SVL: The Standard Vector Library

Mark S. Gockenbach

Department of Mathematical Sciences
Michigan Technological University
Acknowledgements: Much of the work described in this talk has been done by Bill Symes and his students at Rice.

The modifications of HCL leading to SVL originated in discussion with Shannon Scott, an MS student at Rice, and ultimately were prompted by Roscoe Bartlett’s RTOp proposal.
Outline:

- The Hilbert Class Library (HCL)
  - Design considerations
  - Innovations unique to HCL
  - Drawbacks
- The search for a standard vector concept
- The Standard Vector Library (SVL)
The Hilbert Class Library (HCL) is a library of C++ classes defining the mathematical objects necessary for doing calculus, and hence optimization, in Hilbert space:

- vector spaces
- vectors
- functionals
- operators (linear and nonlinear)
HCL was designed to make it possible to code optimization algorithms in such a way that they could be applied to a wide variety of (large-scale) applications.

The goal was to define classes with a high degree of abstraction, so that concepts that are similar mathematically yet implemented quite differently can be handled uniformly.
An optimization problem is defined by:

- the objective function;

- constraint operator(s) (if any);

- the vector representing the problem variables.
The primary difficulties in the interface between optimization code and application code are:

- There is not a single vector data structure that is natural or even workable for all application problems.

- Function and operators frequently require more than just a subroutine to implement; parameters, grid information, etc. are often needed.
Vectors

(Basic proof of concept) HCL_Vector encompasses

- simple in-core (Fortran-style) arrays;
- disk data files;
- data sets distributed over a network.

Virtual base class (HCL_Vector) defines properties of vector (straight from linear algebra book—almost!), and concrete derived classes implement specific data structures mentioned above with code to manipulate them.

Optimization algorithms (e.g. LBFGS) manipulate the base class, and so apply without modification to all types of vectors.
Functionals and operators

HCL_Functional: scalar-valued functions

HCL_LinearOp, HCL_Op: vector-valued operators

The C++ class concept allows the data (e.g. parameters) and algorithms that define a functional or operator to be encapsulated in a single object that can be passed safely to an optimization algorithm.
HCL_Vector, HCL_Functional, HCL_LinearOp, and HCL_Op solve the interface problems typically encountered when trying to link application code to optimization code.

Specifically, the user can choose the data structures for vectors manipulated by the optimization code and can safely and conveniently specify the functionals and operators defining the optimization problem. (No more common blocks, parameter arrays, etc.)
HCL innovations:

- **Vector spaces as objects**—for error checking input and output vectors, and for allocating new vectors (workspace) without knowing the data structures.

- **Evaluation objects**—represent 
  \[(x, f(x), \nabla f(x)) \text{ or } (x, f(x), \nabla f(x), \nabla^2 f(x))\].
  Provide place to save intermediate results shared in evaluation of \(f(x), \nabla f(x), \nabla^2 f(x)\).
• Tool classes: product vectors \((x, y)\), block operators \[
\begin{bmatrix}
A & B \\
C & D
\end{bmatrix}
\], composition operators \(F(G(x))\), etc.

• Algorithm classes—abstract implementation of popular algorithms such as LBFGS, \(k\)-step Arnoldi, etc.
Unfortunately, HCL contains a significant limitation, and it’s in the most important object: HCL_Vector.

Although the mathematical definition of a vector mentions only two (coordinate-free) operations, addition and scalar multiplication, constrained optimization algorithms need to perform various “array” operations defined directly on the components of a vector.
Some examples compiled by Bartlett:

\[
\begin{align*}
    y_i & \leftarrow y_i + \alpha, \quad i = 1, 2, \ldots, n \\
    y_i & \leftarrow \frac{y_i}{x_i}, \quad i = 1, 2, \ldots, n \\
    y_i & \leftarrow y_i + \alpha x_i z_i, \quad i = 1, 2, \ldots, n \\
    \alpha & \leftarrow \max\{\alpha |x + \alpha d \geq \beta\} \\
    \gamma & \leftarrow \max\left\{\frac{|d_i|}{1 + |x_i|} : i = 1, 2, \ldots, n\right\}
\end{align*}
\]

HCL\_Vector grew to contain 35 methods implementing array operations (certainly a major deviation from the design philosophy!) in a losing effort to provide for such needs.
The search for a standard vector concept

Clearly, no implementor of a vector class can anticipate all of the array operations that current and future algorithms will need.

