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

ANTHONY D. PADULA and SHANNON D. SCOTT and WILLIAM W. SYMES
Rice University

The Rice Vector Library is a collection of C++ classes expressing core concepts (vector, function,...) of calculus in Hilbert space with minimal implementation dependence, and providing 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.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures

General Terms: Algorithms, Design

Additional Key Words and Phrases: abstract numerical algorithms, numerical optimization, complex simulation

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 modeling 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) phys-

Current addresses: Anthony D. Padula, AEW (FATE IPT), Northrop Grumman Co., Bethpage NY; Shannon D. Scott, Imtec, Los Alamos NM; William W. Symes, Department of Computational and Applied Mathematics, Rice University, Houston TX 77251-1892 USA, email `symes@caam.rice.edu`. The work reported here was supported in part by National Science Foundation grants DMS-9973423, DMS-9973308, and EAR-9977697, by the Department of Energy under Contracts 74837-001-03-49 and 86192-001-04-49 through the Los Alamos National Laboratory Computer Science Institute (LACSI), by the Department of Energy EMSP grant DE-FG07-97-ER14827, by grants from ExxonMobil Upstream Research Co., and by the sponsors of The Rice Inversion Project (TRIP). The Rice Terascale Cluster is funded by the National Science Foundation under Grant EIA-0216467, and by grants from Intel and HP.

ical 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* (sometimes called *matrix-free*) algorithms use only the intrinsic operations of linear algebra and calculus in Hilbert space, and form an active subject of research in the numerical optimization community. Examples include Krylov subspace methods for the solution of linear systems and eigenvalue problems, Newton and quasi-Newton methods for unconstrained optimization, and many constrained optimization methods. See for example Nocedal and Wright [1999] for a comprehensive overview of modern numerical optimization, and Ridzal [2006] for recent work on matrix-free constrained optimization.

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, reverse communication,...) lead to software that is difficult to debug and maintain and nearly impossible to modify, extend, or reuse outside of the originating context.

Object oriented programming appears to offer a way out of this dilemma. The two principal elements of object orientation, data abstraction and polymorphism, have immediate implications for design of reusable numerical code. Data abstraction defines data objects in terms of the operations used to manipulate them, rather than in terms of their composition out of simpler objects. This device allows implementation details in one part of a program to be hidden completely from other parts which do not intrinsically involve them. For example, an optimization algorithm might manipulate only the parameters (eg. nodal values) of a finite element mesh. Without data abstraction, the algorithm must be written to reference the entire finite element data structure; the implementation becomes dependent on data which is foreign to the algorithm, such as element geometry, and cannot be used without modification in a different context. An abstract data array type can expose only those details (in this case, the array of nodal values) needed in context of a specific algorithm or class of algorithms, via a set of operations on these details, and hide other details entirely. Thus data abstraction frees the algorithm writer from the necessity to refer explicitly to all aspects of a data structure.

Polymorphism complements data abstraction to enable reuse of numerical algorithms across many applications. Polymorphic functions accept a variety of argument types, and perform operations dictated by the type of their arguments. This concept is familiar: for example, the "+" operator in Fortran 77 accepts arguments of all arithmetic types and performs the appropriate form of addition for each. Abstract polymorphic functions take this concept a step further: they accept arguments of abstract data types, and rely for their definitions only on the attributes of these types. For example, Krylov-subspace algorithms for solution of linear sys-

tems use only matrix-vector multiplication. Some matrices, such those generated by time-dependent simulation, or element stiffness matrices in non-assembled form, may define matrix-vector products while providing only indirect access to explicit matrix elements. Other matrices may exist computationally as arrays of elements, and matrix-vector products for these are naturally written in terms of the elements. A polymorphic linear operator type defines the matrix-vector product as an input-output operation on an abstract vector type, and can be realized concretely using either type of definition, accessing the data of the vector arguments as is appropriate. An abstract polymorphic linear operator type does not specify the means by which the matrix-vector product is carried out: it merely provides a promise that it is done. An algorithm written in terms of an abstract polymorphic linear operator type can thus be used in contexts involving either "implicit" or ordinary, explicit matrices, without recoding: the specific instance of the type need supply only the computations mandated by the type, without any reference to the explicit computations by which they are carried out.

These concepts do not in themselves mandate a type system for numerical optimization with any precision. Indeed, at least two major varieties of polymorphism - parametric (compile-time) and inheritance (run-time or late-binding) polymorphism - offer a wide variety of possible approaches. We have elected in the work reported in this paper to use largely (though not entirely) the runtime polymorphism paradigm, implemented by inheritance or subtyping. In addition, we have attempted to model the computational types as closely as we could on the mathematical types appearing in the abstract, mathematical descriptions of algorithms. That is, when a type of mathematical object is defined by a specific set of attributes, we have attempted to create a parallel, computational type with a comparable set of attributes. For example, the most general abstract mathematical definition of vector does not mention the concept of dimension - dimension is neither necessary nor sensible for the definition of many operations in linear algebra. The abstract computational vector type developed in the following pages also abjures the dimension attribute, deferring it to subtypes with additional structure for which dimension make sense.

These design principles have led us to formulate class library presented in this paper, the Rice Vector Library ("RVL"), an object-oriented software framework for the expression of coordinate-free algorithms in 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 polymorphic types or *classes*, such as `RVL::Space` and `RVL::Operator`, along with implemented (concrete) classes constructed out of the abstract components, such as `RVL::Vector`, which express related concepts. As mentioned above, 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, while simultaneously accommodating the computational requirements of large scale scientific programming. Critical to the success of our approach is the provision of another collection of abstract "data management" classes (`RVL::DataContainer` and `RVL::FunctionObject` in RVL), which offer uniform data abstraction methods

for hiding implementation detail.

A number of other projects have realized various aspects of an "OO numerics" program for numerical linear algebra and/or optimization [Nichols et al. 1992; 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; Dongarra et al. 2004]. 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. At various places throughout the paper, we will contrast the features of RVL with those of its progenitor HCL. RVL improves over HCL in more cleanly separating "calculus" and "data management" layers, and in providing the applications programmer with a simpler API typically requiring less coding to achieve the same functionality. Introduction of types for function evaluations was an important HCL innovation; RVL offers improved evaluation type semantics.

HCL and RVL are unique amongst OO numerics libraries (to our knowledge) in offering a maximally faithful computational realization of the core concepts of calculus in Hilbert space, and in proposing a type system which provides such realization as a functionally complete framework for the target class of algorithms. A few other OO numerics packages, such as Thyra and NOX from the Trilinos collection, also target abstractly defined algorithms, and define types bearing some relationship to calculus concepts. With the exception of NOX, these other packages heavily (sometimes exclusively) concentrate on *linear* problems, whereas HCL and RVL were designed from the ground up to express nonlinear problems and solution algorithms for them. Also, other OO numerics packages do not *primarily* aim to mimic the underlying mathematics of coordinate-free algorithms, but rather to provide an interoperability layer for a large variety of numerical algorithm packages (Thyra) or an API for nonlinear equation solvers (NOX), or aim at some other design goal. As one might therefore expect, the fundamental semantics expressed in these packages often diverges from that of vector calculus. The thesis of the HCL and RVL projects can be restated as follows: the mathematics itself, carefully emulated in a system of computational types, provides a functional API for coordinate-free optimization and linear algebra algorithms. "Functional" in this context means (amongst other things): allows for an effective approach to interoperation with other numerical libraries, and for high-performance and/or parallel implementation.

