Armadillo: An Efficient Framework for Numerical Linear Algebra

Conrad Sanderson [†‡] and Ryan Curtin [⋄]

[†] Data61 / CSIRO, Australia; [‡] Griffith University, Australia; [⋄] NumFOCUS Inc., USA

***Abstract*—A major challenge in the deployment of scientific software solutions is the adaptation of research prototypes to production-grade code. While high-level languages like MATLAB are useful for rapid prototyping, they lack the resource efficiency required for scalable production applications, necessitating translation into lower level languages like C++. Further, for machine learning and signal processing applications, the underlying linear algebra primitives, generally provided by the standard BLAS and LAPACK libraries, are unwieldy and difficult to use, requiring manual memory management and other tedium. To address this challenge, the Armadillo C++ linear algebra library provides an intuitive interface for writing linear algebra expressions that are easily compiled into efficient production-grade implementations. We describe the expression optimisations we have implemented in Armadillo, exploiting template metaprogramming. We demonstrate that these optimisations result in considerable efficiency gains on a variety of benchmark linear algebra expressions.**

***Index Terms*—numerical linear algebra, BLAS, LAPACK, automated mapping, metaprogramming, expression optimisation.**

I. Introduction

Deployment and productisation of various machine learning and signal processing algorithms often requires conversion of research code written in a high-level language (eg., Matlab [1]) into a lower level language such as C or C++, which is considerably more resource efficient [2]. Resource efficiency is an important concern: in datacenter environments, the efficiency of production code is directly connected to cost (power costs and/or cloud resource costs). In environments with constrained computational resources, such as robots, unmanned aerial vehicles and spacecraft, efficiency is especially important as prototype code may be entirely unable to run on the target device due to limited memory or computational power.

Many algorithms inherently rely on numerical linear algebra operations, which are typically provided by the well-tested industry standard BLAS and LAPACK toolkits [3], [4], and their high-performance drop-in substitutes like OpenBLAS [5]. However, converting arbitrary linear algebra expressions into an efficient sequence of well-matched calls to BLAS and LAPACK routines is non-trivial [6], [7]; manual conversion can be laborious and error-prone, and requires good understanding of the intricacies of BLAS and LAPACK, including various trade-offs across available routines and storage formats.

A further downside of directly using BLAS/LAPACK routines is that the resultant source code is quite verbose, has little similarity to the original mathematical expressions, involves keeping track of many supporting variables, and requires manual memory management. Such aspects significantly reduce the readability of the source code, raise the risk of bugs, and increase the maintenance burden [8], [9].

To address the above issues in a coherent framework, we have implemented the Armadillo linear algebra library for C++ [10], which automatically optimises mathematical expressions (both at compile-time and run-time) and efficiently maps them to BLAS/LAPACK routines, all while providing a user-friendly Matlab-like programming interface directly in C++. Armadillo essentially acts as a high-level *domain specific language* [11] built on top of the host C++ language, allowing for resource efficient numerical linear algebra without the many pain points of low-level code. This enables rapid and low risk conversion of research code into production environments, and even permits direct prototyping of algorithms within C++.

As an expository demonstration of the reduced maintenance burden when using Armadillo, consider the matrix expression $\boldsymbol{c} = \boldsymbol{A}^{-1}\boldsymbol{b}$ for matrix $\boldsymbol{A}$ and vectors $\boldsymbol{b}$ and $\boldsymbol{c}$, which represents the solution to a system of linear equations. Using Armadillo, it can be implemented directly in C++ as a single readable and maintainable line of code: `vec c = inv(A) * b`. Naively mapped, the above code will result in subsequent calls to three LAPACK and BLAS functions: `xGETRF`, `xGETRI`, and `xGEMV`. Each of those three functions has between 6 and 11 parameters and may require manual allocation of workspace memory. In addition to hiding the verbosity and associated burdens with calls to BLAS and LAPACK functions, Armadillo is also able to reinterpret the expression and perform a better mapping to more efficient BLAS/LAPACK functions, avoiding the explicit matrix inverse.

Armadillo employs two strategies for automatically optimising mathematical expressions, both aiming to reduce computational effort: **(i)** compile-time fusion of operations to reduce the need for temporary objects, **(ii)** mixture of compile-time detection of expressions and run-time analysis of matrix properties, with the aim of re-ordering and translating operations. Both strategies extensively use C++ template metaprogramming concepts [12], [13], where the compiler is induced to reason at compile-time to generate code tailored for each expression.

We continue the paper as follows. Section II overviews the techniques for compile-time and run-time expression optimisation. Section III provides an empirical evaluation demonstrating the speedups obtained from the optimisations. Section IV overviews the functionality provided by Armadillo. The salient points and avenues for further exploitation are summarised in Section V.

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: An Efficient Framework for Numerical Linear Algebra.
International Conference on Computer and Automation Engineering, 2025.

II. Expression Optimisation via Metaprogramming

Template metaprogramming induces the C++ compiler to run special programs written in a subset of the C++ language. Such metaprograms are executed entirely at compile-time, and can be used to produce compiled code that is specialised for arbitrary object and element types [13].

Rather than directly and immediately evaluating each component of a mathematical expression, Armadillo exploits template metaprogramming via lightweight marker objects that hold references to matrices and data associated with specific operations. The marker objects are generated via user-accessible functions (such as addition and multiplication) and store the identifier of each operation as a custom *type* only visible to the C++ compiler, rather than an explicit value. The marker objects can be chained together, leading to the full description of an arbitrary mathematical expression to be visible to the C++ compiler as an elaborate type, comprised as a tree of operation types. The evaluation of the entire expression is automatically performed when it is assigned to a target matrix. This approach is known as *delayed evaluation* (also known as *lazy evaluation*), and is in contrast to the traditional *eager evaluation* and *greedy evaluation* approaches [14].

