*Article*

# Practical Sparse Matrices in C++ with Hybrid Storage and Template-Based Expression Optimisation

**Conrad Sanderson [1,2,4,*] and Ryan Curtin [3,4]**

[1] Data61 / CSIRO, Australia
[2] University of Queensland, Australia
[3] RelationalAI, USA
[4] Arroyo Consortium
* Correspondence: conradsand@ieee.org

† This paper is an extended version of our paper published in International Congress on Mathematical Software 2018.

Version 6th July 2019 submitted to Math. Comput. Appl.

**Abstract:** Despite the importance of sparse matrices in numerous fields of science, software implementations remain difficult to use for non-expert users, generally requiring the understanding of underlying details of the chosen sparse matrix storage format. In addition, to achieve good performance, several formats may need to be used in one program, requiring explicit selection and conversion between the formats. This can be both tedious and error-prone, especially for non-expert users. Motivated by these issues, we present a user-friendly and open-source sparse matrix class for the C++ language, with a high-level application programming interface deliberately similar to the widely used MATLAB language. This facilitates prototyping directly in C++ and aids the conversion of research code into production environments. The class internally uses two main approaches to achieve efficient execution: (i) a hybrid storage framework, which automatically and seamlessly switches between three underlying storage formats (compressed sparse column, Red-Black tree, coordinate list) depending on which format is best suited and/or available for specific operations, and (ii) a template-based meta-programming framework to automatically detect and optimise execution of common expression patterns. Empirical evaluations on large sparse matrices with various densities of non-zero elements demonstrate the advantages of the hybrid storage framework and the expression optimisation mechanism.

**Keywords:** mathematical software; C++ language; sparse matrix; numerical linear algebra

**MSC:** 68N99; 65Y04; 65Y15; 65F50

---

## 1. Introduction

Recent decades have seen the frontiers of scientific computing increasingly push towards the use of larger and larger datasets. In fact, frequently the data to be represented is so large that it cannot fully fit into working memory. Fortunately, in many cases the data has many zeros and can be represented in a compact manner, allowing users to work with sparse matrices of extreme size with few non-zero elements. However, converting code from using dense matrices to using sparse matrices—a common task when scaling code to larger data—is not always straightforward.

Current open-source frameworks may provide several separate sparse matrix classes, each with its own data storage format. For example, SciPy [1] has 7 sparse matrix classes, where each storage format is best suited for efficient execution of a specific set of operations (eg., incremental matrix construction vs. matrix multiplication). Other frameworks may provide only one sparse matrix class, with severe runtime penalties if it is not used in the right way. This can be challenging and bewildering for users who simply want to create and use sparse matrices, and do not have the time, expertise, or desire to understand the advantages and disadvantages of each format. To achieve good performance, several formats may need to be used in one program, requiring explicit selection and conversion between the formats. This multitude of sparse matrix classes complicates the programming task, adds to the maintenance burden, and increases the likelihood of bugs.

Driven by the above concerns, we have devised a practical and user-friendly sparse matrix class for the C++ language [2]. The sparse matrix class uses a hybrid storage framework, which *automatically* and *seamlessly* switches between three data storage formats, depending on which format is best suited and/or available for specific operations:

- Compressed Sparse Column (CSC), used for efficient and nuanced implementation of core arithmetic operations such as matrix multiplication and addition, as well as efficient reading of individual elements;
- Red-Black Tree (RBT), used for both robust and efficient incremental construction of sparse matrices (i.e., construction via setting individual elements one-by-one, not necessarily in order);
- Coordinate List (COO), used for low-maintenance and straightforward implementation of relatively complex and/or lesser-used sparse matrix functionality.

The COO format is important to point out, as the source code for the sparse matrix class is distributed and maintained as part of the open-source Armadillo library [3]. Due to its simpler nature, the COO format facilitates functionality contributions from time-constrained and/or non-expert users, as well as reducing maintenance and debugging overhead for the library maintainers.

While there are many other sparse matrix implementations in existence, to our knowledge the presented approach is the first to offer a unified interface with automatic format switching under the hood. Most toolkits are limited to either a single format or multiple formats the user must manually convert between. The comprehensive SPARSKIT package [4] contains 16, and SciPy contains seven formats [1]. In these toolkits the user must manually convert between formats. On the other hand, both MATLAB and GNU Octave [5] contain sparse matrix implementations, but they supply only the CSC format [6], meaning that users must write their code in special ways to ensure its efficiency [7]. This is a similar situation to the Blaze library (bitbucket.org/blaze-lib/blaze) [8], which implements only a CSR/CSC format sparse matrix. Users are explicitly discouraged from individual element insertions and, for efficiency, must construct their sparse matrices in the restricted environment of batch insertion. The Eigen C++ matrix library (eigen.tuxfamily.org) uses a specialised sparse matrix format which has deliberate redundancy and overprovisioned storage. While this can help with reducing the computational effort of element insertion in some situations, it requires manual care to maintain storage efficiency. Furthermore, as the cost of random insertion of elements is still high, the associated documentation recommends to manually construct a COO-like representation of all the elements, from which the actual sparse matrix is then constructed. The IT++ library (itpp.sourceforge.net) has a cumbersome sparse matrix class with a custom format that also employs overprovisioned storage. The format is less efficient storage-wise than CSC unless explicit manual care is taken. Data is stored in unordered fashion which allows for faster element insertion than CSC, but at the cost of reduced performance for linear algebra operations. Thus, overall, the landscape of sparse matrix implementations is composed of libraries where a user must be aware of some of the internal storage details of these implementations in order to produce efficient code; this is not ideal.

To make the situation even more complex, there are also numerous other sparse matrix formats [4, 9]. Examples are the modified compressed row/column format (intended for sparse matrices with all non-zero elements on the diagonal), block compressed storage format (intended for sparse matrices with dense submatrices), compressed diagonal format (intended for straightforward storage of banded sparse matrices under the assumption of constant bandwith), and the skyline format (intended for more efficient storage of banded sparse matrices with irregular bandwith). As these formats are focused on specialised use cases, their utility is typically not very general. Thus we have currently opted against including these formats in our hybrid framework, though it would be relatively easy to accomodate more formats in the future.

To further promote efficient execution, the sparse matrix class internally implements a delayed evaluation framework [10] based on template meta-programming [11,12] combined with operator overloading [2]. In delayed evaluation, the evaluation of a given compound mathematical expression is delayed until its value is required (ie., assigned to a variable). This is in contrast to eager evaluation (also known as strict evaluation), where each component of a compound expression is evaluated immediately. As such, the delayed evaluation framework allows automatic compile-time analysis of compound expressions, which in turns allows for automatic detection and optimisation of common expression patterns. For example, several operations can be combined to reduce the required computational effort.

