The Standard Vector Library: a software framework for coupling complex simulation and optimization

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

ABSTRACT

Object oriented design solves a fundamental programming problem arising in simulation driven optimization: the separation in code of multiple levels of abstraction naturally appearing in solution algorithms for such problems. The Standard Vector Library provides classes expressing core concepts (vector, function,...) of calculus in Hilbert space with minimal implementation dependence, and standardized interfaces behind which to hide application-dependent implementation details (data containers, function objects). Important innovations introduced by this project and its predecessor (the Hilbert Class Library) include vector space and function evaluation objects, natural product structures, and extensive tool or wrapper classes to ease application construction. The library features extensive use of ISO standard C++ support for both object-oriented and generic programming models, a component-friendly structure for support of distributed computing via client-server frameworks, and systematic extensibility of class capabilities through method-forwarding.

INTRODUCTION

Simulation driven optimization arises in a variety of scientific and engineering contexts, notably control, design, and parameter estimation. Specification of a simulator involves geometric meshes or grids, functions on these, equations relating grid functions and embodying (gridded versions of) physical laws, and iterative or recursive algorithms which produce solutions of these equations. Optimization algorithms on the other hand generally have no intrinsic interaction with geometry and physics, involving instead a more abstract layer of mathematical constructs: vectors, functions, gradients,... 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 invarably 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 *",...) lead to software that is difficult to debug and maintain and nearly impossible to reuse outside of the originating context.

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

A canonical example of this difficulty is the treatment of out-of-core data. Disk-to-disk processing is common in fields, such as reflection seismology and meteorological data assimilation, in which data volumes often exceed conveniently available core memory resources. Conventional Fortran or C library code written with the in-core array as primary data structure cannot be used in an out-of-core application without significant modification. Distributed data presents similar obstacles to re-use of procedural libraries.

Object oriented programming appears to offer a way out of this dilemna. Data abstraction permits the implementation details in one part of a program to be hidden completely from other parts which do not intrinsically involve them. Polymorphism and inheritance complement data abstraction and enable reuse of abstract code (for example optimization algorithms) across many applications. A number of projects have realized various parts of this OOP numerics program for optimization: see (Nichols et al., 1993; Douglas et al., 1994; Meza, 1994; Deng et al., 1996; ISIS Development Team, 1997; Tech-X, 2001; Tisdale, 1999; Veldhuizen, 1999; Karmesin, 2000; Benson et al., 2000; Langtangen, 1999; Gockenbach et al., 1999; Heroux et al., 2003; Kolda and Pawlowski, 2003; Bartlett, 2003).

This paper describes a prototype design for OO simulation-driven optimization libraries, which we call the Standard Vector Library ("SVL"). We set these design goals for SVL: it should

- enable the abstract expression in code of "abstract" (i.e. coordinate-independent) algorithms for optimization and linear algebra, with minimal intrusion of low-level implementation details such as memory management, data access and distribution, parallel execution, etc.;
- also permit straightforward expression of coordinate-dependent algorithms, eg. those dependent on dense or sparse matrix algebra, through specialization of interfaces;
- yield code as efficient as the best procedural implementations (i.e. Fortran) for large problem sizes, in both serial and parallel environments, given appropriate specialized implementations transparent at the level of abstract numerical algorithms.

Two less tangible goals, related to all of the above, are *simplicity* and *naturalness*. When an implementation detail may be hidden behind an abstract interface which is conceptually adequate to contain its full functionality, we hide the detail. As a result of diligent adherence to this maxim of simplicity, SVL expressions of algorithms are usually considerably less complex in appearance than analogous expressions using any of the libraries mentioned above.

The concomitant goal of naturalness is achievable because of SVL's focus on a single conceptual domain. SVL classes resemble the Standard Library classes in that they implement (or partially implement) well-defined concepts with clear interrelationships. SVL classes are mostly considerably more than interfaces - as much implementation as possible is provided, consistent with the scope of the ideas which they realize. Just as the design of interfaces makes SVL classes simple to use, the naturalness of implementations makes

SVL classes simple to extend (subclass), as is inevitably necessary to accommodate the data structures and functions of new problem domains.

While its authors can envision SVL as the basis for a standard set of abstract interfaces (whence the name), the immediate aim of the project is much more modest: a delineation of features that *any* effective design for a library of this type should have, demonstrated in a fully functional C++ package. The basis for this delineation is a set of core *mathematical* concepts which underly an important subset of algorithms for simulation-driven optimization, namely relatives of Newton's method, and the related concepts of linear algebra. SVL realizes these concepts (vectors, functions,...) computationally in a set of functional abstract classes, such as `Vector` and `Operator`. We have attempted to make the relationships amongst these "calculus" interfaces as parallel as possible to those amongst their mathematical counterparts. Critical to the success of this approach is the provision of another collection of abstract "data management" interfaces (`DataContainer` and `FunctionObject` in SVL), which offer uniform methods for hiding implementation details.

SVL supersedes the Hilbert Class Library ("HCL", (Gockenbach et al., 1999)), and incorporates many of its innovations. SVL differs from HCL in several important respects:

- extensive use of ISO C++ features - class and function templates, exception handling, namespaces, the standard template library - which were not widely and/or robustly available from compiler vendors at the beginning of the HCL project in the early 90's;
- careful separation of the "calculus" and "data" layers, permitting both much more reliable data encapsulation and natural extension to distributed computing via component frameworks;
- extensive use of *function forwarding*, inspired by Roscoe Bartlett's `RTOp` classes (Bartlett et al., 2004), to provide a simple interface for flexible and efficient evaluation of an essentially unlimited variety of functions;
- a radically simpler approach to memory management, for the most part obviating any need for user interaction with reference counting, smart pointers, and the like, particularly in the "calculus" layer.

The following pages discuss the design of "calculus" and "data management" types, and their interaction; the basic structure of the SVL vector classes follows from this discussion. Optimization algorithms manipulate objects representing vector functions, and SVL offers convenient representations for these. Most scientific data types of any significant complexity are Cartesian products. SVL provides systematic means for dealing with product data structures and functions defined on them. The description of the core SVL classes reveals that SVL differs from some competing projects in its approach to several important issues, such as memory management, exposure of global information about a vector and provisions for support of parallel computation. We discuss the reasoning behind SVL's design decisions on these issues in some detail. To tie the discussion together,

we present simple data gridding application which illustrates the use of SVL in formulating and solving a typical scientific programming problem. We end with a discussion of the relation of SVL to other OO numerics projects, existing and planned extensions and applications of SVL, and with some comments on the general nature and promise of OO numerics.

This purpose of this paper is the presentation of the SVL design and the reasoning underlying it. Accordingly, we describe the SVL classes in only enough detail to illustrate the principles of the design. We refer the reader to the SVL reference manual for a comprehensive description of the classes and their usage (Symes and Padula, 2004).

GENERAL CONSIDERATIONS FOR THE DESIGN OF VECTOR CLASSES

The central concept of calculus and linear algebra is that of *vector*, and the `Vector` interface is its embodiment in SVL. 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 importance of the vector space concept suggests that vector spaces ought to be realized as a computational type. However, 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. The software engineering construct which constructs instances of another type, without revealing the internal details of that type or of the construct itself, is the *Abstract Factory* (Gamma et al., 1994). Computational spaces will thus be Abstract Factories, and will in addition

bind to their products (vectors) specific methods for carrying out the basic operations of linear algebra, so making the products "behave like vectors".

HCL introduced vector space as a type, and SVL adopts this innovation. The `Space` class in SVL provides a method implementing linear combination

$$\mathbf{z} \leftarrow a\mathbf{x} + b\mathbf{y}$$

for vectors $\mathbf{z}, \mathbf{x}$ and $\mathbf{y}$ and scalars a and b. Echoing the typical parametrization of vector spaces by fields of scalars, `Space` is a class template, rather than a class. The (only) template parameter identifies the "field", i.e. numeric type which serves as a proxy for an actual field. Well-tested choices of this template parameter are `float` and `double`, though other choices are in principle possible (a complex type, for example).

In view of the aim of SVL (i.e. to provide a computational framework for calculus in Hilbert Space), `Space` also includes an inner product method

$$\mathbf{x}, \mathbf{y} \mapsto \langle \mathbf{x}, \mathbf{y} \rangle$$

The interfaces defining these methods involve interaction with data, and will be discussed later, after the data management layer of SVL is introduced.

`Space` is an abstract interface (pure virtual base class), and is the root of a tree of types. A standard construction of child classes, implemented in the still-abstract `StdSpace` subclass, reduces the number of methods for which code must be supplied to the bare minimum - nonetheless introduction of a new scientific data structure in an SVL application usually involves the definition of a new `Space` subclass.

It is important to understand that a vector, both mathematically and computationally, must be more than the array of scalar coefficients that specifies its identity with respect to some arbitrary basis - a vector also knows how to add itself to another vector in its `Space`, and to return an inner product with another vector. This shows that, quite apart from their unacceptably limited storage scope, intrinsic type arrays or STL vectors are not adequate abstractions to realize a computational vector class. SVL views vectors as *Composite* types, combining a `Space` (with its linear algebra operations) with an appropriate data container - the `Space` "teaches the data container how to act like a vector". The next sections will describe an appropriate set of data container types for this purpose, and the details of the Composite construction. For now, we merely note that data containers will be allocated dynamically, so that the computational vector will act as a *handle*.

SVL realizes these ideas in its `Vector` class. `SVL::Vector` is concrete, and subclassing it is seldom necessary or useful. A `Vector` owns a reference to its `Space`, and implements its essential linear algebra methods by delegation to the corresponding methods of its `Space`. A partial listing of the `Vector` class definition follows. The essential features of the class are displayed here; comments (used in automated construction of the reference manual) have been stripped out, along with a number of methods which will be discussed later in this document.

```
template<class Scalar>
class Vector {

private:
  const Space<Scalar> & sp;
  ...
public:
  Vector(const Space<Scalar> & _sp, bool initZero = false);
  ...
  const Space<Scalar> & getSpace() const { return sp; }
  virtual Scalar inner(const Vector<Scalar> & y) const;
  virtual void linComb(Scalar a, const Vector<Scalar> & x,
                       Scalar b, const Vector<Scalar> & y);
  virtual void zero();

  virtual void scale(Scalar c) {
    try { linComb(c,*this,0,*this); }
    catch (SVLException & e) {
      e<<"\n*** called from Vector::scale()\n"; throw e;
    }
  }
  ...
  virtual ostream & write(ostream & str) const {
    str<<"Vector Object\n";
    str<<"member of space:\n";
    sp.write(str);
    ...
  }
  virtual void write(SVLException & str) const;
};
```

This code fragment illustrates several features of `Vector`, as well as conventions found throughout SVL:

- like `Space`, `Vector` is a class template parametrized by a scalar type. `Scalar` could in principle be any type for which default construction, assignment, and the usual arithmetic operations are defined. SVL currently makes no guarantees for any choices other than `float` and `double`.