As an illustrative example, let us consider the expression

```
Z = 0.4*X + 0.6*Y
```

where `X`, `Y` and `Z` are pre-defined `Mat` objects, each holding a 100×100 matrix. In a traditional eager evaluation approach, the `0.4*X` operation would be evaluated first, storing the intermediate result in a temporary matrix `T1`. The `0.6*Y` operation would then result in a secondary temporary matrix `T2`. The temporary matrices `T1` and `T2` would then be added, finally storing the result in matrix `Z`. This approach for the evaluation of the entire expression is suboptimal and inefficient, as it requires time-consuming memory allocation for the two temporary matrices and three separate loops over the associated matrix elements.

The delayed evaluation approach implemented in Armadillo aims to address such inefficiencies. Through overloading the `*` operator function, the operation `0.4*X` is not evaluated directly, but is instead automatically converted to a lightweight templated marker object named `Op<Mat, op_mul>`, which holds a reference to the `X` object and a copy of the `0.4` scalar multiplier. The nomenclature `Op<...>` indicates that `Op` is a C++ template class, with the items (types) between '<' and '>' specifying template parameters. A similar `Op` marker object is automatically constructed for the `0.6*Y` operation. The `+` operator function is overloaded to accept `Mat` objects and arbitrary marker objects, generating a templated `Glue` marker object that holds references to the given objects. In this example, it chains the two generated `Op` objects, resulting in the `Glue` object having the following type:

```
Glue< Op<Mat, op_mul>, Op<Mat, op_mul>, glue_plus >
```

The expression evaluation mechanism in Armadillo is then automatically invoked through the `=` operator defined in the `Mat` object. The mechanism interprets (at compile-time) the nested types in the template parameters of the given `Glue` object and automatically generates compiled instructions equivalent to:

```
for(int i=0; i<N; ++i) { Z[i] = 0.4*X[i] + 0.6*Y[i]; }
```

where `N` is the number of elements in matrices `X`, `Y` and `Z`, with `X[i]` indicating the i-th element in matrix `X`. Apart from the lightweight `Op` and `Glue` marker objects (which are automatically generated and pre-allocated at compile-time), no other temporary objects are generated. Furthermore, only one loop over the elements is required, instead of three separate loops in the traditional eager evaluation approach.

As a further efficiency enhancement, modern C++ compilers exploit aggressive optimisation strategies that are able to remove lightweight scaffolding objects. This results in the compiler producing machine code where the temporary `Op` and `Glue` objects are optimised away, leaving only code absolutely necessary for the specialised loop, tailored for the given expression. Moreover, this loop can be automatically *vectorised* by the C++ compiler, where low-level Single-Instruction-Multiple-Data (SIMD) instructions are exploited to achieve higher throughput [15].

The expression evaluation mechanisms in Armadillo include safety checks, to ensure that only compatible sizes can be used for each given operation. For example, checking that two matrices to be added or multiplied have conforming dimensions.

For mathematical expressions involving element-wise operations that can be chained, the evaluation mechanism is able to handle an arbitrary number of components (eg., matrices) within the given expressions. Other expressions are handled through detecting specific template patterns, possibly embedded within longer expressions. For example, the expression `inv(A)*b` is translated to the following `Glue` template type:

```
Glue< Op<Mat, op_inv>, Vec, glue_times >
```

The above pattern is detected at compile-time, and is automatically translated as a call to the `xGESV` function in LAPACK, which solves a system of linear equations without the matrix inverse.

In general, expressions with matrix multiplication are typically translated as calls to the `xGEMM` and `xGEMV` functions in BLAS, which are in turn multi-threaded and hand optimised for specific CPU architectures in high-performance implementations such as OpenBLAS [5].

Expression patterns are not necessarily blindly mapped to loops or BLAS/LAPACK functions. Specific patterns are further analysed at run-time, by analysing the properties of the constituent matrices. For example, run-time analysis is used for detecting that in the expression $\boldsymbol{A}\cdot\boldsymbol{A}^T$, the matrix multiplication involves the same matrix and results in a symmetric matrix. Rather than mapping the expression to the `xGEMM` function by default, the more efficient `xSYRK` function can be used, which exploits the symmetry property.

Analysis of matrix properties is also exploited in the evaluation of matrix multiplication chains. For example, in the expression $\boldsymbol{A}\cdot\boldsymbol{B}\cdot\boldsymbol{C}\cdot\boldsymbol{D}$, each of the possible matrix pairs is examined. The pair which results in the smallest matrix is multiplied first, thereby reducing computational effort in subsequent matrix multiplications. As such, it is possible for the entire expression to be evaluated right-to-left (while respecting general non-commutativity of matrix multiplication), rather than the traditional left-to-right order.

III. Empirical Evaluation

To demonstrate some of the optimisations automatically attainable by the expression processing frameworks implemented in Armadillo, we evaluate the following representative set of expressions.

