A software framework for abstract expression of coordinate-free linear algebra and optimization algorithms

Anthony D. Padula, Shannon D. Scott and William W. Symes *

ABSTRACT

The Rice Vector Library provides C++ classes expressing core concepts (vector, function,...) of calculus in Hilbert space with minimal implementation dependence, and standardized interfaces behind which to hide application-dependent implementation details (data containers, function objects). A variety of coordinate free algorithms from linear algebra and optimization, including Krylov subspace methods and various relatives of Newton's method for nonlinear equations and constrained and unconstrained optimization, may be expressed purely in terms of this system of classes. The resulting code may be used *without alteration* in a wide range of control, design, and parameter estimation applications, in serial and parallel computing environments.

INTRODUCTION

Large-scale simulation driven optimization arises in a variety of scientific and engineering contexts, notably control, design, and parameter estimation. Simulation of physical processes involves a variety of computational types and data structures specific to physical context and numerical implementation. Simulator applications typically include data structures for geometric meshes or grids and rules for their construction and refinement, functions on these grids representing physical fields, equations relating grid functions and embodying (gridded versions of) physical laws, and iterative or recursive algorithms which produce solutions of these equations.

Optimization and linear algebra algorithms on the other hand generally have no intrinsic interaction with physics and its numerical realization, involving instead a more abstract layer of mathematical constructs: vectors, functions, gradients,... Many such algorithms, including some of the most effective for large scale problems, may be expressed *without explicit reference to coordinates*. These *coordinate-free* algorithms use only the intrinsic operations of linear algebra and calculus in Hilbert space. Examples include Krylov

*Department of Computational and Applied Mathematics, Rice University, Houston TX 77251-1892 USA, email `symes@caam.rice.edu`

subspace methods for the solution of linear systems and eigenvalue problems, Newton and quasi-Newton methods for unconstrained optimization, and many constrained optimization methods.

This discrepancy between levels of abstraction is the source of a software engineering problem: in procedural programs to solve simulation driven optimization problems, the details of simulator structure invariably intrude on the optimization code, and vis-versa. Time-honored software "tricks" used to hide these details within procedural code (common blocks, parameter arrays, "void *" parameters,...) lead to software that is difficult to debug and maintain and nearly impossible to reuse outside of the originating context.

Object oriented programming appears to offer a way out of this dilemma. Data abstraction permits the implementation details in one part of a program to be hidden completely from other parts which do not intrinsically involve them. Polymorphism and inheritance complement data abstraction and enable reuse of abstract code (for example optimization algorithms) across many applications.

This paper describes an object-oriented software framework, the Rice Vector Library ("RVL"), for the expression of coordinate-free algorithms for large scale applications of continuous optimization and linear algebra. RVL provides C++ classes emulating a set of core *mathematical* concepts, entirely sufficient to express these algorithms. RVL realizes these concepts (vectors, functions,...) computationally in a set of abstract classes, such as `Space` and `Operator`, along with implemented (concrete) classes constructed out of the abstract components, such as `Vector`, which express related concepts. The central goal of the RVL project is to make the relationships amongst these "calculus" classes as parallel as possible to those amongst the mathematical concepts which they represent. Critical to the success of this approach is the provision of another collection of abstract "data management" classes (`DataContainer` and `FunctionObject` in RVL), which offer uniform methods for hiding implementation details.

An Illustrative Example

A simple example of the abstraction dichotomy is apparent in the least-squares solution of a linear thermal control problem via a Krylov subspace iteration. We shall refer to this example throughout the paper.

An individual step in such an iteration might be expressed as

$$\begin{aligned} q &= Ap \\ \gamma &= \langle r, r \rangle \\ \alpha &= \gamma / \langle p, q \rangle \\ r &= r - \alpha q \\ x &= x + \alpha p \\ \beta &= \langle r, r \rangle / \gamma \\ p &= r + \beta p \end{aligned} \tag{1}$$

Here A stands for the normal operator of the control problem, application of which involves solution of one or more partial differential equations or systems, discretized perhaps via a finite element method. The variables $x, r, ...$ are fields, regarded as vectors in an appropriate Hilbert space, representing the control (perhaps an initial temperature field), the residual in the normal equations, and the like.

Implicit in the above description are two features of this class of problems which motivate the design of the software framework described in this paper. The first line in the above algorithm, involving an application-dependent operator, typically requires several orders of magnitude more floating point arithmetic than do the other lines, which express generic linear algebra operations. This cost dominance of application-dependent operations over generic vector operations is the meaning we assign to the phrase "large scale" in this paper.

Second, note that *no mention* of the coordinates of $x, r, ...$ or of the matrix elements of A occurs in the algorithm (1). The (absolutely vital) details of the physics and numerics incorporated in the implementation of A simply play no role in (1), which presumes only that A defines a symmetric positive definite linear operator on an appropriate Hilbert space. A preconditioned version of this algorithm could be expressed in similar fashion.

Relation to other Libraries

A number of other projects have realized various versions of this "OO numerics" program for numerical linear algebra and/or optimization: see (Nichols et al., 1993; Douglas et al., 1994; Meza, 1994; Deng et al., 1996; ISIS Development Team, 1997; Tech-X, 2001; Tisdale, 1999; Veldhuizen, 1999; Karmesin, 2000; Benson et al., 2000; Langtangen, 1999; Gockenbach et al., 1999; Heroux et al., 2003; Kolda and Pawlowski, 2003; Bartlett, 2003).

RVL is a successor to one of these, the Hilbert Class Library ("HCL", (Gockenbach et al., 1999)), and incorporates many of its innovations and design principles. HCL introduced two important generic types, representing vector spaces and function evaluations respectively. Vector space objects act as *Abstract Factories* (Gamma et al., 1994), and thus implement computationally the notion of "set equipped with operations" mandated by the mathematical definition of vector space. Evaluation objects have the semantics of $f(x)$ for variable x, and provide a natural repository for intermediate data occurring in the computational realization of a function f and its derivatives.

While vector space and evaluation types are also key features of RVL, it differs from HCL in several important respects. RVL makes extensive use of ISO C++ features - class and function templates, exception handling, namespaces, the standard template library - which were not widely and/or robustly available from compiler vendors at the beginning of the HCL project in the early 90's. The "calculus" and "data management" aspects of RVL are more cleanly separated than was the case in HCL, which permits more of the key constructs to be implemented in base classes. For example, HCL required the user to implement a function evaluation class for each user-defined function. RVL defines a concrete universal evaluation class: the user need only implement the corresponding

function, with essentially the same attributes as were required of an HCL function implementation. The stratification of types in RVL also makes parallel execution environments transparent to RVL-encoded algorithms, a feature lacking in HCL. Finally, RVL offers an integral extensibility mechanism via *function forwarding*, inspired by Roscoe Bartlett's `RTOp` classes (Bartlett et al., 2004).

RVL provides computational constructs entirely sufficient to represent the linear algebra algorithm presented above, and many others. However, it is important to understand that RVL is not (amongst other things) a linear algebra library of the sort represented by LAPACK (Anderson et al., 1992) or TNT (Pozo, 2004) or Blitz++ (Veldhuizen, 1999). RVL does not express dense or sparse linear algebra or indeed any other sort of computation referring explicitly to the coordinates of a vector in a basis. That is not its objective. It is entirely possible and even advantageous to build RVL objects as "wrappers" around functions and/or types taken from linear algebra libraries like those mentioned above or many others, and in so doing take advantage of many person-years of development of high-performance numerical code. The design of RVL takes some care to avoid implicitly interfering in the features of these other libraries which lead to good execution speed - for example, RVL avoids aliasing of higher-level objects, which could force aliasing of array arguments in encapsulated coordinate-dependent code.

RVL is similar in some respects to the Toolkit for Advanced Optimization (Benson et al., 2000) and even more the Trilinos Solver Framework (Bartlett et al., 2003), with which it shares some common HCL-derived features. Both of these libraries define types systems for formulation of abstract numerical algorithms of the same sort RVL targets. Amongst other differences, however, both of these libraries impose more structure on their data container types than does RVL, and offer a richer palette of interfaces to be exploited by implementations. RVL aims to express the abstractions occurring in coordinate-free algorithms, as described above, and as little else as possible.

Plan of the Paper

The following pages discuss the design of "calculus" and "data management" types, and their interaction; the basic structure of the RVL vector classes follows from this discussion. Besides its basic interfaces, RVL also provides a collection of specialized interfaces representing various common mathematical structures, such as Cartesian products, which occur frequently in the formulation and solution of design, control, and inverse problems. These specialized types, defined in terms of the core types of the library, considerably ease the construction of applications.

We close by describing several example applications which illustrate the use of RVL to solve simulation-driven optimization problems in both serial and parallel environments. These examples showcase the reuse of abstract algorithm code across disparate application domains and computing platforms, and the feasibility of performance on par with that of competently constructed procedural code.

The base classes which are the principal topic of this paper are entirely abstract, and do not define data access methods. Some means to access data must be specified in order

that an actual application be written; such data access interfaces are outside the scope of RVL proper. In appendices, we describe both a simple, portable data access layer which we have used extensively in our own applications work, and an adaptation of parts of Sandia National Laboratory's Trilinos collection to form an RVL data access layer.

This purpose of this paper is the presentation of the RVL design and the reasoning underlying it. Accordingly, we describe the RVL classes in only enough detail to illustrate the principles of the design. We refer the reader to the RVL reference manual for a comprehensive description of the classes and their usage (Symes and Padula, 2005).

All class names in the following discussion refer to RVL classes. We include the namespace prefix `RVL::` only where there is a danger of confusion with some other namespace. We also strip comments, standard constructors, exception handling, and other boilerplate out of code listings.

VECTORS AND SPACES

The central concept of calculus and linear algebra is that of *vector*. It is important to understand that a vector, both mathematically and computationally, must be more than an array of scalar coordinates with respect to some arbitrary basis - a vector is also bound to specific rules for vector arithmetic, and (for Hilbert spaces) a specific Hermitian scalar product, and it shares these rules with all other vectors in its vector space. All vectors of a vector space may also share common metadata which determine their physical or mathematical significance in contexts other than vector arithmetic, for example finite element grid parameters, or Cartesian product structure, or choices of units. Also the coordinate data (and metadata) of vectors may be stored in core memory, on disk, or distributed over a network. It follows that neither intrinsic type arrays nor standard library containers are adequate stand-ins for abstract vector types.

This section describes the implications of linear algebra for the design of an abstract vector class and associated classes, and some details of one possible realization.

Design Implications of Linear Algebra

A typical definition of *vector space*, from a standard text ((Hoffman and Kunze, 1961), p. 19), reads

... a **vector space***... consists of the following:*
(1) a field F of scalars;
(2) a set V of objects, called vectors;
(3) a rule (or operation), called vector addition,...
(4) a rule (or operation), called scalar multiplication,...

from which we see that

- the fundamental concept is actually that of *vector space* - a vector is merely an

element of a vector space, gets its identity from the space to which it belongs, and is meaningless except in the context of membership in its space;

- the linear combination operation (combining vector addition and scalar multiplication, and having each as a special case) is an attribute of the vector space, not of its individual elements (vectors).

HCL introduced vector space as a type, and RVL adopts this innovation. The RVL vector space type `Space` is a class template. The (only) template parameter identifies the "field" of scalars, i.e. numeric type which serves as a proxy for an actual field. Any numerical type representing a subfield of the complex numbers is in principle admissible as a template parameter. The current release of RVL supports all of the C++ intrinsic numerical types, and the `std::complex` types built upon them, as template parameters.

Unlike its mathematical homolog, the computational space does not (cannot!) call all of its members (vectors) into existence as soon as it is instantiated. Instead, some mechanism must be provided for creation on demand. Such creation parallels the mathematical commonplace "...let x [vector] be a member of X [space]...". In many algorithm formulations, sentences like these occur in which X is a more-or-less arbitrary vector space: that is, creation of vectors must be accomplished in a way that hides the detailed structure of the space. A type which constructs instances of another type, without revealing the internal details of that type or of itself, is an *Abstract Factory* (Gamma et al., 1994). Computational spaces will thus be Abstract Factories, and will in addition bind vectors belonging to them to specific methods for carrying out the basic operations of linear algebra.

Vectors *are not* simply arrays of coordinates, but it is also true that vectors *have* coordinates. Linear combination and inner product must ultimately be realized as operations on coordinate arrays, implemented according to the rules provided by the space to which the vectors belong. These requirements have a natural translation in the structure of the RVL `Vector` class. `Vector` instances own (addresses of) abstract containers for coordinate data, for which RVL provides the `DataContainer` base type. Creating a `Vector` thus entails dynamic allocation of a `DataContainer` by a virtual constructor method of the `Space` class. `Space` provides the linear combination and inner product (virtual) implementations which manipulate `DataContainer` instances. `Vector` must delegate linear combination and inner product to the corresponding `Space` methods. `Vector` must therefore retain a reference to its `Space`, hence "knows" which space it is in. In the language of (Gamma et al., 1994), `RVL::Vector` is a *Facade*: it combines several other types to produce a new set of behaviours. It also functions as a *handle* to its dynamically allocated `DataContainer` member.

