Time Stepping Classes for Optimization

Marco Enriquez and William W. Symes

ABSTRACT

This report introduces the "Time Stepping Package for Optimization", or `TSOpt`, which is an interface for time-stepping simulation written in C++. It packages a simulator together with its derivatives ("sensitivities") and adjoint derivatives with respect to simulation parameters in a single object called a `Jet`, which can be used in conjunction with an optimization algorithm to solve a simulation-driven optimization problem. Further, `TSOpt` interfaces with the Rice Vector Library (`RVL`), allowing `Jet` objects to define a `Operator` subclass.

INTRODUCTION

We are interested in solving optimization problems of the form

$$\min_{c} \qquad J(c) = G(u(c, \cdot)) \tag{1}$$

where the state $u(t)$ and the control $c(t)$ solve the state equation

$$\bar{H}\left(\frac{du}{dt}, u, c\right) = 0\,. \tag{2}$$

Though many types of control and inverse problems fit the mathematical framework above, we note that the procedure of solving such problems numerically oft involve the same procedures (e.g., formation of derivatives). To exploit these commonalities, hence, we created `TSOpt`, or the "Time-Stepping for Optimization" package. `TSOpt` is a "middleware" package written in C++, designed to act as an "interface for time-stepping simulation", providing a way for simulation software to inter-operate with optimization software. `TSOpt` is capable of encapsulating the reference, linearized and adjoint simulators in a single object, and properly arrange their execution. `TSOpt` also aids in providing necessary data structures for the optimization algorithm (e.g., the gradient, formed via the adjoint-state method).

This report is organized as follows: the first section introduces `RVL` and section two discusses the `Alg` framework developed by Tony Padula. `RVL` and the `Alg` framework provides the foundation for `TSOpt`. The most notable features of `TSOpt` include its modular code structure, due to use of the `Alg` framework from the Rice Vector Library (`RVL`), and also accommodation of a generic data structure type through templating. The specifics of the structure of `TSOpt` and its features will be discussed in more detail in section three.

The final section of this report presents the "Optimal Well-Rate Allocation" (OWRA) problem, which is a reservoir engineering inverse poblem. We conclude this report by presenting numerical results for OWRA, obtained via `TSOpt` in conjunction with a reservoir simulation package and an external optimization package.

THE RICE VECTOR LIBRARY (RVL)

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

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

RVL AND THE ALG FRAMEWORK

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

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

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

schemes are algorithms, `TSOpt`'s components can be implemented from `Algorithm` objects. In fact, three subclasses of `Algorithm` serve as the foundation of `TSOpt`. These subclasses are called the `StateAlg`, the `LoopAlg` and the `ListAlg` classes. The UML diagram in figure 1 show these subclasses, along with their methods. Since it is crucial

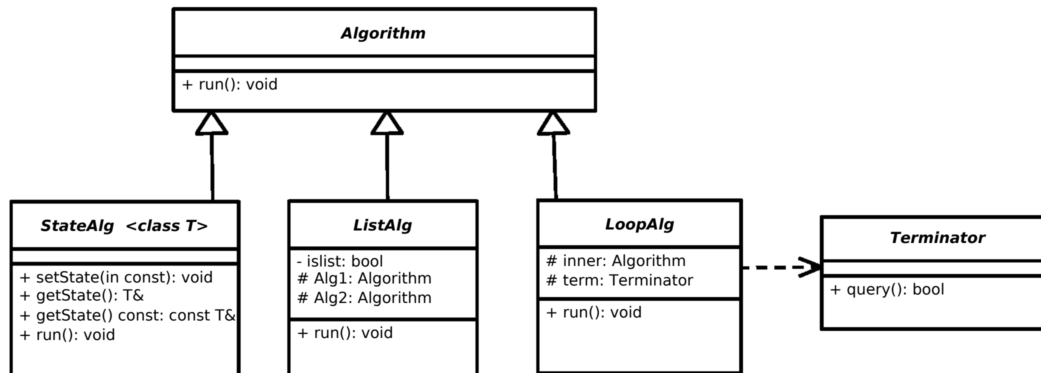

Figure 1: The `Alg` class and its subclasses

that we understand their functionality, they are discussed in detail below.