(1). $\boldsymbol{C} = 0.4{\cdot}\boldsymbol{A} + 0.6{\cdot}\boldsymbol{B}$; this is an instance of a compound expression involving element-wise addition of matrices and element-wise multiplication of matrices by scalars. A naive implementation evaluates each component separately, generating temporary matrices for $0.4{\cdot}\boldsymbol{A}$ and $0.6{\cdot}\boldsymbol{B}$, followed by adding the temporary matrices. An optimised implementation is able to bypass the generation of the temporaries, combining scalar multiplication and element addition into one loop that can exploit high-performance SIMD instructions present in modern CPUs [15]. SIMD instructions such as AVX-512 allow efficient processing of chunks of data in one hit instead of individual elements [16].

(2). $\boldsymbol{C} = \boldsymbol{A}_{(:,1)} + \boldsymbol{B}^T_{(2,:)}$; this expression involves element-wise addition of submatrices (accessing individual columns and rows) in conjunction with matrix transpose. The notation $\boldsymbol{A}_{(:,1)}$ denotes the first column of $\boldsymbol{A}$, while $\boldsymbol{B}_{(2,:)}$ denotes the second row of $\boldsymbol{B}$. A naive implementation explicitly extracts the column and row into temporary vectors, followed by applying a transpose operation that generates a further temporary vector, which is then used for element-wise addition. An optimised implementation bypasses the generation of all temporary vectors as well as the explicit transpose operation, and instead accesses the matrix elements directly, performing an implicit transpose where required.

(3). $\boldsymbol{C} = \text{diagmat}(\boldsymbol{A}){\cdot}\boldsymbol{B}$; this expression demonstrates matrix multiplication where one of the matrices is converted to a diagonal matrix. The $\text{diagmat}(\boldsymbol{A})$ function indicates that all elements not on the main diagonal of $\boldsymbol{A}$ are assumed to be zero. In a naive implementation, the $\text{diagmat}(\boldsymbol{A})$ function extracts the diagonal from $\boldsymbol{A}$, and places it a temporary matrix. The temporary matrix (which is assumed by default to be dense) is then multiplied with $\boldsymbol{B}$ through a call to the standard `xGEMM` function in BLAS. An optimised implementation omits generating the temporary, and instead performs a specialised matrix multiplication which exploits sparsity by assuming that only the diagonal elements of $\boldsymbol{A}$ are non-zero.

(4). $\boldsymbol{C} = \text{diagmat}(\boldsymbol{A}{\cdot}\boldsymbol{B})$; in this expression the result of matrix multiplication is converted into a diagonal matrix. A naive implementation would blindly evaluate $\boldsymbol{A}{\cdot}\boldsymbol{B}$ via the `xGEMM` function in BLAS and store the result in a temporary matrix, followed by extracting the diagonal from the temporary and placing it in the final result matrix. An optimised implementation is able to determine that only the diagonal elements of the matrix multiplication are required, thereby omitting unnecessary computations and temporaries.

(5). $k = \text{trace}(\boldsymbol{A}{\cdot}\boldsymbol{B})$; this expression is similar to the preceding $\text{diagmat}(\boldsymbol{A}{\cdot}\boldsymbol{B})$ expression, with the main difference that the diagonal elements of $\boldsymbol{A}{\cdot}\boldsymbol{B}$ are summed into the scalar k. In a naive implementation full matrix multiplication is performed, while an optimised implementation performs a partial matrix multiplication to obtain only the diagonal elements.

(6). $\boldsymbol{E} = \boldsymbol{A}_{m\times m} \cdot \boldsymbol{B}_{m\times \frac{m}{2}} \cdot \boldsymbol{C}_{\frac{m}{2}\times \frac{m}{3}} \cdot \boldsymbol{D}_{\frac{m}{3}\times \frac{m}{4}}$. this is an instance of chained matrix multiplication resulting in a matrix. Here the matrices are progressively decreasing in size. A naive implementation would evaluate each of the matrix products in the standard left-to-right manner, disregarding the wider context of the expression. An optimised implementation can examine the sizes of all possible matrix products within the expression, and determine that evaluating the products in a reversed order will save computational effort.

(7). $k = \boldsymbol{a}^T \cdot \text{diagmat}(\boldsymbol{B}) \cdot \boldsymbol{c}$; this is an example of chained matrix multiplication that results in a scalar value, where $\boldsymbol{a}$ and $\boldsymbol{c}$ are column vectors. A naive implementation computes each component separately (matrix transpose and generation of diagonal matrix) resulting in temporary matrices, and then performs matrix multiplication involving the temporaries. An optimised implementation can examine the expression and determine that only a single and straightforward element-wise multiply-and-sum loop is required over the underlying components, avoiding unnecessary computations and generation of temporaries. This type of expression optimisation is invoked in Armadillo via the `as_scalar()` function.

(8). $\boldsymbol{B} = \boldsymbol{A} \cdot \boldsymbol{A}^T$; this expression is seemingly straightforward, involving a matrix being multiplied with its transposed version, resulting in a symmetric matrix. A naive implementation disregards this fact and blindly calculates the matrix product by treating the two components as separate matrices after an explicit transpose operation. A semi-optimised implementation can avoid the explicit transpose by appropriate mapping to the `xGEMM` function in BLAS. However, a fully optimised implementation can detect that the two matrices to be multiplied are the same, and map the expression to the more efficient `xDSYRK` function in BLAS, which exploits the symmetry aspect and avoids unnecessary computations.

(9). $\boldsymbol{C} = \boldsymbol{A}^{-1} \cdot \boldsymbol{b}$; this expression indicates that a solution to a system of linear equations is *implicitly* sought. A naive implementation ignores the intent of the expression and calculates the inverse of matrix $\boldsymbol{A}$ followed by a matrix multiplication. Calculating the inverse is not only computationally inefficient, but also potentially numerically unstable. An optimised implementation can detect the intent of the expression and map it to the more appropriate `xGESV` function in LAPACK, which finds the solution through a more numerically stable algorithm [3].

