GMRES Project

Applied Linear Algebra

Duc Phan

March 12, 2019

**Abstract**

This project objective is to create a set of tools to that mainly help with operating and manipulating on sparse matrices in compressed sparse row (CSR) format. The library I chose to implement cover most of the basic operations, which will be discussed later in this paper, for *Vectors*, *Dense Matrices* and *CSR Matrices*. The focus of this project and paper is the Generalized Minimal Residual (GMRES) method/algorithm. This paper is an overview of the tool set mentioned above, the implementation of GMRES and some statistics produced by GMRES.

1. **Introduction**

This project is implemented mostly in Java for the *Vector, Dense/CSR Matrix* operations and NodeJS for data and stats graphing. In this paper, I’ll introduce shortly pseudocode for some algorithms I found interesting to keep the paper short.

Some libraries I used in my program are:

* Java:

|  |
| --- |
| //For parsing String into json  import com.google.gson.Gson;  //IO handler  import java.io.File;  import java.io.FileNotFoundException;  import java.io.PrintStream;  import java.util.Scanner;  //Number handler - This will support number with a lot of decimals  import java.math.BigDecimal;  import java.math.MathContext;  import java.math.RoundingMode;  //Database  import java.util.LinkedList; |

* NodeJS:

|  |
| --- |
| require('fs'); //For file reading/writing  require('path'); //For file searching  require('plotly'); //For graphing |

The code and some sample results can be found at my GitHub repository:

<https://github.com/ptmdmusique/Vector_Dense-CSR-Matrix_Operations>

1. **Acceptable Inputs**

The program will be able to take and parse input of string type in multiple format:

* A string of numbers separated by single space for different columns and newline character (‘\n’) for different rows:

A vector or matrix can be constructed directly using:

1. the object’s constructor *Vector(String input)* or *Matrix(String input)* or *CSRMatrix(String input)*.
2. the *TakeInput(String input)* method.

Example:

|  |
| --- |
| //This will create a vector with entries 1, 2, 3, 4, 5  Vector myVector = new Vector("1 2 3 4 5");  /\* This will create a 2x4 matrix:  1 2 3 4  5 6 7 8  (extra spaces after the last and before the first numbers of each row can lead to bugs!)  \*/  Matrix myMatrix = new Matrix("1 2 3 4\n5 6 7 8");  CSRMatrix myCSRMatrix = new CSRMatrix("1 2 3 4\n5 6 7 8"); |
|  |

* [Matrix Market Exchange Format:](https://math.nist.gov/MatrixMarket/formats.html#mtx)

Before constructing a vector/matrix, a file from Matrix Market Exchange must go through a parsing process. Two support methods for this purpose (can be found in *Main*) are:

1. *static String ReadCSRFromFile(String fileName)* or
2. *static String ReadVectorFromFile(String fileName)*

The functions will then parse the input from file into a String with which we can constructed a Vector, Matrix or CSRMatrix using the method mentioned above.

The default folder to store the input file is: *./bigMatricess*

The path can be changed via *static String inputPath* declared in Main.

1. **Data structures:**
2. **General**:

All data will be stored using [*BigDecimal*](https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html) data type with the default precision of 50. I chose to use this data structure since a lot of files from Matrix Market contain really small number, which require high precision data structure to store and manipulate.

[*BigDecimal*](https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html) library from Java is an immutable and arbitrary-precision signed decimal numbers, meaning it can hold, theoretically, infinite number of decimals. However, it is a huge trade-off between performance and storage.

1. **Vector**:

Vector class will store its data in a *linear array* of *BigDecimal.* Vector class are both row and column major. With the correct context, the program will be able to detect automatically whether the specified Vector is a column vector or a row vector.

For example: *myVector.InnerProduct(Vector parm)* will return the inner product of myVector and parm correctly without getting transpose of any of them.

1. **Dense** **Matrix**:

Matrix class will store its data in a *linear array* of row *Vector*.

1. **CSR** **Matrix**:

CSRMatrix class will store its data in a standard CSR from, which includes a *linear array* of *data,* a *linear array* of row info and a integer colSize which stores the number of column of the matrix.

The *data* array is an array of *Data* which includes the *BigDecimal data* (value of the entry) and *int col* (column of the entry).

