GMRES Project

Applied Linear Algebra

(Professor Panayot Vassilevski)

Duc Phan

March 12, 2019

Abstract

This project objective is to create a set of tools that mainly help with operating and manipulating on sparse matrices in the 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 toolset mentioned above, the implementation of GMRES and some statistics produced by GMRES process.

1. Introduction

This project is implemented mostly in Java for the *Vector, Dense/CSR Matrix* operations and NodeJS for data and stats graphing. I used IntelliJ IDEA for Java coding and Visual Studio for NodeJS coding. In this paper, I'll introduce shortly the pseudocode for some algorithms I found interesting.

All the tasks mentioned in the instruction paper will be discussed on section 5, 6 and 7 of this paper. What will be executed in each task comes with the instruction paper, so I will not mention about the object again.

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

2. Acceptable Inputs

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

- 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:

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 a file into a String with which we can construct a Vector, Matrix or CSRMatrix using the method mentioned above.

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

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

3. Data structures:

1) General:

All data will be stored using *BigDecimal* data type with the default precision of 40. I chose to use this data structure since a lot of files from Matrix Market contain a really small number, which require high precision data structure to store and manipulate.

BigDecimal library from Java is an immutable and arbitrary-precision signed decimal number, meaning it can hold, theoretically, an infinite number of decimals. However, it is a huge trade-off between performance and storage.

2) 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.

3) Dense Matrix:

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

4) 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 an 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).

4. Operations:

Side-Note: There are several helper functions that are defined in Main() as to help us test and print the result of Pre-task, Task One and Task Two. However, I'll not discuss them since they are not the focus of this paper.

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 the inner product of the two vectors.

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

- *Matrix Multiply(Vector parm)*: multiply 2 vectors with the original vector as a 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
```

2) 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 $O(n^3)$ 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 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 the 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 backward substitution and return the result vector. This only works for a 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.
```

3) 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 looking at on row *curRow* is 5. So, for instance, 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 works 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, $a_{i,j}$ will become $a_{j,i}$ in the new matrix. Notice that, from left to right, the row of each data is arranged in 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. Let's look at the entries with column 0 and where they are in the transpose matrix.

Table 1 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 2 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 $\Theta(nnz)$ while the space complexity is $\Theta(numberOfRow)$.

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, for each entry in 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 3 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 4 Vector

Data	5	765	24
Index	0	1	2

We have $result_{0,0}$ = 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 allocating the exact memory. I used the naïve algorithm for matrix multiplication, so the complexity should be $O(n^3)$. A more complex technique used in CSRMatrix time CSRMatrix and I'll explain it there instead.

```
//Example:
myMatrix.Multiply(aNormalMatrix);
//will multiply myMatrix with a normal matrix
```

- *CSRMatrix Multiply(CSRMatrix parm):* multiply 2 CSRMatrices and return the result.

In this method, I'll once again use the lookup array to optimize performance and utilize the characteristics of array of pointers to optimize space complexity.
Instead of first counting the nnz in the result matrix, I'll compute each entry in the result matrix directly and store them in a *linear array* of LinkedList of *Data*. Each entry in the array represents a row in the result matrix.

After the calculation is completed, I'll then dequeue all the entry stored in the temporary array one by one, row by row and it in the result's data array in order. This will ensure that the entries in the result array are sorted by row and by column exactly. Notice that because both the temporary array and the result array are actually an array of pointers, I don't allocate/deallocate any memory at all; instead, what I did was simply move one pointer to another.

The benefits of this are that I don't need to do the calculation twice (one to predict the nnz and one to put it in the result array) which can cost a lot for BigDecimal data type. Even though there are additional memory allocated for pointers, but they are very insignificant comparing to the performance tradeoff and, even with a massive matrix, it won't cost anything much except for the tradeoff of data locality.
For the calculation of each entry, because traversing in a row-wise manner is much less complex than traversing in a column-wise manner in CSRMatrix, instead of multiplying rows with columns, I'll multiply columns with rows to restrict the number of column traversing.

Using the same technique that I used in calculating the transpose of the CSRMatrix, I'll traverse from the first column of the parameter matrix to the last, and, for each column, I'll calculate the inner products of