(10). $\boldsymbol{C} = \text{solve}(\boldsymbol{A}, \boldsymbol{b})$ where $\boldsymbol{A}$ is a tri-diagonal band matrix; this expression indicates that a solution to a system of linear equations is *explicitly* sought, with $\boldsymbol{A}$ having a special sparse structure. A naive implementation would disregard the structure. An optimised implementation can analyse the matrix and choose a more tailored solver function in LAPACK, thereby exploiting the sparse structure to avoid superfluous computations.

For each of the above expressions, the following multiple matrix sizes are used, ranging from small to large: { 100×100, 250×250, 500×500, 1000×1000 }. The evaluation is done on a machine with an AMD Ryzen 7640U x86-64 CPU running at 3.5 GHz. All source code was compiled with the GCC 14.2 C++ compiler. We also used the open-source OpenBLAS 0.3.26 library which provides optimised implementations of BLAS and LAPACK routines [5].

The results shown in Fig. 1 demonstrate that the optimised handling of expressions in Armadillo leads to considerable reduction in computational effort. Across the considered expressions, the reduction in wall-clock time is often over 50%, and in several cases it is over 90%.

Fig. 2 shows a simple Armadillo-based C++ program to demonstrate its intuitive programming syntax. Fig. 3 lists a trace of corresponding internal function calls, hiding from the user the complexity of calling BLAS and LAPACK functions.

(1) expression: $\boldsymbol{C} = 0.4 \cdot \boldsymbol{A} + 0.6 \cdot \boldsymbol{B}$

matrix size	naive	optimised	reduction
100×100	5.51×10^{-6}	2.26×10^{-6}	59.04%
250×250	4.04×10^{-5}	1.66×10^{-5}	58.89%
500×500	1.87×10^{-4}	6.95×10^{-5}	62.87%
1000×1000	2.45×10^{-3}	7.85×10^{-4}	67.90%

(2) expression: $\boldsymbol{C} = \boldsymbol{A}_{(:,1)} + \boldsymbol{B}^{T}_{(2,:)}$

matrix size	naive	optimised	reduction
100×100	9.52×10^{-8}	3.38×10^{-8}	64.50%
250×250	2.94×10^{-7}	1.05×10^{-7}	64.43%
500×500	7.01×10^{-7}	3.37×10^{-7}	51.94%
1000×1000	1.30×10^{-6}	7.38×10^{-7}	43.07%

(3) expression: $\boldsymbol{C} = \text{diagmat}(\boldsymbol{A}) \cdot \boldsymbol{B}$

matrix size	naive	optimised	reduction
100×100	3.84×10^{-5}	2.80×10^{-6}	92.70%
250×250	6.49×10^{-4}	2.81×10^{-5}	95.67%
500×500	5.01×10^{-3}	2.01×10^{-4}	95.99%
1000×1000	4.14×10^{-2}	1.49×10^{-3}	96.40%

(4) expression: $\boldsymbol{C} = \text{diagmat}(\boldsymbol{A} \cdot \boldsymbol{B})$

matrix size	naive	optimised	reduction
100×100	3.88×10^{-5}	4.86×10^{-6}	87.47%
250×250	6.51×10^{-4}	3.99×10^{-5}	93.87%
500×500	5.02×10^{-3}	1.75×10^{-4}	96.51%
1000×1000	4.11×10^{-2}	1.93×10^{-3}	95.31%

(5) expression: $k = \text{trace}(\boldsymbol{A} \cdot \boldsymbol{B})$

matrix size	naive	optimised	reduction
100×100	3.73×10^{-5}	5.98×10^{-11}	99.99%
250×250	6.42×10^{-4}	6.62×10^{-11}	99.99%
500×500	4.93×10^{-3}	6.71×10^{-11}	99.99%
1000×1000	4.03×10^{-2}	2.71×10^{-10}	99.99%

(6) expression: $\boldsymbol{E} = \boldsymbol{A}_{m \times m} \cdot \boldsymbol{B}_{m \times \frac{m}{2}} \cdot \boldsymbol{C}_{\frac{m}{2} \times \frac{m}{3}} \cdot \boldsymbol{D}_{\frac{m}{3} \times \frac{m}{4}}$

matrix size	naive	optimised	reduction
100×100	1.20×10^{-5}	6.17×10^{-6}	48.53%
250×250	2.05×10^{-4}	1.02×10^{-4}	50.20%
500×500	1.54×10^{-3}	7.94×10^{-4}	48.34%
1000×1000	1.21×10^{-2}	6.02×10^{-3}	50.17%

(7) expression: $k = \boldsymbol{a}^{T} \cdot \text{diagmat}(\boldsymbol{B}) \cdot \boldsymbol{c}$

matrix size	naive	optimised	reduction
100×100	1.85×10^{-6}	7.21×10^{-10}	99.96%
250×250	1.14×10^{-5}	8.54×10^{-10}	99.99%
500×500	4.77×10^{-5}	7.21×10^{-10}	99.99%
1000×1000	1.99×10^{-4}	7.24×10^{-10}	99.99%

(8) expression: $\boldsymbol{B} = \boldsymbol{A} \cdot \boldsymbol{A}^{T}$

matrix size	naive	optimised	reduction
100×100	3.97×10^{-5}	3.35×10^{-5}	15.59%
250×250	6.65×10^{-4}	3.78×10^{-4}	43.19%
500×500	5.07×10^{-3}	2.67×10^{-3}	47.41%
1000×1000	4.32×10^{-2}	2.21×10^{-2}	48.89%