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. We will offer several examples below in which such wrapper constructions play a crucial role.

Also, RVL does not pretend applicability beyond the semantic scope specified in

the goal statement. For example, it does not model calculus in *Banach* space, nor does it at present express discrete variable concepts underlying integer and mixed-integer programming (though we have experimented with such an extension, which seems in principle feasible). The `norm` function supplied in the `RVL::Space` 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 `RVL::FunctionObject` interface. However these are not built into the fabric of `RVL::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.

An Illustrative Example

A simple example of the abstraction dichotomy described above 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.. Execution of this line typically requires several orders of magnitude more floating point arithmetic than do the other lines, which express generic linear algebra operations such as linear combination or inner (dot, scalar) product (represented by $\langle , \rangle$). Note that *no mention* of the coordinates of the fields $x, r, ...$ 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 (Hermitian) positive definite linear operator on an appropriate Hilbert space. A preconditioned version of this algorithm could be expressed in similar fashion.

The intent of the RVL project is to provide a system of abstract types which permit the expressions of algorithms such as that described in display (1), without the intrusion of computational details extraneous to the mathematical description of the algorithms. Such details must of course appear somewhere; RVL provides the means to defer these details to other software layers.

Plan of the Paper

The following pages discuss the "calculus" and "data management" types, and their use in expressing coordinate-free algorithms. Several example applications 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, interoperation with other numerics libraries, both OO and non-OO, 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 to present the essential features of RVL classes and their use in expression of coordinate-free algorithms. We refer the reader to the RVL reference manual for a comprehensive description of the classes and their usage [Symes and Padula 2005], and the design document [Symes et al. 2005] for a more complete description of the structure and design principles of the library.

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 fundamental concept of linear algebra is that of *vector space*, a set with associated operations obeying certain axioms. A typical introduction of the concept, 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).

The mathematical basis of the software reported in this paper is the *Hilbert space*, a mathematical structure which adds to the attributes of vector space an *inner product* (also called scalar or dot product), a conjugate-symmetric (Hermitian) conjugate-bilinear (or sesquilinear) positive-definite scalar-valued function. We shall assume without further mention that all vector spaces occurring the the

discussion have this additional structure, that is, are Hilbert spaces.

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. Computational spaces will thus be *Abstract Factories* [Gamma et al. 1994], 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. On the other hand, vectors *have* coordinates once a basis is specified, and coordinates in a specific basis invariably provide the sole computational access to vectors. Linear combination, inner product, and all other operations on vectors must ultimately be realized as operations on coordinates, 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.

Using `Space` and `Vector`

The UML class diagram Figure 1 displays the relations between `Space`, `Vector`, and the two base classes from the data management layer which provide an abstract interface for interaction with data. **Note** that in the interest of clarity, we have suppressed some details from the UML, notably the `Scalar` template which is common to almost all of the RVL classes.

`Vector` instances own abstract containers for coordinate data, for which RVL provides the `DataContainer` base type. Creating a `Vector` thus entails dynamic allocation of a `DataContainer`. Since `Space` is the repository of all information needed to construct `Vector`s, a virtual constructor method of the `Space` class allocates the `DataContainer` data member during `Vector` construction. `Space` also provides the linear combination and inner product (virtual) implementations which manipulate `DataContainer` instances. `Vector` delegates linear combination and inner product to the corresponding `Space` methods, called on its `DataContainer` data member. `Vector` must therefore retain a reference to its `Space`, hence "knows" which space it is in. As will be discussed more extensively below, all manipulations of data encapsulated in `DataContainer`s occurs through an *evaluation* interface involving a *function object* type, `FunctionObject`. `Vector` uses `FunctionObject`s through delegation to the corresponding evaluation methods of `DataContainer`.

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. We emphasize that `DataContainer` *does not* constrain in any way the concrete data structures choses to represent the coordinate data of `Vectors`: these could be native C arrays, standard library containers of various sorts (vectors in the `std::vector` sense, lists, trees,...), out-of-core structures provided with paging rules, or distributed structures of various sorts, to name a few of many possibilities.

A `Space` subclass designer must implement five public methods (pure virtual in the base class), along with constructors initializing whatever internal data is necessary to make these work. The first of these five methods is a virtual constructor for the `DataContainer` subtype encapsulating the data structures specifying `Vectors` in this `Space`, with signature

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

`Space` offers the ability to compare instances:

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

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 `Space<Scalar>::operator==`.

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

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

The first of these implements the inner product, the characteristic attribute of Hilbert space. The second assigns to the data container argument the data defining the zero element of the space (the additive identity). Zero is the only vector whose coordinates are the same in all possible bases (namely, all zeroes). Typical implementations of the `DataContainer` class will provide arrays (or lists, or trees, or some other data structure) containing coordinates. `Space::zero` method will typically assign the scalar zero to the components of these data structures. The final method defines linear combinataion, that is, vector addition and scalar multiplication. Typical implementations, like those of `Space::zero`, will perform the analogous operations on coordinates stored in the argument `DataContainer` s.

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`. Implementations can recognize various special cases (`axpy`, copy,...), as determined by values of the scalars `a` and `b`, and generate efficient code for these. 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.

`Space` subclasses tend to share a great deal of code. 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, i.e. completely implemented. All methods delegate 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 a `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. Construction of vector workspace is very simple: if `sp` is a `Space` instance, then

```
Vector<Scalar> x(sp);
Vector<Scalar> z(sp,true);
```

builds two vectors, the second representing the zero vector.

Vectors know the space that they belong to, and 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. This feature of RVL contrasts with the vector-space relationship implemented in HCL, which permits `HCL_Vector` instantiation without reference to an existing `HCL_Space` object.

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. The linear combination method

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

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 "convenience" methods (`scale`, `copy`, `norm`,...) by delegation to `Space` methods of the same name, which are provided default implementations in terms of the basic `Space` linear algebra methods. Note that norms must

return positive reals. Therefore the `ScalarFieldTraits` specializations typedef `AbsType`, the return type of the absolute value function and for `norm` and `normsq` (implementing $x \mapsto \sqrt{|\langle x, x\rangle|}$ and $x \mapsto |\langle x, x\rangle|$ respectively).

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

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

The first line invokes the `applyOp` 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. For efficiency reasons, the base class `DataContainer` cannot mandate the precise form of data access. RVL defers data access definition to subclasses of `DataContainer`. The appendices discuss some examples.

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 therefore also own information about the layout and location of the data. Thus `DataContainer` is provided with a means to evaluate `FunctionObject`s.

`DataContainer` and `FunctionObject` together form an example of the *Acyclic Visitor* design pattern [Martin et al. 1998; Martin 2002]. 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]. As implied by its role in this pattern, `RVL::FunctionObject` is a *degenerate* base: it defines only a standard reporting method (which may be overridden in subclasses). Its interaction with `DataContainer` is left entirely unspecified. 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 Acyclic Visitor applications.

