Matrix Library: C++ Implementation of a Matrix Class

Authors: Juliet James, Trisha Kholiya

Abstract: *This paper presents MatrixLibrary, a C++ library designed specifically to support computational chemistry workflows. This library is optimized for matrix operations associated with quantum chemical methods such as CNDO/2 and Extended Huckel Theory. The library implements a custom class called Matrix that provides support for matrix arithmetic, linear algebra operations, and matrix decomposition into eigenvalues and eigenvectors. This functionality directly supports computing molecular energies, density matrices, overlap matrices, and other matrices as encountered in CHEM 279. MatrixLibrary is implemented in C++ with specific focus on accuracy and performance. This code is benchmarked in both performance and accuracy to the Armadillo C++ library. This paper will discuss the design, implementation and benchmarking results of MatrixLibrary along with its limitations and opportunities for further improvements of the library.*

Background

Linear algebra is foundational to Quantum Chemistry problems and is needed to calculate many core molecular properties, including energy calculations. Semi-empirical chemical models such as the Complete Neglect of Differential Orbitals (CNDO/2) Model rely on matrices to express a variety of physical quantities. This includes the Fock Matrix, Hamiltonian Matrix, Overlap Matrix and Gamma Matrix. All of these matrices require matrix-based operations such as multiplication, addition, transposition and decomposition into eigenvalues and eigenvectors.

In the CNDO/2 Model and other self-consistent field (SCF) methods, matrix operations are performed iteratively while the energy converges. In each iteration, multiple matrix operations must occur as energy is recalculated. This is why both performance and accuracy of the MatrixLibrary is crucial. Even small inefficiencies or inaccuracies can accumulate and alter the energy result, causing issues for energy calculations.

MatrixLibrary was established as an alternative to Eigen and Armadillo, which are highly-used performance optimized libraries that provide matrix functionality with linear algebra support. Where these other libraries are complex and have features superfluous for computational chemistry needs, MatrixLibrary is simple and specific for functionality for the computational chemistry needs of CHEM 279, a graduate-level course at UC Berkeley titled Numerical Algorithms applied to Computational Quantum Chemistry and contains simple printing mechanics ideal for debugging.

In relation to the C++ standard library, the MatrixLibrary is optimized for matrix operations and includes algorithmic optimizations for performance enhancement. The MatrixLibrary has been developed as a chemistry-oriented matrix library benchmarked to be accurate and speed-optimized.

Objectives

MatrixLibrary has the following objectives:

1. To design and implement a C++ Matrix class that supports matrix arithmetic and linear algebra operations required for computational chemistry and quantum chemical methods.
2. Support external users as a library that can be imported and used in other code spaces.

3. Optimize matrix operations for numerical accuracy and performance.
4. Benchmark the accuracy and performance of the MatrixLibrary against the established, high performing C++ library Armadillo across a wide range of matrix sizes.
5. MatrixLibrary should be able to replace calls made to Eigen or Armadillo for any matrix functionality needed to calculate energy from the Extended Huckel Theory, as needed in Homework 3 of CHEM279.
6. MatrixLibrary should be self-explanatory for the end-user based on code commenting and simple functionality.

Methodology

We began our project by outlining the key components and functionalities needed in order to create a library that could be used as a drop-in replacement for Armadillo or Eigen in Homework 3. Based on the requirements of that assignment, we determined that our library would need to support the following core functionality:

- Constructing basic matrices, including matrices filled with zeros, matrices filled with ones, identity matrices of a specified size, and randomly initialized matrices
- Overloading basic operators, including matrix addition, subtraction, matrix–matrix multiplication, and multiplication by a scalar
- Bounds-checked element access through an overloaded indexing operator
- Printing functionality with overloaded stream operator
- A matrix transpose operation
- Utility functions such as checking matrix symmetry and constructing diagonal matrices
- A full pipeline for diagonalizing square matrices and outputting the corresponding eigenvalues and eigenvectors
- A function analogous to Armadillo's save functionality, allowing results to be saved in a comparable format for Homework 3
- Custom exception handling to provide the user with clear error messages when invalid matrix operations are attempted
    - There are exceptions thrown when matrix dimensions are incompatible, invalid indices are accessed, or when eigendecomposition is called for matrices that do not meet the required constraints (such as non-square or non-symmetric matrices)

