldots 0\,,$$

where $\{\bar{s}_i\}$ denote the adjoint states, initialized by the terminal state $\bar{s}_N$ in the adjoint state space $\bar{S}$.

Further, the buffer that is initially allotted to Griewank's algorithm is treated as a Last-In-First-Out (LIFO) stack that supports the standard commands `push(`s`)` and `pop(`s`)`. (Here, `pop(`s`)` denotes that the popped state is written to s.) The `push(`s`)` and `pop(`s`)` commands will also be used to provide initial states in the forward and adjoint evolution. Using the terminology above, we can examine Griewank's `treeverse` algorithm, which incorporates the forward traversal, the adjoint traversal and a recursive checkpointing scheme.

Algorithm 1 Treeverse

Given: $s \in S$ and $\bar{s} \in \bar{S}$
Allotted: N_B buffers

`treeverse`$(base, start, finish)$
if $start > base$ **then** `push`(s)
for $i = base, \ldots start - 1$ **do** $s \leftarrow f_i(s)$
while $base << finish$ **do**
  pick $kidstart \in (start, finish)$
  `treeverse`$(start, kidstart, finish)$
end while
for $i = start, \ldots finish - 1$ **do** $s \leftarrow \hat{f}_i(s)$
for $i = finish - 1, \ldots start$ **do** $\bar{s} \leftarrow \bar{f}_i(\bar{s})$
if $start > base$ **then** `pop`(s)
return

The inputs to `treeverse` are as follows: the index *base* is defined as the time level reached by the forward evolution and the index range $[start, finish]$ as the range of time levels where we wish the adjoint evolution will take place (i.e., we wish to evolve from $\bar{s}_{finish}$ down to $\bar{s}_{start}$). Hence, a full forward and adjoint traversal can be achieved by making the call `treeverse(0,0,`N`)`.

Looking at algorithm 1 in more detail, we see that the `treeverse` algorithm first forward evolves from s_{base} to s_{start}. Since the adjoint evolution requires access to the forward states in backward order, the `treeverse` algorithm then produces checkpoints – denoted as the index *kidstart* – to help reduce the forward state's time level from $finish$ to a time level close to *start*. Here, "close" means that the forward states within the time levels *start* and $finish$ can all be recorded in the alloted buffer and reversed. (Also, in [5], Griewank notes that "close" is also the negation of the $<<$ notation in the while loop of algorithm 1.) Note that the recursive call in algorithm

1 is guaranteed to terminate; choosing $kidstart \in (start, finish)$ defines a bracketing algorithm which eventually leads to the index *start* being close to the index $finish$ – satisfying the criteria of the while loop. At the end of algorithm 1, the alloted adjoint state $\bar{s}$ will hold the correct adjoint state $\bar{s}_0$.

Given N_B buffers and N_S states such that $N_B \ll N_S$, Griewank derived a scheme for choosing *kidstart* in algorithm 1 such that it only adds logarithmic (i.e., $O(log(N_S))$) recomputation cost [6]. Figure 1 illustrates and explains this logarithmic cost in an example. One simple strategy that achieves this logarithmic complexity is by always choosing $kidstart = \frac{1}{2}(finish - start)$ in algorithm 1, essentially implementing a recursive bisection scheme [5]. Griewank, however, improved on the bisection strategy's logarithmic complexity by choosing a binomial partition function to determine the checkpoints. In [5], Griewank proved two desirable properties of his binomial strategy. First, he showed that the binomial strategy incurs *half* the computational cost of the bisection strategy. Second, he showed that using the binomial partition function leads to a distribution of checkpoints that minimized the number of forward recomputations; Griewank refers to this as "optimal checkpointing".

Griewank implemented a modified version of the `treeverse` algorithm in a package called *Revolve* [6]. Revolve differs from `treeverse` because it does not explicitly perform any of the operations in algorithm 1; Revolve only gives a set of instructions (to forward evolve, to adjoint evolve, to push to or pop from the LIFO stack, etc.) that, when followed, will perform the full adjoint evolution using the optimal check-

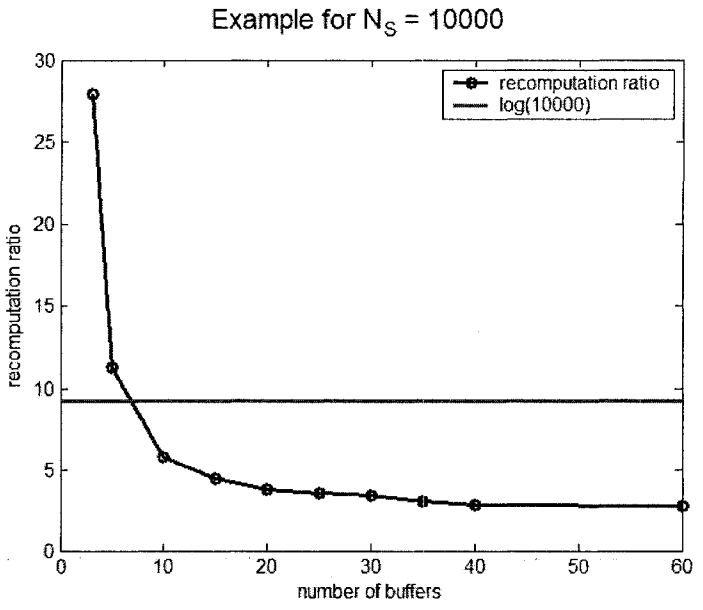


Figure 5.2: This is an example that uses the optimal checkpointing scheme for $N_S = 10000$. The computational work measure here is called the "recomputation ratio", which is defined to be the total number of forward simulations divided by N_S. We see here that given a very small amount of buffers, the recomputation ratio can be driven below $log(N_S)$.

pointing scheme. Revolve has two main phases in its execution. Given the number of time steps to be taken, the *scheduling phase* of Revolve determines the optimal checkpoint placement. Then, the *backward traversal phase* dictates what should be done with the stored checkpoints in order to complete the backward traversal of states; this phase determines if the saved checkpoints should be used, updated, or if a forward simulation (starting from a previously saved state) needs to be performed. Generally, Revolve is used such that the *scheduling phase* is immediately followed by the *backward traversal phase*. I show in the next section, however, that separating the execution of these two phases leads to efficient incorporation of Griewank's

checkpointing scheme into the derived `StateHistory` classes.

5.2.3 The `GriewankStateHistory` Class

The `GriewankStateHistory` is a more practical `StateHistory` subclass; through the use of the Revolve package, it combines the notion of checkpointing with the functionality of `StateHistory` in order to avoid saving a majority of the simulation states.

Due to the incorporation of checkpointing, `GriewankStateHistory`'s `setTime()` function is divided into two modes. The first mode is the forward mode, in which `setTime()` can only take an input time greater than or equal to the current simulation time. This mode is where all the forward simulations take place. This mode is also responsible for saving a few simulation states chosen by the Revolve program's *scheduling phase*. The second mode is the backward mode, which can only be started after the forward phase completes the forward simulation (i.e., when the current system time equals the final simulation time). During the backward phase, `setTime()` can be used to access any prior simulation state, even if that state was not stored. The `setTime()` function's backward phase will end once the current time equals the initial simulation time; this also evokes `setTime()` to enter its forward mode again. The following behavior can be observed in the state diagram in Figure 5.3. Note that this separation of modes is necessary due to the checkpointing scheme being employed in the backward phase; if the modes were allowed to switch at some time $t \in (t_{\text{init}}, t_{\text{final}})$ (i.e., in the middle of the simulation), the optimality of Griewank's