the column and each row, then append the result into the correct entry in the array of linkedlist mentioned above. The support of the lookup array will speed up the seek for the correct entry in each row of the parameter matrix. Combining with the idea of using only nonzero entry to multiply, I'll use only the entry of the original matrix where there is a nonzero corresponding entry of the parameter matrix.
The calculation complexity is $O(n \cdot nnz)$ which is much better than the naïve $O(n^3)$ algorithms if the CSRMatrix is sparse. The space complexity is $\Theta(numberOfRow\_OriginalMatrix)$ for the lookup array in addition with some insignificant pointer memory. I believe my method has much better improvement than the blind prediction on the aspect of space complexity.
The pseudocode for the algorithm can be:

```
//Input a CSR Matrix parm
//Output the product of the original CSR and the parameter CSR Matrix
int[] rowCurIndx;                          //The lookup array
LinkedList<Data>[] tempResult;            //We will temporarily store our result
here
CSRMatrix result;                          //This is our result CSRMatrix
int totalSize = 0;                        //To keep track of the total size for
later allocation
InitializeLookupArray(rowCurIndx);
//Traverse through all columns of parm matrix
for parmCol in 0 to parm.NumberOfCol do
     //Traverse through all rows of original matrix
     for curRow in 0 to numberOfRow do
          temp = 0;
          for k in row[curRow] to row[curRow + 1] - 1 do
               //Since each consecutive entry in CSR data array might not in
               // consecutive column, we need to check whether we are
               // reading the entries from both matrix with the same k
               //This is to ensure that we are using a_curRow k and b_k parmCol
               if parmCol == parm.data[rowCurIndx[data[k].col]].col then
                    temp = temp + data[k].data *
                           parm.data[rowCurIndx[data[k].col]].data;
               //Since we iterate through all data in the same parmCol and
               //all data in the same row of the original matrix, we won't
               //miss any combination
                      //Check whether we got a nonzero
                      if temp != 0 do
                           //Add at the end
                           tempResult[curRow].add(new Data(temp, parmCol);
                           totalSize++; //Increase the nnz
           //Update the rowCurIndx, we are moving on to the next col!
           for i in 0 to numberOfRow do
                if EndOfRow(rowCurIndx[curRow]) then
                     rowCurIndx[i] = -1;
                else
                     rowCurIndx[i]++;
//Then copy all the entries to the result matrix
result.data = new Data[totalSize];   //Allocate with the correct size
int curIndx = 0;
for curRow in 0 to numberOfRow do
     result.row[curRow] = curIndx;
     while (NotEmpty(tempResult[curRow]) do
          //This only remove and add the pointer not allocating
          result.data[curIndx++] = tempResult[curRow].removeFirst();
return result;
```

```
//Usage example:
myMatrix.Multiply(anotherCSR);
//will return the LU factorization of myMatrix
```

- *Vector IterationMethod(Vector rightSide)*: return the approximate vector solution of the equation CSRMatrix . x = rightSide. Since I used the exact algorithm mentioned on the note comes with the assignment instruction, I will not discuss much it.

In addition, I add more detail to store some stats and graph for illustration purposes. There will be 3 types of graph, one for residual length vs max number of iterations, one for running time vs max number of iterations and one for the error between the exact solution and the approximate solution and the max number of iterations. In order to calculate the error, I took the original matrix and multiplied it with the approximate solution; then I take the length of the difference between the product and the right-hand side vector. The numbers are then parsed into a json string, written into a file and graphed using NodeJS. The graph will be included in the later section.

```
//Example:
myMatrix.IterationMethod(Vector rightSide);
//will return the approximate solution of myMatrix . x = rightSide
```

- *Boolean Equal(Matrix parm)*: compare if two matrices are exactly the same. It will return true if and only if each entry of the parm are the same as the original matrix.

```
//Example:
myMatrix.Equal(aNormalMatrix);
//will compare myMatrix to aNormalMatrix
```

- *Boolean Equal(CSRMatrix parm):* compare if two matrices are exactly the same. It will return true if and only if each entry of the parm are the same as the original matrix. This method will compare both the *row* and *data* arrays.

```
//Example:
myMatrix.Equal(anotherCSR);
//will compare myMatrix to anotherCSR
```

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

In this method, I'll iterate through all entries in the *data* array and print them all out. For each consecutive entry that aren't in consecutive columns, I'll print out several 0s which equals to one less than the difference between the two columns. For empty rows (all 0s rows – row[curRow] = row[curRow + 1]), I'll print out *colSize* number of 0s.

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

- *void PrintData():* print all the data of the current matrix.

This method will be used only for the debugging process. It will print out nonzero data based on the *data* and *row* array. This will almost be identical to the printed data in the *.mtx* file from the Matrix Market.

```
//Example:
myMatrix.PrintData();
//will print myMatrix's data
```

- *static Boolean IsSymmetric(CSRMatrix parm):* check if a matrix is symmetric or not. This will only work for a square matrix.

Unlike the *Matrix*'s *IsSymmetric*() method, this method will check all entries in each row instead of just checking from the first column to column *curRow.* The reason is because the missing 0s in the *data* array can lead to inaccurate check if we only checking half of the array.

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

5. Pre-task

The result of pre-task will be written on file *./preTask.txt*.