- the `Space` is stored as a (const) reference. This implies that the scope of the `Space` object referenced by the `Vector` must strictly contain that of the `Vector` - i.e. the `Space` exists externally to the `Vector` and that the `Vector` object depends on it. To make a `Vector`, you have to make a `Space` *first*. This relationship of computational objects mimics precisely the relationship of the corresponding mathematical objects.

SVL makes extensive use of reference data to express such logical dependence, which the semantics of C++ references then enforces. Dependence on an external object introduces a serious danger, of course - the possibility that the reference might "dangle", or refer at some point to a nonexistent object. The conventional solution to this problem is reference counting, usually combined with a so-called smart pointer type. However SVL takes a different approach. C++ scope rules prevent external references from dangling so long as the `Space` and `Vector` instances are allocated on the stack. SVL is designed to make dynamic allocation of `Space` and `Vector` instances unnecessary (in fact, impossible in most circumstances). This aspect of SVL memory management is discussed further in the *Implementation Details* section below.

- a `Vector` knows which `Space` it belongs to, and exposes this information publicly via `getSpace()`.

- initialization of the `Vector`'s data is optional - the only initialization with mathematical (invariant) meaning is initialization by zero, and that is the only option offered as part of construction.

- `Space` objects can be compared, via `operator==`. This comparison operator facilitates various forms of sanity testing (is this vector in the domain of this operator? are these two vectors in the same space?).

- the use of `const` is quite conventional in the "calculus" layer of SVL: essentially every method, argument, and return value that conceivably ought to be `const` is so declared. However const would be entirely redundant for the `Scalar` which `inner` returns by value. The same is true of objects returned by reference, so long as their types are well-encapsulated, i.e. offer the environment no opportunity to alter their internal state. This is the case for example for `getSpace()` - this apparent breach of encapsulation is actually harmless, as `Space` (and any well-designed children) are well-encapsulated.

- the linear combination method `linComb` is of course not `const`, as it is not "logically const" (Stroustrup, 1995).

- The only assignment interface is `zero`, as the zero vector is the only vector well-defined without reference to coordinates.

- We provide default implementations of a number of convenient vector manipulations, like `scale`, in terms of `linComb`. This is efficient when `linComb` (or rather the code in the `Space` subclass to which it delegates) is written so that no unnecessary multiplications or additions are required when the constants take values ± 1.0 and 0.0. Such careful implementation turns out to be quite straightforward. Nonetheless these methods are also declared virtual, so could be overridden.

- All method bodies making nontrivial calls are enclosed in `try` blocks; exceptions are cast in the form of `SVLException` objects. `SVLException` subclasses `std::exception`, so `SVLException`s are caught by standard exception-catchers and have standard exception syntax.
- Every SVL object has two reporting methods, both named `write`, outputting information to `ostream`s and `SVLException`s respectively.
- `Vector` is concrete: *all* of the methods listed above, along with other methods not explicitly shown, are *implemented in the base class*.

Construction of a `Vector` must somehow involve allocation of storage for its data. Also, if any nontrivial computations are to be supported, some mechanism must be provided for interaction with the data owned by `Vector`. As the code fragment above suggests, the `Vector` class itself does not expose its data - no indexing operator or method returning a pointer to `Scalar` appears in the class declaration. Both creation of vectors and interaction with their internal data involve the "data management" layer of SVL, to which we now turn.

GENERAL CONSIDERATIONS FOR THE DESIGN OF DATA MANAGEMENT CLASSES

SVL provides uniform data container interfaces to store `Vector` data: each `Vector` owns a data container. A first inclination might be to define a base data container type that exposes its data, enabling functions defined externally to the `Vector` class to alter the internal state of `Vector`s. Indeed, the data containers in the standard C++ library do just that, via either an indexing operator or a method returning a pointer. HCL's `HCL_Vector` interface combined a computational homolog of the mathematical concept, as described in the last section, with a data container role, and offered direct access to data through its `Data()` method (returning a pointer). We argue however that direct data access is unacceptable in an *abstract* data container base class. [The standard library data containers, of course, are all concrete, so this argument does not criticize them.] For example, it ought to be possible to treat a Cartesian product of data containers as a subtype of the abstract data container type. However if the product data container is required to provide *both* a global reference to data *and* to be constructed from existing factors, then either data must be copied or `operator[]` must be overloaded. However a virtual `operator[]` cannot be inlined, so either option leads to unacceptable inefficiency if designed into an *abstract* data container base type. Even more obviously, mandating a global reference to data in a distributed data container introduces a potentially bothersome amount of network traffic. Thus inclusion of direct data access in an abstract data container base class entails either serious performance penalties or declaration of class methods which are not always implemented. Neither feature is acceptable in a well-structured library.

Of course data must be extracted from *something* in raw form, eventually, either through a function call (`operator[]`) or in the form of a pointer - else nothing can be done with it! Therefore SVL separates the data container concept into two levels:

- the `LocalDataContainer` base (sub)class - which provides direct access to raw data
- the `DataContainer` base class - does not reference data at all,

`LocalDataContainer`s are the primary data storage structures in SVL. Of the two options mentioned above for direct data access - indexing operator vs. pointer - the first may appear preferable, as it permits a slightly greater level of abstraction in programming. However if the `LocalDataContainer` type is to be an interface, rather than a concrete type, the indexing operator must be virtual - but then it cannot be inlined. To avoid introducing virtual function call overhead into inner loops, SVL chooses the second route: the base `LocalDataContainer` interface declares a method returning a pointer to `Scalar`, along with a length:

```
template<class Scalar>
class LocalDataContainer: public DataContainer {
public:
  ...
  virtual int getSize() = 0;
  virtual Scalar * getData() = 0;
  ...
};
```

Many simple scientific data structures are naturally `LocalDataContainer` subtypes. The simplest of these is the usual array data structure:

```
template<class Scalar>
class RnArray: public LocalDataContainer<Scalar> {
private:
  int n;
  Scalar * a;
  ...
  RnArray() {}
public:
  RnArray(const RnArray<Scalar> & x)
    : n(x.n), a(new Scalar[n]), ... { for (int i=0;i<n;i++) a[i]=x.a[i];  }
  RnArray(int _n)
    : n(_n), a(NULL),... { // sanity check }
  ~RnArray() { if (a && ...) delete [] a; }
  virtual DataContainer * clone() { return new RnArray<Scalar>(getSize());  }
  int getSize() { return n; }
  Scalar * getData() {
    if (!a) a = new Scalar[n];
    return a;
   }
```

```
  ...
};
```

Note that the underlying array memory gets allocated only when it is needed, i.e. the first time `getData` is called on the object. This design for local data containers is an important efficiency ingredient in the overall design of SVL: it permits construction of high-level objects, which own instances of low-level data management types like `RnArray`, while deferring storage allocation until it is actually needed. The deferral happens in the data management layer, where it belongs. Other packages, such as HCL and TSFCore, accomplish this goal via dynamic allocation of high-level objects, which entangles the abstract expression of algorithms with low-level memory management issues.

An important caveat: the copy constructor implements a deep copy, so memory is allocated on (copy) construction. However instantiation of `RnArray` instances in high-level SVL objects typically occurs via the main constructor (the second listed above), which defers memory allocation as described.

A slightly more complicated `LocalDataContainer` subtype stores gridded data. Its implementation subclasses `RnArray`, adding a further attribute storing the grid information. The *An example: Data Gridding* section describes two such subtypes.

More complex scientific data structures are often *products*, with many similar factors. The factors are organized into a tree of `DataContainer`s, the leaves (terminal nodes) being `LocalDataContainer`s. The implementation of these product `DataContainer`s is discussed below.

The primary purpose of SVL's `DataContainer` interface is to provide a uniform method for *indirect* interaction with data across a wide range of data structures. Interaction with the data contained in a `DataContainer` occurs *only* via evaluation of a *function object*, an encapsulation of function which can have persistent state (Stroustrup, 1995). The function object concept also plays a large role in the standard library; the close relationship of the SVL and standard library uses of this concept will become clearer in the discussion of product structures below. SVL's abstract base class for function objects is `FunctionObject`.

The base class `DataContainer` offers a pure virtual *evaluation* method (with several overloads - unary, binary, ternary,...), denoted `eval`, which accepts a `FunctionObject` and zero or more data container references as arguments. The `eval` methods apply the function represented by the function object *internally*, consistent with our decision not to expose data in `DataContainer` classes. This approach to operator implementation avoids breaking encapsulation, and yet permits the functions to be coded externally. The only obvious drawback we can identify in this approach to operator implementation is its strangeness, for those (everyone?) used to implementing functions whose interfaces work directly with data in raw form, extracted from the objects on which they operate.

One of the chief roles of data containers is reproduction. Accordingly, `DataContainer` also offers a `clone` method which returns a dynamically allocated `DataContainer` with the same structure (i.e. the same type) as the `DataContainer` on which it is called. This

is not a virtual copy constructor - no guarantee is provided that the data in the returned `DataContainer` is the same as that in its source object, or even that it is initialized at all.

A near-complete listing of the base `DataContainer` declaration follows, missing only trivial constructors and destructor, the reporting methods required of all SVL classes, and a couple more overloads of `eval`:

```
class DataContainer: public DataObject {
public:
  ...
  virtual DataContainer * clone() = 0;
  virtual void eval(FunctionObject & f) = 0;
  virtual void eval(FunctionObject & f,
                    DataContainer & x) = 0;
  ...
};
```

Note that, unlike `LocalDataContainer`, `DataContainer` is an actual class, rather than a class template. Since there is no interaction with intrinsic-type data expressed in the `DataContainer` interface, there is no need to specify such a type!

The function object concept is central to SVL: all interaction with data in SVL applications takes places through evaluation of `FunctionObject`s, and application development tends to consist largely of writing `FunctionObject` subclasses. As the listing of `DataContainer` source reveals, the base `FunctionObject` type is actually an instance of the *Visitor* design pattern (Gamma et al., 1994), and so is characterized by the way in which it is used, rather than by its actions (methods). Bartlett's RTOp design is the direct inspiration of our `FunctionObject` type (Bartlett et al., 2004). As Bartlett points out, the concept implemented by RTOp (and `SVL::FunctionObject`) is *function forwarding*, which is also used for example in the Standard Library. In fact we took our name for the concept from Stroustrup's name for the similar construct in the Standard Library (Stroustrup, 1995).

The Visitor character of `FunctionObject` makes a very simple base type attractive, with essentially no methods other than reporting and assertions about its effect on the `DataContainer` on which it is evaluated. Here is an essentially complete declaration for the `FunctionObject`base class:

```
class FunctionObject: public DataObject {
public:
  virtual bool readsData(int i=0) { return true; }
  virtual bool writesData(int i=0) { return true; }
  virtual string name() = 0;
  virtual void write(SVLException & e);
  virtual ostream & write(ostream & str);
};
```

The readsData and writesData methods take an integer argument, because evaluation of a FunctionObject may involve more than one DataContainer. The "anything goes" defaults may have useful overrides - one nontrivial use of these functions is discussed below, in the section on Functions and Operators.

