Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments

Technical Report

Conrad Sanderson

NICTA, PO Box 6020, St Lucia, QLD 4067, Australia
http://nicta.com.au

September 2010
(revised December 2011)

Abstract

In this report we provide an overview of the open source Armadillo C++ linear algebra library (matrix maths). The library aims to have a good balance between speed and ease of use, and is useful if C++ is the language of choice (due to speed and/or integration capabilities), rather than another language like Matlab or Octave. In particular, Armadillo can be used for fast prototyping and computationally intensive experiments, while at the same time allowing for relatively painless transition of research code into production environments. It is distributed under a license that is applicable in both open source and proprietary software development contexts. The library supports integer, floating point and complex numbers, as well as a subset of trigonometric and statistics functions. Various matrix decompositions are provided through optional integration with LAPACK, or one its high-performance drop-in replacements, such as MKL from Intel or ACML from AMD. A delayed evaluation approach is employed (during compile time) to combine several operations into one and reduce (or eliminate) the need for temporaries. This is accomplished through C++ template meta-programming. Performance comparisons suggest that the library is considerably faster than Matlab and Octave, as well as previous C++ libraries such as IT++ and Newmat.

This report reflects a subset of the functionality present in Armadillo v2.4.

Armadillo can be downloaded from:

- **http://arma.sourceforge.net**

If you use Armadillo in your research and/or software, we would appreciate a citation to this document. Please cite as:

- Conrad Sanderson. *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments*. Technical Report, NICTA, 2010.

Contents

1 Introduction **3**

2 Previous C++ Libraries for Linear Algebra **4**

3 Feature Criteria **5**

4 User Accessible Functionality **5**

5 Speed Comparison **10**

6 License Choice **12**

7 Development Methodology & Observations **12**

8 Internal Architecture Overview **13**

9 Conclusion **15**

List of Tables

1 Member functions and variables of the Mat class (subset). 6

2 Matrix operations involving overloaded C++ operator functions (subset). 7

3 Matrix decompositions and related functions (subset). 7

4 Functions for generating matrices (subset). 7

5 Scalar valued functions (subset). 7

6 Scalar/vector valued functions (subset). For the *dim* argument, *dim = 0* indicates *traverse across rows* (e.g. operate on all elements in a column), while *dim = 1* indicates *traverse across columns* (e.g. operate on all elements in a row). 8

7 Matrix valued functions (subset). 8

8 Examples of Matlab/Octave syntax and conceptually corresponding Armadillo syntax. Note that for submatrix access (e.g. .rows(), .submat(), etc) the exact conversion from Matlab/Octave to Armadillo syntax will require taking into account that indexing starts at 0. 9

9 Operations used in speed tests. Matrices A, B, C, D and Q have the size of N×N, where N = 50 for in-cache tests, and N = 500 for out-of-cache tests. For the "decreasing size matrix multiplication" operation, dimensions of A, B, C, D for the in-cache case are 100×80, 80×60, 60×40, 40×20, respectively; for the out-of-cache case, the dimensions are 1000×800, 800×600, 600×400, 400×200, respectively. 11

10 Speedup factor of Armadillo relative to other software, using in-cache matrices. Tests were performed on an Intel Core2 Duo CPU with 2 Mb cache, running in 64 bit mode at 2 GHz. Each matrix element was stored as a double precision floating point number (8 bytes). A Linux based operating system was used (Fedora 12), incorporating Linux kernel v2.6.32 and the GCC v4.4.4 C++ compiler. Versions of software were as follows: Armadillo 0.9.80, Matlab v7.1.0.183 SP3 64-bit, Octave v3.2.3, Newmat 11 beta, IT++ v4.0.6. 11

11 As per Table 10, but using out-of-cache matrices. 11

1 Introduction

Popular tools such as Matlab[1] (and its open-source counterpart, Octave[2]) are often used for algorithm development and prototyping in academic as well as industrial research contexts. While providing a syntax that is easily accessible to novices, Matlab & Octave have several major problems, limiting their usefulness. The list of problems includes:

- Matlab is a proprietary program, resulting in the reduction of accessibility and deployability of programs written in the Matlab language. As such, Matlab users are in effect subject to a vendor lock-in [3, 9, 17]. While the availability of Octave has reduced this problem, not all of Matlab's functionality is currently present in Octave.
- The Matlab language is weakly dynamically typed, where type checking is performed at run-time as opposed to compile-time. As such, it's possible to write programs which cannot be completely checked for correct syntax usage, increasing the risk of breakage in any deployed programs, or run-time errors occurring during a lengthy experiment. Furthermore, there is a performance penalty associated with run-time checks.
- The Matlab language was originally intended to be interpreted at run-time, often resulting in considerably slower speed of execution when compared to running direct machine code (produced, for example, by a C++ compiler). While recent Matlab editions contain a Just-In-Time (JIT) compiler for improving run-time performance [5], JIT compilers have their own problems, such as startup time delay, considerably increased memory usage and reduced opportunities for program optimisation [15].
- While it's possible to call functions written in other languages (such as C and Fortran) from a Matlab program, functions written in Matlab cannot be easily called from other languages.

Due to the above issues, programs or algorithms written in the Matlab language are typically converted to another language (such as C, C++ or Java) for the purposes of commercial deployment or when targeting embedded platforms. This brings in its own set of problems, such as the need to reimplement many Matlab functions and the requirement for further rounds of testing to ensure correct algorithm translation.