1. **Operations:**

This toolkit provides functions for several basic vector and matrix operations:

1. **Vectors:**

* Vector *Add(BigDecimal parm)*: return a vector with its entries equal the sum of the old data and the parameter/input

|  |
| --- |
| //Example:  myVector.Add(BigDecimal.valueOf(5));  //will add all entries of myVector with 5 |

* *Vector* *Add(BigDecimal parm)*: add the original vector and the parm vector up and return the result.

|  |
| --- |
| //Example:  myVector.Add(anotherVector);  //will add myVector with another Vector |

* *Vector Scale(BigDecimal parm):* scale all entries of the current vector with parm and return the result.

|  |
| --- |
| //Example:  myVector.Scale(BigDecimal.valueOf(-1));  //will multiply all entries of myVector with -1 |

* *BigDecimal InnerProduct(Vector parm)*: calculate inner product of the two vector.

|  |
| --- |
| //Example:  myVector.InnerProduct(anotherVector);  //will calculate the inner product of myVector and anotherVector |

* *Matrix Multiply(Vector parm)*: multiply 2 vector with the original vector as column vector on the left and parm as the row vector on the right and return the result matrix.

|  |
| --- |
| //Example:  myVector.Multiply(anotherVector);  //will multiply myVector with anotherVector |

* *Vector Normalize()*: return a new vector which is the normalized version of the current vector.

|  |
| --- |
| //Example:  myVector.Normalize();  //will return a unit vector based on myVector |

* *void Copy(Vector parm)*: replace all entries with the data from parm.

|  |
| --- |
| Example:  myVector.Copy(anotherVector);  //will copy data from anotherVector to myVector |

* *boolean Equal(Vector parm)*: check if two vectors are equal.

|  |
| --- |
| //Example:  myVector.Equal(anotherVector);  //will return true if anotherVector has the same entries as myVector |

* *void TakeInput(String input):* process an input string and convert it to a corresponding vector

|  |
| --- |
| //Example:  myVector.TakeInput("1 2 3 4 5");  //will produce a vector with entries: 1, 2, 3, 4, 5 |

* *void Print():* print all the data of the current vector

|  |
| --- |
| //Example:  myVector.Print();  //will print myVector as a column vector |

1. **Matrix:**

* Matrix *Add(Matrix parm)*: return a matrix with its entries equal the sum of the old matrix and the parameter/input

|  |
| --- |
| //Example:  myMatrix.Add(newMatrix);  //will add all entries of the corresponding entries of newMatrix |

* *Matrix* *Multiply(Matrix parm)*: multiply 2 matrix and return the result. I used the naïve *O(n3)* algorithm.

|  |
| --- |
| //Example:  myMatrix.Multiply(anotherMatrix);  //will multiply myMatrix with anotherMatrix |

* *Matrix Multiply(Vector parm):* multiply the current matrix with a vector on the right side.

|  |
| --- |
| //Example:  myMatrix.Multiply(aVector);  //will multiply myMatrix with aVector (myMatrix \* aVector) |

* *Matrix AugmentVectorAtEnd(Vector parm)*: augment the specified Vector at the end of the matrix.

|  |
| --- |
| //Example:  myMatrix.AugmentVectorAtEnd(anotherVector);  //will augment anotherVector (column vector) at the end of myMatrix |

* *LinkedList<Matrix> LUFactorization():* return the L and U matrix from the result of LU Factorization. This method will be able to rotate the rows of the current matrix until it produces the a LU factor. However, if after trying all permutation and there is no result, the method will return null.   
  This method is not fully tested, especially with big or complicated matrices.

|  |
| --- |
| //Example:  myMatrix.LUFactorization();  //will return the LU factorization of myMatrix |

* *LinkedList<Matrix> QRFactorization()*: return the Q and R matrix from the result of QR Factorization. This method used the principle of Gram-Schmidt process to produce the correct QR factorization. This method supports any matrix.

|  |
| --- |
| //Example:  myMatrix.QRFactorization();  //will return the QR Factorization of myMatrix |

* *Vector BackwardSubstitution(Vector rhs)*: perform BackwardSubtition and return the result vector. This only works for upper triangular matrix.

|  |
| --- |
| //Example:  myMatrix.BackwardSubstitution(b);  //will perform backward substitution on myMatrix.x = b and return x |

* *void TakeInput(String input):* process an input string and convert it to a corresponding matrix.

|  |
| --- |
| //Example:  myMatrix.TakeInput("1 2\n3 4");  /\*will produce a 2x2 matrix  1 2  3 4  \*/ |

* *void Print():* print all the data of the current matrix

|  |
| --- |
| //Example:  myMatrix.Print();  //will print myMatrix row by row |

* *static boolean IsSymmetric(Matrix parm):* check if a matrix is symmetric or not.

|  |
| --- |
| //Example:  Matrix.IsSymmetric(myMatrix);  //will check for myMatrix symmetry. |

1. **CSRMatrix:**

* Some general note about the algorithms I used in CSRMatrix traversal and manipulation.  
  A lot of time I used a lookup array to travel the CSR’s data array column wise instead of row wise in the matrix. I called it *rowCurIndx*. The purpose is to help us avoid iterating through the array to find the next entry in the current row every single time.

