TEAM: UNIQUE POINTERS

TEAM MEMBERS:
*** MICHEL SFEIR * HANCHAO CHEN * FAZAL KHAN**

Installation Instructions

The extensive installation instructions can be found in the **Readme** section of the github repository.

Software Development Framework

For this project the team utilized a test-driven development approach to building the Solver library. Before we began writing any code, we first created tests that described how the functions should behave and the expected output.

The tests used a C++ library called Catch2, which simply requires importing a header file to make use of the testing framework. In the **test/** folder, we have two files **catch.hpp** which is the header and **test.cpp**, which contains all of the unit tests. Each time we made any changes to the codebase, we ran the unit tests to see if they threw errors, and iteratively debugged any problems.

Testing was by far the most important part of our software development cycle, as it gave the team a unique "view of truth", and so we could independently work around the codebase in a way that was consistent.

In order to run the tests, you can simply go to the **test.cpp** and uncomment the **RUN_ALL_TESTS** preprocessor directive.

Structure of the Library

The library is split into 3 main sections (and corresponding files):

- Matrix: comprised of Matrix.h and Matrix.cpp
- CSRMatrix: comprised of CSRMatrix.h and CSRMatrix.cpp
- Solver: comprised of Solver.h and Solver.cpp

Matrix Class

The Matrix library can generate matrices and output to various formats. In addition, it has operations defined on the Matrix objects. These include matrix-matrix multiplication and matrix-vector multiplication. We also implemented BLAS calls in various parts of the algorithms, such as matrix-matrix multiplication to speed up performance on large matrices.

CSRMatrix Class

CSRMatrix is the part of the library that operates over CSR format matrices, which is a storage format more optimized for sparse structure matrices. As we will later discuss, performance is much better for the CSR format as it reduces the amount of operations the algorithm has to do on a matrix by simply representing it in a more compact format.

The most complex function in the CSRMatrix library is the matrix-matrix multiplication. This function carries out sparse matrix multiplication and is comprised of two parts:

- Symbolic matrix-matrix multiplication, which just derives the rows positions and column indices of the non-zero values. This function can be used to generate the sparsity structure for a given product, allowing the user to chain matrix multiplications together, where the matrices have the same sparsity structure. In essence this function is an optimization.
- Numeric matrix-matrix multiplication looks at the sparsity structure generated by symbolic matrix-matrix multiplication and generates what the actual number should be in each position.

Further Uses for Matrix-Matrix Multiplication

Although we did not have time, we could have utilized Sparse matrix-matrix multiplication to generate solutions to block linear systems. For example, instead of solving for a single right-hand vector b, where Ax = b, we could instead have a matrices for both x and b, and so solve multiple systems which use A as the left hand side matrix. This would mean that wherever we currently have a matrix-vector multiplication, we would instead have a matrix-matrix multiplication in the solvers.

Solver Class

Finally, the Solver class actually implements the various different solvers. These include:

- Jacobi (for dense and sparse)
- Gauss-Seidel (for dense and sparse)
- Conjugate Gradient (for dense and sparse)
- LU (for dense only)
- Gaussian Elimination (for dense only)

The solver class also implements numerous different helper functions to construct the various algorithms. This is an example of *modularity*.

BLAS Optimization

We also used the Open BLAS library to implement some performance enhancements. The performance enhancements can be turned on using the **USE_BLAS** preprocessor directive.

As can be seen form the tables below, the BLAS library increases the performance of the Conjugate Gradient (CG) algorithm considerably. This is most likely as the CG algorithm uses the BLAS optimizations most extensively. The other algorithms do not seem to perform much better, most likely due to their lower use of the optimizations produced by BLAS.

Below is a table showing the performance of the different algorithms (execution time in ms) with and without BLAS optimizations.

Without Blas:

time	**100**	**250**	**500**	**750**	**1000**	**2500**	**5000**	**7500**
Jacobi	9.433	61.519	170.704	345.559	635.819	4817.02	17632.6	41647.4
Gauss-Seidel	3.201	17.251	53.806	117.011	228.516	1550.19	5989.15	13756.2
CG	6.44	27.016	49.117	100.439	178.06	1096.98	3765.52	8330.62
LU	4.756	49.072	245.538	677.191	1564.7	24714.4	180405	576326
Gaussian	2.528	40.488	241.25	566.886	1480.4	21026.7	172160	572453

With Blas:

time	**100**	**250**	**500**	**750**	**1000**	**2500**	**5000**	**7500**
Jacobi	10.3	58.916	266.973	487.474	993.914	5614	23110.3	53952.3
Gauss-Seidel	3.015	19.718	92.207	218.93	365.601	2267.2	9020.08	22163.5
CG	3.814	3.786	14.145	39.053	67.093	305.119	1089.18	2342.77
LU	2.647	28.202	294.846	792.71	2163.57	21977.5	170742	594971
Gaussian	2.031	30.668	169.934	934.317	1628.23	21069.7	185635	586503

Matrices Adjustment

Based on the definition of positive definite matrix, there could be a special case that a positive definite matrix whose diagonal elements have one or more zero values. It is inevitable for large complex matrices. And it also causes error for Jacobi method, Gauss Seidel method and normal LU method. In order to avoid this error, we work out an “sort_mat” function to help users adjust their matrices before using linear solvers.

Function & Explanation:

We change the input matrix by using row elementary operations and get an output matrix whose diagonal elements are all non-zero (We can always do that since the input matrices are positive definite). First, we need to check all columns to find out if there are columns which have only one non-zero element. If we find one or more, we save the non-zero elements’ position, and then use row elementary operations to put them in right diagonal position. The next step is to deal with the rest of columns, we loop over the elements of each row (drop the elements in "unique" columns) and choose one element which has largest dominance in its row. Then we swap its row to put it in right diagonal position again. That's what we need to do in one iteration. After several iterations, all the columns have been checked and the matrix becomes a revised one.