We have encountered all of the above problems during the research, development and commercialisation phases of several projects within NICTA. Faced with the need to use a less time-intensive prototyping and development framework, we selected C++ as the language of choice due to its speed, integration capabilities as well as ease of parallelisation via OpenMP present in modern C++ compilers [4]. Furthermore, in contrast to C and Java, the C++ language allows the overloading of mathematical operators (e.g. $+$, $-$, $*$), thereby allowing formulas to be written in a natural format, in a similar fashion to Matlab.

We looked at several open-source linear algebra libraries available for the C++ language, however none of them were able to adequately replace the required parts of Matlab's functionality while at the same time addressing the limitations of Matlab. As such, we elected to devise and implement our own library for linear algebra and related operations.

The result is the Armadillo C++ library[3], which can be used for rapid prototyping of production quality code. The library is currently comprised of about 44,000 lines of code, or about 69,000 raw lines (including comments, blank lines, etc). It has an associated user documentation comprising about 9,500 lines of HTML code (translating to roughly 16,000 words) and about 11,000 lines of code for test programs. The library has been successfully used for the development of several algorithms and systems, such as [12, 13, 14, 16, 20].

[1] http://www.mathworks.com/products/matlab/
[2] http://www.gnu.org/software/octave/
[3] http://arma.sourceforge.net

The purpose of this report is to provide an overview of the library in terms of functionality, speed and how it differs from previous C++ linear algebra libraries. Furthermore, we describe our choice of the open source license, as well as experiences with C++ template meta-programming and the open-source development model.

The report proceeds as follows. In section 2 we describe some of the limitations of previous C++ linear algebra libraries, leading to Section 3 where we list the design criteria for Armadillo. User accessible functionality is overviewed in Section 4. A speed comparison against Matlab and Octave, as well as the IT++ and Newmat libraries, is given in Section 5. In Section 6 we discuss the choice of the open-source license. The development methodology and observations gleaned from experience are described in Section 7. An overview of Armadillo's internal architecture is given in Section 8. Finally, a summary of the main points is given in Section 9.

2 Previous C++ Libraries for Linear Algebra

We examined several open-source linear algebra libraries available for the C++ language: Newmat[4], uBLAS[5] (one of the Boost C++ libraries), and IT++ (also known as ITPP[6]). None of them were able to adequately replace the required parts of Matlab's functionality while at the same time addressing the limitations of Matlab.

It must be noted that while it is also possible to directly use the widely known BLAS[7] and LAPACK[8] libraries, the verbosity of their programming interfaces makes them quite unwieldy and error-prone in day to day use. However, both BLAS and LAPACK can serve as building blocks as their interfaces are a de-facto industrial standard, with many vendors providing their own high-performance drop-in replacement libraries (e.g. MKL[9] from Intel and ACML[10] from AMD).

The limitations of the Newmat library include: **(i)** unclear license from a legal point of view, thereby providing uncertainty as to whether code based on Newmat can be used in proprietary (closed-source) programs without encumbrance; **(ii)** unnecessary reimplementation of matrix multiplication and decomposition functions, instead of using the well-tested LAPACK library as a building block; **(iii)** as LAPACK is not used, high-performance LAPACK replacements also cannot be used, affecting speed.

While the uBLAS library is distributed under a license that is compatible with commercial development, the library's syntax is, in our view, unnecessarily verbose. While uBLAS provides an efficient framework for matrix manipulations, it does not have basic functionality such as matrix inversion, which is required for our purposes.

In contrast, IT++ provides all the necessary functionality, but is only available under the restrictive GNU General Public License[11] (GPL), making it incompatible with the development of closed source software. We will briefly describe the GPL in section 6.

An further drawback of the IT++ library is the lack of a delayed-operations framework within its internal architecture, in contrast to the Newmat and uBLAS libraries. A delayed-operations framework (also known as lazy evaluation [18]) allows the combination of several operations into one and hence can considerably reduce the generation of temporary objects. This in turn can provide considerable performance improvements as well as reduced memory usage.

[4] http://www.robertnz.net/nm_intro.htm
[5] http://www.boost.org/doc/libs/release/libs/numeric
[6] http://itpp.sourceforge.net
[7] BLAS stands for Basic Linear Algebra Subprograms; a reference implementation is available at http://www.netlib.org/blas/
[8] LAPACK stands for Linear Algebra PACKage; a reference implementation is available at http://www.netlib.org/lapack/
[9] http://software.intel.com/en-us/intel-mkl/
[10] http://www.amd.com/acml
[11] http://opensource.org/licenses/

3 Feature Criteria

Based on our requirements and given the above problems with the Newmat, uBLAS and IT++ libraries, we compiled a list of necessary features and attributes prior to developing Armadillo:

- An easy to use syntax, similar to Matlab and Octave. Easy syntax reduces the time required to develop user software as well as making the development process less error prone.
- Support numerical element types useful in linear algebra as well as in signal and image processing. Specifically, there must be support for integer, floating point and complex elements.
- Use of the LAPACK library as a building block for providing various matrix decompositions. As a side-effect, this also allows processor-specific high-performance LAPACK replacement libraries to be used.
- An efficient delayed-operations framework, based on template meta-programming and recursive templates [1, 19], allowing fast matrix manipulations.
- An efficient mechanism to provide read and write access to sub-matrices.
- Ability to load and save matrices from files.
- Subsets of trigonometric and statistics functions, including keeping statistics of continuously sampled multi-dimensional processes.
- Ability to optionally specify matrix sizes using template arguments, leading to performance improvements due to compile-time memory allocation.
- Allow easy interfacing with other libraries, by providing access to elements via STL-iterators [10] as well as matrix initialisation from an existing memory block.
- Open source development, allowing contributions from outside parties. The contributions include new features as well as bug reports.
- Distribution of the library source code under a license that is appropriate for the development of both open-source and closed source (proprietary) software. We have chosen the LGPL license (which is different to the GPL), as described in Section 6.
- Cross-platform, usable on Unix-like systems (e.g. Linux, Solaris, Mac OS X, BSD), as well as lower grade systems such as MS Windows.