It became clear that the functionality that was going to take the most work to implement was diagonalizing a matrix to be able to output the eigenvalues and eigenvectors of the input matrix. We researched algorithmic approaches to this problem, primarily using Numerical Recipes as our source material. We found that there are many different approaches to this problem—each with its own set of merits and drawbacks. We only needed to be able to diagonalize real symmetric matrices, so we ruled out more generalized eigensolver pipelines based on Hessenberg reduction, which are designed for non-symmetric matrices and thus add additional complexity beyond what was needed for our use case. We also ruled out the Jacobi method because of its computational cost. We decided that using the Householder tridiagonalization algorithm, followed by QL was the best balance of efficiency without

introducing too much complexity beyond what's needed for symmetric matrices *(Press et al.; Trefethen and Bau)*. The steps of this pipeline are as follows:

1. Start with a real symmetric matrix $A \in \mathbb{R}^{(n\,x\,n)}$. Using a symmetric matrix guarantees that the eigenvalues are real, and that the eigenvectors can be chosen orthonormal. These guarantees allow the use of orthogonal transformations, which are numerically stable *(Trefethen and Bau)*.

2. Reduce the matrix to tridiagonal form using Householder reduction. The goal is to transform A into a tridiagonal matrix T with the same eigenvalues. To do this, a series of Householder reflections are applied, where each reflection zeroes out entries below the first subdiagonal of a column. After $n - 2$ reflections only the main diagonal and first sub/superdiagonals are non-zero. The transformation can be written as $T = Q^T AQ$, where Q is orthogonal and T is symmetric tridiagonal *(Press et al.)*. Since eigenvalue algorithms are much faster and simpler on tridiagonal matrices, this step reduces the computational cost of each QL iteration from $O(n^3)$ to $O(n^2)$. Technically, it would be possible to skip the Householder reduction step and still perform the eigendecomposition, but the computational cost of the QL iteration would be significantly higher, making the overall algorithm impractical for larger matrices.

3. Next is the QL algorithm step. This begins with the tridiagonal matrix T, which is iteratively diagonalized. At each iteration $T - \mu l = QL$ is decomposed and reassembled as $T_{new} = LQ + \mu I$, where μ is a shift, chosen to accelerate convergence.

4. Next, implicit shifts are applied to improve convergence speed and numerical stability. The shift value μ is chosen based on the lower 2 x 2 block of the tridiagonal matrix (commonly referred to as a Wilkinson shift). This choice of μ causes the eigenvalues to converge more rapidly and accelerates the decay of off-diagonal elements *(Press at al.)*.

5. These QL iterations are repeated until off-diagonal elements are sufficiently small. When an off-diagonal element is reduced to zero, the matrix effectively splits into smaller blocks, where each block can be solved independently.

6. Once the matrix is effectively diagonal, the diagonal entries are the eigenvalues.

7. The eigenvectors can be recovered by accumulating all of the orthogonal transformations applied during the Householder reduction and QL iteration steps. The final eigenvector matrix is formed as $V = Q_{house} Q_{QL}$, where each column of V corresponds to a normalized eigenvector of the original matrix A.

Library Architecture

This MatrixLibrary is organized with a modular architecture that separates class definitions, implementation, benchmarking and tests. This Library is contained within a Github Repository titled MatrixLibrary. This structure follows C++ conventions for scientific software development as outlined by Wilson et. al. in *Best Practices for Scientific Computing*. Code is modular and re-used as much as possible

and the library contains a build script for automating library download. There are also separate build scripts for building testing and benchmarking frameworks.