Implementing Space and Vector Classes

A UML diagram (Figure 1) displays the relations between `Space`, `Vector`, and `DataContainer`. A `Space` subclass implementer must supply five public virtual function overrides, along with constructors initializing whatever internal data is necessary to make these

work. The first is a virtual constructor for the `DataContainer` subtype encapsulating the data structures specifying `Vectors` in this `Space`:

```
virtual DataContainer * Space<Scalar>::buildDataContainer() const  = 0;
```

`Space` offers the ability to compare instances:

```
virtual bool Space<Scalar>::operator ==(const Space<Scalar> & sp) const = 0;
```

Since few spaces appear in algorithms, in comparison to other sorts of things such as vectors and functions, it's appropriate to test addresses as the first step in any implementation of `operator==`.

Finally, three methods specify the algebraic character of the `Space`:

```
virtual Scalar Space<Scalar>::inner(DataContainer const & x,
                        DataContainer const & y) const = 0;
virtual void Space<Scalar>::zero(DataContainer & x) const = 0;
virtual void Space<Scalar>::linComb(Scalar a, DataContainer const & x,
                        Scalar b, DataContainer & y) const = 0;
```

Following HCL and other OON libraries, RVL uses function-call syntax rather than overloaded arithmetic operators to express linear algebra operations. The efficiency-related reasoning behind this choice is well-known (see for example (Bartlett et al., 2004)).

Invocation of the RVL linear combination method `Space::linComb` asserts that the vector operation

$$y = ax + by.$$

has been performed on the underlying coordinate arrays hidden by the `DataContainer` arguments.

RVL makes no guarantee about the correctness of any call to `Space::linComb` in which the output argument (`DataContainer & y`) is aliased with the input argument `DataContainer const & x`.

Note that implementations can recognize and generate special code for various special cases (`axpy`, copy,...), as determined by values of the scalars `a` and `b`. To make sure that these can be tested reliably (and for other reasons mentioned below), RVL provides a *traits* class (Myers, 1995) which specifies the precise type and value of several constants (one, zero, etc.) not fully specified in `std::numeric_limits`, amongst other things.

It is important to recognize that very often a large part of the `Space` implementation may be shared between subtypes. For example, linear combination works exactly the same way in all coordinate systems. RVL supplies a shortcut to construction of `Space`s, so that usually the only code that need be supplied is the virtual `DataContainer` constructor, and more rarely overloads of `operator==` and/or `inner`. This `StdSpace` construction is discussed below.

`Vector` is concrete, with all methods implemented by delegation to methods of `Space` or `DataContainer`. The principal constructor takes a space and an optional initialization flag:

```
Vector(const Space<Scalar> & _sp, bool initZero = false);
```

Initialization of an `Vector`'s data (behind the abstract data container interface) is optional - the only initialization with coordinate-free meaning is initialization by zero, and that is the only option offered as part of construction.

Vectors should know the space that they belong to, and be able to announce their membership publicly:

```
const Space<Scalar> & Vector<Scalar>::getSpace() const ;
```

An instance of `Vector` thus depends on, and exposes, a `const` reference to a `Space`, which must necessarily refer to a pre-existing `Space` object. This relationship of computational `Vector` and `Space` objects, enforced by semantics of C++ references, mimics precisely the relationship of the corresponding mathematical objects, that is, the membership of a vector in its space. Existence of the space logically precedes that of the vector; a vector cannot exist in the absence of its space.

The only assignment with invariant meaning is assignment to the zero vector. This assignment, the inner product, and linear combination all delegate to the `Space` methods of the same names:

```
virtual Scalar Vector<Scalar>::inner(const Vector<Scalar> & y) const;
virtual void Vector<Scalar>::zero();
virtual void Vector<Scalar>::linComb(Scalar a, const Vector<Scalar> & x,
                        Scalar b=ScalarFieldTraits<Scalar>::One() );
```

The linear combination method implements $y \leftarrow ax + by$, where y is represented by `*this`. The scalar b defaults to 1, interpreted appropriately for the type `Scalar` via the aforementioned traits class template `ScalarFieldTraits`.

`Vector` also supplies the convenience methods (`scale`, `norm`,...) defined in `Space`, by delegation to the latter's methods of the same names. The norm methods bring up another use of the traits class. Norms must return positive reals. It's possible to confuse these with scalars for the real number types (`float`, `double`, possibly extended precision generalizations) but not for the complex types. Therefore the `ScalarFieldTraits` specializations defines an `AbsType` via typedef which is the return type of the absolute value function for the scalar type, and of `norm` and `normsq` (implementing $x \mapsto \sqrt{\langle x, x \rangle}$ and $x \mapsto \langle x, x \rangle$ respectively).

We emphasize that `Vector` is concrete, ready to use. Specification of a new vector type, representing a new underlying data structure, involves construction of a new subtype of `Space` (and likely of `DataContainer`) only. We leave open the possibility of overriding the

implementations of the vector operations by making them virtual, but expect subclassing of `Vector` to be profitable in very few cases, if ever.

The `Vector` class methods provide computational expression for every line but one of the simple Krylov step (1):

```
A.applyOp(p,q);
gamma = r.normsq();
alpha = gamma / p.inner(q);
r.linComb(-alpha, q);
x.linComb(alpha, p);
beta = r.normsq()/gamma;
p.linComb(one, r, beta);
```

The first line invokes the `apply` method on an `RVL::LinearOp` instance - RVL function and operator classes will be discussed in the next section. The other lines involve only vector arithmetic expressed via the methods of `Vector`.

Data Access and Manipulation

Some mechanism must be supplied for manipulation of coordinate arrays, else no actual computations can take place. The base class `DataContainer` cannot mandate such access to data, else implementations in which coordinate arrays are stored on disk or distributed over a network would suffer gross inefficiency. Efficient data access at this level, either by address or by value, is also impossible if Cartesian products of `DataContainer` objects are to be treated as `DataContainer`s. Therefore RVL defers data access to subclasses of `DataContainer`.

RVL's `FunctionObject` base type provides a uniform interface behind which to hide data manipulations of all sorts. Evaluation of concrete `FunctionObject` subtypes uses the data access services of concrete `DataContainer` subtypes. However control over the evaluation must reside in `DataContainer` objects owning that data, which own information about the layout and location of the data. Thus `DataContainer` is provided with a means to evaluate `FunctionObject`s.

This function-forwarding or *double-dispatch* design is inspired by the standard library's scheme for interaction of function objects and containers, and by Bartlett's RTOp package (Bartlett et al., 2004). It is an example of the *Visitor* pattern (Gamma et al., 1994). In the language of (Gamma et al., 1994), `DataContainer` is the base class of an Element hierarchy, each subtype of which accepts visits from types of a Visitor hierarchy, of which the base class is `FunctionObject`. Visitors use the services of the visited Element.

RVL supplies only the base classes of its Element and Visitor hierarchies. with the intention that these be extended; sample completions of these hierarchies are described in the appendices. Thus the design of RVL anticipates the introduction of both new Element (`DataContainer`) and new Visitor (`FunctionObject`) subtypes. The original Visitor pattern, as described in (Gamma et al., 1994), makes new Visitors easy to add.

However new Element subtypes are difficult: every Visitor subtype must define methods to visit every Element subtype. The Visitor pattern's strong coupling of the two hierarchies is thus incompatible with the intended use of RVL.

For this reason, RVL uses a modification of the Visitor pattern, dubbed *Acyclic Visitor*, which breaks the tight coupling between the Visitor and Element hierarchies (Martin et al., 1998; Martin, 2002). A salient characteristic of the Acyclic Visitor pattern is a degenerate Visitor base class, i.e. one with no (nontrivial) methods. `RVL::FunctionObject` defines only a standard reporting method (which may be overridden in subclasses); its interaction with `DataContainer` is left entirely undefined. Child classes of `FunctionObject` will define evaluation methods which access the services provided by child classes of `DataContainer`. Runtime type information will be used to properly associate `FunctionObject` and `DataContainer` subtypes, as is characteristic of applications of Acyclic Visitor.

The `DataContainer::eval` expresses acceptance of a Visitor (`FunctionObject`) by an Element, per the (Acyclic) Visitor pattern. This particular acceptance method also allows for the participation of other `DataContainer`s:

```
virtual void DataContainer::eval(FunctionObject & f,
                  std::vector<DataContainer const *> & sources) = 0;
```

Note that the `FunctionObject` argument is not designated `const`. RVL uses function objects, rather than say pointers to functions, to encapsulate data manipulations because function objects offer persistent state. The standard library uses them for the same reason: its algorithms (`std::for_each`, for instance) take non-`const` function objects whose internal states may be modified in successive calls. The ability to modify `FunctionObject`s via invocation of `DataContainer::eval` plays the same role as the analogous capability of the standard library, and is extremely useful in developing applications. Note that `FunctionObject` therefore does *not* model the behaviour of a mathematical function: repeated evaluation of a `FunctionObject` on the same data can deliver different results, since its internal state may change as the result of evaluation.

`FunctionObject` evaluation is the only general high-level mechanism for data manipulation provided by RVL. Therefore an interface must be provided to evaluate `FunctionObject`s from the vector level. `Vector` accomplishes this task via an `eval` method analogous to `DataContainer`'s, which delegates evaluation to its `DataContainer` data member and its `eval` method.

RVL does not support output aliasing in implementations of `eval` for either `DataContainer` or `Vector` types. That is, RVL does not require that implementations assure correct results when the `DataContainer` or `Vector`, on which `eval` is called, is aliased with any of the input arguments `sources[i]`.

`FunctionObject` is intended to encapsulate calculations which return "large" results, whence the "return value" of an `FunctionObject` is the `DataContainer` on which the `eval` method is called with the `FunctionObject` as first argument. Since the other (possible)

DataContainer arguments are passed by address, opportunities to minimize data motion are preserved by the design.

On the other hand many functions return results with perfectly usable copy semantics, such as scalar field values or booleans. It has become common to term such functions *reductions*, perhaps because the output is generally much smaller than the input. For such functions, RVL provides an abstract class for return values, RetType, which mandates an assignment operator and a default initialization value, and a FunctionObjectRedn interface which is the base class for a second Acyclic Visitor hierarchy. Once again, FunctionObjectRedn specifies no particular form of interaction with DataContainers, but does have a nontrivial role in managing a RetType instance, via setResult and getResult access methods.

Both DataContainer and Vector are provided const methods to evaluate FunctionObjectRednS, the latter by delegation to the former; the target of an FunctionObjectRedn is its internal RetType instance, and the DataContainer on which a FunctionObjectRedn evaluation is called is treated as read-only. A typical example is the implementation of StdSpace::inner, discussed below.

Since arithmetic types are frequently return values of useful functions (inner products come to mind), RVL provides a templated ScalarRetType subtype of RetType, encapsulating the template type and inheriting the usual collection of arithmetic operations. A ScalarReduction mixin template provides alternative access to the result in the form of the template type. The FunctionObjectScalarRedn subclass of FunctionObjectRedn inherits the ScalarReduction interface also. The first appendix presents some simple examples.

Besides providing a natural mechanism for the implementation of (scalar-valued) functions and operators (vector valued functions), discussed in the next section, FunctionObjects provide a natural interface for the many *scalar functions* which appear in algorithms. Scalar functions act on each coordinate of input and output separately: they take the form

$$z_i = f(x_i, y_i, ...), \ i = 0, 1, 2....$$

Bartlett (Bartlett et al., 2004) terms these functions *vector transformation operators*, and points out that only functions of this type (and the analogous reduction scalar functions) can be implemented efficiently without detailed knowledge of data layout. Linear combination is an example of such a function, and indeed a standard implementation of Space uses a FunctionObject encapsulating $y_i = ax_i + by_i, \ i = 0, 1, 2, ...$

RVL does not legislate the method(s) by which DataContainers might expose their data, which types of DataContainers do so, nor what form of data access a particular FunctionObject might expect. The first appendix briefly discusses several possibilities. It is however clear that FunctionObjects encapsulating scalar functions and not depending on DataContainer metadata, like linear combination, can be applied to any DataContainer which exposes its data array in an appropriate way. By extension, such FunctionObjects can also be evaluated on any tree-structured DataContainer whose leaves expose coordinate data compatibly with the FunctionObject's expectations. and which

forward FunctionObjects up the tree to the leaf level for evaluation. The tree traversal mode of FunctionObject evaluation is a typical use of Visitors with *Composite* Element types (Gamma et al., 1994).

Figure 2 is a UML depiction of a typical instance of tree-traversal evaluation. At the top, the eval method is called on a Vector, with an instance of RVLRandomize, a FunctionObject subtype, as argument (i.e. with a length zero std:;vector of source vectors). Control passes to the DataContainer data member, in this case an instance of StdProductDataContainer. The StdProductDataContainer class, discussed later in this paper, is a concrete implementation of a Cartesian product of DataContainers, a Composite in design pattern terms. The components of this product, in the case illustrated, are DataContainers which expose their array data, and own references to no further DataContainers. The specific type is discussed in the first appendix. FunctionObjects are evaluated directly on this latter type: its eval method calls the operator() method of its FunctionObject argument, which exposes the DataContainer's data. The particular FunctionObject named in Figure 2 is one that assigns pseudorandom numbers to the data array of a DataContainer argument. When control returns to the Vector on which the eval method was called, the coordinate array which it encapsulates has been initialized with pseudorandom numbers.