Overall, the sparse matrix class and its associated functions provide a high-level application programming interface (function syntax) that is intuitive, close to a typical dense matrix interface, and deliberately similar to MATLAB. This can help with rapid transition of dense-specific code to sparse-specific code, facilitates prototyping directly in C++, and aids the conversion of research code into production environments.

The paper is continued as follows. In Section 2 we overview the functionality provided by the sparse matrix class and its associated functions. The delayed evaluation approach is overviewed in Section 3. In Section 4 we describe the underlying storage formats used by the class, and the scenarios that each of the formats is best suited for. In Section 5 we discuss the costs for switching between the formats. Section 6 provides an empirical evaluation showing the advantages of the hybrid storage framework and the delayed evaluation approach. The salient points and avenues for further exploitation are summarised in Section 7. This article is a thoroughly revised and extended version of our earlier work [13].

## 2. Functionality

The sparse matrix class and its associated functions provide a user-friendly suite of essential sparse linear algebra functionality, including fundamental operations such as addition, matrix multiplication and submatrix manipulation. The class supports storing elements as integers, single- and double-precision floating point numbers, as well as complex numbers. Various sparse eigendecompositions and linear equation solvers are provided through integration with low-level routines in the de-facto standard ARPACK [14] and SuperLU libraries [15]. The resultant high-level functions automatically take care of tedious and cumbersome details such as memory management, allowing the user to concentrate their programming effort on mathematical details.

C++ language features such as overloading of operators (eg., $*$ and +) [2] are exploited to allow mathematical operations with matrices to be expressed in a concise and easy-to-read manner, in a similar fashion to the proprietary MATLAB language. For example, given sparse matrices `A`, `B`, and `C`, a mathematical expression such as

$$\mathtt{D} = \tfrac{1}{2}(\mathtt{A} + \mathtt{B}) \cdot \mathtt{C}^{\mathtt{T}}$$

can be written directly in C++ as

```
sp_mat D = 0.5 * (A + B) * C.t();
```

where `sp_mat` is our sparse matrix class. Figure 1 contains a complete C++ program which briefly demonstrates usage of the sparse matrix class, while Table 1 lists a subset of the available functionality.

The aggregate of the sparse matrix class, operator overloading and associated functions on sparse matrices is an instance of a Domain Specific Language (sparse linear algebra in this case) embedded within the host C++ language [16,17]. This allows complex algorithms relying on sparse matrices to be easily developed and integrated within a larger C++ program, making the sparse matrix class directly useful in application/product development.

| **Function** | **Description** |
|---|---|
| `sp_mat X(1000,2000)` | Declare sparse matrix with 1000 rows and 2000 columns |
| `sp_cx_mat X(1000,2000)` | As above, but use complex elements |
| `X(1,2) = 3` | Assign value 3 to element at location (1,2) of matrix $X$ |
| `X = 4.56 * A` | Multiply matrix $A$ by scalar |
| `X = A + B` | Add matrices $A$ and $B$ |
| `X = A * B` | Multiply matrices $A$ and $B$ |
| `X( span(1,2), span(3,4) )` | Provide read/write access to submatrix of $X$ |
| `X.diag(k)` | Provide read/write access to diagonal $k$ of $X$ |
| `X.print()` | Print matrix $X$ to terminal |
| `X.save(filename, format)` | Store matrix $X$ as a file |
| `speye(rows, cols)` | Generate sparse matrix with values on diagonal set to one |
| `sprandu(rows, cols, density)` | Generate sparse matrix with random non-zero elements |
| `sum(X, dim)` | Sum of elements in each column (*dim=0*) or row (*dim=1*) |
| `min(X, dim); max(X, dim)` | Obtain extremum value in each column (*dim=0*) or row (*dim=1*) |
| `X.t()` or `trans(X)` | Return transpose of matrix $X$ |
| `kron(A, B)` | Kronecker tensor product of matrices $A$ and $B$ |
| `repmat(X, rows, cols)` | Replicate matrix $X$ in block-like fashion |
| `norm(X, p)` | Compute $p$-norm of vector or matrix $X$ |
| `normalise(X, p, dim)` | Normalise each column (*dim=0*) or row (*dim=1*) to unit $p$-norm |
| `trace(A.t() * B)` | Compute trace of $A^T B$ **without** explicit transpose and multiplication |
| `diagmat(A + B)` | Obtain diagonal matrix from $A + B$ **without** full matrix addition |
| `eigs_gen(eigval, eigvec, X, k)` | Compute $k$ largest eigenvalues and eigenvectors of matrix $X$ |
| `svds(U, s, V, X, k)` | Compute $k$ singular values and singular vectors of matrix $X$ |
| `X = spsolve(A, b)` | Solve sparse system $Ax = b$ for $x$ |

**Table 1.** Subset of available functionality for the sparse matrix class, with brief descriptions. Optional additional arguments have been omitted for brevity. See http://arma.sf.net/docs.html#SpMat for more detailed documentation.

```
#include <armadillo>
using namespace arma;

int main()
  {
  // generate random sparse 1000x1000 matrix with 1% density of non-zero values,
  // with uniform distribution of values in the [0,1] interval
  sp_mat A = sprandu(1000, 1000, 0.01);

  // multiply A by its transpose
  sp_mat B = A * A.t();

  // add scalar to main diagonal
  B.diag() += 0.1;

  // declare dense vector and matrix
  vec eigvals; mat eigvecs;

  // find 3 eigenvectors of sparse matrix B
  eigs_sym(eigvals, eigvecs, B, 3);

  return 0;
  }
```

**Figure 1.** A small C++ program to demonstrate usage of the sparse matrix class (`sp_mat`).

## 3. Template-Based Optimisation of Compound Expressions

The sparse matrix class uses a delayed evaluation approach, allowing several operations to be combined to reduce the amount of computation and/or temporary objects. In contrast to brute-force evaluations, delayed evaluation can provide considerable performance improvements as well as reduced memory usage [18]. The delayed evaluation machinery is accomplished through template meta-programming [11,12], where a type-based signature of a compound expression (set of consecutive mathematical operations) is automatically constructed. The C++ compiler is then automatically induced to detect common expression patterns at compile time, followed by selecting the most computationally efficient implementations.

As an example of the possible efficiency gains, let us consider the expression `trace(A.t() * B)`, which often appears as a fundamental quantity in semidefinite programs [19]. These computations are thus used in a wide variety of diverse fields, most notably machine learning [20–22]. A brute-force implementation would evaluate the transpose first, `A.t()`, and store the result in a temporary matrix `T1`. The next operation would be a time consuming matrix multiplication, `T1 * B`, with the result stored in another temporary matrix `T2`. The trace operation (sum of diagonal elements) would then be applied on `T2`. The explicit transpose, full matrix multiplication and creation of the temporary matrices is suboptimal from an efficiency point of view, as for the trace operation we require only the diagonal elements of the `A.t() * B` expression.

