The Scythe Statistical Library: An Open Source C++ Library for Statistical Computation

Kevin M. Quinn, Berkeley Law
Daniel Pemstein, University of Illinois at Urbana Champaign
Andrew D. Martin, Washington University in St. Louis

Available at: https://works.bepress.com/kevin_quinn/43/
Abstract

The Scythe Statistical Library (Pemstein, Quinn, and Martin 2007) is an open source C++ library for statistical computation. It includes a suite of matrix manipulation functions, a suite of pseudo-random number generators, and a suite of numerical optimization routines. Programs written using Scythe are generally much faster than those written in commonly used interpreted languages, such as R and MATLAB; and can be compiled on any system with the GNU GCC compiler (and perhaps with other C++ compilers). One of the primary design goals of the Scythe developers has been ease of use for non-expert C++ programmers. Ease of use is provided through three primary mechanisms: (1) operator and function over-loading, (2) numerous pre-fabricated utility functions, and (3) clear documentation and example programs. Additionally, Scythe is quite flexible and entirely extensible because the source code is available to all users under the GNU General Public License.

Keywords: matrix operations, pseudo-random number generation, numerical optimization, C++. 

1. Introduction

This paper introduces the Scythe Statistical Library—Scythe for short. Scythe is a open source C++ library for statistical computation. It includes a suite of matrix manipulation functions, a suite of pseudo-random number generators, and a suite of numerical optimization routines. What sets Scythe apart from most other C++ libraries for statistical computing is its intuitive interface and its general ease of use. Writing programs in C++ using Scythe is only slightly more complicated than writing the equivalent program in R or MATLAB. This
is accomplished through (1) operator and function over-loading, (2) numerous pre-fabricated utility functions, and (3) clear documentation and example programs.

We have made ease of use a primary design goal for Scythe for three reasons. First, a clean, relatively intuitive user interface makes it easy to move from ideas sketched in pseudo-code or prototyped in a language such as R to a full implementation in C++ using Scythe. Relatedly, experience has shown us that, because of the Scythe interface, it is not difficult for a reasonably proficient R programmer to pick up Scythe and to start using it to write non-trivial C++ programs even with only a cursory knowledge of C or C++. Finally, because code written using Scythe is typically quite transparent and intuitive, maintaining code that makes extensive use of Scythe is relatively easy.

Ease of use does not come without some costs. While programs written using Scythe will typically be quite fast — oftentimes an order of magnitude or more faster than the equivalent program written in R — they will not typically be as fast as highly optimized code written in C or FORTRAN. However, once development and maintenance time are accounted for, Scythe compares favorably to these other options for most users. Further, as we note below, we are in the process of closing many of these speed gaps by wrapping high quality BLAS and LAPACK routines inside Scythe.

We anticipate that Scythe will be of most use to those users who routinely use R or MATLAB for computationally intensive tasks with runtimes of 10 minutes or more. These users stand to gain noticeable improvements in performance with relatively minor up front costs of learning to use Scythe.

The rest of this article is organized as follows. In Section 2 we review the three primary components of the Scythe library— the Scythe Matrix class, Scythe’s pseudo-random number generators, and Scythe’s routines for numerical optimization. We then provide an extended example of how to use Scythe to perform a parametric bootstrap. This example makes use of pieces of all three primary components of the Scythe library mentioned above. This section also compares the Scythe implementation to two implementations of the same bootstrap procedure in R. In Section 4 we show how C++ code using Scythe can be called from within R. Section 5 concludes.

2. An Overview of the Scythe Library

2.1. The Scythe Matrix Class

The Matrix class is the fundamental component of the Scythe library. Virtually every function in the library operates on or returns Matrix objects. This custom data structure allows us to maximize library efficiency while hiding most of the underlying details from the user. We designed the Matrix class primarily for ease of use, especially for those who are more familiar with mathematics than software development. At the same time, we wished to provide a great deal of flexibility in implementation. Finally, we wanted to minimize the risk of user error when programming with Scythe, while utilizing sophisticated data management techniques under the hood.

Matrix objects allow us to divorce the tasks of data manipulation and matrix arithmetic from the domain-specific capabilities the library provides. This makes coding with Scythe easier for both the Scythe development team and our user base. Utilizing Matrix objects requires users
to familiarize themselves with the interface these objects provide, but this initial learning cost is well worth the long-term gains. If we had used a language primitive, such as two-dimensional arrays, to handle data in Scythe, every update to the library’s internals would break existing code; by using objects we can commit to a particular interface for data manipulation.

The Matrix class provides an interface similar to standard mathematical notation. The class offers a number of unary and binary operators for manipulating matrices. Operators provide such functionality as addition, multiplication, element-by-element multiplication, and access to specific elements within a matrix. One can test two matrices for equality or use provided member functions to test the size, shape, or symmetry of a given matrix. The class also sports a number of facilities for saving, loading, and printing matrices. Related portions of the library allow the user to perform functions from linear algebra, such as transposition, inversion, and decomposition. In addition, the Matrix class is compliant with the Standard Template Library (STL) (Silicon Graphics, Inc 2006) and provides a variety of iterators and accompanying factory methods that make it possible to use the STL’s generic algorithms when working with Matrix objects. Scythe’s variable debug levels allow users to control the degree of error checking done by Matrix objects. While developing an application, users can take advantage of extensive error trapping—including range checking of Matrix element accessor and iterators—to assist in the debugging process. But once the application reaches production quality, it can be compiled with virtually all error checking disabled, maximizing performance. Finally, while Scythe provides C++ definitions for all of its routines, it optionally makes use of the highly optimized LAPACK and BLAS linear algebra packages on systems that provide them. The use of these packages can significantly improve program speed and does not alter the library’s external user interface.

Matrix Templates

We employ C++ templates to make the Matrix class as flexible as possible. Matrices are templated on type, order, and style. In principle, Matrix objects can contain elements of any type, including user-defined types. For the most part, users will wish to fill their matrices with double precision floating point numbers, but matrices of integers, boolean values, complex numbers, and even user-defined classes and structs are all possible and potentially useful.