The tasks will be tested against input coming from the *static String* I defined at the beginning of the *Main* class. Each task for *CSRMatrix* will also be performed for a *Matrix* and the test will be marked as PASSED if and only if both results are the same. That means the *Matrix* class will be used as the testing method for the *CSRMatrix* operations. I have tested against multiple inputs and outputs using the result I got from *Matrix* class and via online calculators, but the file will show only the test against one specific input just for the ease of reading.

6. Task One

The result of task 1 will be written on file *./taskOne.txt*.

The tasks will be tested against input coming from the *static String* I defined at the beginning of the *Main* class. Each task for *CSRMatrix* will also be performed for a *Matrix* and the test will be marked as PASSED if and only if both results are the same. That means the *Matrix* class will be used as the testing method for the *CSRMatrix* operations. I have tested against multiple inputs and outputs using the result I got from *Matrix* class and via online calculators, but the file will show only the test against one specific input just for the ease of reading.

7. Task Two

Task two will be tested against 3 different matrices and 3 different right-hand-side vectors coming from Matrix Market. The inputs are stored in *./bigMatrices* and the results will be stored in *./solutionOutput*. The results are written in json style.

The result will be then taken into NodeJS, which will first beautify the json file and graphed via Plotly's library for NodeJS. The graph will be included in the next section, but I'll also include some HTML links for interactive mode.

Side-note: The NodeJS code is in *./NodeJS/index.js*. The user can use Visual Studio to open the *SolutionGraph.sln* file which has already been set up and ready to use. However, in order to run and graph, user needs to at least register for a free account on plot.ly to generate a username and an API key. I'll not include my own username and API key in the *index.js* file just for security issue.

In order to test this part, during the process, I'll record the length of the residual vector. After multiple runs with increasing max iteration (STEP_LIMT as in my code), the mentioned length decreases on most of the iterations. Moreover, I also calculate the length of the difference between the product of the approximate solution and the matrix and the right-hand-side vector; and, just like the residual length, they converge to 0. Thus, that graphically proves that my implementation of the algorithm given by Professor Panayot Vassilevski is correct.

There is a small limitation in my code, if the number of decimals gets too high while the BigDecimal scale remains unchanged, the program will crash with a division-by-zero exception. In order to fix this, simply change the BIGDECIMAL_SCALE defined at the beginning of the *Main* class. However, the bigger the constant, the slower it will go through the Iteration Method.

Some input produces huge residual length and error spikes on certain STEP_LIMIT. However, this only happened with small BIGDECIMAL_SCALE. Based on my assumption, the reason is the inaccuracy in

calculation due to rounding and max decimal point at some point. However, the distance between the max error/residual length and 0 never exceed 15.