The `DataContainer::eval` expresses acceptance of a Visitor (`FunctionObject`) by an Element (`DataContainer`), 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;
```

RVL does not support output aliasing in implementations of `eval`: implementations need not assure correct results when the `DataContainer`, on which `eval` is called,

is aliased with any of the input arguments `sources[i]`.

`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. Aliasing of output with input arguments is also unsupported for these methods.

RVL provides specializations of `Vector::eval` admitting zero to three `Vector` arguments (in addition to the `Vector` on which `eval` is called). These are often more convenient to use than the generic evaluation method, in terms of which they are implemented in the obvious way. For example, the binary interface, combining two `Vector`s and a `FunctionObject`, is:

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

`FunctionObject` is intended to encapsulate calculations which return "large" results, whence the "return value" of a `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 a `FunctionObjectRedn` base class, the root of another Acyclic Visitor hierarcy. A `FunctionObjectRedn` instance manages an object of an abstract return type `RetType`, to which it offers access via `setResult` and `getResult` methods. Since reductions frequently target scalar types, we also provide a `FunctionObjectScalarRedn` subclass with `setValue` and `getValue` access methods for scalars.

Both `DataContainer` and `Vector` are provided `const` methods to evaluate `FunctionObjectRedn`s, 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.

The sequence diagram in Figure 2 depicts a typical instance of `FunctionObject` 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 in detail in [Symes and Padula 2005; Symes et al. 2005], is a concrete implementation of a Cartesian product of `DataContainer`s, a Composite in design pattern terms. The components of this product, in the case illustrated, are `DataContainer`s which expose their array data, and own references to no further `DataContainer`s. The specific type is discussed in the first appendix. `FunctionObject`s are evaluated directly on this latter type: its `eval` method calls the `operator()` method of its `FunctionObject` argument, which accesses 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.

Scalar functions, i.e. those of the form $z_i = f(x_i, y_i, ...)$, can be encapsulated in `FunctionObject`s which any `DataContainer` offering compatible data access mode can evaluate. Bartlett Bartlett et al. [2004] has termed such functions *vector transformation operators*, and points out that only these can be evaluated efficiently without detailed information on data distribution, assuming only that corresponding components of the coordinate arrays involved in the evaluation are stored in the same local memory.

Linear combination is a scalar function, as is zero assignment. The Euclidean inner product is a scalar reduction (*vector reduction operator* in Bartlett's terminolgy). Implementations of these three as `FunctionObject`s and `FunctionObjectScalarRedn` may be shared across all `Space` subtypes whose associated `DataContainer`s expose data in the same way. This leads to a standard construction of `Space` subclasses with Euclidean inner products, described in Figure 3. The `StdSpace` Facade class combines a `DataContainerFactory` and a `LinearAlgebraPackage` to realize a `Space`. A `LinearAlgebraPackage` packages together to a `FunctionObjectScalarRedn` and two `FunctionObject`s defining inner product, zero assignment, and linear combination, with access methods. `StdSpace` uses a Factory class, `DataContainerFactory`, to implement the virtual `DataContainer` constructor required by `Space`, and offers a comparison method to facilitate `Space` comparison.

The `StdSpace` construction minimizes the programming effort involved in construction of a `Space` subclass. HCL has no abstract definition of data interaction (data management interface, RVL's `DataContainer` and `FunctionObject` /`FunctionObjectRedn`base classes) hence cannot offer a similar labor-saving construction.

FUNCTIONS AND EVALUATIONS

RVL defines three base classes for functions of a vector variable: `LinearOp`, for linear operators; `Functional`, for scalar-valued functions; and `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. 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

As RVL aims to express *coordinate-free* algorithms, its `LinearOp` interface is not required to divulge matrix elements, for example. Only an operator application method need be supplied.

Linear operators on Hilbert space implicitly exist in pairs: each operator has an adjoint. Like the mathematical homolog, "adjoint" here means: with respect to the inner products defined in range and domain, which are also 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`.

Note that several other OO numerics libraries, mentioned in the introduction, define types representing some form of abstract linear operator, for which the definition (or even declaration) of an adjoint is optional. As we have just pointed out, such a class does not mimic the semantics of linear operators in Hilbert space.

The `applyOp` and `applyAdjOp` methods are pure virtual. To build an instantiable `LinearOp` subclass, the user must implement these, 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 use of the methods exposing domain and range is to generate workspace, for example:

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

RVL also supplies an overload of the `LinearOp::applyOp` method which includes linear combination with a vector, i.e. $y \leftarrow \alpha Ax + \beta y$, and a similar method for the adjoint, with obvious (and possibly inefficient) default implementations. These overloads, popular in other OO numerics libraries, are 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.

To facilitate application-building, RVL supplies a unit test `AdjointRelationTest`, as a standalone function. This test checks internal consistency between application of the operator and its adjoint, by choosing random vectors in domain and range (via evaluation of an externally supplied randomization `FunctionObject`), calling `applyOp` and `applyAdjOp`, computing inner products, and reporting the results on the stream argument. Its interface involves only the abstract type `LinearOp`, together with a randomization `FunctionObject` and a specification of output stream.

Nonlinear Functions and Evaluation objects

The definition and use of the RVL classes for (possibly) nonlinear scalar- and vector-valued functions, `Functional` and `Operator`, are intertwined with the properties of *Evaluations*. Evaluation objects store values of a mapping and its derivatives at a point of its domain, along with consistent intermediate results which may be shared by computations of these values. For example, computing a value generated by a finite element simulator involves mesh generation, stiffness matrix assembly, and so on. Computations of derivatives ("sensitivities") of the simulator with respect to parameters may require the very same data, but not necessarily in the same part of the program. Evaluations enable reuse of such data and enforce its internal consistency.

HCL introduced the Evaluation concept, and RVL Evaluations are an evolution of HCL's. The NOX project at Sandia National Laboratories defines a similar Group concept. Most importantly, RVL Evaluations are *concrete* (unlike HCL's): the application developer need only implement a `Operator` or `Functional` object, and RVL supplies the necessary Evaluation code. Semantically, RVL Evaluations represent a mapping'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 $f(x)$; in $f(x)$ these are dynamically linked. To mimic this logical relationship. RVL Evaluations automatically recompute values when these are requested, if the evaluation point `Vector` has undergone a change of internal state since the last call. This dynamic relation contrasts with HCL, which required a new Evaluation object for each update of the evaluation point, and NOX, which requires the algorithm writer to manually force an update of the value(s).

The relation between an Evaluation and the `Vector` evaluation point 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 non-`const` linear algebra methods of `Vector`.These methods invoke the version update method implementing the Observer relation. 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. For example, the various non-virtual value access methods of the mapping classes, such as `LinearOp::applyOp` and `OperatorEvaluation::getValue`, necessarily and implicitly update the versions of their target `Vector`s, as their only possible implementations involve `Vector::eval(FunctionObject,...)` and/or the non-`const` linear algebra methods..

Using 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:

```
Functional<Scalar> * Functional<Scalar>::clone();
```

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:

```
void Functional<Scalar>::apply(const Vector<Scalar> & x,
                               Scalar & val) const;
void Functional<Scalar>::applyGradient(const Vector<Scalar> & x,
                                       Vector<Scalar> & g) const;
void Functional<Scalar>::applyHessian(const Vector<Scalar> & x,
                                      const Vector<Scalar> & yin,
                                      Vector<Scalar> & yout) const;
```

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, which may be allocated as needed.

The public virtual method `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