Matrices may be maintained in either column-major or row-major order. In general, the choice of matrix order is a matter of user preference, but Scythe adopts a bias for column-major matrices when necessary. Most library routines exhibit identical performance across both possible orderings but, when a compromise must be made, we always make it in favor of column-major matrices. This policy is most evident when considering LAPACK/BLAS support in Scythe: currently, the library only takes advantage of LAPACK/BLAS functionality when working with column-major matrices. In addition, although it may sometimes prove useful to work with matrices of both orders in a single program, we discourage this practice in general. While they support cross-order operations, Scythe routines are not generally

---

1 Users can set the amount of error checking done by Scythe routines using the pre-processor flag SCYTHE_DEBUG_LEVEL. See the entry on error.h in Scythe’s Application Programmers’ Interface for details (Pemstein et al. 2007).

2 It is not possible to use all of the matrix operations in Scythe on matrices of all types. For example, if one attempts to use the Matrix class’s addition operator on a Matrix of a user-defined type for which no addition operator exists, the compiler will issue an error. Nonetheless, the basic book-keeping functions of the Matrix class should work with most types.

Scythe matrices use a “data-view” model of data management (Rogue Wave Software 1999; Veldhuizen 2001). This means that it is possible for two or more distinct Matrix objects to refer to—or view—the same underlying block of data. For example, Figure 1 displays a set of four matrices, all of which view some portion of the same block of data. The vectors $a$ and $b$, and the matrix $C$ all view some portion of $X$. If one were to modify the first element in $b$, the change would be reflected in both $a$ and $X$. Notice that all the views in this figure represent sub-matrices of some Matrix, although, in the case of $X$, this submatrix is $X$ itself. All views in Scythe follow this convention; it is not possible, for example, to create a view of the diagonal of $X$.\footnote{Arbitrarily shaped views may appear in forthcoming library releases.}

In virtually all respects, Scythe treats $a$, $b$, $C$, and $X$ identically. All four objects are full-fledged members of the Matrix class and have access to all of the operations that Scythe provides to manipulate Matrix objects. This approach provides great flexibility when working with matrices but it also provides some additional advantages. Most notably, because views represent what are essentially references to underlying blocks of data, we can copy a Matrix view without explicitly duplicating its data, with often substantial benefits to program efficiency.

Views imbue the Matrix class with great flexibility, but bring with them a number of complications. First of all, as we discuss below, the semantics of view copy construction and assignment are somewhat complex.\footnote{Copy construction and assignment are fundamental capabilities of C++ classes. If you are not familiar with these constructs, you may find a standard C++ reference, like Stroustrup (1997), helpful.} Furthermore, the flexibility of views sometimes comes at the cost of efficiency. For both these reasons, we provide users with two styles of Matrix object, concrete matrices and views. The style of a Matrix object describes the policy that governs copy construction of and assignment to the object. Concrete matrices use deep copies for these tasks; when you copy into a concrete Matrix—whether through copy construction, an invocation of the assignment operator, or though the Matrix class’s copy() method—the Matrix allocates a fresh block of data and manually copies the elements of the other object into its own data block. On the other hand, when one copy constructs a view, no copying takes place. At the end of the operation, the newly constructed view simply references the...
Matrix or submatrix upon which it was constructed. View behavior for assignment is also different from that for concrete matrices. While a concrete Matrix object will allocate a new data block that duplicates that of the Matrix on the right hand side of the assignment operator, a view will simply copy the elements in the right-hand-side object into its currently viewed data block, overwriting the existing values.

There are fundamental trade-offs between concrete matrices and views. Some are straightforward; the choice between the copy construction and assignment semantics of concretes and views is often just a matter of personal choice or the problem at hand. But some trade-offs between the two Matrix types are more subtle, and more fundamental. The data array encapsulated by a concrete Matrix is always stored contiguously in memory. A view, on the other hand, might reference only some sub-portion of another Matrix object’s data, which is not guaranteed to reside in memory in contiguous order. Therefore, iterating over a view or accessing a particular element of a view involves more overhead computation than it does for a concrete. On the other hand, the semantics of concrete matrices require that their entire data block be explicitly copied whenever the copy constructor is invoked. Therefore, it is generally much less time-consuming to copy a view than it is to copy a concrete Matrix.

Users should prefer concrete matrices for the bulk of their computation. The reduced efficiency of view iteration and element access has important consequences for program speed; virtually all operations on matrices will run faster on a concrete than they will on a view. In fact, many users will never explicitly construct view matrices. Nonetheless, views perform many important tasks within Scythe—for example, submatrix assignment, which we will discuss in more detail later, implicitly constructs views—and understanding how views fit into the library can greatly improve a user’s ability to write clear and efficient Scythe programs.

Matrix Construction

Perhaps the best way to explain the Matrix class, and the data-view model, is through example. One of the most basic Matrix constructor forms looks like:

```cpp
Matrix<double, Col, Concrete> M(4, 5, true, 0);
```

and constructs a $4 \times 5$ column-major concrete Matrix, filled with double precision floating point values, all initialized to zero. The first two arguments to the constructor provide matrix dimensions, the third argument indicates that the constructor should initialize the Matrix, and the fourth argument provides the initialization value. The later two arguments default

---

5For example, Matrix C in Figure 1 references data that are not stored contiguously in memory because there is a jump between the element with value “8” and the element with value “10” in the data block. Element “10” is two memory places away from “8” if X is stored in column-major order (regardless of C’s order type) and three memory places away if X is stored in row-major order. In fact, if X represents a view of some larger Matrix object, these memory gaps might be larger still.

6Under the hood, views and concretes are implemented in much the same manner. Therefore, it is possible for the library to avoid unnecessary copies of concrete matrices’ data blocks in those instances when there is no possibility of violating the concrete behavioral policy. Consequently, a major efficiency advantage of the “data-view” model—fast copies—does often extend to concrete matrices. Nonetheless, there are many cases where using a concrete instead of a view can lead to unnecessary copying.

7Throughout this article, we write short code snippets assuming that the user has chosen to use both the Scythe and standard library namespaces, by placing the constructs using namespace std; and using namespace scythe; at the top of her source file. The full-length programs in sections 3 and 4 make no assumptions.
to `true` and 0 respectively, so the call `Matrix<double,Col,Concrete>M1(4, 5);` behaves identically to the above line of code. The template type of the `Matrix` is specified between `<>` and specifies the element type, ordering (Col or Row), and style (Concrete or View), always in that order.