The base class, exhibiting the Visitor character of the type, is really a placeholder for one of several subclasses which define interaction with LocalDataContainers and so provide uniform interfaces for computation. These must be class templates, since the LocalDataContainers are templated. The unary case is

```
template<class Scalar>
class UnaryFunctionObject: public FunctionObject {
public:
  virtual void operator () (LocalDataContainer<Scalar> &) = 0;
};
```

SVL currently defines similar binary, ternary and quaternary interfaces. A natural extension would accomodate vectors of LocalDataContainers of arbitrary length; no need for such an interface has arisen so far.

This set of interfaces suffices for FunctionObjects, such as linear combination or assignment to constant, that produce no result other than possible effect on the operator() arguments. However others, for example implementation of an inner product, produce some distinct result of a non-DataContainer type. The Reduction class provides access to a result:

```
class Reduction {
protected:
  RetType & result;
public:
  Reduction( RetType & res) : result(res) {}
  virtual ~Reduction() {}
  virtual void setResult() { result.reinitialize();  }
  virtual void setResult(RetType & res) { result = res; }
  virtual RetType & getResult() { return result; }
```

RetType is an abstract result return type, which supports assignment and has a distinguished value which can be recovered through RetType::reinitialize. A subclass, ScalarRetType, offers the features of a scalar, i.e. the usual arithmetic operations, and the ability to return an object of the template Scalar type.

Reduction FunctionObjects multiply inherit from the FunctionObject and Reduction interfaces. Since Reduction is concrete, no additional result manipulations need be implemented. There is one reduction FunctionObject type for each base FunctionObject type:

```
template <class Scalar> class UnaryFunctionObjectRedn:
  public UnaryFunctionObject<Scalar>, public Reduction { ... }
```

The virtue of an abstract return type lies in the inconvenience of the alternative, viz. to make the return type of a reduction `FunctionObject` a template parameter. That method of introducing the type of the result object forces the calling unit to know - explicitly, at compile time - the type which the function returns. Occasions arise when this information is not conveniently available, for example in distributed execution, when the return value need not be accessed in a component that references the `FunctionObject`.

For convenience, a subclass `UnaryFunctionObjectScalarRedn` is provided (with the usual binary, ternary, and quaternary overloads), based on `ScalarRetType`, with an additional `Scalar getValue()` method which decodes and returns the `Scalar` stored in the `ScalarRetType`. This class provides immediate access to a `Scalar` function value, without requiring the user to deal with `RetType` or its subclasses. The various universally applicable `FunctionObject`s declared in `functions.H`, implementing standard scalar functions (max, min, dot product,...), are subtypes of `...FunctionObjectScalarRedn`.

In some circumstances (eg. parallel execution) it is necessary to aggregate a partial result, encoded in a `RetType` object, into the internal buffer of a reduction `FunctionObject`. The `Accumulation` type provides a uniform abstract mixin interface for accumulation of results. We refer the reader to the documentation for the standard functions mentioned above, some of which inherit from and implement this interface.

For `LocalDataContainer`s, `eval` methods inherited from `DataContainer` are implemented in the base class: for example,

```
void LocalDataContainer<Scalar>::eval(FunctionObject & f) {
  try {
    UnaryFunctionObject<Scalar> & ufo =
      dynamic_cast<UnaryFunctionObject<Scalar> &>(f);
    ufo(*this);
  }
  catch (bad_cast) { ... }
  catch (SVLException & e) {
    e<<"\ncalled from LocalDataContainer::eval(unary)\n";
    throw e;
  }
}
```

Writing a traceback message to a trapped exception is standard throughout SVL.

The `DataContainer::clone()` method necessarily remains virtual in the base `LocalDataContainer` class.

STRUCTURE OF SVL VECTOR CLASSES

The basic relationships between the SVL vector and space classes are now easy to describe. The main attributes of `Space` are (i) a collection of operations on `DataContainer`s, which implement linear combination and inner product, (ii) a (virtual) constructor for `DataContainer`s, which returns dynamically allocated `DataContainer`s for `Vector` data storage. Thus `Space` is an example of the *Abstract Factory* pattern, with some additional methods. The important thing to note here is that both the factory and the additional methods manipulate `DataContainer`s, not `Vector`s. HCL, in contrast, furnished its `HCL_VectorSpace` class with a virtual constructor for `HCL_Vector`s.

The declaration of `Space`, reproduced in stripped-down form here, realizes these ideas:

```
template<class Scalar> class Space {
public:
  virtual DataContainer * buildDataContainer() const  = 0;
  virtual bool operator ==(const Space<Scalar> & sp) const = 0;
  virtual bool operator !=(const Space<Scalar> & sp) const {
    try { return !operator==(sp); }
    catch (SVLException & e) { ... }
  }
  virtual bool isCompatible(DataContainer & dc) const = 0;
  virtual Scalar inner(DataContainer & x, DataContainer & y) const = 0;
  virtual void zero(DataContainer & x) const = 0;
  virtual void linComb(DataContainer & z,
                       Scalar a, DataContainer & x,
                       Scalar b, DataContainer & y) const = 0;
  virtual void write(SVLException & e) const = 0;
  virtual ostream & write(ostream & str) const = 0;
};
```

To implement a `Space` subclass directly under the base class, you must provide

- an implementation of `buildDataContainer()`, usually by passing the object data of a `DataContainer` class to its constructor via operator `new`. The object data must be passed to the constructor of the `Space` subclass (or computed from arguments to the constructor) and stored. For example, an `RnSpace` takes just its dimension as object data, as that's all the `RnArray` constructor requires (see above).

- the comparison `operator==`, using appropriate dynamic casts to expose whatever part of the comparison `Space` object's state is required to be the same for the comparison to return true. Two `RnSpace`s are the same if they have the same dimension, so checking the dimension is sufficient. On the other hand a `Space` based on a grid structure would only be idenfied with an instance of a similar `Space` if the grid data were identical.

- implementations of the linear algebra methods, usually by calling appropriate `FunctionObject`s. SVL supplies a set of standard `FunctionObject`s, declared in the `functions.H` header file, of which the linear combination and set-to-zero functions may be regarded as universal. The inner product function in this collection is the *unscaled* l^2 inner product, which may be combined with a Gram matrix (implemented via another, user-supplied `FunctionObject`) to produce an arbitrary inner product.

To render this set of tasks slightly easier, SVL provides the `StdSpace` subclass, which implements the required methods by delegation to

- a `DataContainerFactory`, which is the repository of the object data necessary to build the correct type of `DataContainer`. This type is an instance of the *Factory* design pattern (Gamma et al., 1994).
- a `LinearAlgebraPackage`, which can return references to the necessary three `FunctionObject`s implementing linear combination, zero initialization, and inner product. The header file `functions.H` also defines a standard `SVLLinearAlgebraPackage` which returns canonical examples of these `FunctionObject`s.

`StdSpace` implements all of the `Space` methods by delegation to these two types, accesed via a pair of virtual "get" methods returning references. This last renders `StdSpace` abstract, of course, but provides flexibility: the `DataContainerFactory` and `LinearAlgebraPackage` references returned by the "get" methods may be initialized either internally, during construction of the `StdSpace` subclass instance, or by assignment from externally initialized references passed to the subclass constructor. For more details see the reference manual (Symes and Padula, 2004).

`Space` defines linear algebra, `DataContainer` stores the data, so what does `Vector` do? In fact, `Vector` is a *Composite* design, combining a `Space` and a `DataContainer`: the `Space` "tells the `DataContainer` how to act like a vector". A slightly more complete listing, including the code for the main `Vector` constructor, reveals that `Vector` is also a *handle* (to a `DataContainer`):

```
template<class Scalar>
class Vector {
...
private:
  const Space<Scalar> & sp;
  DataContainer * d;
  ...
public:
  ...
  Vector(const Space<Scalar> & _sp, bool initZero = false)
    : sp(_sp), d(sp.buildDataContainer()),... {
```

```
    try { if (initZero) zero(); }
    catch (...) {...}
  }
  virtual ~Vector() { if (d) delete d; }
  ...
  void eval(FunctionObject & f) const {
    ...
    try { d->eval(f); }
    catch (...) {...}
  }
  ...
  Scalar inner(Vector<Scalar> & y) {
    if (sp != y.sp) {...} // throw exception ...
    try { return sp.inner(*d,*(y.d)); }
    catch (...) {...}
  }
  ...
};
```

The listing shows how the `Space` reference and the `DataContainer *` data member interact to implement `Vector's` linear algebra operations and `FunctionObject` evaluations. The linear algebra methods of `Space` are all that is required to implement coordinate-invariant linear equation, nonlinear equation, and optimization algorithms, such as Krylov space and quasi-Newton methods. Appropriate `FunctionObject`s can be evaluated to implement more coordinate-dependent operations, such as those appearing in currently popular versions of interior point algorithms for convex programming.

The `Space` reference refers to an external object, whereas the `Vector` object manages the `DataContainer` 's storage. [There is actually one circumstance in which `Vector` does not manage its `DataContainer *` data member - that exception will be discussed below.] The handle role is clear - but `Vector` is what one might call a "closed" or *intrusive* handle (Stroustrup, 1995): the dynamically allocated object which `Vector` handles is allocated internally, not externally through the usual overloaded assignment operator. Because the environment has no direct access at all to the `DataContainer *` data member, no reference counting is necessary: the reference to the dynamically allocated data is not shared with any other object. Partly for this reason, SVL does not employ a pervasive reference-counting scheme (this will be discussed further below, in the *Implementation Details* section). [In the exceptional case mentioned above, in which `Vector` does not have exclusive access to the `DataContainer` to which its data member points, a simple ad hoc form of reference counting suffices - access to the data member is severely restricted, and the necessary reference counting is vastly simpler than that incorporated in the typical smart pointer class ((Langtangen, 1999), for example).]

The chief impact of this hiding of virtual construction behind concrete construction is that algorithms do not need to allocate `Vector` workspace dynamically - it can be

allocated on the stack, and this largely gets algorithm developers out of the memory management business. Algorithms needing internal workspace (i.e. virtually all algorithms) must construct appropriate temporary vectors. HCL's `HCL_Vector` is abstract, and therefore the only possible workspace allocation is via a virtual constructor, located in `HCL_VectorSpace`, which returns control of a dynamically allocated object to the calling unit, which must therefore manage it. In contrast, `Vector` manages its own memory. Applications instantiate `Vector` objects by actual, rather than virtual, construction. Algorithms needing only vector workspace are thus relieved altogether of the need to manage dynamic memory.

Other core SVL "calculus" classes are also so structured that dynamic allocation is unnecessary. This feature of SVL invites a simplified, stack-oriented programming style for the "calculus" layer which achieves efficiency and safe memory management without the apparatus of smart pointers and reference counting. The section *Implementation Details* below further describes this stack-oriented style and its consequences for programming optimization and iterative linear algebra algorithms.

FUNCTIONS, OPERATORS, AND EVALUATIONS

Of course in addition to the linear algebra and other vector operations, optimization algorithms also require the evaluation of scalar- and vector-valued functions. Indeed the very definition of the problems that these algorithms solve requires that these mathematical concepts be manifested in code. In turn, types for scalar- and vector-valued vector functions provide natural interfaces for simulators and associated computations.