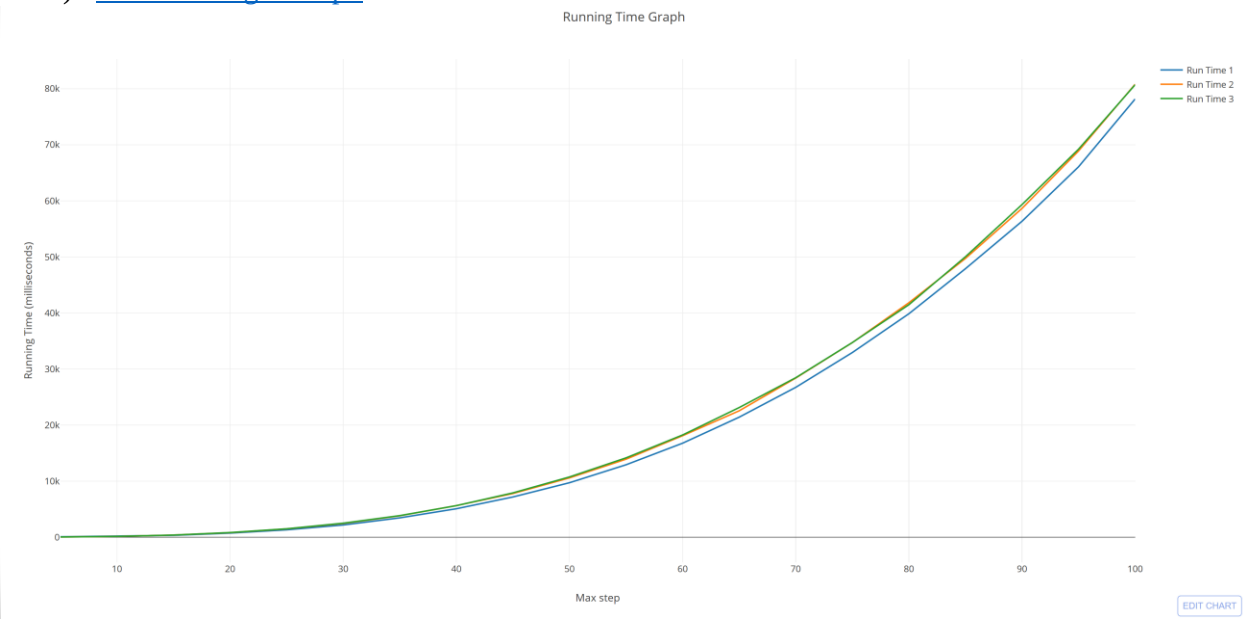
8. Benchmarking for Task Two

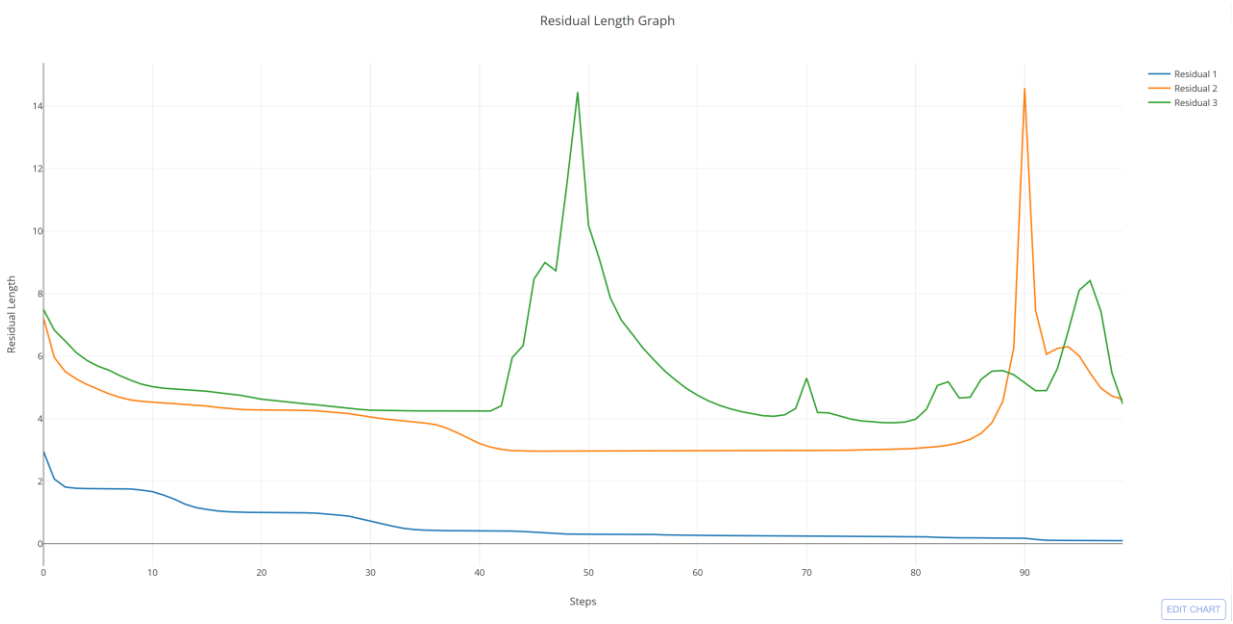
The input files are in *./bigMatrices*. I'll not include the data in this report since there are too much data.