Here is a more interesting example, which creates two possible instantiations of the `Matrix X` from Figure 1 and prints them to the terminal:

\begin{CodeChunk}
\begin{CodeInput}
double vals[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
Matrix<> Xcol(3, 4, vals);
Matrix<double,Row,View> Xrow(3, 4, false);
Xrow = 1, 4, 7, 10, 2, 5, 8, 11, 3, 6, 9, 12;

cout << Xcol << endl << Xrow;
\end{CodeInput}
\begin{CodeOutput}
1.000000 4.000000 7.000000 10.000000
2.000000 5.000000 8.000000 11.000000
3.000000 6.000000 9.000000 12.000000

1.000000 4.000000 7.000000 10.000000
2.000000 5.000000 8.000000 11.000000
3.000000 6.000000 9.000000 12.000000
\end{CodeOutput}
\end{CodeChunk}

This code shows how to create `Matrix` objects from C++ arrays and comma-delimited lists. `Xcol` constructs itself from an array, using the aptly named array constructor, which fills its object in column-major order. To generate `Xrow` we first construct an uninitialized `Matrix` using the constructor employed in the previous example. We then fill it using a special version of the assignment operator, which we will discuss in more detail in the next section. For now, simply observe that, because `Xrow` uses row-major ordering, it fills itself row by row. Notice that the template list in the declaration of `Xcol` is empty. Scythe provides default values for the template parameters of `Matrix` objects, and `Matrix<>`, `Matrix<double>`, `Matrix<double,Col>`, and `Matrix<double,Col,Concrete>` all refer to the same template type. Notice also that `Xrow` is technically a view, although it is in the somewhat peculiar position—for a view—of being the only `Matrix` viewing its particular data block.

Perhaps the most useful constructor is the file constructor, which initializes a `Matrix` object of arbitrary size and shape from a text file. Given a text file `amatrix.txt` containing a space-delimited, row-major ordered list of values, with one row per line, one can construct a `Matrix` object with the call `Matrix<> M("amatrix.txt").`\footnote{Future library releases will provide support for matrices stored in a variety of common file formats, such as comma-separated value.}
Copy Construction and Assignment

Copy construction and assignment are the two primary mechanisms for copying objects in C++. As we have emphasized, concrete matrices and views behave differently from one another in respect to both copy construction and assignment. Consider the following program:

\begin{CodeChunk}
\begin{CodeInput}
// Define aliases for Matrix template types
typedef Matrix<double,Col,Concrete> colmatrix;
typedef Matrix<double,Row,Concrete> rowmatrix;
typedef Matrix<double,Col,View> colview;

// Construct two concrete matrices
colmatrix A(2, 3, true, 0.0);
rowmatrix B(2, 3, true, 1.0);

// Perform some copy construction
rowmatrix C(A);
colview D = B;

// Now perform some assignments
C = B;
D = A;

cout << "A:\n" << A << "B:\n" << B << "C:\n" << C << "D:\n" << D;
\end{CodeInput}
\begin{CodeOutput}
A:
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
B:
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
C:
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
D:
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
\end{CodeOutput}
\end{CodeChunk}

Before considering what this code does, notice that we use the \texttt{typedef} keyword to create aliases for the various \texttt{Matrix} template types that we employ throughout the example. This trick helps to reduces the number of keystrokes necessary to declare complicated \texttt{Matrix} objects and enhances program readability. In this case we create aliases for row- and column-major matrices and column-major views. After defining the type aliases, this code constructs
two 2 × 3 matrices, A and B, filling the first with zeros and the second with ones. It then initializes two more matrices through copy construction. The first of these, C, is a concrete matrix and is a distinct copy of A. The second copy-constructed matrix, D, is a view of B. Notice that the orders of the two matrices involved in a given copy construction need not match. Also, notice the syntax we use to copy construct C differs from that used to construct D. It is important to realize that this later construct also represents an invocation of the copy constructor, even though it uses the = character, something we normally associate with assignment. Finally, we invoke the assignment operator on both the concrete matrix (C) and the view (D). The code \( C = B \) causes C, which was distinct copy of A after construction, to become a distinct copy of B. On the other hand, the instruction \( D = A \) indirectly fills B with the elements in A, as modulated by the view D.

As we foreshadowed in the previous section, Scythe provides a special form of assignment operator for Matrix that allows users to fill Matrix objects with lists of primitive values without using intermediate arrays. The code

\[
\text{Matrix } X(3, 3, \text{false});
\]

\[
X = 1, 2, 3, 4, 5, 6, 7, 8, 9;
\]

constructs a 3 × 3 Matrix object X and fills it with the values one through nine, in the matrix's template (in this case column-major) order. This list-wise assignment operator works identically across concrete matrices and views, always filling a matrix with values and overwriting its original contents, regardless of the style of the matrix. Furthermore, list-wise assignment uses a recycling rule identical to that used by the R language; if the number of elements in the right hand side list is less than that in the Matrix the operator will recycle list elements until the Matrix is full. For example, the assignment \( X = 1 \) fills X with ones,

and the assignment \( X = 1, 0, 1, 0 \) causes X to represent the matrix

\[
\begin{pmatrix}
1 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1
\end{pmatrix}
\]

Working with Elements and Sub-Matrices

The Matrix class provides a number of ways to access distinct sub-portions of matrices using multiple overloaded definitions of the () operator. These access operators all provide range-checking when Scythe's error checking facilities are set to their maximum value.

At the most basic level, users can access individual elements of matrices with index operators. There are two types of Matrix index operator, one- and two-argument. The one-argument index operator accesses a Matrix object's elements in terms of its template order, starting at the index 0. For example, the sequence of assignments

\[
\text{Xcol(0)} = \text{Xcol(3)} = \text{Xcol(6)} = \text{Xcol(9)} = 0;
\]
zeros out the first row of the Matrix Xcol defined above.\footnote{It is also possible to access single elements of a Matrix with the \texttt{[]} operator, but only using the single argument construction. Because this syntax does not generalize to the other forms of the access operator—and because code that passes two arguments to the \texttt{[]} operator will generally compile, causing hard to diagnose run-time errors—we discourage the use of this form of the single argument index operator.}

The two-argument index operator, on the other hand, allows one to reference a Matrix element by row and column subscripts, in that order. The following function uses the two-argument index operator to provide one possible implementation of matrix transposition for an arbitrary Matrix template:

```cpp
template <typename T, matrix_order O, matrix_style S>
Matrix<T, O, S> my_transpose1 (const Matrix<T, O, S>& M)
{
    Matrix<T, 0, S> result(M.cols(), M.rows(), false);
    for (unsigned int i = 0; i < M.rows(); ++i)
        for (unsigned int j = 0; j < M.cols(); ++j)
            result(j, i) = M(i, j);
    return result;
}
```

This function also makes use of two of the Matrix class’s metadata accessors, \texttt{rows()} and \texttt{cols()}, which return a Matrix object’s dimensions. Matrix contains a variety of such accessors, including predicates like \texttt{isSquare()} and \texttt{isSingular()}.

