A Time-Stepping Library for Simulation-Driven Optimization

William W. Symes *

ABSTRACT

The Timestepping Simulation for Optimization ("TSOpt") library provides an interface for time-stepping simulation. It packages a simulator together with its derivatives ("sensitivities") and adjoint derivatives with respect to simulation parameters, and uses the aggregate to define a Rice Vector Library `Operator` subclass.

INTRODUCTION

Control, design, and inverse problems seek to optimize a functional of the solution of a system of differential equations with respect to parameters of the system. The parameters may enter through coefficients, initial conditions, boundary conditions, or a combination of these. The functional may depend on the solution throuthout its domain of definition,or just on a subset. For all of these possibilities, the key feature of such problems is the implicit dependence of the objective on the parameters: computing its value requires solution of a system of equations, i.e. a simulation. Therefore we will refer to all such problems as *simulation driven optimization*. In control theory, the solution of the differential equation system models the state of the physical system, so that the solution is known as the *state*, the differential equations as the *state equation(s)*. The parameter vector in such problems is the *control*, which is to be adjusted so that *cost* functional of the state achieves an extreme value. I will use this control-derived terminology throughout, for all classes of problems under discussion.

Simulation driven optimization is a constrained optimization problem, in which the state equation constrains the control and state vectors on which the objective or cost depends. Both feasible point ("black box", "nested analysis and design (NAND)") and infeasible point ("all at once", "simultaneous analysis and design (SAND)") approaches have attractions and have been applied to various problems. The relative advantages of the two approaches are poorly understood at present.

For time-dependent problems, the (discretized) system of differential equations is block triangular (explicit schemes) or can be regarded as formally block triangular (implicit

*The Rice Inversion Project, Department of Computational and Applied Mathematics, Rice University, Houston, TX 77005-1892 USA, email `symes@caam.rice.edu`

schemes supplied with a step solver), with the blocks corresponding to time levels. Solution of these block triangular systems is called *time stepping*. Time stepping introduces a commonality amongst time dependent simulators: they share a certain abstract structure, which is available for exploitation in constructing software libraries for time dependent simulation driven optimization.

Depending on the nature of the (continuum) problem, it may be reasonable to approach the solution of a simulation-driven optimization problem by means of Newton's method or one of its many approximations. These descent methods require computation of the *gradient* of the cost function, and possibly its *Hessian*, in addition to the cost function value. These computations may share intermediate results, and efficiency demands that this sharing be realized in the implementation. Moreover the code for these computations must be packaged so that coupling to an optimization algorithm is possible.

The software package *Timestepping Simulation for Optimization* ("TSOpt") described in this report provides an object-oriented approach to the construction of efficient simulation-driven optimization applications. The advantages of object orientation for coupling simulation and optimization are well-established and have been discussed at length elsewhere (Gockenbach et al., 1999; Benson et al., 2000; Deng et al., 1996; Meza, 1994). TSOpt packages the simulator and its various derivatives in a single object, and provides a natural interface to optimization algorithms via the *Rice Vector Library*.

The Rice Vector Library

This library ("RVL" in the sequel) is the software framework for TSOpt . RVL is a collection of C++ base classes which realize in code the principal mathematical components of calculus in Hilbert space, the natural conceptual realm of Newton-based optimization. The base classes of RVL are

- calculus classes: `Space` (vector spaces), `Vector` (vector), `Functional` (scalar valued function), `Operator` (vector valued function), `LinearOp` (linear operator), and
- data management classes: `DataContainer` (abstract data container), `LocalDataContainer` (concrete data container), `FunctionObject` (encapsulated function, evaluated by `DataContainer` objects).

For detailed descriptions of these classes and the design of RVL, see (Symes et al., 2005).

RVL is written in ISO C++. All base classes are templated on a scalar type (the field over which related vector spaces are defined), which we shall mostly suppress in the following discussion.

Design goals for TSOpt

- Define a generic `RVL::Operator` interface for use in the NAND approach to time domain simulation driven optimization;

The cost functions of these problems depend on the the solution of the state equations or on a part or *sampling* of the solution. The dependence can be realized in various ways. RVL provides appropriate tools for creating functions of vectors. These tools produce the required derivatives as well as values, and the resulting `RVL::Functional` objects interface directly with optimization algorithms implemented in RVL. The ability to couple disparate codes through the sharing of abstract types is the chief justification for an object-oriented approach to simulation-driven optimization. The `RVL::Operator` objects constructed by TSOpt produce simulator (RVL::Vector) outputs, which serve as inputs to the cost function implemented as an `RVL::Functional`. tt RVL::Operators combine functions and their derivatives in one object with persistent state, which permits the sharing of intermediate data necessary for efficiency in many computations. This feature of `RVL::Operator` is critical to the design of TSOpt.

The sheer size of time-dependent simulations - for a 3D problem, the full time history of the state may have dimension 10^{14} or more - argues for the NAND approach, in which only a few time levels of the state need be stored. As computer memory capacity increases, more problems will come in range of the SAND approach, of course. TSOpt is aimed primarily at NAND, but many of its components are useful in constructing SAND applications as well.

- factor the simulation problem so that the inputs to the TSOpt `RVL::Operator` construction come as close as possible to representing independent concepts, which can be implementeded independently;

In the current version of TSOpt, the ingredients of a simulation are (i) the static or single time step description of the state and control data structures, (ii) the dynamic specification, i.e. the "right hand side" of the (continuum) state equation, (iii) the time stepping rule, which creates the discrete time step out of the RHS of the state equation, and (iv) a time keeping device, storing and providing access to start time, end time, steps, etc. Through the use of abstract RVL types to represent the data structures shared amongst these ingredients, the code that implements them is kept as independent, and therefore as reusable, as possible. For example, time stepping rules (Euler, Runge-Kutta,...) are formulated in a manner independent of other problem details, so that they may be used across a wide variety of projects.

- provide tools to test the validity of the inputs, and a contract that the `Operator` code is valid if its inputs are;

The components of TSOpt's `RVL::Operator` implementation are relatively simple, and validating their constructions is much simpler than validating the entire timestepping code *ab initio*. Therefore the capability envisioned in this goal should be a considerable help in rapid construction of applications.

The subtlety here is in the meaning of "valid". The essential task of TSOpt is to provide an accurate simulator, together with accurate approximations to associated objects

(derivatives, adjoint derivatives, etc.). The accuracy of the simulator is not an aspect of validity of the TSOpt code *per se*, but the consistency of the various components certainly is. Tests for consistency are straightforward in some cases, not in others. For example, as we point out below, adaptive gridding sometimes prevents a straightforward relationship between the *discrete* simulator and its derivative simulator - that is, the latter does not (cannot!) produce the derivative of the former, to floating point accuracy. When the derivative relationship does hold on the discrete level, it is possible to test code validity directly, as we shall show in the final section of this report. Otherwise, the tests must of necessity be less direct.

- achieve efficiency equivalent to that of the best procedural implementations on "large" problems;

This criterion actually confers considerable freedom on the design. It means that a judicious amount of virtual function call overhead can be absorbed in other arithmetic costs, so long as the number of virtual function calls is roughly independent of the size of the state and control at a single time level. A previous incarnation of TSOpt (Gockenbach et al., 2002) has already shown that this goal is eminently achievable.

- accommodate adaptive time steps and adaptive spatial gridding for PDEs;

This is essential: adaptivity is enabling technology in many applications, making simulations feasible which would otherwise be entirely out of reach, in some cases for the foreseeable future.

One perhaps surprising consequence is that the natural precise relations between the simulation operator and its derivatives are necessarily lost on the discrete level! That is, the "derivative" (with respect to control parameters) provided by TSOpt is generally only an approximation to the derivative of the operator provided by TSOpt - provided that the latter has a derivative! The only contract implicit in a correct TSOpt implementation is that the various discrete objects are convergent approximations to the underlying continuum objects. [This approach goes under the unfortunate name "optimize then discretize" in the recent control literature.] The unavoidable nature of this imprecision does not seem to be widely appreciated, so we present some very simple examples in an appendix which illustrates this point.

In some instances it is possible to preserve the continuum relations amongst the discrete attributes of a TSOpt `Operator` object. I strongly recommend taking advantage of any such opportunity, perhaps by means of automatic differentiation (Gockenbach and Symes, 2002; Gockenbach et al., 2001; Gockenbach et al., 2002). The task of code verification is much easier if the derivative is actually the derivative on the discrete level, similarly for the adjoint, etc.

- permit internal representation of state, control to differ from external representation appearing in the `RVL::Operator` evaluation interface, and to be computed as part of evaluation;

Adaptive gridding makes this feature essential.

- incorporate an optimal solution to the checkpointing problem of the adjoint state method;

If the evolution of the system is time reversible (on both discrete and continuum levels), the most efficient implementation of the adjoint state method (the natural computation of the adjoint derivative) simply steps the reference state backward in time. For time irreversible dynamics (the vast majority of problems), TSOpt uses an optimal checkpointing algorithm (Griewank, 1992; Griewank, 2000). This is also enabling technology, as it renders feasible many adjoint computations whose complexity would otherwise make them inaccessible.

- accommodate one step and multistep methods in the same framework, also both implicit methods and implicit formulations of differential equations, as are for instance natural in finite element discretizations of time-dependent PDEs. .