For example: *rowCurIndx[curRow]* = 5 means the index, on the *data* array of the *CSRMatrix*, of the current entry we are talking a look at on row *curRow* is 5. So if we want to print the next entry in the same row the next time we come back to that row, we only need to get the next index from *rowCurIndx [curRow]* (which is 6) instead of traversing from the start index of the row.

This is a tradeoff between performance and space. However, the space we allocate for the array is only *Θ(numberOfRow)* for the entire matrix while the worst case of performance can reach up to *O(numberOfCol)* for each row traverse and *O(nnz)* for the entire array. Thus, I believe the lookup array is worth the tradeoff.

The pseudocode for initializing the lookup array can be:

|  |
| --- |
| //We have an array to keep track of the start index of each row in the //data array. I call it *row*  //Notice that if the start index of the current row is equals to //the start index of the next row then it means the current row //entries are all 0 (empty row)  int[] rowCurIndx = new int[totalRow];  **for** i in 0 to totalRow – 2 **do**  **if** (row[i] == row[i + 1]) **then**  rowCurIndx[i] = -1; //Empty row, set to -1  **else**  rowCurIndx[i] = row[i];//Not empty, set to the start index  **if** (row[row.length – 1] == data.length) **then**  rowCurIndx[row.length – 1] = -1; //Last row is empty  **else**  rowCurIndx[row.length – 1] = row[row.length – 1]; |
|  |

* Matrix GetMatrixForm*()*: return a matrix with its entries are the entries of the CSRMatrix.

|  |
| --- |
| //Example:  myMatrix.GetMatrixForm();  //will copy myMatrix and return it in regular matrix form. |

* *CSRMatrix* *GetTranspose()*: return the transpose of the current CSRMatrix.

In this method, I used the lookup array technique I mentioned above. Let’s look at an example to see how my algorithm work in this.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Row | 0 | | 1 | | | | 2 | |
| Data | 124 | 516 | 575 | 513 | 654 | 847 | 514 | 321 |
| Column | 0 | 2 | 0 | 1 | 2 | 3 | 1 | 3 |

Since we are trying to construct a transpose, a­i,j will become aj,i in the new matrix. Notice that, from left to right, the row of each data is arranged in an increasing order. Thus, if I traverse column (which will become row of the new matrix) wise (from column 0 to 3) then I’ll be able to put each entry in same column of the old matrix in the new *data* array of the transpose matrix next to each other while their new columns (which is the old row) is in increasing order.

For example: Let’s look at the entries with column 0 and where they are in the transpose matrix.

Table Old Matrix

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Row | 0 | | 1 | | | | 2 | |
| Data | 124 | 516 | 575 | 513 | 654 | 847 | 514 | 321 |
| Column | 0 | 2 | 0 | 1 | 2 | 3 | 1 | 3 |

Table Transpose Matrix

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Row | 0 | | 1 | | 2 | | 3 | |
| Data | 124 | 575 | 513 | 514 | 516 | 654 | 847 | 321 |
| Column | 0 | 1 | 1 | 2 | 0 | 1 | 1 | 2 |

After my algorithm finishes, the two marked entries are in the same row and their columns are arranged in an increasing fashion. However, to travel column by column, because the *data* array is arranged in a row-wise fashion, either I have to travel from the begin index (in the *data* array) of the current row to the end index to search for the pick up the last entry where I left off, or I need some kind of table to keep track of the last index (which also gives us the entry’s value and its column) in the *data* array of the current row. Thus, the lookup array will come into use. With this method, I’ll be able to travel from the first column to the last column much faster, and every single time I go back to the same row to search for the entry with the next column, I can easily pick up where I left off.

The time complexity of this method is *Θ(nnz)* while the space complexity is *Θ(numberOfRow)*.