It is important to emphasize that all of this happens as the result of a single line of user code:

```
x.eval(rand);
```

provided that the objects x and rand have been properly initialized.

Convenience Features and Classes

The Space class provides a number of "convenience" methods, such as norm, normsq, scale, negate, and copy, with virtual default implementations built out of linComb and inner. These default implementations create temporary storage and imply more passes through data than would be required by tailor-made implementations. However recall that the target applications of RVL are those in which function evaluation (simulator) cost overwhelms the overhead of vector arithmetic. We have found that in such applications the default performance of the convenience operations is perfectly acceptable. Also careful implementation of linComb detects and efficiently handles the special cases occuring in the convenience methods.

Experience has shown that the flexibility in number of arguments built into the Vector::eval methods for evaluating FunctionObjects and FunctionObjectRedns is often neither useful nor convenient. Most function objects turning up of their own accord in algorithm formulation and application have between one and four arguments, and interfaces explicitly specifying such small numbers of arguments are easier to implement than the generic access method. Therefore RVL provides specializations of Vector::eval admitting zero to three Vector arguments (in addition to the Vector on which eval is called). These are implemented in the obvious way by delegation to the generic evaluation

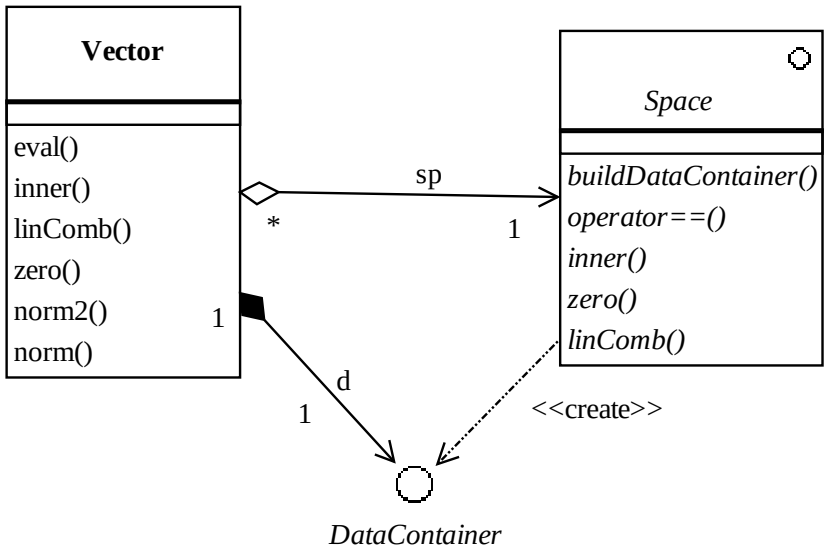

FIG. 1. `Vector`, `Space`, and `DataContainer`.

method. For example, the binary interface, combining two `Vector`s and a `FunctionObject`, is:

```
void Vector<Scalar>::eval(FunctionObject & f,
                          Vector<Scalar> const & source) {...}
```

A `Space` requires two types of functionality: the ability to construct dynamically a specific type of `DataContainer`, and the capacity to provide several linear algebra operations in the guise of member functions. It is convenient to separate these two aspects of `Space`, by providing natural types with these two sets of attributes, together with a construction of `Space` in terms of these types. RVL defines a Factory base type, `DataContainerFactory`, which fulfills the first role. A `LinearAlgebraPackage` provides access to a `FunctionObjectScalarRedn` and two `FunctionObject`s defining inner product, zero initialization, and linear combination, with access methods. The `StdSpace` Facade class combines a `DataContainerFactory` and a `LinearAlgebraPackage` to realize a `Space`. This construction facilitates code reuse: for example, any class of Hilbert spaces whose inner products share a given Gram matrix in their standard bases can be defined by the same `LinearAlgebraPackage`

FUNCTIONS AND EVALUATIONS

RVL offers three base classes for functions of a vector variable:

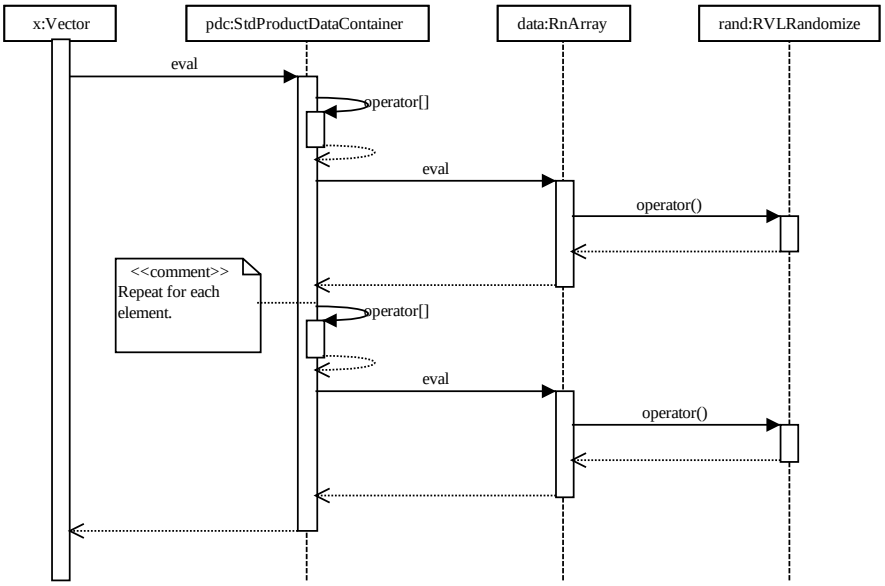

FIG. 2. Typical evaluation of a `FunctionObject` on a `Vector`.

- `LinearOp`, for linear operators;
- `Functional`, for scalar-valued functions;
- `Operator`, for vector-valued functions.

Mathematically, all three of these possibilities are special cases of a general vector function concept. However computationally it is not convenient to derive all three types from a common parent. C++ distinguishes between intrinsic and object types (unlike Java, say). Therefore identifying the scalars with vectors in a 1-D vector space, harmless mathematically, leads to annoying complications and inefficiencies in the definition of function types. Consequently RVL distinguishes between scalar- and vector-valued functions, and does not attempt to represent them as subtypes of a common parent. The general vector-valued function type must offer access to a derivative as a linear-operator-valued function. If linear operators are viewed as a subtype of the general vector-valued function type, it is difficult to take advantage of the fact that the derivative of a linear operator is constant, and equal to itself. For these reasons, RVL offers three distinct types, with convenient access to the features of each natural to their mathematical uses, rather than trying to fit all three to a uniform parent interface.

In some cases common mathematical usage also conforms to these distinctions. For instance, both scalar-valued and vector-valued function interfaces must include, in some form, access to the derivative as a linear map. However in Hilbert space optimization theory and practice, the useful form of the derivative of a scalar-valued function is the

gradient, i.e. the Riesz representer of the derivative, rather than the derivative itself. RVL conforms to this usage pattern, returning the derivative in the `Operator` type as a `LinearOp`, in the `Functional` type as a `Vector`.

Linear operators

Many OO libraries expressing linear algebra algorithms define some sort of linear operator or matrix type. As RVL aims to express *coordinate-free* algorithms, its `LinearOp` interface is not required to divulge matrix elements, for example. Only a operator application method `apply` (or matrix-vector product, if you insist on thinking of linear operators as identified with their matrices) need be supplied to express the coordinate-free usage of linear operators. `LinearOp`s know their domain and range, and offer access methods exposing these as `Space`s.

The structure of Hilbert space implies that linear operators exist in pairs: each operator has an adjoint. Adjoints are critically important in definition of many algorithms, for example playing a key role in the computation of gradients. Unlike the mathematical homolog, however, a `LinearOp` object does not automatically have its adjoint defined when the operator itself is defined - a separate implementation must be supplied. Like the mathematical homolog, "adjoint" here means: with respect to the inner products defined in range and domain, which as noted above are attributes of the `LinearOp` object.

Therefore the fundamental linear mapping type in RVL encapsulates *pairs* of linear operators, adjoint to each other. A `LinearOp` offers two public methods for operator application:

```
void applyOp(const Vector<Scalar> & x, Vector<Scalar> & y) const;
void applyAdjOp(const Vector<Scalar> & x, Vector<Scalar> & y) const;
```

applying the operator and its adjoint to the input vector represented by `x` and storing the output in the vector represented by `y`.

The `applyOp` and `applyAdjOp` methods are implemented in the base class. They delegate to the protected `apply` and `applyAdj` methods, which are pure virtual. To build an instantiable `LinearOp` subclass, the user must implement these latter:

```
void apply(const Vector<Scalar> & x, Vector<Scalar> & y) const;
void applyAdj(const Vector<Scalar> & x, Vector<Scalar> & y) const;
```

as well as public methods exposing the domain and range spaces:

```
const Space<Scalar> & getDomain() const;
const Space<Scalar> & getRange() const;
```

and a clone method:

```
LinearOp<Scalar> * clone() const;
```

along with constructors which initialize whatever subclass data members these implementations require. For example, domain and range `Space`s might be passed by reference to the constructor. and stored as `const Space &` data members. A typical implementation of the `clone` method calls the subclass copy constructor on `*this`.

A typical use of the methods exposing domain and range is to generate workspace, for example:

```
Vector<Scalar> x(A.getDomain());
```

RVL also supplies overloads of the `LinearOp::applyOp` methods which include linear combination with a vector, i.e.

$$y \leftarrow \alpha Ax + \beta y$$

and the analogue for the adjoint. These overloads are provided default implementations in the base class, which may be overridden to supply a degree of loop fusion. This device, popular in other OO numerics libraries, is less useful than one might think; annoyingly often, algorithms require access to Ax as well as to the linear combination, so nothing is gained. The Krylov step (1) is an example of this phenomenon: since the result of the operator-vector product is needed as a factor in an inner product, nothing is gained by fusing the product with a vector addition as is needed in the line following the inner product.

In fact the problem which this overload occasionally solves is an instance of a very important open problem in computer science, that of cross-type method optimization. A good solution seems to be some way off. In the meanwhile, the preferred RVL solution is to write *nonlinear* operator subtypes (note that the `apply` overloads actually define *nonlinear* maps!) and provide their implementations with appropriately fused loops, for example to output the inner product and linear combination appearing in (1) as the result of a single instruction, when utility warrants.

To facilitate application-building, `LinearOp` also implements a `checkAdjointRelation` method. This method is an example of a *built-in unit test*, a device which RVL uses wherever possible (as did HCL - this test was an attribute of `HCL_LinearOp` as well). This test checks internal consistency between application of the operator and its adjoint, by choosing random vectors in domain and range, calling `apply` and `applyAdj`, computing inner products, and reporting the results on the stream argument. The return value is *false* when the test succeeds, i.e. encounters no problems, meaning that the obvious pair of inner products differ relatively by less than `tolfac` times the machine epsilon for the scalar type. Failure of this test, or any exception thrown during execution of the method, causes the return value to be `true` (in particular, exceptions are trapped within `checkAdjointRelation`). This structure leads to a very simple regression test template:

```
int main() {
  RVLRandomize<double> rnd;
  // initialize a linear op
  if (op.checkAdjointRelation(rnd,cerr)) exit(1);
}
```

Random initialization is not a basic linear algebra function, and has been excluded from the `Vector` interface. Therefore it must be implemented as a `FunctionObject` and passed as an argument to the test method. As a corollary, the linear operator class does not guarantee correctness of this function object - it is external. The contract between the algorithm writer (in this case also the class designer!) and the user is this: if the user supplies a function object which randomly initializes the components of a `LocalDataContainer`, then this method reveals whether the implemented operator and its adjoint, applied to random vectors, produce results which stand in the correct relation vis-a-vis the inner product. This sort of guarantee is maximal.

Some software experts opine that unit tests such as `checkAdjointRelation` should be implemented as standalone (nonmember) functions, rather than as class methods. No obstacle exists to separating this test from the `LinearOp` class. However the test seems to us an integral attribute of the type, therefore appropriately implemented as a method.

Nonlinear functions and evaluation objects

The definitions of types for general scalar- and vector-valued functions is inevitably more complicated: while the derivative of a linear map is itself, and higher derivatives vanish, RVL must find some way to represent at least the first two derivatives of a general function as operator-valued functions. An immediate complication is that computations of the value of a function and of its derivatives tend to share intermediate results. For example, a finite element simulator will need mesh generation, stiffness matrix assembly, etc., and the derivatives ("sensitivities") may well require precisely the same data. However not all of these values are necessarily needed at any one point in a program. For example, a typical line search method will require a gradient for computation of a search direction, but then undertake a line search which may require only function values. To avoid (possibly) very expensive redundant computation, some means is needed to keep consistent sets of intermediate results, depending on the evaluation point, between calls for values. Since a function may be evaluated at many points, a single object representing a function does not offer a convenient framework to meet this need.