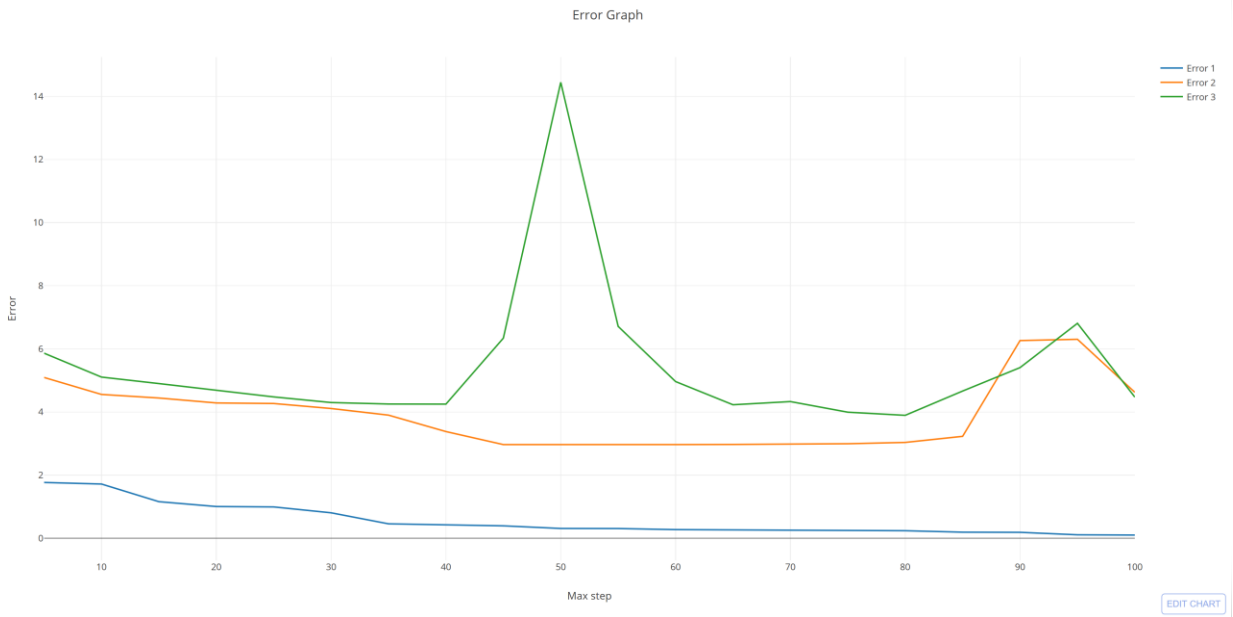
In this section, I'll run 3 different benchmarking with 3 different BIGDECIMAL_SCALE. The purpose of this section is to show how a higher BIGDECIMAL_SCALE can help us to represent and calculate on a more precise set of data but, at the same time, it will greatly affect the performance of the program in whole.

Benchmarking 1: Running with BIGDECIMAL_SCALE = 70 and STEP_LIMIT = 100. The 3 interactive graphs can be found at:

1) Error Graph
2) RunTime Graph
3) ResidualLength Graph







The benchmarking result is quite interesting.

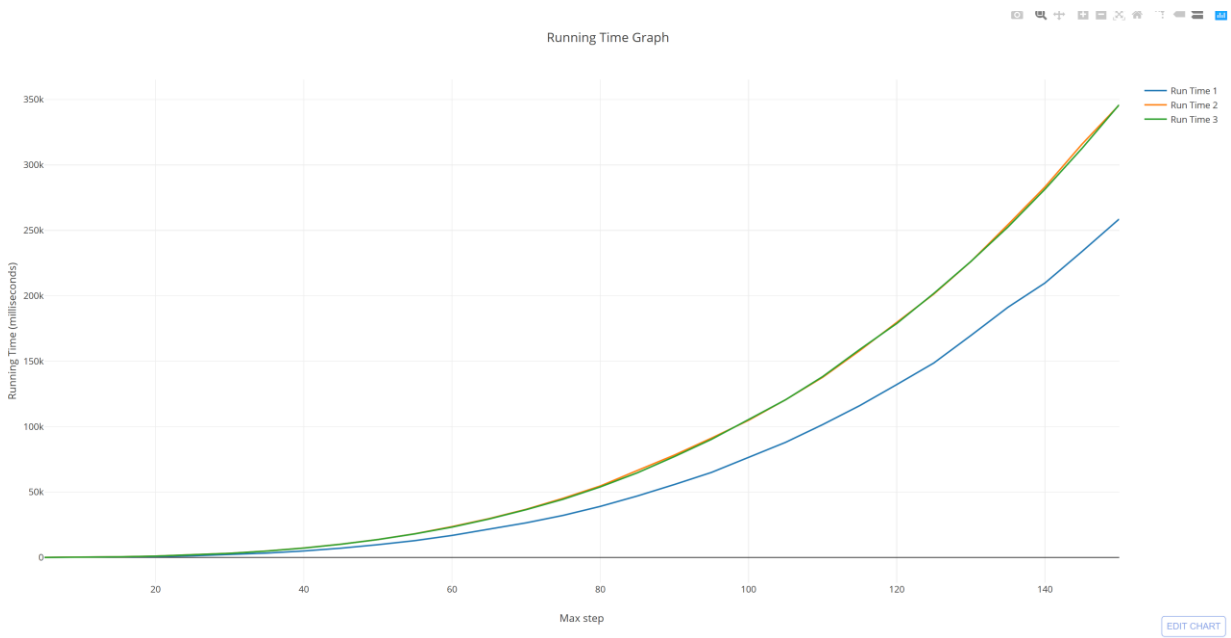
The graph of the running time is strictly increasing, but that's very predictable. However, I believe that if I chose to use and implement a more lightweight data structure to hold decimals, then the results can be improved significantly. Moreover, even though the scatter patterns in all three inputs are quite different, the running times are almost the same.

