Assignment 4

Arithmetic Operations on Sparse Matrices

Pedram Pasandide

Due Date: 7 August 2024

1 Introduction

Arithmetic operations on sparse matrices are fundamental in computational mathematics, particularly in fields such as scientific computing and machine learning. Sparse matrices, characterized by a large proportion of zero elements, allow for efficient storage and computation by focusing only on the non-zero elements. Common arithmetic operations, including **addition**, **subtraction**, **multiplication**, and transpose, are optimized to take advantage of this sparsity, thereby reducing computational complexity and memory usage. Specialized algorithms and data structures, such as Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC), are employed to facilitate these operations, enabling the handling of large-scale problems that would be infeasible with dense matrix representations.

1.1 Matrix Market Format

If a matrix has many zeros, we want to skip saving zeros to lower the amount of memory taken. Read the file `MatrixMarket.pdf` to understand what a Matrix Market Format is. There are many ways to read Matrix Market format files and one of them is explained in Appendix - GSL Library. We can read a Matrix Market Format (files with `.mtx` extension) and save it into:

- Coordinate Storage (COO),
- Compressed Sparse Column (CSC), and
- Compressed Sparse Row (CSR) format.

The following matrix has a dimension of 7×7 (49 elements), with 15 non-zero values. Saving this matrix in double precision (64 bits for each double floating-point), takes 49×64 bits memory size. Saving this matrix in a conventional 2D array can waste the memory by saving all zero values.

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0.45 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0.45 \\ -0.03599942 & 0 & 0 & 0 & 1 & 0 & 0 \\ -0.0176371 & 0 & 0 & 0 & 0 & 1 & 0 \\ -0.007721779 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The file `b1_ss.mtx` has the same matrix in Matrix Market Format. Open `b1_ss.mtx` file and see the values. You should see:

```
%%MatrixMarket matrix coordinate real general
%-------------------------------------------------------------------------------
% UF Sparse Matrix Collection, Tim Davis
% http://www.cise.ufl.edu/research/sparse/matrices/Grund/b1_ss
% name: Grund/b1_ss
% [Unsymmetric Matrix b1_ss, F. Grund, Dec 1994.]
% id: 449
% date: 1997
% author: F. Grund
% ed: F. Grund
% fields: title A b name id date author ed kind
% kind: chemical process simulation problem
%-------------------------------------------------------------------------------
7 7 15
5 1 -.03599942
6 1 -.0176371
7 1 -.007721779
1 2 1
2 2 -1
1 3 1
```

```
3 3 -1
1 4 1
4 4 -1
2 5 .45
5 5 1
3 6 .1
6 6 1
4 7 .45
7 7 1
```

Any lines starts with `%` is a comment. The first row, `7 7 15`, always shows the number of rows, columns and none-zero values. A `.mtx` file has always three columns. The first column is the row indexes, and the second column is column indexes, and the third column are values corresponding to the rows and columns. In this case, we have 2 index columns with integer type (32 bits), and one column holding values with floating-point precision (lets say 64 bits double). The size it would take to save the same matrix with this format into the memory is $2 \times 15 \times 32 + 15 \times 64$ which is about 40% less memory compared with saving the matrix with all zeros in a 2D array.

1.2 Compressed Sparse Row (CSR)

The Compressed Sparse Row (CSR) format is a widely used data structure for representing sparse matrices efficiently. In CSR format, a matrix is stored using three one-dimensional arrays that compress the storage requirements by focusing only on the non-zero elements. These arrays are:

- Values Array (values): Contains all the non-zero elements of the matrix, stored row by row.
- Column Indices Array (col_indices): Stores the column indices corresponding to each non-zero element in the values array.
- Row Pointers Array (row_ptr): Holds the index in the values array where each row starts. The length of this array is one more than the number of rows in the matrix, with the last element pointing to the end of the values array.

You can find more numerical examples on the internet or you can also check here or/and here. CSR format is particularly beneficial for performing efficient arithmetic operations, matrix-vector multiplications, and saving memory space compared to dense matrix representations. Its structure allows quick access to rows and efficient iteration over non-zero elements, making it a popular choice in scientific computing and machine learning applications.