There is no reason not to do so. We conjecture that a modest extension of TSOpt could treat differential-algebraic equations as well.

- allow for multicomponent vector representation of control;

In order to do the least damage to natural external representations of control data structures, TSOpt accommodates `RVL::ProductSpace` realizations of control vector spaces. These are natural in those (common) instances in which the control is made up of a number of *a priori* independent components. For example, control problems based on linear elasticity may use as controls the (up to 21) Hooke tensor components, or useful combinations of these (eg. Poisson's ratio, shear and compressional moduli,...), and these may be treated as independent data structures combined in a Cartesian product via the `RVL::ProductSpace` interface.

- accommodate multisimulations, in which output of simulator is multicomponent vector, each component of which is itself the output of a simulation;

Many inverse problems (for instance those occurring in seismology, meteorology, and oceanography) seek to adjust parameters to fit simultaneously the results of many separate but similar experiments, differing in experimental parameters (eg. location of measurement devices) but otherwise based on identical physics. Each experiment can be simulated; the data is to be fit by the aggregate of many simulation outputs. Such multisimulations could be modeled by creating suitable block-structured operator types. Instead, the current version of TSOpt permits the output (data, state) vector have product structure, each factor of which represents a separate experiment. This implicit block structure is consistent with the implementation of the TSOpt `RVL::Operator` by means of `RVL::FunctionObject`s, which is the next and final design goal.

- implement using `RVL::FunctionObject` interface.

`RVL::FunctionObject` is the interface for virtually all interaction with data in RVL applications. `RVL::Vector` objects for example do not expose their data - indeed they cannot, as it is likely to be stored out of core or distributed around a network. However they can evaluate `RVL::FunctionObjects`. To interact with the data of an `RVL::Vector` in virtually any way, it is necessary to write a `RVL::FunctionObject` to do the actual interacting. The innards of the TSOpt `Operator` class are like any other RVL application in this respect.

`RVL::Vector`s own `RVL::DataContainer`s and delegate evaluation of `RVL::FunctionObject`s to them. A vector representing a product data structure owns a `RVL::ProductDataContainer`. The evaluation methods of `RVL::ProductDataContainer` loop over the factor `RVL::DataContainer`s, evaluating the input `RVL::FunctionObject` on each factor in turn. This built-in looping over factors automatically achieves the last two mentioned design goals, so long as TSOpt `RVL::Operators` are implemented using `RVL::FunctionObject`s.

Parallelism is an enabling technology for large-scale scientific computation, and it might seem odd that distributed computation does not appear in the preceding list. In fact the core TSOpt classes are almost sufficient in themselves for parallel execution in SPMD mode (Pacheco, 1997), using either a domain decomposition approach for problems too large to execute in single processes, or a queue-of-tasks design for multisimulations, or a combination of these two approaches. The key to parallel TSOpt is realization of appropriate parallel data structures as `RVL::DataContainer` subclasses. Once that step is accomplished, a slightly revised TSOpt `RVL::Operator` definition enables SPMD execution. Thus parallelization of TSOpt is essentially a by-product of its design. A nontrivial example of parallel TSOpt appears in the final section.

Relation to the FDTD Package

TSOpt is a direct descendent of the *FDTD* package described in (Gockenbach et al., 2002), which relied on a predecessor library of RVL, the *Hilbert Class Library* ("HCL"), for its basic constructs (Gockenbach et al., 1999). FDTD shared a number of features with TSOpt. It used object orientation to organize the interface between simulator and optimization algorithms. Experiments comparing FDTD to raw Fortran implementations showed that object orientation exacted a negligible performance penalty. Also FDTD illustrated the *structured* use of automatic differentiation ("AD") to produce applications beyond the reach of contemporary AD tools, by focussing the use of AD on those code components unique to each application, "hardwiring" the common (abstract) computations. TSOpt also permits this structured use of AD, as illustrated in the final section of this report.

FDTD differs from TSOpt in making no provision for adaptation, or for any distinction between internal and internal representation of state and control. Thus FDTD applications require the user to work directly with (indeed, to supply) the internal details of the simulation. Also FDTD was implemented using the HCL calculus classes throughout.

TSOpt's reliance on the `RVL::FunctionObject` interface simplifies both the overall design and the transition to parallel computation (Scott, 2001).

Contents of this report

The next section establishes some necessary notation by recounting the mathematical structure of simulation-driven optimization problems. The following section describes basic implementation concepts of TSOpt. Some of these involve types that do not figure in the user interface, and which the user may not actually see except by accident. However understanding these internal implementation decisions clarifies the rationale behind the public structure of the operator interface.

Section 4 describes the types appearing in the public operator interface. These are the types which users will extend to generate new TSOpt applications, and their description is given at appropriate length. Section 5 presents the operator interface: once the user understands the constituent types, this interface should be transparent. One of the two examples presented in Section 6 is a simple ODE control problem, which has been used throughout the preceding sectionsto illustrate various features of the TSOpt design. The other example discussed in Section 6 is more substantial: a 2D model seismic inverse problem, exercising most of TSOpt's design. Timings using this example justify our assertion that the performance penalty paid for TSOpt's object orientation is negligible, provided the problem is of even modest size.

MATHEMATICAL FRAMEWORK

Many control, inverse, and optimal design problems for time-dependent physics take the form

$$\min_{d,c} J[d,c] \tag{1}$$

where the *control* $c(t)$ and the *observations* or *data* $d(t)$ are constrained by the *state equation*

$$\frac{du}{dt}(t) = H(u(t),c(t),t),\ 0 \le t \le T;\ H,u \equiv 0, t < 0 \tag{2}$$

and the *sampling rule*

$$d(t) = S(t)u(t),\ 0 \le t \le T. \tag{3}$$

in which $S(t)$ is a linear operator valued function of t.

Note that this formulation includes nonzero initial conditions, either dependent on or independent of the control c: to achieve $u(0) = u_0$ (or $u_0[c]$), define

$$H[u(t),c(t),t] = u_0\delta(t) + \bar{H}[u(t),c(t),t]$$

with $\bar{H}$ continuous on $0 \le t \le T$. In this way the formalism (2) accomodates the case in which the control is the initial state $u(0)$ of the system, for example.

We regard J as a function on a product $\mathbf{D}\times\mathbf{C}$ of Hilbert spaces, with $\mathbf{D} = L^2([0,T], D, d\mu)$ for another Hilbert space D and a positive measure μ, and set

$$\langle d_1, d_2\rangle_{\mathbf{D}} = \int_0^T d\mu(t)\, \langle d_1(t), d_2(t)\rangle_D$$

We don't specify the details of the $\mathbf{C}$ inner product $\langle c_1, c_2\rangle_{\mathbf{C}}$; it will be necessary to compute adjoints of mappings between $\mathbf{C}$ and D, and we assume that the user can arrange to do so.

Thie problem posed by equations (1), (2), and (3) is not the most general possible form of the type considered. For example, implicit state equations of the form

$$F\left(\frac{du}{dt}, u, c, t\right) = 0$$

are natural in some applications, either as expressions of physical principle or as a result of (semi-) discretization. However it covers many examples, including those presented in detail in this paper. The generalizations necessary to accomodate broader classes of problems are quite straightforward, and will be mentioned in due course.

We assume that all components of this description are sufficiently regular in that the various constructions described here make sense, without giving precise hypotheses. The computations to follow are therefore formal.

The control c may or may not actually be time-dependent. The formalism accommodates the case (Mayer form control problems, for example) in which the data is a function of the state at only one *sample time*, say the final time $t = T$, by setting $d\mu(t) = \delta(t-T)dt$ (of course, the implementation of S in that case would need only produce output at or near $t = T$). In many problems of this type, the cost or objective function J is an integral over the time range of the simulation, the integrand at time t depending on $d(t)$ and $c(t)$.

The state and sampling equations (2) and (3), regarded as hard constraints, express d as a function $F[c]$ of c. Following terminology common in the literature on inverse problems, we call this function the *forward map*. Expressing d as a function of c within $J[d, c]$ gives the reduced, or NAND, form of the problem:

$$\min_c J^{\mathrm{red}}[c],\ \ J^{\mathrm{red}}[c] = J[F[c], c] \tag{4}$$

Newton's method and its relatives for NAND require that the gradient of J^{red}, and possibly its Hessian, be made available to an optimization algorithm. These are easily expressed in terms of the derivatives of J and F and their adjoints.

$$\begin{aligned}\langle \nabla_c J^{\mathrm{red}}[c], \delta c\rangle &= D_d J[F[c], c] D_c F[c]\delta c + D_c J[F[c], c]\delta c \\ &= \langle \nabla_d J[F[c], c], D_c F[c]\delta c\rangle_{\mathbf{D}} + \langle \nabla_c J[F[c], c], \delta c\rangle_{\mathbf{C}}\end{aligned} \tag{5}$$

whence

$$\nabla_c J^{\mathrm{red}}[c] = D_c F[c]^* \nabla_d J[F[c], c] + \nabla_c J[F[c], c] \tag{6}$$

The derivative of F is (at least formally)

$$D_c F[c]\delta c = S\delta u \tag{7}$$

where δu is the solution of another evolution problem, sometimes called the *sensitivity equations*:

$$\frac{d\delta u}{dt}(t) = D_u H[u(t), c(t), t]\delta u(t) + D_c H[u(t), c(t), t]\delta c(t),\ \delta u(0) \equiv 0,\ t < 0 \tag{8}$$