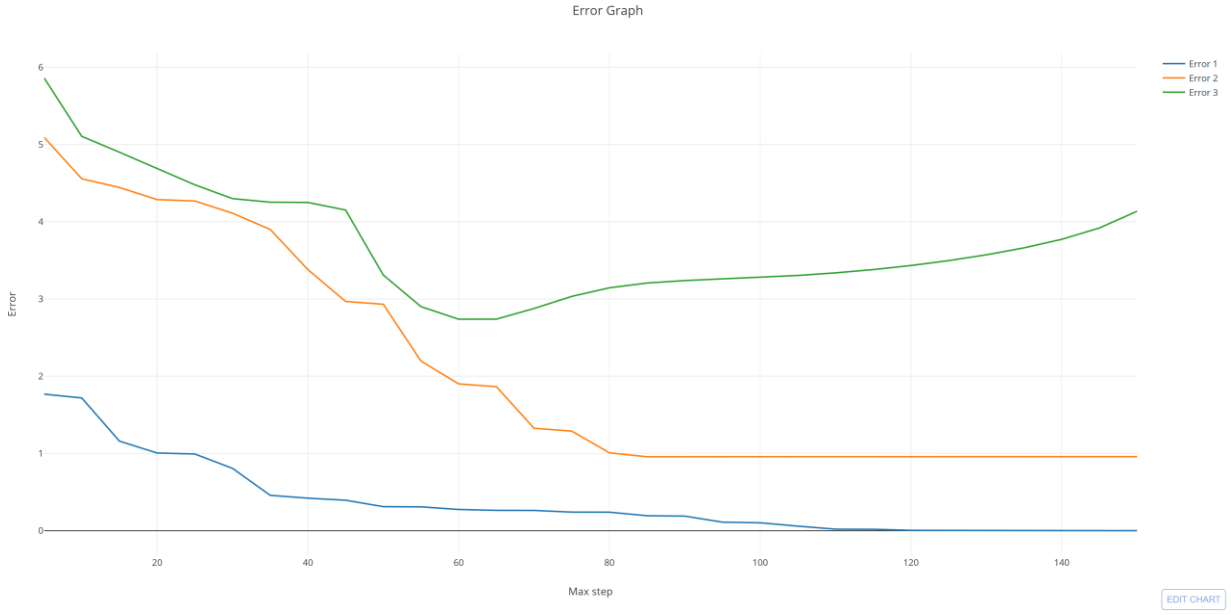
The fascinating thing is the huge spikes in error and residual length. However, since the length and error start to decrease again after some certain points, I think there is a high chance that this is a bug in Java *BigDecimal* library when encountering very small data (which are all over the inputs).

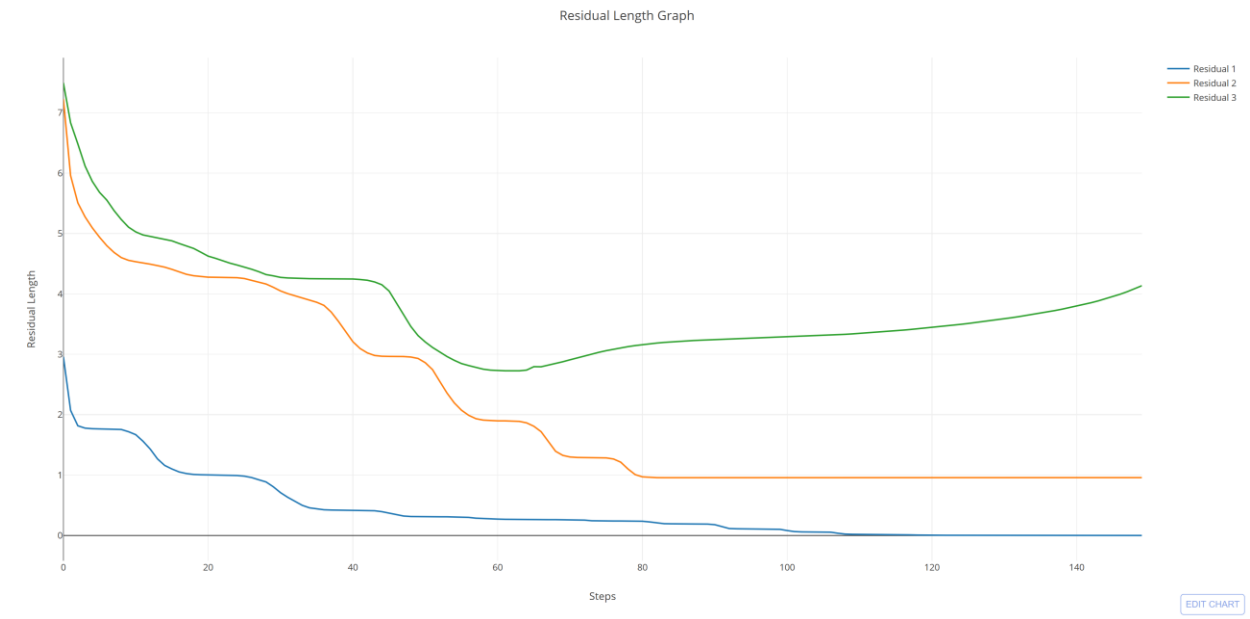
Overall, both graphs show that the error and residual length have the tendency to converge to 0.