4 User Accessible Functionality

In addition to elementary arithmetic functions on matrices (such as addition and multiplication), Armadillo provides efficient submatrix access as well as various matrix decompositions and related functions. An overview of a subset of the functionality[12] is given in Tables 1 through to 7. Table 1 briefly describes the main member functions of the *Mat* class[13], Table 2 lists the main subset of overloaded C++ operators, Table 3 outlines available matrix decompositions, Table 4 overviews functions for generating matrices, while Tables 5, 6 and 7 describe scalar, vector and matrix valued functions of matrices, respectively.

As one of the aims of Armadillo is to facilitate conversion of code written in Matlab/Octave into C++, Table 8 provides examples of conversion to Armadillo syntax. Figure 1 shows a simple Armadillo-based C++ program.

[12] The full documentation is available online at **http://arma.sourceforge.net/docs.html**

[13] As described in Section 8, the *Mat* is template class parameterised by the element type (eg. *double*). For convenience and ease of use, there are several predefined typedefs, eg. *mat* and *cx_mat* are equivalent to *Mat<double>* and *Mat< std::complex<double> >*, respectively.

Function/Variable	Description
.n_rows	number of rows (read only)
.n_cols	number of columns (read only)
.n_elem	total number of elements (read only)
(i)	access the *i*-th element, assuming a column-by-column layout
(r, c)	access the element at row *r* and column *c*
[i]	as per (i), but no bounds check
.at(r, c)	as per (r, c), but no bounds check
.memptr()	obtain the raw memory pointer to element data
.in_range(i)	test whether the *i*-th element can be accessed
.in_range(r, c)	test whether the element at row *r* and column *c* can be accessed
.reset()	set the number of elements to zero
.copy_size(A)	set the size to be the same as matrix *A*
.set_size(rows, cols)	change the size to specified dimensions, without preserving data
.reshape(rows, cols)	change the size to specified dimensions, while preserving data
.fill(k)	set all elements to be equal to *k*
.ones(rows,cols)	set all elements to one, optionally first resizing to specified dimensions
.zeros(rows,cols)	as above, but set all elements to zero
.randu(rows,cols)	as above, but the elements are set to uniformly distributed random values in the [0,1] interval
.randn(rows,cols)	as above, but use a Gaussian/normal distribution with $\mu = 0$ and $\sigma = 1$
.is_empty()	test whether there are no elements
.is_finite()	test whether all elements are finite
.is_square()	test whether the matrix is square
.is_vec()	test whether the matrix is a vector
iterators:	STL-style iterators for accessing elements
.begin()	iterator pointing at the first element
.end()	iterator pointing at the *past-the-end* element
.begin_row(i)	iterator pointing at first element of row *i*
.end_row(j)	iterator pointing at one element past row *j*
.begin_col(i)	iterator pointing at first element of column *i*
.end_col(j)	iterator pointing at one element past column *j*
.print(header)	print elements to the *cout* stream, with an optional header
.print_trans(header)	as per .print(), but print the transposed version
storage in files:	
.save(name, format)	store matrix in the specified file, optionally specifying storage format
.load(name, format)	retrieve matrix from the specified file, optionally specifying format
submatrix views:	
.diag(k)	read/write access to the *k*-th diagonal
.row(i)	read/write access to row *i*
.col(j)	read/write access to column *j*
.rows(a,b)	read/write access to a submatrix, spanning from row *a* to row *b*
.cols(c,d)	read/write access to a submatrix, spanning from column *c* to column *d*
.submat(p, q, r, s)	read/write access to a submatrix, starting at *(p,q)* and ending at *(r,s)*
.submat(span(p,r), span(q,s))	read/write access to a submatrix spanning rows *p* to *r* and columns *q* to *s*
.swap_rows(p,q)	swap the contents of specified rows
.swap_cols(p,q)	swap the contents of specified columns
.insert_rows(row, X)	insert a copy of *X* at the specified row
.insert_cols(col, X)	insert a copy of *X* at the specified column
.shed_rows(first_row, last_row)	remove the specified range of rows
.shed_cols(first_col, last_col)	remove the specified range of columns

Table 1: Member functions and variables of the Mat class (subset).

Operation	Description
A − k	subtract scalar *k* from all elements in *A*
k − A	subtract each element in *A* from scalar *k*
A + k, k + A	add scalar *k* to all elements in *A*
A ∗ k, k ∗ A	multiply matrix *A* by scalar *k*
A + B	add matrices *A* and *B*
A − B	subtract matrix *B* from *A*
A ∗ B	matrix multiplication of *A* and *B*
A % B	element-wise multiplication of *A* and *B*
A / B	element-wise division of *A* by *B*

Table 2: Matrix operations involving overloaded C++ operator functions (subset).