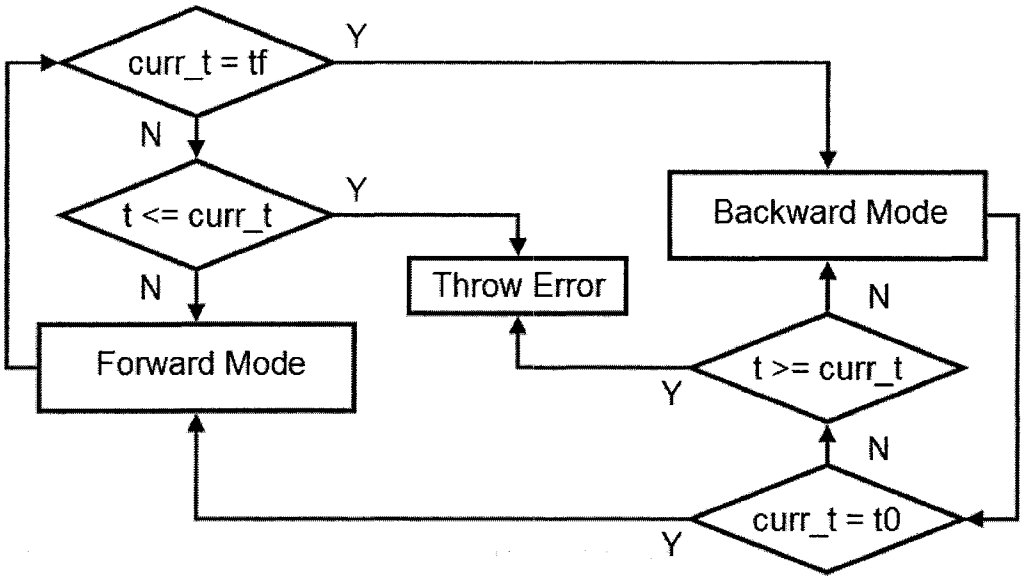


Figure 5.3: State Diagram for `setTime()` in `GriewankStateHistory`, for some scalar time input `t`. Transitions between states only happen when the current simulation time `curr_t` is either the initial simulation time `t0` or the final simulation time `tf`. Note that both Forward and Backward modes are responsible for updating `curr_t`.

checkpointing scheme will be lost since the checkpointed states can be altered. I will now discuss the implementation of the forward and backward modes.

The forward mode of `GriewankStateHistory`'s `setTime()` has two parts: first, compute the optimal checkpointing schedule (given a simulation time-window and a step length) by only using the *scheduling phase* of the Revolve program. I made this possible by giving the `GriewankStateHistory` class its own instance of a Revolve object. This gave the Revolve object "memory" – allowing it remember its current state inside an instantiation of a `GriewankStateHistory` object. In turn, this allows Revolve's execution to be stopped at the end of its *scheduling phase* such that it

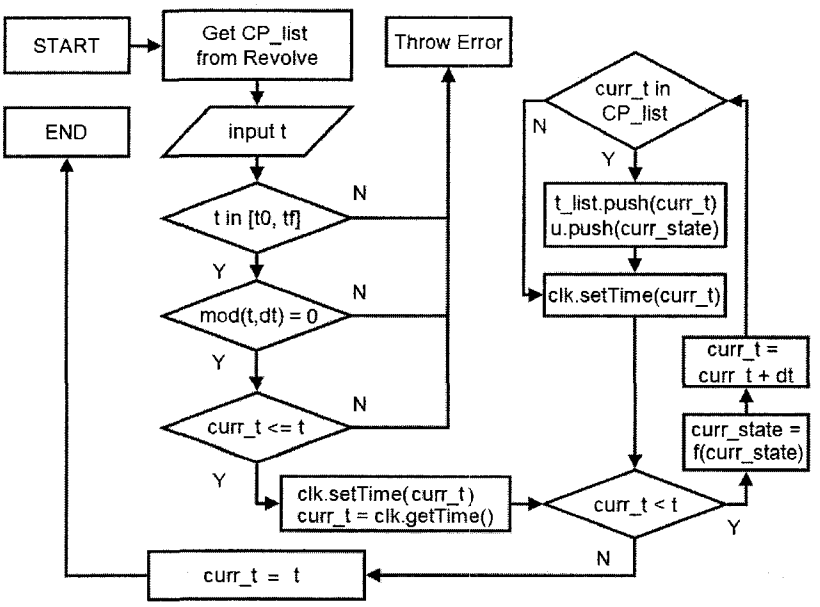


Figure 5.4: Flowchart of the forward mode of `setTime()` defined in `GriewankStateHistory`. Note the similarities and differences between this flowchart and the flowchart in Figure 5.1.

will be ready to start its *backward traversal phase* once it is executed again. The second part involves running `setTime()` as described in `SmartStateHistory`; this time, however, we only save the states and time levels corresponding to the computed checkpointed schedule. This part of `GriewankStateHistory`'s `setTime()` function is illustrated in figure 5.4.

Note that the flowchart in figure (5.4) is fairly similar to `SmartStateHistory`'s flowchart. The main difference between the two derived `StateHistory` classes lie in their respective time-stepping loops. `GriewankStateHistory`'s time-stepping loop can be written in pseudocode as follows:

```
While (curr_t < t) {
```

```
curr_state = f(curr_state);   // Overwrite curr_state with f(curr_state).
                              //    Note that f here is a one-step
                              //    or a multi-step time-stepping
                              //    algorithm.
curr_t = curr_t + dt;     // Update current time
if (in_CPList(curr_t) = true)    // poll checkpoint list
{
u.push(curr_state);     // Note u is the state vector.
                        //    This line appends curr_state
                        //    to the state list
t_list.push(curr_t)     // Append curr_t to the time list
}
clk.setTime(curr_t)     // make curr_t the current simulation time
}
```

The only feature in this algorithm that is different from `SmartStateHistory`'s `setTime()` function is the polling step "`if (in_CPList(curr_t) = true)`". The function `in_CPList()` determines if the input (always taken to be the current time) matches the list of checkpoint times obtained from Revolve. If this function returns `true`, the algorithm stores the current state; otherwise, it updates the current state by performing the forward evolution (3.5). The function `in_CPList()` is an array traversal scheme; using this traversal scheme is justifiable because the size of the

checkpointing list is always going to be much less than the total number of states. Hence, the array traversal should not be very expensive to execute during the time-stepping iteration; this single array traversal is still a better alternative than one forward resimulation for sufficiently large problems.

The backward mode of `GriewankStateHistory`'s `setTime()` then uses the *backward traversal phase* of the Revolve program. (Recall that the Revolve object's *forward phase* and the *backward traversal phase* have been separated.) Basically, given some time less than the current simulation time, `setTime()` in the backward mode follows the checkpointing procedure (as dictated by Revolve) to obtain the state corresponding to the requested time. This is a nontrivial task due to the fact that all the simulated states were not saved in this derived `StateHistory` class, unlike in `SmartStateHistory`. A detailed explanation of how Revolve's *backward traversal phase* works can be found in [6]. Figure 5.5 shows how Revolve's *backward phase* works, and the actions it necessitates on the stored states and stored times obtained from the forward mode of `setTime()`. The "instructions" that required pushing and popping state `LocalDataContainers` and time `LocalDataContainers` were accomplished through use of the C++ `<vector>` library's commands. Please refer to Appendix C for the implementation of the `GriewankStateHistory` class' forward and backward modes.

There is an inherent virtue of this approach that is not immediately apparent; by separating `setTime()` into two modes (utilizing Revolve's two separate phases),

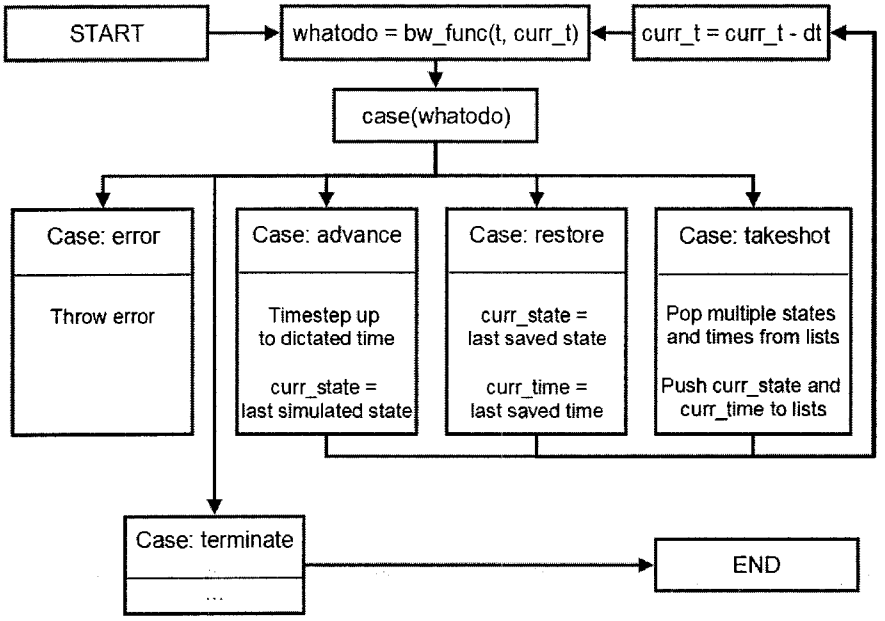


Figure 5.5: Flowchart for the backward mode of `setTime()` in `GriewankStateHistory`. The function `bw_func` makes use of `Revolve`, which in turn dictates actions to be taken during this phase. These actions will either: pop and push checkpointed states and checkpointed times using the vector of `LocalDataContainers` called `check_states` and `check_times`, forward simulate up to a certain time (and update `curr_state`), or restore a checkpointed state and checkpointed time. In the case that `Revolve` terminates irregularly, an error is thrown.