Template-based expression optimisation can avoid the unnecessary operations. Let us declare two lightweight objects, `Op` and `Glue`, where `Op` objects are used for representing unary operations, while `Glue` objects are used for representing binary operations. The objects are lightweight as they do not store actual sparse matrix data; instead the objects only store references to matrices and/or other `Op` and `Glue` objects. Ternary and more complex operations are represented through combinations of `Op` and `Glue` objects. The exact type of each `Op` and `Glue` object is automatically inferred from a given mathematical expression through template meta-programming.

In our example, the expression `A.t()` is automatically converted to an instance of the lightweight `Op` object with the following type:

```
Op<sp_mat, op_trans>
```

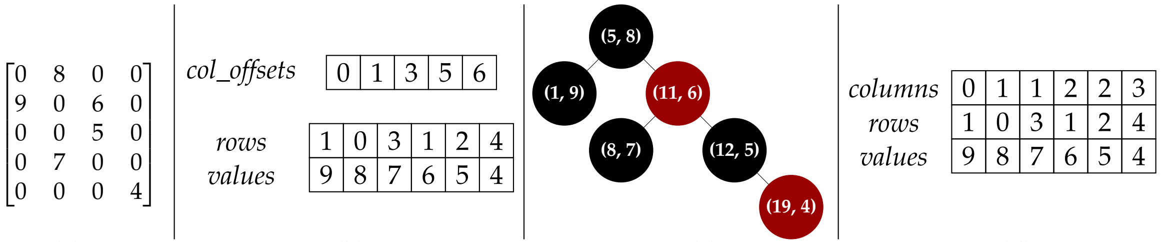
where `Op<...>` indicates that `Op` is a template class, with the items between '<' and '>' specifying template parameters. In this case the `Op<sp_mat, op_trans>` object type indicates that a reference to a matrix is stored and that a transpose operation is requested. In turn, the compound expression `A.t() * B` is converted to an instance of the lightweight `Glue` object with the following type:

```
Glue< Op<sp_mat, op_trans>, sp_mat, glue_times>
```

where the `Glue` object type in this case indicates that a reference to the preceding `Op` object is stored, a reference to a matrix is stored, and that a matrix multiplication operation is requested. In other words, when a user writes the expression `trace(A.t() * B)`, the C++ compiler is induced to represent it internally as `trace(Glue< Op<sp_mat, op_trans>, sp_mat, glue_times>(A,B))`.

There are several implemented forms of the `trace()` function, one of which is automatically chosen by the C++ compiler to handle the `Glue< Op<sp_mat, op_trans>, sp_mat, glue_times>` expression. The specific form of `trace()` takes references to the `A` and `B` matrices, and executes a *partial* matrix multiplication to obtain only the diagonal elements of the `A.t() * B` expression. All of this is accomplished without generating temporary matrices. Furthermore, as the `Glue` and `Op` objects only hold references, they are in effect optimised away by modern C++ compilers [12]: the resultant machine code appears as if the `Glue` and `Op` objects never existed in the first place.

The template-based delayed evaluation approach has also been employed for other functions, such as the `diagmat()` function, which obtains a diagonal matrix from a given expression. For example, in the expression `diagmat(A + B)`, only the diagonal components of the `A + B` expression are evaluated.



(a) (b) (c) (d)

**Figure 2.** Illustration of sparse matrix representations: (a) example sparse matrix with 5 rows, 4 columns and 6 non-zero values, shown in traditional mathematical notation; (b) corresponding CSC representation; (c) corresponding RBT representation, where each node is expressed by $(i, v)$, with $i$ indicating a linearly encoded matrix location and $v$ indicating the value held at that location; (d) corresponding COO representation. Following C++ convention [2], we use zero-based indexing.

## 4. Storage Formats for Sparse Data

We have chosen the three underlying storage formats (CSC, RBT, COO) to give overall efficient execution across several use cases, as well as to minimise the difficulty of implementation and code maintenance burden where possible. Specifically, our focus is on the following main use cases:

1. Flexible ad-hoc construction and element-wise modification of sparse matrices via unordered insertion of elements, where each new element is inserted at a random location.
2. Incremental construction of sparse matrices via quasi-ordered insertion of elements, where each new element is inserted at a location that is past all the previous elements according to column-major ordering.
3. Multiplication of dense vectors with sparse matrices.
4. Multiplication of two sparse matrices.
5. Operations involving bulk coordinate transformations, such as flipping matrices column- or row-wise.

The three storage formats as well as their benefits and limitations are briefly described below. We use $N$ to indicate the number of non-zero elements of the matrix, while *n_rows* and *n_cols* indicate the number of rows and columns, respectively.

### *4.1. Compressed Sparse Column (CSC)*

The CSC format [4] uses column-major ordering where the elements are stored column-by-column, with consecutive non-zero elements in each column stored consecutively in memory. Three arrays are used to represent a sparse matrix:

1. The *values* array, which is a contiguous array of $N$ floating point numbers holding the non-zero elements.
2. The *rows* array, which is a contiguous array of $N$ integers holding the corresponding row indices (ie., the $n$-th entry contains the row of the $n$-th element).
3. The *col_offsets* array, which is a contiguous array of $n\_cols + 1$ integers holding offsets to the *values* array, with each offset indicating the start of elements belonging to each column.

Following C++ convention [2], all arrays use zero-based indexing, ie., the initial position in each array is denoted by 0. For consistency, element locations within a matrix are also encoded as starting at zero, ie., the initial row and column are both denoted by 0. Furthermore, the row indices for elements in each column are kept sorted in ascending manner. In many applications, sparse matrices have more non-zero elements than the number of columns, leading to the *col_offsets* array being typically much smaller than the *values* array.

Let us denote the $i$-th entry in the *col_offsets* array as $c[i]$, the $j$-th entry in the *rows* array as $r[j]$, and the $n$-th entry in the *values* array as $v[n]$. The number of non-zero elements in column $i$ is determined using $c[i{+}1] - c[i]$, where, by definition, $c[0]$ is always 0 and $c[n\_cols]$ is equal to $N$. If column $i$ has non-zero elements, then the first element is obtained via $v[\,c[i]\,]$, and $r[\,c[i]\,]$ is the corresponding row of the element. An example of this format is shown in Figure 2(b).

The CSC format is well-suited for efficient sparse linear algebra operations such as vector-matrix multiplication. This is due to consecutive non-zero elements in each column being stored next to each other in memory, which allows modern CPUs to speculatively read ahead elements from the main memory into fast cache memory [23]. The CSC format is also suited for operations that do not change the structure of the matrix, such as element-wise operations on the non-zero elements (eg., multiplication by a scalar). The format also affords relatively efficient random element access; to locate an element (or determine that it is not stored), a single lookup to the beginning of the desired column can be performed, followed by a binary search [24] through the *rows* array to find the element.

