A Time-Stepping Library for Simulation-Driven Optimization

*William W. Symes, Anthony D, Padula, Hala Dajani, Eric Dussaud**

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 Standard 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.

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

For time-dependent problems, the (discretized) system of differential equations is block triangular (explicit schemes) or can be regarded as formally block triangular (implicit 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 (**?**; **?**; **?**; **?**). TSOpt packages the simulator and its various derivatives in a single object, and provides a natural interface to optimization algorithms via the *Standard Vector Library*.

The Standard Vector Library

This library ("SVL" in the sequel) is the software framework for TSOpt . SVL 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 SVL 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 SVL, see (**?**).

SVL 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 `SVL::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, and TSOpt does not encompass this aspect of the problem formulation. SVL provides a number of tools for creating functions of vectors. These tools produce the required derivatives as well, and the resulting `SVL::Functional` objects interface directly with optimization algorithms implemented in SVL. 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. TSOpt merely produces the vector output of the simulator from which the cost function can be computed via the construction of an `SVL::Functional`. That is, TSOpt presents the simulator as a vector valued function of the control (vector). SVL realizes such vector valued functions in its `Operator` interface. `SVL::Operator`s combine functions and their derivatives in one object with persistent state, which permits the sharing of intermediate data necessary for efficiency in many computations.

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 `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 SVL 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 inputs to the `Operator` constructor 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 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 (REF to FDTD paper) 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 (REFS). 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 `Operator` 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 (REFS TO GRIEWANK). 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 `SVL::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), and these may be treated as independent data structures combined in a Cartesian product via the `SVL::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 experiments. 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 `Operator` by means of `SVL::FunctionObject`s, which is the next and final design goal.

- implement using `FunctionObject` interface

`SVL::FunctionObject` is the interface for virtually all interaction with data in SVL applications. `SVL::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 `SVL::FunctionObjects`. To interact with the data of an `SVL::Vector` in virtually any way, it is necessary to write a `SVL::FunctionObject` to do the actual interacting. The innards of the TSOpt `Operator` class are like any other SVL application in this respect.

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

However there is a deeper reason for this design decision. SVL was designed with component oriented programming in mind: that is, construction of applications out of *components*, separate processes likely running on distinct platforms. In particular, components involving only part of the core SVL interface set may include arbitrary software features which are accessible to the rest of SVL through a component framework without needing to pervade SVL. `FunctionObject` and `DataContainer` are the only two class subtrees in SVL intended to interact directly with the communications layer in a component framework (REF HALA'S THESIS, TONY'S REPORT). It is possible to design a parallel `SVL::DataContainer` using MPI (REF), for example, and include it in a component process. Then SVL applications can take advantage of SPMD parallelism, for example, without requiring any MPI "awareness" on the part of the "naturally serial" SVL calculus classes (`Vector`, `Space`, `Functional`,...), *or in any algorithms formulated in terms of the calculus classes*, via a component framework and appropriately structured `FunctionObject`s. Since parallelism will be critically important to many TSOpt applications, implementing TSOpt objects in terms of `FunctionObject`s seems completely natural.

Relation to the FDTD Package

TSOpt is a direct descendent of the *FDTD* package described in (REF), which relied on a predecessor library of SVL, the *Hilbert Class Library* ("HCL"), for its basic constructs. 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 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. Therefore access to paralellism through component (client-server) design (or in any other way, for that matter) required that virtually *all* HCL classes be modified to optionally interact with a communications layer (REF SHANNONS THESIS). In contrast, as noted above this interaction is confined to a few classes in SVL (REF HALA'S THESIS).

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 $0 \leq t \leq T$. 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:

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

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.

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

whence

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

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

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

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

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

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

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

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

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: The SVL `FunctionObject` classes are divided by number of inputs: unary, binary,... Unary `FunctionObjects` have an evaluation method with one argument, for example. However both the input (control) and output (data) of a TSOpt application may have arbitrary numbers of components. Each component is realized as a `DataContainer` object. The components may themselves divide into components, and this hierarchy may in principle have arbitrary depth. The atomic or indivisible `DataContainer` in SVL is the `LocalDataContainer`, which provides a simple interface to the actual data (access methods for the length and pointer of a scalar array). The inputs of a `FunctionObject` are `LocalDataContainers`. So the TSOpt `FunctionObject` interface must accommodate arbitrary numbers of input `LocalDataContainer`s, with the actual number determined at run time.