The pseudocode for the algorithm can be

|  |
| --- |
| //Allocate the space for new transpose matrix  //Put the data of the current matrix in the correct spot and return the //result  int[] rowCurIndx;  int filledElement = 0; //Keep track of number of filled entries  int curCol = 0; //Keep track of the column we are filling  InitializeLookup(rowCurIndx); //Initialize the lookup array  **while** (filledElement < nnz) **do**  //Traverse row to row to search for the right column  **for** curRow **in** 0 **to** numberOfRow – 1 **do**  //We need to make sure we are filling using correct data  **if** data[rowCurIndx[curRow]].col == curCol **then**  transpose.data[filledElement] = data[rowCurIndx[curRow]].data;  filledElement++;  //Update the rowCurIndx  **if** EndOfRow(rowCurIndx[curRow])  rowCurIndx[curRow] = -1;  **else**  rowCurIndx[curRow]++; |
| curCol++; //Move on to the next column |

|  |
| --- |
| //Example of usage:  myMatrix.GetTranpose();  //will return the transpose of the myMatrix |

* *Vector Multiply(Vector parm):* multiply the current matrix with a vector on the right side.

In this method, I used the technique mentioned in class, where, foreach each entry in the every row, I will use its column to access the correct entry of the vector then calculate the sum of their products and ignore the rest of the vector (since their corresponding matrix entries are 0, and it is not useful to multiply a number with 0). For example:

Table CSR Matrix

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Row | 0 | | 1 | | | | 2 | |
| Data | 124 | 516 | 575 | 513 | 654 | 847 | 514 | 321 |
| Column | 0 | 2 | 0 | 1 | 2 | 3 | 1 | 3 |

Table Vector

|  |  |  |  |
| --- | --- | --- | --- |
| Data | 5 | 765 | 24 |
| Index | 0 | 1 | 2 |

We have result0,0 = 124 \* 5 + 516 \* 24 and we will ignore 765 in the vector since its corresponding entry in the CSR Matrix is 0.

This technique will ensure that there will be no unnecessary access and operation performed during the multiplication process.

The pseudocode of the algorithm can be:

|  |
| --- |
| //Input: the Vector parm to multiply with  //Output: the result product vector of this CSR and the parameter vector.  **for** curRow **in** 0 **to** numberOfRow **do**  temp = 0;  **for** i **in** row[curRow] **to** row[curRow + 1] **do**  temp = temp + data[i].data \* parm[data[i].col];  result[curRow] = temp; |

|  |
| --- |
| //Example of usage:  myMatrix.Multiply(aVector);  //will multiply myMatrix with aVector (myMatrix \* aVector) |

* *CSRMatrix Multiply(Matrix parm)*: multiply the matrix with another matrix on the right hand side if possible and return the result.

In this method, I used the same technique as the one in CSRMatrix times a vector, but instead of just with 1 column, it will go through all the combination instead. However, I first count the nnz of the result first before allocate the exact memory. I used the naïve algorithm for matrix multiplication, so the complexity should be *O(n3)*

|  |
| --- |
| //Example:  myMatrix.Multiply(aNormalMatrix);  //will augment anotherVector (column vector) at the end of myMatrix |

* *LinkedList<Matrix> LUFactorization():* return the L and U matrix from the result of LU Factorization. This method will be able to rotate the rows of the current matrix until it produces the a LU factor. However, if after trying all permutation and there is no result, the method will return null.   
  This method is not fully tested, especially with big or complicated matrices.

|  |
| --- |
| //Example:  myMatrix.LUFactorization();  //will return the LU factorization of myMatrix |

* *LinkedList<Matrix> QRFactorization()*: return the Q and R matrix from the result of QR Factorization. This method used the principle of Gram-Schmidt process to produce the correct QR factorization. This method supports any matrix.

|  |
| --- |
| //Example:  myMatrix.QRFactorization();  //will return the QR Factorization of myMatrix |

* *Vector BackwardSubstitution(Vector rhs)*: perform BackwardSubtition and return the result vector. This only works for upper triangular matrix.

|  |
| --- |
| //Example:  myMatrix.BackwardSubstitution(b);  //will perform backward substitution on myMatrix.x = b and return x |

* *void TakeInput(String input):* process an input string and convert it to a corresponding matrix.

|  |
| --- |
| //Example:  myMatrix.TakeInput("1 2\n3 4");  /\*will produce a 2x2 matrix  1 2  3 4  \*/ |

* *void Print():* print all the data of the current matrix

|  |
| --- |
| //Example:  myMatrix.Print();  //will print myMatrix row by row |

* *static boolean IsSymmetric(Matrix parm):* check if a matrix is symmetric or not.

|  |
| --- |
| //Example:  Matrix.IsSymmetric(myMatrix);  //will check for myMatrix symmetry. |