It is important to understand that `FunctionObject` is not an adequate abstraction for this purpose. `FunctionObject` behaviour is both too flexible and too rigid to represent the mathematical concept of function of a vector variable. `FunctionObject`s, for instance, do not refer to `Vector`s, but to low-level data containers. Because `FunctionObject`s have persistent internal state, the effect of a `FunctionObject` evaluation on any of its arguments may differ from call to call. This mutability is actually very useful, but is incompatible with the intrinsic properties of a (mathematical) function. On the other hand, the way in which a `FunctionObject` evaluates on a Cartesian product object, described in the next section, is rigidly prescribed and is far more limited than the possibilities open to a function.

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

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

Mathematically, all three of these possibilities are special cases of a general vector function type. However computationally it is not convenient to derive all three types from a common parent. C++ distinguishes between intrinsic and object types (unlike Java,

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

In some cases the distinction is actually consistent with mathematical usage. For instance, the vector-valued function interface 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. SVL conforms to these habits of thought, returning the derivative in the `Operator` type as a `LinearOp`, in the `Functional` type as a `Vector`.

Linear operators

`LinearOp` is the simplest interface of the three, for various reasons. A complete class declaration, stripped of comments, trivial con/destructors, and the usual reporting functions, is:

```
template <class Scalar> class LinearOp {
public:
  virtual const Space<Scalar> & getDomain() const = 0;
  virtual const Space<Scalar> & getRange() const = 0;
  virtual void apply(const Vector<Scalar> & x, Vector<Scalar> & y) const = 0;
  virtual void apply(Scalar alpha, const Vector<Scalar> x,
     Scalar beta, const Vector<Scalar> y,
     Vector<Scalar> z);
  virtual void applyAdj(const Vector<Scalar> & x, Vector<Scalar> & y) const = 0;
  virtual void applyAdj(Scalar alpha, const Vector<Scalar> x,
    Scalar beta, const Vector<Scalar> y,
    Vector<Scalar> z);
  bool checkAdjointRelation(UnaryFunctionObject<Scalar> & randomize,
                            ostream & str, Scalar tolfac = 100.0) const;
  ...
};
```

Several things to note:

- the operator identifies its domain and range. Typically these will be stored as `const Space &` data members, and typical use is to generate workspace, eg.

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

The alternative would be something like `getVectorInDomain()`, which would respect data encapsulation but entail memory management and offer less flexibility. Therefore SVL accepts the less-than-perfect encapsulation, using `const` in conformance with the widespread but naive belief that it helps in such cases. In fact, this exposure of internal data is completely harmless, but not because of the use of `const` (which could always be cast away): the C++ reference semantics do not allow the references themselves to be changed. Moreover, the `Space` class (and any well-designed subclass) offers no methods exposing internal state, so exposure of a `Space` data member is completely harmless.

- the use of `const` is conventional, i.e. aggressive - every method that could be either bitwise or logically `const` is so declared, likewise every argument and return value.

- along with a method to apply the operator, a method is included to apply the adjoint. Unlike the mathematical homolog, a `LinearOp` object does not automatically have its adjoint defined when the operator itself is defined - a separate implementation must be supplied. Like the mathematical homolog, "adjoint" here means: with respect to the inner products defined in range and domain, which as noted above are attributes of the `LinearOp` object. Logically, then a `LinearOp` is really a *pair* of linear operators, adjoint to each other. It's usually convenient to build one of these first, then then other. During the construction of the `apply` method, the `applyAdj` method can be implemented to throw an exception, for instance. Rather than use a flag, as does for example TSFCore, to indicate which of the adjoint pair is being applied, SVL simply uses two distinct method names.

- Overloads of the `apply` methods are supplied which include linear combination with a vector, i.e.

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

and the analogue for the adjoint. These are provided default implementations in terms of the basic `apply` methods (i.e. in terms of $z \leftarrow Ax$ and linear combination) but may be overridden to supply a degree of loop fusion. This device, popular in other OO numerics libraries, is less useful than one might think; annoyingly often, algorithms require access to Ax as well as to the linear combination, so nothing is gained. In fact the problem which this overload (sometimes) solves is an instance of a very important open problem in computer science, that of cross-type method optimization. A good solution seems to be some way off. In the meanwhile, the preferred SVL solution is to write special nonlinear operators (of which the `apply` overloads are actually instances) and provide their implementations with appropriately fused loops.

- the `checkAdjointRelation` method is an example of a *built-in unit test*, a device which SVL uses wherever possible (as did HCL - this test was an attribute of

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

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

The top of the checkAdjointRelation implementation listing is instructive:

```
template<class Scalar>  bool LinearOp<Scalar>::checkAdjointRelation
(UnaryFunctionObject<Scalar> & randomize,
 ostream & str,
 Scalar tolfac) const {
  try {
    Vector<Scalar> xin(getDomain());
    Vector<Scalar> yin(getRange());
    Vector<Scalar> xout(getDomain());
    Vector<Scalar> yout(getRange());

    xin.eval(randomize);
    yin.eval(randomize);

    apply(xin,yout);
    applyAdj(yin,xout);
    ...
```

Note the use of the domain and range access methods together with the standard Vector constructor to create workspace on the stack. This is a typical workspace allocation in SVL algorithm code.

Note also that a function object is passed to the method call, whose function is to randomly initialize the workspace. Random initialization is not a basic linear algebra function, and has been excluded from the Vector interface. Therefore it must be implemented as a FunctionObject. As a corollary, the linear operator class does not guarantee correctness of this function object - it is external. The

contract between the algorithm writer (in this case also the class designer!) and the user is this: if the user supplies a function object which randomly initializes the components of a `LocalDataContainer`, then this method reveals whether the implemented operator and its adjoint, applied to random vectors, produce results which stand in the correct relation vis-a-vis the inner product. This sort of guarantee is maximal.

Nonlinear functions and evaluation objects

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

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

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

SVL Evaluation objects are an evolution of HCL's, somewhat simpler in structure and requiring less (and in some ways more natural) code from the application developer. A (fully initialized) SVL Evaluation is a pair consisting of an (invariant) reference to a `Vector` and an independent copy of the object representing the function (`Operator` or `Functional`). All access to function values in SVL occurs through Evaluations.

SVL Evaluations are instances of the Composite pattern, reminiscent of `Vector`: they are handles to function types, and use their `Vector` member to tell the function how to behave (where to evaluate itself), just as `Vector` uses its `Space` reference to tell its `DataContainer` how to behave like a vector.

SVL Evaluation classes manage the storage of function results: they retain values in data members and return references to them. In order to avoid the dangling reference problems that would otherwise result, all data members exposed by reference are allocated on the stack, i.e. their lifetimes are identical to those of the Evaluations that own them. It might be objected that some results could contain significant data (eg. a `Vector` such as a computed gradient), whence their allocation should be deferred until possible use. Such a strategy would require dynamic allocation of the these results, and ultimately reference counting. However, as pointed out previously, the preferred approach to deferred allocation in SVL is to provide storage on demand *at the lowest possible level*, not in high level constructs. For example, the `LocalDataContainer` components which ultimately store the data of a `Vector` should defer dynamic allocation of their data arrays until these are requested, as illustrated above in the implementation of `RnArray`. Thus member initialization of a gradient `Vector`, for example, does not immediately allocate a great deal of storage. This strategy accomplishes the desired goal - for example, if only `Functional` values are required, memory for gradient data is never allocated - without requiring dynamic allocation of high-level constructs like `Vector` or the attendant need for bookkeeping of shared references.

An objection to this design should immediately occur to the reader: what is to guarantee that the referenced evaluation point remains consistent with the results managed by the `FunctionalEvaluation` object? There is no way to use `const` to ensure this consistency, as the relation is entirely run-time, rather than compile-time. Moreover, the evaluation point is not necessarily constant during the lifetime of the `FunctionalEvaluation` object. In essence, a `FunctionalEvaluation` is semantically akin to the expression $f(x)$, in its typical mathematical usage. The data contained in x may change within its mathematical scope, and with it the value $f(x)$, but in $f(x)$ these are dynamically linked.

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

- transparent to the application developer;
- (because it is) implemented in the base classes;
- a run-time relation between internal states, rather than a compile-time language feature;
- secure, because the `Vector` methods implementing it are protected.

The *watch* relationship uses a simple version tracking scheme. Application to the evaluation point `Vector` of any non-`const` linear algebra operation, or evaluation on it of an `FunctionObject` which writes data, advances the version index (an `int` data member of `Vector`). The Evaluation's copy of the version index is then smaller. Calling any of the access methods then causes all internal data to be recomputed, and the Evaluation's copy of the version index to be updated to match the evaluation point's. This design makes nontrivial use of the `FunctionObject::writesData` method to detect modification of the

evaluation point. Since `writesData` returns true by default, only deliberate intervention by the programmer can avoid recomputation of all internal Evaluation data.

Evaluation objects also expose some of their internal data via access methods returning references. For example, `FunctionalEvaluation` owns data members storing function value, gradient, and Hessian, and returns references to them for external use in algorithms. For the scalar value, returned by value, and the Hessian, which is returned as well-encapsulated `HessianEvaluation` object, this exposure poses no risk to the respective internal states. However the `Vector` representing the gradient are in principal susceptible to alteration. As only the Evaluation object itself can be permitted to alter this data, some means of making it read-only is needed. All such access methods return `const` references, but in addition `Vector` provides a `lock` attribute which `FunctionalEvaluation` sets on its gradient `Vector` data member, for example. A locked vector throws an exception if a non-`const` linear algebra method or `operator()` of a `FunctionObject` with `writesData()==true` is called on it.

Both *watch* and *lock* mechanisms are forms of reference counting. Unlike the typical reference count used in smart pointer classes and the like, their function is to protect the internal state of the watched or locked object, rather than regulate its lifetime. These devices permit Evaluation objects to safely use references to evaluation points rather than copies, enhancing efficiency. They also permit Evaluation objects to expose `Vector` results without risking data corruption. All of this happens transparently to the user - unless the user attempts to alter locked vectors!

Note that SVL Evaluation classes are *concrete*: all methods are implemented. The application developer need only write the appropriate `Functional` or `Operator` classes, and can simply use the corresponding Evaluations in algorithm implementations. This is an important difference with HCL, whose Evaluation classes are abstract and must be subclassed by the HCL user.

`Functional` and `FunctionalEvaluation`

The description of Evaluation as a composite handle to function implies the entire design of both classes, at least approximately. We will describe the scalar-valued function class `Functional` and its associated Evaluation, `FunctionalEvaluation`, in detail. The structure of the corresponding classes for vector-valued functions is precisely parallel.

Since evaluation will take place within *independent copies* or clones of a function type instance, one clone for each evaluation point, the function types are intended to store all intermediate data needed in any evaluation, in principle as write-once, read-many data. The base `Functional` class makes no attempt to specify the precise form of this storage, of course: it simply provides pure virtual interfaces for evaluation of a function and some of its derivatives. Since a `Functional` instance is intended to be cloned, an efficient implementation will allocate storage for intermediate data dynamically, as needed. The `Functional::apply` (evaluation) methods need be accessible *only* to Evaluation objects, as the latter provide the only access to values. Accordingly, the `apply` methods are protected, and the `FunctionalEvaluation` class is a friend of `Functional`.

