Armadillo: a template-based C++ library for linear algebra

Conrad Sanderson[†] and Ryan Curtin[‡]

[†] Data61, CSIRO, Australia
[‡] Symantec Corporation, USA

The C++ language is often used for implementing functionality that is performance and/or resource sensitive. While the standard C++ library provides many useful algorithms (such as sorting), in its current form it does not provide direct handling of linear algebra (matrix maths).

Armadillo is an open source linear algebra library for the C++ language, aiming towards a good balance between speed and ease of use. Its high-level application programming interface (function syntax) is deliberately similar to the widely used Matlab and Octave languages [4], so that mathematical operations can be expressed in a familiar and natural manner. The library is useful for algorithm development directly in C++, or relatively quick conversion of research code into production environments.

Armadillo provides efficient objects for vectors, matrices and cubes (third order tensors), as well as over 200 associated functions for manipulating data stored in the objects. Integer, floating point and complex numbers are supported, as well as dense and sparse storage formats. Various matrix factorisations are provided through integration with LAPACK [3], or one of its high performance drop-in replacements such as Intel MKL [6] or OpenBLAS [9]. It is also possible to use Armadillo in conjunction with NVBLAS to obtain GPU-accelerated matrix multiplication [7].

Armadillo is used as a base for other open source projects, such as MLPACK, a C++ library for machine learning and pattern recognition [2], and RcppArmadillo, a bridge between the R language and C++ in order to speed up computations [5]. Armadillo internally employs an expression evaluator based on template meta-programming techniques [1], to automatically combine several operations in order to increase speed and efficiency. An overview of the internal architecture is given in [8].

An overview of the available functionality (as of Armadillo version 7.200) is given in Tables 1 through to 8. Table 1 briefly describes the member functions and variables of the main matrix class; Table 2 lists the main subset of overloaded C++ operators; Table 3 outlines matrix decompositions and equation solvers; Table 4 overviews functions for generating matrices; Table 5 lists the main forms of general functions of matrices; Table 6 lists various element-wise functions of matrices; Table 7 summarises the set of functions and classes focused on statistics; Table 8 lists functions specific to signal and image processing.

As one of the aims of Armadillo is to facilitate conversion of code written in Matlab/Octave into C++, Table 9 provides examples of conversion to Armadillo syntax. Figure 1 shows a simple Armadillo based C++ program.

Armadillo can be obtained from:

- http://arma.sourceforge.net

If you use Armadillo in your research and/or software, we would appreciate a citation to this document. Citations are useful for the continued development and maintenance of the library. Please cite as:

- Conrad Sanderson and Ryan Curtin.
  Armadillo: a template-based C++ library for linear algebra.
  Journal of Open Source Software, Vol. 1, pp. 26, 2016.
  http://dx.doi.org/10.21105/joss.00026

Table 1: Subset of member functions and variables of the *mat* class, the main matrix object in Armadillo.

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; use only after debugging
.at(r, c)	as per (r, c), but no bounds check; use only after debugging
.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 size to specified dimensions, without preserving data (fast)
.reshape(rows, cols)	change size to specified dimensions, with elements copied column-wise (slow)
.resize(rows, cols)	change size to specified dimensions, while preserving elements & their layout (slow)
.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 set elements to uniformly distributed random values in [0,1] interval
.randn(rows, cols)	as above, but use a Gaussian/normal distribution with $\mu = 0$ and $\sigma = 1$
.fill(k)	set all elements to be equal to *k*
.for_each([ ](double& val) {...})	for each element, pass its reference to a lambda function (C++11 only)
.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
.is_sorted()	test whether the matrix is sorted
.has_inf()	test whether any element is $\pm\infty$
.has_nan()	test whether any element is not-a-number
.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 text header
.raw_print(header)	as per .print(), but do not change stream settings
.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
.diag(k)	read/write access to *k*-th diagonal
.row(i)	read/write access to row *i*
.col(i)	read/write access to column *i*
.rows(a, b)	read/write access to submatrix, spanning from row *a* to row *b*
.cols(c, d)	read/write access to submatrix, spanning from column *c* to column *d*
.submat(span(a,b), span(c,d))	read/write access to submatrix spanning rows *a* to *b* and columns *c* to *d*
.submat(p, q, size(A))	read/write access to submatrix starting at row *p* and col *q* with size same as matrix *A*
.rows(vector_of_row_indices)	read/write access to rows corresponding to the specified indices
.cols(vector_of_col_indices)	read/write access to columns corresponding to the specified indices
.elem(vector_of_indices)	read/write access to matrix elements corresponding to the specified indices
.each_row()	repeat a vector operation on each row (eg. *A.each_row() += row_vector*)
.each_col()	repeat a vector operation on each column (eg. *A.each_col() += col_vector*)
.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
.min()	return minimum value
.max()	return maximum value
.index_min()	return index of minimum value
.index_max()	return index of maximum value

Table 2: Subset of matrix operations involving overloaded C++ operators.