No modification of the evaluation method signature conveniently accommodates arbitrary, run-time determined numbers of arguments. The only available alternative in C++ is to evaluate the `FunctionObject` many times, and use the persistent state property of `FunctionObjects` to retain input information between evaluations. Therefore the top-level internal interface of TSOpt consists of *unary* `FunctionObjects` `TSApplyFO`, `TSApplyDerFO`, `TSApplyAdjFO`,... which implement the simulation, its derivative, its adjoint derivative,...

The `Operator` interface for TSOpt is concrete (fully implemented). Its critical methods evaluate appropriate `FunctionObject`s on its input and output vectors:

```
void apply(Vector & x, Vector & y) {
  ...
  TSApplyFO f(...);
```

```
  x.eval(f);
  y.eval(f);
}

void applyDer(Vector & x, Vector & dx, Vector & dy) {
  ...
  TSApplyDerFO df(...);
  x.eval(df);
  dx.eval(df);
  dy.eval(df);
}
...
```

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

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 `TSApply...::operator()` method body. Sanity checking of inputs must be handled largely at runtime in this method.

The arguments of the `TSApply...` constructors are the described in the next section, and are common to all applications.

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 the `Operator` interface of TSOpt 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 (and a sequence of other simulation parameters) within the scope of each `TSApply...` method. As is evident in the code fragments presented above, each component of the input (control) and output (state, data) is visited precisely once, and this visit (the `eval` method) is the only communication with the `TSApply...` objects. That is, each simulation must have its additional parameters conveyed *by visiting the output data components* corresponding to that simulation. Also, since each component is visited exactly once, the first output component of every simulation must carry all auxiliary information necessary to initialize that simulation.

The inclusion of 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 contain within it enough information to interpret it properly, so represents inadequate data encapsulation. Seismic reflection is an example of a technology in which it has been normal for decades to structure data so that all information necessary for interpretation and processing is integral to the data structure. 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 discussed in the paragraphs to follow encompass these four functions.

Statics

The static aspect of the simulation needs to define the external representation of state and control, their internal representations, and mechanisms for translation between these representations. To offer a compatibility guarantee, TSOpt presents these items 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.

The static or time-independent description of the system defines the state and control data. Various components of TSOpt use these definitions, which implicitly also determine the types of state and control perturbations which appear in the derivative and adjoint

derivative computations, and possibly workspace for step computations and checkpoints for adjoint state loops. The static description also entails the relation between the *internal* objects just mentioned and their *external* or archival representation - for simulators of more than minimal complexity, the form in which the data of the control is supplied to the simulation will not be the same as that required by the time stepping computations. For example, PDE problems require spatial discretization, and the meshes used in the computation may be computed as a function of the data, or even adapted during the course of the simulation, and therefore cannot be externally specified.

Similarly, control and inverse problem formulations seldom require the entire time history of the state, sampled precisely at the times used in the simulation. Instead, such applications usually require samples at times determined by the process using the simulator (the final time, or uniformly sampled times when the internal time step is adaptively chosen and nonuniform, or...). Moreover commonly only a function of the state is required - for example, one coordinate of a multicomponent state, or sampling of the state along a spatial surface for a PDE, or...

The uses of internal and external representations are quite different. The external representation naturally interacts with the `Operator` interface which the simulator presents to its user applications. `Space`s are therefore the natural specification of the external state and control types. [Consistent with their appearance in inverse problems, I will use *data* as an alias for the externalized aspect of the state.]

Since the user application will not allocate internal state and control objects, the `Statics` object must provide a means to do so. Since these internal details of the simulation will not be needed, or even visible, outside the simulator, the `Statics` object exposes factory objects which return dynamically allocated state and control objects on demand. In the current version of TSOpt, these `ModelBuilder` objects have `build...` methods which return control of `LocalDataContainer` instances. 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. Since the internal data takes `LocalDataContainer` form, a `FunctionObject` interface is natural for the state and control sampling operators.