To proceed we must choose a Hilbert structure for the state: we will assume that $u \in C^1([0,T], U)$ in which U is another Hilbert space representing some assumed regularity of the state. Given $d \in \mathbf{D}$, define the *adjoint state* field $w \in C^1([0,T], U)$ as the solution of the backwards - in - time evolution problem

$$\frac{dw}{dt}(t) = -D_u H[u(t), c(t), t]^* w(t) - S(t)^* d(t),\ w \equiv 0 \,\text{for}\, t > T \tag{9}$$

in which S is regarded as a map from U to D and its adjoint defined accordingly. Then

$$\langle d, DF[c]\delta c\rangle_{\mathbf{D}} = \int_0^T d\mu(t)\, \langle d(t), S(t)\delta u\rangle_D$$

$$= -\int_0^T dt\, \langle \left(\frac{dw}{dt}(t) + D_u H[u(t), c(t), t]^* w(t)\right), \delta u(t)\rangle_U$$

$$= \int_0^T dt\, \langle w(t), \frac{d\delta u}{dt} - D_u H[u(t), c(t), t]\delta u\rangle_U$$

$$= \int_0^T dt\, \langle w(t), D_c H[u(t), c(t), t]\delta c(t)\rangle_U$$

From this last equation we can identify

$$DF[c]^* d = D_c H[u, c, \cdot]^* w \tag{10}$$

in which $D_c H[...]^*$ is the adjoint of $D_c H[...]$.

A more explicit prescription for compuing ∇J^{red} requires a more explicit description of $\mathbf{C}$. For examples of such explicit descriptions, see the final section. The gist of the above computations is that ∇J^{red} can be computed for the price of a single additional solution of an evolution problem, the adjoint state equation (9), of the same type as the state equation (2), plus a few more straightforward computations.

Discretization in time (and space, for PDE problems) replaces the state equation (2) and its relatives with time stepping equations:

$$u^{n+1} = u^n + \Delta t^n \mathcal{H}^n(u^n, c^n),\ u^n = 0, n < 0 \tag{11}$$

Here $u^n \simeq u(t^n)$, etc. The function $\mathcal{H}$ characterizes the time stepping rule: for example, $\mathcal{H}^n(u,c) = H(u,c,t^n)$ for the forward Euler method. The method (11) is (formally) one

step, but of course multistep methods can be disguised in this form, so (11) is a perfectly general basis for the design of TSOpt.

The sampling operator S is also discretized to depend on n. The discrete sampling rule becomes

$$d \leftarrow d + S^n(u^n) \tag{12}$$

i.e. there is no requirement that t^n be a time represented in the output data. Instead, S^n injects the current value of the state into the output data in whatever way is appropriate.

There is no presumption that $\Delta t^n = t^{n+1} - t^n$ should be independent of n, and indeed the time step may depend on u^n, i.e. be adaptive. The same goes for the spatial discretization, i.e. the structure of u^n may vary with n. In that case the time stepping rule (11) must be augmented with a discretization-changing operator.

The sensitivity equations (8) have a similar discrete representation

$$\delta u^{n+1} = \delta u^n + \Delta t^n [\mathcal{H}_u^n(u^n, c^n)\delta u^n + \mathcal{H}_c^n(u^n, c^n)\delta c^n],\ \delta u^n = 0, n < 0 \tag{13}$$

A very important point: the functions $\mathcal{H}_u, \mathcal{H}_c$ should define a consistent time stepping method for the sensitivity equations (8), of the same order of accuracy as the reference method (11), but are not necessarily in general the partial derivatives of $\mathcal{H}$. Similarly, TSOpt permits the time steps $\{t^n\}$ appearing in the derivative method (13) to differ from those which appear in the reference method (11). This freedom is absolutely essential for adaptive time stepping, as the examples in Appendix A demonstrate. As mentioned in the introduction, a side effect is that the computed discrete derivative δu^n may differ by a discretization error from the actual derivative of u^n with respect to the control parameters.

On the other hand, it is also true that when $\mathcal{H}_u, \mathcal{H}_c$ are actually partial derivative of $\mathcal{H}$, and when the time steps used for (11) and (13) are the same, then δu is actually the derivative of u on the discrete level as well. Precision in this relationship makes it rather easier to verify the implementation than in the general case. Automatic differentiation tools can make the construction of such precise discrete derivatives relatively straightforward and reliable. The examples presented in the last section satisfy this constraint, and will illustrate its value.

Finally, the discrete approximation to the adjoint map $DF[c]^*$, and eventually the discretized objective gradient, is computed via a discrete version of the adjoint state equation

$$w^n = w^{n+1} + \Delta t^n \mathcal{H}_u^{a,n}(u^n, c^n)w^{n+1} + (S^n)^* d,\ w^n = 0, n > N \tag{14}$$

(N being the time index corresponding to $t = T$).

Then the discrete approximation to $DF[c]^* d$ is

$$\sum_n \mathcal{H}_c^{a,n} w^n \tag{15}$$

which can be accumulated during the backwards-in-time stepping of (14).

Again, if the approximate adjoint derivatives $\mathcal{H}_u^a, \mathcal{H}_c^a$ are actually the adjoints of the partial derivatives of $\mathcal{H}$, and the two discrete evolutions share time steps, then the computed adjoint (approximation to $DF[c]^*$) will actually be the adjoint of the computed $DF[c]$, and this is relatively straightforward to check. Automatic differentiation tools can also be a great help in constructing machine precision adjoints.

SIMULATOR STRUCTURE

The design criteria set out in the introduction imply a number of constraints on the implementation of TSOpt. Most of these implementation details are invisible to the user of TSOpt: a later section describes the `Operator` interface, which is the only public interface with which the user need be concerned. However this interface involves several additional abstract types, the principal simulator components described in the next section, and the rationale for their structures follows from various internal design decisions. This section therefore overviews a number of objects with which the user will ordinarily not be directly concerned, as they are completely implemented in the TSOpt package and hidden behind the `Operator` interface. A general understanding of these internals will inform the discussion of principal simulator components in the next section.

Unary Interface: One major structural decision flows directly from the nature of simulation, especially of multisimulation, and from TSOpt's reliance on the `RVL::FunctionObject` interface.

The RVL `FunctionObject` and `DataContainer` classes realize the *Visitor* design pattern (Gamma et al., 1994), in its *acyclic* variant (Martin, 2002). As is characteristic of this variant, the `RVL::FunctionObject` base class is *degenerate*, with no nontrivial methods, so does not constrain the interactions of its children with children of `DataContainer`, which play the role of visited Elements in the Visitor pattern. TSOpt relies on the *LocalRVL* package, auxiliary to RVL, which defines a collection of subclasses of `FunctionObject` and `DataContainer` centered around local access to data by address. The generic `LocalFunctionObject` visits an arbitrarily long list of `LocalDataContainer`s. Specialized `LocalFunctionObject` subtypes visit specified numbers of `LocalDataContainer`s: the unary, binary, ternary, and quaternary subtypes cover essentially all cases arising in practice. `UnaryLocalFunctionObject`s for example visit single `LocalDataContainer`s, via their evaluation (`operator()`) method with one `LocalDataContainer` argument, for example:

```
void UnaryLocalFunctionObject::operator()(LocalDataContainer &);
```

As evaluation is permitted to change the internal state of the `LocalFunctionObject`, the evaluation method is non-`const`.

The `RVL::DataContainer`s encapsulating output (data) of a TSOpt application may each themselves have arbitrary numbers of component `LocalDataContainer`s, which may not all be immediately accessible to a TSOpt process - their data may reside on disk, or

elsewhere on the network, and access may require a specific request. So the TSOpt `LocalFunctionObject` interface must accommodate arbitrary numbers of `LocalDataContainer`s, with the actual number determined at run time.

As is implicit in its name, the `LocalDataContainer` interface provides explicit, local access to data:

```
virtual int LocalDataContainer::getSize() const = 0;
virtual Scalar * LocalDataContainer::getData() = 0;
virtual Scalar const  * LocalDataContainer::getData() const = 0;
```

No modification of the evaluation method signature conveniently accommodates arbitrary, run-time determined numbers of arguments, not all immediately accessible to the process. The only available alternative in C++ is to evaluate the `FunctionObject` many times, and use the persistent state property of `FunctionObjects` as well as shared data to retain input and intermediate output information between evaluations. Therefore the top-level internal interface of TSOpt consists of a suite of `UnaryLccalFunctionObject`s implementing the simulator, its derivative, its adjoint derivative,...

The critical methods of the TSOpt `Operator` interface `TSOp` evaluate nput and output vectors on `LocalFunctionObject`s. The `LocalFunctionObject`s refer to a `Stencil` object, which is a data member of the `TSOp` class and is initialized on instantiation of a `TSOp` object. Simulation is accomplished by the `apply`, method, for which skeletal pseudocode is

```
void apply(Vector const & x, Vector & y)  const {
  ...
  TSInitFOR fi(sten);
  TSApplyFO f(sten,itmax,verbose);
  x.eval(fi);
  y.eval(f);
}
```

[I have left out boilerplate sanity checking details and comments, as will be the case with all code fragments presented in this report. I have also left out comments, template arguments, and other necessary details.]

