Advanced Programming Group Project

Matrix Solver Report

Akhil Mohan, Ben Walsh, Nikhil Kumar

Github Repository

January 28, 2022

Code Design

Interface

The interface is designed to provide the user with a comprehensive set of options to get to the solution, while remaining quick and easy to use. The matrices in the software are stored on the heap to allow access across different parts of the interface.

Matrix Class

The Matrix class was implemented to provide a templated interface that allows solving a system of linear equations on a dense matrix. The class uses a smart pointer to store the values stored in the Matrix, allowing for easier memory management. The class provides a jacobi solver for a diagonally dominant matrix and a Cholesky solver for a symmetric positive-definite (SPD) matrix. It also provides Matrix-Vector multiplication.

CSRMatrix Class

The CSRMatrix class, a sub-class of Matrix was implemented to store a sparse matrix in Compressed Sparse Row (CSR) format. It provides the same functionality as the Matrix class, short of the Cholesky solver. Instead, for SPD matrices, the CSR Matrix class uses a Conjugate Gradient solver.

General Design Decisions

Our choice to template both matrix classes needs justification. For one, it only allows numeric data types. Another issue can be that a linear system with integer LHS and RHS might not have an integer solution. Thus, while it is templated, we strongly urge any users to only use matrices of type float or double, or of a numeric data type class that they implement.

The choice to use inheritance is also an intuitive one, as the functionality provided in each class has significant overlap, but it required some clever usage of the public class members in the Matrix class. The use of smart pointers made memory management easier but led to strange edge cases caught by our test suite, so we had to use deep copies in certain places.

Software Implementation and Usage

Implementation

There is a `main.cpp` file that shows an example of how to use the library, and test suites in the `test/` folder. We also have a benchmarking tool (`test/benchmark.cpp`) and a tool to trigger the interface (`src/ladsSolverInteractive.cpp`). We used a Makefile to compile across 3 operating systems - Windows (through WSL), MacOS, and Ubuntu 18.04. We also used Github Actions to ensure the project would build and run on a remote Linux system.

Usage

Firstly, the user is greeted with a welcome message, followed by a question to determine whether they are solving a dense or sparse style matrix. If the user chooses to solve for a dense matrix, they choose to load a matrix from file by providing a filename, or input it

manually. In the latter case, they will be asked for the dimensions of the matrix, as well as the value of each element in the matrix. In the case they choose to solve for a sparse matrix, they will only be given the option to input a text file.

If the user passes incompatible inputs, at any point throughout the interface, they will be asked to restate their inputs.

Next, the user selects the specific solver they would like to use, to give the user full control of the software. The linear solver's purpose is to solve the problem $Ax = b$, so the user must manually input the elements for vector, b in order to solve for x. The solver outputs the solution vector x to the screen and exits, after freeing up any memory it used.

Benchmarks

We now explore how the implemented solvers behave with increasingly large matrix inputs. Matrices are created based on the limitations for the solver (e.g Diagonally Dominant for Jacobi, SPD for Cholesky etc.). These matrix inputs were created via a python script, found in the `tools/` directory of the repository. Benchmarks were run using GitHub actions on a remote Linux system, thus eliminating variables like other programs running on the system simultaneously. The 1000×1000 benchmark was run using the exact same matrix (80% sparse) for all solvers, but other benchmarks varied in this regard.

Table 1: Time taken (s) for linear solve

Matrix Dimensions	Dense Cholesky	Dense Jacobi	Sparse Jacobi	Sparse CG
10×10	4.3301e-5	3.610e-5	3.830e-5s	8.800e-6
20×20	7.080e-5	6.320e-5	9.310e-5	3.160e-5
40×40	4.795e-4	1.626e-4	5.898e-4	2.204e-4
100×100	6.326e-3	7.234e-4	3.320e-3	1.552e-3
200×200	4.771e-2	2.404e-3	1.860e-2	1.275e-2
1000×1000	5.573	4.918e-1	4.283	1.472

These results show that the Cholesky algorithm is $O(N^3)$ as evidenced by the run-times between $N = 100$ and $N = 1000$. The sparse Jacobi is surprisingly slower than our dense Jacobi. They were mostly tested on different matrices, but even on the 1000×1000 matrix, the dense Jacobi outdid our sparse implementation. We think that the dense Jacobi method's performance owes something to the compiler optimizations of its inner loops, and perhaps to its better cache performance.

The run-time comparisons would imply that our iterative solvers are better, performance-wise, but this comes at the cost of accuracy. The iterative methods were given an L^2 norm tolerance of 1e-6 as a stopping condition, whereas the Cholesky hits an accuracy closer to machine epsilon. Both dense methods could benefit from BLAS/LAPACK calls that would make things faster, but our implementations aimed for simple for-loops that could benefit from compiler optimization. This, however is something that could be improved in future revisions.

