A User-Friendly Hybrid Sparse Matrix Class in C++

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

[1] Data61, CSIRO, Australia
[2] Symantec Corporation, USA
[3] University of Queensland, Australia
[4] Arroyo Consortium

Abstract. When implementing functionality which requires sparse matrices, there are numerous storage formats to choose from, each with advantages and disadvantages. 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 this issue, we present a user-friendly sparse matrix class for the C++ language, with a high-level application programming interface deliberately similar to the widely used MATLAB language. 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, coordinate list, Red-Black tree) depending on which format is best suited for specific operations, and (ii) template-based meta-programming to automatically detect and optimise execution of common expression patterns. To facilitate relatively quick conversion of research code into production environments, the class and its associated functions provide a suite of essential sparse linear algebra functionality (eg., arithmetic operations, submatrix manipulation) as well as high-level functions for sparse eigendecompositions and linear equation solvers. The latter are achieved by providing easy-to-use abstractions of the low-level ARPACK and SuperLU libraries. The source code is open and provided under the permissive Apache 2.0 license, allowing unencumbered use in commercial products.

Keywords: numerical linear algebra, sparse matrix, C++ language

1 Introduction

Modern scientific computing often requires working with data so large it cannot fully fit in working memory. In many cases, the data can be represented as sparse, allowing users to work with matrices of extreme size with few nonzero elements. However, converting code from using dense matrices to using sparse matrices is not always straightforward.

Existing open-source frameworks may provide several separate sparse matrix classes, each with their own data storage format. For instance, SciPy [10] has 7 sparse matrix classes: `bsr_matrix`, `coo_matrix`, `csc_matrix`, `csr_matrix`, `dia_matrix`, `dok_matrix`, and `lil_matrix`. Each storage format is best suited for efficient execution of a specific set of operations (eg., matrix multiplication vs. incremental matrix construction). 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

⋆ **Published in:** International Congress on Mathematical Software, 2018.

for users who simply want to create and use sparse matrices, and do not have the 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 plurality of sparse matrix classes complicates the programming task, increases the likelihood of bugs, and adds to the maintenance burden.

Motivated by the above issues, we present a user-friendly sparse matrix class for the C++ language, with a high-level application programming interface (function syntax) that is deliberately similar to MATLAB. 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 for specific operations: **(i)** Compressed Sparse Column (CSC), used for efficient fundamental arithmetic operations such as matrix multiplication and addition, as well as efficient reading of individual elements; **(ii)** Co-Ordinate List (COO), used for facilitating operations involving bulk coordinate transformations; **(iii)** Red-Black Tree (RBT), used for both robust and efficient incremental construction of sparse matrices (ie., construction via setting individual elements one-by-one, not necessarily in order). To further promote efficient execution, the class exploits C++ features such as template meta-programming to provide a compile-time expression evaluator, which can automatically detect and optimise common mathematical expression patterns.

The sparse matrix class provides an intuitive interface that is very close to a typical dense matrix API; this can help with rapid transition of dense-specific code to sparse-specific code. In addition, we demonstrate that the overhead of the hybrid format is minimal, and that the format is able to choose the optimal representation for a variety of sparse linear algebra tasks. This makes the format and implementation suitable for real-world prototyping and production usage.

Although there are many other sparse matrix implementations in existence, to our knowledge ours 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. As mentioned earlier, SciPy contains no fewer than seven formats, and the comprehensive SPARSEKIT package [12] contains 16. 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, meaning that users must write their code in special ways to ensure its efficiency [9].

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 [13], available from `http://arma.sourceforge.net`. The code is provided under the permissive Apache 2.0 license [11], allowing unencumbered use in commercial products.

We continue the paper as follows. In Section 2 we overview the functionality provided by the sparse matrix class and its associated functions. In Section 3 we briefly describe the underlying storage formats used by the class, and the tasks that each of the formats is best suited for. Section 4 provides an empirical evaluation showing the performance of the hybrid storage framework in relation to the underlying storage formats. The salient points and avenues for further exploration are summarised in Section 5.