The `RVL::Vector` objects `x` and `y` in the signature of the `apply` method implement their `eval` methods by delegation to the `eval` methods of their `DataContainer` data members. Each invocation of `DataContainer::eval` is an implicit loop over components. Eventually the call tree descends to the `LocalDataContainer` level, at which point control passes to the `LocalFunctionObject:::operator()` method body. Evaluation of `TSInitFOR::eval` on the input `x` (control) in the above method body initializes internal storage of `Stencil`, which is then used during the evaluation of `TSApplyFO` on the output `y` (data).

All of the classes mentioned so far (`TSOp`, `Stencil`, `TSApplyFO`,...) are fully implemented class templates. To instantiate objects of these classes, the user must supply

concrete realizations of a number of constructor inputs, which are described in the next section.

Adaptation TSOpt permits adaptation of internal details of the simulation - both the number and times of time steps and structure of the state vector itself (i.e. mesh or grid, for PDEs) may be computed on construction of the constituent `FunctionObjects`, or even updated during the simulation run. Moreover, internal structure may differ between simulation, derivative simulation, adjoint derivative simulation,..., though in most cases it will be necessary to enable communication between these structures, as is clear from inspecting the sensitivity equations (8) for example.

One immediate consequence of this design decision is that the external representation of the control and state data - the data structures holding this information in the ambient computational environment in which `TSOp` will be accessed - is very unlikely to be the same as the internal representation, with which the time stepping computation works. An intrinsic part of the *static* description of the state and control data structures will therefore be some means to translate between internal and external representations.

Decisions about keeping or updating time steps are part of the role of a time stepping method. The user-defined principal simulator components, to be described in the next section, will encapsulate the logic of these decisions, and record the selected time steps when necessary. The same is true of adaptive spatial grids, i.e. mutable structure of the internal state representation.

Multisimulation The version of TSOpt discussed in this report carries out multiple simulations with the same control input. Each component of the output (`y` in the pseudocode displayed above) is the output of one simulation out of the multisimulation implicit in the output data structure. The data structure encapsulated by `y` implies a loop over simulations, one corresponding to each `LocalDataContainer` component of `y`. Each component is therefore presumed to carry *on call* all necessary metadata required to specify those attributes of the simulation not inherent in the control.

The inclusion of such metadata, or auxiliary simulation information, in output data structures may be novel in some applications. Certainly many simulation codes have been written which produce only an array of scalars, or a flat file containing such an array, as output. However, such "raw" output does not usually contain within it enough information to interpret it properly in the context of the application, so represents inadequate data encapsulation. Seismic reflection processing is an example of a technology in which it has been normal practice for decades that all information necessary for interpretation and processing is integral to standard data structures. TSOpt assumes that output data structures are well encapsulated in this sense.

SIMULATOR INPUTS

The essential ingredients of a time stepping simulator are:

- the *static* definition of state and control, allowing for the possibility that the internal representation of these objects may be different from their external representation;
- the simulation *dynamics*, i.e. the evolution law of the system, embodied in the right-hand side of the state equation (2);
- a time discretization rule or *step*;
- some means of monitoring the time during the simulation, determining its start and end, storing the record of time steps if that is necessary, etc.

The four classes (and auxiliary types) discussed in the paragraphs to follow encompass these four functions. In each case, a discussion of the general rationale for the class precedes a detailed description of the interfaces and intended functionality. A simple example follows each discussion. The next section shows how these necessary steps come together in constructing a TSOpt application.

Statics

For reasons already discussed, the internal data structures used in a simulator may differ from those external data structures manipulated by the application using the simulator. The static aspect of the simulation therefore encompasses two major tasks: it must provide (i) access to the internal or working representation of state and control, and (ii) mechanisms for translation between internal and external or archival representations. To offer a compatibility guarantee, TSOpt combines these provisions as attributes of a single `Statics` object.

The `Statics` interface is the most complicated of the four TSOpt components, and will typically require the most time and effort. Fortunately large problem classes tend to share the same general static description, and with a bit of care the effort involved in a `Statics` implementation can be amortized over many applications. The simple example presented later in this section, for example, covers a very wide variety of ODE-based control and inverse problems. The regular grid `Statics` class sketched in the final section is another example of a `Statics` construction applying to a large class of problems.

The uses of internal and external representations are quite different. The external representation naturally interacts with the `RVL::Operator` interface `TSOp` which the simulator presents to its user applications. As mandated by the mathematical role of such objects, `TSOp` instances expose domain (control) and range (data) data structures implicitly, as (attributes of) `RVL::Space`s. RVL supplies a mechanism for constructing vectors in vector spaces, so applications can use the `TSOp` interface to allocate external control and data representatives.

User applications however are not (necessarily) aware of the simulator's internal data structures, as these may be computed as part of the simulation, and therefore cannot allocate internal state and control objects. In the TSOpt framework `Statics` object provides a means to build these internal data structures. Since these internal details of the

simulation will not be needed, or even visible, outside the simulator, the `Statics` object exposes *abstract factories* (Gamma et al., 1994) which return dynamically allocated state and control objects on demand. In the current version of TSOpt, these `ModelBuilder` factory objects have `build...` methods which return control of `LocalDataContainer` instances encapsulating internal state and control data structures. These methods are used throughout TSOpt wherever state or control workspace is required.

Translation between internal and external representation takes the form of *sampling* in many cases, often requiring interpolation. Accordingly, the objects that carry out the translation are `Sampler`s in TSOpt (even in those cases where entire data structures are recorded unaltered). The current version of TSOpt assumes that the sampling operation is *linear* - i.e. that any nonlinearity in the relation between the internal state and its external expression is incorporated in the computation of the internal state itself. [This assumption could be relaxed at the price of some added programming complexity, but so far has not impeded any application.] Since the internal data takes `LocalDataContainer` form, TSOpt represents sampling of state, resp. control, by pairs of `LocalFunction-Objects` expressing an adjoint pair of linear maps.

The mandate to accommodate *multisimulations* brings up a subtle point. Typical sampling rules (especially for data, but possibly also for control) depend on simulation parameters. If the `TSOp` object and its `Statics` data member are to encompass the entire multisimulation within their object lifetimes, the sampler `FunctionObject`s themselves are not the natural attributes of a `Statics` object. Instead, the `Statics` object must be able to produce a sampler `LocalFunctionObject` on demand, referencing simulation parameters. Accordingly, the `Statics` interface provides access to `SamplerFactory` objects, whose interfaces allow for interaction with simulation data.

The public part of the `Statics` interface therefore looks like this:

```
/** return reference to factory, which builds control and state instances
      (i.e. internal representations) */
virtual ModelBuilder & getModelBuilder() = 0;
/** return reference to factory, which builds translators between internal
      and external control representations */
virtual SamplerFactory & getControlSamplerFactory() = 0;
/** return reference to factory, which builds translators between internal
      and external state representations */
virtual SamplerFactory & getDataSamplerFactory() = 0;
```

ModelBuilder is a factory with two products:

```
/** dynamically allocate control LDC */
virtual LocalDataContainer * buildControl() = 0;
/** dynamically allocate state LDC */
virtual LocalDataContainer * buildState() = 0;
```

The allocation is dynamic, and storage returned by the `build...` methods must be managed by the calling objects. Note that the return type may be any sort of `LocalDataContainer`, including whatever metadata is useful in carrying out the simulation (grid axes, finite element mesh data,...)

For ODEs, i.e. simulators whose state and control data structures are simple arrays with no auxiliary data beyond length, we have implemented a simple `RnModelBuilder` class whose `build...` methods return `RnArrays` (the RVL simple array class). The base class header file `model.H` includes the `RnModelBuilder` definition. The object data consists of the lengths of the control and state vectors respectively: the principal constructor signature is

```
RnModelBuilder(int dimu, int dimc);
```

in which `dimu` represents the dimension of the state space, `dimc` the dimension of the control space.

SamplerFactory is also a factory with two products, and its public interface is almost as simple:

```
/** dynamically allocate forward sampler FO  */
virtual FwdSampler * buildFwd() const = 0;
/** dynamically allocate adjoint sampler FO */
virtual AdjSampler * buildAdj(LocalDataContainer const & d) const = 0;
```

The "direction" implicit in the names is internal $\rightarrow$ external: thus, `FwdSampler` objects map internal data to external data, and `AdjSampler` objects do the opposite, as in equations (12) and (14).

Because `AdjSampler` is somewhat simpler in structure, we discuss it first.

`AdjSampler` is a subclass of `LocalUnaryFunctionObject`. The crucial pure virtual method of the latter class, for which a concrete subclass must supply an implementation, is `operator()`:

```
virtual void LocalUnaryFunctionObject::operator()(LocalDataContainer & x) = 0;
```

This method is left pure virtual in `AdjSampler`: it defines the sampling operation, which is of course application-dependent.

The virtual constructor for `AdjSampler` objects declared in the `SamplerFactory` interface builds a an `AdjSampler` (or rather an instance of a concrete subtype), by calling the constructor with any necessary arguments, then the initialization method

```
virtual void AdjSampler::set(LocalDataContainer const & x)
```

which stores an internal (and independent) copy of its argument. The `operator()` method will sample this internal buffer onto its target. The `SamplerFactory::buildAdj` interface takes a `LocalDataContainer` argument, which is the data to be copied to the internal buffer of a `AdjSampler` instance. Typically the sampling operation will depend on metadata supplied with a particular subtype of `LocalDataContainer`, which is why implementation of `operator()` must be deferred to concrete base classes.