A difficult problem, not entirely resolved in scientific computation, is the description of more or less arbitrary subsets of high dimensional spaces. Since the domain of an arbitrary function can be such a set, we are faced directly with this problem. In view of the intended use in optimization, HCL chose to adopt an implicit approach, including a method in the function classes returning signed distance to the boundary of the domain in a specified direction, from a specified point. SVL has also adopted this approach.

These considerations completely specify the structure of the `Functional` class, a stripped listing of which follows:

```
template<class Scalar> class Functional {
  friend class FunctionalEvaluation<Scalar>;
protected:
  virtual Functional<Scalar> * clone() = 0;
  virtual void apply(const Vector<Scalar> & x,
                     Scalar & val) const = 0;
  virtual void applyGradient(const Vector<Scalar> & x,
                             Vector<Scalar> & g) const = 0;
  virtual void applyHessian(const Vector<Scalar> & x,
                            const Vector<Scalar> & yin,
                            Vector<Scalar> & yout) const = 0;
public:
  ...
  virtual const Space<Scalar> & getDomain() const = 0;
  virtual Scalar getMaxStep(const Vector<Scalar> & x,
                            const Vector<Scalar> & dx) {
    return numeric_limits<Scalar>::max();
  }
  bool checkGradient(const Vector<Scalar> & y,
                     const Vector<Scalar> & p,
                     ostream & str,
                     int n=10,
                     Scalar hmin=0.1,
                     Scalar hmax=1.0,
                     Scalar minrat=1.95);
  bool checkHessian(const Vector<Scalar> & y,
                    const Vector<Scalar> & p,
                    ostream & str,
                    int n=10,
                    Scalar hmin=0.1,
                    Scalar hmax=1.0,
                    Scalar minrat=1.95);
  ...
};
```

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

We emphasize that Evaluation classes are concrete, and do not have to be re-implemented by the SVL user.

A partial listing of `FunctionalEvaluation` displays the necessary data members, and the way in which they are initialized and used:

```
template<class Scalar> class FunctionalEvaluation {
  friend class HessianEvaluation<Scalar>;
private:
  const Functional<Scalar> & fref;
  mutable Functional<Scalar> * f;
  WatchedVecRef<Scalar>  wx;
  Scalar val;
  Vector<Scalar>  grad;
  HessianEvaluation<Scalar> h;
  mutable bool applied;
  mutable bool gapplied;
  void reset() const {
    try { if (f) delete f; f = fref.clone();
      applied=false; gapplied=false; }
    catch (SVLException & e) {...} }
  ...
public:
  FunctionalEvaluation(const Functional<Scalar> & ff,
                       const Vector<Scalar> & x)
    : fref(ff), f(fref.clone()), wx(x), grad(fref.getDomain()),
      h(*this), applied(false), gapplied(false) { grad.setLock(); }
  ...
  virtual const Space<Scalar> & getDomain() const
    { return fref.getDomain(); }
  const Vector<Scalar> & getPoint() {
    try { return wx.get(); }
    catch (SVLException & e) {...} }
  Scalar getValue() {
    try {
      if (wx.update()) reset();
      if (!applied) { f->apply(wx.get(),val); applied = true; }
```

```
      return val; }
    catch (SVLException & e) { ... }  }
  const Vector<Scalar> & getGradient() {
    try {
      if (wx.update()) reset();
      if (!gapplied) {
        grad.relLock();
        f->applyGradient(wx.get(),grad);
        grad.setLock(); gapplied = true; }
      return grad;
    catch (SVLException & e) { ... }  }
  const LinearOp<Scalar> & getHessian() { return hess; }
  ...
};
```

A `const` reference to the `Functional` is maintained, and a working copy returned by `Functional::clone` (allocated dynamically) on construction and whenever the watch mechanism detects an evaluation point update. The `WatchedVecRef` data member manages the evaluation point versioning via (protected) methods of `Vector`. All references to the evaluation point within the implementation of this class delegate to this `WatchedVecRef` data member via its `get()` method; its `update()` method returns a boolean flag indicating an update. The gradient workspace `grad` belongs to the `FunctionalEvaluation` object, so a *lock* relationship is more appropriate than a *watch* relationship: no external entity is permitted to modify the gradient during the lifetime of the `FunctionalEvaluation` object. Data members (`val, grad, hess`) are uninitialized up to the point of use. Since these variables own no intrinsic attribute indicating initialization, the class uses boolean flags (`applied` etc.) to manage the required (re)computations. The Hessian is stored in a `HessianEvaluation` object `hess`. `HessianEvaluation` is a subtype of `LinearOp`; its data consists only of a reference to the parent `FunctionalEvaluation`, to which it delegates its `apply...` methods. The `applyHessian` method of `FunctionalEvaluation` monitors the evaluation point reference for updates, so neither `HessianEvaluation` nor `FunctionalEvaluation::getHessian` need to.

The core code of a limited memory quasi-Newton (BFGS) implementation shows a typical use of these classes. For a mathematical description of this algorithm, consult (Nocedal and Wright, 1999). In this code, `H` is a BFGS inverse Hessian approximation, and `ls` is an object encapsulating a line search algorithm. The current point is `x`, for which temporary storage `xprev` is also provided. The update of `H` requires both the current and previous gradients, so another `Vector`, `gprev`, is also provided for gradient storage. The search direction is `dir`, and `fx` is a `FunctionalEvaluation` object representing the 2-jet of the function to be optimized. In our realization, `H` is a `LinearOp`, and `ls` is an `Algorithm` object (as is the object encapsulating the BFGS algorithm). The Algorithm package, an offshoot of the SVL project described in more detail below, defines this `Algorithm` type: the chief attribute of an `Algorithm` is that you can `run` it (Padula, 2004).

```
...
FunctionalEvaluation<Scalar> fx(f,x);
...
bool run() { try {
    // compute current search direction
    H.apply(fx.getGradient(), dir);
    dir.negate();

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

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

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

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

The externally defined line search object `ls` acquires references to the base point, search direction, and function to be evaluated via its `set` method. A *dynamic reference* helper class enables dynamic initialization of references, which C++ reference semantics forbids in its straightforward form.

CARTESIAN PRODUCT STRUCTURES

As mentioned earlier, all scientific data structures other than the very simplest are Cartesian products. SVL's realization of Cartesian product data structure is the `ProductDataContainer` interface. `ProductDataContainer` provides abstract access to components of a Cartesian product `DataContainer`, and defines evaluation of `FunctionObject`s by delegation to the components. This is very similar to the `foreach` mechanism of the C++ standard library (see (Stroustrup, 1995)) and was inspired directly by Roscoe Bartlett's RTOp concept (Bartlett et al., 2004), used extensively in TSFCore (Bartlett et al., 2003) and several other recently developed packages from Sandia National Laboratories (Heroux et al., 2003).

```
template<class Scalar>
class ProductDataContainer: public DataContainer<Scalar> {
public:
  ...
```

```
  virtual int getSize() = 0;
  virtual DataContainer<Scalar> & operator[](int i) = 0;
  ...
  virtual void eval( BinaryFunctionObject<Scalar> & f,
     DataContainer<Scalar> & x) {
    try {
      ProductDataContainer<Scalar> & xp
        = dynamic_cast<ProductDataContainer<Scalar> &>(x);
      if (xp.getSize() != getSize()) { // throw exception }
      for (int i=0;i<getSize();i++) { (*this)[i].eval(f,xp[i]);  }
    }
    catch (bad_cast) { // throw exception }
    catch (...) { // throw exception }
  }
  ...
};
```

Evaluation of a `FunctionObject` proceeds through delegation to the factor data containers, to which `operator[]` provides access. That is, the action of a `FunctionObject` on a Cartesian product of `DataContainer`s is formally block diagonal. Since `FunctionObject`s have persistent internal state, in fact non-diagonal actions are possible, but the interface is designed specifically to make block diagonal actions easy to implement.

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

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

There are no product `Vector`s, per se, only `Vector`s produced from `ProductSpace`s. This is necessary, since `Vector` is concrete. To enable access to components of a product vector, an independent `Components` class offers indexed access. The `Vector` is input to the `Components` constructor, which uses protected `Vector` functions to access the constituent `Space` and `DataContainer`. If these are products, the `Components` object constructs a new `Vector` for each component and offers reference access to these through

the indexing `operator[]` - if not, then the number of components is 1 and `operator[]` simply returns a reference to the data `Vector`. This device implicitly views `Vector`s composed of *nonproduct* `Space` and `DataContainer` as product vectors of length 1, without altering the formal class hierarchy - a useful feature in some settings.

The `Components` construction can be used recursively, i.e. the factors in a `Product-DataContainer`, `ProductSpace` pair may themselves be Cartesian products.

A special `Vector` constructor is required: since the `Space` and `DataContainer` data members of `Vector` are not normally accessible, converting the pair of Cartesian products (represented by a `ProductSpace` and a `ProductDataContainer`) into a Cartesian product of pairs (component `Vectors`) would otherwise be impossible. Vector offers a protected constructor, which initializes the `DataContainer *` member by storing an externally supplied address, and protected access methods exposing the `Space` and `DataContainer`. With `Components` declared as a `friend`, component access becomes possible without damage to data encapsulation.

The mechanics of component access are transparent to the user, who simply uses the constructor and index operators:

```
ProductSpace<double> & sp = ...
Vector<double> x(sp);
...
Components<double> cx(x);
for (int i=0;i<cx.getSize();i++) { cx[i].eval(....); }
```

IMPLEMENTATION DETAILS

SVL's approach to some important implementation choices differs from that taken by other OO numerics projects. In this section we argue for

- a strictly stack-oriented programming style for the "calculus" layer of SVL;
- a component design for distributed execution;
- restrictive access to global vector attributes such as dimension.

Memory Management

Efficient manipulation of data practically demands that complex objects be passed by reference, and that compound objects be built up out of data members which refer to other objects. The necessity of data motion limits libraries which do not employ reference sharing to computationally small problem domains. However reference sharing engenders two notorious memory management problems:

- dangling references: an object owning a reference to another object has no assurance that the referenced object exists at the point of use;

- ownership: if several objects share a reference, there is *a priori* no way to decide which of the many reference owners should manage the referenced memory.

The conventional solution to these problems, reference counted smart pointers, is explained in many excellent articles and books, eg. (Stroustrup, 1995; Barton and Nackman, 1994). A nice introduction to these ideas in the context of scientific computing is (Langtangen, 1999). HCL used reference counting, but not smart pointers. TSFCore uses essentially HCL's core classes in combination with a smart pointer implementation from the Teuchos library. For example, instantiation of a vector `vec` in a space `sp` in TSFCore is typically written