HCL solved this problem through the introduction of *Evaluation* types. An Evaluation expresses the jet, i.e. the sequence of derivatives, of a function at a point. Since an Evaluation is an object with persistent state, and depends on both an evaluation point and on a function to be evaluated, all of the intermediate data can be stored without fear of internal inconsistency. In our opinion, the introduction of Evaluation types was one of HCL's two main contributions to object oriented numerics, the other being the recognition of the central role of vector space as a computational type.

Other groups have proposed similar solutions to the evaluation problem. For example, the NOX project at Sandia National Laboratories uses a type very similar to Evaluation, which they call a Group - an object which *groups* together a function, its value, and the values of its derivatives at a point (Kolda and Pawlowski, 2003).

RVL Evaluation objects are an evolution of HCL's. Most importantly, RVL Evaluations are *concrete*: the application developer need only implement a `Operator` or

`Functional` object, and RVL supplies the necessary Evaluation code. Semantically, RVL Evaluations represent a function's value at a *variable* argument, i.e. $f(x)$ for variable x. The data contained in x may change within its mathematical scope, and with it the value $f(x)$, but in $f(x)$ these are dynamically linked. In contrast to HCL, which required a new Evaluation object for each new argument value, and NOX, which requires the algorithm writer to manually force an update of the value upon change in argument value, RVL Evaluations automatically recompute values when arguments change.

RVL Evaluations are instances of the Facade pattern, reminiscent of `Vector`. Internally, a (fully initialized) RVL Evaluation is a pair consisting of a reference to an external `Vector`, and an independent (internal) copy of the object representing the function (`Operator` or `Functional`). All access to function values in RVL occurs through Evaluations. Evaluations are handles to function types, and use their `Vector` member to tell the function how to behave (where to evaluate itself), just as `Vector` uses its `Space` reference to tell its `DataContainer` how to behave like a vector.

RVL Evaluation classes manage the storage of function results: they retain values in data members and return references to them. In order to avoid the dangling reference problems that would otherwise result, all data members exposed by reference are allocated on the stack, i.e. their lifetimes are identical to those of the Evaluations that own them. It might be objected that some results could contain significant data (eg. a `Vector` such as a computed gradient), whence their allocation should be deferred until possible use. Such a strategy would require dynamic allocation of the these results, and ultimately reference counting. However, member initialization of a gradient `Vector`, for example, does not immediately allocate its `DataContainer` data member: this occurs only when the `DataContainer` is first written. This strategy accomplishes the desired goal - for example, if only `Functional` values are required, memory for gradient data is never allocated - without requiring dynamic allocation of high-level constructs like `Vector` or the attendant need for bookkeeping of shared references.

An objection to this design should immediately occur to the reader: what is to guarantee that the referenced evaluation point remains consistent with the results managed by the Evaluation object? Use of `const` cannot ensure this consistency, as the relation not defined at compile time.

RVL solves this problem by introducing a *watch* relationship between an `Evaluation` object and its reference evaluation point `Vector` which is

- transparent to the RVL application developer;
- (because it is) implemented in the base classes;
- a run-time relation between internal states, rather than a compile-time language feature;
- secure, because the data on which it depends is well-encapsulated.

The *watch* relationship is an instance of the *Observer* pattern. It works as advertised because the *only* public access to a `Vector`'s internal state is through fully implemented methods of other RVL classes : the non-`const` overload of `Vector::eval`, the several linear algebra methods of `Vector`, and the value access methods of the various mapping classes, such as `LinearOp::applyOp` and `OperatorEvaluation::getValue()`. These methods invoke the version update method on which the watch relation is based. Because RVL allows no other *public* way to change the state of a `Vector`, other than invocation of these methods, the user is assured that Evaluations maintain the natural dynamic dependence of function values on arguments.

`Functional` and `FunctionalEvaluation`

We will describe the scalar-valued function class `Functional` and its associated evaluation type, `FunctionalEvaluation`, in detail. The structure of the corresponding classes for vector-valued functions is precisely parallel.

Building a `Functional` involves implementing six methods, four of these protected, and appropriate constructors. The protected `clone` method is a virtual copy constructor:

```
virtual Functional<Scalar> * Functional<Scalar>::clone() = 0;
```

The usual subclass implementation uses the subclass's copy constructor in the obvious way.

The heart of a `Functional` is the computation of value, gradient, and Hessian, encapsulated in the other three protected methods:

```
virtual void Functional<Scalar>::apply(const Vector<Scalar> & x,
                                       Scalar & val) const = 0;
virtual void Functional<Scalar>::applyapplyGradient(const Vector<Scalar> & x,
                                                    Vector<Scalar> & g) const = 0;
virtual void Functional<Scalar>::applyapplyHessian(const Vector<Scalar> & x,
                                                   const Vector<Scalar> & yin,
                                                  Vector<Scalar> & yout) const = 0;
```

Since evaluation will take place within *independent copies* or clones of a function type instance, one clone for each evaluation point, the function types are intended to store all intermediate data needed in any evaluation, in principle as write-once, read-many data. The base `Functional` class makes no attempt to specify the precise form of this storage, of course: it simply provides pure virtual interfaces for evaluation of a function and some of its derivatives. Since a `Functional` instance is intended to be cloned, an efficient implementation will allocate storage for intermediate data dynamically, as needed. These allocations typically occur in the bodies of the `Functional::apply` method definitions. Intermediate data, once initialized, can be regarded as fixed for the life of the instance, since the evaluation point on which it depends is bound to the `Functional` instance through an Evaluation object, as explained in the last subsection. . Therefore,

the `Functional::apply` methods should be accessible *only* to Evaluation objects. Accordingly, the `apply` methods are protected, and the `FunctionalEvaluation` class is a friend of `Functional`. Essentially the only `friend` declarations in RVL make Evaluation types friends of their associated function types.

Of the two public virtual methods `getDomain` returns a `const Space` reference. It must be implemented in any instantiable subclass, usually by returning a reference to a data member. The other public method, which should often be overridden, is

```
virtual Scalar getMaxStep(const Vector<Scalar> & x,
                          const Vector<Scalar> & dx) {
  return numeric_limits<Scalar>::max();
}
```

This method represents our attempt to solve a difficult problem in scientific computation: the description of more or less arbitrary subsets of high dimensional spaces. Since the domain of an arbitrary function can be such a set, we are faced directly with this problem. In view of the intended use in optimization, HCL chose to adopt an implicit approach, including a method in the function classes returning signed distance to the boundary of the domain in a specified direction, from a specified point. RVL has also adopted this approach. ...

The methods `Functional::checkGradient` and `Functional::checkHessian` are unit tests, and are implemented in the base class. They estimate the convergence rate of a second order finite difference approximation to the directional first, respectively second, derivative to that produced by calls to the `apply` methods. These rates should converge to 2, and it is usually possible to catch coding errors rather quickly by running these tests. Since the structure of the tests is independent of the particular function, it is possible to code them in the base class and so provide one-line access to the application developer.

A natural construction of a `Functional` subtype might rely on the services of several appropriate `FunctionObject` and `FunctionObjectRedn` instances to implement its `apply` methods. RVL provides a `StdFOFunctional` (partially implemented) subclass which facilitates this type of construction. To take advantage of any data shared between value, gradient, and Hessian evaluation, these `FunctionObject` and `FunctionObjectRedn` objects will need to share access to an external object which serves as a repository for the shared data.

In contrast to a `Functional` subclass, which requires quite a bit of programming, `FunctionalEvaluation` is completely implemented. The user need only construct one: given a `Functional f` and a `Vector x`, the evaluation is simply `FunctionalEvaluation fx(f,x)`. Public methods of `FunctionalEvaluation` provide access to the domain `Space` (delegation to its `Functional` data member) and access to the results:

```
Vector<Scalar> & getPoint() const;
Scalar getValue();
Vector<Scalar> const & getGradient();
LinearOp<Scalar> const & getHessian()
```

The const method FunctionalEvaluation::getPoint() returns a reference to the evaluation point. Since the evaluation point is mutable - for instance, may be updated in the course of an iterative algorithm - the reference is non-const. The FunctionalEvaluation value access methods, on the other hand, cannot be const as the internal state of the object may change on invoking them. However the latter two return const references - references for avoidance of data motion, const because algorithms which use them should not update them except indirectly, by updating the evaluation point.

The vector-valued function class Operator and its associated evaluation class work in a precisely similar way. We refer the reader to the reference guide for details. While Functional provides access to the second derivative, Operator is equipped only with the first derivative in the current release. This is sufficient to define Newton's method for nonlinear equations, and secant and Gauss-Newton or Gauss-Newton-Krylov methods for nonlinear least squares optimization. Addition of higher derivatives to these interfaces is straightforward, provided that appropriate multilinear operator classes are defined. HCL provided a bilinear operator class and a second derivative interface for vector valued functions. If such augmentation proves useful it will be added to subsequent versions of RVL.

A Quasi-Newton Algorithm Expressed in RVL

The core code of a limited memory quasi-Newton (BFGS) implementation shows a typical use of Functional, FunctionalEvaluation, and other core RVL classes. For a mathematical description of this algorithm, consult (Nocedal and Wright, 1999).

In this code, the Vectors x and xprev represent the current and previous search points. The LinearOp object H stores a BFGS inverse Hessian approximationThe update of H requires both the current and previous gradients, so another Vector, gprev, is also provided for gradient storage. The search direction is dir, and fx is a FunctionalEvaluation object representing the 2-jet of the function to be optimized at the search point x. Note that both xprev and gprev must be independent objects, not dependent on x or fx, so the Vector::copy invocations are mandated by the logical structure of the algorithm.

The line search is encapsulated in the Algorithm object ls. The Algorithm package, an offshoot of the RVL project described in more detail below, defines this type: the chief attribute of an Algorithm is that you can run it, and the run terminates, either successfully or not, as signified by the return value of the run method (Padula, 2004). Note that the BFGS algorithm itself is also an Algorithm.

```
Functional<Scalar> f;
Vector<Scalar> x;
...
FunctionalEvaluation<Scalar> fx(f,x);
...
bool run() { try {
    // compute current search direction
```

```
    H.applyOp(fx.getGradient(), dir);
    dir.negate();

    // copy current iterate, gradient
    xprev.copy(x);
    gprev.copy(fx.getGradient());

    // Line Search - will typicall update x, hence fx!
    ls.set(xprev, x, dir, fx);
    if( ! ls.run() ) { ... } // throw exception

    // update inverse Hessian approximation
    H.update(xprev, x, gprev, fx.getGradient());

  } catch (RVLException & e) { ... }
  return true;
}
```

The externally defined line search object `ls` acquires references to the base point, search direction, and function to be evaluated via its `set` method. Whenever the search point `Vector x` is updated, for example in the line search (`ls.run()`), the `Functional-Evaluation fx` updates its results.

A somewhat more involved illustration of the use of these classes is the `run` method core of a simple backtracking linesearch algorithm, displayed below with some inessential detail stripped out. Construction of this particular line search object takes the `FunctionalEvaluation fx`, a maximum number of steps, a step `tstep` which is managed by a linesearch superclass, and a step reduction factor `gamma`. The linesearch superclass also supplies a method to check for steps that are too small.

```
Vector<Scalar> & x = fx.getPoint(); // current iterate
Vector<Scalar> x0(x); // base point of search (copy construction)
Scalar fval =  fx.getValue();;      // value of f(x) at base
Scalar gfdx = dx.inner(fx.getGradient()));  // descent rate at base
Scalar cgfdx =  con*gfdx; // scaled descent rate
Scalar maxstp = fx.getMaxStep(dx); // step to boundary of domain

if (gfdx > 0.0) { return false; } // not a descent direction
tstep = ... // code to set initial step
// check that step is not too small
if (!this->checkMinStep()) { return false; } //

// first update
x.copy(x0); x.linComb(tstep, dx);
```

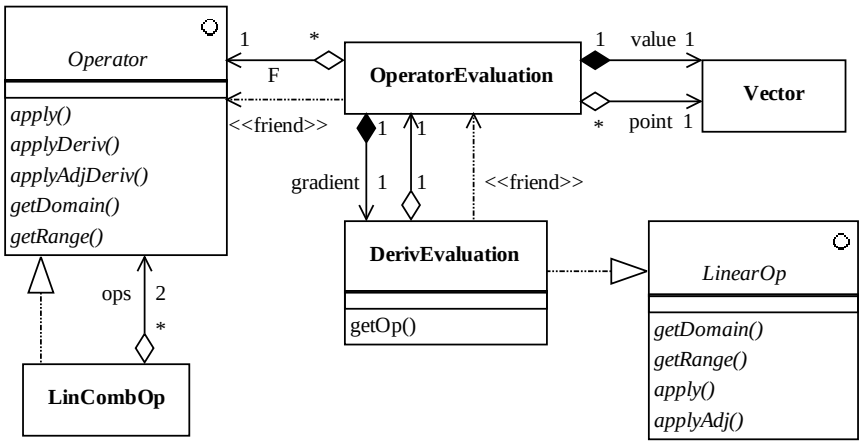

FIG. 3. Structure of the RVL `Operator` class.

```
  // while not sufficient decrease, shrink step
  int bt = 1; // number of backtracks
  while( fx.getValue() > fval + tstep*cgfdx &&
         this->checkMinStep() &&
         bt <= maxsteps) {
    tstep *= gamma; x.copy(x0); x.linComb(tstep, dx);  bt++;}
  // insufficient decrease or too many steps
  if (fx.getValue() > fval + tstep*cgfdx &&
      this->checkMinStep()) { x.copy(x0);  return false; }
  // sufficient decrease but step too small
  if ( !this->checkMinStep() ) { x.copy(x0);  return false }
  // successful line search
  return true;
}
```

Note that the first line defines `x` as an alias for the evaluation point of the `Functional-Evaluation` object `fx`, and that `x` is updated via linear combination in the line commented "first update". Consequently, calling `getValue` on `fx` a few lines later entails recomputation of all results encapsulated in `fx`. We emphasize that this dependence of the state of `fx` on the state of `x` is automatic, requiring no explicit instruction to be inserted in the algorithm expression.