The codebase is divided into distinct directories depending on their relation to the library. The `include/` directory contains all of the class definitions for the Matrix class. The implementation details are contained within various source files (.cpp) that are located in the `src/` directory. The `include/` directory also contains the custom exception that is defined in this library (`MatrixInvalidSize`) so that the end user is able to try and catch any potential errors and understand the error message. The `include/` directory also contains code to print vectors, which are simply just referred to by the typedef `vec`. By separating the declaration and the code definitions of the functions, users are able to utilize the Matrix class without exposure to the internal implementation details that are located in `src/`.

The MatrixLibrary is built using the CMake build system. CMake is an open source software compilation package that is widely used for software compilation. The `CMakeLists` file contains commands specific to ensuring that this package can be installed as a library for anyone interested in downloading it. It ensures that users have all necessary dependencies installed and provides user-friendly error messages that explain any missing components. Since this repository is being used as a library, it also contains a CMake configuration file that is filed out by the compiler at runtime. This build system ensures that end users are able to access and use the MatrixLibrary.

In addition to the core library functionality, this library also contains benchmarking, testing and code commenting. Documentation of code is essential for usage. As per Fritz, et. al., Doxygen was used as a documentation system, since it's the main documentation tool for the C++ community. Doxygen generates documentation, including PDF documentation that explains the MatrixLibrary functionality. Benchmarking was done with Google Benchmark against the Armadillo library. Testing was done using Google Testing against the Armadillo library. Both benchmarking and testing results will be discussed in the results section.

Our repository also supports test suite layering, as recommended by Fritz, et. al. Upon every push or PR to the GitHub Repository, all tests are run automatically to ensure that no new code change breaks the core functionality and accuracy of the library.

Error Handling

The matrix library supports error handling in a variety of functions. The main error message that the MatrixLibrary throws is `InvalidMatrixSize`. This error is thrown any time the size of the matrix is not compatible with an operation. For example, if the user tries to initialize a matrix with dimensions 2x2 and data {1}, this is not possible, so the program will output an error. This error is only thrown for incompatible operations, like addition of two matrices of different sizes. Error handling is also crucial for the accessing operator to ensure that it is only accessing elements within the confines of the matrix space.

Performance Considerations

To optimize performance, the MatrixLibrary stores matrix elements in a single contiguous one-dimensional vector rather than storing elements using a vector of vector representation. As discussed in *A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers* by

John McCalpin, the most crucial aspect for memory bandwidth is cache miss latency. Storing data as contiguous data representations decreases cache misses and optimizes memory. When stored in a vector of vectors-, accessing elements and looping through elements became performance hindering. Storing the matrix as a flat vector improved performance drastically. This change in data layout within the class required implementing a sufficient accessor that allowed for accessing elements in a 2D way to ensure the user interface emulated matrix layout.

MCalpin's work also shows that one of the simplest ways to optimize performance of cache systems is to optimize for unit-stride access. Flat vector data storage ensures unit stride movements between data elements instead of jumps in data locations that are seen with vector of vector representation. The operators within the MatrixLibrary use unit stride accesses to access data wherever possible to optimize performance.

To further optimize performance of operations in the MatrixLibrary, pointers were used to directly point at the data in the vectors. Then, loops were done using the pointers to access the address in memory containing the data of interest directly, reducing minor data overhead.

Matrix multiplication requires a lot of memory due to the various loops of calculations that need to occur between elements of the matrices. This makes it a prime candidate for algorithmic optimizations. Prior work has shown that high-performance implementations of matrix matrix multiplication achieve substantial speedups by restructuring the computation to maximize data reuse, rather than relying on dot-product algorithms (Goto and van de Geijn, 2008). MatrixLibrary applies these principles by using an outer product loop for Matrix Multiplication, which reduces memory access overhead by reducing temporary variables and memory accesses of the same data within the nested loops. This improved performance drastically in comparison with previous triple-nested dot-product implementation, as shown in Figure 1.