The `StateAlg` Class

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

```
template<class T>
class StateAlg: public Algorithm {
public:
  virtual void setState(const T & x) = 0;
  virtual const T & getState() const = 0;
  virtual T & getState() = 0;
};
```

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

The `LoopAlg` and `terminator` Classes

The `Alg` Framework also has a class capable of abstracting looping algorithms, such as GMRES. This class, which derives from `Algorithm` is called `LoopAlg`. A `LoopAlg` object's

job is to repeat execution of an `Algorithm` object (through the `run()` method) until some criteria is met. This criteria is encapsulated in something called a `Terminator` object. The `Terminator` base class is implemented the following way:

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

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

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

  return t1;
}
```

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

The `ListAlg` Class

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

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

  return (t1 && t2);
}
```

THE SOFTWARE FRAMEWORK OF TSOPT

After discussing RVL and the Alg framework, we can now discuss TSOpt. TSOpt is a software package that encapsulates reference, linearized and adjoint simulations in a single object. As mentioned in earlier sections, TSOpt uses RVL and the Alg package as the foundation of its framework. This section presents the main components of the TSOpt framework, which consist of the time, state, timestep, sim, terminator and jet classes.

The time Hierarchy

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

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

The State Class

The State class is not, strictly speaking, a part of TSOpt – though a couple of different concrete State classes have been implemented in TSOpt. Users of TSOpt can implement their own State class to act as an interface between their preferred simulator data structure and TSOpt. A State object is composed of two objects: a data structure to hold data (e.g., an array) and a time object, which holds the current simulation time associated with the data. This relationship can be seen in the UML diagram, figure (2). All State classes must implement methods to get and set the time object, and methods to access and alter its internal data structure. There are two examples of State subclasses that have been implemented in TSOpt, to accompany the two different time types: RnState and RealRnState. The RnState class contains a DiscreteTime object, and is used for fixed time step simulations. (The "Rn" refers to the vector space $\mathbb{R}^n$.). The RnState class internally contains an rn struct, defined with the following components:

```
typedef struct {
   /** time index */
```

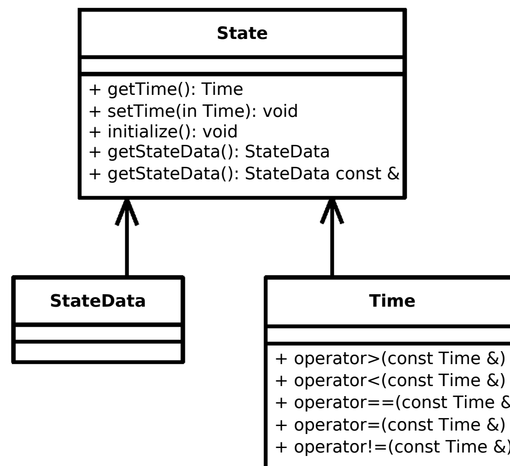

Figure 2: The State class and its components

```
    int it;
    /** state dim */
    int nu;
    /** control dim */
    int nc;
    /** state samples */
    float * u;
    /** control samples */
    float * c;

  } rn;
```

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

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

The `TimeStep` Class

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

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

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

The `Sim` Hierarchy

The `Sim` class, as its name implies, is a simulator class. It orchestrates a `StateAlg` object, a `Terminator` object and a `Time` object in order to perform the simulation. Concrete subclasses of `Sim` also implement different simulation/memory managing schemes for use in either the linearized or adjoint computations. The UML diagram 3 show the subclasses of the `Sim` class. These subclasses, the `StdSim`, `RASim` and `CPSim` classes, will be explained in more detail below.