While the CSC format provides a compact representation yielding efficient execution of linear algebra operations, it has two main disadvantages. The first disadvantage is that the design and implementation of efficient algorithms for many sparse matrix operations (such as matrix-matrix multiplication) tend to be non-trivial [4,25]. This stems not only from the sparse nature of the data, but also due to the need to (i) explicitly keep track of the column offsets, and (ii) keep the row indices for elements in each column sorted in ascending manner. In our experience, the process of designing and implementing efficient matrix processing algorithms in CSC is a time-consuming affair — it is both finicky and prone to subtle bugs.

The second disadvantage of CSC is the computational effort required to insert a new element [6]. In the worst-case scenario, memory for three new larger-sized arrays (containing the values and locations) must first be allocated, the position of the new element determined within the arrays, data from the old arrays copied to the new arrays, data for the new element placed in the new arrays, and finally the memory used by the old arrays deallocated. As the number of elements in the matrix grows, the entire process becomes slower.

There are opportunities for some optimisation, such as using oversized storage to reduce memory allocations, where a new element past all the previous elements can be readily inserted. However, this does not help when a new non-zero element is inserted between two existing non-zero elements. It is also possible to perform batch insertions with some speedup by first sorting all the elements to be inserted and then merging with the existing data arrays. While the above approaches can be effective, they require the user to explicitly deal with cumbersome low-level storage details instead of focusing on high-level functionality.

The CSC format was chosen over the related Compressed Sparse Row (CSR) format [4] for two main reasons: (i) to ensure compatibility with external libraries such as the SuperLU solver [15], and (ii) to ensure consistency with the surrounding infrastructure provided by the Armadillo library, which uses column-major dense matrix representation to take advantage of low-level functions provided by LAPACK [26].

### 4.2. Red-Black Tree (RBT)

To address the efficiency problems with element insertion at arbitrary locations, we first represent each element as a 2-tuple, $l = (index, value)$, where *index* encodes the location of the element as $index = row + column \times n\_rows$. Zero-based indexing is used. This encoding implicitly assumes column-major ordering of the elements. Secondly, rather than using a simple linked list or an array based representation, the list of the tuples is stored as a Red-Black Tree (RBT), a self-balancing binary search tree [24].

Briefly, an RBT is a collection of nodes, with each node containing the 2-tuple described above and links to two children nodes. There are two constraints: (i) each link points to a unique child node, and (ii) there are no links to the root node. The *index* within each 2-tuple is used as the key to identify each node. An example of this structure for a simple sparse matrix is shown in Figure 2(c). The ordering of the nodes and height of the tree (number of node levels below the root node) is controlled so that searching for a specific index (ie., retrieving an element at a specific location) has worst-case complexity of $\mathcal{O}(\log N)$. Insertion and removal of nodes (ie., matrix elements), also has the worst-case complexity of $\mathcal{O}(\log N)$. If a node to be inserted is known to have the largest index so far (eg., during incremental matrix construction), the search for where to place the node can be omitted, which in practice can considerably speed up the insertion process.

With the above element encoding, traversing an RBT in an ordered fashion (from the smallest to largest index) is equivalent to reading the elements in column-major ordering. This in turn allows for quick conversion of matrix data stored in RBT format into CSC format. The location of each element is simply decoded via $row = (index \mod n\_rows)$, and $column = \lfloor index / n\_rows \rfloor$, where, for clarity, $\lfloor z \rfloor$ is the integer version of $z$, rounded towards zero. These operations are accomplished via direct integer arithmetic on CPUs. More details on the conversion are given in Section 5.

Within the hybrid storage framework, the RBT format is used for incremental construction of sparse matrices, either in an ordered or unordered fashion, and a subset of element-wise operations (such as inplace addition of values to specified elements). This in turn enables users to construct sparse matrices in the same way they might construct dense matrices—for instance, a loop over elements to be inserted without regard to storage format.

While the RBT format allows for fast element insertion, it is less suited than CSC for efficient linear algebra operations. The CSC format allows for exploitation of fast caches in modern CPUs due to the consecutive storage of non-zero elements in memory [23]. In contrast, accessing consecutive elements in the RBT format requires traversing the tree (following links from node to node), which in turn entails accessing node data that is not guaranteed to be consecutively stored in memory. Furthermore, obtaining the column and row indices requires explicit decoding of the index stored in each node, rather than a simple lookup in the CSC format.

### 4.3. Coordinate List Representation (COO)

The Coordinate List (COO) is a general concept where a list $L = (l_1, l_2, \cdots, l_N)$ of 3-tuples represents the non-zero elements in a matrix. Each 3-tuple contains the location indices and value of the element, ie., $l = (row, column, value)$. The format does not prescribe any ordering of the elements, and a simple linked list [24] can be used to represent $L$. However, in a computational implementation geared towards linear algebra operations [4], $L$ is often represented as a set of three arrays:

1. The *values* array, which is a contiguous array of $N$ floating point numbers holding the non-zero elements of the matrix.
2. The *rows* array, a contiguous array of $N$ integers holding the row index of the corresponding value.
3. The *columns* array, a contiguous array of $N$ integers holding the column index of the corresponding value.

As per the CSC format, all arrays use zero-based indexing, ie., the initial position in each array is 0. The elements in each array are sorted in column-major order for efficient lookup.

The array-based representation of COO is related to CSC, with the main difference that for each element the column indices are explicitly stored. This leads to the primary advantage of the COO format: it can greatly simplify the implementation of matrix processing algorithms. It also tends to be a natural format many non-expert users expect when first encountering sparse matrices. However, due to the explicit representation of column indices, the COO format contains redundancy and is hence less efficient (spacewise) than CSC for representing sparse matrices. An example of this is shown in Figure 2(d).

To contrast the differences in effort required in implementing matrix processing algorithms in CSC and COO, let us consider the problem of sparse matrix transposition. When using the COO format this is trivial to implement: simply swap the *rows* array with the *columns* array and then re-sort the elements so that column-major ordering is maintained. However, the same task for the CSC format is considerably more specialised: an efficient implementation in CSC would likely use an approach such as the elaborate `TRANSP` algorithm by Bank and Douglas [25], which is described through a 47-line pseudocode algorithm with annotations across two pages of text.

Our initial implementation of sparse matrix transposition used the COO based approach. COO was used simply due to shortage of available time for development and the need to flesh out other parts of sparse matrix functionality. When time allowed, we reimplemented sparse matrix transposition to use the abovementioned `TRANSP` algorithm. This resulted in considerable speedups, due to no longer requiring the time-consuming sort operation. We verified that the new CSC-based implementation is correct by comparing its output against the previous COO-based implementation on a large set of test matrices.

