bad bois

Vagif Aliyev, Nikolas Vornehm, Sotiris Gkoulimaris

February 2, 2020

1 Introduction

For this assignment we were asked to implement algorithms that solve the linear system $Ab = x$. The program itself used and expanded upon the Matrix.cpp and Matrix.h libraries that were partially constructed in-class.

Our solution expands upon the Matrix class by defining the CSRMatrix, BandMatrix and SymMatrix subclasses. The solvers we decided to implement are:

	Dense	CSR Sparse	Banded	Symmetric
Inverse	x			
Gaussian Elim	x			
LU Solver	x			
Conjugate Gradient	x	x	x	x
SOR	x	x	x	x
Chebyshev Iterative	x	x	x	x
Multi CG	x			
Multi SOR	x			

2 Class Specifications

All classes have a similar class structure inhereted by the Matrix base class as well as functions that are necessary to solve linear systems (such as matVecMult).

2.1 Matrix.cpp

Matrix.cpp is the base class of our library. It holds the private variables of rows, cols and a 'values' unique pointer array that stores the values of our dense matrix (surpriiiise!). A number of helper functions have been implemented to facilitate the functionality of our class.

2.1.1 Helper functions and Operations

Most of the implemented functions are self-explanatory. Note that appart from printMatrix, every other function is limited to the base class.

matVecMult() and matMatMult() perform matrix-vector and matrix-matrix multiplications. matVecMult() has a loop than runs N times and a nested loop that runs N times as well; thus, the complexity of this algorithm is $O(N^2)$. matMatMult() has three loops that run N times, two of which are nested; that results in a time complexity of $O(N^3)$.

luDecomposition() performs the LU decomposition on our matrix. The algorithm iterates through N rows, for the first row it makes 2N(N-1) operations, for the second 2(N-1)(N-2) and so on; thus, the overall complexity of our algorithm is of the order $O(N^3)$.

generateSPD() is a helper function that generates Symmetric Possitive Definite Matrices. It receives an integer as input (the type of SPD matrix to be produced - check Matrix.h). For every matrix type, the algorithm loops over all rows and columns using nested for loops; thus, the time complexity of our algorithm is $O(N^2)$.

2.2 CSRMatrix.cpp

CSRMatrix is derived from Matrix and corresponds to a sparse matrix stored in a CSR format. CSR uses the values array to store the non-zero values as well as two extra arrays to store row and column positions. Using this format we have the advantage of saving up space and doing less operations when we compute matrix-vector and matrix-matrix multiplications. One drawback would be that iterating over the elements of the sparse matrix accsess all three different arrays; memory jumps can be quite expensive and thus hindering performance.

2.2.1 Helper functions and Operations

The self-explanatory functions are: dense2csr(), csr2dense(). compareMatResInner() and compareMatResOut() are used in tandem with the sort algorithm in matMatMult.

matVecMult() is similar to the base class. The main difference is that instead of iterating over all rows and columns, we take advantage of the CSR format and iterate over row position, using those indexes to access the column elements. Thus, the time complexity of this algorithm is $O(N^2)$

In order to compute matMatMult() we decided to compute the individual multiplication results, together with row and column indexes in a vector of matRes structs. To elaborate, instead of adding the values during the for loop, we store them at each iteration and end up with a list of all non zero values with their position indexes (on the output matrix). To add them together, first sort them according to column (compareMatResInner()) and then row (compareMatResOut()). Then, we iterate through the sorted matRes elements and populate the output matrix's values, row position and column index). The time complexity of this algorithm $O(N^3)$.

2.3 BandMatrix.cpp

BandMatrix is the banded form of a dense matrix. BandMatrix stores only the elements in the bands of the dense matrix. Essentially, from an NxN dense matrix, we get a NxM matrix, where M is the number of bands. The structure of the class is changed accordingly. With this format, we can greatly decrease the ammount of memory required to store a given matrix and also speed up our computations, since we will only iterate over the bands instead of all the columns.

2.3.1 Helper functions and Operations

As with our Matrix and CSR class, a number of helper and operation functions have been implemented. We defined the functions dense2band() and band2dense() to transform a matrix from dense to banded form and vice-versa.

The function matVecMult() performs the same operation as the base function. This time however, the computation is much faster, since instead of iterating over all the columns, we only need to iterate through bands. As a result, we can have significant speed-up since the time complexity of this function is now O(MN).

2.4 Symmetric.cpp

A symmetric matrix is a square matrix that is equal to its transpose, hence halving the storage required.

2.4.1 Helper functions and Operations

As usual, symm2band() and symm2dense() are used to transform a matrix from dense to banded form and vice-versa. A MatVecMult has also been implemented for use by the Solvers.

3 Linear Solvers

As mentioned in the introduction, a number of linear solvers have been implemented, all of which are used for a defualt dense matrix.

3.1 Inverse Solver

This is the most basic solver we implemented. We use the inverse function to compute the inverse of input matrix and then we perform a matVecMult operation between the inverse and the input vector. This algorithm uses the determinant(), coFactor() and adjugate() helper functions and as a result has a pretty high time complexity $O(N^4)$.

3.2 Gaussian Elimination

This is also a fairly basic solver. The algorithm we used is essentially the one we implemented in ACSE-3. It computes the upper triangular matrix of our input matrix and performs back substitution to find the solution to our linear system. The complexity of this algorithm will depend on the individual complexities of the our two functions (since they are called sequentially). The upper triangle function has a time complexity of $O(N^3)$. back substitution has a time complexity of $O(N^2)$. As a result, the overall time complexity of our Gauss Elimination is $O(N^3)$.

3.3 LU Solver

The LU solver is also the same algorithm that we were taught in ACSE-3. It calculates the Lower and Upper decompositions of our input matrix, then it performs a forward substitution between the Lower matrix and the input vector, followed by a backward substitution giving the result to the linear system. The time complexity for the LU decomposition is O(N^3) the forward and back substitutions have a time complexity of $O(N^2)$. Thus, the overall time complexity of this solver is $O(N^3)$.

3.4 Conjugate Gradient

The Conjugate Gradient method is here implemented as an iterative solver. The algorithm used here is found on `https://en.wikipedia.org/wiki/Conjugate_gradient_method`. It is capable of computing for all the matrix types implemented in this library. The conjugate gradient takes adjantage of the Symmetric Positive Definite Matrices by exploiting its properties.

3.5 SOR

The Successive Over-Relaxation (SOR) method is an iterative solver that make use of an relaxation factor omega given by the user. Omega can take values between 0 and 2 (0 < omega 2). When omega is set to 1 the SOR is equivalent to the Gauss-Seidel (and Jacobi). The algorithm used in this library is extracted from: `https://en.wikipedia.org/wiki/Successive_over-relaxation`.

3.6 Chebyshev

Lastly, implemententend a version of Chebyshev found in `https://en.wikipedia.org/wiki/Chebyshev_iteration`. This iterative solver takes advantage of extra information (namely the upper and lower estimate of eigenvalues) provided by the user to avoid computation of inner products (as they can be quite expensive). Our algorithm's performance depends on the number of iterations, the input information given by the user and the performane of matVecMult. Assuming that the algorithm converges given the upper limit for iterations, the time complexity of Chebyshev is identical to matVecMult.

3.7 Multi Linear Solvers

Versions of SOR and CG were implemented to solve multiple vectors in matrix form. Essentially we solve multile linear systems were the b vectors are stacked as a matrix.