PRODUCTS AND COMPOSITIONS

Virtually all scientific data structures other than the very simplest are Cartesian products. Also, the functions and operators appearing in simulation may often be compositions or linear combinations of simpler operators and functions. RVL provides standardized constructions of these derivative types, which are helpful in application construction and especially useful in rapid prototyping exercises.

Product Data Structures and Vector Spaces

RVL's realization of Cartesian product data structure is the `ProductDataContainer` interface. `ProductDataContainer` provides abstract access to components of a Cartesian product `DataContainer` , and defines evaluation of `FunctionObject`s by delegation to the components. `ProductDataContainer` is a *Composite* design, combining a set of objects of a given type into another object of the same type.

A typical use case for the Visitor pattern is delegation of Visitor acceptance to the factors of a Composite Element. `ProductDataContainer` implements this use case: Evaluation of a `FunctionObject` proceeds through delegation to the factor data containers, to which `operator[]` provides access. That is, the action of a `FunctionObject` on a Cartesian product of `DataContainer` s is formally block diagonal. Since `FunctionObject`s have persistent internal state, in fact non-diagonal actions are possible, but the interface is designed specifically to make block diagonal actions easy to implement.

A `StdProductDataContainer` child class implements this type in the obvious way (as a wrapper around a `std::vector` of `DataContainer`s), but the base class leaves the key methods `getSize()` and `operator[]` pure virtual. We have used this freedom to build an out-of-core seismic data class, for example, which subclasses `ProductDataContainer` in a natural but non-standard way. Distributed `DataContainer`s may provide other examples of nonstandard `ProductDataContainer`s.

The abstract base class `ProductSpace` represents the Cartesian product of vector spaces. This base class leaves the factory method `buildDataContainer` virtual, and adds size (`int getSize() const`) and component (`const Space<Scalar> & operator[](int i) const`) access methods, both pure virtual. The other (linear algebra and comparison) methods of the `Space` interface are implemented in terms of these latter two methods. Note particularly that this implicitly diagonal implementation of the inner product is certainly not mandatory, whereas there is little choice in the structure of the (say) the linear combination implementation. Non-diagonal Gram matrices may be implemented as overrides of the default implementation in subclasses.

RVL defines a natural standard `StdProductSpace` subclass, using `std::vector` and `StdProductDataContainer`, which works in the obvious way.

There are no `ProductVector`s in RVL, just `Vector`s in `ProductSpace`s, A `Components` class provides access to components as `Vector` instances, via the `int getSize() const` and `Vector<Scalar> & operator[](int i)` (both `const` and non-`const`) attributes of a

product structure. `Components` is another `Facade` type. To instantiate a `Components` object, you must first instantiate a `Vector`:

```
Vector<Scalar> v(sp);
Components<Scalar> cv(v);
for (int i=0;i<cv.getSize();i++) { pv[i].eval(...); }
```

The `Space` appearing in this construction need not necessarily be a `ProductSpace`. If not, then the `Components` object constructed on a `Vector` in this space has a single component.

The `ProductSpace` construction can be used recursively, i.e. the factors of a `ProductSpace` may themselves be Cartesian products.

Linear Combinations and Compositions of Functions

Both linear combination and composition of functions occur naturally in many applications and algorithms. RVL provides concrete subclasses of the function base classes implementing these concepts. These classes are intended to provide easy construction of various composite functions.

For example, `LinCombFunctional` builds a linear combination of two functions, as might occur for example in construction of a regularized objective function for parameter estimation. Given two `Functional` instances `f1` and `f2` implementing functions f_1 and f_2, the linear combination $a_1 f_1 + a_2 f_2$ is realized as

```
Scalar a1 = ...
Scalar a2 = ...
LinCombFunctional<Scalar> lc(a1,f1,a2,f2);
```

The summand `Functional`s must of course have the same domains, and this is checked as part of the construction.

The linear combination constructs for vector-valued functions, `LinCombOperator` and `LinCombLinearOp`, work the same way.

Composition of operators is a useful tool in application-building: various stages of a simulation may be built and tested independently, then composed to render the correct mathematics. If X, Y, and Z are Hilbert spaces, $f : A \to Y$ and $g : B \to Z$ are functions with domains $A \subset X, B \subset Y$, and $f(A) \subset B$, then the composition $g \circ f : A \to Z$ is well-defined by $g \circ f(x) = g(f(x))$. If `f` and `g` are `Operator` objects with `f.getRange() == g.getDomain()`, then RVL defines a composition `OpComp<Scalar> gof(f,g)`. Note that `Operator::getDomain()` actually returns the `Space` of which the domain of the represented operator is a (possibly proper) subset. Unless the domain is the entire space, therefore, additional tests beyond the use of `Space::operator==` are essential to guarantee sensible computations.

This issue does not exist for linear operators, whose domains are their entire domain Hilbert spaces. Thus checking that `A.getRange() == B.getDomain()` is sufficient to guarantee success of the composition `LinearOpComp BoA(A,B)`.

We explained earlier that subclassing `LinearOp` from `Operator` is inconvenient for both mathematical and computational reasons. However it is frequently useful (and mathematically reasonable) to *view* a linear operator as a nonlinear operator. RVL offers a *wrapper* class, `LNLOp`, which constructs an `Operator` using the services of a `LinearOp`.

RVL also provides a class for composition of a scalar-valued function with a vector-valued function. This combination occurs frequently in simulation-driven optimization problems, as a means of defining objective functions. Various versions of least-squares objective functions occur often enough that RVL defines them as well, using this `FcnlOpComp` constructor and other composites just described. For example, `RegLeastSquaresFcnlGN` is constructed from RVL objects representing an operator F, a linear operator R with the same domain space as F, a data vector d, and a regularization weight λ. The result is a `Functional` representing the penalized least squares objective

$$x \mapsto \frac{1}{2}(\|F[x] - d\|^2 + \lambda^2 \|Rx\|^2)$$

Efficient evaluation of these composite functions mandates that they have direct access to their constituent's `apply` methods. It is for this reason that the `apply` methods of the various function classes are protected, rather than private: for reasons explained before, it is essential that these not be publicly accessible, but efficient composition demands limited access across class boundaries. For compositions within one function class hierarchy, the protected access mode does the right thing. For the composition of scalar- and vector-valued functions, a friendship designation is necessary.

Despite the provision of direct access to `apply`, several inefficiencies are still built into these constructions. Temporaries to hold intermediate results are an inevitable by-product of this approach, as are multiple visits to data items where carefully optimized implementations would require only one, or at least fewer. Thus the composite function constructions in RVL may be more useful for rapid prototyping than for production code construction, as they do not take advantage of implicit opportunities for data motion optimization and loop fusion. Note that the RVL composite function classes implement deferred evaluation - but at too high a level of abstraction to expose optimization opportunities. Despite these missed opportunities, we have found that for "large scale" applications (in the sense explained in the introduction), the performance penalty exacted by our composite constructions are usually not particularly onerous.

EXAMPLES

Newton's method for complex polynomials

Newton's method is an iteration defined for a differentiable map $F : X \to X$ of a Banach space X over a field $\mathbf{E}$: it produces a sequence $\mathbf{x}_i$ defined by its initial member $\mathbf{x}_0$ and the rule

$$\mathbf{x}_{i+1} = \mathbf{x}_i - DF(\mathbf{x}_i)^{-1} F(\mathbf{x}_i).$$

When the sequence $\mathbf{x}_i$ converges to a point at which the derivative DF is nonsingular, the limit is necessarily a root of the function F.

Newton's method is a test example for the Algorithm package. The RVL implementation uses a subclass of `LinearOp` which supplies the action of the inverse derivative:

```
template<class Scalar>
class LinearOpWithInverse: public LinearOp<Scalar>,
                                public Invertible<Scalar> {
...
public:
  const LinearOp<Scalar> & getInv() const {...}
  const LinearOp<Scalar> & getAdjInv() const {...}
}
```

`Invertible` is a mixin interface which adds appropriate protected `apply` methods to implement the inverse and adjoint inverse operators. For Newton's method proper, the adjoint operators are not necessary, but since RVL is dedicated to calculus in Hilbert space, they must be supplied anyway, for reasons already explained.

An `OperatorWithInvertibleDeriv` naturally supplies a `LinearOpPairWithInverse` as the return value of its `getDeriv()` method. The RVL-Algorithm implementation of Newton's method looks like this:

```
NewtonSolverStep( OperatorEvaluation<Scalar> & opeval ) {...}

bool NewtonSolverStep<Scalar>::run() {
  Vector<Scalar> dx(opeval.getDomain());
  const Vector<Scalar> & Fx = opeval.getValue();
  const RVL::LinearOpWithInverse<Scalar> & DF =
      dynamic_cast<const RVL::LinearOpWithInverse<Scalar> &>
        (opeval.getDeriv());
  DF.getInv().apply(Fx, dx);
  Scalar one = ScalarFieldTraits<Scalar>::One();
  x.linComb(-one,dx);
}
```

We've tried this out with a *scalar polynomial map*, i.e.

$$F(\mathbf{x})_j = p(x_j),\ j = 1, ..., n$$

in which p is an ordinary polynomial of a scalar variable. Implementation of a `Functional` encapsulating this map is straightforward in terms of a `DataContainer` class that exposes its data, and compatible `FunctionObject`s.

The choice of scala field is completely open: to emphasize that it is possible to work with complex arithmetic in RVL, we chose $\mathbf{E} = \mathbf{C}$ for this application, represented computationally by `std::complex<double>`. The polynomial used in the example is quintic:

$$p(z) = z^5 - 0.84z^3 - 0.16z$$

Its roots are $0, \pm 1, \pm 4i$. The test code sets the dimension $n = 10$, and initializes the complex 10-vector $\mathbf{x}$ with pseudorandom numbers by evaluating an appropriate `FunctionObject`. The method terminates when the norm of the vector $F(\mathbf{x_i})$ drops below a specified tolerance. All of the components of $\mathbf{x}_i$ are then close to the roots of p, and the usual quadratic convergence of Newton's method is observed.

Of course the same computation could be performed for real polynomials with very little change in the components - essentially just a change of template parameter in the driver source. In particular, the Newton solver code (like other RVL-Algorithm classes) requires no change whatsoever to change the field, as it is a template parametrized by field type.

Seismic Velocity Analysis

One of the major steps in the standard industrial seismic processing stream is a process called velocity analysis, in which collections of time series are subjected to parametrized changes of variable. The object is to align the oscillations within the time series and so reveal their coherence. The parameters in the change of variables are interpreted as functions of seismic wave velocities, so are themselves physically meaningful. The comprehensive reference (Yilmaz, 2001) explains this and many other aspects of industrial seismic data processing.

Contemporary practice partly automates velocity analysis, but still requires considerable manual intervention. One of the authors (WWS) has proposed an objective approach to this task (and related, more complex tasks) which turns it into an optimization problem with regular objective (Symes, 1986; Symes, 1998). This objective involves differencing pairs of time series after changes of variable, and the formation of the mean square of the results:

$$J[\mathbf{v}] = \frac{1}{2} \sum_{i,t_0} |d_{i+1}(\tau_{i+1}[\mathbf{v}](t_0)) - d_i(\tau_i[\mathbf{v}](t_0))|^2$$

The change of variable τ_i depends both on the time series d_i, or rather on its metadata (data acquisition geometry and other attributes), and on the vector $\mathbf{v}$ of velocity parameters. Approximation of a change of variable for discrete data requires interpolation. Local cubic interpolation proves to be sufficiently regular to yield reliable derivatives. The gradient $\nabla_{\mathbf{v}} J$ must also be computed.