The relatively straightforward nature of COO format hence makes it well-suited for: (i) functionality contributed by time-constrained and/or non-expert users, (ii) relatively complex and/or less-common sparse matrix operations, and (iii) verifying the correct implementation of algorithms in the more complex CSC format. The volunteer driven nature of the Armadillo project makes its vibrancy and vitality depend in part on contributions received from users and the maintainability of the codebase. The number of core developers is small (ie., the authors of this paper), and hence difficult-to-understand or difficult-to-maintain code tends to be avoided, since the resources are simply not available to handle that burden.

The COO format is currently employed for less-commonly used tasks that involve bulk coordinate transformations, such as `reverse()` for flipping matrices column- or row-wise, and `repelem()`, where a matrix is generated by replicating each element several times from a given matrix. While it is certainly possible to adapt these functions to directly use the more complex CSC format, at the time of writing we have spent our time-constrained efforts on optimising and debugging more commonly used parts of the sparse matrix class.

## 5. Automatic Conversion Between Storage Formats

To circumvent the problems associated with selection and manual conversion between storage formats, our sparse matrix class employs a hybrid storage framework that *automatically* and *seamlessly* switches between the formats described in Section 4. By default, matrix elements are stored in CSC format. When needed, data in CSC format is internally converted to either the RBT or COO format, on which an operation or set of operations is performed. The matrix is automatically converted ('synced') back to the CSC format the next time an operation requiring the CSC format is performed.

The storage details and conversion operations are completely hidden from the user, who may not necessarily be knowledgeable about (or care to learn about) sparse matrix storage formats. This allows for simplified user code that focuses on high-level algorithm logic, which in turn increases readability and lowers maintenance. In contrast, other toolkits without automatic format conversion can cause either slow execution (as a non-optimal storage format might be used), or require many manual conversions. As an example, Figure 3 shows a short Python program using the SciPy toolkit [1] and a corresponding C++ program using the hybrid sparse matrix class. Manually initiated format conversions are required for efficient execution in the SciPy version; this causes both development time and code required to increase. If the user does not carefully consider the type of their sparse matrix at all times, they are likely to write inefficient code. In contrast, in the C++ program the format conversion is done automatically and behind the scenes.

A potential drawback of the automatic conversion between formats is the added computational cost. However, it turns out that COO/CSC conversions can be done in time that is linear in the number of non-zero elements in the matrix, and that CSC/RBT conversions can be done at worst in log-linear time. Since most sparse matrix operations are more expensive (eg., matrix multiplication), the conversion overhead turns out to be mostly negligible in practice. Below we present straightforward algorithms for conversion and note their asymptotic complexity in terms of the $\mathcal{O}$ notation [24]. This

```
X = scipy.sparse.rand(1000, 1000, 0.01)

# manually convert to LIL format
# to allow insertion of elements
X = X.tolil()
X[1,1]  = 1.23
X[3,4] += 4.56

# random dense vector
V = numpy.random.rand((1000))

# manually convert X to CSC format
# for efficient multiplication
X = X.tocsc()
W = V * X
```

```
sp_mat X = sprandu(1000, 1000, 0.01);

// automatic conversion to RBT format
// for fast insertion of elements

X(1,1)  = 1.23;
X(3,4) += 4.56;

// random dense vector
rowvec V(1000, fill::randu);

// automatic conversion of X to CSC
// prior to multiplication

rowvec W = V * X;
```

**Figure 3.** Left panel: a Python program using the SciPy toolkit, requiring explicit conversions between sparse format types to achieve efficient execution; if an unsuitable sparse format is used for a given operation, SciPy will emit *TypeError* or *SparseEfficiencyWarning*. Right panel: A corresponding C++ program using the sparse matrix class, with the format conversions automatically done by the class.

is followed by discussing practical considerations that are not directly taken into account by the $\mathcal{O}$ notation.

### *5.1. Conversion Between COO and CSC*

Since the COO and CSC formats are quite similar, the conversion algorithms are straightforward. In fact the only parts of the formats to be converted are the *columns* and *col_offsets* arrays with the *rows* and *values* arrays remaining unchanged.

The algorithm for converting COO to CSC is given in Figure 4(a). In summary, the algorithm first determines the number of elements in each column (lines 6-8), and then ensures that the values in the *col_offsets* array are consecutively increasing (lines 9-10) so that they indicate the starting index of elements belonging to each column within the *values* array. The operations listed on line 5 and lines 9-10 each have a complexity of approximately $\mathcal{O}(n\_cols)$, while the operation listed on lines 6-8 has a complexity of $\mathcal{O}(N)$, where $N$ is the number of non-zero elements in the matrix and $n\_cols$ is the number of columns. The complexity is hence $\mathcal{O}(N + 2n\_cols)$. As in most applications the number of non-zero elements will be considerably greater than the number of columns, the overall asymptotic complexity in these cases is $\mathcal{O}(N)$.

The corresponding algorithm for converting CSC to COO is shown in Figure 4(b). In essence the *col_offsets* array is unpacked into a *columns* array with length $N$. As such, the asymptotic complexity of this operation is $\mathcal{O}(N)$.

### *5.2. Conversion Between CSC and RBT*

The conversion between the CSC and RBT formats is also straightforward and can be accomplished using the algorithms shown in Figure 5. In essence, the CSC to RBT conversion involves encoding the location of each matrix element to a linear index, followed by inserting a node with that index and the corresponding element value into the RBT. The worst-case complexity for inserting all elements into an RBT is $\mathcal{O}(N \cdot \log N)$. However, as the elements in the CSC format are guaranteed to be stored according to column-major ordering (as per Section 4.1), and the location encoding assumes column-major ordering (as per Section 4.2), the insertion of a node into an RBT can be accomplished without searching for the node location. While the worst-case cost of $\mathcal{O}(N \cdot \log N)$ is maintained due to tree maintenance (ie., controlling the height of the tree) [24], in practice the amortised insertion cost is typically lower due to avoidance of the search.

```
proc COO_to_CSC
  input: $N$, $n\_cols$ (integer scalars)
  input: $values$, $rows$, $columns$ (COO arrays)
  allocate array $col\_offsets$ with length $n\_cols + 1$
  forall $j \in [0, n\_cols]$: $col\_offsets[j] \leftarrow 0$
  forall $i \in [0, N)$:
    $j \leftarrow columns[i] + 1$
    $col\_offsets[j] \leftarrow col\_offsets[j] + 1$
  forall $j \in [1, n\_cols]$:
    $col\_offsets[j] \leftarrow col\_offsets[j] + col\_offsets[j-1]$
  output: $values$, $rows$, $col\_offsets$ (CSC arrays)
```