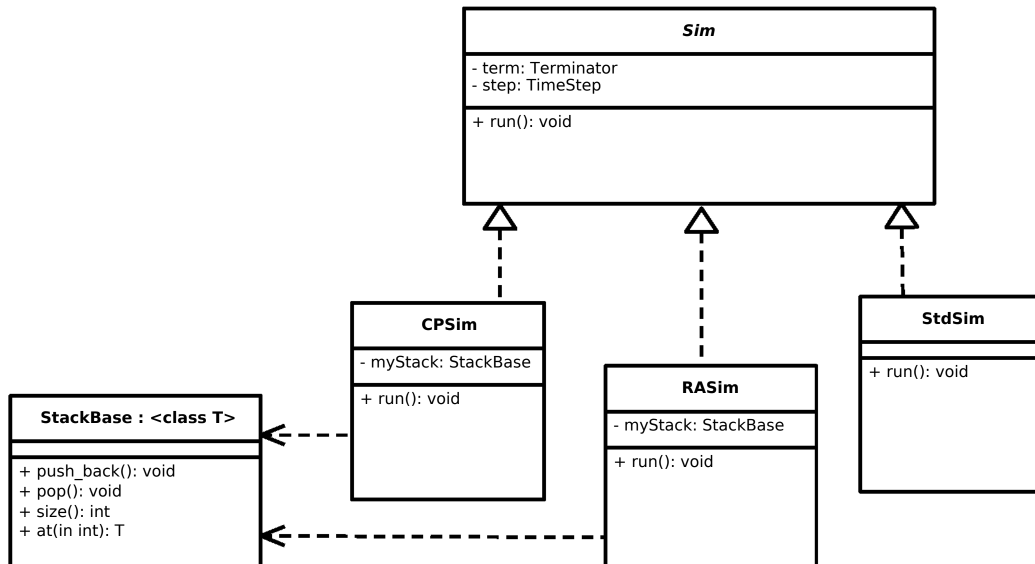

Figure 3: The `Sim` class and its derived classes.

The subclass `StdSim` is a "forgetful" simulator; to provide the appropriate reference state during the adjoint evolution, the `StdSim` will run the reference simulator from the

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

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

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

In order to create a stack in `TSOpt`, users must implement a concrete subclass of the `stackBase` class, which is shown in the UML diagram 4 All `Sim` subclasses whose function-

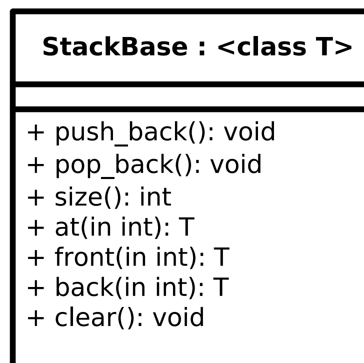

Figure 4: The stackBase class and its methods.

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

Other `Sim` subclasses exist in `TSOpt`; of note is the `CPSim` class, which uses Griewank's optimal checkpointing scheme (Griewank and Walther, 2000). Checkpointing is the "middle ground" between the two aforementioned strategies of a "forgetful" simulator and a "remember-all' simulator. Two types of checkpointing exist in `TSOpt`: offline mode for fixed time step simulations, and online mode for adaptive simulations. I discuss the notion behind checkpointing in more detail below.

Checkpointing

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

To avoid the steep computational and storage costs associated with the "forgetful" and "Remember-All" strategies, Griewank proposed an algorithm called checkpointing (Griewank and Walther, 2000). The idea behind checkpointing is actually an intelligent combination of the two previously mentioned strategies: save a few states in some buffer (called checkpoints), and then forward-simulate from the nearest saved state until the time of interest. As the backward traversal continues, the checkpoints are updated such that none have been passed (and rendered useless) by the traversal. Through this process, checkpointing eliminates the need to store the whole state vector while minimizing the recomputation of states. Given some assumptions of the costs of memory access and recomputation, Griewank also proved the optimality of his checkpointing algorithm in Griewank and Walther (2000); given N_B buffers and N_S states such that $N_B \ll N_S$, his checkpointing scheme only adds logarithmic (i.e., $O(log(N_S))$) recomputation cost.

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

Adaptive Checkpointing

In ARevolve, adaptive checkpointing works like fixed-step checkpointing algorithms, with the exception of not requiring an input of the number of time-steps to be taken. In exchange, however, the user must set an algorithmic flag to denote that the forward evolution is finished, and the simulations are ready for the adjoint simulation. The biggest limitation of Hinze and Sternberg (2005)'s checkpointing algorithm, however, is that it does not cater to taking adaptive simulation in the adjoint field. ARevolve makes the assumption that the time levels in the adjoint and reference field align, implying that the adjoint time grid will be dictated by the reference simulation. This assumption is often incorrect, as the adjoint dynamics may have very little similarities with the reference dynamics (e.g., adaptive quadrature).