Another version of the two-argument index operator—the vector access operator—uses the placeholder object \_ to access entire sub-vectors. For example, another way to zero out the entire first row of Xcol requires only the single assignment \texttt{Xcol(0, \_)} = 0.\footnote{Note that this assignment takes advantage of both the recycling behavior of the list-wise assignment operator and the data-view model. First, the vector access operator constructs and returns a vector view of the Matrix it is invoked upon. Then, the list-wise assignment operator is invoked on that view, recycling the value on the right hand side of the equation until the view is full.}

View access motivates another possible implementation of matrix transposition:

```cpp
template <typename T, matrix_order O, matrix_style S>
Matrix<T, O, S> my_transpose2 (const Matrix<T, O, S>& M)
{
    Matrix<T, 0, S> result(M.cols(), M.rows(), false);
    for (unsigned int i = 0; i < M.rows(); ++i)
        result(_, i) = M(i, _);
    return result;
}
```

The final form of the () operator is the sub-matrix access operator. This operator allows the caller to access a rectangular region of an existing Matrix object. The operator takes four arguments: the row and column indices of the upper left corner of the sub-matrix followed
by the row and column indices of the bottom right corner of the rectangular region. Using this operator, and the vector access operator, we are now in a position to define the set of matrices described by Figure 1:

```cpp
Matrix<> X(3, 4, false);
X = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12;
Matrix<double,Col,View> a = X(_, 0);
Matrix<double,Col,View> b = X(1, _);
Matrix<double,Col,View> C = X(0, 2, 1, 3);
```

**Matrix Iterators and the STL**

Access operators do not provide the only way to access individual Matrix elements. Scythe provides a set of STL-compliant iterators that allow one to traverse a Matrix in either row-major or column-major order. The factory methods `begin()` and `end()` return iterator objects pointing to the first and last element in a Matrix object, respectively. The iterators returned by these methods traverse the Matrix object they reference in its template order. We also provide overloaded template versions of these factory methods, which allow the user to iterate over any Matrix in either row- or column-major order. Iterators play an important role within the library because, among other reasons, iterators provide a method of traversing contiguous elements in a view that is substantially faster than that provided by the index accessors. But iterators are also an invaluable tool for library users. To demonstrate the utility of these factory methods, and iterators in general, consider yet another possible implementation of transpose:

```cpp
template <typename T, matrix_order O, matrix_style S>
Matrix<T, O, S> my_transpose3 (const Matrix<T, O, S>& M)
{
    if (O == Row) {
        return Matrix<T, O, S> result(M.cols(), M.rows(), M.template begin<Col>());
    } else {
        return Matrix<T, O, S> result(M.cols(), M.rows(), M.template begin<Row>());
    }
}
```

This function uses the array constructor, which is also known as the iterator constructor. When the function argument `M` is stored in row-major order the function calls the constructor in such a way that it traverses `M` in column-major order—as specified by the factory method call `M.template begin<Col>()`—and does the opposite when `M`'s order is column-major.

---

13 For a detailed description of the different types of STL-compliant iterators see Josuttis (1999). For a more complete description of the different types of iterators provided by Scythe, see Pemstein et al. (2007).
14 For concrete matrices, index accessors incur no performance penalty.
15 Array variables in C++ are simply pointers to the first element of an array, stored sequentially in memory. Pointers are, technically, a form of iterator. Therefore the Matrix array constructor is actually an iterator constructor.
16 The `template` keyword in this call helps the compiler to correctly identify the version of `begin()` to use for this invocation.
The most important advantage afforded by iterators is that they put the power of the STL at Scythe users’ fingertips. Among other things, the STL provides a set of algorithms for performing common computational tasks on sets of objects. To abstract away from the details of myriad possible container classes, STL algorithms rely on iterators to describe ranges of data. For example, using the STL, one can shuffle X by writing

\[
\text{random\_shuffle}(X,\text{begin()}, \text{end}());
\]

or sort the second column of X with the line

\[
\text{sort}(X(_, 1).\text{begin}(), X(_, 1).\text{end}());
\]

In general, iterators allow Scythe matrices to interact with a diverse array of existing generic software.

**Arithmetic, Logical, and Linear Algebraic Operations**

The final set of facilities for working with Matrix objects consist of a variety of arithmetic and logical operators and a number of linear algebra functions. These routines allow us to add, subtract, and multiply matrices, check matrix equality, and perform common matrix transformations such as inversion and decomposition. Consider the basic problem of finding the least-squares coefficients in the linear model

\[
y = X\beta + \epsilon, \quad \epsilon \sim N_n(0, \sigma^2 I_n).
\]

Using Scythe, one can solve this problem in C++ much as one would in an interpreted language like Matlab or R, by writing

\[
\text{Matrix}<> \beta_\text{hat} = \text{invpd}(\text{crossprod}(X)) * t(X) * y;
\]

where invpd() calculates the inverse of a positive definite symmetric matrix, crossprod() computes the quantity X'X, t() returns the transpose of a matrix, and * is the matrix multiplication operation.

In short, Scythe turns C++ into a convenient language for doing matrix algebra. Scythe’s matrix operators do not only perform mathematical functions, but also support a host of logical operations. One can negate the values in a matrix X with the invocation !X or compare the elements in two conforming matrices X and Y using the ==, !=, <, >, <=, and >= operators, where, for example, X == Y returns a matrix of type `Matrix<bool>` with the same dimensions as X and Y. And, to test if X and Y are identical, one simply writes X.equals(Y). Finally, as the linear regression example demonstrated, Scythe’s mathematical and logical matrix operators are complemented by a variety of work-horse linear algebra routines. Table 1 provides an overview of Scythe’s linear algebra support and indicates whether each operation is implemented in native C++, as a wrapper to a BLAS or LAPACK routine, or both.