(9) expression: $\boldsymbol{C} = \boldsymbol{A}^{-1} \cdot \boldsymbol{b}$

matrix size	naive	optimised	reduction
100×100	1.47×10^{-4}	5.45×10^{-5}	62.92%
250×250	1.46×10^{-3}	4.69×10^{-4}	67.91%
500×500	8.23×10^{-3}	2.79×10^{-3}	66.16%
1000×1000	5.33×10^{-2}	1.90×10^{-2}	64.34%

(10) expression: $\boldsymbol{C} = \text{solve}(\boldsymbol{A}, \boldsymbol{b})$ where $\boldsymbol{A}$ is a tri-diagonal

matrix size	naive	optimised	reduction
100×100	8.11×10^{-5}	2.16×10^{-5}	73.40%
250×250	6.19×10^{-4}	7.40×10^{-5}	88.04%
500×500	3.31×10^{-3}	2.06×10^{-4}	93.77%
1000×1000	2.13×10^{-2}	1.30×10^{-3}	93.91%

Fig. 1: Comparison of time taken (in seconds) for various matrix expressions, using naive (non-optimised) and automatically optimised implementations within the Armadillo linear algebra library. Average wall-clock time across 1000 runs is reported. Evaluations were performed on an AMD Ryzen 7640U CPU, running at 3.5 GHz. Code was compiled with the GCC 14.2 C++ compiler with the following flags: `-O3 -march=native`. OpenBLAS 0.3.26 was used for optimised implementations of BLAS and LAPACK routines [5].

```
01: #include <armadillo>
02:
03: using namespace arma;
04:
05: int main()
06:   {
07:   // generate random 100x100 matrix
08:   mat A(100, 100, fill::randu);
09:
10:   // generate random 100x1 vector
11:   vec b(100, fill::randu);
12:
13:   // solve for x in random symmetric system AA'x = b
14:   vec x = solve( A * A.t(), b );
15:
16:   x.print("x:");
17:
18:   return 0;
19:   }
```

Fig. 2: A simple Armadillo-based C++ program, solving a random symmetric system of linear equations.

```
Op<T1, op_type>::Op(T1&) [T1 = Mat; op_type = op_htrans]
operator*(T1&, T2&) [T1 = Mat; T2 = Op<Mat,op_htrans>]
Glue<T1, T2, glue_type>::Glue(T1&, T2&) [T1 = Mat; T2 = Op<Mat,op_htrans>; glue_type = glue_times]
solve(Base<double, T1>&, Base<double, T2>&)
Glue<T1, T2, glue_type>::Glue(T1&, T2&) [... glue_type = glue_solve_gen_def]
Col::Col(Base<double,T1>&) [T1 = Glue<Glue<Mat,Op<Mat,op_htrans>,glue_times>,Mat,glue_solve_gen_def>]
Mat::operator=(Glue<T1, T2, glue_type>&) [... glue_type = glue_solve_gen_def]
glue_solve_gen_def::apply(Mat&, Glue<T1, T2, glue_solve_gen_def>&)
glue_solve_gen_full::apply(Mat&, Base<double, T1>&, Base<double, T2>&, uword)
Mat::Mat(Glue<T1, T2, glue_type>&) [T1 = Mat; T2 = Op<Mat,op_htrans>; glue_type = glue_times]
glue_times::apply(Mat&, Glue<T1, T2, glue_times>&) [T1 = Mat; T2 = Op<Mat,op_htrans>]
glue_times_redirect<2>::apply(Mat&, Glue<T1, T2, glue_times>&) [T1 = Mat; T2 = Op<Mat,op_htrans>]
glue_times::apply(Mat&, TA&, TB&, double) [trans_A = false; trans_B = true; TA = Mat; TB = Mat]
Mat::set_size(uword, uword) [uword = long long unsigned int] [in_n_rows: 100; in_n_cols: 100]
Mat::init(): acquiring memory
blas::syrk(...)
glue_solve_gen_full::apply(): detected square system
band_helper::is_band(uword&, uword&, Mat&, uword) [uword = long long unsigned int]
trimat_helper::is_triu(Mat&)
trimat_helper::is_tril(Mat&)
glue_solve_gen_full::apply(): rcond + sym
auxlib::solve_sym_rcond(Mat&, double&, Mat&, Base<double, T1>&) [T1 = Mat; ...]
Mat::operator=(Mat&) [this: e0a67920; in_mat: e0a67860]
Mat::init_warm(uword, uword) [uword = long long unsigned int] [in_n_rows: 100; in_n_cols: 1]
Mat::init(): acquiring memory
lapack::lansy(...)
lapack::sytrf(...)
lapack::sytrs(...)
lapack::sycon(...)
Mat::destructor: releasing memory
```

Fig. 3: An abridged trace of internal function calls and debugging messages resulting from line 14 in Fig. 2, containing the expression `vec x = solve( A * A.t(), b )`.

IV. Functionality

The full documentation of the classes and functions provided by Armadillo is available at https://arma.sourceforge.net/docs.html. An overview of the functionality is given in Tables I through to IX. Table I provides examples of Matlab syntax and corresponding Armadillo syntax. Table II briefly describes the member functions and variables of the *mat* class, the main matrix class in Armadillo. Table III lists the main subset of overloaded C++ operators. Table IV overviews functions for generating matrices. Table V lists various element-wise functions of matrices. Table VI lists the main forms of general functions of matrices. Table VII outlines matrix decompositions, inverses, and equation solvers. Table VIII summarises the set of functions and classes focused on statistics. Table IX lists functions specific to signal and image processing. In each table, only the main form of each function is shown; refer to the online documentation to see all available functions and their corresponding forms.