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

Similar to the checkpointing algorithm in (Enriquez, 2008), the adaptive checkpointing algorithm consists of a forward mode and a backward mode – ensuring that the full forward evolution runs only once before the adjoint evolution takes place. The key difference here is the incorporation of the interpolation buffer, which itself is a deque that is being managed by a class. (The deque is a good choice for such an algorithm since push and pop operations are supported at both ends of the buffer, for $O(1)$ computational complexity.) Every time we "update" the interpolation buffer, it simply means that one slot in the buffer is replaced with a new interpolation node, such that the interpolation nodes are in order (in time).

The Time Terminator Hierarchy

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

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

The `jet` Hierarchy

The term "jet", in applied mathematics, refers to a collection of a function, its derivative and its adjoint. True to this definition, the `jet` class is meant to hold the reference, linearized and adjoint simulators, and is at the highest level of `TSOpt` hierarchy. The `jet` subclasses require a `Sim` object for the forward evolution, and two triples of `timestep`, `stateAlg` and `timeTerminator` objects for both the linearized and adjoint evolution. This class assumes that the collection of objects pertaining to the forward, linearized and adjoint evolution are related in the appropriate sense. The following figure is a UML diagram showing the relationship between the `jet` class and its components.

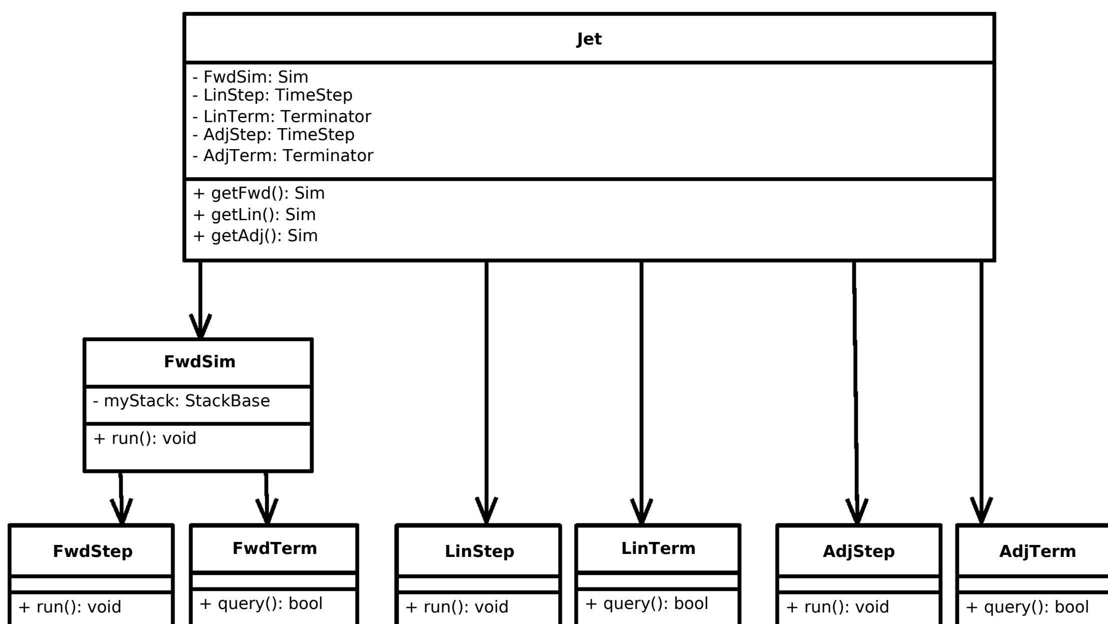

Figure 5: The jet class and its components.

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

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

TSOPT AND UMIN

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

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

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

```
class GradFunctionObject{
private:
   jet j;

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

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

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

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

  }

};
```

The `Functional` object is used to make a `FunctionalEvaluation` object, which in turn, can then be passed to the `UMin` framework to perform the optimization. For a more thorough discussion of this process, and the associated classes, see (Padula et al., 2009).