(a)

```
proc CSC_to_COO
  input: $N$, $n\_cols$ (integer scalars)
  input: $values$, $rows$, $col\_offsets$ (CSC arrays)
  allocate array $columns$ with length $N$
  $k \leftarrow 0$
  forall $j \in [0, n\_cols)$:
    $M \leftarrow col\_offsets[j+1] - col\_offsets[j]$
    forall $l \in [0, M)$:
      $columns[k+l] \leftarrow j$
    $k \leftarrow k + M$
  output: $values$, $rows$, $columns$ (COO arrays)
```

(b)

**Figure 4.** Algorithms for: (a) conversion from COO to CSC, and (b) conversion from CSC to COO. Matrix elements in COO format are assumed to be stored in column-major ordering. All arrays and matrix locations use zero-based indexing. $N$ indicates the number of non-zero elements, while *n_cols* indicates the number of columns. Details for the CSC and COO arrays are given in Section 4.

```
proc CSC_to_RBT
  input: $N$, $n\_rows$, $n\_cols$ (integer scalars)
  input: $values$, $rows$, $col\_offsets$ (CSC arrays)
  declare red-black tree $T$
  forall $j \in [0, n\_cols)$:
    $start \leftarrow col\_offsets[j]$
    $end \leftarrow col\_offsets[j+1]$
    forall $k \in [start, end)$:
      $index \leftarrow row\_indices[k] + j * n\_rows$
      $l \leftarrow (index, values[k])$
      insert node $l$ into $T$
  output: $T$ (red-black tree)
```

(a)

```
proc RBT_to_CSC
  input: $N$, $n\_rows$, $n\_cols$ (integer scalars)
  input: $T$ (red-black tree)
  allocate array $values$ with length $N$
  allocate array $row\_indices$ with length $N$
  allocate array $col\_offsets$ with length $n\_cols + 1$
  forall $j \in [0, n\_cols]$: $col\_offsets[j] \leftarrow 0$
  $k \leftarrow 0$
  foreach node $l \in T$, where $l = (index, value)$:
    $values[k] \leftarrow value$
    $row\_indices[k] \leftarrow index \bmod n\_rows$
    $j \leftarrow \lfloor index / n\_rows \rfloor$
    $col\_offsets[j+1] \leftarrow col\_offsets[j+1] + 1$
    $k \leftarrow k + 1$
  forall $j \in [1, n\_cols]$:
    $col\_offsets[j] \leftarrow col\_offsets[j] + col\_offsets[j-1]$
  output: $values$, $rows$, $col\_offsets$ (CSC arrays)
```

(b)

**Figure 5.** Algorithms for: (a) conversion from CSC to RBT, and (b) conversion from RBT to CSC. All arrays and matrix locations use zero-based indexing. $N$ indicates the number of non-zero elements, while *n_rows* and *n_cols* indicate the number of row and columns, respectively. Details for the CSC arrays are given in Section 4.

Converting an RBT to CSC involves traversing through the nodes of the tree from the lowest to highest index, which is equivalent to reading the elements in column-major format. The value stored in each node is hence simply copied into the corresponding location in the CSC *values* array. The index stored in each node is decoded into row and column indices, as per Section 4.2, with the CSC *row_indices* and *col_offsets* arrays adjusted accordingly. The worst-case cost for finding each element in the RBT is $\mathcal{O}(\log N)$, which results in the asymptotic worst-case cost of $\mathcal{O}(N \cdot \log N)$ for the whole conversion. However, in practice most consecutive elements are in nearby nodes, which on average reduces the number of traversals across nodes, resulting in considerably lower amortised conversion cost.

### *5.3. Practical Considerations*

Since the conversion algorithms given in Figures 4 and 5 are quite straightforward, the $\mathcal{O}$ notation does not hide any large constant factors. For COO/CSC conversions the cost is $\mathcal{O}(N)$, while for CSC/RBT conversions the worst-case cost in $\mathcal{O}(N \cdot \log N)$. In contrast, many mathematical operations on sparse matrices have much higher computational cost than the conversion algorithms. Even simply adding two sparse matrices can be much more expensive than a conversion. Although the addition operation still takes $\mathcal{O}(N)$ time (assuming $N$ is identical for both matrices), there is a lot of hidden constant overhead, since the sparsity pattern of the resulting matrix must be computed first [4]. A similar situation applies for multiplication of two sparse matrices, which for square matrices takes $\mathcal{O}(N + \texttt{n\_cols})$ time [27], but in practice tends to be much slower due to the many passes and extra overhead of computing the output sparsity structure [25].

Sparse matrix factorisations are much more expensive, meaning that any conversion overhead is essentially negligible. A sparse LU factorisation is superlinear [28] as well as other factorisations like the Cholesky factorisation, which costs $\mathcal{O}(\texttt{n\_cols}^{3/2})$ time [29]. Other factorisations and higher-level operations exhibit similar complexity characteristics. Given this, the cost of format conversions is heavily outweighed by the user convenience that they allow.

## 6. Empirical Evaluation

To demonstrate the advantages of the hybrid storage framework and the template-based expression optimisation mechanism, we have performed a set of experiments, measuring the wall-clock time (elapsed real time) required for:

1. Unordered element insertion into a sparse matrix, where the elements are inserted at random locations in random order.
2. Quasi-ordered element insertion into a sparse matrix, where each new inserted element is at a random location that is past the previously inserted element, under the constraint of column-major ordering.
3. Calculation of trace$(A^T B)$, where $A$ and $B$ are randomly generated sparse matrices.
4. Obtaining a diagonal matrix from the $(A + B)$ expression, where $A$ and $B$ are randomly generated sparse matrices.

In all cases the sparse matrices have a size of 10,000×10,000, with four settings for the density of non-zero elements: 0.01%, 0.1%, 1%, 10%. The experiments were done on a machine with an Intel Xeon E5-2630L CPU running at 2 GHz, using the GCC v5.4 compiler. Each experiment was repeated 10 times, and the average wall-clock time is reported. The wall-clock time measures the total time taken from the start to the end of each run, and includes necessary overheads such as memory allocation.

Figure 6 shows the average wall-clock time taken for element insertion done directly using the underlying storage formats (ie., CSC, COO, RBT, as per Section 4), as well as the hybrid approach which uses RBT followed by conversion to CSC. The CSC and COO formats use oversized storage as a form of optimisation (as mentioned in Section 4.1), where the underlying arrays are grown in chunks of 1024 elements in order to reduce both the number of memory reallocations and array copy operations due to element insertions.