We have implemented this algorithm using the RVL framework. The chief task, as in other optimization applications of RVL, is construction of a `Functional` implementing $J[\mathbf{v}]$ and its derivatives up to order 2. Because seismic data sets tend to be very large - typical numbers of time series in one such data set range from tens of thousands to several million, each time series having several thousand samples - out-of-core construction is mandatory. Moreover, the metadata already alluded to is essential to proper handling of the data. To encapsulate both data and metadata, we used a standard seismic data exchange structure, the *SEGY trace* (Barry et al., 1980), as implemented in the Seismic Unix ("SU") library of data processing tools (Cohen and Stockwell, 2004). This public domain library offers a wide variety of i/o and processing functions carefully coded in C

and based on the SEGY standard. It is widely used around the world in both academic and industrial settings, and is a *de facto* standard open source data processing package.

We wrapped the SU SEGY trace i/o functions in `RecordServer` classes. These templates have `get` and `put` methods abstracting reads and writes of the template type, in this case the `segy` struct defined in SU. The computation of J (and its gradient, which in this case might as well be computed at the same time) proceeds by setting a random access specialization of `RecordServer` (holding the data in a disk file) at start-of-file. Then `get` is called until it fails, signaling EOF. On each successive pair of traces read the computation described above is performed, and the result accumulated in J. The computation naturally resets itself when the begins and ends of data subsets are encountered at the boundaries of which signal coherency is expected to fail. These boundaries are flagged by the trace metadata.

The velocity is a physical field, with extent and distribution in space. To reflect this reality, the parameter vector $\mathbf{v}$ is represented in our code by a `Vector` in a `GridSpace`, a `Space` subclass based on the `GridData` subclass of `DataContainer`. The domain of the `Functional` representing J is thus a `GridSpace`. Metadata carried along in a `GridData` object (and the `Vector` that owns it) describes a regular grid in space (of dimensions 1, 2, or 3 in this application). The `GridSpace::inner` function uses this metadata to properly scale the inner product, for example, so that the computed gradient of J is stable against changes in sampling.

To assess the overhead of calls through the RVL virtual functions, exception tracebacks, and so on, we constructed a command from the components used in the computation of J, which simply performs the change of variable and outputs the results. To make a convenient-to-use command in the "Unix filter" style of SU, we constructed a `StreamServer` specialization of `RecordServer` which takes data from `stdin` and writes to `stdout`, and use this as the data source/sink.

The function of this command is exactly the same as the SU command `sunmo`, with even roughly the same approach to implementation of changes of variable via local polynomial interpolation. We applied both the RVL and SU commands to process a data set consisting of 24000 traces each with 1250 samples (a total data volume, with metadata, of 126 MB). This set, small by contemporary standards, required 28 s for either command to run on an Apple Macintosh Powerbook (1 GHz G4 CPU, OS-X 3, gcc 3.4.3). To make sure that we were not seeing merely i/o, we constructed a command that did just that, copying the input stream to the output stream. This command required 22 s to execute on the same data, leading us to conclude that slightly more than 20% of the execution time in both RVL and SU change-of-variable commands was devoted to floating point operations, the remainder to i/o. This comparison suggests that, at least in this case, any overhead imposed by the RVL implementation is insignificant.

The computation of J and its gradient are considerably heavier in floating point arithmetic than the mere change of variable, so that optimization using the RVL-Algorithm implementation of Limited Memory BFGS (described above) is less i/o-bound. Achieving a reduction in gradient length of 10^{-2} for a very small data set of 2150 traces of

750 samples each required about 15 s to execute 12 quasi-Newton steps, again using the Powerbook G4. It is impossible to compare this optimization to a pure SU (procedural) implementation; for example, SU does not provide a derivative computation for its change--of-variables operator. However we believe that this algorithm, as we have implemented it, is on par with contemporary industrial solutions in speed and reliability.

Of course the RVL-Algorithm optimization code required no change whatsoever to accommodate out-of-core evaluation of the objective function. All such details are hidden well behind the `Functional - Vector` interface. We have also used the `RecordServer` classes to construct an out-of-core `DataContainer` subclass for SEGY seismic data and a corresponding `Space` class. We have used these in other applications in which seismic data plays a vector role, such as least squares model-based data fitting and sensitivity estimation. These applications also required no change whatever in the RVL-Algorithm code expressing optimization and iterative linear algebra algorithms.

Source Estimation for Steady State Diffusion

Coordinate–free algorithmic code should function correctly in either serial or parallel computing environments - that is, should not require *any* modification in a port from serial to parallel. As proof that the RVL classes can be used to produce code which is reusable in this sense, we used an RVL - Algorithm implementation of the LBFGS quasi-Newton algorithm, described in part in the preceding sections, to solve an advection–diffusion optimal control problem in parallel.

This implementation uses parallel linear algebra tools from the Trilinos collection (Heroux et al., 2003), adapting their interfaces to create RVL subclasses. We use Epetra (Heroux, 2003) to define distributed `DataContainer`s, and AztecOO (Heroux, 2004) parallel linear solvers and preconditioners to create `Functional` objects. This leveraging of other packages amounts to yet another sort of code reuse, possible because Epetra and AztecOO have compatible interfaces and because RVL classes, being abstract, do not define conflicting interfaces. Such interoperability was another goal of the RVL project, as mentioned in the introduction, and this example supports the conclusion that the design achieves it, at least to some extent.

After a brief description of the physics of the advection–diffusion problem, we outline the discretization approach and the conversion of the optimal control problem to an unconstrained optimization problem, and overview the construction of an `Functional` using Epetra and AztecOO. Performance of the code on a medium-sized cluster indicates that parallel execution proceeds as expected - even though *the abstract algorithm code, expressed in RVL, is exactly the same as that used in serial solution of the same problem!*

Advection–Diffusion Problem and Galerkin Discretization. The model used in this example approximates the physics of planar steady-state passive transport and diffusion of a substance with concentration $y(x)$ through a domain $\Omega \subset \mathbf{R}^2$, in a fluid with

velocity field $c(x)$:

$$\begin{aligned} -\epsilon\Delta y(x) + \mathbf{c}(x)\cdot\nabla y(x) + r(x)y(x) &= f(x) + u(x), \quad x \in \Omega \\ y(x) &= d(x), \quad x \in \partial\Omega_d \\ \epsilon\tfrac{\partial}{\partial \mathbf{n}} y(x) &= g(x), \quad x \in \partial\Omega_n \end{aligned}$$

where $\partial\Omega_d \cap \partial\Omega_n = \emptyset$, $\partial\Omega_d \cup \partial\Omega_n = \partial\Omega$, $\mathbf{c}$, d, f, g, r are prescribed, $\epsilon < 0$ is a given scalar, and $\mathbf{n}$ denotes the outward unit normal (Bartlett et al., 2005). The diffusion constant ϵ is positive, and the boundary terms restrict either the concentration or the flux at positions on the boundary of the domain. The RHS summand $u(x)$ represents the source of pollutant, an is to be estimated from measurements of $y(x)$. For further discussion of this problem, see Section 8 in (Quateroni and Valli, 1994) and Section 9 in (Knabner and Angermann, 2003).

The 2D domain Ω used in this example models an airport terminal Figure 4. The boundary set $\partial\Omega_n$ represents vents in the climate–control system of the airport and I impose a 0 concentration on the rest of the boundary $\partial\Omega_d$. The diffusion term is set $\epsilon = 1e-4$.

Somewhat unrealistically, we assume that concentraion measurements $\hat{y}(x)$ are supplied for every point $x \in \Omega$. Inference of the source u may be accomplished by minimizing the mean square prediction error:

$$\min_{y,u} \frac{1}{2}\int_\Omega (y(x) - \hat{y}(x))^2 dx + \frac{\alpha}{2}\int_\Omega u^2(x) dx$$

for some given data $\hat{y}(x)$, subject to the steady–state advection–diffusion constraint ().

Finite–element discretization of advection–diffusion constrained problem leads to a finite dimensional version of the optimization problem (**??**):

$$\begin{array}{rl} \min_{\mathbf{y},\mathbf{u}} & \frac{1}{2}(\mathbf{y}-\hat{\mathbf{y}})^T Q(\mathbf{y}-\hat{\mathbf{y}}) + \frac{\alpha}{2}\mathbf{u}^T R\mathbf{u} \\ \text{s.t.} & A\mathbf{y} - B\mathbf{u} - b = 0 \end{array} \tag{2}$$

R is the discretization of either the identity operator (L^2 Tikhonov regularization) or the Laplace operator (H^1 seminorm regularization).

Assuming A is nonsingular, a unique feasible point $\mathbf{y}(\mathbf{u})$ exists for each $\mathbf{u}$, solving

$$A\mathbf{y}(\mathbf{u}) - B\mathbf{u} - b = 0$$

Thus the constrained optimization problem (2) is equivalent to the unconstrained problem of minimizing

$$F(\mathbf{u}) = \frac{1}{2}(A^{-1}(B\mathbf{u}+\mathbf{b}) - \hat{\mathbf{y}})^T Q(A^{-1}(B\mathbf{u}+\mathbf{b}) - \hat{\mathbf{y}}) + \frac{\alpha}{2}\mathbf{u}^T R\mathbf{u} \tag{3}$$

A Newton-related algorithm for the solution of (3) requires the gradient of F, given by

$$\nabla_{\mathbf{u}} F(\mathbf{u}) = B^T A^{-T} Q A^{-1}(B\mathbf{u}+\mathbf{b}) + \alpha R\mathbf{u}$$

Computation of the discretized functional and gradient involves the application of linear operators, linear–system solves, and some vector algebra.

Implementing Parallel Data Structures and Matrices using Epetra. The arrays A, B, Q, b and R must be generated and stored, as they are used throughout the solution of this problem. The Epetra library from the Trilinos collection (Heroux, 2003) provides distributed array implementations within an object framework, useful both for internal storage of arrays used in function and gradient evaluation, and for the internals of suitable `DataContainer` and `Space` subclasses for this application. An ad-hoc "problem holder" container class constructs three Epetra matrices and one Epetra vector, using an `Epetra_Comm` object passed to its constructor by the `Functional` constructor described below. `Epetra_Comm` objects encapsulate data layout and communication information used by other Epetra components built by the problem holder. This `AdvDiffProblemHolder` also constructs the domain/range space of the linear system as an object of the `EpetraMultiVectorSpace` subclass of `RVL::Space`, which contains a virtual constructor for the corresponding `DataContainer` subclass. Details are described in Appendix B.

To populate the finite element matrices $A, B, ...$ we made use of the Trilcode package developed by Denis Ridzal (Bartlett et al., 2005). This package fetches mesh information from a file (created by yet another external meshing package), creates element matrices using the mesh data, assembles them into global stiffness matrices and the like, and stores them in the Epetra array objects supplied by the `AdvDiffProblemHolder`.

A `Functional` subclass defines computationally the optimization problem (3). Construction of this `AdvDiffFunctional` follows the Facade pattern described above (the class `StdFunctional`), which combines `FunctionObject`s and `FunctionObjectRedn`s to produce the behaviour of a `Functional`. The `FunctionObjectRedn` defining the function evaluation and the `FunctionObject`s defining the gradient and Hessian share a reference to a common `AdvDiffProblemHolder`. The `AdvDiffFunctional` owns the `AdvDiffProblemHolder`, constructing it using an `Epetra_Comm` passed from the driver, and builds the `FunctionObjects` as they are needed. The functional also stores the computed value of the state $y(u)$ to avoid recomputing it.

Implementing Function Objects and Functionals using AztecOO. The bulk of the computational work involved in evaluation of these function objects goes into solution of linear systems: computation of the feasible point $y(u) = A^{-1}(Bu + b)$, and of $z = A^{-T}y(u)$ in the gradient calculation. We adapted the AztecOO linear solver package to accomplish this task.

AztecOO is a package of object oriented interfaces to the Aztec solver library (Heroux, 2004), designed to work with Epetra data objects. Aztec is a linear solver framework, aimed a parallel solution of large sparse systems. From the extensive list of solvers and preconditioners offered by AztecOO, we chose to use GMRES in conjunction with one of three preconditioners: Jacobi, Neumann, and additive Schwarz. See (Tuminaro et al., 1999) for description and references.

The datum required to instantiate an AztecOO `Solver` is a "problem" object, in this case a `Epetra_LinearProblem` encapsulating Epetra representations of the matrix A, right–hand side vector b, and a solution vector x of a linear system,. Creation and

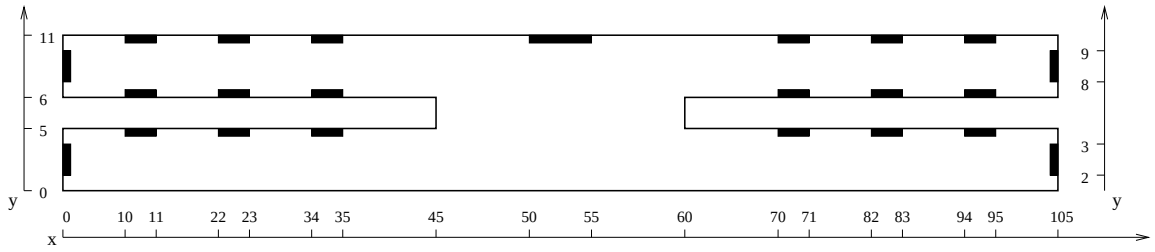

FIG. 4. Airport model obtained from Sandia National Labs.

invocation of an AztecOO solver requires only a few lines of code:

```
AztecOO Solver(Problem);
Solver.SetAztecOption( AZ_output, AZ_warnings);
Solver.SetAztecOption( AZ_solver, AZ_gmres );
Solver.SetAztecOption( AZ_precond, AZ_Jacobi );
int res;
res = Solver.Iterate(1000,1e-12);
```

Computational Results The driver code for this exercise

1. constructed a `Functional` by following the steps described in the last two paragraphs,
2. instantiated an LBFGS object from the RVL-Algorithm package, taking this `Functional` as input to the constructor, and
3. invoked the LBFGS object's `run` method.