The only class methods particular to the TSOpt sampler classes (as opposed to inherited from their bases) provides access to simulation time:

```
virtual void AdjSampler::setTime(Scalar _t) { t = _t.; }
virtual Scalar getTime() const { return t; }
```

(`FwdSampler` has an identical pair of methods.) As the display suggests, thess methods are *implemented* - as they are declared virtual, the user may override either or both, but this should seldom be useful or necessary. The TSOpt code uses `setTime` to pass time information to the samplers. This is necessary since sampling decisions are typically based on simulation time. For example, Mayer form control problems, in which only the final value of the state figures into the cost, will use a sampler which is a no-op unless the time is equal to the final simulation time (up to a tolerance which would also need to be a data member of the sampler subclass, for example, Mayer form control problems, in which only the final value of the state figures into the cost, will use a sampler which is a no-op unless the time is equal to the final simulation time (up to a tolerance which would need to be a data member of the sampler subclass, and therefore must be supplied by the `SamplerFactory` object hence should be part of the latter's object data). The `setTime` method permits a calling unit to pass the time to the sampler object.

The following code sketch shows a typical use case for `AdjSample`:

```
SamplerFactory fac(...);
LocalDataContainer y(...);
AdjSampler * f = fac.buildAdj(y);
LocalDataContainer x(...);
f.setTime(t);
(*f)(x);
```

The effect is to adjoint-sample `y` onto `x`, provided that the time `t` is a sampling time. Here "sample" means: carry out an implicitly linear interpolation, projection,..., and "adjoint-sample" is means: apply the transpose linear mapping, adjoint to sampling.

Note that all necessary uses of the the sampler classes are already coded into the implemented TSOpt classes. We show the typical use case above to assist the reader in imagining how these classes might be implemented.

`FwdSampler` is a subclass of `UnaryLocalReduction`, a so-called reduction object. Its `operator()` method has a slightly different signature than that of `UnaryLocalFunction-Object`:

```
void FwdSampler()::operator()(LocalDataContainer const &);
```

Note that the argument is declared `const`: it is to be used in a read-only sense, and sampled *onto* an internal `LocalDataContainer` object initialized when the `FwdSampler` object is constructed. The `SamplerFactory` must therefore retain as data members any metadata necessary to instantiate appropriate `LocalDataContainer` internal buffers for its `FwdSampler` product.

Its role in data generation and adjoint state computation dictates that the appropriate initialization of the internal buffer of the `FwdSampler` is a zero array of data. Invocation of the `operator()` should *sample* its argument and *accumulate* the result (by addition) onto the internal buffer. A class method exposes the internal buffer in read-only form for

```
LocalDataContainer const & FwdSampler::get() const;
```

The complete public interface of the base `FwdSampler` class consists of the pure virtual methods just described (`operator()`, `get()`) and the implemented timing methods `setTime()`, `getTime()` which share both signature and implementation with `AdjSampler`.

The names also imply that these function objects implement a linear operator adjoint pair: indeed it is part of the current framework, as mentioned in the preceding section, that sampling in TSOpt is *linear*. Note that core RVL provides a `LinearOpFO` class which constructs a `LinearOp` from a pair of function objects such as the sampler function objects discussed here. This construction makes a simple test of the adjoint relationship (built into `LinearOp`) immediately available. The appendix on testing describes the necessary framework for such adjoint tests.

A simple example of this setup is the `RnSnapSampler` family, which assumes that the data structures of internal and external representations are simple arrays (represented by `RnArray` objects), and are the same. The factory class constructor takes a dimension specification, a sample time, and a tolerance for fitting the sample time. The sampler objects own buffers of the specified size, allocated on construction. When the time (set from some external source by `setTime`) coincides with the sample time to within the specified tolerance, `RnSnapFwdSampler::operator()` copies its argument onto the buffer, i.e. takes a "snapshot" of the input field, hence the name. The `RnSnapFwdSampler::get()` method returns a `const` reference the buffer. The `RnSnapAdjSampler` methods do exactly the opposite copies.

The class definitions of the `RnSnapSampler` family are included in the base class header file `sample.H`.

A Simple Statics Class: For test purposes, and because it may be genuinely useful for some ODE control problems, we have combined the simple `Rn...` examples described in the preceding paragraphs.

The `RnStatics` class combines `RnSpaces` for domain and range, an `RnModelBuilder`, and `RnSnapSamplerFactorys` for control and data. Typical use will be to set the sample

time of the control sampling to the beginning of the simulation; since the control sampler is invoked as part of initialization, this has the effect of storing the control parameters in their internal representation at the beginning of the simulation. Note that the use of the `RnSnapSampler` classes to input control data implies that the control is input at exactly one time, i.e. is autonomous. Similarly, the use of the same sampler classes for data implies that the state is sampled at only one time, typically the end of the simulation.

The main constructor signature is

```
RnStatics(int dimu, int dimc, Scalar tc, Scalar tu, Scalar dt);
```

The state and control dimensions `dimu` and `dimc` pass to the constructors of the `RnSpace`s, the `RnModelBuilder`, and the `RnSnapSamplerFactory`s, to which the `RnStatics` objects provide access, ensuring that these are compatible. The times `tc` and `tu` pass to the sampler for the control and state respectively, and `dt` (which would typically be the step in a fixed-step simulation, as the name implies) to both as sample time tolerance.

Dynamics

`Dynamics` objects represent the RHS of the dynamical system (2), i.e. the function H, together with derivatives and adjoints. Although the discrete RHS $\mathcal{H}$ is not identical to the continuum RHS H (except for the forward Euler method), $\mathcal{H}$ is always built up out of evaluations of H (or of a discretization of H, for PDEs).

The methods implementing these functions need to be called once per time step, so efficiency is a big issue. While the author may include as much sanity checking as seems reasonable, the cost can be steep for small problems. I expect that in most examples method *bodies* will be written in near-procedural fashion (in fact, `Dynamics` objects will contain the interface to Fortran or C procedures in many implementations). On the other hand the arguments are `LocalDataContainers`, so that checking compatibility between arguments is certainly possible, which would not be the case if the method *interfaces* were written in procedural style. More compatibility checking makes sense whenever the arithmetic of the RHS computation overwhelms the cost of the checks, i.e. for "large" applications.

The public interface of the `Dynamics` type is

```
virtual void rhs(LocalDataContainer & u,
 LocalDataContainer & c,
 Scalar a, Scalar t) = 0;
virtual void drhs(LocalDataContainer & du,
  LocalDataContainer & dc,
  LocalDataContainer & u,
  LocalDataContainer & c,
  Scalar a, Scalar t) = 0;
virtual void arhs(LocalDataContainer & du,
  LocalDataContainer & dc,
```

```
LocalDataContainer & u,
LocalDataContainer & c,
Scalar a, Scalar t) = 0;
```

As will be emphasized below, these are very general recursion interfaces: the only guarantees required for correct functioning within the TSOpt framework are that `drhs` implement the derivative of the update implemented in `rhs`, and `arhs` its adjoint. That is, if `rhs` implements the assignment

$$u \leftarrow F(u, c, t) \tag{16}$$

then `drhs` should implement the assignment

$$\begin{aligned} \delta u &= D_u F(u, c, t)\delta u + D_c F(u, c, t)\delta c & (17)\\ u &= F(u, c, t) & (18) \end{aligned}$$

(note that order is important here!) and `arhs` the assignment

$$\begin{aligned} \delta c &= D_c F(u, c, t)^* \delta u & (19)\\ \delta u &= D_u F(u, c, t)^* \delta u, & (20) \end{aligned}$$

One natural interpretation is that `rhs` should implement the first order update

$$u = u + aH(u, c, t) \tag{21}$$

in which H is the RHS of the state equation (2). This interpretation makes for natural implementations of various Runge-Kutta methods, including Euler. In this case, the second method implements the pair of assignments

$$\begin{aligned} \delta u &= \delta u + a[D_u H(u, c, t)\delta u + D_c H(u, c, t)\delta c] & (22)\\ u &= u + aH(u, c, t) & (23) \end{aligned}$$

i.e. the dynamics for the state and sensitivity equations are combined. Note that since the second assignment overwrites u, the ordering is important. The third method implements the "pure" adjoint computation (no reference state update)

$$\begin{aligned} \delta c &= \delta c + aD_c H(u, c, t)^* \delta u & (24)\\ \delta u &= \delta u + aD_u H(u, c, t)^* \delta u & (25) \end{aligned}$$

Note that the "+=" form of the rule for δc automatically accomodates both time-dependent and time-independent control perturbations δc.

The reference state u also needs to be stepped backwards in time during the adjoint state computation. TSOpt provides two families of methods to do this backwards step independently. The first family is completely independent of the `Dynamics` type, and implements the *optimal checkpointing* method developed by Griewank (Griewank, 1992; Griewank, 2000). No code beyond that required to implement a concrete `Dynamics` subclass is required for backwards stepping by checkpointing.