Table I: Examples of Matlab syntax and conceptually corresponding Armadillo syntax. For submatrix access the exact conversion from Matlab to Armadillo syntax requires taking into account that indexing starts at 0.

Matlab	Armadillo	Notes
`A = zeros(k)`	`mat A = zeros(k,k)`	generate square matrix with all elements set to zero
`A = ones(k)`	`mat A = ones(k,k)`	generate square matrix with all elements set to one
`A = zeros(size(A))`	`A.zeros()`	set all elements to zero
`A = ones(size(A))`	`A.ones()`	set all elements to one
`A = rand(4,5)`	`mat A = randu(4,5)`	generate matrix with random numbers drawn from uniform distribution
`B = randn(4,5)`	`mat B = randn(4,5)`	generate matrix with random numbers drawn from normal distribution
`C = complex(A,B)`	`cx_mat C = cx_mat(A,B)`	construct complex matrix out of two real matrices
`A = [ 1, 2; 3, 4; ]`	`mat A = { {1, 2}, {3, 4} }`	generate matrix by directly specifying values
`A(1, 1) = k`	`A(0, 0) = k`	indexing in Armadillo starts at 0, following C++ convention
`A`	`A.print("A:")`	print the contents of a matrix to the standard output (std::cout)
`size(A,1)`	`A.n_rows`	member variables are read only
`size(A,2)`	`A.n_cols`	
`numel(A)`	`A.n_elem`	.n_elem indicates the total number of elements
`A(:, k)`	`A.col(k)`	read/write access to specific column
`A(k, :)`	`A.row(k)`	read/write access to specific row
`A(:, p:q)`	`A.cols(p, q)`	read/write access to submatrix spanning the specified columns
`A(p:q, :)`	`A.rows(p, q)`	read/write access to submatrix spanning the specified rows
`A(p:q, r:s)`	`A( span(p, q), span(r, s) )`	A(span(first_row, last_row), span(first_col, last_col))
`A'`	`A.t() or trans(A)`	transpose (for complex matrices the conjugate is taken)
`A.'`	`A.st() or strans(A)`	simple transpose (for complex matrices the conjugate is not taken)
`A * B`	`A * B`	* indicates matrix multiplication
`A .* B`	`A % B`	% indicates element-wise multiplication
`A ./ B`	`A / B`	/ indicates element-wise division
`A \ B`	`solve(A,B)`	solve system of linear equations
`X = A(:)`	`X = vectorise(A)`	flatten matrix into column vector
`X = [ A B ]`	`X = join_rows(A,B)`	
`X = [ A; B ]`	`X = join_cols(A,B)`	
`csvwrite('A.csv', A)`	`A.save("A.txt", csv_ascii)`	store data in comma-separated-value (CSV) format
`A = csvread('A.csv')`	`A.load("A.txt", csv_ascii)`	

Table II: 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 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; **caveat:** use with caution
`.reset()`	set the number of elements to zero
`.set_size(n_rows, n_cols)`	change size to given dimensions, without preserving data (fast)
`.reshape(n_rows, n_cols)`	change size to given dimensions, with elements copied column-wise (slow)
`.resize(n_rows, n_cols)`	change size to given dimensions, while preserving elements and their layout (slow)
`.zeros(n_rows, n_cols)`	set all elements to zero, optionally first resizing to given dimensions
`.ones(n_rows, n_cols)`	as above, but set all elements to one
`.randu(n_rows, n_cols)`	as above, but set elements to uniformly distributed random values in [0,1] interval
`.randn(n_rows, n_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
`.replace(old_val, new_val)`	replace all instances of *old_val* with *new_val*
`.for_each( [](double& val) {...} )`	for each element, pass its reference to the specified lambda function
`.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
`.is_symmetric(tolerance)`	test whether the matrix is symmetric within optionally specified tolerance
`.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_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
`.brief_print(header)`	print a shortened/abridged version of the matrix, with optional text header
`.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 (default: $k = 0$)
`.col(i)`	read/write access to column i
`.row(i)`	read/write access to row i
`.cols(a, b)`	read/write access to submatrix, spanning from column a to column b
`.rows(a, b)`	read/write access to submatrix, spanning from row a to row b
`.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
`.cols( vector_of_col_indices )`	read/write access to columns corresponding to the specified indices
`.rows( vector_of_row_indices )`	read/write access to rows corresponding to the specified indices
`.elem( vector_of_indices )`	read/write access to matrix elements corresponding to the specified indices
`.each_col()`	repeat a vector operation on each column (eg. *A.each_col() += col_vector*)
`.each_row()`	repeat a vector operation on each row (eg. *A.each_row() += row_vector*)
`.swap_cols(p, q)`	swap contents of specified columns
`.swap_rows(p, q)`	swap contents of specified rows
`.insert_cols(col, X)`	insert copy of X at specified column
`.insert_rows(row, X)`	insert copy of X at specified row
`.shed_cols(first_col, last_col)`	remove the specified range of columns
`.shed_rows(first_row, last_row)`	remove the specified range of rows
`.min()`	return minimum value
`.max()`	return maximum value
`.index_min()`	return index of minimum value
`.index_max()`	return index of maximum value
`.t()`	return transposed version of the matrix; for complex matrices, conjugate is taken
`.st()`	return transposed version of the matrix, without taking the conjugate
`.as_col()`	return flattened version of the matrix as a column vector, via concatenating all columns
`.as_row()`	return flattened version of the matrix as a row vector, via concatenating all rows