Providing a limited number of “standard” array operations and expecting algorithm implementors to create any needed operation as a composition of the standard operations leads to obvious inefficiencies (multiple passes through data).
What is needed is a mechanism for allowing users of abstract vector classes (e.g. algorithm writers) to specify the needed array operations to the vector class for it to apply.

This mechanism should not restrict the data structures (in-core vs. out-of-core, serial vs. parallel) that can be used to implement the vector concept.
Roscoe Bartlett proposed such a mechanism: \( \text{RTOP} \), an interface for reduction and transformation operators using a method-forwarding mechanism.

“We” (mostly Shannon Scott and Bill Symes) have adapted his proposal to suit some of our predilections:

- Vector operations (addition, scalar multiplication, inner product) should be presented as class methods (required for all vectors).

- Array operations, which are not part of the definition of vectors, can be implemented by users of a vector class and passed to it via a function object.

- Function objects need not be restricted to componentwise (diagonal) operations.
SVL

The Standard Vector Library is our improvement (we hope!) of HCL that is based on what we propose as a standard for vector class interfaces.

SVL has a new name because we decided to make a number of other less important but nontrivial design changes that make backward-compatibility with HCL impossible:

- Classes are templated on scalar type (e.g. float vs. double).

- STL containers are used extensively in place of ad-hoc containers.

- C++ exception handling is used.

- etc.
Vectors and (vector) spaces in SVL

Vector is a concrete class in SVL. Each vector points to a Space and has a DataContainer object.

Space is an abstract (pure virtual) base class defined by the axioms of an inner product space:

• A vector space is a set of vectors. Since computer programming is not the same as mathematics, this property is realized in the following way: a Space can allocate a DataContainer of the correct type and hence create new vectors from the space.
• A Space can form a linear combination of two vectors from the space.

• A Space can compute the inner product of two vectors from the space.

• (Not axiomatic but useful) A Space can compare itself to another Space and determine if the two are equal. Useful for abstract error checking.
Vector implements the linear combination and inner product operations by calling the corresponding methods from the space class.

Vector also has an (overloaded) eval method that takes a function object and passes it onto the data container.

Vector::eval is our mechanism for allowing arbitrary array operations. (Much cleaner than providing dozens of array operations as class methods!)
Function objects can be unary, binary, ternary, etc. and of the reduction or transformation type.

A function object has one primary method: operator()

That is, a function object $f$ can be applied as $f(x)$.

The argument $x$ is a LocalDataContainer.
LocalDataContainer provides direct access to data samples through the getData() method (returns a pointer to a primitive (C-style) array of values).

Floating point ops are actually done in local data containers.

Data containers form a layer between local data containers and vectors.
Data containers can represent:

- a product of local data containers;

- out-of-core data;

- distributed data.

The eval method provides a uniform way for optimization algorithms to perform array operations on these various data structures.
Typical scenario: Algorithm writer needs an array operation, e.g.

\[ \sum_{i=1}^{N} \log(x_i) \]

He or she must write a small class, e.g. LogSum in this case derived from UnaryFunctionObjectRedn.

Actual computation is done in the operator() method (the rest of the class is just boilerplate, and there isn’t much of it).

Algorithm then applies the array operation as follows:

LogSum<double> logsum; // Create function object
x.eval( logsum );       // Apply function object
val=logsum.getResult(); // Extract result
Status of SVL

Basic design of SVL seems to be valid.

Base classes need some refinement.

Parallel vector classes and constrained optimization algorithms needed to prove the concept more fully.