The second family of backwards steppers supposes an explicit backwards step. The mixin `TRDynamics` adds to the `Dynamics` interface the method

```
virtual void bhs(LocalDataContainer & u,
 LocalDataContainer & c,
 Scalar a, Scalar t) = 0;
```

The contract which guarantees correct functioning of `TRDynamics` is that the mapping $u \leftarrow G(u, c, t)$ implemented by `bhs` (with the last two arguments regarded as parameters) be inverse to that implemented by `rhs`:

$$F(G(u, c, t), c, t) \equiv u$$

for the values of the state u that actually appear in the time stepping loop. For example, one possible implementation of `bhs` simply stores the time history of the state, and computes the inverse or backwards step by lookup. The `step.hh` header file provides an implementation of this lookup approach in the form of the `MRDynamics` subclass of `Dynamics` and `TRDynamics`.

The lookup approach to backwards timestepping of the reference field in the adjoint computation is quite commonly used, but is of course vastly expensive in terms of storage, for large scale problems. For 3D fluid dynamics or elastodynamics simulations, the required storage typically exceeds reasonable core memory capacity of contemporary machines by several orders of magnitude, and will continue to do so for a considerable time. Therefore disk storage of the state history is imperative in this approach. The `MRDynamics` computation relies on an object of `RecordServer` type (defined in the RVL base library) to access time levels of the state history; this interface can hide disk access when it is necessary. The computational cost of disk access is relatively large. Of course the Griewank checkpointing scheme implemented in TSOpt also requires storage of part of the state history - but the number of state values which much be stored is *logarithmic* in the length of the time step loop, hence vanishingly small for large simulations, as is the amount of state recomputation required by the scheme. In general, the author expects checkpointing to be a more efficient adjoint strategy than lookup, often by orders of magnitude for large problems.

Some problems, for example in wave propagation, are *conservative* or nearly so: that is, the time step is reversible, possibly at the cost of storing an amount of data considerably smaller than the full state (eg. boundary values). In such cases a very efficient implementation of `TRDynamics::bhs` is possible. An example of this construction appears below.

On the other hand, as discussed in an Appendix, adaptive time stepping typically produces a state history with time sampling completely different from that required by the adjoint state computation. In this case, interpolation is required: the state history should be interpolated to the time samples appearing in the (adaptive) adjoint state computation by a rule of accuracy comparable to that of the time step. A concrete implementation of such an interpolated backward stepper may be added to a future release of TSOpt.

Since it may not be possible to access the initial state outside of `Dynamics`, the class provides initialization interfaces for simulation and its derivative, and a final value extraction method for adjoint computation.

```
virtual void initialize(LocalDataContainer & u,
                                LocalDataContainer & c) = 0;
virtual void initializeDer(LocalDataContainer & du,
                                LocalDataContainer & dc,
                                LocalDataContainer & u,
                                LocalDataContainer & c) = 0;
virtual void finalizeAdj(LocalDataContainer & du,
                                LocalDataContainer & dc,
                                LocalDataContainer & u,
                                LocalDataContainer & c) = 0;
```

Two specific initializations - initial state equals control and zero respectively - are common enough that TSOpt provides implementations of these methods, in `Dynamics` subclasses `DynamicsISEC` and `DynamicsISEZ` respectively. To derive a concrete `Dynamics` subclass from `DynamicsISEC` or `DynamicsISEZ`, the user need only implement the right hand side methods `rhs`, `drhs`, and `arhs`. Both of these specialized `Dynamics` classes are defined in the header file `step.H`.

A Simple `Dynamics` Example Simple test examples provided with the TSOpt package make use of the `testdyn1` subclass of `DynamicsISEC`, which corresponds to a system of uncoupled logistic equations. Its `rhs` method is displayed here in all its glory:

```
virtual void rhs(LocalDataContainer<double> & u,
 LocalDataContainer<double> & c,
 double a, double t) {
  int n = u.getSize();
  double * uptr = u.getData();
  for (int i=0;i<n;i++) {
    uptr[i] = uptr[i] + a*(1.0 - uptr[i]*uptr[i]);
  }
}
```

It is clear from this example that the target applications for TSOpt are necessarily large systems: for small systems, the virtual function call overhead will be significant.

Of course the overhead could be reduced further by passing low-level arrays, i.e. pointers, instead of `LocalDataContainer`s, and in this example nothing would be lost - except that the loop limits would have to be passed as well, cluttering up the interface and making it less general. The simpler C-style interface would not, however, cleanly accomodate grid or finite element mesh information, for example. This inability to pass auxiliary information needed in dynamics calculations in a type-safe, generic manner in Fortran or C was one of the main motivators towards OO design, at least for the this author. Of course, such auxliary information can easily be encapsulated in appropriate `LocalDataContainer`s.

The derivative and adjoint derivative methods (`drhs` and `arhs`) are coded similarly. Since the class has no private data, construction is by default, with trivial copy constructor.

Step

TSOpt encapsulates time step¡ping methods in the `Step` type. The base class is abstract; its public interface is

```
virtual void fwdStep(Model & mdl,
     Dynamics & dyn,
     Clock & clk) = 0;

virtual void derStep(Model & mdl,
     Dynamics & dyn,
     Clock & clk) = 0;

virtual void adjStep(Model & mdl,
     Dynamics & dyn,
     Clock & clk) = 0;
```

The first type in these argument lists is `Model`, which has not yet been discussed. `Model` is a concrete type, which uses a `ModelBuilder` to allocate storage for state and control as needed. [That is, the user does *not* need to implement `Model` - it is already implemented in the TSOpt package! The user only needs to supply a `ModelBuilder`, accessed through a `Statics` interface as described above.] `Model` is another *internal handle* class, similar to `RVL::Vector` and others in RVL applications, which manages the storage which it allocates on an as-needed basis, providing access by reference. The part of its public interface which concerns the user is

```
virtual LocalDataContainer & getState();
virtual LocalDataContainer & getControl();
virtual LocalDataContainer & getStatePert();
virtual LocalDataContainer & getControlPert();
```

Note that these methods are all implemented; the purpose of each is well-described by its name.

Time steps should be written independently of the internal details of the state, and of the method of computation of the RHS. This class permits you to do so.

Euler: The header file `step.H` supplies an implementation of the Euler forward method in the class `EulerStep`, the guts of which are

```
template<class Scalar>
class EulerStep: public Step {
...
  virtual void fwdStep(Model & mdl,
       Dynamics & dyn,
       Clock & clk) {
    dyn.rhs(mdl.getState(),mdl.getControl(),
      clk.getTime(),
      clk.getTimeStep());
    ...
  }
...
}
```

Code for the derivative and adjoint derivative steps is similar.

The header file `step.H` also contains an implementation of several Runge-Kutta methods. For example, the `fwdStep` method body for the "improved Euler" second order R-K method (based on the midpoint rule) is

```
dyn.rhs(mdl.getState(),mdl.getControl(),mdl.getWorkState(kind),
0.5*clk.getTimeStep(),clk.getTime());
dyn.rhs(mdl.getWorkState(kind),mdl.getControl(),mdl.getWorkState(upind),
clk.getTimeStep(),clk.getTime()+0.5*clk.getTimeStep());
cp(mdl.getState(), mdl.getWorkState(upind));
```

As for Euler, there is a "one pass" version that avoids the final copy, for use when the dependencies within the dynamics permit.

The header file `step.hh` also includes several Runge-Kutta steps, amongst other examples, mostly as demonstrations that other time-stepping rules can be implemented directly to use the same `Dynamics::rhs` methods. However the author has found that generally it is much simpler to implement other timestepping rules as formal equivalents of Euler, *via* redefinition of the `rhs` method. For example, multistep methods are conveniently implemented using formal Euler dynamics by a trick analogous to the equivalence between higher-order and first-order systems of differential equations. The section below discusses the formal Euler implementation of multistep methods, and presents an implementation of the leapfrog scheme as a detailed example.

Clock

The final ingredient in composing a TSOpt application is an object which keeps track of time. In a blinding flash of inspiration, it came to us that such a thing should be called a `Clock`. The methods useful in formulating `Step` classes are `Clock::getTime()` and `Clock::getTimeStep`, which have already appeared in the discussion of `Step` examples. Adaptive stepping methods will require in addition `Clock::setTimeStep`. The `Clock`

interface provides a large number of other functions which are of use in various parts of TSOpt, but the three just listed are likely to be the only ones needed in user code.

Constant time step clocks all work the same way, and are realized in the concrete class `ConstClock`. Its main constructor has the signature

```
ConstClock(Scalar tbeg, Scalar tend,Scalar dt);
```

the three arguments to which are the start time, end time, and time step. The time step is adjusted internally so that the time interval `tend - tbeg` is an integral multiple of `dt`.

The definition of `ConstClock` is included in the `clock.H` header file.

OPERATOR INTERFACE

The only specialized part of the public interface is the main constructor:

```
TSOp(Statics & stat,
     Dynamics & dyn,
     Step & step,
     Clock & clock,
     int _itmax=10000,
     bool _verbose=0,
     int snaps=0);
```

The types of the first four arguments were discussed in detail in the last section. An absolute limit on the length of the time step loop should always be included - this is the role of `itmax`. Verbose output (priting the time step index and some other information on `stderr` is sometimes useful; `verbose` is the switch for this output option. Finally, the checkpointing scheme for adjoint computation requires that the number `snaps` of checkpoints be specified.