2.2. Pseudo-Random Deviates in Scythe

Scythe is not just a matrix library. It also contains extensive support for (pseudo)random number generation. The library takes an object-oriented approach to this problem, providing an abstract base class, `rng`, that defines an interface for random number generators (RNGs) and contains the code necessary to simulate random variates from a multitude of commonly used probability distributions, including, but not limited to, the normal, beta, binomial, and gamma distributions.

It is not possible to instantiate `rng` objects directly. Instead, the user must construct an instance of an extending class which implements the `rng` interface by providing methods that generate random uniform variates. Scythe provides two such classes. The first of these, `mersenne`, encapsulates the Mersenne Twister pseudorandom number generator developed by Matsumoto and Nishimura (1998) and is based on source code made freely available by the authors (Matsumoto and Nishimura 2002). This generator is fast and reliable, exhibiting both an extremely high period of $2^{19937} - 1$ and an order of equidistribution of 623 dimensions, and is suitable for most applications. The library also sports a second implementation of the `rng` interface, the `lecuyer` class, which uses an algorithm developed by L’Ecuyer, Simard, Chen, and Kelton (2002) and is also based on source code made freely available by the authors (L’Ecuyer 2001). This RNG provides an interface for generating multiple parallel streams of random numbers and is based on an underlying generator with a period of approximately $3.1 \times 10^{57}$ and that, according to the authors, “performs well on the spectral test in up to (at least) 45 dimensions (L’Ecuyer et al. 2002).” This RNG is useful for threaded applications that must simultaneously generate multiple independent streams of random numbers, such as programs implementing certain Markov chain Monte Carlo (MCMC) algorithms.

Constructing and using a Scythe RNG is straightforward. For example, the following code

---

17In Scythe *, performs matrix multiplication while % performs element by element multiplication. Other common mathematical operators include +, -, /, and the method kronecker().

18Where possible, we plan to provide both native C++ and BLAS/LAPACK wrappers for all linear algebra routines in future library releases. Scythe also provides a variety of matrix utility operations—column and row binding, vectorization and expansion, sorting, and so forth—and functions for solving systems of linear equations (using the various decompositions listed in Table 1) that we do not describe here. See Pemstein et al. (2007) for a full listing of these functions.

19For a full list of the probability distributions supported by Scythe’s random number generation routines, see Pemstein et al. (2007). The library also includes probability density and cumulative distribution functions for these distributions, using a syntax much like R’s.
calculates the sum of two random uniform numbers on the interval \((0, 1)\), one random variate simulated from the normal distribution with mean zero and variance one, and a single simulated value from the F distribution with 2 and 50 degrees of freedom, using the Mersenne Twister:

```cpp
mersenne myrng;
double sum = myrng() + myrng.runif() + myrng.rnorm(0, 1) + myrng.rf(2, 50);
```

Note that we can generate random uniform numbers with one of two calls. This first uniform variate in the example is generated with an invocation of the () operator, while the second is produced by the `runif()` method. Behaviorally, these calls are identical and `rng`-extending classes such as `mersenne` need only implement the `runif()` method.\(^{20}\) The () operator is implemented by the base class and allows Scythe RNGs to behave as function objects which return random uniform numbers when invoked.

**Scythe** also provides tools for using externally provided quasi-random number generators with the `rng` class. The `wrapped_generator` class allows users to extend the `rng` class by wrapping any function object that returns uniform random numbers on the \((0, 1)\) interval when its function call operator is invoked. For example, the Boost C++ libraries (http://www.boost.org) include a number of quasi-random number generators—including an alternative implementation of the Mersenne Twister generator provided by Scythe—that behave in this way. The following code instantiates an instance of the Boost Mersenne Twister, wraps it with Scythe’s `rng` class, and uses the resulting object to generate a six by six matrix of standard normal random variates:

```cpp
#include <iostream>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/uniform_real.hpp>
#include <boost/random/variate_generator.hpp>
#include <scythestat/rng/wrapped_generator.h>

// Some short-hand for the boost generator’s type
typedef boost::variate_generator<boost::mt19937&, boost::uniform_real<> > boost_twister;

// Create a function object that generates pseudorandom uniform
// variates on \((0, 1)\) using Boost’s version of the Mersenne Twister.
boost::mt19937 generator(42u);
boost_twister uni(generator, boost::uniform_real<> (0, 1));

// Wrap the boost generator
wrapped_generator<boost_twister> wgen(uni);

// Print a 6x6 matrix of standard normal variates.
cout << wgen.rnorm(6, 6, 0, 1) << endl;
```

\(^{20}\)In fact, implementing the `runif()` method, and two overloaded templates of the method, are the only requirements placed on a class implementing the `rng` interface.
As we have seen, Scythe takes advantage of C++’s polymorphic features to transparently allow users to draw random numbers from a vast array of distributions while permitting users tremendous flexibility in their choice of underlying pseudorandom number generator. The approach that rng and its extending classes use to achieve dynamic function dispatch is somewhat unconventional and deserves some discussion. A traditional implementation of the RNG class hierarchy would look something like this:

```cpp
class rng {
    public:
    virtual double runif() = 0;

    ...;
}

class rng_impl : public rng {
    public:
    double runif ()
    {
        ...
    }

    ...;
}
```

Under this traditional model, invoking the `runif()` function through a `rng` pointer or reference would dynamically invoke the function defined by the extending class. This is exactly the behavior we want, but virtual function dispatch in C++ is fraught with efficiency issues: it requires extra memory accesses and inhibits most C++ compilers from effectively inlining and optimizing the code within the virtual function. This can substantially decrease the performance of code that calls these functions often. Statistical applications, most notably MCMC estimators, often invoke the random uniform number generator tens of thousands of times in a single run and are especially susceptible to the ill effects of virtual function dispatch in their random number generators.