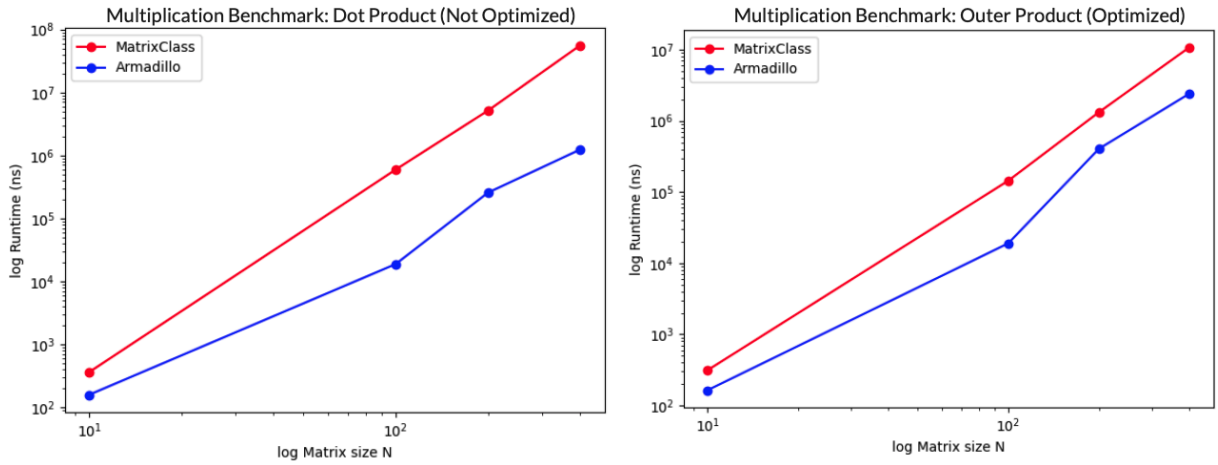

Figure 1. Benchmarking results from Google Benchmarking tests for Matrix Multiplication using the two algorithmic differences, dot product versus outer product. The outer product matrix multiplication has considerable performance improvements in comparison to the dot product matrix multiplication. Most of these improvements can be attributed to improved memory access and reduced temporary variables.

The MatrixLibrary can be further optimized for performance and further optimization will be discussed in the discussion section.

Benchmarking

Performance benchmarking was conducted using the Google Benchmark framework to ensure consistent time measurements and compare performance. The MatrixLibrary was benchmarked against the Armadillo linear algebra library, which supports similar functionality to the MatrixLibrary and is well-used in chemistry-based code.

Benchmarking focused on core operations relevant to computational chemistry workflows and most widely used matrix operations that includes:

- Matrix construction and Initiatialization
- Matrix Addition
- Matrix Multiplication
- Matrix Transposition
- Matrix Accessor
- Matrix Decomposition into Eigenvalues and Eigenvectors

For each operation, benchmarking was done across a wide range of matrix sizes to see performance trends and ensure that performance scaled as expected with matrix size.

Results

Benchmarking Results

Benchmarking of the MatrixLibrary indicated that the MatrixLibrary achieves comparable performance to Armadillo for moderate matrix sizes, but has higher relative runtimes for small matrix sizes. This behavior is expected because the Armadillo library contains many optimizations for small matrix operations. This includes expression templates, loop unrolling and specialized kernels that minimize overhead (Sanderson and Curtin, 2016).

For these smaller matrices, the program runtime is mainly dominated by overhead from the function call overhead rather than the actual arithmetic operations and algorithms used for the operation. The Armadillo library has aggressive optimization that focuses on the cost associated with these calls for small matrices. On the other hand, the MatrixLibrary focuses on an implementation strategy that is general to all matrix sizes and optimizes performance with memory efficient code.

As matrix size increases, performance time becomes dominated by memory access patterns and cache behavior. For these sizes, MatrixLibrary's use of contiguous storage and cache-aware loop ordering results in performance that approaches that of Armadillo. However, Armadillo's HPC optimizations, including its building upon BLAS/LAPACK libraries and ability to be further user optimized with NVIDIA GPU still gives it a performance edge over MatrixLibrary. Armadillo's ability to avoid temporaries through its use of expression templates also give it a slight performance advantage over MatrixLibrary for matrices of all sizes.