TSOpt only needs to run the full forward simulation *once* for every backward traversal needed. This, in turn, allows for efficient construction of the objective function's gradient.

Chapter 6

Numerical Results

This chapter verifies `setTime()`'s functionality, which is the main component of the `StateHistory` class. For each of the derived classes, `SmartStateHistory` and `GriewankStateHistory`, I will first highlight its error trapping mechanisms. Then, I will show through a simple example that `setTime()` behaves as described in the previous chapter.

6.1 Tests on `SmartStateHistory`'s `setTime()` Function

For these tests, consider the differential equation:

$$\frac{du}{dt} = (1 - u^2) \quad t \in [0.0, 0.09] \tag{6.1}$$

$$u: \quad \mathbb{R} \rightarrow \mathbb{R}^2 \tag{6.2}$$

$$u(0) = [0.5,\, 0.5]^T . \tag{6.3}$$

The solution to the problem (6.1) is $u(t) = \tanh(t + \tanh^{-1}(0.5))[1, 1]^T$. Solving this differential equation numerically via the forward Euler scheme, with a fixed time step

of $\Delta t = 0.1$, yields the following discrete evolution:

$$\begin{aligned} u_{n+1} &= u_n + \Delta t(1 - u_n) \text{ for } n = 0, 1, \ldots, 9 \\ u_0 &= [0.5, 0.5]^T . \end{aligned}$$

Using this discrete evolution, the following subsections examines `SmartStateHistory`'s error checking mechanisms and verifies its output.

6.1.1 Error Checking

Errors are handled in TSOpt through the use of C++ `try` and `catch` blocks. Should an error occur, an error message in the form of an `RVLException` is thrown, stopping the program and outputting the error. All error messages are generated by the respective derived `StateHistory` class. I will now discuss the cases that generate an error in `SmartStateHistory`'s `setTime()` function.

Case 1. User attempts to access a time that falls outside the time-window of simulation

```
CODE SEGMENT:
double t = 11;          // note that t is in [0.0, 0.09]
setTime(t);

OUTPUT:
terminate called after throwing an instance of 'RVL::RVLException'
  what():  Error: SmartStateHistory::setTime
  time requested = 11 is greater than
```

```
  final simulation time = 0.09.
```

Case 2. User attempts to access a time that is not aligned with the grid, though falling in the time-window of simulation.

```
CODE SEGMENT:
double t = 0.125;
setTime(t);          // note that dt = 0.01

OUTPUT:
terminate called after throwing an instance of 'RVL::RVLException'
  what():  Error: SmartStateHistory::setTime
  time requested = 0.125 is not an integer multiple of
  the proper time level, as dictated by dt = 0.01.
```

6.1.2 A Full Forward Traversal

This subsection shows a full forward traversal executed by the `setTime()` function. Since the simulation end time in this example is $t = 0.09$ (and it is assumed that the current simulation time is equal to the initial simulation time, $t = 0.0$), executing the `setTime()` function with the input 0.09 will run the full forward simulation.

```
CODE:
double t = 0.09;
setTime(t);

OUTPUT:
[Timelist]
0
0.01
0.02
0.03
0.04
```

```
0.05
0.06
0.07
0.08
0.09

[Sim. States]
u[0] = 0.5
u[1] = 0.5

u[0] = 0.5075
u[1] = 0.5075

u[0] = 0.514924
u[1] = 0.514924

u[0] = 0.522273
u[1] = 0.522273

u[0] = 0.529545
u[1] = 0.529545

u[0] = 0.536741
u[1] = 0.536741

u[0] = 0.54386
u[1] = 0.54386

u[0] = 0.550902
u[1] = 0.550902

u[0] = 0.557867
u[1] = 0.557867

u[0]  = 0.564755
u[1]  = 0.564755
```

The (forward) states generated by this time-stepping scheme can be easily verified through MATLAB, component-wise. Figure (6.1) plots the true solution $u(t)$ and the simulated states. Since `SmartStateHistory`'s `setTime()` function stores the entire

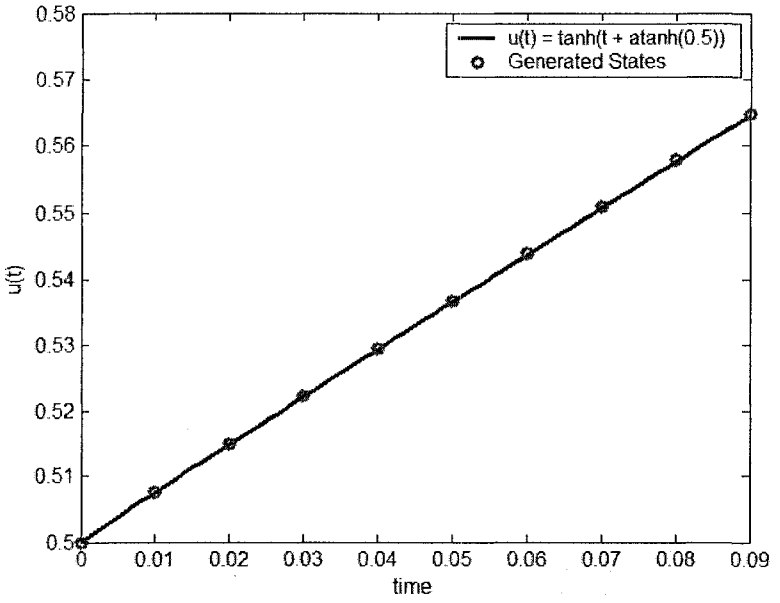


Figure 6.1: Plot of the true solution $u(t)$ of the differential equation (6.1) and the simulated states $\{u_i\}$

simulation history, the forward states can be easily accessed (through the `getState()` function) in the reverse order for use in the adjoint-state evolution.

6.2 Tests on `GriewankStateHistory`'s `setTime()` Function

For these tests, again consider the (discrete) evolution equation derived from the problem (6.1), with $\Delta t = 0.01$:

$$\begin{aligned} u_{n+1} &= u_n + \Delta t(1 - u_n) \text{ for } n = 0, 1, \ldots, 9 \\ u_0 &= [0.5, 0.5]^T. \end{aligned}$$

Recall that the class `GriewankStateHistory`, however, will only save a certain number of states in the state vector. In this example, the `GriewankStateHistory` object was allotted three buffers (which take the form of `LocalDataContainers`) to use in the checkpointing scheme.

6.2.1 Error Checking

This subsection discusses the instances that generate an error in `GriewankStateHistory`'s `setTime()` function. Note that, since this implementation of `setTime()` runs under two functional modes (the forward and backward mode), there are two extra error cases to check for.