2 Functionality

To allow prototyping directly in C++ as well as to facilitate relatively quick conversion of research code into production environments, 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. Various sparse eigendecompositions and linear equation solvers are also provided. C++ language features such as overloading of operators (eg., `*` and `+`) [14] are exploited to allow mathematical operations with matrices to be expressed in a concise and easy-to-read manner. For instance, given sparse matrices `A`, `B`, and `C`, a mathematical expression such as

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

can be written directly in C++ as

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

Low-level details such as memory management are hidden, allowing the user to concentrate effort on mathematical details. Table 1 lists a subset of the available functionality for the sparse matrix class, `sp_mat`.

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. The delayed evaluation machinery is accomplished through template meta-programming [15], where a type-based signature of a set of consecutive mathematical operations is automatically constructed. The C++ compiler is then induced to detect common expression subpatterns at compile time, and selects the corresponding optimised implementations. For example, in the expression `trace(A.t() * B)`, the explicit transpose and time-consuming matrix multiplication are omitted; only the diagonal elements of `A.t() * B` are accumulated.

Sparse eigendecompositions and linear equation solutions are accomplished through integration with low-level routines in the de facto standard ARPACK [7] and SuperLU libraries [8]. The resultant high-level functions automatically take care of the cumbersome and error-prone low-level management required with these libraries.

3 Underlying Sparse Storage Formats

The three underlying storage formats (CSC, COO, RBT) were chosen so that the sparse matrix class can achieve overall efficient execution of the following five main use cases: (i) 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; (ii) flexible ad-hoc construction or element-wise modification of sparse matrices via unordered insertion of elements, where each new element is inserted at a random location; (iii) operations involving bulk coordinate transformations; (iv) multiplication of dense vectors with sparse matrices; (v) multiplication of two sparse matrices.

Below we briefly describe each storage format and its limitations. 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.

Function	Description
`sp_mat X(100,200)`	Declare sparse matrix with 100 rows and 200 columns
`sp_cx_mat X(100,200)`	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 = kron(A, B)`	Kronecker tensor product of 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 col. (*dim=0*) or row (*dim=1*)
`X.t()` or `trans(X)`	Return transpose of matrix *X*
`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 col. (*dim=0*) or row (*dim=1*) to unit *p*-norm
`trace(A.t() * B)`	Compute trace **omitting** explicit transpose and multiplication
`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. Selected functionality of the sparse matrix class, with brief descriptions. See http://arma.sourceforge.net/docs.html#SpMat for more detailed documentation. Several optional additional arguments have been omitted for brevity.

3.1 Compressed Sparse Column

In the CSC format [12], three arrays are used: (i) the *values* array, which is a contiguous array of N floating point numbers holding the non-zero elements, (ii) the *row indices* 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), and (iii) the *column 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. Let us denote the i-th entry in the column offsets array as $c[i]$, the j-th entry in the row indices array as $r[j]$, and the n-th entry in the values array as $v[n]$. All arrays use zero-based indexing, ie., the initial position in each array is denoted by 0. Then, $v[\,c[i]\,]$ is the first element in column i, and $r[\,c[i]\,]$ is the corresponding row of the element. The number of 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.

The CSC format is well-suited for sparse linear algebra operations such as summation and vector-matrix multiplication. It is also suited for operations that do not change the structure of the matrix, such as element-wise operations on the nonzero elements. 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 to find the element.

The main disadvantage of CSC is the effort required to insert a new element. 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. 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 low-level storage details instead of focusing on high-level functionality.

The CSC format was chosen over the related Compressed Sparse Row (CSR) format [12] for two main reasons: (i) to ensure compatibility with external libraries such as the SuperLU solver [8], and (ii) to ensure consistency with the surrounding infrastructure provided by the Armadillo library, which uses column-major dense matrix representation for compatibility with LAPACK [1].