Figure 2 shows the benchmarking results for all benchmarked operations of the MatrixLibrary against Armadillo. Note that these are plotted on log-log plots. Importantly, MatrixLibrary shows consistent performance as matrix size scales.

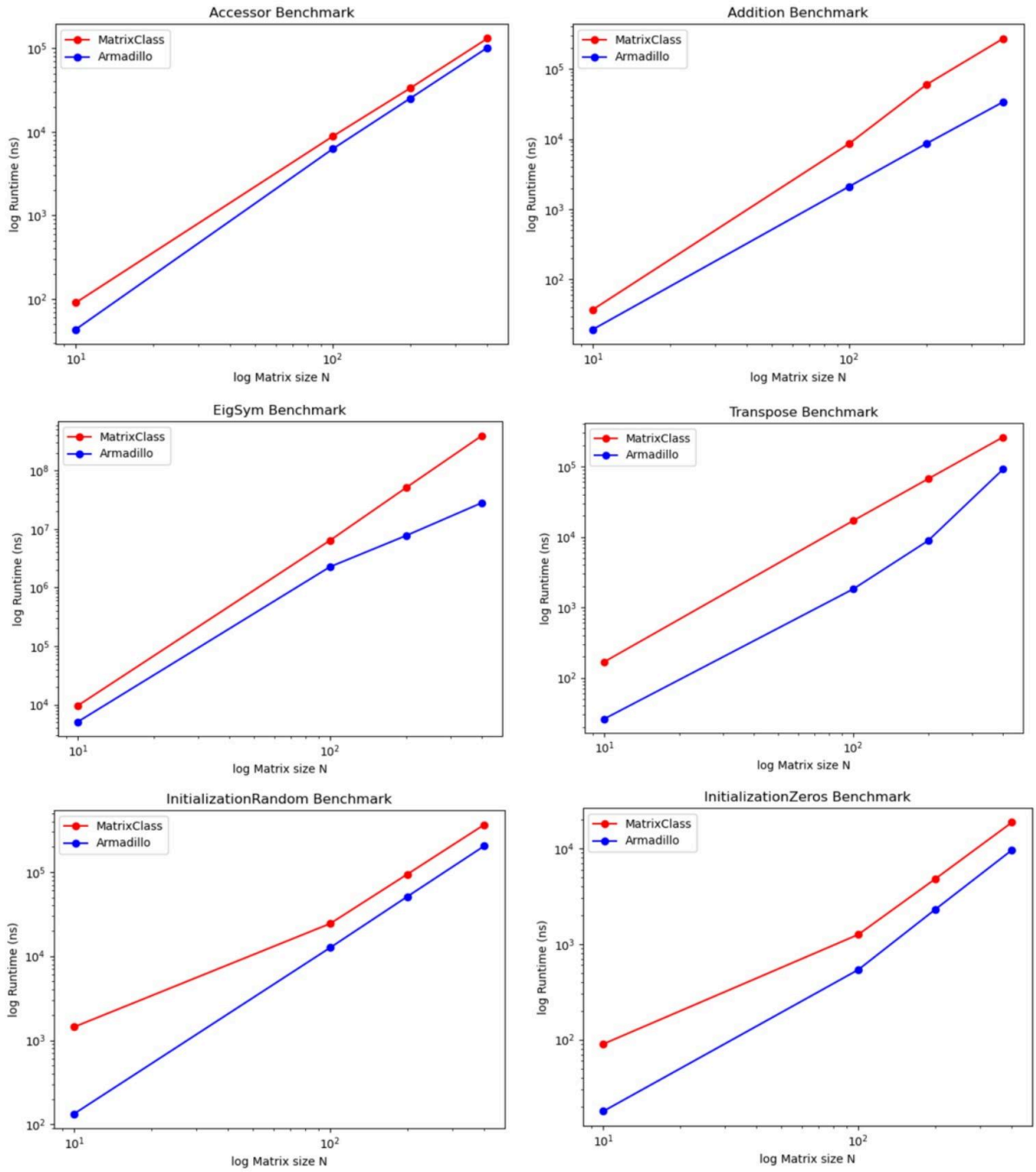

Figure 2. Benchmarking results from Google Benchmark tests of MatrixLibrary versus Armadillo Library. Benchmarking results above show runtime comparisons for accessing elements, matrix-matrix addition, matrix decomposition into eigenvalues and eigenvectors, matrix transposition and constructing a matrix of all random numbers [0.0, 1.0) or of zeros. All of the benchmarking tests are done for matrices of different sizes.