Case 1. User attempts to access a time that falls outside the time-window of simulation

```
CODE:
double t = 22;       // note that t is in [0.0,0.09]
setTime(t);

OUTPUT:
terminate called after throwing an instance of 'RVL::RVLException'
  what():  Error: GriewankStateHistory::setTime
  time requested = 22 is greater than
  final simulation time = 0.09
```

Case 2. User attempts to access a time that is not aligned with the grid, though falling in the time-window of simulation.

```
CODE:
double t = 0.125;      // note that dt = .01
setTime(t);

OUTPUT:
terminate called after throwing an instance of 'RVL::RVLException'
  what():  Error: GriewankStateHistory::setTime
  time requested = 0.125 is not an integer multiple of
  the proper time level, as dictated by dt = 0.01.
```

Case 3. User attempts to access a time less than current simulation time during forward mode

```
CODE:
setTime(.05);         // perform a forward traversal to time .05
setTime(.04);         // try to backward traverse to time .04

OUTPUT:
terminate called after throwing an instance of 'RVL::RVLException'
  what():  Error: GriewankStateHistory::setTime
  time requested = 0.04 cannot be less than
  current time = 0.05 during forward phase
```

Case 4. User attempts to access a time greater than current simulation time during backward mode

```
CODE:
setTime(.09);         // perform the full forward traversal
setTime(.08);          // backward traverse to time .08
setTime(.09);          // try to forward traverse to time .09

OUTPUT:
terminate called after throwing an instance of 'RVL::RVLException'
  what():  Error: GriewankStateHistory::setTime
  time requested = 0.09 cannot be greater than
  current time = 0.08  during backward phase
```

6.2.2 A Full Forward and (a Partial) Backward Traversal

This section, as in the example of the `SmartStateHistory`'s `setTime()`, completes the forward simulation first; recall, however, that `setTime()` only has three state buffers and three time buffers (whose values are determined by the checkpointing scheme embedded in `GriewankStateHistory`). By completing the forward simulation, `setTime()`'s backward mode is invoked, allowing the user to access previously computed states in the simulation. In this example, the backward traversal is being performed down to $t = 0.04$.

```
CODE:
setTime(0.09);          // perform full forward traversal
setTime(0.04);          // backward traverse to t = 0.04

OUTPUT:
(Data that corresponds to forward traversal setTime(0.09))
curr_time = 0.0900

curr_state[0] = 0.5648
curr_state[1] = 0.5648

[Timelist]:
0.0000
0.0400
0.0700

[Sim. States]:
u[0] = 0.5000
u[1] = 0.5000

u[0] = 0.5295
u[1] = 0.5295

u[0] = 0.5509
```

```
u[1] = 0.5509

(Data that corresponds to backward traversal setTime(0.04))
current time = 0.0400

curr_state[0] = 0.5295
curr_state[1] = 0.5295

[Timelist]:
0.0000
0.0400
0.0500

[Sim. States]:
u[0] = 0.5000
u[1] = 0.5000

u[0] = 0.5295
u[1] = 0.5295

u[0] = 0.5367
u[1] = 0.5367
```

Though we did not have access to the full state vector, note that the current state (`curr_state`) and the current time (`curr_time`) hold the proper values – in both the forward traversal and the backward traversal.

Chapter 7

Future Work and Conclusions

This thesis establishes the state-accessing class hierarchy `StateHistory` as an addition to the software framework TSOpt. Derived `StateHistory` classes implement five functions, `setTime()`, a const and non-const version of `getState()`, `getTol()` and `getTime()`. All instances of the `setTime()` function, regardless of which derived `StateHistory` class it belongs to, can alter the current simulation time. The `getState()` functions return a (const or mutable) reference to the state associated with the current simulation time. The `getTol()` function returns the specified tolerance for the simulator, while the `getTime()` function returns the current simulation time. Together, the `setTime()` and `getState()` functions can be used to access to an arbitrary state in the simulation history, as long as this state exists in the forward grid. Through the `StateHistory` interface, execution of the adjoint-state evolution – even when the entire simulation state history was not stored – was made possible. There still are, however, computational issues involving the adjoint-state evolution that need to be considered. Three such issues that I intend to examine in the future are the repeated formation of gradients, the use of the software package OpenAD and

the use of adaptivity in adjoint-state methods.

For certain optimization algorithms it will be important to examine the case where the gradient needs to be generated repeatedly, such as the nonlinear conjugate gradient methods Fletcher-Reeves or Polak-Ribière [13] or the conjugate gradient method applied to the normal equations. These are cases where the adjoint-state method needs to be executed multiple times. This, in turn, implies multiple backward traversals of the state vector, which is unfortunately not supported by Revolve; Revolve inherently assumes that the backward traversal only needs to be performed *once* per every complete forward simulation.

Determining the optimal strategy for accomplishing multiple backward traversals is a nontrivial task. Though it would still be prudent to use Griewank's optimal checkpointing algorithm, there needs to be an external mechanism that will allow Revolve to perform the backward traversal repeatedly. Furthermore, this repeated backward traversal should be accomplished *without* repeating the forward simulation to reset the initial checkpointed states and times.

However, perhaps it is possible to bypass use of Revolve in TSOpt. It may be possible to instead use the automatic differentiation package OpenAD, which is the product of many pioneers in the field of automatic differentiation [19]. OpenAD is capable of computing adjoints, while employing optimized algorithms following from combinatorial optimization and graph theory. OpenAD is even capable of using optimal checkpointing schemes at the subroutine level. Of course, use of this package

will necessitate changes to TSOpt's structure. How to introduce this software package to TSOpt's framework should first be carefully investigated.

Next, I want to discuss how `StateHistory`'s `setTime()` and `getState()` functions could be used to incorporate adaptivity with the adjoint-state method. In its present form, `setTime()` (along with `getState()`) could be used to access any past, present, or future state as long as these states align themselves with the grid (which, in this thesis, was dictated by a fixed time-step). Of course, this grid will no longer be uniform for the case where there is adaptivity. Further, this implies that the forward, derivative and adjoint grid might no longer be the same. This discretion between the grids is a major problem because the adjoint state method relies on the (forward) states. As the backward traversal of the adjoint-state method progresses, it might try to access the state at a time level that does not exist in the forward grid.

The `StateHistory` class establishes the software framework needed to allow construction of states between two existing grid levels. This could be accomplished by creating a derived `StateHistory` class, called `AdaptiveStateHistory`, with an interpolating `setTime()` method. The function `setTime()` can then use an interpolation scheme (applied to the saved states) to internally generate an approximation of the state at any given time inside the simulation time-window. This approach, however, inherently assumes that an adequate amount of saved states exist for interpolation (depending on the desired order of accuracy). For problems where only a subset of the forward states were saved, extra work needs to be done in order to meet this

assumption.

Ideally, the `AdaptiveStateHistory` class should incorporate Griewank's optimal checkpointing scheme. The main problem, however, is that one cannot compute the total number of time-steps *a priori* when considering adaptive time stepping; Griewank's algorithm requires knowledge of the total number of time steps in order to determine optimal checkpoint placement. Hinze et al. recognized this problem and created the program *A-Revolve* to solve it [8]. A-Revolve adaptively assigns checkpoints as it performs the forward sweep – unlike Revolve, which first determines the checkpoint placement before it performs the forward sweep. The trade-off for A-Revolve's adaptive checkpoint placement, however, is that some forward steps will inevitably be repeated unnecessarily. Despite this fact, Hinze et al. still show that A-Revolve performs less redundant forward simulations than the "brute force" approach of first forward traversing to determine the number of time steps, then applying Griewank's algorithm to perform the backward traversal (which implicitly requires another forward traversal). Thus, the implementation of `AdaptiveStateHistory` will require proper integration of A-Revolve, interpolation schemes, and adaptive time-stepping algorithms (such as RK45).

In conclusion, by adding the `StateHistory` class hierarchy to TSOpt, I have established a general framework that can support a variety of solutions to the (forward) state-access problem that can occur during the adjoint evolution. I have also implemented a non-trivial derived `StateHistory` class, `GriewankStateHistory`, which

uses Griewank's optimal checkpointing scheme to enable execution of the adjoint-state method without having to save all the forward simulation states. By exploiting `StateHistory`'s functions and framework, it will be possible to address more difficult computational issues arising from the adjoint-state evolution, such as the use of adaptive time-stepping.

Appendix A

The StateHistory Base Class

```
/** Abstract state history class - mixin interface. */
template<class Scalar>
class StateHistory{
typedef typename ScalarFieldTraits<Scalar>::AbsType NormType;

private:
  NormType tol;

public:
  /** sets current time to t */
  virtual void setTime(Scalar t) = 0;

  /** query for time */
  virtual Scalar getTime() const = 0;

  /** query for tolerance */
  NormType getTol() const { return tol; }

  /** returns non-const reference to current state */
  virtual LocalDataContainer<Scalar> & getState() = 0;

  /** returns const reference to current state */
  virtual LocalDataContainer<Scalar> const & getState() const = 0;
};
```