In all cases bar one, the RBT format is the quickest for insertion, generally by one or two orders of magnitude. The conversion from RBT to CSC adds negligible overhead. For the single case of quasi-ordered insertion to reach the density of 0.01%, the COO format is slightly quicker than RBT. This is due to the relatively simple nature of the COO format, as well as the ordered nature of the element insertion where the elements are directly placed into the oversized COO arrays (ie., no sorting required). Furthermore, due to the very low density of non-zero elements and the chunked nature of COO array growth, the number of reallocations of the COO arrays is relatively low. In contrast, inserting a new element into RBT requires the allocation of memory for a new node, and modifying the tree to append the node. For larger densities (≥ 0.1%), the COO element insertion process quickly becomes more time consuming than RBT element insertion, due to to an increased amount of array reallocations and the increased size of the copied arrays. Compared to COO, the CSC format is more complex and has the additional burden of recalculating the column offsets (*col_offsets*) array for each inserted element.

Figure 7 shows the wall-clock time taken to calculate the expressions `trace(A.t()*B)` and `diagmat(A+B)`, with and without the aid of the automatic template-based optimisation of compound

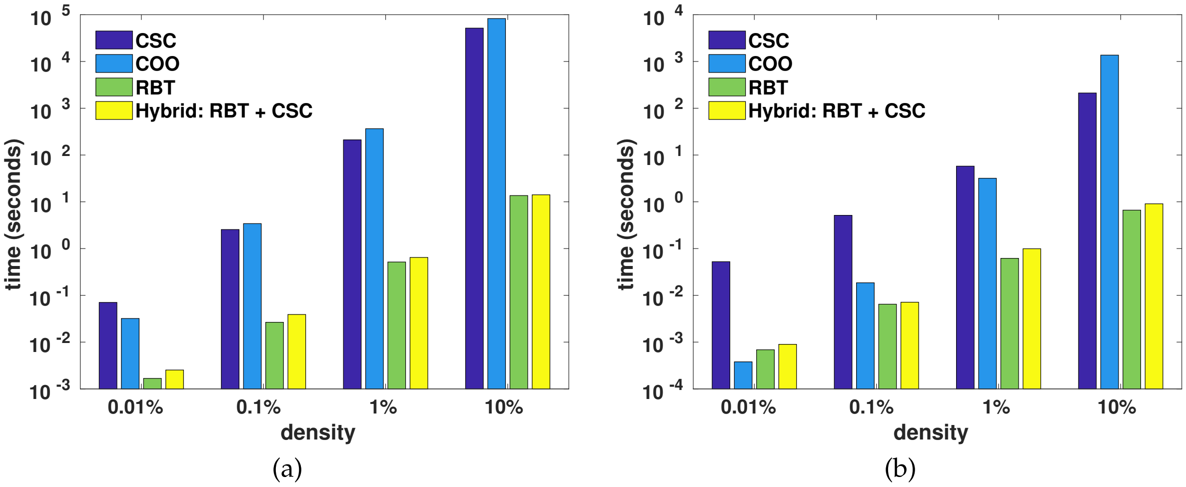


**Figure 6.** Wall-clock time taken to insert elements into a 10,000×10,000 sparse matrix to achieve various densities of non-zero elements. In **(a)**, the elements are inserted at random locations in random order. In **(b)**, the elements are inserted in a quasi-ordered fashion, where each new inserted element is at a random location that is past the previously inserted element, using column-major ordering.

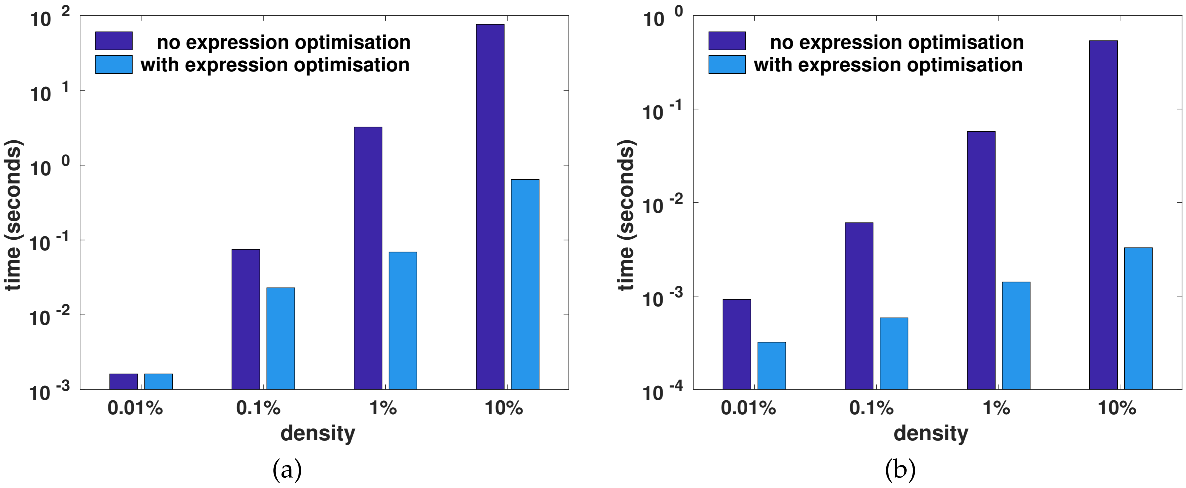


**Figure 7.** Wall-clock time taken to calculate the expressions **(a)** `trace(A.t()*B)` and **(b)** `diagmat(A + B)`, where `A` and `B` are randomly generated sparse matrices with a size of 10,000×10,000 and various densities of non-zero elements. The expressions were calculated with and without the aid of the template-based optimisation of compound expression described in Section 3. As per Table 1, `X.t()` returns the transpose of matrix `X`, while `diagmat(X)` returns a diagonal matrix constructed from the main diagonal of `X`.

expression described in Section 3. For both expressions, employing expression optimisation leads to considerable reduction in the wall-clock time. As the density increases (ie., more non-zero elements), more time is saved via expression optimisation.

For the `trace(A.t()*B)` expression, the expression optimisation computes the trace by omitting the explicit transpose operation and performing a partial matrix multiplication to obtain only the diagonal elements. In a similar fashion, the expression optimisation for the `diagmat(A+B)` expression directly generates the diagonal matrix by performing a partial matrix addition, where only the diagonal elements of the two matrices are added. As well as avoiding full matrix addition, the generation of a temporary intermediary matrix to hold the complete result of the matrix addition is also avoided.

## 7. Conclusion

Driven by a scarcity of easy-to-use tools for algorithm development that requires use of sparse matrices, we have devised a practical sparse matrix class for the C++ language. The sparse matrix class internally uses a hybrid storage framework, which automatically and seamlessly switches between several underlying formats, depending on which format is best suited and/or available for specific operations. This allows the user to write sparse linear algebra without requiring to consider the intricacies and limitations of various storage formats. Furthermore, the sparse matrix class employs a template meta-programming framework that can automatically optimise several common expression patterns, resulting in faster execution.