3.2 Coordinate List Representation

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., $l = (\texttt{row}, \texttt{column}, \texttt{value})$. The format does not prescribe any ordering of the elements, and a linked list [2] can be used to represent L. However, in a computational implementation geared towards linear algebra operations [12], L is often represented as a set of three arrays: (i) the *values* array, which is a contiguous array of N floating point numbers holding the non-zero elements of the matrix, and the (ii) *rows* and (iii) *columns* arrays, which are contiguous arrays of N integers, holding the row and column indices of the corresponding values.

The array-based representation of COO is related to CSC, with the main difference that for each element the column indices are explicitly stored. As such, the COO format contains redundancy and is hence less efficient than CSC for representing sparse matrices. However, in the COO format the coordinates of all elements can be directly read and modified in a batch manner, which facilitates specialised/niche operations that involve bulk transformation of matrix coordinates (eg., circular shifts). In the CSC format such operations are more time-consuming and/or more difficult to implement, as the compressed structure must be taken into account. The general disadvantages of the array-based representation of COO are similar as for the CSC format, in that element insertion is typically a slow process.

3.3 Red-Black Tree

To address the problems with element insertion at arbitrary locations, we first represent each element as a 2-tuple, $l = (\texttt{index}, \texttt{value})$, where `index` encodes the location of the element as $\texttt{index} = \texttt{row} + \texttt{column} \times \texttt{n\_rows}$. This encoding implicitly assumes column-major ordering of the elements. Secondly, rather than using a 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 [2].

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 ordering of the nodes and height of the tree is explicitly 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, thereby speeding up the insertion process close to $\mathcal{O}(1)$ complexity.

Traversing the tree in an ordered fashion (from the smallest to largest index) is equivalent to reading the elements in column-major ordering. This in turn allows the quick conversion of matrix data stored in RBT format into CSC format. Each element's location is simply decoded via `row` = `index` $\bmod$ `n_rows` and `column` = $\lfloor$`index`/`n_rows`$\rfloor$, with the operations accomplished via direct integer arithmetic on CPUs.

In our hybrid format, the RBT format is used for incremental construction of sparse matrices, either in an ordered or unordered fashion, and a subset of elementwise operations. 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.

4 Automatically Switching Between Storage Formats

To avoid the problems associated with selection and manual conversion between formats, our sparse matrix class uses a hybrid storage framework that *automatically* and *seamlessly* switches between the data storage formats described in Section 3.

By default, matrix elements are stored in CSC format. When required, 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 actual underlying 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 code, 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, Fig. 1 shows a short Python program using the SciPy toolkit and a corresponding C++ program using the sparse matrix class. Manually initiated format conversions are required for efficient execution in the SciPy version; this causes both development time and code size to increase.

To empirically demonstrate the usefulness of the hybrid storage framework we have performed several experiments: (i) quasi-ordered element insertion, ie., incremental construction, (ii) unordered (random) insertion, and (iii) matrix multiplication. 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%.

```
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;
```

Fig. 1. 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.

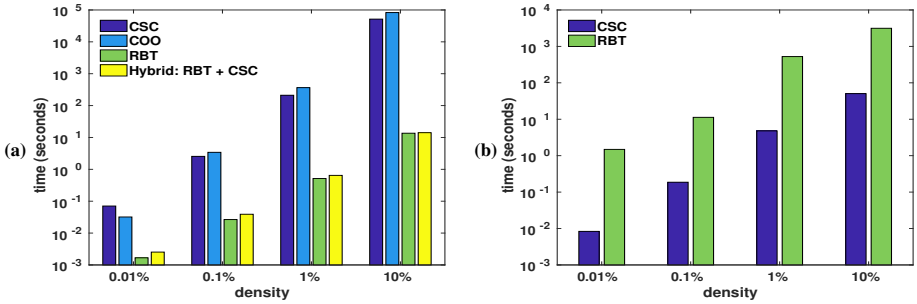

Fig. 2. Time taken to **(a)** insert elements at random locations into a sparse matrix to achieve various densities of non-zero elements, and **(b)** multiply two sparse matrices with elements at random locations and various densities. In both cases the sparse matrices have a size of 10,000×10,000.