```
abstype getMaxStep(const Vector<Scalar> & x,
                   const Vector<Scalar> & dx) const;
```

This method returns the signed distance to the boundary of the domain in a specified direction, from a specified point. Its default implementation returns the maximum `ScalarFieldTraits<Scalar>::AbsType()`.

RVL supplies standalone unit tests `GradientTest` and `HessianTest`, implemented in terms of the base class interfaces. 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.

A natural construction of a `Functional` subtype might rely on the services of several appropriate `FunctionObject` and `FunctionObjectRedn` subclasses 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.

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 vector-valued function class Operator and its associated evaluation class work in a precisely similar way, as described in detail in Figure 5.

A Quasi-Newton Algorithm Expressed in RVL

The core code for a quasi-Newton algorithm with line search globalization shows a typical use of Functional, FunctionalEvaluation, and other core RVL classes. For a mathematical description of this algorithm and others like it, consult Nocedal and Wright [1999].

This algorithm combines a search direction computation and a line search, represented by appropriate abstract types. The line search is encapsulated in the RVLUmin::LineSearchAlg class, a subclass of RVLAlg::Algorithm. The RVLAlg::Algorithm type is defined in the Algorithm package, an independent but compatible offshoot of the RVL project [Padula 2005]. The chief attribute of an RVLAlg::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. In RVLUmin::LineSearchAlg, run and other attributes are delegated: an initialization method dynamically allocates an instance of another RVLAlg::Algorithm subtype, RVLUmin::LineSearchBase, which is maintained as private data, and which actually performs the line search. This allocation via a pure virtual build method makes RVLUmin::LineSearchAlg an abstract factory. Subtypes implement the build method to construct instances of specific line search algorithms. The line search type has a number of other attributes, exposing the past and current search points, objective values, and gradients.

The direction computation base class, RVLUmin::UMinDir, is an interface with two pure virtual methods: calcDir computes a search direction given a FunctionalEvaluation, and updateDir accesses the facilities of a line search object to update the search direction computation.

On construction, RVLUmin::UMinStep acquires references to externally defined FunctionalEvaluation, RVLUmin::LineSearchAlg, and RVLUmin::UminDir objects, say fx, ls, and dc respectively. The RVLUmin::UMinStep::run method combines these to take a step: stripped of some inessential detail,

```
bool run() {
  try {
  // compute current search direction
  dc.calcDir(dir,fx);
  // Line Search - will typically update x, hence fx!
```

```
    ls.initialize(fx,dir);
    if( ! ls.run() ) return false;
    // update direction computation
    return dc.updateDir(ls);
    } catch (...) {...}
  }
```

A successful line search updates the evaluation point of `fx`, which serves as the current search point. This in turn causes the internal data of `fx` to be updated, as explained above.

As its name indicates, `RVLUmin::UMinStep` performs a step in a quasi-Newton iteration. To make a full-fledged algorithm, one must iterate calls to the `run` method of this class. The UMin package contains a `RVLUmin::UMinAlg` type, which wraps the iteration, and initializes the `FunctionalEvaluation fx` referenced above.

A full description of the design and structure of Algorithm and UMin may be found in the first author's PhD thesis, [Padula 2005].

A somewhat more involved illustration of the use of these classes is the `run` method core of a simple backtracking linesearch algorithm, a concrete `RVLUmin::LineSearchBase` subtype. The code is 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 `RVLUmin::LineSearchBase` superclass also supplies a method to check for steps that are too small.

```
bool run() { try {
  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);

  // 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;
  } catch (...) {...}
}
```

Note that the first line defines `x` as an alias for the evaluation point of the `FunctionalEvaluation` 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.

A particular quasi-Newton algorithm, the limited memory variant of Broyden--Fletcher-Goldfarb-Shanno, appears in several of the examples to be discussed below. See Nocedal and Wright [1999] for a description of this algorithm, the name of which we shall abbreviate as "LBFGS". The `RVLUmin::LBFGSDir` subclass of `RVLUmin::UMinDir` encapsulates the characteristic direction computation and internal update of this algorithm. These depend in turn on a representation of the BFGS inverse Hessian approximation as a `LinearOp` subtype `RVLUmin::LBFGSOp`. Denoting by `H` the `RVLUmin::LBFGSOp` data member of `RVLUmin::LBFGSDir`, the body of the `calcDir` method is

```
bool calcDir(Vector<Scalar> & dir,
             FunctionalEvaluation<Scalar> & fx) {
  Vector<Scalar> const & grad = fx.getGradient();
  H.applyOp(grad,dir);
  dir.negate();
}
```

`RVLUmin::LBFGSOp` has an `update` method, which uses previous and current search points and gradients to compute the rank-two BFGS update to the inverse Hessian, maintaining the finite list of such updates characteristic of the limited memory variant. The `updateDir` method of `RVLUmin::LBFGSDir` calls this `update` method of the inverse Hessian class.

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. Some intrinsic inefficiency is inherent in the construction of compositions, for example, in that it uses temporary `Vector` storage and misses hidden opportunities for loop fusion. Once the component operators are constructed, however, very little additional code is required to build the composition.

For details on the structure and usage of these classes, consult [Symes et al. 2005; Symes and Padula 2005].

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 `LinearOpWithInverse` of `LinearOp` which supplies the action of the inverse derivative and its adjoint, via another pair of protected methods:

```
const LinearOp<Scalar> & applyInv() const;
const LinearOp<Scalar> & applyAdjInv() const;
```

and public (implemented) `applyInvOp`, `applyAdjInvOp` methods which delegate to these.

An `OperatorWithInvertibleDeriv` naturally supplies a `LinearOpWithInverse` as the return value of its `getDeriv()` method, as can be revealed by a cast. The RVL-Algorithm implementation of Newton's method looks like this:

```
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.applyInvOp(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 scalar 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 $0, \pm 1, \pm 0.4i$. The test driver 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 roots of p, and the usual quadratic convergence of Newton's method is observed. See Padula [2005] for more details.

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.

Polynomials Division by Least Squares

The discerning reader will have noticed that the `Space` interface does not provide a method returning the dimension of the represented vector space. Two reasons underly this design decision: (1) the target class of coordinate free algorithms does not require it, and (2) the RVL class can in fact directly represent vector spaces which do not have well-defined dimension - in other words, infinite dimensional spaces.

To illustrate this possibility, we have constructed a `Space` representing the vector space $\mathbf{P}$ of polynomials of arbitrary degree, mathematically equivalent to the vector space of finite (coefficient) sequences: as a set,

$$\mathbf{P} = \{a : \mathbf{N} \to \mathbf{R} : \text{for some } N \geq 0, n > N \Rightarrow a_n = 0\}$$

Supplied with the usual Euclidean inner product, $\mathbf{P}$ is a so-called pre-Hilbert space: its completion is the Hilbert space l^2 of square-summable sequences.