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

Function	Description
`eye(n_rows, n_cols)`	matrix with elements on main diagonal set to one (identity matrix for n_rows = n_cols)
`ones(n_rows, n_cols)`	matrix with all elements set to one
`zeros(n_rows, n_cols)`	matrix with all elements set to zero
`randu(n_rows, n_cols, distr_param(a,b))`	matrix with uniformly distributed random values in $[a, b]$ interval; default: a=0, b=1
`randn(n_rows, n_cols, distr_param(`μ`,`σ`))`	matrix with random values from a normal distribution; default: μ=0, σ=1
`randi(n_rows, n_cols, distr_param(a,b))`	matrix with random integer values in $[a, b]$ interval
`randg(n_rows, n_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 + $M\Delta$)], where M = floor((end - start)/Δ), so that (start + $M\Delta$) $\leq$ end

Table V: Element-wise functions: produce a matrix 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 $\pm\infty$
`square(A)`	square: x^2
`sqrt(A)`	square root: $\sqrt{x}$
`pow(A, p)`	raise to the power of p: x^p
`abs(A)`	magnitude of each element: $\lvert x\rvert$
`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
`tgamma(A)`	gamma function
`lgamma(A)`	natural log of the absolute value of gamma function
`sign(A)`	signum function; for each element a in A, the corresponding element b in the produced matrix is: $b = \begin{cases} -1 & \text{if } a < 0 \\ 0 & \text{if } a = 0 \\ +1 & \text{if } a > 0 \end{cases}$
`trig(A)`	trigonometric function, where *trig* is one of: *cos*, *acos*, *cosh*, *acosh*, *sin*, *asin*, *sinh*, *asinh*, *tan*, *atan*, *tanh*, *atanh*

Table VI: Subset of general functions of matrices, showing their main form. For functions with the *dim* argument, *dim = 0* indicates to process each column, while *dim = 1* indicates to process each row); by default *dim = 0*.

Function	Description
`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 approximately 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 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: $k = 0$)
`diags(V, D, n_rows, n_cols)`	generate matrix with diagonals specified by vector D copied from corresponding vectors in matrix V
`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
`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
`imag(C) / real(C)`	extract the imaginary / real 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
`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"
`norm2est(A)`	fast estimate of the 2-norm (spectral norm) of matrix A
`normalise(A, p, dim)`	return the normalised version of A, with each column or row normalised to unit p-norm
`powmat(A, n)`	raise square matrix A to the power of n
`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
`repelem(A, nr, nc)`	generate matrix by replicating each element of matrix A, with nr and nc copies per row and column
`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
`reverse(A, dim)`	generate copy of matrix A with the order of elements reversed in each column or row
`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
`trace(A)`	sum of the elements on the main diagonal of matrix A
`trans(A)`	transpose of matrix A (for complex matrices, conjugate is taken); use *A.t()* for shorter form
`strans(C)`	simple matrix transpose of complex matrix C, without taking the conjugate
`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 upper/lower triangular matrix from square matrix A
`unique(A)`	return the unique elements of A, sorted in ascending order
`vecnorm(A, p, dim)`	compute the p-norm of each column or row vector in matrix A
`vectorise(A, dim)`	generate flattened version of matrix A

Table VII: Subset of functions for matrix decompositions, factorisations, inverses and equation solvers, showing their main form.

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 (generalised inverse) of 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 VIII: Subset of functions for statistics, showing their main form. For functions with the *dim* argument, $dim = 0$ indicates to process each column, while $dim = 1$ indicates to process each 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` / `gmm_full`	classes for modelling data as multivariate Gaussian mixture models, using either diagonal or full covariance matrices
`kmeans(means, A, k, ...)`	cluster column vectors in matrix A into k disjoint sets, storing the set centers in *means*
`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
`quantile(A, P, dim)`	for each row or column vector of matrix A, calculate the quantiles corresponding to the cumulative probability values in the given P vector
`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
`normpdf(A, M, S)`	for each scalar in A, compute its probability density function according to a Gaussian distribution using the corresponding mean in M and the corresponding standard deviation in S
`normcdf(A, M, S)`	as per normpdf(), but compute the cumulative distribution function
`mvnrnd(M, C, N)`	generate a matrix with N random column vectors from a multivariate Gaussian distribution with mean M and covariance matrix C
`wishrnd(S, df)`	generate a random matrix sampled from the Wishart distribution with parameters S and *df*, where S is a symmetric positive definite matrix and *df* specifies the degrees of freedom

Table IX: 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
`fft2(A, n_rows, n_cols)`	fast Fourier transform of matrix A, with transform size of *n_rows* and *n_cols*
`ifft(C, n)`	inverse fast Fourier transform of complex vector C, with transform length n
`ifft2(C, n_rows, n_cols)`	inverse fast Fourier transform of complex matrix C, with transform size of *n_rows* and *n_cols*
`interp1(X, Y, XI, YI)`	given a 1D function specified in vector X (locations) and vector Y (values), generate vector *YI* containing interpolated values at given locations *XI*
`interp2(X, Y, Z, XI, YI, ZI)`	given a 2D function specified by matrix Z with coordinates given by vectors X and Y, generate matrix *ZI* which contains interpolated values at the coordinates given by vectors *XI* and *YI*

V. Conclusion

Armadillo facilitates easy and maintainable representation of arbitrary linear algebra expressions in C++ that are efficiently mapped to underlying BLAS and LAPACK operations. Users do not need to worry about cumbersome manual memory management or complicated calls to BLAS and LAPACK subroutines. There is virtually no performance penalty for the abstractions provided by Armadillo. Moreover, through under-the-hood template metaprogramming and automatic optimisations of expressions, Armadillo can achieve considerable reductions in processing time over direct and/or naive implementations.