Appendix B

The SmartStateHistory and SmartStateHistoryFactory Classes

```
/* Simplest state history implementation - and most
   memory-intensive. Stores everything. Holds an ordered list of
   states (and respective times) and permits user to access or
   compute a state depending on the given time t.  There are four
   situations possible:

   1. If t matches up to a given tolerance one of the times in
   timelist, the corresponding state is returned.

   2. If t is larger then the latest time in timelist, a new state
   is computed by forward simulator. Forward simulator uses the
   latest time in timelist and corresponding state as initial data.

   3. If t is neither in the range of timelist, nor is larger than
   the latest time, an exception is thrown.

   4. If t is in [t_0, t_f], but is not divisible by dt (i.e., t has
   fallen or will fall in the middle of two adjacent time levels),
   then an exception is thrown
*/

template<class Scalar>
class SmartStateHistory: public StateHistory<Scalar>{
  typedef typename ScalarFieldTraits<Scalar>::AbsType NormType;
private:
```

```
  Dynamics<Scalar> & step;
  AdjSampler<Scalar> & csamp;
  Clock<Scalar> * myclk;
  const ModelBuilder<Scalar> & mb;
  vector<LocalDataContainer<Scalar> *> ldclist;
  vector<Scalar> timelist;
  LocalDataContainer<Scalar> * ctrl;
  int it;

public:

  StateHistory<Scalar>* clone() {
    return new  SmartStateHistory (*this);
  }

  SmartStateHistory(const SmartStateHistory<Scalar> & csh)
    : StateHistory<Scalar>(csh), ldclist(0), timelist(0),
  step(csh.step),
  csamp(csh.csamp),
  myclk(csh.myclk->clone()),
  mb(csh.mb),
  ctrl(NULL), it(csh.it) {

  for (int i = 0; i<csh.timelist.size();i++ ){
   timelist.push_back(csh.timelist[i]);
   LocalDataContainer<Scalar> * tmp = mb.buildState();
   RVLCopy<Scalar> cp;
   cp(*tmp,*csh.ldclist.at(i));
   ldclist.push_back(tmp);
   }

  ctrl = mb.buildControl();

  }

  SmartStateHistory(Dynamics<Scalar> & _step,
            AdjSampler<Scalar> & _csamp,
            Clock<Scalar> const & clk,
            const ModelBuilder<Scalar> & _mb)
    : StateHistory<Scalar>(), ldclist(0), timelist(0),
  step(_step), csamp(_csamp), myclk(clk.clone()), mb(_mb),
```

```
ctrl(NULL), it(0) { }

~SmartStateHistory(){
  for (int i=0; i<ldclist.size(); i++) delete ldclist.at(i);
  if (ctrl) delete ctrl;
  if (myclk) delete myclk;
}

int getSize()  const { return ldclist.at(it)->getSize(); }
LocalDataContainer<Scalar> &getState()  {return * ldclist.at(it);}
LocalDataContainer<Scalar> const &getState() const {
 return *ldclist.at(it);
}

void setTime(Scalar t) {
  int nt=timelist.size();

  // if this is first time, initialize everything
  if (!nt) {
 myclk->resetStart();
 timelist.push_back(myclk->getTime());
 LocalDataContainer<Scalar> * tmp = mb.buildState();
 ctrl = mb.buildControl();
 csamp.setTime (myclk->getTime());
 csamp(*ctrl);
 step.initialize(*tmp, *ctrl);
 ldclist.push_back(tmp);
 nt++;
  }

  Scalar time=timelist[nt-1];
   for (int i=0;i<nt;i++) {
   if (fabs(t-timelist[i])<myclk->getTol()){
   it=i;
   return;
   }
  }

  // Error Checking Phase:
  // 1. Check to see if t is in [t_0, t_f],
  // 2. Check to see if mod(t, delta_t) != 0. If so, this implies
  //    the t trying to be accessed will not align with the grid.
```

```
  if (t<time) {

RVLException e;

if ((fabs(fmod(t+numeric_limits<Scalar>::epsilon()
           -myclk->getStartTime(),myclk->getTimeStep()))
           > myclk->getTol())
    &&(fabs(fmod(t-numeric_limits<Scalar>::epsilon()
         -myclk->getStartTime(),myclk->getTimeStep()))
         > myclk->getTol())
    &&(fabs(fmod(t-myclk->getStartTime(),
      myclk->getTimeStep()))
      > myclk->getTol())) {

  e<<"Error: SmartStateHistory::setTime\n";
  e<<"  time requested = "<<t<<
    " is not an integer multiple of\n";
  e<<"  dt = "<<myclk->getTimeStep()<<".\n";
}

else if (t>myclk->getEndTime()+myclk->getTol()) {
  e<<"Error: SmartSH::setTime\n";
  e<<"requested time = "<<t<<" larger than
   end time of simulation "<<"\n";
  e<<" = "<<myclk->getEndTime()<<" plus
   tolerance "<<myclk->getTol()<<"\n";
}

else if (t>time+myclk->getTol()) {
  e<<"Error: SmartSH::setTime\n";
  e<<"requested time = "<<t<<" larger than
   final time in list "<<"\n";
  e<<" = "<<time<<" plus tolerance "<<myclk->getTol()<<"\n";
}

else if (t<myclk->getStartTime()-myclk->getTol()) {
  e<<"Error: SmartSH::setTime\n";
  e<<"requested time = "<<t<<" less than
   start time of simulation "<<"\n";
  e<<" = "<<myclk->getStartTime()<<" minus
   tolerance "<<myclk->getTol()<<"\n";
```

```
  }

  else {
    e<<"Error: SmartSH::setTime\n";
    e<<"requested time = "<<t<<" does not
     appear in list, within "<<"\n";
    e<<"tolerance "<<myclk->getTol()<<"\n";
  }

  throw e;
  }

  myclk->setTime(time);
  RVLCopy<Scalar> cp;
  it=nt-1;

  while (t - myclk->getTime() > myclk->getTol()) {
   LocalDataContainer<Scalar> * tmp = mb.buildState();
   mb.getCopyFunction()(*tmp,*ldclist.at(it));
   csamp.setTime(myclk->getTime());
   csamp(*ctrl);
   step.fwdStep(*tmp,*ctrl,*myclk);
   myclk->fwdStep();
   it++;
   ldclist.push_back(tmp);
   timelist.push_back(myclk->getTime());
  }

 }

  Scalar getTime() const { return timelist[it]; }

  ostream & write(ostream & str) const {
    str<<"State History object"<<endl;
    str<<"length = "<<this->getSize()<<endl;
    return str;
  }
};



template <class Scalar>
```

```
class SmartHistoryFactory: public StateHistoryFactory<Scalar>{
public:
  SmartHistoryFactory(Dynamics<Scalar> &_dyn,
          Clock<Scalar> & _clk,
          const ModelBuilder<Scalar> &_mb)
    :StateHistoryFactory<Scalar>(_dyn, _clk, _mb){}
  SmartHistoryFactory(const StateHistory<Scalar> &smartfac)
    :StateHistoryFactory<Scalar>(smartfac){}
  ~SmartHistoryFactory(){}

  StateHistory<Scalar>  *buildStateHistory
   (AdjSampler<Scalar> &csamp) const{
     try{ return new SmartStateHistory<Scalar>(this->getDynamics(),
                      csamp, this->getClock(),
                      this->getModelBuilder());
    }

    catch (RVLException &e){
     e<<"\ncalled from SmartHistoryFactory
     ::buildStateHistory\n";
     throw e;
    }
  }

  ostream& write(ostream & str) const {
    str<<"SmartHistoryFactory\n";
    return str;
  }
};
```