Function	Description
inv(X)	Inverse of a square matrix *X*
pinv(X)	Moore-Penrose pseudo-inverse of a non-square matrix *X*
solve(A,B)	Solve a system of linear equations *AX* = *B*, where *X* is unknown
svd(X)	Singular value decomposition of *X*
eig_sym(X)	Eigen decomposition of a symmetric/hermitian matrix *X*
eig_gen(val, left, right, X)	Eigen decomposition of a general (non-symmetric/non-hermitian) square matrix *X*, with the resulting eigenvalues stored in *val*, while the left and right eigenvector matrices are stored in *left* and *right*
princomp(X)	Principal component analysis of *X*, where each row of *X* is an observation and each column is a variable
chol(X)	Cholesky decomposition of a symmetric, positive-definite matrix *X*, such that *R.t()*R = X*
qr(Q, R, X)	QR decomposition of *X*, such that *QR* = *X*
lu(L, U, P, X)	Lower-upper decomposition of *X*, such that *PX* = *LU* and *X* = *P'LU*

Table 3: Matrix decompositions and related functions (subset).

Function	Description
eye(rows, cols)	matrix with the elements along the main diagonal set to one; if *rows* = *cols*, an identity matrix is generated
ones(rows, cols)	matrix with all elements set to one
zeros(rows, cols)	matrix with all elements set to zero
randu(rows, cols)	matrix with uniformly distributed random values in the [0,1] interval
randn(rows, cols)	matrix with random values from a normal distribution with $\mu = 0$ and $\sigma = 1$
linspace(start, end, n)	vector with *n* elements, linearly increasing from *start* upto (and including) *end*
repmat(A, p, q)	replicate matrix *A* in a block-like fashion, resulting in *p* by *q* blocks of *A*
toeplitz(A, B)	Toeplitz matrix, with first col specified by *A* and first row optionally specified by *B*

Table 4: Functions for generating matrices (subset).

Function	Description
accu(A)	accumulate (sum) all elements of *A*
as_scalar(expression)	evaluate an expression that results in a 1×1 matrix, convert the result to a scalar
det(A)	determinant of square matrix *A*
log_det(x, sign, A)	log determinant of square matrix *A*, such that the determinant is exp(*x*)*sign
dot(A,B)	dot product of *A* and *B*, assuming they are vectors with equal number of elements
norm_dot(A,B)	normalised version of *dot(A,B)*
norm(A,p)	*p*-norm of *A*, with $p = 1, 2, \cdots,$ or *p* = "-inf", "inf", "fro"
rank(A)	rank of matrix *A*
trace(A)	sum of the diagonal elements of square matrix *A*

Table 5: Scalar valued functions (subset).

Function	Description
max(A, dim)	find the maximum in each column of A (dim = 0), or each row of A (dim = 1) (default: dim = 0)
min(A, dim)	as above, but find the minimum
prod(A, dim)	as above, but find the product
sum(A, dim)	as above, but find the sum
statistics:	
mean(A, dim)	as above, but find the average
median(A, dim)	as above, but find the median
stddev(A, dim)	as above, but find the standard deviation
var(A, dim)	as above, but find the standard variance
diagvec(A, k)	extract the k-th diagonal from matrix A (default: $k = 0$)

Table 6: Scalar/vector valued functions (subset). For the *dim* argument, *dim = 0* indicates *traverse across rows* (e.g. operate on all elements in a column), while *dim = 1* indicates *traverse across columns* (e.g. operate on all elements in a row).

Function	Description
cor(A, B)	the element at (i, j) of cor(A, B) is the correlation between the i-th variable in A and the j-th variable in B, where each row of A and B is an observation and each column is a variable.
cov(A, B)	as per cor(A,B), but calculate the covariance
conv(A, B)	convolution of A and B, assuming they are vectors
cross(A, B)	cross product of A and B, assuming they are 3 dimensional vectors
kron(A, B)	Kronecker tensor product of A and B
find(A)	find the indices of non-zero elements of A; can use *find(A > k)* to find indices of all elements that are $> k$
fliplr(A)	copy A with the order of the columns reversed
flipud(A)	copy A with the order of the rows reversed
join_rows(A, B)	append each row of B to its respective row of A
join_cols(A, B)	append each column of B to its respective column of A
reshape(A, r, c)	copy A with dimensions set to r rows and c columns
conj(C)	complex conjugate of complex matrix C
real(C)	extract the real part of complex matrix C
imag(C)	extract the imaginary part of complex matrix C
shuffle(A, dim)	copy A with the rows ($dim = 0$), or columns ($dim = 1$) shuffled
sort(A, dim)	copy A with the elements sorted in each column ($dim = 0$) or row ($dim = 1$)
sort_index(A)	Assuming A is a vector, generate a vector which describes the sorted order of A's elements (i.e. the indices)
trans(A)	Hermitian transpose of A (for complex matrices, conjugate is taken)
strans(A)	simple matrix transpose of A (complex conjugate is not taken)
diagmat(A)	interpret matrix A as a diagonal matrix (this can save computation time during multiplication)
misc(A)	apply a miscellaneous function to each element of A, where *misc* can be: *pow*, *exp*, *log*, *log10*, *sqrt*, *square*, ...
trig(A)	apply a trigonometric function to each element of A, where *trig* is one of: *cos*, *sin*, *tan*, *acos*, *asin*, *atan*, ...

Table 7: Matrix valued functions (subset).