Solver Methods - Analysis

Dense Jacobi Solver

The Jacobi Solver within our matrix class aims to iteratively solve diagonally-dominant matrices. The solution iteration continues until a certain convergence is reached or a maximum number of iterations is reached. By default, the maximum number of iterations is 1000. The convergence tolerance is an argument to the function but the UI uses a default of 1e-6. Considerations on run-time are the tolerance given to the function and the initial guess passed - these can impact the results. The run-time of the Jacobi algorithm is largely determined by the number of iterations to convergence, with the main cost of each iteration being equivalent to a Matrix-Vector product.

Although the Jacobi Solver only guarantees convergence for diagonally dominant matrices, and matrices with spectral radius less than 1, the solver can sometimes converge on matrices that don't meet the criteria. This limitation is less strict than other solvers, making the Jacobi method a versatile solver implementation.

Dense Cholesky Solver

The Cholesky method performs a decomposition of a matrix object into a lower triangular, and lower triangular transpose matrix. The solver requires a matrix to be SPD, and takes a RHS vector b as an argument to the function. In contrast to the Jacobi method, Cholesky solves decompose the initial matrix into a matrix product LL^T where L is lower triangular which can then be solved efficiently. This gives us a run-time complexity proportional to the size of the matrix ($O(N^3)$) instead of to the convergence criterion.

The Cholesky decomposition only works for an SPD matrix, but is very efficient, operations-wise for this kind of matrix. Its accuracy for SPD matrices also hits the machine limit, losing accuracy only due to numerical cancellation. Due to some implementation differences, and potentially compiler optimizations, the benchmarks do not show the full power of the algorithm.

Sparse Jacobi Solver

For a sparse implementation of the Jacobi Method, the advantages and disadvantages of the dense Jacobi Solver still apply. Instead, we discuss the sparse-specific changes to the Sparse Jacobi Solver. The dense Jacobi solver relies on a loop that performs essentially a dot product between a row of the matrix and the RHS vector b. In the CSR Format, we skip all zero entries in each row of A, making for a more efficient implementation in theory. Our benchmarks did not corroborate this, and we think a primary reason for that is the CSR Matrix data structure that needs to loop over 3 different arrays (row-index, column-index, and values) while accessing elements. In theory this could hurt cache performance even while accessing sequential values, as chunks of each array are probably loaded into different caches. This problem seemed to only grow with scale, so between this cache issue and the potential for compilers to optimize the dense solver, we think this loss of performance is understood. That being said, the sparse Jacobi solver is efficient in terms of the number of operations it executes (only looking at non-zeros) and allows for a space-efficient means to store a large sparse matrix. Thus, despite the loss in performance, this solver still has its niche.

Sparse Conjugate Gradient Solver

The Conjugate Gradient implementation in our code is a sparse method that solves SPD matrices using an iterative method. The method has input arguments RHS Vector b, convergence tolerance, and an initial guess vector. Similar to the Jacobi Solver, the default tolerance when using this via the interface is 1e-6.

The Conjugate Gradient implementation returns a solution within N iterations, where N is the number of rows of the matrix, and this is generally preferred to the Jacobi method which can fail to converge for quite some time. The idea of the method and the fast convergence is a consequence of Krylov subspaces and orthogonality. The run-time in each iteration is dominated by a matrix-vector product, which should be fairly efficient using the CSR format, for reasons similar to the sparse Jacobi method's efficiency. The run-time is bounded above by the N iterations needed (ignoring numerical cancellation) and the cost of a Matrix-Vector product which is $O(nnz)$ where nnz is the number of non-zeros in the matrix. Thus, the worst-case complexity of this algorithm is $O(N.nnz)$.

Contributions

Overall, we did pair programming for the vast majority of the code in the project, with insights being shared across all 3 members. Broadly, the focus was Akhil Mohan - Sparse Matrix Solvers, Continuous Integration, Git Version Control. Nikhil Kumar - Jacobi Solver, Report Writing. Ben Walsh - Interface, File I/O. Any contributions to the code in the `src/` directory were paired with the appropriate test found in the `test/` directory. We had some difficulty testing the interface, but we are able to see if it works correctly (white-box testing) by using a file `data/interfaceTest.txt` to simulate inputs from `cin`, and trigger a full end-to-end run of the interface. This can be seen from any of the later Github Actions workflows.

Work on the code-base was driven by version control, with each member working on an independent branch whose changes could then be merged into `main` via a pull request. We chose to squash and merge pull requests, and thus, the low commit count on the `main` branch of the repository does not reflect the real number of commits made during development. The development process was test-driven - tests were always added before any implementations were created, allowing for robust validation.

Conclusions

The Lads Solver™(patent pending) is an honest attempt at a well-documented, easy-to-use linear solver library. It may not have the performance required to be viable in the real world, but it was made with love and care. We hope it sees some use in the coming years.