Work on Armadillo started in 2008. Over the years the library has been iteratively and collaboratively developed with feedback from the wider scientific and engineering communities. The library provides over 200 functions; in addition to elementary operations, there are functions for statistics, signal processing, non-contiguous submatrix views, and various matrix factorisations. The library is currently comprised of about 135,000 lines of templated code, excluding BLAS and LAPACK routines. Support is provided for matrices with single- and double-precision floating point elements (in both real and complex forms), as well as integer elements. Dense and sparse storage formats are supported.

Armadillo is now in a mature state and in wide production use. For example, Armadillo has been successfully used to accelerate computations in open-source projects such as the *ensmallen* library for numerical optimisation [17] and the *mlpack* library for machine learning [18], which provide production-ready applications for a variety of environments, including low-resource devices such as small microcontrollers. Armadillo has also been used for accelerating over 1000 packages for the R statistical environment [19].

Armadillo can be obtained from `https://arma.sourceforge.net`, with the source code provided under the permissive Apache 2.0 license [20], [21], which allows unencumbered use in commercial products. Armadillo is also included as part of all major Linux distributions.

In future work we plan to extend Armadillo to include the support for half-precision floating point and 'brain floating point' (BF16) element types [22], as well as to bring the same kinds of expression optimisations to GPU-based linear algebra via the companion Bandicoot library [23].

Acknowledgements

The authors would like to thank the wider open-source community as well as all bug reporters and contributors to Armadillo; without them this work would not have been possible. Ryan Curtin's contributions are based on work supported by the National Aeronautics and Space Administration (NASA) under the ROSES-23 HPOSS program, grant no. 80NSSC24K1524.

References

[1] D. J. Higham and N. J. Higham, *MATLAB Guide*, 3rd ed. SIAM, 2017.

[2] B. Stroustrup, *Programming: Principles and Practice Using C++*, 3rd ed. Addison-Wesley, 2024.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. SIAM, 1999.

[4] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Trans. Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[5] Z. Xianyi and M. Kroeker, "OpenBLAS: An optimized BLAS library," 2025, `http://www.openmathlib.org/OpenBLAS/`.

[6] D. Berényi, A. Leitereg, and G. Lehel, "Towards scalable pattern-based optimization for dense linear algebra," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 22, 2018.

[7] C. Psarras, H. Barthels, and P. Bientinesi, "The linear algebra mapping problem: Current state of linear algebra languages and libraries," *ACM Transactions on Mathematical Software*, vol. 48, no. 3, 2022.

[8] H. M. Sneed, "A cost model for software maintenance & evolution," in *IEEE International Conference on Software Maintenance*, 2004.

[9] R. Malhotra and A. Chug, "Software maintainability: Systematic literature review and current trends," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 8, pp. 1221–1253, 2016.

[10] C. Sanderson and R. Curtin, "Armadillo: a template-based C++ library for linear algebra," *Journal of Open Source Software*, vol. 1, no. 2, p. 26, 2016.

[11] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[12] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen, "Generative programming and active libraries," in *Lecture Notes in Computer Science (LNCS)*, vol. 1766, 2000, pp. 25–39.

[13] D. Vandevoorde, N. Josuttis, and D. Gregor, *C++ Templates: The Complete Guide*, 2nd ed. Addison-Wesley, 2017.

[14] D. A. Watt, *Programming Language Design Concepts*. Wiley, 2004.

[15] K. Stock, L.-N. Pouchet, and P. Sadayappan, "Using machine learning to improve automatic vectorization," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, 2012.

[16] J. M. Cebrian, L. Natvig, and M. Jahre, "Scalability analysis of AVX-512 extensions," *The Journal of Supercomputing*, vol. 76, no. 3, pp. 2082–2097, 2020.

[17] R. R. Curtin, M. Edel, R. G. Prabhu, S. Basak, Z. Lou, and C. Sanderson, "The ensmallen library for flexible numerical optimization," *Journal of Machine Learning Research*, vol. 22, no. 166, 2021.

[18] R. R. Curtin, M. Edel, O. Shrit *et al.*, "mlpack 4: a fast, header-only C++ machine learning library," *Journal of Open Source Software*, vol. 8, no. 82, 2023.

[19] D. Eddelbuettel and C. Sanderson, "RcppArmadillo: Accelerating R with high-performance C++ linear algebra," *Computational Statistics and Data Analysis*, vol. 71, pp. 1054–1063, 2014.

[20] A. St. Laurent, *Understanding Open Source and Free Software Licensing*. O'Reilly Media, 2004.

[21] X. Li, Y. Zhang, C. Osborne *et al.*, "Systematic literature review of commercial participation in open source software," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, 2025.

[22] G. Henry, P. T. P. Tang, and A. Heinecke, "Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations," in *IEEE Symposium on Computer Arithmetic (ARITH)*, 2019, pp. 69–76.

[23] R. R. Curtin, M. Edel, and C. Sanderson, "Bandicoot: C++ library for GPU linear algebra and scientific computing," *arXiv:2308.03120*, 2023.