To avoid virtual function dispatch, and to allow the compiler to effectively optimize our RNGs, we use a technique often called the “Barton and Nackman Trick” to achieve a form of “dynamic” dispatch that is performed at compile time (Barton and Nackman 1994). `rng` and its extending classes take the form:

```cpp
template <class RNGTYPE>
class rng {
    public:
    RNGTYPE& as derived()
    {
        return static_cast<RNGTYPE&>(*this);
    }

    double runif()
```
class rng_impl : public rng<rng_impl>
{
    public:
    double runif()
    {
        ...
    }
    ...
};

In this approach, the base class is templated on the type of the derived class, allowing it to statically invoke functions in the derived class and mimic virtual function dispatch without incurring the unwanted overhead.

Users can use rng_impl in the above example as a template when implementing Scythe-compatible RNGs on top of their preferred uniform generators, automatically inheriting the ability to generate variates from multiple probability distributions from rng. However, even users who are happy with the RNG implementations provided by Scythe should be aware of this implementation wrinkle. When writing functions that take an arbitrary rng object as an argument, one needs to write definitions that appropriately deal with the fact that rng is a template class:

    // Correct
    template <typename RNGTYPE>
    void foo (rng<RNGTYPE>& generator);

    // Wrong!
    void foo (rng& generator);

2.3. Numerical Utilities in Scythe

Scythe’s last major code module is a small suite of routines—listed in Table 2—that perform numerical optimization, integration, and related operations. The keystone of this portion of the library is the BFGS() routine, which solves unconstrained nonlinear optimization problems using the Broyden-Fletcher-Goldfarb-Shanno method,\textsuperscript{21} and allows Scythe users to perform maximum likelihood estimation from within their C++ programs.

\textsuperscript{21}The numerical optimization and integration routines represent the least mature portion of Scythe. As library development progresses, we will replace BFGS() with a generic optimize() function and provide access to a variety of different optimization algorithms.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Function Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>intsimp(), adaptsimp()</td>
</tr>
<tr>
<td>Differentiation</td>
<td>gradfdifs()</td>
</tr>
<tr>
<td>Gradient Calculation</td>
<td>gradfdif()</td>
</tr>
<tr>
<td>Hessian Calculation</td>
<td>hesscdif()</td>
</tr>
<tr>
<td>Jacobian Calculation</td>
<td>jacfdif()</td>
</tr>
<tr>
<td>Optimization</td>
<td>BFGS(), zoom(), linesearch1(), linesearch2()</td>
</tr>
<tr>
<td>Solving Nonlinear Systems</td>
<td>nls_broyden()</td>
</tr>
</tbody>
</table>

Table 2: Numerical Routines

Scythe’s optimization functions all perform operations on other, user-defined, functions. For example, say we wish to calculate \( \int_{0}^{4} f(x) \, dx \) where \( f(x) = x^3 + 2x \). To accomplish this task, we first need to implement a function encapsulating \( f(\cdot) \):

```c
double x_cubed_plus_2x (double x)
{
    return (x * x * x + 2 * x);
}
```

We could then print the result using Scythe’s adaptive integration routine

```c
\begin{CodeChunk}
\begin{CodeInput}
cout << adaptsimp(x_cubed_plus_2x, 0, 4, 10) << endl;
\end{CodeInput}
\begin{CodeOutput}
80
\end{CodeOutput}
\end{CodeChunk}
```

where the last argument to `adaptsimp()` indicates the number of subintervals that the function should use when performing its calculation. While this approach—passing the function to `adaptsimp()` as a function pointer—will work, it is not the recommended way to pass function arguments to `adaptsimp()` and other Scythe methods that operate on functions. In the previous section, we noted that virtual function dispatch can often adversely affect program performance. Dereferencing function pointers is intimately related to virtual function dispatch and brings with it the same performance issues. This penalty can be quite substantial in routines, like numerical optimization or integration, where a function is repeatedly evaluated in a tight loop. Therefore, we recommend using function objects to encapsulate functional concepts.\textsuperscript{22} For example, a better way to evaluate \( \int_{0}^{4} x^3 + 2x \, dx \) is:

```c
struct x_cubed_plus_2x
{
```

\textsuperscript{22}For a detailed discussion of function objects, see Stroustrup (1997, chap. 18.4).
double operator() (double x) const
{
    return (x * x * x + 2 * x);
}

adaptsimp(x_cubed_plus_2x(), 0, 4, 10);

Using function objects with these procedures provides advantages beyond simple efficiency. As we will demonstrate in the following section, function objects allow Scythe’s optimization and integration routines to interact with functions that maintain arbitrary state information across invocations.

3. An Example Using Scythe: A Parametric Bootstrap

In this section we provide an example of how Scythe can be used to code a parametric bootstrap procedure in C++. We go on to compare the implementation to two equivalent implementations in R. We find that the Scythe implementation is only slightly more complicated than the R implementations. Further, the Scythe runtime is approximately 11% of the runtime of the R implementations.

Consider a Poisson regression model:

\[
y_i \sim \text{Poisson}(\mu_i) \quad i = 1, \ldots, n
\]

\[
\mu_i = \exp(x_i' \beta)
\]

with observed data

\[
y = \begin{bmatrix}
5 \\
6 \\
7 \\
8 \\
9
\end{bmatrix}
\]

and

\[
X = \begin{bmatrix}
1 & 1 & 0 \\
1 & 1 & 1 \\
1 & 2 & 0 \\
1 & 2 & 1 \\
1 & 3 & 0
\end{bmatrix}
\]

Instead of using the standard asymptotic results to calculate standard errors we decide to use the parametric bootstrap. One can do this in R using the boot (original by Angelo Canty <cantya@mcmaster.ca>. R port by Brian Ripley <ripley@stats.ox.ac.uk>. 2005) package:

```r
#******************************************************************************
#dataframe with the observed data
thedata <- data.frame( y=c(5,6,7,8,9),
```

\cite{efron-tibshirani93} for an introduction to the parametric bootstrap as well as the bootstrap more generally.
x1=c(1,1,2,2,3),
x2=c(0,1,0,1,0)

 glm.fit <- glm(y ~ x1 + x2, data=thedata, family=poisson)

## the easiest way (use boot)
library(boot)

 glm.fit <- glm(y ~ x1 + x2, data=thedata, family=poisson)

pois.dgp <- function(data, beta){
  X <- cbind(1, data$x1, data$x2)
  m <- exp(X %*% beta)
  newdata <- data
  newdata$y <- rpois(nrow(data), m)
  return(newdata)
}

stat.fun <- function(data){
  coef(glm(y ~ x1 + x2, data=data, family=poisson))
}

boot.out <- boot(thedata, statistic=stat.fun, R=10000,
                   sim="parametric", ran.gen=pois.dgp,
                   mle=coef(glm.fit))

Executing this code on a dual 2 GHz PowerPC G5 with 2GB DDR SDRAM running OS X version 10.4.8 and R version 2.2.0 takes approximately 167 seconds.