The driver program was written SPMD style: the same code, including the RVL-Algorithm code, ran on all processes. The RVL-Algorithm implementation of LBFGS was precisely the code described in the preceding sections of this paper. It was not altered in any way - not a single character was changed in any header file or implementation file - to adapt it to SPMD execution.

The experiments reported here are based on the two-dimensional model of airflow in a two-story airport terminal building, shown in Figure 4. The figure shows a roughly 'H' shaped cross-section of the building's interior volume, with a large open well connecting the two floors. Several air in- and outlets serve as sources and sinks for the flow. A steady-state velocity field is determined for this configuration and used as input to the convection-diffusion problem presented above.

Discretization by P1 finite elements gave a mesh with 1654965 elements and 832510 nodes. Roughly 1 GB of mesh data was stored on disk and read in as needed. For the computational experiments reported below, the mesh is partitioned into 16, 32, and

64 subdomains, A natural domain decomposition scheme was implemented using Epetra utilities.

The platform for the parallel execution tests was the Rice Terascale Cluster, consisting of 272 Intel Itanium 2 64 bit processors, each running at roughly 900 MHz. Of several interconnect possibilities, we chose to use Myrinet for its relatively low latency.

We tested all combinations of $16, 32, 64$ processors, $10^{-2}, 10^{-4}, 10^{-8}$ GMRES convergence tolerances, and three different preconditioners. The tolerances are given to AztecOO, which will stop GMRES when $\|r\|/\|r_0\| < tol$ or a specified maximum number of iterations have been run (we set this maximum at 1000). Occasionally, the 10^{-8} tolerance could not be satisfied in the maximum permitted number iterations. In that case, the approximate GMRES solution was accepted anyway.

The three preconditioners (provided by AztecOO) and default parameter selections were:

- k–step Jacobi, $k = 3$.
- Neumann series polynomial of order k, $k = 3$.
- An additive Schwarz preconditioner, tailored to domain decomposition problems; each processor approximately solves the local subsystem using Saad's ILUT.

Figure 5 demonstrates close to linear speedup, except for the cases using the additive Schwarz preconditioner, which performs so well that the communication overhead destroys the expected speedup. These are precisely the results one would expect. Thus the performance of the application is dominated by the usual factors which regulate parallel computation. Regardless of these factors, the optimization algorithm can use the functional values and gradients it gets to find a local minimum, which in this case must be global since the problem to be solved is a convex quadratic program. Virtual function call overhead imposed by the RVL layer of the application appears to have no significant effect on performance.

PROGRAMMING NOTES

RVL proper consists entirely of header files. The same is true for LocalRVL, described in Appendix A. Most of the classes composing RVL are actually class templates, and most build systems require template definitions to be supplied in headers. Thus installation of RVL and LocalRVL amounts to unpacking the tarballs. Both packages contain Doxygen-generated documentation in HTML format, providing syntax and usage details about every method of every class, and inheritance diagrams.

A small unit test suite is also available, which may be useful as a preliminary exercise in installation of RVL applications.

RVL "calculus" classes - `Space`, `Vector`, `Functional`, `Operator`, `LinearOp`, and the Evaluation classes - should not be allocated dynamically, both because no compelling

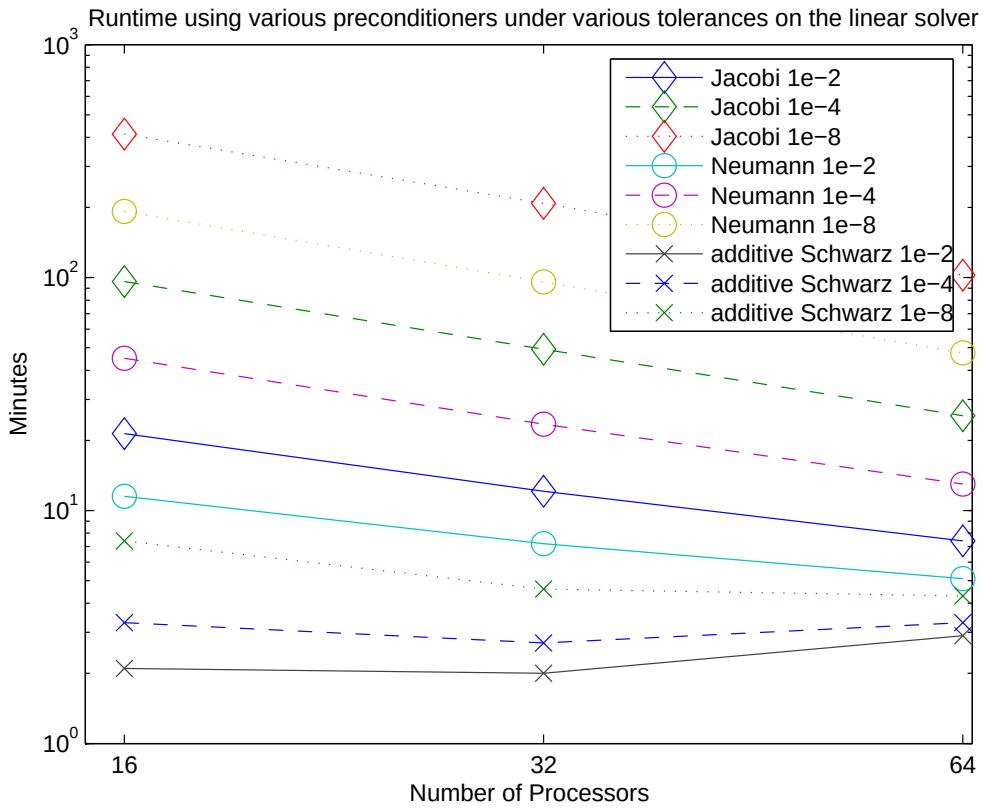

FIG. 5. Runtimes using various preconditioners, GMRES residual tolerance $= 10^{-8}$.

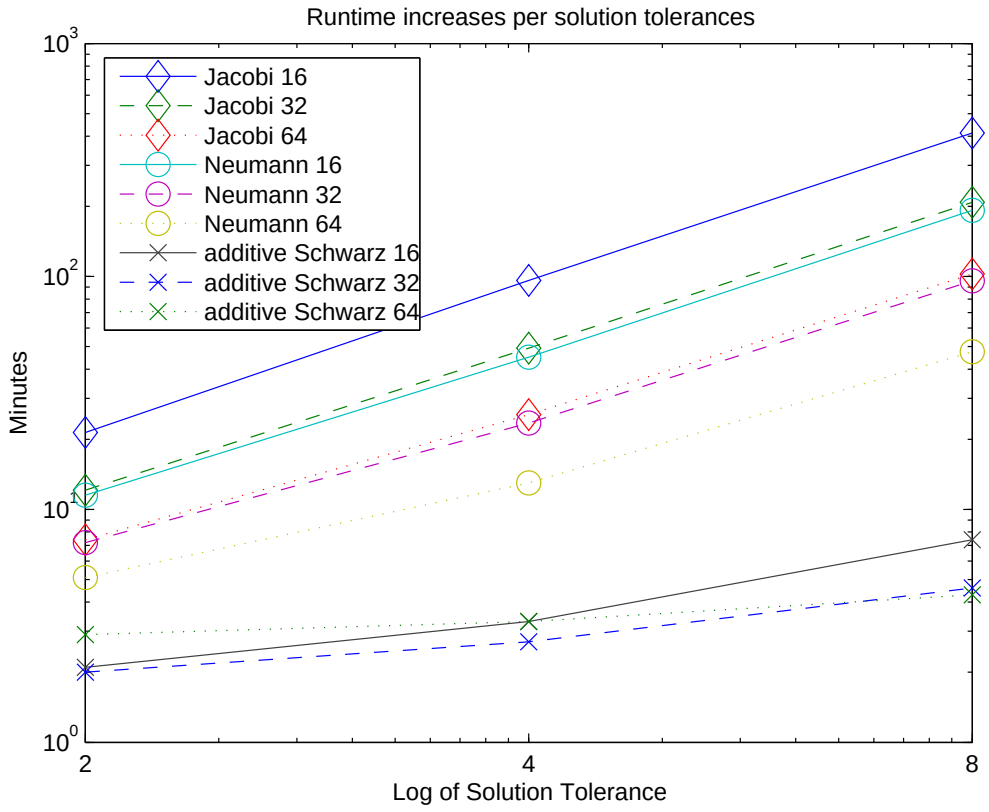

FIG. 6. Runtime as function of GMRES residual tolerance.

reason can exist to do so and because dynamic allocation implicitly contradicts the design principles of RVL. In fact, dynamic allocation of these types, and implicitly of their subtypes, is disallowed: we have made `operator new` private in these classes. This design decision is not unique to RVL (see for instance (Meyers, 1996)), but it is certainly unusual, and some explanation seems in order.

Use cases which might seem at first glance to suggest dynamic allocation of `Space`, `Functional`, `Operator`, or `LinearOp` subclass instances are either incompatible with the mathematics-derived semantics of these classes, or already covered by derived types. For example, a tuple of vector spaces, implemented as a `std::vector<Space<Scalar> *>` the components of which might be allocated on the heap, might at first blush seem useful. However `std::vector` realizes the semantics of Cartesian product only in the set-theoretic sense A tuple of spaces has *in itself* no meaning in the context of linear algebra and calculus. A Cartesian product of spaces certainly does have mathematical sense, but is not merely a Cartesian product in the sense of sets: it also inherits linear algebra operations in a canonical way. RVL provides the `ProductSpace` class to accommodate this abstraction.

The concrete core classes (`Vector`, the Evaluations) have constructors, and do not require virtual construction. Each of them is logically dependent on other types (a vector cannot exist in the absence of the space of which it is a member) and this dependence is realized through the scope rules.

The reader will have noticed that abstract `apply...` methods of the function classes, which must be implemented in any child class, are protected hence not accessible in application code. This might seem strange - put a lot of effort into an implementation that can't be used directly. However, permitting public access to `apply...` would either force on the user responsibility for implementing the Observer pattern relating `Vector` and the Evaluation objects, or prevent implementation of this pattern altogether. With `apply...` methods available only via the value access methods of Evaluations, the dynamic dependence of the Evaluation on the evaluation point can be relegated to the base classes.

One other peculiar feature of RVL deserves comment. Most other OO numerics libraries constrain their space objects (or more usually their vector objects, as most such libraries provide no explicit space representation) to report a dimension. RVL does not provide this information at the level of `Space`, for a straightforward reason: it is not always possible, and we reject base class interface features which cannot be realized in all useful instances. To demonstrate the infeasibility of assigning a dimension to every space representable as an `RVL::Space`, we have created a `Space` subclass `PolynomialSpace`, which realizes the vector space of polynomials (or, equally well, finite sequences) over a field. Any two of these can be added, and even multiplied, or multiplied by a scalar, to produce another. They form a vector space, and any vector lies in a finite dimensional subspace (a trivial statement). However the vector space represented by `PolynomialSpace` is infinite dimensional, i.e. has no dimension, even though it is computationally representable. Q.E.D.

CONCLUSION

The central goal of the RVL project is: to create a class hierarchy that mimics as closely as possible the basic concepts of calculus in Hilbert space, while deferring many implementation details as possible to subclasses separate from this hierarchy. It should be possible to express coordinate free algorithms of linear algebra and optimization entirely in such a type system, without reference to the deferred details. Moreover, the algorithms so expressed should be be reusable across a wide variety of applications, data storage modes (core, disk, network...) and execution strategies (serial, client-server, SPMD,...). By "reusable" we mean *without any alteration of source code whatsoever*.

We have demonstrated a feasible design for a C++ class library achieving these goals. Typical algorithms and applications illustrate the reuse of algorithm code which is the project's chief aim. We have implemented relatively simple but representative algorithms of the target class, and reused them in a variety of applications and computing environments. The programming style is a simple as possible: each line of code mimics a corresponding statement in a typical mathematical algorithm description, and the essential business of memory management, data access, and function implementation is hidden from view. Interfaces behind which to hide these details are provided in a canonical and minimal way. We believe that this design approximates conformance to the Einsteinian dictum: "...as simple as possible, but no simpler".

RVL does not pretend applicability beyond the semantic scope specified in the goal statement. For example, it does not model calculus in *Banach* space. The `norm` function supplied in the vector base class is intended to represent a Hilbert norm, derived from an inner product. Other norms may be provided, of course - they are amongst the variety of functions which may be hidden behind the `FunctionObject` interface. However these are not built into the fabric of `Space`, which is a computational token for Hilbert space. We make this restriction for one reason alone: calculus in Hilbert space is the abstract framework upon which nonlinear programming is founded.