The mandate to accommodate *multisimulations* brings up a subtle point: the sampler `FunctionObject`s themselves are not the natural attributes of a `Statics` object - instead, the ability to produce a sampler `FunctionObject` on demand, initialized by information passed as part of the external control and/or data state, is the natural attribute. Accordingly, the `Statics` interface provides access to `SamplerFactory` objects; these have appropriate initialization interfaces.

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

```
/** return reference to control space (external representation of control data) */
virtual Space<Scalar> & getControlSpace() = 0;
/** return reference to state space (external representation of state data) */
virtual Space<Scalar> & getDataSpace() = 0;
```

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

The `Space` interface is part of core SVL and is described elsewhere [REF!].

ModelBuilder is a factory with two products:

```
/** dynamically allocate control LDC */
virtual LocalDataContainer<Scalar> * buildControl() = 0;
/** dynamically allocate state LDC */
virtual LocalDataContainer<Scalar> * 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 auxiliary information is useful.

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 `RnArray`s (the SVL 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:

```
/** use auxliary data of LDC to initialize internal data..*/
virtual void initialize(LocalDataContainer<Scalar> & d) = 0;
/** dynamically allocate forward sampler FO  */
virtual FwdSampler<Scalar> * buildFwd() = 0;
/** dynamically allocate adjoint sampler FO */
virtual AdjSampler<Scalar> * buildAdj() = 0;
```

`SamplerFactory::initialize` is an opportunity to pass any data not proper to the SamplerFactory itself, i.e. to all component simulations of a multisimulation. It is of course moot (and can be implemented as a no-op) when only one simulation is to be performed.

Since the simulation function object (see below) visits the data of each simulation only once, it is essential that any auxiliary information needed to (re)initialize a `SamplerFactory` be present in the *first* component of the data of each simulation. Note that this is not an issue if the data for each simulation has only one component, as is often the case.

`FwdSampler` and `AdjSampler` are subclasses of `UnaryFunctionObject`, so inherit its evaluation interface. The "direction" implicit in the names is internal → external: thus, `FwdSampler` objects map internal data to external data, and `AdjSampler` objects do the opposite, as in equations (12) and (14). 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 SVL 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 for example testing of the adjoint relationship (built into `LinearOp`) immediately available.

The complete public interface of the base `FwdSampler` class is

```
/** Provides access to sample time */
virtual void setTime(Scalar t) = 0;
/** number of LDCs onto which sample data is to be written */
virtual int getNumberOfData() = 0;
/** write method - output of data onto (archival) LDCs,
    index 0 <= i < getNumberOfData() */
virtual void save(LocalDataContainer<Scalar> & d, int i) = 0;
/** (inherited from UnaryFunctionObject) */
virtual void operator()(LocalDataContainer<Scalar> & x) = 0;
```

The interface for `AdjSampler` differs only in having a `load` method in place of `save`, with otherwise the same signature.

Sampling is typically time-dependent - 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 `FwdSampler` subclass). The `setTime` method permits a calling unit to pass the time to the `FwdSampler`, typically by copying the argument into a `Scalar` buffer.

The *external* representation of the state may be a product vector, whose data container decomposes (eventually) into some number of local data containers. This information about structure needs to be available for proper functioning of the sampler users (other components of TSOpt). Providing this count is the role of `getNumberOfData`. For simple simulations, in which both control and data are packaged as `LocalDataContainer`s externally (i.e. as `LocalVector`s), the return value of `getNumberOfData` is 1.

[In principle this information is implicit in the `Space` manifestation of the external representations, so should be extracted from there, rather than inserted in the sampler classes by hand. A modification for the future...].

A typical implementation of these classes will create a buffer in which to store either the internal representation, or the external representation, or something intermediate, as is convenient. For example, time series sampling would naturally be accomplished by storing the time samples as they are created, at the sample rate chosen internally by the simulator, then when the simulation is done interpolating onto the externally specified time grid. The `save` method is the locus of this final step: its argument `i` is the index of a `LocalDataContainer` component, referenced by the argument `x`, in the external representation. A call to `save` causes the proper data from the buffer to be written to `x`. This step may involve simple copying, interpolating, or more complex (linear) operations.

The `load` method of an `AdjSampler` works analogously.

Finally, the `operator()` evaluation method performs the dual operation: for `FwdSampler`s, it samples its argument `LocalDataContainer`, which represents the internal state, into the buffer. For a time snapshot, this might well simply copy the internal state into a replica of itself. The design of the buffer, and the partition of work between `operator()` and `save(...)`, will depend on efficiency considerations for each sampler design.

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 takes a dimension specification, a sample time, and a tolerance for fitting the sample time. The sampler objects own a buffer of the specified size. 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::save` method copies the buffer onto its argument. The `RnSnapAdjSampler` methods do exactly the opposite copies. The `getNumberOfData()` method returns 1.

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 `RnSpace`s for domain and range, an `RnModelBuilder`, and `RnSnapSamplerFactory`s 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<Scalar> & u,
 LocalDataContainer<Scalar> & c,
 LocalDataContainer<Scalar> & up,
 Scalar a, Scalar t) = 0;

virtual void drhs(LocalDataContainer<Scalar> & du,
  LocalDataContainer<Scalar> & dc,
  LocalDataContainer<Scalar> & dup,
  LocalDataContainer<Scalar> & u,
  LocalDataContainer<Scalar> & c,
  LocalDataContainer<Scalar> & up,
  Scalar a, Scalar t) = 0;

virtual void arhs(LocalDataContainer<Scalar> & du,
  LocalDataContainer<Scalar> & dc,
  LocalDataContainer<Scalar> & dup,
```