It is also relatively easy to conduct the same parametric bootstrap by hand in R:

theodata <- data.frame( y=c(5,6,7,8,9),
                        x1=c(1,1,2,2,3),
                        x2=c(0,1,0,1,0))

 glm.fit <- glm(y ~ x1 + x2, data=thedata, family=poisson)

## a slightly more complicated way-- all by hand
M <- 10000
beta.store <- matrix(NA, M, 3)

X <- cbind(1, thedata$x1, thedata$x2)
beta.mle <- coef(glm.fit)

for (i in 1:M){
  m <- exp(X %*% beta.mle)
y.new <- rpois(nrow(thedata), m)
glm.fit.i <- glm(y.new ~ thedata$x1 + thedata$x2, family=poisson)
beta.store[i,] <- coef(glm.fit.i)
}

This hand-rolled procedure takes about 175 seconds on the same machine as described above. It is also relatively easy to code this same procedure in C++ using Scythe. The following is the source code:
/*******Begin pboot.cc*************/
#include <iostream>
#include <scythestat/rng/mersenne.h>
#include <scythestat/distributions.h>
#include <scythestat/ide.h>
#include <scythestat/la.h>
#include <scythestat/matrix.h>
#include <scythestat/rng.h>
#include <scythestat/smath.h>
#include <scythestat/stat.h>
#include <scythestat/optimize.h>

using namespace scythe;
using namespace std;

// The PoissonModel class stores additional data to be passed to BFGS
class PoissonModel {
public:
    double operator() (const Matrix<double> beta){
        const int n = y_.rows();

        // the linear predictor
        Matrix<double> eta = X_ * beta;

        // Poisson mean parameters
        Matrix<double> m = exp(eta);

        // the loglikelihood
        double loglike = 0.0;
        for (int i=0; i<n; ++i)
            loglike += y_(i) * log(m(i)) - m(i);

        return -1.0 * loglike;
    }

    Matrix<double> y_;  
    Matrix<double> X_; 
};

int main (){ 

    // set the type of pseudo random number generator (Mersenne Twister)
    mersenne myrng;

}
// generate synthetic data
Matrix< double > y = seqa(5, 1, 5);
Matrix< double > X(5, 3, false);
X = 1, 1, 1, 1, 1, 1, 1, 2, 2, 3, 0, 1, 0, 1, 0;

// instantiate the PoissonModel
PoissonModel poisson_model;
poisson_model.y_ = y;
poisson_model.X_ = X;

// starting values (OLS estimates with logged y)
Matrix< double > theta = invpd(crossprod(X)) * t(X) * log(y);

// Compute the Poisson regression MLEs
Matrix< double > beta_MLE = BFGS(poisson_model, theta, myrng, 100, 1e-5, true);

const int n = y.rows(); // number of observations
const int M = 10000; // number of bootstrap samples
Matrix< double > beta_bs_store(M, 3); // storage matrix for bootstrap samples
Matrix< double > y_bs(5, 1); // holder for bootstrap y

// The parametric bootstrap
for (int i=0; i<M; ++i){

    // the linear predictor
    Matrix< double > eta = X * beta_MLE;

    // Poisson mean parameters
    Matrix< double > m = exp(eta);

    // generate parametric bootstrap y
    for (int j=0; j<n; ++j)
        y_bs(j) = myrng.rpois(m(j));

    poisson_model.y_ = y_bs;

    // calculate the bootstrap value of beta
    Matrix< double > beta_bs = BFGS(poisson_model, beta_MLE, myrng, 100, 1e-5);

    // store the bootstrap value of beta
    beta_bs_store(i,_) = beta_bs;
}

<table>
<thead>
<tr>
<th></th>
<th>R (boot)</th>
<th>R (by hand)</th>
<th>Scythe</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MLE</td>
<td>Bootstrap SE</td>
<td>MLE</td>
</tr>
<tr>
<td>$\beta_1$</td>
<td>1.340</td>
<td>0.612</td>
<td>1.340</td>
</tr>
<tr>
<td>$\beta_2$</td>
<td>0.289</td>
<td>0.257</td>
<td>0.289</td>
</tr>
<tr>
<td>$\beta_3$</td>
<td>0.162</td>
<td>0.400</td>
<td>0.162</td>
</tr>
</tbody>
</table>

Table 3: Parametric Bootstrap Results

```cpp
// Print a summary of the results
cout << "The MLEs are: " << endl;
std::cout << t(beta_MLE) << "\n";
cout << "The bootstrap SEs are: " << endl;
std::cout << sdc(beta_bs_store) << "\n";

// write the bootstrap samples to a file
beta_bs_store.save("out.txt");

return 0;
}
```

This code can be compiled using GCC with:
```
g++ -O3 -funroll-loops pboot.cc -o pboot
```

and run in the usual way:
```
./pboot
```

Doing so, we find that this code takes approximately 19 seconds to run.\(^{24}\) Figure 2 displays a comparison of the run times for the three implementations of the parametric bootstrap. Note that while the Scythe version requires more code to be written, the individual pieces of code are not difficult to decipher by someone who has some experience with a language such as R or Matlab— even if they have no experience with Scythe or C++. Further, the Scythe implementation is dramatically faster than either R implementation. Finally, as Table 3 shows, the three implementations generate virtually identical maximum likelihood estimates and bootstrapped standard errors.

4. Using Scythe in R Packages

It is also very easy to use Scythe in C++ code that is called from R. Indeed, several R packages already make use of an older version of Scythe.\(^{25}\)

---

\(^{24}\) Additionally, compiling with the `-O3` and `-funroll-loops` options took 11 seconds.