```
Teuchos::RefCountedPtr<Vector<Scalar> > vec = sp.createMember();
```

(Bartlett et al., 2003). Note that `TSFCore::VectorSpace::createMember()` allocates a `TSFCore::Vector` on the heap, and the address is handed to the smart pointer through the usual overloaded assignment operator mechanism. Thereafter the smart pointer manages the memory it has been assigned.

While this solution is conventional, widely accepted, and well understood, we rejected it for several reasons. First and foremost, SVL objects *are* handles, managing dynamically allocated memory just as smart pointers do. Consider the SVL statement analogous to the TSFCore vector instantiation above:

```
Vector<Scalar> vec(sp);
```

Both objects named `vec` in the two statements are allocated on the stack. The SVL statement actually involves a virtual constructor analogous to `createMember` - it is `buildDataContainer`. However the resulting dynamically allocated `DataContainer` is not accessible via the `Vector` interface, so cannot be further shared, so no reference counting is required. Similarly `Functional`, `LinearOp` and `Operator` subclasses typically manage internal data. These abstract classes, unlike the concrete `Vector`, do not specify the types of internal storage - those specifications are left to concrete subclasses. They nonetheless define interfaces for handle classes. Amongst core SVL "calculus" classes, only `Space` is an exception - it is a Factory, rather than a handle.

The obvious difference between SVL handle classes and those provided, say, by Teuchos or DiffPack is that the SVL handles are *intrusive* or closed. They do not share access to their dynamically allocated data members, and each is designed to encapsulate some particular type or types to yield a specific behavior mimicking a mathematical concept. This design is actually very familiar: `std::string` is an example. SVL can get away with such carefully tailored handle designs because the number of *abstract* mathematical types proper to calculus in Hilbert space is small and fixed: spaces, vectors in them, functions on them, and combinations of these - Cartesian products, linear combinations, compositions. Computationally, the fundamental types correspond to the interfaces and handle types supplied by SVL. SVL also supplies some of the combinations, and others can be constructed type- and memory-safely out of the base classes, following the principles

laid down here and exemplified by the library classes. The essential relations between the mathematical types are expressed rather precisely by the public methods of the base interfaces.

A natural byproduct of this restricted handle design is simplicity of use: the `SVL ::Vector` object `vec` in the example above is actually the object whose behavior is specified by the SVL design, not another object owning a reference to it. There is no need for another layer of indirection. The resulting simplification helped us meet one of the design goals of SVL: to ban low-level details such as memory management from abstract numerical algorithm formulation. The `checkAdjointRelation` unit test kernel, described in the *Functions and Operators* section, and the conjugate gradient implementation described below show how straightforward the formulation of abstract algorithms becomes. We strongly recommend comparison with code fragments from other libraries, eg. (Bartlett et al., 2003), p. 41, to clarify the implications of these choices.

We have still to explain how SVL solves the dangling-reference and ownership problems mentioned above. The principles behind the solution (which applies *only* in the "calculus" layer, see below) are:

- types for which references are shared *between type hierarchies* are allocated *only* on the stack, and sharing occurs via passing (C++) references (as opposed to pointers);
- references to objects allocated dynamically are not (directly) shared.

The types for which references may be shared are `Space`, `Vector`, `LinearOp`, `Operator`, `OperatorEvaluation`, `Functional`, `FunctionalEvaluation` and their children - i.e. the core "calculus" classes in terms of which it is possible to write essentially all abstract algorithms. For these types, `operator new` is overloaded as a protected method, hence is rendered unavailable to the application developer: these objects can be allocated *only* on the stack (in a general context - a child or friend class can of course use the protected `operator new` overloads). Since the lifetime of a stack-allocated object is the scope in which it is defined, and since a reference must be initialized to refer to an existing object, reference data members of these types must refer to already-constructed objects. The scope of an object referring to a second object must be strictly contained in the that of the second, provided that both are allocated on the stack. Thus there is no possibility that the referenced object will be destroyed while the reference is needed - that can only occur if the referenced object goes out of scope, which cannot happen before the referring object goes out of scope. Finally, reference semantics leave memory management firmly in control of the (stack-allocated) referenced object, so the ownership problem is solved as well. These principles form a viable basis for abstract algorithm expression because *the scope relations between automatically allocated objects mimic the dependencies amongst mathematical objects.*

Several aspects of these rules and their consequences deserve mention:

- they pertain to "user" applications of SVL using the "calculus" classes; internally, some SVL classes do share references to dynamically allocated data across hierarchies (eg. `Vector` and `Components`). All such references are protected by an ad-hoc

reference count. These are internal implementation details of the SVL classes, hidden at the abstract algorithm level. No need exists for similar constructions in abstract numerical code built out of the SVL calculus classes.

- the "data management" layer involves somewhat more extensive use of dynamic memory, and reference counted smart pointers might be used to some advantage there. However we have found that aggressive reliance on the stack in preference to the heap, of member initialization in preference to dynamic allocation of data members, and application whenever possible of the two rules above, eliminate almost all memory management issues in this layer as well.

- while we have made `operator new` publicly unavailable for some types, there are plenty of ways in which it's possible to abuse dynamic memory with this library - simply don't follow the rules stated above for managing "calculus" objects. The same is true of the other libraries mentioned in the introduction: no one can force you to follow programming practices appropriate to the structure of each library. Our object is to make inadvertent, rather than malicious, memory mismanagement difficult by developing a set of clear-cut use patterns for these classes.

Parallelism

The discussion so far does not address distributed implementation, for example of optimization algorithms. One approach is to parallelize the optimization code, eg. running the same application in all processes in an SPMD design. This is the approach take by TAO and by PETSc (Benson et al., 2000; Balay et al., 2001). We also intended to take this route in parallelizing HCL, at first. However we realized quickly that the necessary code modifications would be pervasive: we would end up with two versions of HCL, parallel and serial, and moreover the parallel version would be tied to a particular distribution library (presumably MPI). The resulting compromise in software integrity was unacceptable to us.

In fact, integrating data distribution information with the abstract code layer used in implementing an optimization algorithm is a grave design error. The notion that parallel performance should be achieved that way has led to conceptual abominations such as "parallel optimization", which was actually the topic of conference sessions not too many years ago. Even so-called parallel optimization algorithms, such as Torczon's PDS (Hough et al., 2001), are not really parallel, of course: they simply require at some stage multiple function evaluations or search directions or whatever, then do something with them. How these multiple items are acquired in computer execution of the algorithm - one after another in a serial implementation, in parallel on an SMP or a cluster, from the Sears catalog, or by divine inspiration - has absolutely nothing to do with the algorithm, which is a mathematical entity entirely divorced from its low level implementation details. As Stroustrup says, "Independent concepts should be independently represented and should be combined only when needed" (Stroustrup, 1995).

Component design provides a method - in fact, essentially the only method - to insulate optimization algorithm implementations from low-level but pervasive system details

like data distribution. The actual details of a component framework are irrelevant: the essential feature is the integration of various levels of abstraction in various components (usually processes), and the provision of communication between layers of the components *at a common level of abstraction.*

SVL's design facilitates this compartmentalization of communication into well-defined abstraction layers. The "calculus" layer of SVL has no need to interact with any distributed execution details, and is naturally confined to one or more serial processes. The "data management" layer on the other hand is open to various distribution schemes. It is natural to write server application using only the data management classes, and implement "calculus" algorithms such as optimization as clients.

We have carried out a simple version of this client-server design using both HCL and SVL. The HCL-based client-server application used CORBA as its communications layer, and parallelized the server using MPI. This design confined all MPI code to the server, but the structure of HCL required that communications layer artifacts be inserted in virtually all of the concrete classes and that essentially the entire package be installed on both client and server platforms. The design did keep references to low-level communication issues out of the base classes and the optimization algorithms implemented in terms of the base classes (Scott, 2001) - these did not change between serial and client-server/parallel applications.

SVL's separation of "calculus" and "data management" layers permits *all* interaction with component framework and distributed execution software to be confined to `DataContainer` and `FunctionObject` classes, with no inclusion of communications code in the "calculus" layer at all. A `Streamable` type and judicious use of multiple inheritance produces remote execution pairs of `FunctionObject`s, the remote parts of which can be implemented in parallel (and independently of the component framework). Distributed servers rely on parallel `LocalDataContainer` classes, which return the memory segments assigned to each process in the distributed framework via the same `getData()` interface used in the serial implementation. The basic design of this `Remote` extension of SVL is described in (Dajani, 2003; Symes et al., 2004a).

Why `Space::Dim()` is a bad idea

Virtually all proposed vector class interfaces provide a method which returns the dimension or number of components of a vector (such as ESI's `ESI_Vector::getGlobalSize(...)`, (ESI Working Group, 2001)). In our design, this method would naturally belong to the `Space` class; The current HCL release calls it `HCL_VectorSpace::Dim()`. In this section we argue that it has no business being there.

An obvious objection to requiring the dimension to be part of the base class interface is that it's unnecessary. Nothing else depends on it: if anything important did, by implication that feature would break data encapsulation. For example, imagine an out-of-core vector class which instantiates vector objects using filenames. A constructor for such a thing should associate the object with the file, by opening the file, presumeably. However there is no need *a priori* for dimension of the data to be known at construction: that

requires reading the file, and there is no need to do that until something is done with the data. In other words, there is no reason not to defer the file access - however a `Dim()` function would require immediate access to the file, ie. another pass through the data, for no obvious immediate use.

This efficiency objection might seem somewhat frivolous. However consider the possibility of an adaptive grid vector class, which is very likely useful in many PDE-related settings. One could imagine an addition rule for such a class which permits the addition of two grid functions defined on different grids through union of the grids and interpolation. In effect, one is actually working with functions and adjusting the representation as required to carry out vector arithmetic. Operators (like grid refinement) which change the number of gridpoints would also have a natural role. Note what has happened: the number of degrees of freedom characterizing a vector is not constant during execution for such a class. This concept is reminiscent of infinite-precision floating point arithmetic systems, and implementation is entirely feasible. But now the notion of dimension is *meaningless*. In fact, such a class provides a computational realization of infinite dimensional spaces, i.e. spaces which do not have dimensions, because they do not have (finite) bases. At any point in the execution of a program using such classes, the functions involved would all be members of finite-dimensional subspaces, but the subspaces occuring in the execution would vary dynamically, with no *a priori* limit on dimensionality.

Since we strongly object to including functions in abstract interfaces which cannot always be implemented, this example seems to us to present an unanswerable objection to `Dim()`.

AN EXAMPLE: LEAST SQUARES DATA GRIDDING

The gridding problem - defining data on a regular grid which captures information measured at a arbitrary set of points - serves to illustrate how SVL may be used in the solution of a simple but nontrivial scientific computation problem. We formulate this problem as an optimization: find the regular grid data which interpolate (piecewise bilinearly) onto the irregularly spaced sample points, with least mean square error. This least squares formulation is hardly the only, or even the best, approach to the gridding problem - for example, a least absolute sum formulation would be less susceptible to bias from a few badly erroneous samples (Guitton and Claerbout, 2004). Other formulations could be cast into code in a similar way. Least squares has the advantage that the first order conditions for optimality are both necessary and sufficient, and are equivalent to a positive (semi)definite symmetric linear system. We shall solve a version of this problem which is too large for dense linear algebra methods on most contemporary computers (133,000 equations in 40,000 unknowns), so we will use an iterative method instead. The conjugate gradient method is a natural choice; our implementation combines SVL and the *Algorithm* package (Padula, 2004), an independent library which expresses iterative algorithms.

Problem setup - data structures

The first task in any such project is to define the key data structures, as `DataContainer` subclasses. Since computations of the target size can be handled in core on contemporary workstations, we make these `DataContainer`s local. Both types of data (regularly and irregularly gridded) combine geometric information with an array of samples. The data abstraction capability of object orientation makes this combination natural.

For regular grid data, the geometric aspect is abstracted in a `Grid` class. A `Grid` instance specifies the number of axes `naxes` and, for each axis `a` (= 1, 2, 3,...) the number of gridpoints along that axis `na`, the step between points along that axis `da`, and the coordinate of the first point `oa`. The `Grid` interface provides access to these numbers, as well as to the total number of gridpoints, the volume of the grid cell, and several other useful quantities.

A `GridData` object refers to a `Grid` object and *is* an `RnArray` (for definition, see *General Considerations on the Design of Data Container Classes*, above):