Benchmarking 2: Running with BIGDECIMAL_SCALE = 100 and STEP_LIMIT = 150. The 3 interactive graphs can be found at:

1) Error Graph
2) RunTime Graph
3) ResidualLength Graph



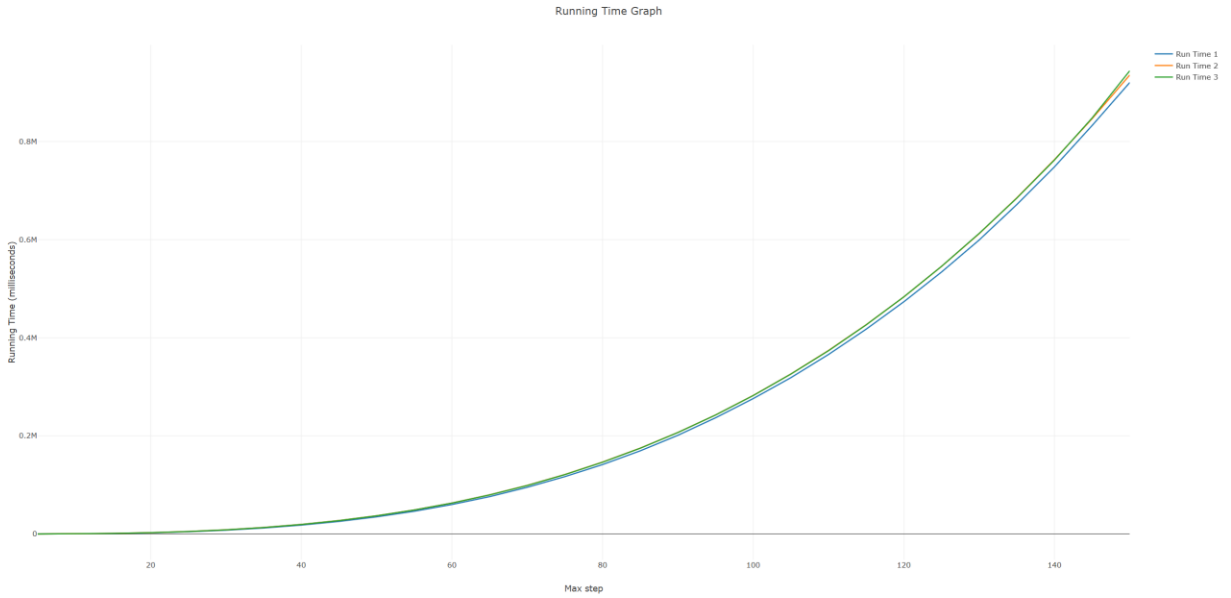


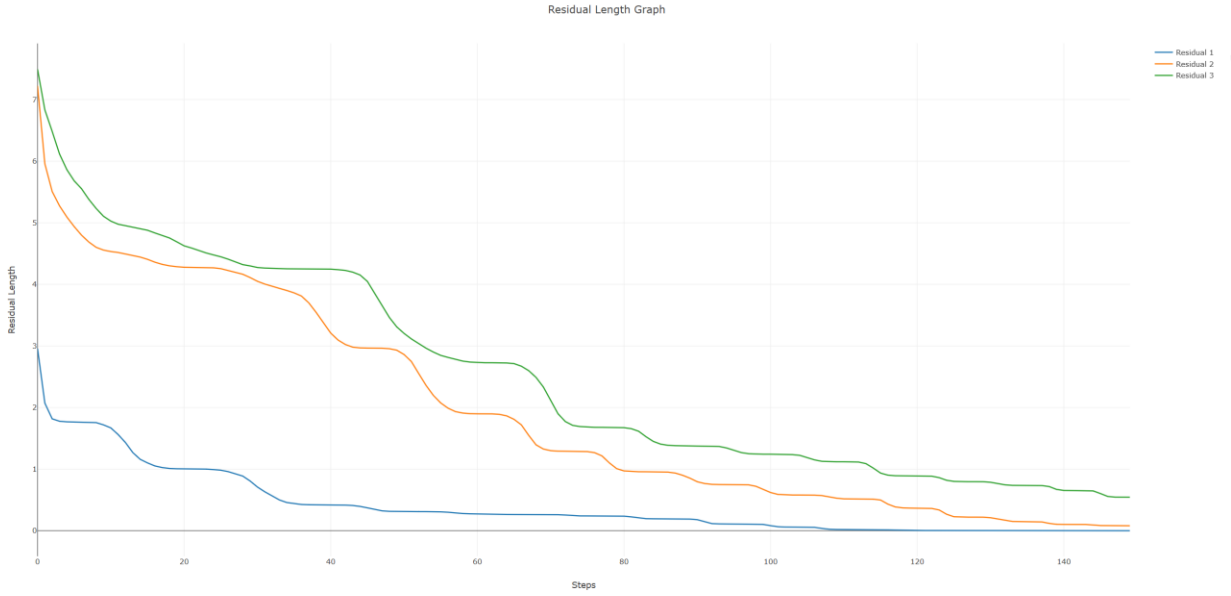


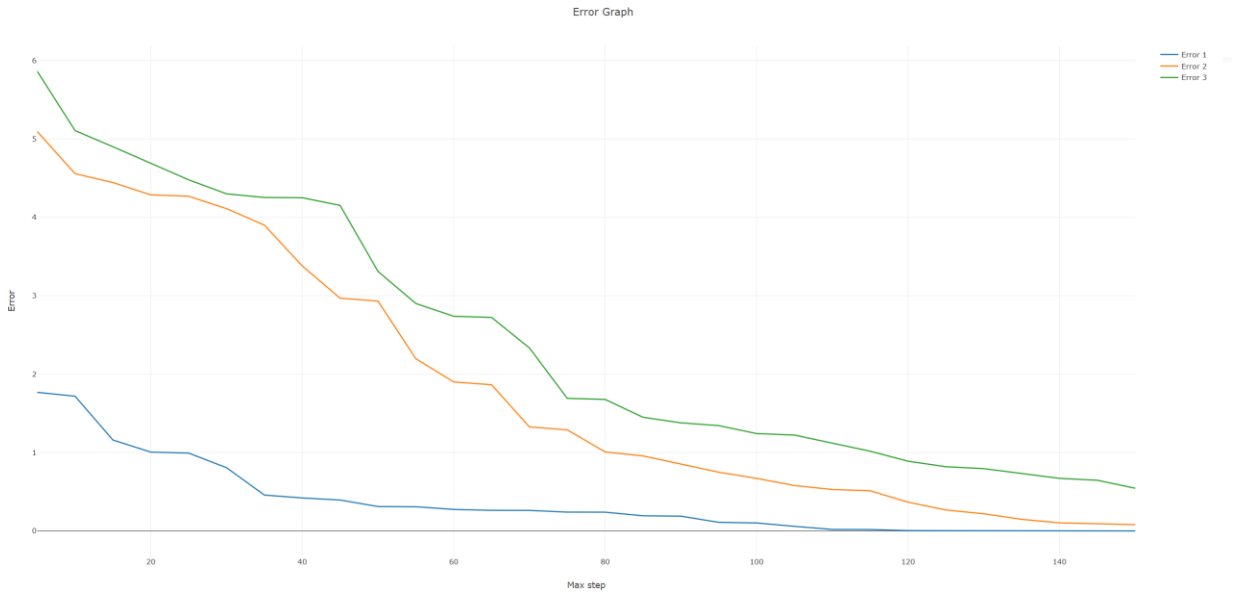
Surprisingly, as I increase the max decimal scale, the spikes in the value ease down and both graphs show a better convergence to 0. Thus, I believe this is a limitation in the BigDecimal library of Java. Moreover, since the limit of each graph is much closer than in the previous benchmarking, the increase in precision show a positive result.

Benchmarking 3: Running with BIGDECIMAL_SCALE = 200 and STEP_LIMIT = 150. The 3 interactive graphs can be found at:

1) Error Graph
2) RunTime Graph
3) ResidualLength Graph







In this last benchmarking, the graphs show some promising results. Both the error and the residual length are now strictly converging to 0. This graphically proves that my way of implementing the GMRES algorithm is correct and the higher the BIGDECIMAL_SCALE is, the better the result will be.

However, as an obvious tradeoff, the performance graph skyrocketed compared to the other two benchmarks. The run with the third input and 150 iterations took 15 minutes to complete!

9. Reference

1) Matrix multiplication calculator: https://matrix.reshish.com/multiplication.php
2) QR Factorization calculator: http://math.utoledo.edu/~codenth/Linear_Algebra/Calculators/QR_factorization.html
3) LU Factorization calculator: http://www.gregthatcher.com/Mathematics/LU_Factorization.aspx
4) System of linear equations solver: https://matrixcalc.org/en/slu.html
5) Plotly: https://plot.ly/#/
6) GMRES Note: Link
7) Project Instruction: Link