\(^{25}\) R packages that make use of some version of Scythe include: MasterBayes Hadfield (2006), Matching (Sekhon 2005), MCMCpack (Martin and Quinn 2007), and smoothSurv (Komarek 2005).
Figure 2: Run time comparison for parametric bootstrap example.
Perhaps the easiest way to use Scythe in conjunction with R is to install Scythe on a local machine as described in the Scythe distribution and to then #include the necessary Scythe header files from the library path and build accordingly. For instance, the following code (in a file called Axplusb1.cc) calculates \( Ax + b \) quickly in C++ using the Scythe gaxpy() function.

```cpp
#include <scythestat/la.h>
#include <scythestat/matrix.h>

using namespace scythe;

extern "C"{

    // computes \( y = A \cdot x + b \) and returns the result in ydata
    void AxplusbScythe (const double* Adata, const int* Arow, const int* Acol,
                        const double* xdata, const int* xrow, const int* xcol,
                        const double* bdata, const int* brow, const int* bcol,
                        double* ydata){

        Matrix<double, Col> A(*Arow, *Acol, Adata);
        Matrix<double, Col> x(*xrow, *xcol, xdata);
        Matrix<double, Col> b(*brow, *bcol, bdata);

        // calculate \( y = A \cdot x + b \)
        Matrix<double> y = gaxpy(A, x, b);

        // put result back in ydata
        for (int i = 0; i < *brow; ++i){
            ydata[i] = y(i);
        }
    }
}
```

To build this into a shared library Unix, Linux, or Mac OS X, one simply types:

```
R CMD SHLIB Axplusb1.cc
```

at the shell prompt. This generates a shared object file called `Axplusb1.so` that can be loaded into R and called in the usual way:
Axplusb <- function(A, x, b){

dyn.load("Axplusb1.so")

output <- .C("AxplusbScythe",
            Adata = as.double(A),
            Arow = as.integer(nrow(A)),
            Acol = as.integer(ncol(A)),
            xdata = as.double(x),
            xrow = as.integer(nrow(x)),
            xcol = as.integer(ncol(x)),
            bdata = as.double(b),
            brow = as.integer(nrow(b)),
            bcol = as.integer(ncol(b)),
            ydata = as.double(b))

result <- as.matrix(output$ydata)

return (result)
}

In some situations — such as when Scythe is bundled with a full R package — it is useful to make use of local copies of all the Scythe header files. Consider the following code in a file called Axplusb2.cc. This code assumes that the two Scythe header files that are used (la.h, and matrix.h) are in the same directory as Axplusb2.cc. This can be built into a shared object file in the same manner as above with the exception that a Makevars file should be created in the same directory as Axplusb2.cc and the header files. This Makevars file should contain the line:

\[
\text{PKG_CXXFLAGS = -DSCYTHE\_COMPILE\_DIRECT}
\]

which alerts the compiler that all of the header files are in a single directory.

```c
#include "la.h"
#include "matrix.h"
using namespace scythe;
extern "C"{

// computes y = A \times x + b and returns the result in ydata
void AxplusbScythe (const double* Adata, const int* Arow, const int* Acol,
                    const double* xdata, const int* xrow, const int* xcol,
                    const double* bdata, const int* brow, const int* bcol,
```

}

```cpp
double* ydata)

    Matrix<double, Col> A(*Arow, *Acol, Adata);
    Matrix<double, Col> x(*xrow, *xcol, xdata);
    Matrix<double, Col> b(*brow, *bcol, bdata);

    // calculate y = A * x + b
    Matrix<double> y = gaxpy(A, x, b);

    // put result back in ydata
    for (int i = 0; i < *brow; ++i)
        ydata[i] = y(i);
}

R code similar to that above — with the exception of a change to the argument to `dyn.load` — can be used to load and call this function.

5. Discussion

In this paper we have discussed the major design points of the Scythe library and provided examples of how it can be used. We have attempted to make Scythe nearly as easy to work with as R while also being nearly as computationally fast as pure C.

Nonetheless, Scythe is a work in progress and we plan to continue developing Scythe in the future. In particular, we hope to:

- provide additional linear algebra routines, numerical optimizers, and distributions, densities, and pseudorandom number generators
- continue to optimize the code base for speed, both by improving native Scythe code and by wrapping additional high quality BLAS and LAPACK routines
- provide more general interfaces to several broad classes of functions such as the matrix decomposition functions and the numerical optimization functions
- provide generic tools for MCMC so that MCMC routines could be written quickly with very few lines of code

We also welcome feedback and code from Scythe users.

6. Acknowledgements

We have received a great deal of support for the Scythe project. First, we thank those people who have used our software, found bugs, and provided suggestions, feedback, and patches.
Second, we would like to thank all of the research assistants who have worked with us at Harvard University and Washington University over the years on this project: Matthew Fasman, Steve Haptonstahl, Kate Jensen, Laura Keys, Kevin Rompala, Joe Sheehan, and Jean Yang. Finally, we gratefully acknowledge financial support from: the U.S. National Science Foundation, Program in Methodology, Measurement, and Statistics, Grants SES-0350646 and SES-0350613, the Institute for Quantitative Social Sciences (http://iq.harvard.edu) at Harvard University, the Weidenbaum Center on the Economy, Government, and Public Policy (http://wc.wustl.edu) at Washington University, and the Center for Empirical Research in the Law (http://cerl.wustl.edu) at Washington University. Neither the National Science Foundation, Washington University, or Harvard University bear any responsibility for the content of this package. In addition, Quinn gratefully acknowledges the hospitality and support of the Center for Advanced Study in the Behavioral Sciences.

This work is licensed under a Creative Commons Attribution-No Derivative Works 3.0 License.

References


original by Angelo Canty <cantya@mcmasterca> R port by Brian Ripley <ripley@statsoxacuk> S (2005). *boot: Bootstrap R (S-Plus) Functions (Canty).* R package version 1.2-24.


**Affiliation:**

Daniel Pemstein  
Department of Political Science  
University of Illinois  
361 Lincoln Hall  
702 S. Wright Street  
Urbana, IL 61801  
E-mail: dbp@uiuc.edu  
URL: [http://www.danpemstein.com/](http://www.danpemstein.com/)

Kevin M. Quinn  
Department of Government and  
The Institute for Quantitative Social Science  
Harvard University  
1737 Cambridge Street  
Cambridge, MA 02138  
E-mail: kevin.quinn@harvard.edu  
URL: [http://www.people.fas.harvard.edu/~kquinn/](http://www.people.fas.harvard.edu/~kquinn/)