Appendix C

The GriewankStateHistory and GriewankStateHistoryFactory Classes

```
#ifndef __RVL_GR_STATEHISTORY
#define __RVL_GR_STATEHISTORY

#include "statehistory.hh"
#include "statehistoryfactory.hh"
#include "dynamics.hh"
#include "sample.hh"
#include "revolve.hh"
#include <cmath>

namespace TSOpt{
  using namespace RVL;

  /** GriewankStateHistory is a derived State History class that
      uses Griewank's checkpointing scheme. This class will allow
      access to the entire state history, while only storing a small
      fraction of forward simulation states. */

  template<class Scalar>
  class GriewankStateHistory:
    public StateHistory<Scalar>, public LocalDataContainer<Scalar> {
    typedef typename ScalarFieldTraits<Scalar>::AbsType NormType;

  private:
    Dynamics<Scalar> & step;
```

```
AdjSampler<Scalar> & csamp;
Clock<Scalar> * myclk;
const ModelBuilder<Scalar> & mb;
vector<LocalDataContainer<Scalar> *> ldclist;
vector<Scalar> timelist;
vector<int> checklist;

// workspace for simulation
LocalDataContainer<Scalar> * ctrl;

// holds latest state generated by time-stepping
LocalDataContainer<Scalar> * curr_state;


// holds latest time generated by time-stepping
Scalar curr_time;

// current iteration
int it;

// tolerance data member
NormType tol;

// keeps track if initial checkpoint
//    schedule was already calculated
bool gotsched;

// keeps track of the number of checkpoints made
// (*NOTE*: This count starts from 0, not 1, due to indexing.
//   We always have at least one checkpointed state, since the
//   first state is always checkpointed.)
int check_count;

// last checkpointed time
Scalar check_time;

// Revolve object
Revolve *r;

// Revolve-related ints
int steps;
int snaps;
```

```
  int info;
  int capo;
  int check;
  int fine;

  // flags for program execution
  bool sweepflag;

  vector<int> checklevels;

public:

  GriewankStateHistory<Scalar>* clone() {
   return new GriewankStateHistory (*this);
  }

  GriewankStateHistory(const GriewankStateHistory<Scalar> & csh)
    : ldclist(0), timelist(0),
  step(csh.step), csamp(csh.csamp), myclk(csh.myclk->clone()),
  mb(csh.mb), ctrl(NULL), it(csh.it), tol(csh.tol),
  gotsched(false), check_count(0), snaps( csh.snaps), r(NULL)
  {

    for (int i = 0; i<csh.timelist.size();i++) {
     timelist.push_back(csh.timelist.at(i));
     LocalDataContainer<Scalar> * tmp = mb.buildState();
     RVLCopy<Scalar> cp;
     mb.getCopyFunction ()(*tmp,*csh.ldclist.at(i));
     ldclist.push_back(tmp);
    }

    ctrl = mb.buildControl();
    curr_state = mb.buildState();

    mb.getCopyFunction()(*curr_state, csh.getState());
    curr_time = csh.getTime();

  }

  GriewankStateHistory(Dynamics<Scalar> & _step,
           AdjSampler<Scalar> & _csamp,
           Clock<Scalar> const & clk,
```

```
          const ModelBuilder<Scalar> & _mb,
          unsigned int nsnaps)

  : ldclist(0), timelist(0),
step(_step), csamp(_csamp), myclk(clk.clone()), mb(_mb),
ctrl(NULL), it(0), tol( clk.getTol()), gotsched(false),
check_count(0), snaps(nsnaps), r(NULL) {
 timelist.clear();
}

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

  if (ctrl) delete ctrl;
  if (myclk) delete myclk;
  if (curr_state) delete curr_state;
  if (r) delete r;
}

int getSize()  const {
 return ldclist.at(check_count)->getSize();
}

LocalDataContainer<Scalar> &getState() {return *curr_state;}
LocalDataContainer<Scalar> const &getState() const {
 return *curr_state;
}

Scalar getTime() const {return curr_time;}

// getData() functions mostly for outputting ldclist
Scalar * getData()  {return ldclist.at(check_count)->getData();}
Scalar const * getData() const {
 return ldclist.at(check_count)->getData();
}

Scalar * getData(int i) {return ldclist.at(i)->getData();}

void setTime(Scalar t) {
```

```
 int nt=ldclist.size();
 RVLCopy<Scalar> cp;

 // if this is first time, initialize everything
 if (!nt) {

myclk->resetStart();
timelist.push_back(myclk->getTime());
LocalDataContainer<Scalar> * tmp = mb.buildState ();
ctrl = mb.buildControl();
curr_state = mb.buildState();
curr_time = timelist.at(0);

csamp.setTime(myclk->getTime());
csamp(*ctrl);

step.initialize(*tmp, *ctrl);
step.initialize(*curr_state, *ctrl);
ldclist.push_back(tmp);
sweepflag = true;
nt++;

 }

 check_time=timelist.at(nt-1);

 if (sweepflag){

// Forward Phase Error Checking:
// 1. Check to see if t is in [t_0, t_f].
// 2. Check to see if mod(t, delta_t) != 0. If so, this implies
//    the t trying to be accessed will not align with the grid.

if (t < myclk->getTime() - myclk->getTol()) {
  RVLException e;
  e<<"Error: GriewankStateHistory::setTime\n";
  e<<"  time requested = "<<t<<" cannot be less than\n";
  e<<"  current time = " << myclk->getTime() <<
   " minus tolerance\n";
  e <<"  " <<  myclk->getTol() << " during forward phase\n";
   throw e;
}
```

```
 if (t > myclk->getEndTime() + myclk->getTol()) {

   RVLException e;
   e<<"Error: GriewankStateHistory::setTime\n";
   e<<"  time requested = "<<t<<" is greater than\n";
   e<<"  final simulation time = "<<myclk->getEndTime()<<"\n";
   e<<"  plus tolerance " << myclk->getTol() << " \n";
   throw e;
 }

 if ((fabs(fmod(t+numeric_limits<Scalar>::epsilon()
           -myclk->getStartTime(),myclk->getTimeStep()))
           >myclk->getTol())
    &&(fabs(fmod(t-numeric_limits<Scalar>::epsilon()
         -myclk->getStartTime(),myclk->getTimeStep()))
         >myclk->getTol())
    &&(fabs(fmod(t-myclk->getStartTime(),
        myclk->getTimeStep()))
        >myclk->getTol()))
  {
    RVLException e;
    e<<"Error: GriewankStateHistory::setTime\n";
    e<<"  time requested = "<<t<<
      " is not an integer multiple of\n";
    e<<"  dt = "<<myclk->getTimeStep()<<".\n";
  }

// Run Revolve to get the initial checkpoint list
 if (!gotsched){
   enum ACTION::action whatodo;
   capo = 0;
   info = 3;
   // Note: snaps set upon construction now

   steps = myclk->getNumberOfSteps();
   fine = steps + capo;
   check = -1;
   check_count = 0;

  // Declare Revolve Object
  r = new Revolve(steps, snaps, info);
```

```
  // Clear checkpointed times
  timelist.clear();
  checklist.clear();

  do {
      whatodo = r->revolve(&check, &capo, &fine, snaps, &info);

      // only care if I'm saving (time) checkpoints
      if ((whatodo == ACTION::takeshot) && (info > 1)){
    checklist.push_back(capo);
     check_count ++;
     }


    if (whatodo == ACTION::error) {
     RVLException e;
     e << "Error: GriewankStateHistory::setTime()\n";
     e << " irregular termination of revolve \n";
     throw e;
    }


  } while ((whatodo != ACTION::terminate) &&
   (whatodo != ACTION::error) &&
   (check_count < snaps));


  gotsched = true;
  myclk->setTime(myclk->getStartTime());
  myclk->setTimeLevel(0);
  check_count = 0;

  timelist.push_back(myclk->getTime());

} // matches 'if (!gotsched)'


myclk->setTime(check_time);

it=nt-1;
```

```
check_count = it;

// should user try to run fwdstep after bwd steps...
if (!sweepflag) {
  ldclist.clear();
  timelist.clear();
  sweepflag = true;
}

LocalDataContainer<Scalar> * tmp = mb.buildState();

// set *tmp to last current state to avoid resimulation
mb.getCopyFunction()(*tmp,*curr_state);

// ... and also set the last current time that matches the
// last current state
myclk->setTime(curr_time);


// time-stepping loop
while (myclk->getTime()<t){

  csamp.setTime(myclk->getTime());
  csamp(*ctrl);

  step.fwdStep(*tmp,*ctrl,*myclk);
  myclk->fwdStep();


  it++;


  // need to keep curr_state updated for bwd traversal
  mb.getCopyFunction()(*curr_state, *tmp);

  // update curr_time
  curr_time = myclk->getTime();

  if ( timeToCheckToo(myclk->getTimeLevel())) {


    LocalDataContainer<Scalar> * tmp2 = mb.buildState();
```

```
      mb.getCopyFunction()(*tmp2, *tmp);

      ldclist.push_back(tmp2);
      timelist.push_back(curr_time);

      check_count++;
    }

  }


  // can't do backward traversal until t = t_final - dt
  if ( fabs(myclk->getTime() -
    (myclk->getEndTime()-myclk->getTimeStep())) < tol ){
    sweepflag = false;

  }


}

else{

  // Backward Phase Error Checking:
  // 1. Check to see if t is in [t_0, t_f]
  // 2. Check to see if mod(t, delta_t) != 0. If so, this implies
  //    the t trying to be accessed will not align with the grid.

  if (t > myclk->getTime()+myclk->getTol()) {
    RVLException e;
    e<<"Error: GriewankStateHistory::setTime\n";
    e<<"  time requested = "<<t<<" cannot be greater than\n";
    e<<"  current time = " << myclk->getTime() <<
      " plus tolerance\n";
    e<< myclk->getTol() << " during backward phase\n";
    throw e;
  }


  if (t < myclk->getStartTime()-myclk->getTol()) {

    RVLException e;
```

```
  e<<"Error: GriewankStateHistory::setTime\n";
  e<<"  time requested = "<<t<<" is less than\n";
  e<<"  final simulation time = "<<myclk->getStartTime()<<"\n";
  e<<"  minus tolerance = " << myclk->getTol() << ".\n";
  throw e;
}

if ((fabs(fmod(t+numeric_limits<Scalar>::epsilon()
           -myclk->getStartTime(),myclk->getTimeStep()))
           >myclk->getTol())
    &&(fabs(fmod(t-numeric_limits<Scalar>::epsilon()
         -myclk->getStartTime(),myclk->getTimeStep()))
         >myclk->getTol())
    &&(fabs(fmod(t-myclk->getStartTime(),
       myclk->getTimeStep()))
       >myclk->getTol()))
  {
    RVLException e;
    e<<"Error: GriewankStateHistory::setTime\n";
    e<<"  time requested = "<<t<<
      " is not an integer multiple of\n";
    e<<"  dt = "<<myclk->getTimeStep()<<".\n";
  }

if (fabs(myclk->getStartTime() - t) > tol) {

  while( (myclk->getTime() - t >= tol) ) {

    bwdStep();

  }

}

// no need to bwdStep since initial time is *always* checkpointed
else {
  mb.getCopyFunction()(*curr_state, *ldclist.at(0));
  myclk->setTime( timelist.at(0));

  curr_time = timelist.at(0);
}
```

```
// Reset values again for forward mode
if (fabs(myclk->getTime() - myclk->getStartTime())<tol) {
  sweepflag = true;
  nt = 0;
  ldclist.clear();
  timelist.clear();
  checklist.clear();
  gotsched = false;
  timelist.push_back(myclk->getTime());
}

  }
}



bool timeToCheck(Scalar t){
  for (int k=0; k< timelist.size(); k++){
if (fabs(timelist[k] - t ) < tol)
  return true;
  }
  return false;
}

bool timeToCheckToo(Scalar ii) {
  for (int k=0; k< checklist.size(); k++){
if (fabs(checklist[k] - static_cast<int>(ii) ) < tol)
  return true;
  }
  return false;

}


ostream & write(ostream & str) const {
  str<<"Griewank State History object"<<endl;
  str<<"length = "<<this->getSize()<<endl;
  return str;
}
```

```
bool bwdStep() {
  enum ACTION::action whatodo;
  int oldcapo;
  RVLCopy<Scalar> cp;

if (myclk->getNumberOfSteps() > 2) {

if (!sweepflag){

  do {

    oldcapo = capo;
    whatodo = r->revolve(&check, &capo, &fine, snaps, &info);


    // covers cases:
    //     takeshot -- push()
    //     restore  -- pop()
    //     advance  -- setTime() to do fwd steps
    //     terminate -- return false
    //     error -- throw error
    if ((whatodo == ACTION::takeshot) && (info > 1)) {

      LocalDataContainer<Scalar> * tmp = mb.buildState();
      mb.getCopyFunction()(*tmp, *curr_state);

      while (check <= check_count){
     ldclist.pop_back();
     timelist.pop_back();
     check_count --;
      }

      ldclist.push_back(tmp);
      timelist.push_back(myclk->getTime());
      check_count ++;

    }

    if ((whatodo == ACTION::advance) && (info > 2)) {

      Scalar limit = timelist.at(check) +
           fabs((capo-oldcapo)*myclk->getTimeStep());
```

```
        while(myclk->getTime() < limit - myclk->getTol()) {
       csamp.setTime(myclk->getTime());
       csamp(*ctrl);

       step.fwdStep(*curr_state,*ctrl,*myclk);
        myclk->fwdStep();

         }
      }

      if ((whatodo == ACTION::restore) && (info > 2)) {
        mb.getCopyFunction()(*curr_state, *ldclist.at(check));
        myclk->setTime(timelist.at (check));
      }

      if (whatodo == ACTION::error) {

        RVLException e;
        e << "Error GriewankStateHistory::bwdStep()\n";
        e << "Irregular execution of Revolve\n";

        throw e;
      }

      curr_time = myclk->getTime();


    }  while((whatodo != ACTION::youturn) &&
       (whatodo != ACTION::terminate));

    return true;
}

else {

    RVLException e;
    e << "Error: GriewankStateHistory::bwdStep()\n";
    e << "Cannot call bwdStep until forward simulation has\n";
    e << "been done\n";
    throw e;
```

```
    }
  }

    else {
     capo=0; check=0; info=0; fine=0; snaps=1;
     oldcapo=capo;
     whatodo = static_cast<ACTION::action>(0);
     myclk->resetEnd();

    mb.getCopyFunction()(*curr_state, *ldclist.at(check));
     myclk->setTime(timelist.at(check));
     return false;

    }

  } // end of bwdStep()

};


  template <class Scalar>
  class GriewankStateHistoryFactory:
    public StateHistoryFactory<Scalar> {
   private:
     unsigned int snaps;
   public:
     GriewankStateHistoryFactory(unsigned int nsnaps)
       :StateHistoryFactory<Scalar>(), snaps(nsnaps){}

     GriewankStateHistoryFactory
      (GriewankStateHistoryFactory<Scalar> const & gshfac)
        : StateHistoryFactory<Scalar>(gshfac), snaps( gshfac.snaps) {}

     ~GriewankStateHistoryFactory() {}

      StateHistory<Scalar> * buildStateHistory(Dynamics<Scalar> & dyn,
                           AdjSampler<Scalar> & csamp ,
                           Clock<Scalar> & clk,
                           ModelBuilder<Scalar> const & mb) const {
        try{
      return new GriewankStateHistory<Scalar>(dyn,csamp,clk,mb,snaps);
        }
```

```
      catch (RVLException &e){
     e<<"\ncalled from GriewankStateHistoryFactory
     ::buildStateHistory\n";
     throw e;
      }
    }

    ostream& write(ostream & str) const {
      str<<"GriewankStateHistoryFactory\n";
      return str;
    }
  };

}   // matches 'namespace TSOpt'

#endif
```