Matlab & Octave	Armadillo	Notes
A(1, 1) A(k, k)	A(0, 0) A(k-1, k-1)	indexing in Armadillo starts at 0, following C++ convention
size(A,1) size(A,2) size(Q,3) numel(A)	A.n_rows A.n_cols Q.n_slices A.n_elem	member variables are read only Q is a cube (3D array) .n_elem indicates the total number of elements
A(:, k) A(k, :) A(:, p:q) A(p:q, :) A(p:q, r:s)	A.col(k) A.row(k) A.cols(p, q) A.rows(p, q) A.submat(p, r, q, s) or A.submat(span(p, q), span(r,s))	read/write access to a specific column read/write access to a specific row read/write access to a submatrix spanning the specified cols read/write access to a submatrix spanning the specified rows A.submat(first_row, first_col, last_row, last_col) A.submat(span(first_row, last_row), span(first_col, last_col))
Q(:, :, k) Q(:, :, t:u)	Q.slice(k) Q.slices(t, u)	Q is a cube (3D array)
A′ A.′	A.t() or trans(A) A.st() or strans(A)	Hermitian transpose (for complex matrices, the conjugate of each element is taken) simple transpose (the conjugate of each element is not taken)
A = zeros(size(A)) A = ones(size(A)) A = zeros(k) A = ones(k)	A.zeros() A.ones() A = zeros<mat>(k,k) A = ones<mat>(k,k)	set all elements to zero set all elements to one create a matrix with elements set to zero create a matrix with elements set to one
C = complex(A,B)	cx_mat C = cx_mat(A,B)	construct a complex matrix out of two real matrices
A .∗ B A ./ B A \ B A = A + 1 A = A − 1	A % B A / B solve(A,B) A++ A−−	% indicates element-wise multiplication / indicates element-wise division solve a system of linear equations
A = [1 2; 3 4;]	A << 1 << 2 << endr << 3 << 4 << endr	special element *endr* indicates *end of row*
X = [A B] X = [A; B]	X = join_rows(A,B) X = join_cols(A,B)	
A	cout << A << endl or A.print("A:")	print the contents of a matrix to the standard output
save -ascii 'A.dat' A load -ascii 'A.dat'	A.save("A.dat", raw_ascii) A.load("A.dat", raw_ascii)	Matlab/Octave matrices saved as ascii are readable by Armadillo (and vice-versa)
S = { 'ab', 'cd'; 'ef', 'gh'; }	field<std::string> S(2,2); S(0,0) = "ab"; S(0,1) = "cd"; S(1,0) = "ef"; S(1,1) = "gh";	the field class can store arbitrary objects, such as strings or matrices

Table 8: Examples of Matlab/Octave syntax and conceptually corresponding Armadillo syntax. Note that for submatrix access (e.g. .rows(), .submat(), etc) the exact conversion from Matlab/Octave to Armadillo syntax will require taking into account that indexing starts at 0.

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

int main(int argc, char** argv)
  {
  mat A = randu<mat>(4,5);
  mat B = randu<mat>(4,5);

  cout << A * trans(B) << endl;

  return 0;
  }
```

Figure 1: A simple Armadillo-based C++ program.

5 Speed Comparison

We performed several tests in order to gauge the speed of the library in relation to Matlab, Octave, Newmat and IT++. The tests covered a range of operations which can be expected to be used in various algorithms. The operations, along with their implementation in Matlab & Octave, are listed in Table 9. Implementations for Armadillo, IT++ and Newmat were adapted to the syntax of each library.

Each operation was placed in a loop that ran for at least 5 seconds. The time taken to execute the loop was then divided by the number of iterations of the loop, providing the time taken for a particular operation. The average of 10 runs of each test was taken as the final result.

The tests were performed on an Intel Core2 Duo CPU with 2 Mb cache, running in 64 bit mode at 2 GHz. All power management functionality of the operating system (e.g. CPU frequency scaling based on demand) was turned off. Each matrix element was stored as a double precision floating point number (8 bytes). Each test was performed for two cases: **(i)** in-cache, where all matrices fit within the CPU's cache, and **(ii)** out-of-cache, where not all matrices fit in the cache.

Tests using Armadillo, IT++ and Newmat were compiled with the GCC C++ 4.4 compiler, using the following command line:

```
g++ prog.cpp -o prog -O2 -fwhole-program -l lib
```

where `lib` is one of: `armadillo`, `itpp` or `newmat`. The `-O2` and `-fwhole-program` options enable general and interprocedural optimisation, respectively, both with an aim to improve performance. The Armadillo and IT++ libraries were configured to use the open source ATLAS 3.8 library[14], which can be used as an optimised replacement for LAPACK and BLAS.

The results for the in-cache and out-of-cache cases are listed in Tables 10 and 11, respectively. In almost all cases, test programs using Armadillo are noticeably faster than the corresponding programs using Matlab, Octave, Newmat or IT++. In two cases Armadillo obtains the same speed, involving the "transpose, matrix multiplication and in-place addition" operation.

[14] http://math-atlas.sourceforge.net/

Operation	Matlab & Octave notation
Addition and scalar multiplication	Q = 0.1*A + 0.2*B + 0.3*C;
Transpose, matrix multiplication and in-place addition	Q = Q + 0.1*A' * 0.2*B;
Decreasing size matrix multiplication	Q = A*B*C*D;
Submatrix copy	A(2:N, 2:N) = B(1:N-1, 1:N-1);

Direct element access	for c = 1:N
	  for r = 1:N
	  Q(r,c) = A(N+1-r, c)
	  + B(r, N+1-c)
	  + C(N+1-r, N+1-c);
	  end
	end

Table 9: Operations used in speed tests. Matrices A, B, C, D and Q have the size of N×N, where N = 50 for in-cache tests, and N = 500 for out-of-cache tests. For the "decreasing size matrix multiplication" operation, dimensions of A, B, C, D for the in-cache case are 100×80, 80×60, 60×40, 40×20, respectively; for the out-of-cache case, the dimensions are 1000×800, 800×600, 600×400, 400×200, respectively.

Operation	Matlab	Octave	Newmat	IT++
Addition and scalar multiplication	2.9	4.4	2.8	3.9
Transpose, matrix multiplication and in-place addition	1.0	1.3	4.0	1.1
Decreasing size matrix multiplication	1.8	2.1	12.5	2.0
Submatrix copy	3.6	11.5	2.6	10.6
Direct element access	14.7	2592.3	2.5	3.1

Table 10: Speedup factor of Armadillo relative to other software, using in-cache matrices. Tests were performed on an Intel Core2 Duo CPU with 2 Mb cache, running in 64 bit mode at 2 GHz. Each matrix element was stored as a double precision floating point number (8 bytes). A Linux based operating system was used (Fedora 12), incorporating Linux kernel v2.6.32 and the GCC v4.4.4 C++ compiler. Versions of software were as follows: Armadillo 0.9.80, Matlab v7.1.0.183 SP3 64-bit, Octave v3.2.3, Newmat 11 beta, IT++ v4.0.6.

Operation	Matlab	Octave	Newmat	IT++
Addition & scalar multiplication	1.2	3.3	3.7	5.1
Transpose, matrix multiplication & in-place addition	1.3	1.0	9.6	1.1
Decreasing size matrix multiplication	3.3	2.4	29.1	2.4
Submatrix copy	2.0	2.1	2.0	1.6
Direct element access	10.4	2174.3	5.1	2.7

Table 11: As per Table 10, but using out-of-cache matrices.

6 License Choice

There is a myriad of open source software licenses available, ranging from the restrictive and ideologically motivated General Public License (GPL) on one extreme, to the laissez-faire Berkeley Software Distribution (BSD) style licenses on the other extreme. All open source licenses rely on internationally accepted copyright law, providing legally binding conditions on the use and distribution of the source code as well as the binaries (e.g. executable programs) resulting from the compilation of the source code (for example, by a C++ compiler). The full text of many open source licenses is available from the Open Source Initiative website[15].

The GPL in effect states that any software that is directly based on GPL-licensed software will itself be subject to the GPL. In other words, all software directly based on a library such as IT++ becomes "infected" with the GPL. A further requirement of the GPL is that distribution of any GPL-licensed software must, in effect, come with the source code to the software. While this type of licensing is quite appropriate for many projects[16], we prefer to allow the user to choose whether to release the source code for their Armadillo-based programs.

Typically used BSD-style licenses allow the unlimited redistribution of the software and the resulting binaries for any purpose as long as the software's copyright notices and the licenses' disclaimers of warranty are maintained. In contrast the the GPL, BSD licensed code can be embedded within a proprietary program, and there is no requirement to release the source code (i.e. closed source).

In between the two extremes there are licenses such as the Lesser GPL (LGPL), also known as the "Library GPL". The LGPL is an extension of the GPL, relaxing some of the restrictive requirements. The LGPL in effect states that software which only *links* with LGPL covered libraries (at compile- or run-time) can have a different license, including a closed source or proprietary license. However, any changes to the library itself (e.g. bug fixes or extensions) need to be made available under the LGPL license.

We have selected the LGPL to cover the Armadillo library, as we believe it provides a good trade-off between freedom of use and the requirement to share bug fixes and extensions.

7 Development Methodology & Observations

The development of the library is undertaken in the open, with the aid of freely available and globally accessible source code repository provided by SourceForge[17]. Following the "release early and release often" approach advocated by Raymond [11], snapshots are released at regular intervals to obtain bug reports and suggestions from users. This iterative process can be interpreted a form of collaborative innovation [8] and is related to agile software development [6].

Given the complexity of the library and the exponential number of possible execution paths, it is our experience that it's difficult at the outset to design tests that cover all possible usage cases. As such, regular feedback from users helps in tracking down errors. Apart from bug reports, users have also contributed patches to extend the functionality of Armadillo, taking advantage of the availability of the source code. This in turn makes the library attractive to more users (as it satisfies more needs) thereby expanding the number of people effectively testing the library.

We have observed that extensively templated C++ library code has little resemblance to C, Java, or the often used object oriented programming (OOP) subset of C++. As such, heavy template code can be difficult to debug if deliberate precautions are not taken. Specifically, we employed the function signature determination mechanism available in the GCC C++ compiler[18], in order to have a clear view of the input types created through template meta-programming. As we will describe in the following section, the input types can be quite long and convoluted, depending on the mathematical expression being evaluated.