The source code for the sparse matrix class and its associated functions is included in recent releases of the cross-platform and open-source Armadillo linear algebra library [3], available from http://arma.sourceforge.net. The code is provided under the permissive Apache 2.0 license [30], allowing unencumbered use in both open-source and proprietary projects (eg. product development).

The sparse matrix class has already been successfully used in open-source projects such as the *mlpack* library for machine learning [31], and the *ensmallen* library for mathematical function optimisation [32]. In both cases the sparse matrix class is used to allow various algorithms to be run on either sparse or dense datasets. Furthermore, bi-directional bindings for the class are provided to the R environment via the Rcpp bridge [33]. Avenues for further exploration include expanding the hybrid storage framework with more sparse matrix formats [4,9] in order to provide speedups for specialised use cases.

**Author Contributions:** The authors (C.S. and R.C.) contributed equally in all aspects.

**Funding:** This research received no external funding.

**Acknowledgments:** The authors would like to thank their colleagues at the University of Queensland (Ian Hayes, George Havas, Arnold Wiliem) and Data61/CSIRO (Dan Pagendam, Josh Bowden, Regis Riveret) for discussions leading to the improvements of this article.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Nunez-Iglesias, J.; van der Walt, S.; Dashnow, H. *Elegant SciPy: The Art of Scientific Python*; O'Reilly Media, 2017.
2. Stroustrup, B. *The C++ Programming Language*, 4th ed.; Addison-Wesley, 2013.
3. Sanderson, C.; Curtin, R. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software* **2016**, *1*, 26.
4. Saad, Y. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report NASA-CR-185876, NASA Ames Research Center, 1990.
5. Eaton, J.W.; Bateman, D.; Hauberg, S.; Wehbring, R. *GNU Octave 4.2 Reference Manual*; Samurai Media Limited, 2017.
6. Davis, T.A.; Rajamanickam, S.; Sid-Lakhdar, W.M. A survey of direct methods for sparse linear systems. *Acta Numerica* **2016**, *25*, 383–566.
7. MathWorks. MATLAB Documentation - Accessing Sparse Matrices. https://www.mathworks.com/help/matlab/math/accessing-sparse-matrices.html, 2018.
8. Iglberger, K.; Hager, G.; Treibig, J.; Rüde, U. Expression templates revisited: a performance analysis of current methodologies. *SIAM Journal on Scientific Computing* **2012**, *34*, C42–C69.
9. Duff, I.S.; Erisman, A.M.; Reid, J.K. *Direct methods for sparse matrices*, 2nd ed.; Oxford University Press, 2017.
10. Liniker, P.; Beckmann, O.; Kelly, P.H. Delayed Evaluation, Self-optimising Software Components as a Programming Model. European Conference on Parallel Processing - Euro-Par 2002. Lecture Notes in Computer Science (LNCS), 2002, Vol. 2400, pp. 666–673.
11. Abrahams, D.; Gurtovoy, A. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*; Addison-Wesley Professional, 2004.
12. Vandevoorde, D.; Josuttis, N.M. *C++ Templates: The Complete Guide*, 2nd ed.; Addison-Wesley, 2017.
13. Sanderson, C.; Curtin, R. A User-Friendly Hybrid Sparse Matrix Class in C++. Mathematical Software - ICMS 2018. Lecture Notes in Computer Science (LNCS), 2018, Vol. 10931, pp. 422–430.
14. Lehoucq, R.B.; Sorensen, D.C.; Yang, C. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*; SIAM, 1998.
15. Li, X.S. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)* **2005**, *31*, 302–325.
16. Mernik, M.; Heering, J.; Sloane, A.M. When and how to develop domain-specific languages. *ACM Computing Surveys* **2005**, *37*, 316–344.
17. Scherr, M.; Chiba, S. Almost first-class language embedding: taming staged embedded DSLs. ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, 2015, pp. 21–30.
18. Veldhuizen, T.L. C++ Templates as Partial Evaluation. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, 1999, pp. 13–18.
19. Vandenberghe, L.; Boyd, S. Semidefinite Programming. *SIAM Review* **1996**, *38*, 49–95.
20. Boumal, N.; Voroninski, V.; Bandeira, A. The non-convex Burer-Monteiro approach works on smooth semidefinite programs. Advances in Neural Information Processing Systems, 2016, pp. 2757–2765.
21. El Ghaoui, L.; Lebret, H. Robust solutions to least-squares problems with uncertain data. *SIAM Journal on Matrix Analysis and Applications* **1997**, *18*, 1035–1064.
22. Lanckriet, G.R.; Cristianini, N.; Bartlett, P.; Ghaoui, L.E.; Jordan, M.I. Learning the Kernel Matrix with Semidefinite Programming. *Journal of Machine Learning Research* **2004**, *5*, 27–72.

23. Mittal, S. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Computing Surveys* **2016**, *49*, 35:1–35:35.
24. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*, 3rd ed.; MIT Press, 2009.
25. Bank, R.E.; Douglas, C.C. Sparse matrix multiplication package (SMMP). *Advances in Computational Mathematics* **1993**, *1*, 127–137.
26. Anderson, E.; Bai, Z.; Bischof, C.; Blackford, S.; Demmel, J.; Dongarra, J.; Du Croz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A.; Sorensen, D. *LAPACK Users' Guide*; SIAM, 1999.
27. Davis, T.A. *Direct methods for sparse linear systems*; SIAM, 2006.
28. Gilbert, J.R.; Li, X.S.; Ng, E.G.; Peyton, B.W. Computing row and column counts for sparse QR and LU factorization. *BIT Numerical Mathematics* **2001**, *41*, 693–710.
29. George, A.; Ng, E. On the complexity of sparse QR and LU factorization of finite-element matrices. *SIAM Journal on Scientific and Statistical Computing* **1988**, *9*, 849–861.
30. St. Laurent, A. *Understanding Open Source and Free Software Licensing*; O'Reilly Media, 2008.
31. Curtin, R.; Edel, M.; Lozhnikov, M.; Mentekidis, Y.; Ghaisas, S.; Zhang, S. mlpack 3: a fast, flexible machine learning library. *Journal of Open Source Software* **2018**, *3*, 726.
32. Bhardwaj, S.; Curtin, R.; Edel, M.; Mentekidis, Y.; Sanderson, C. ensmallen: a flexible C++ library for efficient function optimization. *Workshop on Systems for ML and Open Source Software at NIPS / NeurIPS* **2018**.
33. Eddelbuettel, D.; Sanderson, C. RcppArmadillo: Accelerating R with high-performance C++ linear algebra. *Computational Statistics & Data Analysis* **2014**, *71*, 1054–1063.