Appendix D

Executable Code for `SmartStateHistory` and `GriewankStateHistory` Classes

This code, called `TestAdj`, performs the "inner product test" using either the `SmartStateHistory` class or the `GriewankStateHistory` class. `TestAdj` tests the following conditional:

$$\frac{|\langle Ax, y\rangle - \langle x, A^T y\rangle|}{\|Ax\|\|y\|} \leq tol\,. \tag{D.1}$$

If the adjoint generated from the `StateHistory` subclasses is correct, the conditional (D.1) should yield `true`. For this test, the tolerance *tol* was set to (100ϵ), where ϵ denotes the machine precision epsilon. This code was used to generate the data presented in the "Numerical Results" section of this thesis.

```
#include "tsop.hh"
#include "stencil.hh"
#include "smartsh.hh"
#include "griewanksh.hh"
#include "testdyn1.hh"
#include "adjtest.hh"
```

```
int main(int argc, char ** argv) {

  using namespace TSOpt;
  using namespace RVL;

  try {
    srand(getpid());

    int nu = 2;
    int nc = 2;

    double tbeg = 0.0;
    double tend = 0.10;
    double dt   = 0.01;


    /** domain */
    RnSpace<double> dom(nc);

    /** range */
    RnSpace<double> rng(nu);

    /** Statics */
    RnStatics<double> stat(nu,nc,tbeg,tend,dt);

    int verbose = 0;

    //Dynamics
    testdyn1 dyn(tbeg, 0.001*dt, verbose);

    //Clock
    ConstClock<double> clock(tbeg,tend, dt);

    StateHistoryFactory<double> * histfac = NULL;

    if (argc>1) {
      int nsnaps=0;
      if (sscanf(argv[1],"%d",&nsnaps)) {
if (nsnaps > 0) {
  GriewankStateHistoryFactory<double> * ghistfac =
    new GriewankStateHistoryFactory<double>(nsnaps);
  histfac = ghistfac;
```

```
}
      }
    }

    if (!histfac) {
      SmartHistoryFactory<double> * shistfac =
       new SmartHistoryFactory<double>;
      histfac = shistfac;
    }

    // Op
    int itmax = 100000;

    TSOp<double> op(dom, rng, stat, dyn, clock,
     *histfac, itmax, verbose);

    Vector<double> c(dom);
    Vector<double> dc(dom);

    RVLAssignConst<double> ac(0.5);
    c.eval(ac);

    RVLRandomize<double> adc(getpid(),-1.0,1.0);

    OperatorEvaluation<double> oe(op,c);
    AdjointTest(oe.getDeriv(),adc,cout);

  }
  catch (RVLException & e) {
    e.write(cerr);
  }

}
```