TSOPT AND EXTERNAL OPTIMIZATION PACKAGES

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

EXAMPLE: OPTIMAL WELL-RATE ALLOCATION

The Optimal Well-Rate Allocation (OWRA) can be posed as the following problem: find the pumping and injecting rates for reservoir wells over a certain time window, as to maximize profit. Solving OWRA via optimal control theory is not a new topic; previous attempts have been made by Brouwer and Jansen (2004) and Sarma and Aziz (2005), for example. In this report, we solve the problem posed by Wiegand et al. (2008), that finds the optimal well rate that will maximize revenue from oil production, while penalizing water injection and production:

$$\min_{q_i \;\; i \in I \cup P} \qquad J(q) = \int_0^T dt \left(\sum_{i \in P} \alpha (1 - s_a) q_i(t) + \sum_{i \in P} \frac{\beta}{2} s_a q_i^2(t) + \sum_{i \in I} \gamma q_i(t) \right) , \quad (3)$$

where q_i are the well rate at the i is an index representation a location in the domain, I is set of indices that correspond to injecting wells, P is a set of indices that correspond to producing wells, α, β and γ are scalar variables. By convention, we assume that the

producing well rates are represented as negative numbers, and injecting well rates are represented by positive nmbers. The aqueous pressure p and aqueous saturation s_a solve the 2D, two-phase, incompressible Black-Oil equations:

$$-\nabla(K(x)\lambda_{tot}(s_w(x,t)\nabla p(x,t)) = \sum_{i\in P}(1-s_a)q_i(t)\delta(x-x_i) \tag{4}$$

$$+\sum_{i\in P\cup I} s_a q_i(t)\delta(x-x_i) \tag{5}$$

$$\phi(x)\frac{\partial}{\partial t}s_a(x,t) - \nabla\cdot(K(x)\lambda_a(s_a(x,t))\nabla p(x,t)) = \sum_{i\in P\cup I} s_a q_i(t)\delta(x-x_i)\,. \tag{6}$$

In the equation above, K represents permeability, λ represents phase mobility and ϕ represents rock porosity. The Black-Oil Equations stem from the *phase continuity* equations, which capture simultaneous, physical fluid flow behavior of up to three immiscible phases (namely: water, oil and gas). The Black-Oil Equations assumes that no mass transfer behavior between the water phase and the other phases occur, and is often used to model low-volatility oil systems (Peaceman, 1977).

Further, we incorporate explicit equality and inequality constraints on the well rates to model the physical limitation of the reservoir and the wells. We require that the sum of the well-rates add up to zero, which enforces a reservoir pressure condition. Also, we require that the well-rates satisfy bounds. Incorporting these *explicit* constraints, using a finite volume spatial discretization and a backward-Euler time-stepping scheme yields the following discretized optimal control problem:

$$\min \qquad J_{\Delta t}(q) = \Delta t\sum_{k=1}^{N} l(t^k, s^k, q^k) \tag{7}$$

$$s.t. \qquad e^T q^k = 0 \tag{8}$$

$$q_{min} \le q^k \le q_{max}\,, \tag{9}$$

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

$$\begin{bmatrix} g(t^{k+1}, s_a^{k+1}, p^{k+1}, q^{k+1}) \\ f(t^{k+1}, s_a^{k+1}, p^{k+1}, q^{k+1}) \end{bmatrix} := \begin{bmatrix} q - Ap^{k+1} \\ D^{-1}(q_a - \tilde{A}p^{k+1}) \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{s_a^{k+1}-s_a^k}{\Delta t} \end{bmatrix}, \tag{10}$$

and the function l hides the objective function integrand:

$$l(t^k, s_a^k, q^k) = \sum_{i\in P}\alpha(1-s_a^k)q_i^k + \sum_{i\in P}\frac{\beta}{2}s_a^k(q_i^k)^2 + \sum_{i\in I}\gamma q_i^k\,. \tag{11}$$

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

$$A_{i,j} = -T_{i,j}\lambda_{t_{i,j}} \qquad A_{i,i} = \sum_j T_{i,j}\lambda_{t_{i,j}} \tag{12}$$

$$D_{i,i} = \phi_i\cdot|\Omega_i| \tag{13}$$

$$\tilde{A}_{i,j} = -T_{i,j}\lambda_{a_{i,j}} \qquad \tilde{A}_{i,i} = \sum_j T_{i,j}\lambda_{a_{i,j}}\,. \tag{14}$$

The transmissibility between cell i and j, $T_{i,j}$ is defined as

$$T_{i,j} = \frac{K_{i,j} A_{i,j}}{l_{i,j}}\,, \tag{15}$$

where the length between the barycenter of the cells i and j are denoted as $l_{i,j}$ and the area of the face between two cells are denoted as $A_{i,j}$.