[15] http://www.opensource.org/

[16] For example, the Linux kernel is distributed under the GPL, but in general the conditions of the GPL do not extend to programs which simply run under Linux. As such, it's perfectly legal to run non-GPL (e.g. proprietary) software under Linux.

[17] http://sourceforge.net

[18] http://gcc.gnu.org/

During the early stages of development we have also observed that not all C++ compilers can handle heavy template meta-programming or fully conform to the C++ standard. In particular, MS Visual C++ (especially prior to the 2008 version) can be troublesome. Nevertheless, we made efforts to allow the library to be used with a wide variety of C++ compilers, providing workarounds where necessary.

8 Internal Architecture Overview

The internal architecture of Armadillo is primarily comprised of two base objects (*Base*, *BaseCube*), three object classes used for storing numerical elements (*Mat*, *Cube*, *field*) and four families of supporting classes used for evaluation of mathematical expressions. The first family is for binary operations involving the *Mat* class (*Glue*, *eGlue*, *mtGlue*), while the second family is for unary operations involving the *Mat* class (*Op*, *eOp*, *mtOp*). All classes in the first and second families are derived from the *Base* class. The third and fourth families are analogs of the first and second, with the difference that they are derived from the *BaseCube* class and are designed for operations involving the *Cube* class.

The *Base* and *BaseCube* classes are used for static polymorphism[19], modelled after the "Curiously Recurring Template Pattern" [2, 7]. They are used for type-safe downcasting in functions that restrict their inputs to be classes that are derived from *Base* or *BaseCube*.

The *Mat* class, derived from *Base*, is designed to store elements that are accessed as a 2D array. The storage layout is compatible with the BLAS and LAPACK libraries, i.e., the elements are stored linearly in memory in a column-by-column manner. *Mat* has two derived classes, *Row* and *Col*, which are constrained to be matrices with one row and one column, respectively.

The *Cube* class, derived from *BaseCube*, is designed to store elements that are accessed as a 3D array. The elements are stored in a slice-by-slice manner. Each slice is an instance of the *Mat* class, with the memory for the slices stored in a contiguous manner (i.e. next to each other).

The type of elements that can be stored in an instance of the *Mat* or the *Cube* class is constrained to be one of: *char*, *short*, *int*, *long*, *float*, *double*, *std::complex<float>*, *std::complex<double>* as well as the unsigned versions of the integral types (i.e., *unsigned char*, *unsigned short*, *unsigned int*, *unsigned long*).

The *field* class is designed to store elements of an arbitrary type that are accessed as a 2D array. The elements can be strings, matrices or any C++ class that has the *copy operator* and *operator=* member functions defined. Unlike the *Mat* class, which is optimised for numerical elements, the *field* class is not optimised for any particular element type.

The *Mat*, *Cube* and *field* classes are template classes parameterised by the element type. Several pre-defined typedefs exist, such as *mat* and *fmat* which specify matrices with elements of type *double* and *float*, respectively.

The *Glue* class is used for storing references to two arbitrary objects that are also derived from *Base*. The main use is for binary operations (e.g. multiplication of two matrices). As any object derived from *Base* can be stored, *Glue* can store references to other *Glue* objects, thereby allowing the storage of arbitrarily long mathematical expressions. *Glue* is restricted to store references to objects that have the same underlying element type (e.g., *int*, *float*, *double*, etc).

The *mtGlue* class allows the storage of references to *Base* derived objects that have a different underlying element type. For example, addition of an *int* matrix with a *float* matrix. The *mt* prefix stands for *mixed type*.

The *eGlue* class is similar to *Glue*, with two further constraints: **(i)** it can be used only for a binary operation that results in a matrix that has the same dimensions as the two input objects, and **(ii)** the operation is applied individually element-by-element, using two corresponding elements from the two objects. For example, the binary operation can be matrix addition, but not matrix multiplication. The *e* prefix stands for *element*.

[19] polymorphism without run-time virtual-table lookups.

The *Op*, *mtOp* and *eOp* classes are for unary operations (e.g. matrix transpose) and are analogs of *Glue*, *mtGlue* and *eGlue* classes, respectively. Specifically, the *Op* class is for operations that result in matrices with the same element type as the input object, the *mtOp* class is for operations that produce matrices with an element type that is different to the element type of the input object, and the *eOp* class is for operations that are applied individually element-by-element, resulting in matrices that have the same dimensions and the same element type as the input object.

For the *Glue*, *eGlue*, *Op*, *eOp* and classes, the operation identifiers and types of input objects are stored as template parameters. The *mtGlue* and *mtOp* classes additionally have the output element type specified as a template parameter.

Functions for binary operators (such as $+$ and $-$) are overloaded to accept objects derived from *Base* and *BaseCube* classes. Each binary operator function then produces an appropriate instance of either the *Glue*, *mtGlue* or *eGlue* class. The instance can then be evaluated by the *Mat* class (using the *class constructor* or *operator*= member functions), or used as input to further binary or unary operators. For example, given three matrices A, B, and C, the expression

$$A + B - C$$

is translated by the C++ compiler to an *eGlue* object, defined with the following template type:

eGlue< eGlue<Mat, Mat, eglue_plus>, Mat, eglue_minus>

In a similar manner there are user-accessible functions for unary operations, each producing an appropriate instance of either the *Op*, *mtOp* or *eOp* class. For example, given a matrix X, the expression

inv(square(X))

is translated to an *Op* object defined with the following template type:

Op< eOp<Mat, eop_square>, op_inv>

Binary and unary operations can be combined. For example, given two matrices A and B, the expression

trans(A) * inv(B)

is translated to a *Glue* object defined with the following template type:

Glue< Op<Mat, op_trans>, Op<Mat, op_inv>, glue_times>