Operation	Description
A − k	subtract scalar *k* from all elements in matrix *A*
k − A	subtract each element in matrix *A* from scalar *k*
A + k, k + A	add scalar *k* to all elements in matrix *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 matrices *A* and *B*
A / B	element-wise division of matrix *A* by matrix *B*
A == B	element-wise equality evaluation between matrices *A* and *B* [**caveat**: use *approx_equal()* to test whether all corresponding elements are approximately equal]
A != B	element-wise non-equality evaluation between matrices *A* and *B*
A >= B	element-wise evaluation whether elements in matrix *A* are greater-than-or-equal to elements in *B*
A <= B	element-wise evaluation whether elements in matrix *A* are less-than-or-equal to elements in *B*
A > B	element-wise evaluation whether elements in matrix *A* are greater than elements in *B*
A < B	element-wise evaluation whether elements in matrix *A* are less than elements in *B*

Table 3: Subset of functions for matrix decompositions, factorisations, inverses and equation solvers.

Function	Description
chol(X)	Cholesky decomposition of symmetric positive-definite matrix *X*
eig_sym(X)	eigen decomposition of a symmetric/hermitian matrix *X*
eig_gen(X)	eigen decomposition of a general (non-symmetric/non-hermitian) square matrix *X*
eig_pair(A, B)	eigen decomposition for pair of general square matrices *A* and *B*
inv(X)	inverse of a square matrix *X*
inv_sympd(X)	inverse of symmetric positive definite matrix *X*
lu(L, U, P, X)	lower-upper decomposition of *X*, such that *PX = LU* and *X = P'LU*
null(X)	orthonormal basis of the null space of matrix *X*
orth(X)	orthonormal basis of the range space of matrix *X*
pinv(X)	Moore-Penrose pseudo-inverse of a non-square matrix *X*
qr(Q, R, X)	QR decomposition of *X*, such that *QR = X*
qr_econ(Q, R, X)	economical QR decomposition
qz(AA, BB, Q, Z, A, B)	generalised Schur decomposition for pair of general square matrices *A* and *B*
schur(X)	Schur decomposition of 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*
svd_econ(X)	economical singular value decomposition of *X*
syl(X)	Sylvester equation solver

Table 4: Subset of functions for generating matrices and vectors, showing their main form.

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 $\sigma = 1$
randi(rows, cols, distr_param(a,b))	matrix with random integer values in the $[a, b]$ interval
randg(rows, cols, distr_param(a,b))	matrix with random values from a gamma distribution $p(x) = \frac{x^{a-1}\exp(-x/b)}{b^a\Gamma(a)}$
linspace(start, end, n)	vector with *n* elements, linearly spaced from *start* upto (and including) *end*
logspace(A, B, n)	vector with *n* elements, logarithmically spaced from 10^A upto (and including) 10^B
regspace(start, Δ, end)	vector with regularly spaced elements: [start, (start + Δ), (start + 2Δ), ..., (start + MΔ)], where M = floor((end - start)/Δ), so that (start + MΔ) ≤ end

Table 5: Subset of general functions of matrices, showing their main form. For functions with the *dim* argument, *dim = 0* indicates *traverse across rows* (ie. operate on all elements in a column), while *dim = 1* indicates *traverse across columns* (ie. operate on all elements in a row); by default *dim = 0*.