There are lots of ways to represent $\mathbf{P}$ computationally, corresponding to the many choices of suitable `DataContainer` subclass and underlying data structure. This choice defines how (and which) function objects are to be evaluated. For implementation of our `SeqSpace` subclass of `Space`, we chose to use `std::list` as the underlying data structure. Any container that permits systematic access to elements would suffice: native C++ arrays, `std::vector`, even `std::tree` would work equally well. This freedom to use any reasonable data structure to represent vector data appears to single out RVL amongst the current generation of OO "abstract numerical algorithms" libraries.

We defined `FunctionObject` subtypes to carry out the standard linear algebra operations, as well as translation wrapper classes for arbitrary `LocalFO`s and `LocalFOR`s from the LocalRVL library (see Appendix I for a description of these classes). The `Local` classes are array-oriented, and are compatible with function object classes defined in other libraries, for example RTOp vector transformation and reduction operators from Trilinos [Bartlett et al. 2004]. Thus simple wrapper constructions make a wide variety of functions available.

The wrapper construction implicitly defines LocalRVL operations as inductive limits (essentially, sequences of operations defined on the nested sequence of finite-dimensional coordinate subspaces). For elementwise operations (eg. all of RTOp) the results are stable, i.e. independent of the finite-dimensional subspace so long as it's large enough. For other (non-elementwise) operations the results are well-

defined but may be surprising, in particular not (necessarily) stable in the sense just defined. RVL makes no claims regarding this type of stability (indeed, RVL itself makes no claim at all about what `FO`s do or how they do it!).

We used the framework provided by `SeqSpace` to attack the simple polynomial division problem:Given $\alpha \in \mathbf{R}$, find p so that

$$(1 + \alpha x)p(x) = 1.$$

Of course there is no solution in $\mathbf{P}$ (except for the $\alpha = 0$ case), but there is a solution in l^2 when $|\alpha| < 1$. Thus we pose the problem as a linear least squares problem in $\mathbf{P}$, to be solved by Conjugate Gradient iteration and implicitly regularized by truncating the iteration.

The operation of multiplication by $1 + \alpha x$ is represented by a `LinearOp` subclass `PolyMultOp`. The `applyOp` and `applyAdjOp` methods, i.e. polynomical multiplication and its adjoint, have a very natural implementation using the `push_front` and `pop_front` methods of `std::list` and the associated iterators.

We checked that the `PolyMultOp` implementation was adjoint-correct using the `AdjointTest` unit test function supplied by RVL. The RVL-Algorithm implementation of conjugate gradient iteration for the normal equations then produced precisely the correct results. The number of coefficients computed goes up (in principle without bound) as either the solution tolerance shrinks or $|\alpha|$ gets close to 1.

This example of approximate solution of an infinite-dimensional problem, encompassed within the RVL framework, hints that a similar approach might be taken in contexts such as adaptive mesh refinement for solution of PDE-based control problems. The polynomial division problem illustrates in a simple way the capacity of RVL to represent infinite-dimensional problems directly, and justifies our decision not to require that `Space` subtypes define a dimension.

Note that the RVL-Algorithm CG implementation is the same one used for problems defined in terms of the array-based LocalRVL classes, even though the underlying data representation differs from that used in `SeqSpace`.

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; 1998; Li and Symes 2007]. 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.

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], a *de facto* standard open source data processing package.

We used the SU SEGY trace i/o functions, wrapped as `RVL::RecordServer` objects, to compute J (and its gradient, which in this case might as well be computed at the same time) by reading traces until EOF. On each successive pair of traces read the computation described above is performed, and the result accumulated in J. The parameter vector $\mathbf{v}$ is represented 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.

We also produced a change-of-variable driver, carrying out only part of the computation. Because we had used the SU i/o facilities, this command differed from the SU command `sunmo` essentially only in its use of RVL classes and a few virtual function calls. 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). 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. 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 steps of the LBFGS algorithm described above, again using the Powerbook G4. We estimate that 60% of the execution time was spent in floating point arithmetic and core memory access, the remainder in disk i/o. 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.

Least Squares Fitting of Seismic Data

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.

For example, we have coupled the RVL-Algorithm LBFGS implementation to a finite-difference solver for the 2D wave equation to experiment with least squares data fitting for the acoustic model of seismology. The finite difference code is written in Fortran 77, partly in order to have access to efficient Automatic Differentiation ("AD") tools available in that language, partly to take advantage of the efficient compilation of F77 in this type of application. Profiling a considerable number of examples showed that 98-99.6 % of the execution time (in a serial Linux or OS-X workstation environment) was spent in the F77 subroutines. The superstructure of RVL classes and virtual function calls appeared to have an insignificant effect on computation speed in these examples.

Source Estimation for Steady State Diffusion

The model used in this example approximates the physics of planar steady-state passive transport and diffusion of a substance throughout a volume, with all fields assumed translation-invariant in one direction so that the problem is effectively 2D. For further discussion of this problem, see Section 8 in Quateroni and Valli [1994] and Section 9 in Knabner and Angermann [2003]. After P1 finite element discretization, the concentration of the diffusing substance is represented by the vector $\mathbf{y}$, and its measurements by the vector $\hat{\mathbf{y}}$. Somewhat unrealistically, we assume that concentraion measurements are supplied for every point in the volume, so that the two vectors have the same length. Denote by $\mathbf{u}$ the source concentration, which appears on the RHS of the steady-state diffusion problem, and by $\mathbf{b}$ a vector representing passive sources and sinks in the system. Then the least-squares best fit of the concentration to its data results in the quadratic optimization problem

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

in which A is the stiffness matrix of the Laplace operator, B is the mass matrix, Q is a weight matrix defining the cost of data misfit, and R is a regularization matrix. A Newton-related algorithm for the solution of (2) 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.

For distributed storage of the arrays A, B, Q, b and R, we used the Epetra library from the Trilinos collection [Heroux 2003] to construct suitable `DataContainer` and `Space` subclasses for this application. Appendix B sketches this construction. The `Functional` subclass constructor accepts an `Epetra_Comm` object, encapsulating data layout and communication information, from the driver. The constructor

initializes an `AdvDiffProblemHolder` instance, an ad-hoc 'container for four Epetra matrices and one Epetra vector, using the `Epetra_Comm` object passed from the `Functional` constructor. The `AdvDiffProblemHolder` also initializes 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. The remainder of the `Functional` construction implements the Facade pattern by subclassing `StdFOFunctional`, initializing `FunctionObject`s and `FunctionObjectRedn`s to used to define the `apply...` methods. The `FunctionObjectRedn` defining the function evaluation and the `FunctionObject`s defining the gradient and Hessian share a reference to the common `AdvDiffProblemHolder`, thus reusing intermediate results.

The bulk of the computational work involved in evaluation of these `FunctionObject` objects goes into solution of linear systems: computation of the feasible point $\mathbf{y}(\mathbf{u}) = A^{-1}(B\mathbf{u} + \mathbf{b})$, and of $\mathbf{z} = A^{-T}\mathbf{y}(\mathbf{u})$ in the gradient calculation. We adapted the AztecOO linear solver package to accomplish this task. AztecOO is a sparse linear solver library [Heroux 2004], designed to work with Epetra data objects. 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, and Padula [2005] for more details on the construction the of RVL adapter classes..