In other ways the `TSOOp` functions exactly as is any other `RVL::Operator` class, i.e. primarily through formation of `OperatorEvaluation` objects. These objects pair an `Operator` with a `Vector`, and provide access to the value of the operator at the vector, the value of its derivative, etc., i.e. its *jet* at a point in its domain. RVL provides no access to these values through the `Operator` class itself. They are accessible only an `OperatorEvaluation` object, which guarantees the consistency of the various derivatives (i.e. they are all evaluated at the same point) and provides shared workspace for intermediate results. See (Gockenbach et al., 1999; Symes et al., 2005) for further discussion of the evaluation object concept, which was pioneered by HCL.

EXAMPLES

A simple ODE example: The first example combines the simple `Rn...` classes of simulator components explained in the preceding paragraphs to solve a system of logistics equations. The construction of the TSOpt `Operator` object is straightforward:

```
RnStatics<double> stat(nu,nc,tbeg,tend,dt); // Statics
testdyn1 dyn;  // Dynamics
EulerStep<double> step; // Step
ConstClock<double> clock(tbeg,tend,dt); // Clock
TSOp<double> op(stat,dyn,step,clock); // Operator
```

Note the presence of the template argument; as mentioned earlier, all of the TSOpt types are actually class templates, and work (at least) for the standard floating point arithmetic types `float` and `double` and for the `std::complex` types templated on these.

This code is preceded by code defining the number of state variables `nu`, the number of control variables `nc` (= `nu` in this example), the start and end times of the simulation `tbeg` and `tend`, and the time step `dt`.

A simple application of this construction is a check of the correctness of the derivative code. Since no adaptivity is involved in this example, the discrete derivative should actually be the derivative of the discrete simulation. This check is implemented in the base class, in the `Operator::checkDeriv` method. A point in the domain and a tangent vector are required. In the example source `testop.C` (in the `testsrc` directory), these are conveniently constructed using standard RVL devices for setting and randomizing vector components:

```
Vector<double> c(op.getDomain()); // point in domain ( = entire vector space)
Vector<double> dc(op.getDomain()); // tangent vector
RVLAssignConst<double> ac(0.5) // set components = 0.5
RVLRandomize<double> rnd; // random initialization
c.eval.(ac);
dc.eval(rnd);

op.checkDeriv(c,dc,cout);
```

The function of `checkDeriv` is to predict the directional derivative of `op` at `c` in the direction `dc` using the derivative (linear) operator produced by the evaluation at `c`, then compare with centered divided differences for a variety of steps. The number and value of steps is optionally controlled by overriding defaults in the call to `checkDeriv`; see the documentation of `Operator` for details. We have used the defaults in this example.

The output of `checkDeriv` (in this case on the standard output stream `cout`) includes an estimate for the rate of convergence of the centered divided differences to the predicted directional derivative, displayed as the last column of output. If all is well, the rate should approach 2. For the example constructed here, a typical run yields

```
Operator::checkDeriv

  h          norm of diff.              rel. error           convg. rate
  1   0.005644841897628637     0.04080948932243057       ---------------
```

```
0.9   0.004534169622781486     0.03277986348647303     2.079528803190778
0.8   0.003556026416183277     0.02570835901045809     2.063100301946764
0.7    0.00270492266847007     0.01955528866154824      2.04873488807581
0.6   0.001976187466984309     0.01428688398994783     2.036344290497774
0.5   0.001365898694310513    0.009874789974968731     2.025854085324662
0.4  0.0008708259677460666    0.006295652504874876     2.017202482258696
0.3  0.0004883847729693176    0.003530786785383984     2.010339331108153
0.2  0.0002166005087637713    0.001565917400333482     2.005225270160088
0.1  5.408146312150648e-05   0.0003909829419183031     2.001830543660448
```

which suggests that the derivative was constructed correctly.

[SHOW checkAdj output which demonstrates checkpointing]

[SHOW inversion run using BFGS]

An Acoustic Finite Difference Application

[brief description of `fdgrid` and `a2c` packages]

MULTISTEP METHODS, IMPLICIT SCHEMES, and IMPLICIT EQUATIONS

The description of TSOpt supplied so far suffices to guide the construction of the simplest instance only. Several common complications not discussed so far are multistep and implicit timestepping rules and implicit equations. The key to handling all of these complications within the TSOpt framework is the realization that the `Dynamics` methods provide interfaces behind which to hide very general recursive update formula. While the forward Euler (and more generally Runge-Kutta) steps have obvious implementations, `Dynamics::rhs` can be reinterpreted in a variety of ways which accommodate much more general timestepping rules.

Multistep Methods:

The TSOpt implementations of multistep methods rely on the facilities of subclasses `MSModelBuilder` and `MSModel` of `ModelBuilder` and `Model` respectively to provide access to implicit `MultiStepLDC` product structures superimposed on the `LocalDataContainer` classes. The subvectors defined by the `MSModel` classes come with compatibility contracts, so obviate the need for compatibility checking in the multistep constructions which use them.

Multistep methods are commonly implemented via a cyclic storage approach. That is, an m-step method stores successive time levels $\{u_{n-m+1}, ..., u_n\}$ along with a set of pointers to the time levels. The step overwrites u_{n+1} on the storage for u_{n-m+1}, and cyclically permutes the pointer set. A Newton divided difference organization of multistep methods is also possible. In this approach the time levels of the solution are expressed implicitly via divided differences. This approach avoids having to keep track of a set of

pointers into the data; the current time level always resides in the same storage. The price is a small amount of additional arithmetic and some loads and stores.

The `MSModel` construction enables coding of multstep schemes via invocation of the formal Euler step `EulerStep`, provided that the meaning of `Dynamics::rhs` is redefined appropriately. For example, leapfrog requires that `rhs` express the system version of the two step scheme:

$$\begin{aligned} u_0 &= u_1 + 2\Delta t H(u_0, c, t) \\ t &= t + \Delta t \end{aligned}$$

after which the indices 0 and 1 are switched. A convenient implementation constructs a `Dynamics::rhs` subclass with the behaviour, for example

```
void rhs(LocalDataContainer & u,
                LocalDataContainer & c,
                Scalar twodt, Scalar t) {
  try {
    MultiStepLDC & mu =
      dynamic_cast<MultiStepLDC &>(u);
    dyn.rhs(mu[0],c,zero,t);
    lc.setScalar(one,twodt);
    lc(mu[0],mu[1]);
    mu.fwdCycle();
  }
  catch (bad_cast) {...}
  ...
}
```

Here `dyn` is yet another `Dynamics` subclass whose `rhs` method overwrites u with $H(u, c, t)$ (and effectively ignores its third argument), and `lc` is an instance of `RVLlinCombObject`, a part of the RVL local linear algebra package which performs linear combination. If `lc.setScalar(a,b)` is called before `lc(u,v)`, the result is to overwrite `u` with the data of `b` times the data array of `u` plus `a` times the data array of `v`. In order to save a flop, `twodt` has value $2 * \Delta t$. and `zero` and `one` are the appropriate `Scalar`s. Finally, `mu.fwdCycle` uses the *indirect indexing* facility of `MultiStepLDC` to implicitly permute the time levels of the state vector.

This formulation is appropriate that the double time step `twodt` might be adaptive. If the time step is fixed, the setup of the linear combination can be hoisted into the the constructor of the `Dynamics` object, hence out of the loop.

For general linear multistep methods, u_0 will be a linear combination of steps and possibly RHS evaluations. An obvious extension of the above construction, with more complex auxiliary `Dynamics` classes.

The formulation of a multistep method also presumes a multistep state construction. This is provided by the `MultiStepModel` class, which is a collects together `MultiStepLDC`s,

local data containers with a dynamic indexing operators permitting cyclic index permutation, hence storage of new steps over old, no-longer-needed steps with no data motion. These are in turn built out of `ProductLocalDataContainer`, a core RVL class which provides a local data container with a Cartesian product structure. The `MultiStepModelBuilder` object which the user must develop (or borrow) to implement a multistep method in TSOpt returns dynamically allocated `ProductLocalDataContainer`s. One additional complication in devising an appropriate `Statics` class to enable multistep treatment of a problem is the necessity of devising an appropriate `ProductLocalDataContainer` type. The example section describes one such construction.

Implicit Methods and Equations

TSOpt also accomodates implicit methods for both implicit and explicit DEs. Consider for example the general implicit differential equation

$$F\left(\frac{du}{dt}, u, c, t\right) = 0$$

It should not be surprising that the TSOpt approach to such problems uses a `Dynamics` method for the vector $(u, du/dt)^T$, implemented as a `MultiStepLDC`.