Fig. 2(a) shows the time taken for unordered element insertion done directly using the underlying storage formats (ie., CSC, COO, RBT, as per Sec. 3), 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. The RBT format is the quickest, generally by one or two orders of magnitude, with the conversion from RBT to CSC adding negligible overhead. The results for quasi-ordered insertion (not shown) follow a similar pattern.

Fig. 2(b) shows the time taken to multiply two sparse matrices in either CSC or RBT format, with the matrix elements already stored in each format. The COO format was omitted due to its similarity with CSC. The hybrid storage format automatically uses CSC for matrix multiplication, which is faster than RBT by about two orders of magnitude.

5 Conclusion

Motivated by a lack of easy-to-use tools for sparse matrix development, we have proposed and implemented a sparse matrix class in C++ that internally uses a hybrid format. The hybrid format automatically converts between good representations for specific functionality, allowing the user to write sparse linear algebra without requiring to consider the underlying storage format. Internally, the hybrid format uses the CSC (compressed sparse column), COO (coordinate list), and RBT (red-black tree) formats. In addition, template meta-programming is used to optimise common expression patterns. We have made our implementation available as part of the open-source Armadillo C++ library [13].

The class has already been successfully used in open-source projects such as MLPACK, a C++ library for machine learning and pattern recognition [3]. It is used there to allow machine learning algorithms to be run on either sparse or dense datasets. Furthermore, bindings are provided to the R environment via RcppArmadillo [6].

Future avenues for exploration include integrating more specialised matrix formats in order to automatically speed up specific operations. For example, the Skyline formats [4] are useful for Cholesky factorisation and related operations.

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., et al.: LAPACK Users' Guide. SIAM (1999)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, 3rd edn. (2009)
3. Curtin, R., Cline, J., Slagle, N., March, W., Ram, P., Mehta, N., Gray, A.: MLPACK: a scalable C++ machine learning library. J. Machine Learning Research **14**, 801–805 (2013)
4. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct methods for sparse matrices. Oxford University Press, 2nd edn. (2017)
5. Eaton, J.W., Bateman, D., Hauberg, S., Wehbring, R.: GNU Octave 4.2 Reference Manual. Samurai Media Limited (2017)
6. Eddelbuettel, D., Sanderson, C.: RcppArmadillo: Accelerating R with high-performance C++ linear algebra. Computational Statistics & Data Analysis **71**, 1054–1063 (2014)
7. Lehoucq, R.B., Sorensen, D.C., Yang, C.: ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods. SIAM (1998)
8. Li, X.S.: An overview of SuperLU: Algorithms, implementation, and user interface. ACM Transactions on Mathematical Software (TOMS) **31**(3), 302–325 (2005)
9. MathWorks: MATLAB Documentation - Accessing Sparse Matrices. https://www.mathworks.com/help/matlab/math/accessing-sparse-matrices.html (2018)
10. Nunez-Iglesias, J., van der Walt, S., Dashnow, H.: Elegant SciPy: The Art of Scientific Python. O'Reilly Media (2017)
11. Rosen, L.: Open Source Licensing. Prentice Hall (2004)
12. Saad, Y.: SPARSKIT: A basic tool kit for sparse matrix computations. Tech. Rep. NASA-CR-185876, NASA Ames Research Center (1990)
13. Sanderson, C., Curtin, R.: Armadillo: a template-based C++ library for linear algebra. Journal of Open Source Software **1**, 26 (2016)
14. Stroustrup, B.: The C++ Programming Language. Addison-Wesley, 4th edn. (2013)
15. Vandevoorde, D., Josuttis, N.M.: C++ Templates: The Complete Guide. Addison-Wesley, 2nd edn. (2017)