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. We used LBFGS on this quadratic problem only for comparison purposes - of course a Krylov linear solver could have been used instead to solve the normal equation.

The P1 finite element mesh used in this example had 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 6 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: the performance of the application is dominated by the usual factors which regulate efficiency in parallel computation. Virtual function calls and other overhead imposed by the RVL layer of the application appeared to have no significant effect on performance.

DOWNLOAD INSTRUCTIONS

The Rice Inversion Project website (`www.trip.caam.rice.edu`) contains a link to the RVL project home page. The home page offers download and installation instructions for the current version of RVL, Algorithm, and possibly other related packages. The `SeqSpace` example, described above, is a subpackage of Algorithm.

CONCLUSION

The RVL project has as its central goal the creation a class hierarchy that mimics as closely as possible the basic concepts of calculus in Hilbert space, while deferring as 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 minimally intrusive way. We believe that this design approximates conformance to the Einsteinian dictum: "...as simple as possible, but no simpler".

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]. This appendix briefly sketches our use of Epetra facilities to construct RVL classes, which greatly eased the construction of this application. For additional details see [Padula 2005].

The principal adaptation targets are `Epetra_Vector` and `Epetra_MultiVector` (Epetra does not define a namespace, but uses the prefix `Epetra_` to signify package membership). `Epetra_Vector` provides several data access modes, and each of these could be used in creating a `DataContainer` subclass. We chose to subclass `LocalDataContainer<double>` (`double` being the only real type admitted by Epetra). We used the `Epetra_Vector::ExtractView` method, which exposes a pointer to the part of the object's data stored locally, to implement `LocalDataContainer::getData`. The layout of data across processors in a distributed platform is regulated by an `Epetra_Comm` data member of `Epetra_Vector`. The `EpetraVectorLDC` class has in turn an `Epetra_Vector` data member. This approach made `LocalFunctionObject`s available for manipulation of `EpetraVectorLDC`s, as well as native Epetra operations which can also be wrapped as `FunctionObject`s. These adaptor

FunctionObjects are not LocalFunctionObjects, of course, as they use the services of the Epetra classes and cannot be applied to general LocalDataContainers.

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 LocalFunctionObjects and LocalFunctionObjectRedns 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. Figure 7 diagrams EpetraMultiVectorDC, EpetraVectorLDC, and their relationships with RVL and Epetra classes.

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 FunctionObjects and FunctionObjectRedns using the Epetra parallel methods.

Acknowledgements

We are grateful to Mark Gockenbach, Michael Gertz, Amr el Bakry, Roscoe Bartlett, and Michael Heroux for many insights which were important in our work, and to Hala Dajani, Eric Dussaud, and Denis Ridzal for assistance with various aspects of the project. We are also grateful to the Associate Editor and three anonymous referees for useful criticism and suggestions which helped us to materially improve the manuscript.

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. A. 2003. MOOCHO: Multifunctional Object-Oriented arCHitecture for Optimization, User's Guide. Tech. rep., Sandia National Laboratory, 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 (Mar.), 62–85.

Benson, S., McInnes, L. C., and Moré, J. 2000. TAO: Toolkit for advanced optimization. Tech. rep., 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.

Dongarra, J., Lumsdaine, R., Pozo, R., and Remington, K. 2004. IML++ home page. math.nist.gov/iml++.

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. Tech. rep., 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++). Tech. rep., Sandia National Laboratories, `z.ca.sandia.gov/isis/`.

Karmesin, S. 2000. POOMA: Parallel object oriented methods and applications. Tech. rep., 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. Tech. rep., Sandia National Laboratories, Livermore, CA.

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

Li, J. and Symes, W. 2007. Interval velocity estimation via nmo-based differential semblance. *Geophysics 72*, U75–U88.

Martin, R. C. 2002. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, Englewood Cliffs, NJ.

Martin, R. C., Riehle, D., and Buschmann, F. 1998. *Pattern Languages of Program Design 3*. 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. 1992. The C++ language in physical science. Tech. Rep. SEP-75, Stanford Exploration Project, Department of Geophysics, Stanford University, Stanford, California, USA.

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

Padula, A. D. 2005. Software design for simulation-driven optimization. Tech. Rep. 05-11, 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.

Ridzal, D. 2006. Trust region SQP methods with inexact linear system solves for large-scale optimization. Ph.D. thesis, Rice University.

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

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

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. Tech. Rep. 05-12, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, USA.

Tech-X. 2001. OptSolve++. Tech. rep., 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.

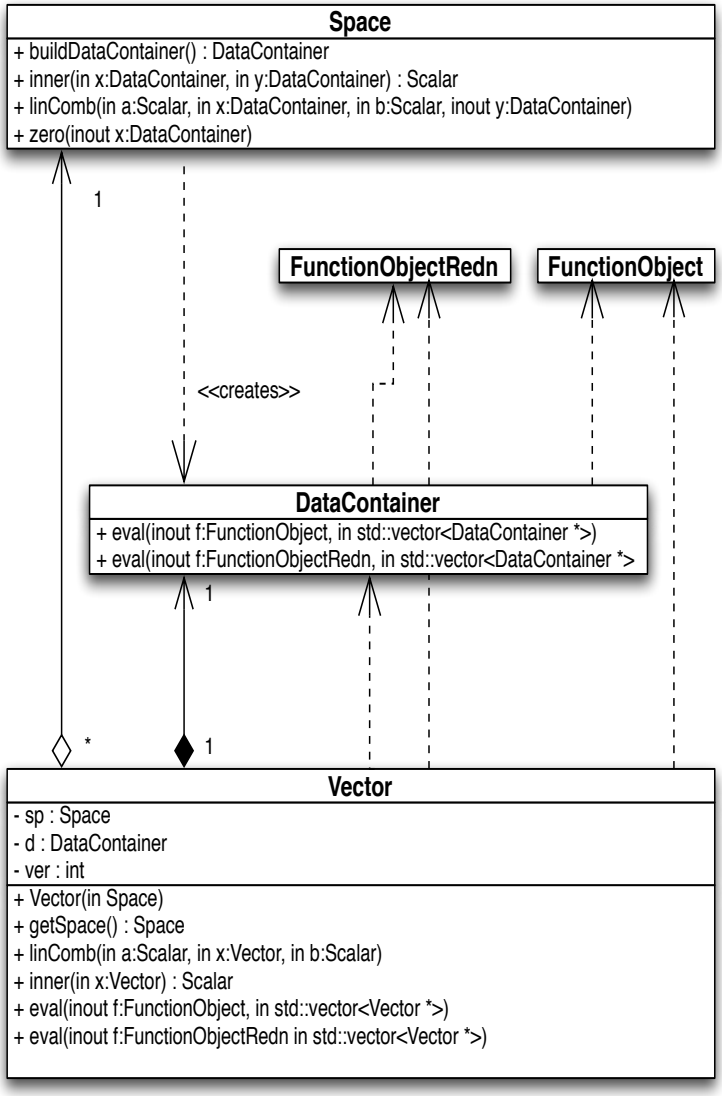

Fig. 1. `Vector` is implemented, and refers to `Space` (abstract) for its linear algebra services. Vector may also evaluate `FunctionObjects` and `FunctionObjectRedns` , by delegation to its data member `DataContainer`.