Validating and Benchmarking Accuracy

To confirm and benchmark accuracy, GoogleTest was used to create a testing framework to validate the different functionalities supported by the MatrixLibrary. Armadillo's outputs were used as the source of

truth for comparison and validation, since it relies on well-established and highly optimized numerical routines.

First, the accuracy of the basic matrix operations supported by MatrixLibrary, including addition, subtraction, multiplication, and transpose, were tested. These tests were performed with small matrices of varying sizes. For addition, subtraction, and transpose, our results showed negligible to zero numerical error when compared to Armadillo's outputs. In cases where the error was exactly zero, this is likely because these operations are either commutative or deterministic and were carried out in the same order as Armadillo, leading to identical floating-point results. Matrix–matrix multiplication is not commutative and involves a larger number of floating-point operations, so it makes sense that small numerical differences would be seen due to rounding and accumulation of floating-point error. Small discrepancies were observed between MatrixLibrary results and Armadillo's, which were well within acceptable numerical tolerances.

Operation	Matrix Size / Shape	Max Absolute Error
Construction from Vector	2×3	0
Addition	1×1	0
Addition	5×5	0
Addition	10×10	0
Subtraction	5×5	0
Scalar multiplication	10×10	0
Matrix multiplication	$3 \times 3 \cdot 3 \times 3$	0
Matrix multiplication	$2 \times 3 \cdot 3 \times 4$	1.11×10^{-16}
Transpose	$2 \times 3 \rightarrow 3 \times 2$	0
Transpose	$4 \times 4 \rightarrow 4 \times 4$	0

Table 1. *Maximum absolute error between MatrixLibrary and Armadillo for basic matrix operations across representative matrix sizes and shapes, calculated using Google Benchmark.*

Then, testing was scaled up to benchmark the accuracy of MatrixLibrary's eigensolver, which is implemented in the `eigsym()` function. This was directly compared to Armadillo's `eig_sym()` function. Groups of randomly generated symmetric matrices for square matrix sizes ranging from $n = 1$ to $n = 50$ were tested. Since we evaluated accuracy relative to Armadillo's `eig_sym()` routine, our goal was to assess typical numerical behavior and floating-point error rather than to reproduce identical results across testing runs. This, combined with the broader coverage of possible matrix entries, is why we chose to use randomly generated matrices rather than fixed deterministic inputs for testing our `eigsym()` function's

accuracy. To evaluate the accuracy of the eigendecomposition, we used three different metrics. First, we compared the eigenvalues produced by our `eigsym()` function to those output by Armadillo, using the maximum absolute difference between corresponding sorted eigenvalues to quantify numerical error. Second, we evaluated the orthogonality of the computed eigenvectors by measuring how closely the eigenvector matrix V satisfied $V^T V = I$. Finally, we verified the correctness of the full eigendecomposition by checking how well the relation $AV \approx V\Lambda$ was satisfied, where Λ is the diagonal matrix of eigenvalues.

In addition to the randomized test cases, several edge cases were tested to validate MatrixLibrary for these scenarios. These cases included 1 x 1 matrices, for which the eigenvalue problem is trivial, and diagonal symmetric matrices, where the eigenvalues are known exactly and the eigenvectors are simple unit vectors along each coordinate axis. These tests help to confirm that the implementation behaves correctly and that errors observed for larger matrices arise from accumulation of small numerical errors rather than from logical flaws in the algorithm itself.