A more interesting extension might be to functions of a discrete variable and integer programming. We have experimented with such an extension, and there seems to be in principle no obstacle. Interior point algorithms for linear programming can certainly be expressed in RVL; we have implemented a primal IP algorithm in several variants. On the other hand the simplex method appears to resemble dense matrix linear algebra algorithms: it can be encapsulated in an RVL-like object but is not amenable to useful expression in coordinate-free fashion. It remains to be seen whether concepts such as convex relaxation, branch-and-bound, cutting planes, etc. are sufficiently abstract to benefit from RVL-like treatment. Since many applications of simulation driven optimization actually have significant discrete aspects, an extension encompassing mixed integer-nonlinear programs could conceivably be very useful.

Acknowledgements

This work was supported in part by National Science Foundation grants DMS-9973423, DMS-9973308, and EAR-9977697, by the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23, by the Department of Energy EMSP grant DE-FG07-97 ER14827, by ExxonMobil Upstream Research Co., and by The Rice Inversion Project (TRIP). TRIP sponsors for 2003 were Amerada Hess Corp., Conoco Inc., Landmark Graphics Corp., Sensorwise Inc., Shell International Research, and Western Geco.

REFERENCES

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. (1992). *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia.

Barry, K., Cavers, D., and Kneale, C. (1980). SEG-Y - recommended standards for digital tape formats. In *Digital Tape Standards*. Society of Exploration Geophysicists, Tulsa.

Bartlett, R., Heinkenschloss, M., Ridzal, D., and van Bloemen Waanders, B. (2005). Domain decomposition methods for advection dominated linear–quadratic elliptic optimal controlproblems. Technical report, Sandia National Laboratories.

Bartlett, R. A. (2003). MOOCHO: Multifunctional Object-Oriented arCHitecture for Optimization, User's Guide. Technical report, Sandia National Laboratory, Albuquerque, NM.

Bartlett, R. A., Heroux, M. A., and Long, K. R. (2003). TSFCore 1.0: A package of light-weight object-oriented abstractions for the development of abstract numerical algorithms and interfacing to linear algebra libraries and applications. Technical report, Sandia National Laboratories, Albuquerque, NM.

Bartlett, R. A., Van Bloemen Waanders, B. G., and Heroux, M. A. (2004). Vector reduction/transformation operators. *ACM Transactions on Mathematical Software*, 30(1):62–85.

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/`.

Cohen, J. K. and Stockwell, J. J. W. (2004). CWP/SU: Seismic Unix release no.37: a free package for seismic research and processing. Center for Wave Phenomena, Colorado School of Mines.

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

Douglas, C., George, D., and Henderson, M. (1994). Object classes for numerical analysis. In *OON-SKI '94*, pages 32–49. Proceedings of the Second Annual Object-Oriented Numerics Conference.

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

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.

Heroux, M. A. (2003). Epetra home page. http://software.sandia.gov/trilinos/packages/-epetra/.

Heroux, M. A. (2004). *AztecOO User Guide*. Sandia National Laboratories.

Heroux, M. A., Barth, T., Day, D., Hoekstra, R., Lehoucq, R., Long, K., Pawlowski, R., Tuminaro, R., and Williams, A. (2003). Trilinos: object-oriented, high-performance parallel solver ligraries for the solution of large-scale complex multi-physics engineering and scientific applications. Technical report, Sandia National Laboratories, Albuquerque, NM.

Hoffman, K. and Kunze, R. (1961). *Linear Algebra*. Prentice-Hall, Inc., Englewood Cliffs, N. J.

ISIS Development Team (1997). ISIS++: Iterative scalable implicit solver (in C++). Technical report, Sandia National Laboratories, `z.ca.sandia.gov/isis/`.

Karmesin, S. (2000). POOMA: Parallel object oriented methods and applications. Technical report, Los Alamos National Laboratory, `www.acl.lanl.gov/pooma/`.

Knabner, P. and Angermann, L. (2003). *Numerical Methods for Partial Differential Equations*. Texts in Applied Mathematics, Vol. 44. Springer–Verlag, Berlin, Heidelberg, New York.

Kolda, T. and Pawlowski, R. (2003). NOX: An object-oriented, nonlinear solver package. Technical report, Sandia National Laboratories, Livermore, CA.

Langtangen, H. P. (1999). *Computational Partial Differential Equations: numerical methods and Diffpack programming*. Springer-Velag, New York, Berlin, Heidelberg.

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

Martin, R. C., Riehle, D., and Buschmann, F. (1998). *Pattern Languages of Program Design 3*. Addison-Wesley.

Meyers, S. (1996). *More Effective C++*. Addison-Wesley, New York.

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

Myers, N. C. (1995). Traits: a new and useful template technique. C++ Report, `http://www.cantrip.org/traits.html`.

Nichols, D., Dunbar, G., and Claerbout, J. (1993). The C++ language in physical science. In *OON-SKI '93*, pages 339–353. Proceedings of the First Annual Object-Oriented Numerics Conference.

Nocedal, J. and Wright, S. (1999). *Numerical Optimization*. Springer Verlag, New York.

Padula, A. D. (2004). Object–oriented algorithm design for scientific computing. Technical Report 04-XX, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, USA.

Pozo, R. (2004). Template numerical toolkit: home page. `math.nist.gov/tnt`.

Quateroni, A. and Valli, A. (1994). *Numerical Approximation of Partial Differential Equations*. Springer, Berlin, Heidelberg, New York.

Symes, W. (1986). Stability and instability results for inverse problems in several-dimensional wave propagation. In Glowinski, R. and Lions, J., editors, *Proc. 7th International Conference on Computing Methods in Applied Science and Engineering*. North-Holland.

Symes, W. W. (1998). High frequency asymptotics, differential semblance, and velocity analysis. In *Expanded Abstracts*, pages 1616–1619, Tulsa. Society of Exploration Geophysicists.

Symes, W. W. and Padula, A. D. (2005). Rice vector library home page. `http://www.trip.caam.rice.edu/txt/tripinfo/rvl.html`.

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.

Tech-X (2001). OptSolve++. Technical report, Tech-X Corporation, `www.techxhome.com/products/optsolve/index.html`.

Tisdale, E. R. (1999). The C++ scalar, vector, matrix, and tensor class library standard page. `www.netwood.net/ edwin/svmt/`.

Tuminaro, R., Heroux, M., Hutchinson, S. A., and Shadid, J. (1999). *Official Aztec User's Guide: Version 2.1*. Sandia National Laboratories.

Veldhuizen, T. L. (1999). Blitz++ home page. `www.oonumerics.org/blitz`.

Yilmaz, O. (2001). Seismic data processing. In *Investigations in Geophysics No. 10*. Society of Exploration Geophysicists, Tulsa.

APPENDIX A: Using `LocalRVL`

As mentioned in the introduction, RVL itself defines no mechanism to access data, leaving this task to subclasses of `DataContainer`, `FunctionObject`, and `FunctionObjectRedn`. In this appendix we describe briefly a simple, portable collection of classes which completes the Visitor hierarchies in RVL. We have used this LocalRVL package to implement most of the applications described in this paper, and a number of others.

The simplest data access mode which can be made pure virtual yet incur negligible performance penalty for large data sets is exposure of a pointer to data. The `LocalDataContainer` class template provides natural access methods for arrays of `DataType` elements:

```
int LocalDataContainer<DataType>::getSize() const;
DataType * LocalDataContainer<DataType>::getData();
DataType const * LocalDataContainer<DataType>::getData() const;
```

These are pure virtual in the base class `LocalDataContainer`, to give freedom of implementation but provide an interface for definition of general `FunctionObject` and `FunctionObjectRedn` subclasses which access data by pointer. A very simple `RnArray` concrete subclass stores an array of `DataType`s, and serves as the base class for other concrete `LocalDataContainer` subclasses which add various types of metadata and access to it, such as `GridData` for regular grids. Nonstandard implementations, not specializing `RnArray`, include the SEGY and Epetra `DataContainer` classes mentioned in the Examples section.

As is typical for Acyclic Visitor "concrete Element" classes, `LocalDataContainer` evaluates only compatible subtypes of `FunctionObject` and `FunctionObjectRedn`, tested via RTTI. For technical reasons we specify the interaction through so-called mixin interfaces, `LocalEvaluation` and `LocalReduction`. The `LocalDataContainer::eval` method downcasts its `DataContainer` arguments to `LocalDataContainer`s, its `FunctionObject` argument to a `LocalEvaluation` (or its `FunctionObjectRedn` argument to a `LocalReduction`), and invokes the evaluation method of the function object.

`LocalFunctionObject`s and `LocalFunctionObjectRedn`s inherit the `FunctionObject` and `LocalEvaluation`, respectively `FunctionObjectRedn` and `LocalReduction`, interfaces. The evaluation method is an overload of `operator()` in both cases. For example, the generic `LocalEvaluation` evaluation method is

```
void LocalEvaluation<DataType>::operator()
  (LocalDataContainer<DataType> & target,
   std::vector< LocalDataContainer<DataType> const *> sources);
```

The similar method for `LocalReduction` lacks the first argument. Both are, of course, pure virtual.

This interface is flexible but somewhat painful to use. LocalRVL supplies restricted classes of `LocalEvaluations` and `LocalReductions` with specified numbers of arguments, and the overwhelming majority of useful function objects conform to these interfaces. For example, the `BinaryLocalEvaluation` mixin includes this specialized pure virtual evaluation method:

```
void BinaryLocalEvaluation<DataType>::operator()
  (LocalDataContainer<DataType> & target,
   LocalDataContainer<DataType> const & source);
```

and implements the generic `operator()` method by delegation to the specialized one.

The `BinaryLocalFunctionObject` class inherits from `FunctionObject` and `BinaryLocalEvaluation`. Since the generic `operator()` is implemented in the latter class, only

the specialized binary `operator()` need be implemented in a concrete child. For example, a linear combination `BinaryLocalFunctionObject`, used to implement a concrete `LinearAlgebraPackage`, has the obvious (stripped-down) `operator()` implementation

```
void operator( )(LocalDataContainer<DataType> & y,
                 LocalDataContainer<DataType> const & x) {
   int n = y.getSize();
   // check size consistency
   for (int i=0;i<n;i++) {
     y.getData()[i]=a*x.getData()[i]+b*y.getData()[i];
   }
 }
```

the scalars `a` and `b` being set in the constructor. A natural LocalRVL specialization of `LinearAlgebraPackage` uses essentially this `FunctionObject` to define linear combination for any `Space` whose `DataContainer` type is either a `LocalDataContainer` or a recursive composite of `LocalDataContainer`s. Another example is the `RVLRandomize` class mentioned in several examples above, which is a `UnaryLocalFunctionObject`.

APPENDIX B: Adapting Epetra to RVL

Our treatment of the advection-diffusion inverse problem as a parallel SPMD application used Epetra, a library of parallel linear algebra types developed at Sandia National Laboratory and now part of the Trilinos collection (Heroux, 2003). We were able to write adaptor code to access the facilities of Epetra objects within RVL objects, which greatly eased the construction of this application.

The principal adaptation targets are `Epetra_Vector` and `Epetra_MultiVector` (Epetra does not define a namespace, but uses the prefix `Epetra_` to signify membership). `Epetra_Vector` exposes its data by pointer and array size, as regulated by an `Epetra_Comm` object: the pointer and size returned in any process refer to the part of the object's data stored locally, on the processor in question. Thus it was trivial to construct an `EpetraVectorLDC` subclass of `LocalDataContainer<double>` (`double` being the only real type admitted by Epetra), by making an `Epetra_Vector` a data member and delegating to its services. We also exposed a reference to the `Epetra_Vector` member, for reasons to be explained shortly.

Logically, an `Epetra_MultiVector` object is an array of `Epetra_Vector` objects of identical size. It is therefore easy to give it the structure of a `ProductDataContainer` (Symes et al., 2005; Symes and Padula, 2005). `ProductDataContainer` realizes the Composite design pattern, which interacts with the Visitor pattern in a standard way (Gamma et al., 1994), so that `LocalFunctionObject`s and `LocalFunctionObjectRedn`s which act on the factors have naturally defined actions on the `ProductDataContainer`. It is intrinsic to the construction that this action executes correctly as an SPMD process.

Epetra has already implemented a variety of parallel computations, such as Euclidean inner product, as `Epetra_Vector` methods. To take advantage of this code base, we

wrapped a number of these methods in `operator()` methods of `LocalFunctionObject`s, via downcast of its arguments to `EpetraVectorLDC` and delegation to the Epetra methods called on the exposed `Epetra_Vector` data members.

Having defined an appropriate `DataContainer` subclass, it is straightforward to build a corresponding `Space` subclass `EpetraMultiVectorSpace`. The `buildDataContainer` method invokes the `EpetraMultiVectorDC` constructor, and the linear algebra methods are implemented via `FunctionObject`s and `FunctionObjectRedn`s using the Epetra parallel methods, as in the last paragraph.