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

  virtual void reset(LocalDataContainer<Scalar> & u,
              LocalDataContainer<Scalar> & c) = 0;
```

The first method implements

$$u_p = u + aH(u, c, t)$$

The inclusion of the (arbitrary) scale factor a permits some economies of storage and data access in implementations of many common time steps, as will be discussed below.

As explained in the section on mathematical formulation, the RHS function H includes the initial data for the problem as well. The initial data may depend on, or even be, the control. In this computational interface for computing the RHS, initialization requires recognition that the time is the initial time to within a tolerance. Both intiial time and tolerance must also be supplied as obect data for the `Dynamics` subclass. In view of the many different possible approaches to coding the initialization, the code is left to the user.

The second method implements

$$\begin{aligned} u_p &= u + aH(u, c, t) && (16) \\ \delta u_p &= \delta u + a[D_u H(u, c, t)\delta u + D_c H(u, c, t)\delta c] && (17) \end{aligned}$$

i.e. the dynamics for the state and sensitivity equations are combined. This structure permits a useful amount of loop fusion, and corresponds to the natural mode of several Automatic Differentiation packages (REF).

The third method implements the "pure" adjoint computation (no reference state update)

$$\begin{aligned} \delta u &= \delta u_p + aD_u H(u, c, t)^* \delta u_p && (18) \\ \delta c &= \delta c + aD_c H(u, c, t)^* \delta u_p && (19) \end{aligned}$$

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, but TSOpt provides methods to do this independently of the `Dynamics` type. These *checkpointing* methods, of which we implement one due to Griewank (REF), require that the initial data be available. Since there is no guarantee that `u` and `up` in the arguments of `Dynamics::rhs` necessarily refer to different objects, it may not be possible to access the initial state outside of `Dynamics` with the aid of `rhs` alone. Therefore we provide a `reset` method which does this initialization. We have also provided (abstract) subclasses `DynamicsISEC` and `DynamicsISEZ` which give simple implementations of `reset` when initial state equals control and zero respectively.

A Simple `Dynamics` Example Simple test examples provided with the TSOpt package make use of the `testdyn1` dynamics class, 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,
   LocalDataContainer<double> & up,
   double a, double t) {
    int n = u.getSize();
    double * uc = u.getData();
    if (abs(tinit-t) < tol) {
      for (int i=0;i<n;i++) {
uc[i] = c.getData()[i];
up.getData()[i] = uc[i] + a*(1.0 - uc[i]*uc[i]);
      }
    }
    else {
      for (int i=0;i<n;i++) {
up.getData()[i] = uc[i] + a*(1.0 - uc[i]*uc[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. Also note the absence of sanity checking: as would be the case with raw Fortran or C arrays, we simply presume that the loop indices always make sense. If this makes you nervous, you could add