```
template<class Scalar> class GridData: public RnArray<Scalar> {
private:
  Grid<Scalar> & g;
  ...
public:
  GridData(Grid<Scalar> & gg): RnArray<Scalar>(gg.getSize()), g(gg) {}
  ...
}
```

A `GridSpace` refers to a `Grid` and builds `GridData` objects dynamically using `operator new` and the constructor just described.

For archival storage of gridded data, we have chosen the file format pioneered by the Stanford Exploration Project (Claerbout, 2003). This format stores the data in two parts: an ASCII file containing the grid geometry in the form of "key=value" pairs, together with a line which specifies the name of another file; this second file is a binary flat file containing the data samples. The `Grid` class has a constructor which takes the ASCII filename and initializes the required data from the file.

To initialize a `Vector` in a `GridSpace` from disk data in this format, one instantiates a `Grid` from disk (usually using the same file name), a `GridSpace` from the `Grid`, a `Vector` in the `GridSpace`, a `GridLoad FunctionObject` by handing the ASCII file name to the constructor. Then one evaluates this `FunctionObject` on the target `Vector` object:

```
string fname = "stuff.sep";
Grid<float> gr(fname);
GridSpace<float> gsp(gr);
Vector<float> v(gsp);
```

```
GridLoad<float> gl(fname);
v.eval(gl);
```

`GridSave` functions the same way to archive gridded data to a disk file in SEP format.

Irregular grids are simply lists of coordinate pairs (in this example, we restrict ourselves to the 2D case). A `UMesh` object stores such a list, together with its length, and defines the geometry of unstructured mesh data just as `Grid` defines the geometry of regularly gridded data. `UMeshData` is the corresponding `DataContainer` subclass, also derived from `RnArray`, and `UMeshSpace` the space class. A simple ASCII file format simply lists the coordinate pairs, or triples if data is included, one to a line. `UMeshReader` and `UMeshWriter` function objects read data from, respectively write data to, files in this format. The `UMesh` class is provided with a constructor reading coordinates from a file, analogously to `Grid`.

Problem setup - functions

Having defined the data structures and corresponding vector spaces, we turn to the defining functions of the problem - in this case there is just one, the bilinear interpolation operator. Since piecewise (bi)linear interpolation is a linear map, it is natural to realize this operator as a subclass `Interp2DOp` of `LinearOp<float>`. We implemented the `apply` methods via calls to Fortran subroutines. This is a typical SVL design: the numeric kernels are coded in procedural style, and address large blocks of intrinsic type data. We have found that in such implementations, as soon as problem size becomes sufficiently large, the overhead of virtual function calls and other object-oriented apparatus becomes negligible: codes structured as we have indicated here are as fast as purely procedural implementations, and are much easier to maintain.

To make interpolation efficient, the `Interp2DOp` class stores the interpolation coefficients and upper - left - hand - corner indices for each interpolation output point as private data of a `Interp2DObjectData` object. This object merely requires for its construction the input `Grid` and output `UMesh`, i.e. the geometric description of the interpolation problem. The `Interp2DOp` constructor takes `GridSpace` and `UMeshSpace` references specifying domain and range, and constructs its `Interp2DObjectData` data by member initialization. Mostly to illustrate the use of `FunctionObject`s in implementing function types, we chose to build the `apply` methods out of `FunctionObject`s which compute that interpolation operation and its adjoint. The critical block of code is simply

```
void Interp2DOp::apply(const Vector<float> & x,Vector<float> & y) const {
  try { y.eval(fwd,x); }
  catch (SVLException & e) { ...  }
}
```

Here `fwd` is a `Interp2DFwd` instance (a data member of `Interp2DOp`), a binary `FunctionObject` whose `operator()` method is implemented as follows:

```
void Interp2DFwd::operator()(LocalDataContainer<float> & output,
     LocalDataContainer<float> & input) {  try {
    Grid<float> & g = od.getGrid();
    UMesh & um = od.getUMesh();
    if (input.getSize() != g.getSize()) {
       /* some sanity checking... */
    }
    // initialize parameters needed by Fortran
    int ier=0; int idbg=0; int ipdmp=7;
    int n1 = g.get_n(0); int n2 = g.get_n(1);
    int n = um.getSize(); float vol = g.getVol();
    float rvol = 1.0; float one = 1.0;
    if (ProtectedDivision<float>(one,vol,rvol)) {
      SVLException e;
      e<<"Error: Interp2DFwd::operator()\n";
      e<<"zerodivide by cell volume\n";
      throw e;
    }
    // call Fortran interpolation routine
    interp2df_(n1,n2, input.getData(),
       n, output.getData(),
       od.getInd(), od.getCoeff(),
       rvol,idbg,ipdmp,ier);
    if (ier) { /* throw exception */  }
  }
  catch (SVLException & e) { ... }
}
```

This code fragment illustrates

- the use of `ProtectedDivision`, a utility supplied with SVL which sanity - checks floating point division. This function is quite expensive, so it should be kept out of inner loops. However it allows the exception handler rather than the run-time system to trap zerodivides.
- the interaction with procedural code - particularly in the case of Fortran 77, all arguments are implicitly mutable and must be of intrinsic type. Types like `Interp-2DObjectData`, functioning as containers of data to be passed to such an interface, must expose their data, so one is thrown back on traditional procedural programming practice as the only defense against memory management errors. It is for this reason that we have left memory management flexible in the "data management" layer of SVL, which will often be called upon to interact with procedural code as in this example.

The adjoint application (`applyAdj`) is coded similarly.

Solution Algorithms - AlgPack

With the data structures encapsulated in vector spaces, and the appropriate functions defined, the least squares formulation of the gridding problem is completely specified. To solve it, we use the conjugate gradient method to solve the normal equations ("CGNE"). In a related project (Padula, 2004), we have proposed a class library for encapsulating iterative algorithms such as CGNE. This AlgPack library is independent of SVL, but uses compatible design principles and is easily combined with SVL to implement matrix-free algorithms such as CG. Matrix-dependent algorithms (such as active set methods for constrained optimization) are also available, but depend on specialized interfaces and `FunctionObject`s not provided in the SVL base classes.

In this subsection we briefly describe AlgPack in general and its implementation of CGNE in particular. We emphasize that, in contrast to the work described in the last two subsections, the code described in this section is *already implemented*, and is *not* part of the problem setup. The CGNE algorithm used here is provided with AlgPack, along with a number of other useful optimization and linear algebra algorithms. AlgPack is intended to provide a framework for implementation of many algorithm classes beyond those included with the package, but we regard authoring of these algorithms and applying them to scientific problems as distinct activities. Indeed, a core purpose of the SVL project (and of similar OO numerics projects) is to make this division of labor possible!

Algorithms are simply Turing machines that terminate. Thus AlgPack provides an abstract base class for algorithms, `Alg`, which has a single nontrivial virtual method, namely `bool Alg::run()` (algorithms run successfully or not, hence the boolean return). In most cases access to the state of an algorithm is also essential. `StateAlg` is a subclass of `Alg` abstracting algorithms which can externalize their state. The state type is a template parameter of `StateAlg`.

One of the main operational principles of AlgPack is that single steps of iterative algorithms are themselves algorithms, and more complex algorithms are built up out of less complex algorithms and stopping criteria using a few simple patterns. The AlgPack implementation of CGNE is a straightforward example. A single step of CGNE is encapsulated in `CGNEStep`, which subclasses `StateAlg`. The usual initializations of the CG loop occur in the constructor of `CGNEStep`. Stripped of comments and sanity tests and rewritten as concisely as possible, it illustrates well the stack-oriented programming style which SVL's design promotes:

```
template <class Scalar>
class CGNEStep : public StateAlg<Vector<Scalar> >{
private:
  // external inputs
  LinearOp<Scalar> & A;
  Vector<Scalar> & x, b;
  // work space
  Vector<Scalar> r, s, z, p;
  Scalar res0, res,sts, nres0, nres;
```

```
public:
  CGNEStep(LinearOp<Scalar> & AA,
           Vector<Scalar> & xx,
           Vector<Scalar> & bb)
    : A(AA), x(xx), b(bb), r(A.getRange()), s(A.getDomain()),
      z(A.getRange()), p(A.getDomain())  {
      A.apply(x,z);
      r.linComb(1.0,b,-1.0,z);
      res0=sqrt(r.inner(r));
      res=res0;
      A.applyAdj(r,s);
      p.copy(s);
      sts = s.inner(s);
      nres0 = sqrt(sts);
      nres  = nres0;
 }
```

Since conjugate gradient iteration involves only operations intrinsic to linear algebra in Hilbert space, the entire `CGNEStep::run` method uses only methods of the abstract SVL base classes:

```
  bool run() { try {
      A.apply(p,z);
      Scalar ztz = z.inner(z);
      Scalar alpha;
      if (ProtectedDivision<Scalar>(sts,ztz,alpha)) { ... }
      x.linComb(1.0,x,alpha,p);
      r.linComb(1.0,r,-alpha,z);
      A.applyAdj(r,s);
      Scalar sts1 = s.inner(s);
      Scalar beta;
      if (ProtectedDivision<Scalar>(sts1,sts,beta)) {...}
      p.linComb(beta,p,1.0,s);
      sts=sts1;
      res=sqrt(r.inner(r));
      nres=sqrt(sts);
    }
    catch (SVLException & e) {...}
  }
```

Because the step has been implemented as an algorithm *object*, it has (or can have) internal state. Therefore the `run` method can be called repeatedly in a loop to implement the CGNE iteration. AlgPack provides a suite of tools for combining algorithmic steps with termination criteria, instances of the `Terminator` base class, to produce complete

algorithms. The `LoopAlg` type is appropriate for a simple loop like CG iteration. A `LoopAlg` refers to an `Alg` and a `Terminator`; it runs the `Alg` until the `Terminator` returns true.

A standard CG implementation combines a `CGNEStep` instance with a `CountingThresholdIterationTable` terminator, a logical combination of simpler `Terminator`s which limits the number of iterations and also exits if a function of the iteration state falls below a threshold, also prints a summary table of the history of this function. `Alg` subtypes provide public reference access to various functions of state, so it is merely necessary to pass a method invocation as an argument to the `Terminator` constructor. In particular a method of `CGNEStep` returns a reference to the residual norm. Passing this reference to the `CountingThresholdIterationTable` constructor causes the terminator to return true when the residual norm falls below a threshold (also a constructor argument).

AlgPack actually packages the choices we've just outline as the `CGNE` class. However to display more clearly the way in which AlgPack synthesizes complex algorithms from simpler pieces, we've chosen to use `CGNEStep` and `CountingThresholdIterationTable` directly.

Drivers and Example

Our solution of the gridding problem proceeds as follows:

- construct the irregular mesh of data samples by reading data from a file;
- deduce from this data a regular grid of suitable extent;
- initialize a data vector from the data samples;
- initialize a solution vector from the regular grid information;
- construct an interpolation operator (adjoint pair) interpolating regular grid onto irregular mesh data;
- iteratively update the solution vector to produce a best fit interpolation to the irregular mesh data, using the CGNE algorithm.

The driver program which one might write for this application simply restates these steps in code, using the `UMesh`, `Grid`, `Interp2DOp` and related classes explained above, and the `AlgPack` implementation of `CGNE`. The essential lines in the driver procedure are

```
UMesh um(datafile);
UMeshSpace rng(um);
UMeshReader umr(datafile);
Vector<float> b(rng);
b.eval(umr);
// ...
```

```
// some code to compute a grid box containing all sample points,
// using the UMesh access functions, then a Grid of 201 points on
// a side filling this box. Call this Grid gr.
// ...
GridSpace<float> dom(gr);
Vector<float> x(dom);
x.zero();
Interp2DOp op(dom,rng);
CGNEStep<float> cgstep(op,x,b);
int maxit=20;
float tol = 1.e-2;
string s = "Norm of Resid";
CountingThresholdIterationTable<float>
  term(maxit, cgstep.getNormResidual(),tol, s);
LoopAlg cg(cgstep,term);
cg.run();
```

We have used this driver to grid a data set of depth soundings from the Sea of Galilee (Guitton and Claerbout, 2004). This bathymetric data set consists of roughtly 133,000 data points (x,y,z pairs), which are to be mapped onto a 201×201 grid. There are thus 133,000 equations in 40,000 unknowns (roughly) to be solved. This gridding problem does not appear to be particularly ill-posed: 40 conjugate gradient iterations reduced the residual (unweighted l^2) norm from 2.3×10^6 to less than 1×10^3. Execution time on a Mac Powerbook G4 1 GHz machine was 24 seconds, approximately 21 seconds of which were devoted to reading the ASCII data file twice.

DISCUSSION

We have described in great detail the design approach and decisions that have shaped SVL. It remains to describe the relationship of SVL to other OO numerics projects, various extensions and applications of SVL currently underway or completed, and the role which the authors foresee for this approach to scientific computation.

Relation to TSFCore and other packages

The goals of the SVL project are similar to those that have shaped a number of other OO numerics projects. To Illuminate the consequences of the SVL design, comparison with other similar libraries is useful. As mentioned in the introduction, SVL is the direct descendent of HCL, and many similarities and differences in design principles between these two libraries, and consequences of these, have already be identified. The closest contemporary project in goals and principles to SVL is Sandia National Laboratory's TSFCore - unsurprisingly, as TSFCore borrowed features from HCL also inherited by SVL, and SVL borrowed key ideas from TSFCore and its predecessors.

On first inspection, obvious differences emerge in programming style and aesthetics. Both packages make extensive use of handle classes to hide memory management details,

but TSFCore uses a general purpose reference counted pointer class, whereas SVL base classes are intrusive handles (or abstract "prehandles"). SVL often (though not always) follows the pattern of "simultaneous instantiation and initialization" whereas TSFCore classes systematically separate these steps (see (Meyers, 1996) for a good discussion of these options). The *Multivector* concept is central to TSFCore (and to a number of other packages in the same family), whereas SVL offers similar but more general functional behaviour as a special case of its `ProductLocalDataContainer` interface. The `RTOp` classes play a role in TSFCore similar to that of `FunctionObject`s in SVL, but are restricted for efficiency reasons to diagonal operators; these efficiency considerations are implicitly regarded as implementation details of `SVL::FunctionObject` subtypes, and are not legislated.

Two fundamental differences between the packages actually organize this welter of apparently stylistic variance. The first critical difference is in the level of abstraction attained: `SVL::Vector` for example has no explicit analog in TSFCore. The `TSFCore::Vector` type is actually a functional near-isomorph of `SVL::DataContainer`, with some functionality of `SVL::LocalDataContainer` mixed in. Similarly, `TSFCore::LinearOp` is most closely mimicked by a special subtype of `SVL::FunctionObject`, rather than by `SVL::LinearOp`. Seen through this lens, most of the contrasts mentioned in the preceding paragraph reveal themselves as essentially syntactic. Because of the close correspondence between TSFCore and the part of SVL below the calculus layer, it is possible to interoperate the two packages quite efficiently, by means of fairly straightforward adaptor classes, as envisioned by the authors of TSFCore (Bartlett et al., 2003). This adaptation, together with a nontrivial application interoperating the two packages and others, is described in detail in (Padula, 2003).

The second critical difference lies in the problem domains around which the two classes were designed: TSFCore focusses (at least in its current incarnation) exclusively on expression of linear algebra algorithms, whereas SVL attempts to provide a platform for calculus in Hilbert Space (as did HCL). Thus TSFCore offers no analogues of the nonlinear vector function types `SVL::Functional, SVL::Operator`, and has no need to realize the Evaluation concept. Optimization packages directly compatible with TSFCore (for example MOOCHO, (Bartlett, 2003)) realize nonlinear functions in various ways. Amongst the current Trilinos packages (Heroux et al., 2003), NOX (Kolda and Pawlowski, 2003) offers the most complete suite of abstract types for expression of nonlinear optimization algorithms, including a realization of the Evaluation concept. However at present only SVL actually integrates both linear and nonlinear functionality in one package.

Virtually all of the other OO numerics libraries mentioned in the introduction correspond conceptually to the `FunctionObject` - `LocalDataContainer` layer of SVL.

Extensions, Applications of SVL

SVL plays a key role in several follow-on projects, some already mentioned:

- a simple SVL-compatible component framework, along with corresponding "remote"

variants of SVL classes, enabling distributed execution of SVL applications in a heterogeneous network (Dajani, 2003; Symes et al., 2004a);

- a time-stepping simulation library TSOpt, incorporating sensitivity and adjoint sensitivity computations (the latter with optimal checkpointing) in a system of SVL-based classes (Symes et al., 2004b);
- an independent collection of interfaces for iterative algorithms, providing a uniform implementation framework for both SVL-based Newton-related optimization algorithms and for solution of problem classes beyond the scope of SVL, such as mixed integer-continuous programming problems (Padula, 2004);
- an abstract trust region interior point SQP implementation, accommodating a variety of linear algebra approaches (dense, sparse, iterative);
- a collection of adaptor classes used to interoperate SVL and applications (in particular TSOpt) with Trilinos packages (TSFCore, MOOCHO) (Padula, 2003), and to delineate the features of OO numerics packages which enhance or inhibit interoperation.

All of these packages are under development, having achieved initial demonstration–of–concept implementations. An Epetra (Heroux, 2002) based parallel server class is under construction, as are several other applications in control and parameter identification. A longer term project just beginning is extension of the SVL conceptual base to the discrete domain, to encompass integer and categorical variables and problem formulations in discrete variable modeling languages such as AMPL (Fourer et al., 2002).

The role of OO Numerics

The promise of OO numerics lies in the introduction of abstract types which realize the behavior of numerical mathematics concepts. These abstract types hide details of data storage and manipulation, and so permit the abstract expression of algorithms, which may be (re)used across the full variety of applications expressible via the abstract types. The critical step in building a library of such abstract types is the identification of the key concepts and their behaviors, disentangled from representation details. Since most of these concepts have familiar procedural implementations which thoroughly entangle the mathematics with low-level computational details, this step in package design is often difficult. Once the key abstractions are identified, interfaces must be constructed which realize their structure while permitting efficient and flexible implementation.

All of the packages mentioned in the preceding pages carry out this program to some extent. SVL differs from all of the others in the level of abstraction which it achieves. The central contention of our project, as stated in the introduction to this paper, is that the abstract "calculus" layer, with its simplicity and naturalness as a vehicle to express abstract numerical algorithms, permits a greater degree of reusability and maintainability in implementations of such algorithms than can possibly be achieved using types which

are more entangled in data representation details. In conjunction with the "data management" layer which provides uniform interfaces for interaction with data, the "calculus" layer permits coupling of abstractly formulated linear algebra and optimization algorithms with simulators of essentially arbitrary complexity, with near optimal performance on serial and parallel platforms.

Acknowledgements

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

REFERENCES

Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Curfman McInnes, L., and Smith, B. F. (2001). PETSc home page. `www-fp.mcs.anl.gov/petsc`.

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

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

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

Barton, J. G. and Nackman, L. R. (1994). *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley, New York.

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

Claerbout, J. (2003). SEPlib. Technical report, Stanford Exploration Project, Stanford University, Stanford, California, USA.

Dajani, H. (2003). Client-Server Component Architecture for Scientific Computing. Technical Report 03-07, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, USA.

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

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

ESI Working Group (2001). Equation Solver Interface home page. `z.ca.sandia.gov/esi/`.

Fourer, R., Gay, D. M., and Kernighan, B. W. (2002). *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, New York.

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.

Guitton, A. and Claerbout, J. (2004). Interpolation of bathymetry data from the Sea of Galilee: A noise attenuation problem. *Geophysics*, 69:608–616.

Heroux, M. A. (2002). Epetra:Concrete C++ linear algebra classes for parallel linear algebra. Technical report, Sandia National Laboratories, Albuquerque, NM.

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

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

Hough, P., Kolda, T., and Torczon, V. (2001). Asynchronous parallel pattern search for nonlinear optimization. *SIAM Journal on Scientific Computing*, 23:134–156.

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

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

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

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

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

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

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

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

Padula, A. D. (2003). Interoperation of SVL and TSFCore. Technical Report 03-XX, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, USA.

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

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

Stroustrup, B. (1995). *The C++ Programming Language*. Addison-Wesley, Reading, MA, third edition.

Symes, W. W., Dajani, H., and Padula, A. D. (2004a). Remote: a simple component architecture for scientific computing. Technical Report 04-XX, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, USA.

Symes, W. W., Dussaud, E., Dajani, H., and Padula, A. D. (2004b). A time-stepping library for simulation-driven optimization. Technical Report 04-XX, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, USA.

Symes, W. W. and Padula, A. D. (2004). SVL user's guide. Technical Report 04-XX, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, USA.

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

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

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