As shown in *Figure 3*, the numerical error in all three metrics gradually increases as the matrix size increases. This is expected and primarily due to the accumulation of floating-point rounding errors over many iterative operations in the Householder-QL algorithm pipeline *(Press et al.)*. Of the three metrics, the reconstruction error is expected to be the largest because it involves forming the product VDV^T, which compounds the error from both the eigenvalues and eigenvectors, as well as from additional floating-point operations from matrix multiplication (*Trefethen and Bau)*. Armadillo uses highly optimized and numerically stable LAPACK routines to help avoid this accumulation of error, which, for the scope of this project, was not feasible to implement in MatrixLibrary. Despite this, the results demonstrate that the `eigsym()` implementation is accurate and numerically stable for small to moderately sized matrices.

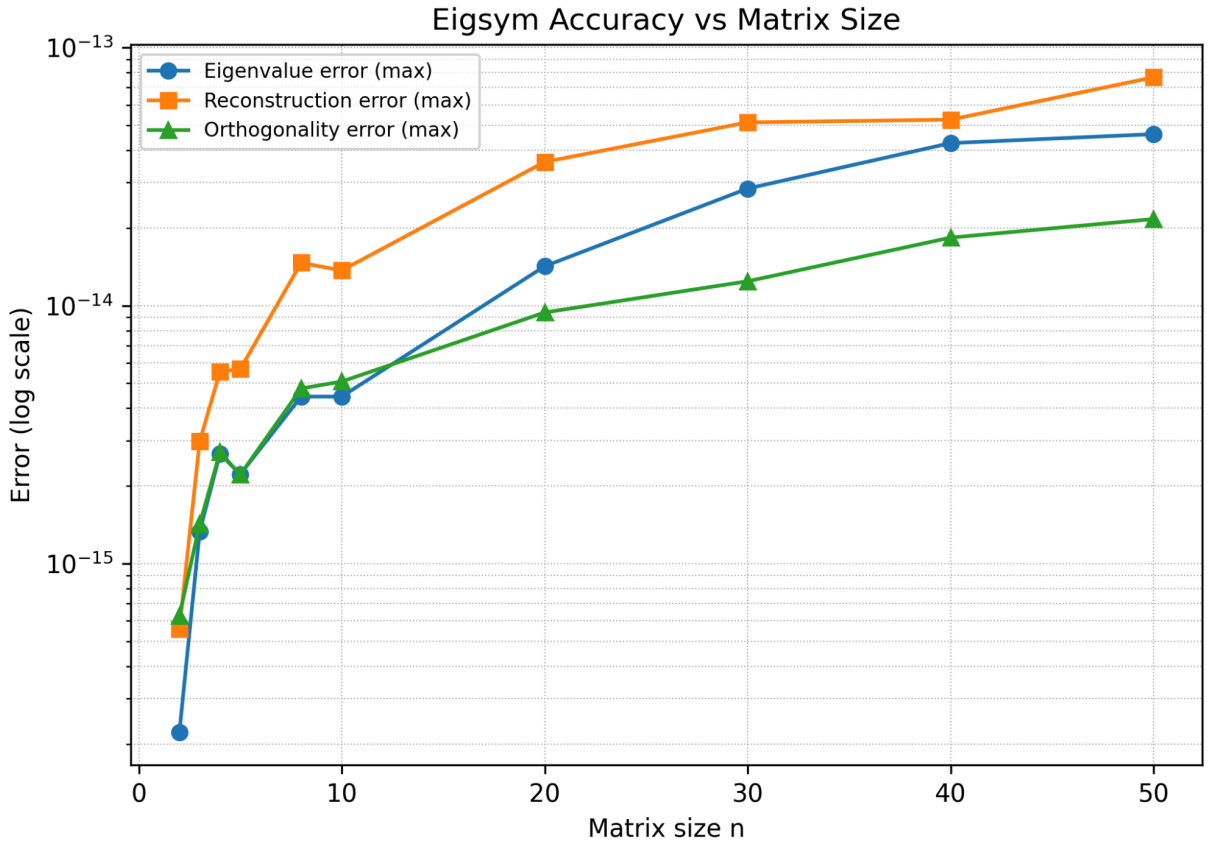

Figure 3. Accuracy of the MatrixLibrary eigensolver (eigsym()) as a function of matrix size for randomly generated symmetric matrices. The figure shows the maximum observed eigenvalue error, reconstruction error, and orthogonality error, measured relative to Armadillo's eig_sym() routine. Errors are plotted on a logarithmic scale and increase gradually with matrix size due to the accumulation of floating-point rounding errors in the Householder–QL algorithm pipeline.