The following code (which consists of a header file and a C++ file) defines the dynamical system that is being solved in the code above.

```
#ifndef __TEST_DYN1
#define __TEST_DYN1
```

```
#include "../include/dynamics.hh"

namespace TSOpt {

using RVL::LocalDataContainer;

class testdyn1: public DynamicsISEC<double> {
private:
  double tinit;
  double tol;
  bool verbose;
public:
  testdyn1(double _tinit = 0.0,
   double _tol = 100*numeric_limits<double>::epsilon(),
   bool _verbose = false)
    : tinit(_tinit), tol(_tol), verbose(_verbose) {}
  testdyn1(const testdyn1 & td)
    : tinit(td.tinit), tol(td.tol), verbose(td.verbose) {}
  ~testdyn1() {}

  virtual void fwdStep(LocalDataContainer<double> & u,
LocalDataContainer<double> const & c,
Clock<double> const & clk);

  virtual void derStep(LocalDataContainer<double> & u,
LocalDataContainer<double> & du,
LocalDataContainer<double> const & c,
LocalDataContainer<double> const & dc,
   Clock<double> const & clk);

  virtual void  adjStep(LocalDataContainer<double> & u,
LocalDataContainer<double> & du,
LocalDataContainer<double> const & c,
LocalDataContainer<double> & dc,
Clock<double> const & clk);
  virtual void write(RVLException & e) const {
    e<<"Test Dynamics object - system of uncoupled
        logistic equations\n";
  }
  virtual ostream & write(ostream & str) const {
    str<<"Test Dynamics object - system of uncoupled
```

```
     logistic equations\n";
    return str;
  }
};

}

#endif

-----------------------------------------------------------

#include "testdyn1.hh"

namespace TSOpt {

using RVL::LocalDataContainer;

void testdyn1:: fwdStep(LocalDataContainer<double> & u,
 LocalDataContainer<double> const & c,
 Clock<double> const & clk){

  verbose = 'true';

  double a = clk.getTimeStep();

int n = u.getSize();

if (verbose) {
cerr<<endl<<"////////////////////////////////////"<<endl;
cerr<<"rhs: a = "<<a<<" t = "<<clk.getTime()<<endl;
cerr<<"before:"<<endl;
cerr<<"c  = "<<c.getData()[0]<<endl;

for (int i=0; i<n; ++i)
  cerr<<"u[" << i << "] = "<<u.getData()[i]<<endl;
}


double * uc = u.getData();

for (int i=0;i<n;i++) {
```

```
  uc[i] = uc[i] + a*(1.0 - uc[i]*uc[i]);

}

cout << endl;

if (verbose) {
cerr<<"after:"<<endl;
cerr<<"c  = "<<c.getData()[0]<<endl;

for (int i=0; i<n; ++i)
  cerr<<"u[" << i << "] = "<<u.getData()[i]<<endl;

cerr<<"////////////////////////////////////"<<endl;
}

}

void testdyn1::derStep(LocalDataContainer<double> & u,
 LocalDataContainer<double> & du,
 LocalDataContainer<double> const & c,
 LocalDataContainer<double> const & dc,
 Clock<double> const & clk) {
double a = clk.getTimeStep();

if (verbose) {
cerr<<endl<<"////////////////////////////////////"<<endl;
cerr<<"drhs: a = "<<a<<" t = "<<clk.getTime()<<endl;
cerr<<"before:"<<endl;
cerr<<"c  = "<<c.getData()[0]<<endl;
cerr<<"u  = "<<u.getData()[0]<<endl;
cerr<<"dc = "<<dc.getData()[0]<<endl;
cerr<<"du = "<<du.getData()[0]<<endl;
}
int n = u.getSize();
double * uc = u.getData();
double * duc = du.getData();
for (int i=0;i<n;i++) {
duc[i] = duc[i]*(1.0 - 2.0*a*uc[i]);
// uc[i] = uc[i] + a*(1.0 - uc[i]*uc[i]);
}
if (verbose) {
```

```
cerr<<"after:"<<endl;
cerr<<"c  = "<<c.getData()[0]<<endl;
cerr<<"u  = "<<u.getData()[0]<<endl;
cerr<<"dc = "<<dc.getData()[0]<<endl;
cerr<<"du = "<<du.getData()[0]<<endl;
cerr<<"////////////////////////////////////"<<endl;
}

}
void testdyn1::adjStep(LocalDataContainer<double> & u,
 LocalDataContainer<double> & du,
 LocalDataContainer<double> const & c,
 LocalDataContainer<double> & dc,
 Clock<double> const & clk) {

  cout << "adjstep: " << endl;

        double a = clk.getTimeStep();
if (verbose) {
cerr<<endl<<"////////////////////////////////////"<<endl;
cerr<<"arhs: a = "<<a<<" t = "<<clk.getTime()<<endl;
cerr<<"before:"<<endl;
cerr<<"c  = "<<c.getData()[0]<<endl;
cerr<<"u  = "<<u.getData()[0]<<endl;
cerr<<"dc = "<<dc.getData()[0]<<endl;
cerr<<"du = "<<du.getData()[0]<<endl;
}
int n = u.getSize();
for (int i=0;i<n;i++) {
du.getData()[i] = du.getData()[i]*(1.0 - 2.0*a*u.getData()[i]);
}
if (verbose) {
cerr<<"after:"<<endl;
cerr<<"c  = "<<c.getData()[0]<<endl;
cerr<<"u  = "<<u.getData()[0]<<endl;
cerr<<"dc = "<<dc.getData()[0]<<endl;
cerr<<"du = "<<du.getData()[0]<<endl;
cerr<<"////////////////////////////////////"<<endl;
}
}
}
```

Bibliography

[1] Brouwer and Jansen. Dynamic optimization of waterflooding with smart wells using optimal control theory. *SPE*, 2004.

[2] Coffey. Rythmos: Transient integration of differential equations. `http://software.sandia.gov/trilinos/packages/docs/dev...` `/packages/rythmos/doc/html/index.html`. Date Accessed: April 2, 2007.

[3] Gamma, Helm, Johnson, and Vlissides. *Design Patters: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1998.

[4] Gockenbach, Symes, Reynolds, and Shen. Efficient and automatic implementation of the adjoint state method. *ACM TOMS*, 28(1):22–24, March 2002.

[5] Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[6] Griewank and Walther. Revolve: An implementation of checkpointing of the reverse or adjoint mode of computational differentiation. *ACM TOMS*, 26:19–45, 2000.

[7] Hager. Runge-Kutta methods in optimal control and the transformed adjoint system. *Numerishe Mathematik*, 87:247–282, 1999.

[8] Hinze and 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.

[9] Jahn. *Introduction to the Theory of Nonlinear Optimization.* Springer-Verlag, 2nd edition edition, 1996.

[10] Kirchgraber. Multi-step methods are essentially one-step methods. *Numerische Mathematik*, 48:85–90, 1986.

[11] Kolda, Lewis, and Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review*, 45:385–482, 2003.

[12] Lions. *Optimal Control of Systems Governed by Partial Differential Equations.* Spring-Verlag, 1971.

[13] Nocedal and Wright. *Numerical Optimization.* Springer, 1999.

[14] Padula, Scott, and Symes. A software framework for abstract expression of coordinate-free linear algebra and optimization algorithms. Technical report, Rice University, 2005.

[15] Renders and Flasse. Hybrid methods using genetic algorithms for global optimization. *IEEE Transactions on Systems*, 26:243–258, 1996.

[16] Sarma and Aziz. Implementation of adjoint solution for optimal control of smart wells. *SPE*, 2005.

[17] Symes. Reverse time migration with optimal checkpointing. Technical report, Rice University, 2006.

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

[19] Utke, Naumann, Fagan, Tallent, Strout, Heimbach, Hill, and Wunsch. OpenAD/F: A modular, open-source tool for automatic differentiation of fortran codes. *ACM TOMS*, 2006.