For example, one version of the Crank-Nicholson scheme (which boils down to the trapezoidal rule for explicit equations) for this system is

$$F\left(\left(\frac{u^{n+1}-u^n}{\Delta t}\right), u^{n+1}, t^{n+1}\right) + F\left(\left(\frac{u^{n+1}-u^n}{\Delta t}\right), u^n, t^n\right) = 0$$

Applying Newton's method to this system, along with another approximation of the same order as already made, gives an iteration for a sequence $\{u_\nu^{n+1} = u_{\nu-1}^{n+1} + \Delta t \delta u_\nu^n\}$:

$$\begin{aligned}\frac{1}{2}\Bigg[&D_{u'}F\left(\frac{u_{\nu-1}^{n+1}-u^n}{\Delta t}, u_{\nu-1}^{n+1}, t^{n+1}\right) + D_{u'}F\left(\frac{u_{\nu-1}^{n+1}-u^n}{\Delta t}, u^n, t^n\right)\\ &+\Delta t D_u F\left(\frac{u_{\nu-1}^{n+1}-u^n}{\Delta t}, u_{\nu-1}^{n+1}, t^{n+1}\right)\Bigg]\delta u_\nu^n\\ = -\frac{1}{2}&\left[F\left(\frac{u_{\nu-1}^{n+1}-u^n}{\Delta t}, u_{\nu-1}^{n+1}, t^{n+1}\right) + F\left(\frac{u_{\nu-1}^{n+1}-u^n}{\Delta t}, u^n, t^n\right)\right] \qquad (26)\end{aligned}$$

The TSOpt approach to this system requlres an auxiliary `Dynamics::rhs` method implements the accumulation

$$v = aF(v, u, t)$$

while storing the pair (u, v) as the two components of a `MultiStepLDC` (that is, the first component is left unaltered, the second is updated as indicated).

Call this auxiliary `Dynamics` object `dyn`, as before. Computation of the RHS of the Crank-Nicholson step (26) requires two additional `MultiStepLDC` workspace objects `u0`,

`u1`. The correspondence is: u^n is stored in `mu0 [0]`, `mu[0]` stores the current iterate for the next time step $u_{\nu-1}^{n+1}$, and `mu0[1]` stores the derivative estimate $(u_{\nu-1}^{n+1} - u^n)/\Delta t$ - this occurs at the end of the first block of code below. At the end of the second block, `mu1[1]` stores the first summand on the RHS of (26). After the third block, `mu1[1]` stores the second summand, and `mu[1]` the first. After the final block, `mu[1]` stores the RHS of (26):

```
MultiStepLDC & mu =
  dynamic_cast<MultiStepLDC &>(u);
MultiStepLDC & mu0 =
  dynamic_cast<MultiStepLDC &>(u0);
MultiStepLDC & mu1 =
  dynamic_cast<MultiStepLDC &>(u1);
...
lc.setScalar(dtr,-dtr);
...
cp(mu0[1],mu0[0]);
lc(mu0[1],mu[0])

cp(mu1[0],mu[0]);
cp(mu1[1],mu0[0]);
lc(mu1[1],mu[0])
dyn.rhs(u1,c,one,t);

cp(mu[1],mu1[1]);
cp(u1,u0);
dyn.rhs(u1,c,one,t);

ls.setScalar(half,half);
lc(mu[1],mu1[1]);
```

The additional interface required by the Newton-Crank-Nicholson scheme is a linear solver. It is conventional to use a *frozen Newton* approach to the LHS of (26) in which the second argument of the partial derivatives is held fixed at the previous state value u^n, and t^{n+1} is replaced by t^n. With these replacements, the operator on the left hand side of (26) becomes

$$D_{u'}F\left(\frac{u_{\nu-1}^{n+1} - u^n}{\Delta t}, u^n, t^n\right) + \frac{\Delta t}{2} D_u F\left(\frac{u_{\nu-1}^{n+1} - u^n}{\Delta t}, u^n, t^n\right) \tag{27}$$

which is of the form

$$D_{u'}F(u', u, t) + aD_u F(u', u, t)$$

This computation is represented by the `Dynamics::drhssol` interface:

```
virtual void drhssol(LocalDataContainer & u0,
                     LocalDataContainer & b,
                     LocalDataContainer & c,
                     Scalar a, Scalar t) = 0
```

On call `u0` is a multivector containing the velocity and state arguments of the operators in (27), `b` is the RHS in this equation, and `c` is the control as usual. Note that, at the end of the block of code sketched above, `u0` stores exactly the right quantities, `b` is `mu[1]`.

An important example of this construction, which arises for example in finite element semi-discretization of parabolic evolution equations, is

$$F(u', u, c, t) = Mu' - K(u, c, t)$$

in which the *mass matrix* M is independent of t. The usual Crank-Nicholson scheme for this system is

$$M\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{2}(K(u^{n+1}, c, t^{n+1}) + K(u^n, c, t^n))$$

The "frozen Newton" system for this implicit scheme is

$$\left[M - \frac{\Delta t}{2} D_u K(u^n, c, t^n)\right] \delta u_\nu^n = -M(u_{\nu-1}^{n+1} - u^n) + \frac{\Delta t}{2}\left[K(u_{\nu-1}^{n+1}, c, t^{n+1}) + K(u^n, c, t)\right]$$

which is exactly the scheme (26) with the frozen operator replacement (27) for this case. An implementation of `Dynamics::drhssol` should solve systems with matrix

$$M + \alpha D_u K(u, c, t)$$

and the right hand side is assembled via the code displayed above.

Computation of derivatives and adjoints follows the same pattern as for explicit schemes and explicit ODEs, and we omit the derivations and results. Note that in general it is not practical, or necessary, to enforce a strict derivative relationship between the computations implemented by `Dynamics::rhs` and `Dynamics::drhs`. For example, it is much simpler, and perfectly adequate, to apply the Crank-Nicholson method directly to the sensitivity equations (8), even though the resulting linear map differs from the derivative of map implemented under `Dynamics::rhs` by a discretization error.

Testing

The validity contract goal mentioned in the introductory section should guarantee that this test will succeed, provided that the inputs to the `TSOp` constructor pass their own validity tests. The tests implemented in TSOp are:

- check that the forward and adjoint samplers provided by the `Statics` object are in fact an adjoint pair;

- check that the `Dynamics` object defines a consistent 1-jet, i.e. a function together with accurate approximations to its derivatives (this test will be extended to the 2-jet when we get around to adding Hessians);

- check that `Step::fwdStep`, `Step::derStep`, and `Step::adjStep` stand in the appropriate relations.

Since we expect `Step` child classes such as `EulerStep` to be reused over many applications, the third test will typically not be necessary in constructing a particular application. The first test pertains to the data structures used in the internal and external representations of state and control, which will typically also amortize over many applications. So we expect quality assurance for most TSOpt applications will focus on validity of the `Dynamics` object code. In effect, if the right hand side, hence a single step, is coded properly, then the entire construction will work. This test is quite simple in the absence of adaptivity: an auxiliary class defined in TSOpt wraps the `Dynamics` object in an appropriate `Operator` interface, for which the RVL base classes provide an appropriate set of tests (like `Operator::checkDeriv`).

Adaptive methods can only be checked in fixed discretization mode, which complicates the writing of wrapper classes for testing purposes.

Discussion

[mention role of procedural code in implementing `rhs` instances]

Acknowledgements

This work was supported in part by National Science Foundation, by the sponsors of The Rice Inversion Project (TRIP), and by grants from ExxonMobil Upstream Research Co. The author is grateful to Anthony Padula, Hala Dajani, Eric Dussaud, Shannon Scott, Mark Gockenbach, Matthias Heinkenschloss, Denis Ridzal, Michael Heroux, Roscoe Bartlett, Amr el-Bakry, Michael Gertz, and Adam Singer for many insights and discussions on the topic of this report and related topics.

REFERENCES

Benson, S., McInnes, L. C., and Moré, J. (2000). TAO: Toolkit for advanced optimization. Technical report, Argonne National Laboratory, `www-fp.mcs.anl.gov/tao/`.

Deng, L., Gouveia, W., and Scales, J. (1996). The CWP object-oriented optimization library. *The Leading Edge*, 15(5):365–369.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, New York.

Gockenbach, M., Reynolds, D., Shen, P., and Symes, W. (2002). Efficient and automatic implementation of the adjoint state method. *ACM Transactions on Mathematical Software*, 28:22–44.

Gockenbach, M., Reynolds, D., and Symes, W. (2001). Automatic differentiation and the adjoint state method. In Corliss, G., editor, *AD2000: from Simulation to Optimization*, New York. Springer Verlag.

Gockenbach, M. and Symes, W. (2002). Adaptive simulation, the adjoint state method, and optimization. In Heinkenschloss, M., editor, *First Sandia Workshop on Large Scale PDE-constrained Optimization*, New York. Springer Verlag.

Gockenbach, M. S., Petro, M. J., and Symes, W. W. (1999). C++ classes for linking optimization with complex simulations. *ACM Transactions on Mathematical Software*, 25:191–212.

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

Griewank, A. (2000). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics (Frontiers in Applied Mathematics 19), Philadelphia.

Martin, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall.

Meza, J. (1994). OPT++: An object-oriented class library for nonlinear optimization. Technical Report 94–8225, Sandia National Laboratories, Sandia National Laboratories, Livermore, CA.

Pacheco, P. (1997). *Parallel Programming with MPI*. Morgan Kaufmann, San Francisco.

Scott, S. D. (2001). Software components for simulation and optimization. Technical Report 01-06, Department of Computational and Applied Mathematics, Rice University, `www.caam.rice.edu/caam/caam-techrep.html`. (MA Thesis).

Symes, W. W., Padula, A. D., and Scott, S. D. (2005). A software framework for the abstract expression of coordinate-free linear algebra and optimization algorithms. Technical Report 05-12, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, USA.