Armadillo uses the above template meta-programming as part of a mechanism for implementing a delayed evaluation approach (also known as lazy evaluation [18]) for evaluating mathematical expressions. The approach is capable of combining several operations into one, in order to reduce (or eliminate) the need for compiler-generated temporary objects, as well as to work around structural limitations of the class construct within the current version of the C++ language.

While further details of the meta-programming based approach are beyond the scope of this report[20], we note that all operations involving the *eGlue* and *eOp* classes (e.g. addition, subtraction, element-wise division, element-wise multiplication, division and multiplication by a scalar, square of each element, etc) can be combined. Furthermore, certain operations involving the *Op* class can also be combined. For example, the inverse of a diagonal matrix constructed out of a vector takes into account that only diagonal elements are non-zero.

Armadillo has several other intelligent expression evaluators, including: **(i)** optimisation of the order of matrix multiplication, with a view to reduce the amount of time and memory taken (accomplished by reducing the size of the required temporary matrices); **(ii)** the *as_scalar(expression)* function, which will try to exploit the fact that the result of *expression* is a 1×1 matrix, in order to reduce the amount of computation.

[20] We may expand this report in the future to provide the details.

9 Conclusion

In this report we provided an overview of the open source Armadillo C++ linear algebra library, which we developed with an aim to obtain a good balance between speed and ease of use. It supports integer, floating point and complex numbers, as well as a subset of trigonometric and statistics functions. Various matrix decompositions are provided through optional integration with LAPACK, or one its processor-specific, high-performance drop-in replacements.

C++ template meta-programming is employed as a mechanism for implementing a delayed evaluation approach, capable of combining several operations into one, in order to reduce (or eliminate) the need for temporaries.

The library can be used for converting Matlab (or Octave) based code into production quality software, useful for commercial deployments or algorithm implementation on embedded hardware. The library can also be used for stand-alone prototyping and computationally intensive experimentation, thereby shortening the development process by omitting the Matlab prototyping stage. Performance comparisons suggest that the library is considerably faster than Matlab and Octave, as well as previous C++ libraries such as IT++ and Newmat.

We hope that the library can be useful to other researchers and practitioners. Its open source nature allows customisation, extensions and bug fixing without being tied to any particular vendor or support contract. The library is licensed under the Lesser GPL (LGPL) license, making it useful in the development of both open source and proprietary software.

Acknowledgements

NICTA is funded by the Australian Government as represented by the *Department of Broadband, Communications and the Digital Economy*, as well as the Australian Research Council through the *ICT Centre of Excellence* program.

The author thanks the following persons for their contributions (eg. code, bug reports, suggestions) to Armadillo: Eric R. Anderson, Benoît Bayol, Salim Bcoin, Justin Bedo, Dimitrios Bouzas, Darius Braziunas, Ted Campbell, Shaokang Chen, Clement Creusot, Ian Cullinan, Ryan Curtin, Chris Davey, Dirk Eddelbuettel, Romain Francois, Stanislav Funiak, Piotr Gawron, Charles Gretton, Benjamin Herzog, Edmund Highcock, Kshitij Kulshreshtha, Oka Kurniawan, David Lawrence, Sandra Mau, Carlos Mendes, Artem Novikov, Martin Orlob, Ken Panici, Adam Piatyszek, Jayden Platell, Vikas Reddy, Ola Rinta-Koski, James Sanders, Alexander Scherbatey, Gerhard Schreiber, Shane Stainsby, Petter Strandmark, Paul Torfs, Simon Urbanek, Arnold Wiliem, Yongkang Wong.

References

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.

[2] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001.

[3] M. Castelluccio. Enterprise open source adoption. *Strategic Finance*, 90(5), 2008.

[4] B. Chapman, G. Jost, R. van-der Pas, and D. J. Kuck. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[5] S. J. Chapman. *Essentials of MATLAB Programming*. CL-Engineering, 2nd edition, 2008.

[6] A. Cockburn. *Agile Software Development: The Cooperative Game*. Addison-Wesley, 2nd edition, 2006.

[7] J. O. Coplien. Curiously recurring template patterns. *C++ Report*, 7:24–27, 1995.

[8] M. Dodgson, D. M. Gann, and A. Salter. *The Management of Technological Innovation: Strategy and Practice*. Oxford University Press, 2008.

[9] S. Forge. The rain forest and the rock garden: the economic impacts of open source software. *Info*, 8(3):12–31, 2006.

[10] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley Professional, 3rd edition, 2009.

[11] E. S. Raymond. *The Cathedral & the Bazaar*. O'Reilly Media, 2001.

[12] V. Reddy, C. Sanderson, and B. C. Lovell. Improved anomaly detection in crowded scenes via cell-based analysis of foreground speed, size and texture. In *Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 55–61, 2011. DOI: 10.1109/CVPRW.2011.5981799.

[13] V. Reddy, C. Sanderson, A. Sanin, and B. C. Lovell. Adaptive patch-based background modelling for improved foreground object segmentation and tracking. In *International Conference on Advanced Video and Signal-Based Surveillance (AVSS)*, pages 172–179, 2010. DOI: 10.1109/AVSS.2010.84.

[14] V. Reddy, C. Sanderson, A. Sanin, and B. C. Lovell. MRF-based background initialisation for improved foreground detection in cluttered surveillance videos. In *Lecture Notes in Computer Science (LNCS)*, volume 6494, pages 547–559, 2011. DOI: 10.1007/978-3-642-19318-7_43.

[15] V. O. Safonov. *Trustworthy Compilers*. Wiley, 2010.

[16] C. Sanderson and B. C. Lovell. Multi-region probabilistic histograms for robust and scalable identity inference. In *Lecture Notes in Computer Science (LNCS)*, volume 5558, pages 199–208, 2009. DOI: 10.1007/978-3-642-01793-3_21.

[17] M. Sieverding. Choice in government software procurement: A winning strategy. *Journal of Public Procurement*, 8(1):70–97, 2008.

[18] P. Van-Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.

[19] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2002.

[20] Y. Wong, S. Chen, S. Mau, C. Sanderson, and B. C. Lovell. Patch-based probabilistic image quality assessment for face selection and improved video-based face recognition. In *Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 74–81, 2011. DOI: 10.1109/CVPRW.2011.5981881.