Function	Description
abs(A)	obtain element-wise magnitude of each element of matrix A
accu(A)	accumulate (sum) all elements of matrix A into a scalar
all(A,dim)	return a vector indicating whether all elements in each column or row of A are non-zero
any(A,dim)	return a vector indicating whether any element in each column or row of A is non-zero
approx_equal(A, B, met, tol)	return a bool indicating whether all corresponding elements in A and B are approx. equal
as_scalar(expression)	evaluate an expression that results in a 1×1 matrix, then convert the result to a pure scalar
clamp(A, min, max)	create a copy of matrix A with each element clamped to be between *min* and *max*
cond(A)	condition number of matrix A (the ratio of the largest singular value to the smallest)
conj(C)	complex conjugate of complex matrix C
cross(A, B)	cross product of A and B, assuming they are 3 dimensional vectors
cumprod(A, dim)	cumulative product of elements in each column or row of matrix A
cumsum(A, dim)	cumulative sum of elements in each column or row of matrix A
det(A)	determinant of square matrix A
diagmat(A, k)	interpret matrix A as a diagonal matrix (elements not on k-th diagonal are treated as zero)
diagvec(A, k)	extract the k-th diagonal from matrix A (default: $k = 0$)
diff(A, k, dim)	differences between elements in each column or each row of A; k = number of recursions
dot(A,B)	dot product of A and B, assuming they are vectors with equal number of elements
eps(A)	distance of each element of A to next largest representable floating point number
expmat(A)	matrix exponential of square matrix A
find(A)	find indices of non-zero elements of A; *find(A > k)* finds indices of elements greater than k
fliplr(A)	copy A with the order of the columns reversed
flipud(A)	copy A with the order of the rows reversed
imag(C)	extract the imaginary part of complex matrix C
ind2sub(size(A), index)	convert a linear index (or vector of indices) to subscript notation, using the size of matrix A
inplace_trans(A, method)	in-place / in-situ transpose of matrix A, optionally using a low-memory method
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
kron(A, B)	Kronecker tensor product of A and B
log_det(x, sign, A)	log determinant of square matrix A, such that the determinant is exp(x)*$sign$
logmat(A)	complex matrix logarithm of square matrix A
min(A, dim)	find the minimum in each column or row of matrix A
max(A, dim)	find the maximum in each column or row of matrix A
nonzeros(A)	return a column vector containing the non-zero values of matrix A
norm(A,p)	p-norm of matrix A, with $p = 1, 2, \cdots,$ or p = "-inf", "inf", "fro"
normalise(A, p, dim)	return the normalised version of A, with each column or row normalised to unit p-norm
prod(A, dim)	product of elements in each column or row of matrix A
rank(A)	rank of matrix A
rcond(A)	estimate the reciprocal of the condition number of square matrix A
real(C)	extract the real part of complex matrix C
repmat(A, p, q)	replicate matrix A in a block-like fashion, resulting in p by q blocks of matrix A
reshape(A, r, c)	create matrix with r rows and c columns by copying elements from A column-wise
resize(A, r, c)	create matrix with r rows and c columns by copying elements and their layout from A
shift(A, n, dim)	copy matrix A with the elements shifted by n positions in each column or row
shuffle(A, dim)	copy matrix A with elements shuffled in each column or row
size(A)	obtain the dimensions of matrix A
sort(A, direction, dim)	copy A with elements sorted (in ascending or descending direction) in each column or row
sort_index(A, direction)	generate a vector of indices describing the sorted order of the elements in matrix A
sqrtmat(A)	complex square root of square matrix A
sum(A, dim)	sum of elements in each column or row of matrix A
sub2ind(size(A), row, col)	convert subscript notation *(row,col)* to a linear index, using the size of matrix A
symmatu(A) / symmatl(A)	generate symmetric matrix from square matrix A
strans(C)	simple matrix transpose of complex matrix C, without taking the conjugate
trans(A)	transpose of matrix A (for complex matrices, conjugate is taken); use *A.t()* for shorter form
trace(A)	sum of the elements on the main diagonal of matrix A
trapz(A, B, dim)	trapezoidal integral of B with respect to spacing in A, in each column or row of B
trimatu(A) / trimatl(A)	generate triangular matrix from square matrix A
unique(A)	return the unique elements of A, sorted in ascending order
vectorise(A, dim)	generate a column or row vector from matrix A

Table 6: Element-wise functions: matrix *B* is produced by applying a function to each element of matrix *A*.

Function	Description
exp(A)	base-e exponential: e^x
exp2(A)	base-2 exponential: 2^x
exp10(A)	base-10 exponential: 10^x
trunc_exp(A)	base-e exponential, truncated to avoid ∞
log(A)	natural log: $\log_e(x)$
log2(A)	base-2 log: $\log_2(x)$
log10(A)	base-10 log: $\log_{10}(x)$
trunc_log(A)	natural log, truncated to avoid $\pm\infty$
pow(A, p)	raise to the power of p: x^p
square(A)	square: x^2
sqrt(A)	square root: $\sqrt{x}$
floor(A)	largest integral value that is not greater than the input value
ceil(A)	smallest integral value that is not less than the input value
round(A)	round to nearest integer, with halfway cases rounded away from zero
trunc(A)	round to nearest integer, towards zero
erf(A)	error function
erfc(A)	complementary error function
lgamma(A)	natural log of the gamma function
sign(A)	signum function; for each element *a* in *A*, the corresponding element b in B is: $b = \begin{cases} -1 & \text{if} \quad a < 0 \\ 0 & \text{if} \quad a = 0 \\ +1 & \text{if} \quad a > 0 \end{cases}$
trig(A)	trignometric function, where *trig* is one of: *cos*, *acos*, *cosh*, *acosh*, *sin*, *asin*, *sinh*, *asinh*, *tan*, *atan*, *tanh*, *atanh*

Table 7: Subset of functions for statistics, showing their main form. For functions with the *dim* argument, *dim = 0* indicates *traverse across rows* (ie. operate on all elements in a column), while *dim = 1* indicates *traverse across columns* (ie. operate on all elements in a row); by default *dim = 0*.