Applying the optimality conditions to the fully discretized optimal control problem above yields the following adjoint evolution scheme. For $k = N-1, \ldots, 1$, simultaneously solve for the adjoint variables λ_s^k and λ_p^k in the following equation:

$$-\frac{\lambda_s^{k+1} - \lambda_s^k}{\Delta t} = D_s f(\ldots^k)^T \lambda_s^k - D_s g(\ldots^k)^T \lambda_p^k - \nabla_s l(\ldots^k) \tag{16}$$

$$0 = -D_p f(\ldots^k)^T \lambda_s^k + D_p g(\ldots^k)^T \lambda_p^k\,. \tag{17}$$

After completing the adjoint evolution, the directional derivative of the objective function with respect to the wellrates q can be obtained from the following expression:

$$\nabla J(q)\delta q = \sum_{k=1}^{N} \Delta t [\nabla_q l(\cdot^k) - D_{q^k} f(\ldots^k)^T \lambda_s^k + D_{q^k} g(\ldots^k)^T \lambda_p^k]^T \delta q^k\,. \tag{18}$$

In order to use `TSOpt` to solve OWRA, we create a reference and an adjoint simulator, as well as a stack type that holds the primary variables p and s_a. These simulators were then used to create an existing `Jet` class in `TSOpt`. The following show the algorithmic work behind the forward and the adjoint simulator.

- `BlackOil` **Forward Simulator** (solves discretized Black-Oil equations)

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

  Solution approach:

  * solve for p^{k+1} and s^{k+1} simultaneously using Newton-Raphson
  * linear system solves via UMFPACK

- `BlackOil` **Adjoint Simulator**

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

  Solution Approach:

  * solve for λ_p^{k+1} and λ_s^{k+1} (linear system solve via UMFPACK)
  * note: gradient accumulation is accomplished during the adjoint simulation

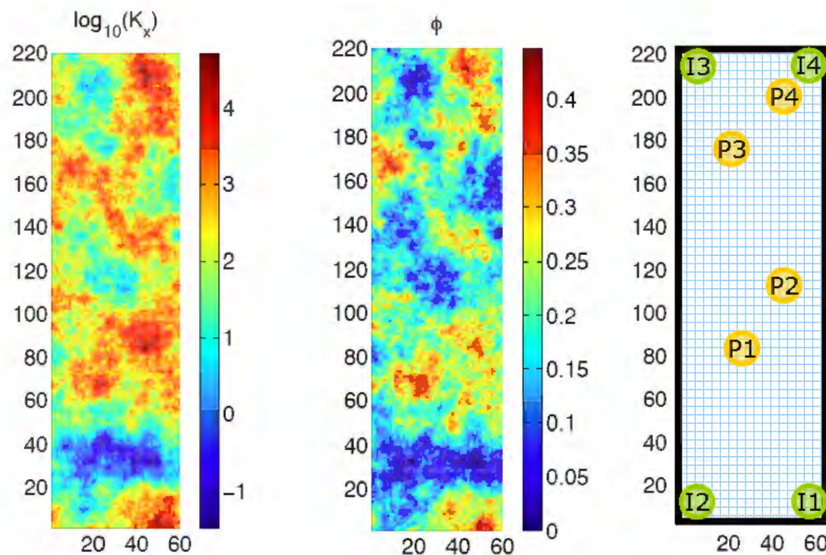

Figure 6: [l] Porosity and permeability plot of the SPE10 model, top layer; [r] Placement of injector (I) and producer (P) wells in the domain.

We now try to solve OWRA on a rectangular domain, composed of 220×60 cells, over a 200 day timespan. We set the initial guesses of the wellrates to 10 barrels per day, for all the injecting and producing wells. Further, we impose that wellrates must stay within the bound $[0, 20]$ barrels per day. We couple the `BlackOil` simulators with `TSOpt` and the optimization software `IPOpt`. Further, we use the data seen in figure 6 for porosity, permeability and well locations. Looking at the plot of producers and injectors we see that producing well 4 and injecting well 4 are deemed “too close” to one another. After some time, the water that placed into the reservoir by injector 4 will immediately be ejected by producer 4, implying a waste of resources. Hence, we expect to see the optimizer to throttle the rates for either producing well 4 or injecting well 4. As we can see from figure 8, the fourth producing well’s rate was indeed throttled by the optimizer. We note that the objective function (figure 7) significantly increased in value, over the initial guess for the wellrates.