```
    if (n != up.getSize()) {
       // throw exception
       ...
    }
```

after the first line of the method body, and if `n` is large the additional cost would be negligible.

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 first

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.

The `reset` is defined in the `DynamicsISEC`, which stands for "Dynamics with Initial State Equals Control". This definition of initialization - that the control is simply the initial state of the system - is useful enough that we have encapsulated it in an abstract subclass, which implements the initialization methods. To derive a concrete `Dynamics` subclass from `DynamicsISEC`, the user need only implement the right hand side method and its derivative and adjoint derivative. We have also supplied a class `DynamicsISEZ`, for problems in which the initial state is the zero vector. Both of these specialized `Dynamics` classes are defined in the header file `step.H`.

Multistep Methods, Implicit Equations: the Extended Interface Multistep methods require more extensive combinations of prior and current state data than is conveniently available through the `Dynamics` interface described above. While implementation of multistep methods using `Dynamics::rhse` and derivatives is possible, it requires temporary storage and multiple passes through data. This is not an obstacle for small systems, and indeed all procedural packages for ODE solution are implemented in essentially this way (REFS: DIFSUB, DVODE,...). However for large systems the penalty is unacceptable.

The extended interface `Dynamics::rhse` implements the combination

$$u_p = b_0 u_0 + bu + aH(u, c, t) \tag{20}$$

Here u_0 is another state vector. The signature of this method is

```
virtual void rhse(LocalDataContainer<Scalar> & u0,
                  LocalDataContainer<Scalar> & u,
                  LocalDataContainer<Scalar> & c,
                  LocalDataContainer<Scalar> & up,
                  Scalar a, Scalar b0, Scalar b, Scalar t)
```

For linear multistep methods, u_0 will be a linear combination of steps and possibly RHS evaluations. Simple examples of two-step methods are leapfrog ($b_0 = 1$, $b = 0$, $a = 2\Delta t$) and leapfrog for second order systems ($b_0 = -1$, $b = 2$, and $a = \Delta t^2$), in both cases u_0 being the state at the previous time step. Higher order multistep methods require a preliminary calculation to store a combination of state and RHS data in u_0, and therefore slightly more data access than a low-level procedural implementation. A properly constructed "helper" function object can store the appropriate combination of data into u_0 in a single pass.

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 SVL 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.

The interfaces `Dynamics::drhse` and `Dynamics::arhse` give the derivative and adjoint derivative of the output $[u_p, u]$ in terms of the input $[u, u_0, c]$. Note that for proper linking to AD software it is necessary to treat u as both input and output, even though it is left unchanged by the computation (20). The signatures are appropriate generalizations of those for `Dynamics::rhse`.

The extended interface, with one further additional method, also enables the formulation of implicit methods for both implicit and explicit DEs. Consider for example the general implicit differential equation

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

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

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

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

$$\begin{aligned}\Bigg[D_{u'}F\left(\frac{u_{\nu-1}^{n+1}-u^n}{\Delta t}, u_{\nu-1}^{n+1}, t^{n+1}\right) + D_{u'}F\left(\frac{u_{\nu-1}^{n+1}-u^n}{\Delta t}, u^n, t^n\right) \\ +\Delta t D_u F\left(\frac{u_{\nu-1}^{n+1}-u^n}{\Delta t}, u_{\nu-1}^{n+1}, t^{n+1}\right)\Bigg]\delta u_\nu^n \\ = -\left[F\left(\frac{u_{\nu-1}^{n+1}-u^n}{\Delta t}, u_{\nu-1}^{n+1}, t^{n+1}\right) + F\left(\frac{u_{\nu-1}^{n+1}-u^n}{\Delta t}, u^n, t^n\right)\right] \qquad (21)\end{aligned}$$

The RHS of this system is already available for explicit systems, in the extended interface: indeed, in that case the residual, of which the RHS of (21) is a combination of two values, is

$$F(u', u, c, t) = u' - f(u, c, t) = u_p$$

where u_p is the left-hand side of (20) with $b_0 = a = 1$, $b = 0$. This observation suggests the appropriate generalization of the computation (20): for implicit (and explicit!) systems, it computes the residual, or more properly the scaled quantity

$$u_p = aF\left(\frac{b_0}{a}u_0, u, c, t\right) + bu \tag{22}$$

Note that the interface `Dynamics::rhse` requires *no change whatsoever* to accommodate this reinterpretation, which is compatible with its previously assigned meaning for explicit systems.

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 (21) 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 (21) becomes

$$2D_{u'}F\left(\frac{u_{\nu-1}^{n+1} - u^n}{\Delta t}, u^n, t^n\right) + \Delta t D_u F\left(\frac{u_{\nu-1}^{n+1} - u^n}{\Delta t}, u^n, t^n\right) \tag{23}$$

which is of the form

$$D_{u'}F(u', u, t) + \alpha D_u F(u', u, t)$$

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