Received Month Year; revised Month Year; accepted Month Year.

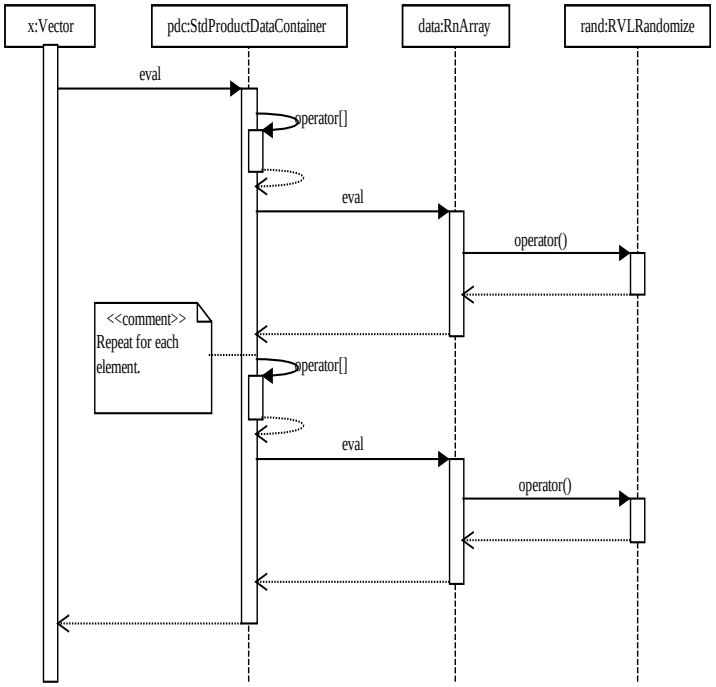

Fig. 2. Typical evaluation of a `FunctionObject` on a `Vector`. The `DataContainer` to which the `Vector` delegates the evaluation is in this case a Cartesian product (composite). The Visitor role of the `FunctionObject` interacts with the composite structure in a standard way, visiting the components in sequence.

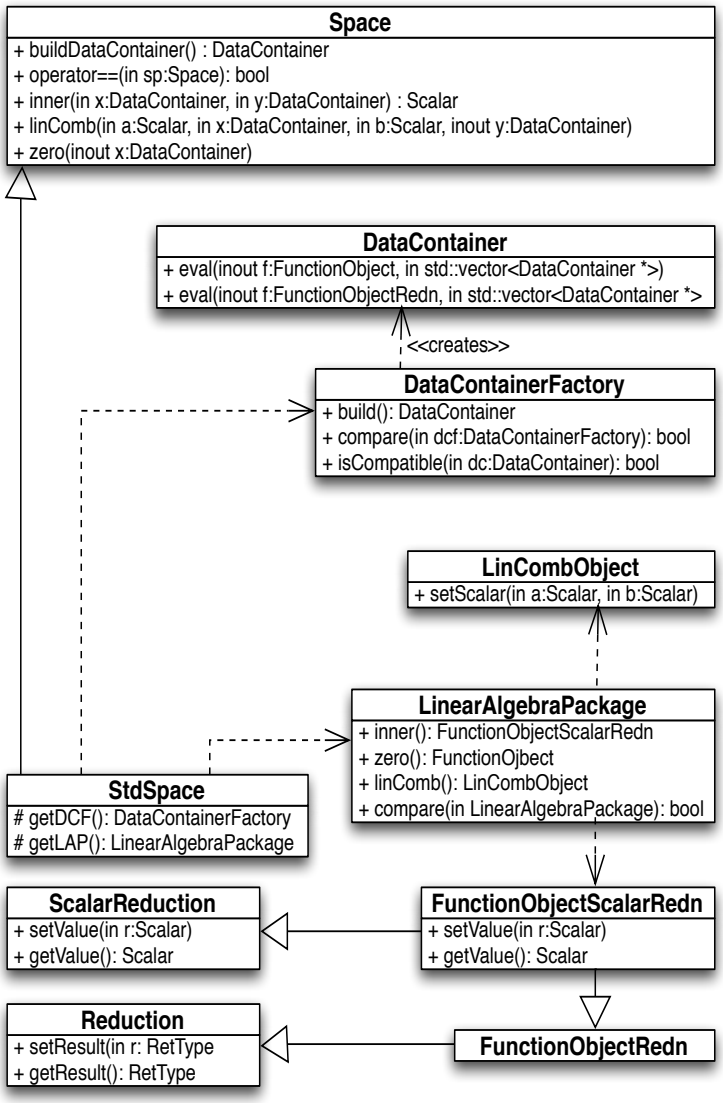

Fig. 3. A canonical construction of `Space` via the Facade class `StdSpace`, which combines a `LinearAlgebraPackage` and a `DataContainerFactory`. The `LinearAlgebraPackage` provides access to `FunctionObjects` which implement the arithmetic methods of `Space`, and the `DataContainer-Factory` is called to implement `buildDataContainer`.

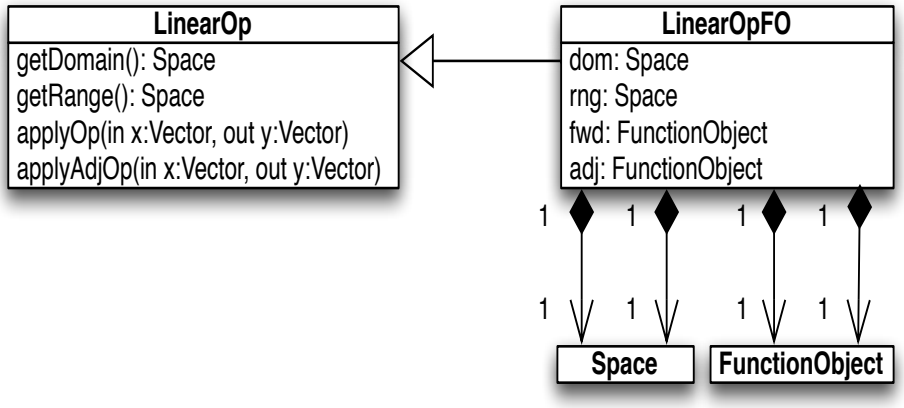

Fig. 4. Structure of the RVL `LinearOp` class, and of a typical implementation using a pair of `FunctionObjects`. The `apply` methods invoke evaluation of the (binary) `FunctionObjects` on their `Vector` arguments.