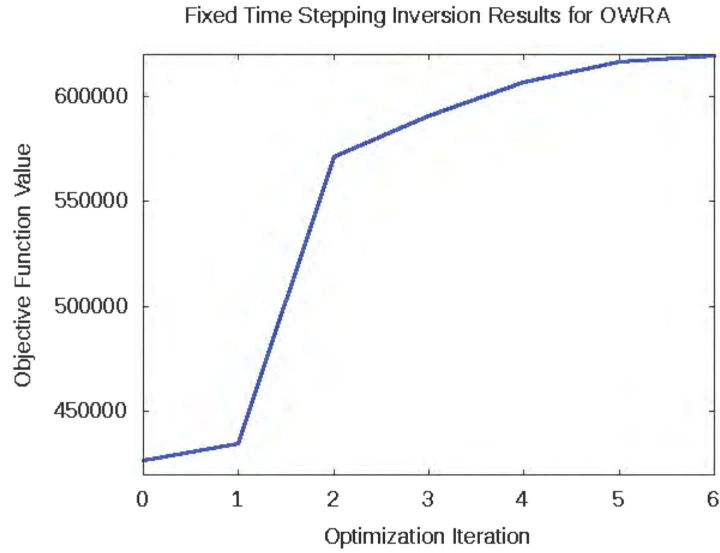

Figure 7: Objective function for the fixed time-stepping approach to solving OWRA.

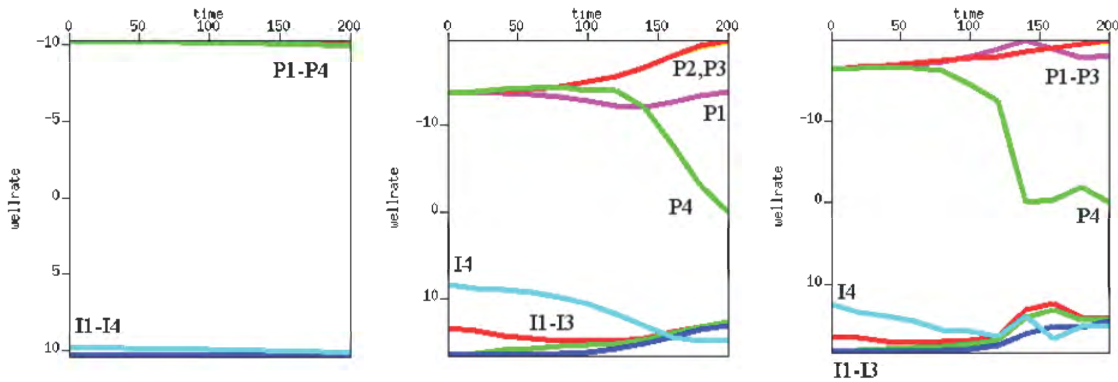

Figure 8: Progression of the control parameter for the first [l], third [m] and sixth [r] optimization iteration, taking fixed time steps. Note the well labels on the figure: "P" represents the producing wells and "I" represents the injecting wells.

BIBLIOGRAPHY

D.R. Brouwer and J.-D. Jansen. Dynamic optimization of waterflooding with smart wells using optimal control theory. *SPE*, 2004.

Marco Enriquez. A C++ class supporting adjoint state methods. Master's thesis, Rice University, 2008.

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

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

Michael Hinze and Julia Sternberg. A-Revolve: An adaptive memory-reduced procedure for calculating adjoints; with an application to computing adjoints of instationary navier-stokes system. *Optimization Methods and Software*, 20:645–663, 2005.

Anthony D. Padula, Shannon D. Scott, and William W. Symes. A software framework for abstract expression of coordinate-free linear algebra and optimization algorithms. *ACM Trans. Math. Softw.*, 36(2):1–36, 2009. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/1499096.1499097.

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

P. Sarma and K. Aziz. Implementation of adjoint solution for optimal control of smart wells. *SPE*, 2005.

Andreas Wachter. *An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering*. PhD thesis, Carnegie Mellon University, 2002.

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