```
virtual void drhssol(LocalDataContainer u0,
                     LocalDataContainer u,
                     LocalDataContainer up,
                     Scalar alpha, Scalar t)
  { ... }
```

This method is implemented in the base class to throw an exception (as are all essential methods of the `Dynamics` class). A concrete child class which is to be used in an implicit solver will need to provide an implementation.

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 (21) with the frozen operator replacement (23) 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 two calls to `Dynamics::rhse`.

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::rhse` and `Dynamics::drhse`. 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::rhse` by a discretization error.

Step

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

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

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

virtual void adjStep(Model<Scalar> & mdl,
     Dynamics<Scalar> & dyn,
     Clock<Scalar> & 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 `SVL::Vector` and others in SVL 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<Scalar> & getState();
virtual LocalDataContainer<Scalar> & getControl();
virtual LocalDataContainer<Scalar> & getStatePert();
virtual LocalDataContainer<Scalar> & 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: We have supplied an implementation of the Euler forward method in the class `EulerStep`. The header file `step.H` also defines `EulerStepOnePass`, which assumes that you can immediately overwrite the state component-by-component. Here for example is the guts of the Euler implementation (`EulerStep`):

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

`upind` is a fixed index into the workspace array for states, maintained by the `Model` object `mdl`; `cp` is an instance of the `SVLCopy` function object. Both are private data managed by the `EulerStep` object.

Code for the derivative and adjoint derivative steps is similar.

The one-pass version of `EulerStep` avoids the copy at the end and the extra workspace by using `mdl.getState()` in place of `mdl.getWorkState(upind)` in the call to `dyn.rhs`. This is correct if the dynamics computation is arrange so as to use each component of the input state vector only once, and only in the statement in which it is updated.

Runge-Kutta methods: 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.

Note that one fewer work vector is needed than would be the case if this R-K method were written in standard fashion. The "scaled update" form of the `Dynamics` methods enables this storage economy.

Multistep Methods: The subclasses `MSModelBuilder` and `MSModel` provide access to implicit 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.

For example, a leapfrog method for (2) would normally be formulated by storing time steps u_a, u_b with $a = 0$ or 1 and $b = 1 - a$; u_a represents the previous time step, u_b the current step. The method (assuming constant time step Δt and time-independent control c for simplicity) then reads

$$\begin{aligned} u_a &\leftarrow u_a + 2\Delta t H(u_b, c, t) \\ a &\leftrightarrow b \\ t &\leftarrow t + \Delta t \end{aligned}$$

If we store instead $u = u_1$ and $v = (u_1 - u_0)/\Delta t$, with u_1 always representing the current time step and u_0 the previous one, then leapfrog amounts to

$$\begin{aligned} v &\leftarrow 2H(u, c, t) - v \\ u &\leftarrow u + \Delta t v \\ t &\leftarrow t + \Delta t \end{aligned}$$

and no index switching is required. Moreover the method becomes formally Euler's method with an appropriate definition of the discrete dynamics $\mathcal{H}$.

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<Scalar> & stat,
Dynamics<Scalar> & dyn,
Step<Scalar> & step,
Clock<Scalar> & clock,
int _itmax=10000, int _verbose=0);
```

The types of the first four arguments were discussed in detail in the last section. The last two arguments bound the maximum number of time steps permitted and regulate the terminal output: at the moment, any nonzero value of `verbose` causes a one-line message to be printed identifying each time step, plus other messages at major events during the simulation. Eventually other more refined levels of verbosity may be introduced.

In other ways the `TSOOp` type is used exactly as is any other `SVL::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. SVL 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 HCLREF and SVLREF 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
EulerStepOnePass<double> step; // Step
ConstClock<double> clock(tbeg,tend,dt); // Clock
TSOp<double> op(stat,dyn,step,clock); // Operator
```

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 SVL 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
SVLAssignConst<double> ac(0.5) // set components = 0.5
SVLRandomize<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.

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 SVL 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.