Function/Class	Description
cor(A, B)	generate matrix of correlation coefficients between variables in *A* and *B*
cov(A, B)	generate matrix of covariances between variables in *A* and *B*
gmm_diag	class for modelling data as a multi-variate Gaussian Mixture Model (GMM)
hist(A, centers, dim)	generate matrix of histogram counts for each column or row of *A*, using given bin centers
histc(A, edges, dim)	generate matrix of histogram counts for each column or row of *A*, using given bin edges
kmeans(means, A, k, ...)	cluster column vectors in matrix *A* into *k* disjoint sets, storing the set centers in *means*
princomp(A)	principal component analysis of matrix *A*
running_stat	class for running statistics of a continuously sampled one dimensional signal
running_stat_vec	class for running statistics of a continuously sampled multi-dimensional signal
mean(A, dim)	find the mean in each column or row of matrix *A*
median(A, dim)	find the median in each column or row of matrix *A*
stddev(A, norm_type, dim)	find the standard deviation in each column or row of *A*, using specified normalisation
var(A, norm_type, dim)	find the variance in each column or row of matrix *A*, using specified normalisation

Table 8: Subset of functions for signal and image processing, showing their main form.

Function/Class	Description
conv(A, B)	1D convolution of vectors *A* and *B*
conv2(A, B)	2D convolution of matrices *A* and *B*
fft(A, n)	fast Fourier transform of vector *A*, with transform length *n*
ifft(C, n)	inverse fast Fourier transform of complex vector *C*, with transform length *n*
fft2(A, rows, cols)	fast Fourier transform of matrix *A*, with transform size of *rows* and *cols*
ifft2(C, rows, cols)	inverse fast Fourier transform of complex matrix *C*, with transform size of *rows* and *cols*
interp1(X, Y, XI, YI)	given a 1D function specified in vectors *X* (locations) and *Y* (values), generate vector *YI* containing interpolated values at given locations *XI*

Table 9: Examples of Matlab/Octave syntax and conceptually corresponding Armadillo syntax. Note that for submatrix access the exact conversion from Matlab/Octave to Armadillo syntax will require taking into account that indexing starts at 0.

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(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(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)	transpose (for complex matrices the conjugate is taken) simple transpose (for complex matrices the conjugate is not taken)
A = zeros(size(A)) A = ones(size(A)) A = zeros(k) A = ones(k)	A.zeros() A.ones() A = zeros(k,k) A = ones(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 \ B A = A + 1 A = A − 1	A ∗ B 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 }, { 3, 4 } }	requires C++11 compiler
X = [A B] X = [A; B]	X = join_rows(A,B) X = join_cols(A,B)	
A	A.print("A:") or cout << A << endl	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 text are readable by Armadillo (and vice-versa)

A = rand(2,3);
B = randn(4,5);
F = { A; B };
	mat A = randu(2,3);
	mat B = randn(4,5);
	field<mat> F(2,1);
	F(0,0) = A; F(1,2) = B;
		randu generates uniformly distributed random numbers
		
		the field class can store multiple varying size matrices

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

using namespace std;
using namespace arma;

int main(int argc, char** argv)
  {
  mat A(4, 5, fill::randu);
  mat B(4, 5, fill::randu);

  cout << A * B.t() << endl;

  return 0;
  }
```

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

References

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

[2] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray. MLPACK: a scalable C++ machine learning library. *Journal of Machine Learning Research*, 14(Mar):801–805, 2013. http://jmlr.org/papers/v14/curtin13a.html.

[3] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[4] J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring. *GNU Octave version 4.0.0 manual: a high-level interactive language for numerical computations*. 2015. http://www.gnu.org/software/octave/.

[5] D. Eddelbuettel and C. Sanderson. RcppArmadillo: Accelerating R with high-performance C++ linear algebra. *Computational Statistics & Data Analysis*, 71:1054–1063, 2014. http://dx.doi.org/10.1016/j.csda.2013.02.005.

[6] Intel. Math Kernel Library (MKL), 2016. http://software.intel.com/en-us/intel-mkl/.

[7] NVIDIA. NVBLAS Library, 2015. http://docs.nvidia.com/cuda/nvblas/.

[8] C. Sanderson. Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, 2010.

[9] Z. Xianyi, W. Qian, and W. Saar. OpenBLAS: An optimized BLAS library, 2016. http://www.openblas.net/.