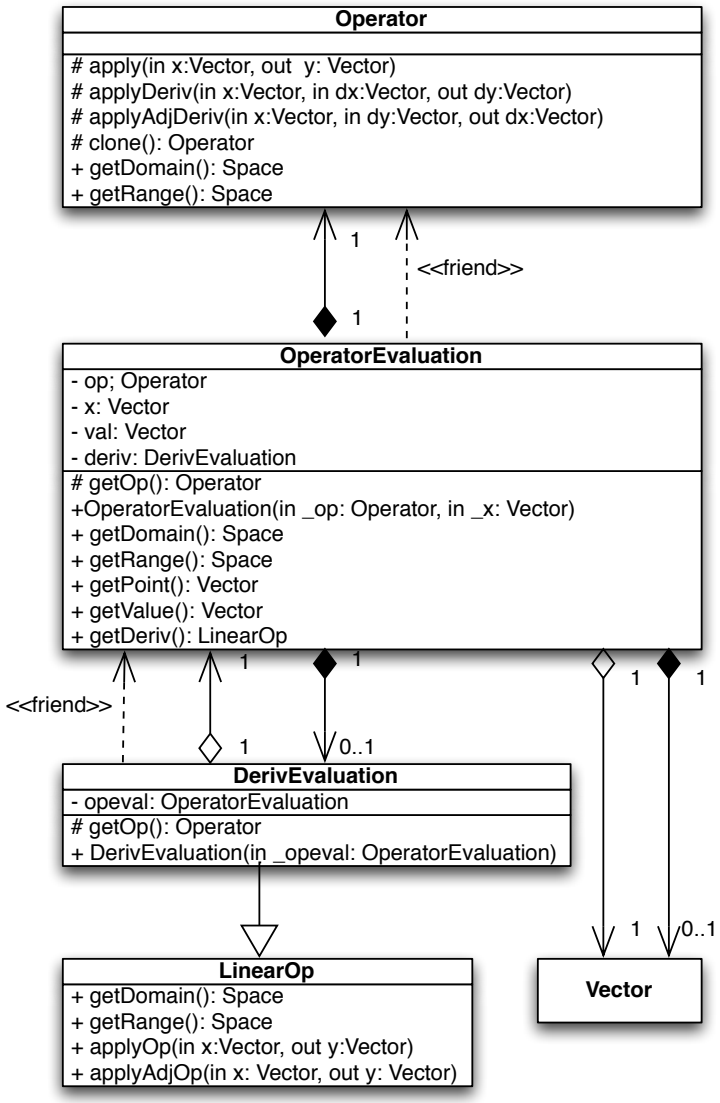

Fig. 5. The concrete class `OperatorEvaluation` clones operator and delegates its key computations to its `Operator` data member, and has an *Observer* relation with the evaluation point `x` (`Vector`).

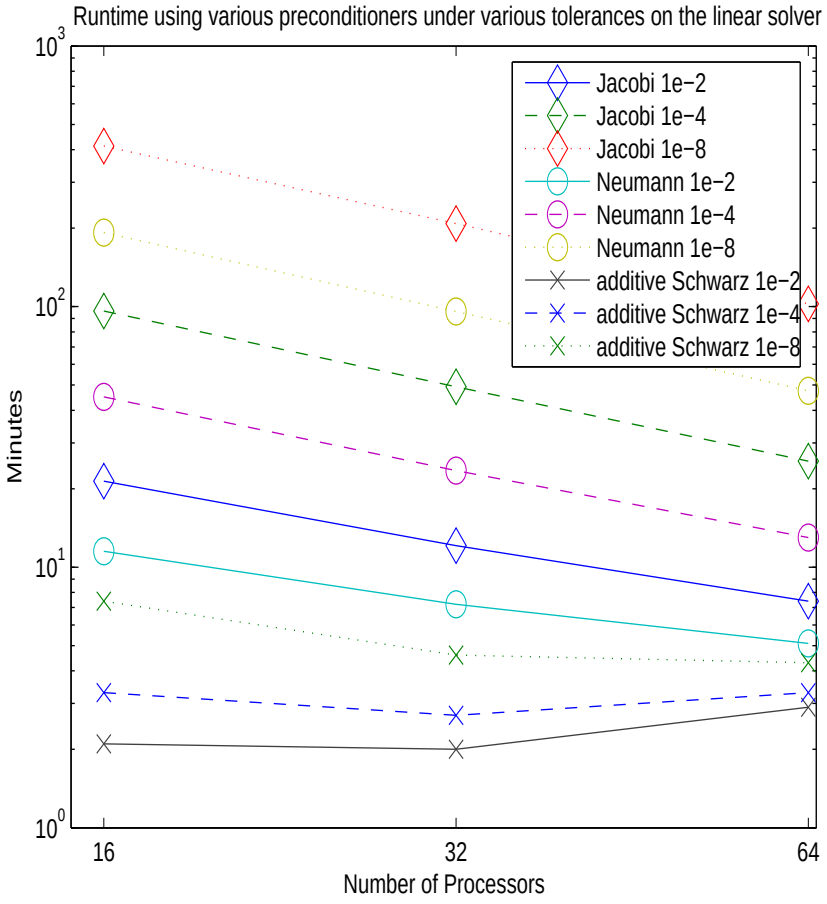

Fig. 6. Runtimes using various preconditioners, GMRES residual tolerance $= 10^{-8}$.

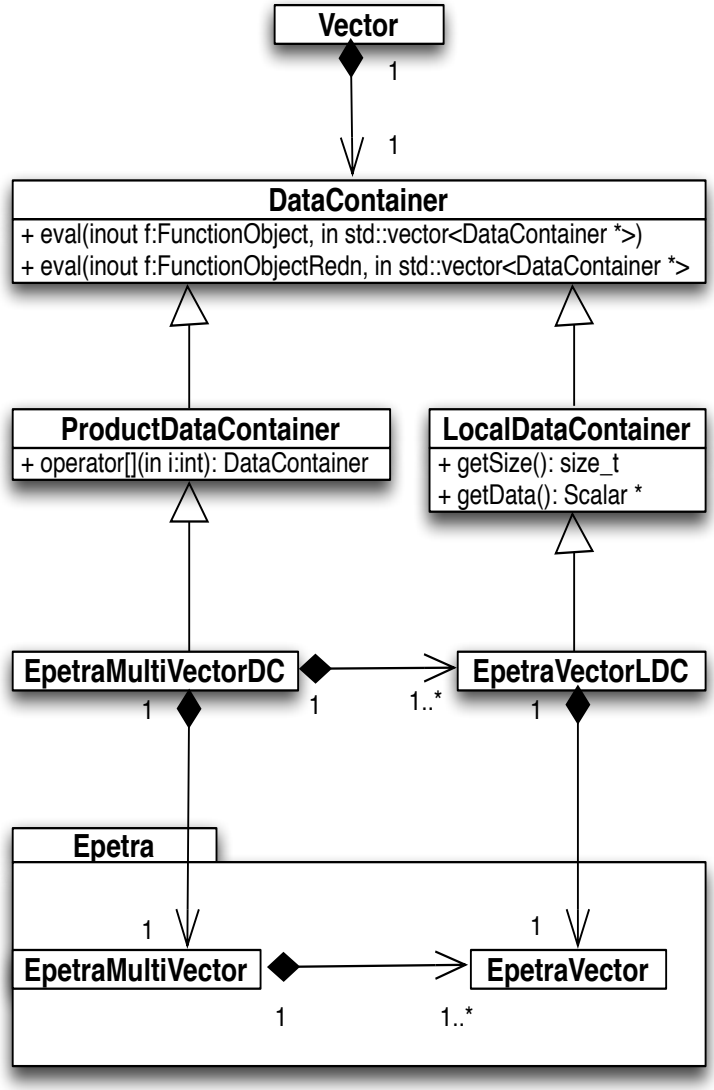

Fig. 7. Adaptor classes using `Epetra_MultiVector` and `Epetra_Vector` to implement `RVL::DataContainer` and `RVL::LocalDataContainer` respectively.