Conclusion

MatrixLibrary achieves all of its objectives as a linear algebra library with sufficient functionality to completely replace Armadillo in Homework 3 for CHEM 279. This library is packaged in a modular and self-contained way such that it is distributable to other users. The MatrixLibrary was verified to be a successful drop-in replacement for Armadillo in Homework 3, the standard output matches the expected results and all of the test cases pass. MatrixLibrary contains algorithms that balance computational cost and code complexity while maintaining accuracy at the level of machine precision. It was compared to Armadillo for both accuracy and runtime, and found that, as expected, Armadillo outperforms the MatrixLibrary.

Areas for Future Research and Improvement

As discussed above, there are various areas for improvement in the MatrixLibrary. A logical next step would be to implement a branch of the `eigsym()` function that handles non-symmetric matrices so that the eigendecomposition functionality of this library isn't limited to symmetric matrices. Research into using the highly optimized routines in LAPACK could also provide significant performance improvements for existing functionalities and could be used in expanding the linear algebra capabilities beyond basic operations and eigendecomposition. Another area for possible improvement would be using more advanced convergence criteria or shifting strategies could improve numerical stability for larger matrix sizes. Further performance improvements could be achieved through techniques such as template unrolling and other compile-time optimizations.

Supporting Information

The complete source code for the MatrixLibrary project, including the C++ implementation, build system (CMake), google tests, and benchmarking scripts, is publicly available on GitHub at: https://github.com/trishakholiya/MatrixLibrary

Contributions

Both participants commented code, worked on matrix functionality, wrote associated Library report, and contributed to the repository as needed to ensure MatrixLibrary functionality worked as needed.

Trisha Kholiya:
Contributed to the development of the Matrix library, including an even divide of basic Matrix operations (addition, subtraction, constructors, saving Matrix files). Worked on optimizing code for performance and implementing changes to improve benchmarking results across functions. Created CMakeBuild System and ensured that library was downloadable by external users. Worked on the benchmarking framework using Google Benchmark.

Juliet James:
Contributed to MatrixLibrary operations (transpose function, constructors). Worked on researching and implementing the eigensolver pipeline, and benchmarking the accuracy of the output eigenvalues and eigenvectors. Created a testing framework for validation of MatrixLibrary functionalities and accuracy comparison to Armadillo.

References

Goto, K., & van de Geijn, R. A. (2008). *Anatomy of High-Performance Matrix Multiplication.* ACM Transactions on Mathematical Software, 34(3), Article 12.

J. Fritz, M. Mesiti, and J. P. Thiele, “A concise guide to good practices for automated testing and documentation of Research Software”, *ECEASST*, vol. 83, Feb. 2025.

L. N. Trefethen and D. Bau III, *Numerical Linear Algebra*, SIAM, 1997, Lectures 10–11 and 27.

McCalpin, J. D. *Memory Bandwidth and Machine Balance in Current High Performance Computers*. IEEE Technical Committee on Computer Architecture (TCCA) Newsletter, 1995.

Sanderson et al., (2016), Armadillo: a template-based C++ library for linear algebra, Journal of Open Source Software, 1(2), 26, doi:10.21105/joss.00026

Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, Haddock SH, Huff KD, Mitchell IM, Plumbley MD, Waugh B, White EP, Wilson P. Best practices for scientific computing. PLoS Biol. 2014 Jan;12(1):e1001745. doi: 10.1371/journal.pbio.1001745. Epub 2014 Jan 7. PMID: 24415924; PMCID: PMC3886731.

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, 3rd ed., Cambridge University Press, 2007, Ch. 11.