2 Implementation

2.1 Reading and saving `.mtx` into CSR format (5 points)

To conduct arithmetic operations on matrices, first we need to read `.mtx` and save them on memory in **CSR format**. You need to understand what CSR format is before programming this section. Files you have downloaded are:

- Files with `.mtx` extension downloaded from University of Florida Sparse Matrix Collection.
- `main.c` and `functions.h`.

Write your own code to read `.mtx` files and save them into **CSR** format. You **must NOT** use GSL library, and you have to drive the code from scratch. This is what you see in `functions.h`:

```c
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

////////////////////////////////////////////////////////////////////
// Do not change this part
typedef struct {
 double *csr_data;   // Array of non-zero values
 int *col_ind;       // Array of column indices
 int *row_ptr;       // Array of row pointers
 int num_non_zeros;  // Number of non-zero elements
 int num_rows;       // Number of rows in matrix
 int num_cols;       // Number of columns in matrix
} CSRMatrix;


void ReadMMtoCSR(const char *filename, CSRMatrix *matrix);

////////////////////////////////////////////////////////////////////

/* <Here you can add the declaration of functions you need.>
<The actual implementation must be in functions.c>
Here what "potentially" you need:
1. "addition" function receiving const CSRMatrix A, const CSRMatrix B, and computing C=A+B
2. "subtraction" function receiving const CSRMatrix A, const CSRMatrix B, and computing C=A-B
3. "multiplication" function receiving const CSRMatrix A, const CSRMatrix B, and computing C=A*B
4. "transpose" function receiving const CSRMatrix A, computing the transpose of A (C=A^T)

It is up to you how to save and return the product of each function, matrix C
*/


#endif
```

`CSRMatrix` is the name of structure holding the CSR format read from a `.mtx` file. That's why in the `main.c`, you need to declare the matrix `A` with the type of `CSRMatrix` structure:

```
// <Your CODE: include library(s)?

int main(int argc, char *argv[]) {

 // <<Your CODE: Handle the inputs here>
 const char *filename = argv[1];

 CSRMatrix A;
 ReadMMtoCSR(filename, &A);

 // <Your CODE: The rest of your code goes here>

}
```

The program must be able to get the `filename` from the command-line argument. For example, if the object file is named `main`, I can give the inputs to the program like `./main b1_ss.mtx`. Run the program with `jgl009.mtx`, `rel3.mtx.mtx`, `n3c4-b1.mtx.mtx` examples, and print out the row pointer, column index and values, and report them in `README.tex` file. For example, when I run my code with `./main b1_ss.mtx`, I have the following outputs:

```
Number of non-zeros:15
Row Pointer: 0 3 5 7 9 11 13 15
Column Index: 1 2 3 1 4 2 5 3 6 0 4 0 5 0 6
Values: 1.0000 1.0000 1.0000 -1.0000 0.4500 -1.0000 0.1000 -1.0000 0.4500 -0.0360
     1.0000 -0.0176 1.0000 -0.0077 1.0000
```

These values can be used later by accessing the members of structure `CSRMatrix`. This part is really important and you have to make sure you can read the file correctly. Otherwise, for the rest of assignment you will get wrong results.

2.2 Arithmetic Operations on Sparse Matrices (12 points)

Complete the code by writing a function for:

1. Sparse Matrix `addition`
2. Sparse Matrix `subtraction`
3. Sparse Matrix `multiplication`
4. Sparse Matrix `transpose`

I have to be able to run your program using the following format:

```
./main <file1.mtx> <file2.mtx> <operation> <print>
```

`<print>` is an option to print the matrices and it can be either `0` or `1`. If it is `1`, then all the matrices must be printed out in the terminal. `<operation>` is the type of operation that needs to be done. Let's say if we want to compute C = A+B, then I can run my code using:

```
./main <file1.mtx> <file2.mtx> addition 1
```

In this case, `file1.mtx` and `file2.mtx` will be saved as matrix A and B. Since the print option is `1`, all three matrices A, B, and C must be printed out in the terminal. Mind you that when it comes to operation `transpose`, the number of inputs are less than the others. For example, I should be able to get the transpose of `file1.mtx` using:

```
./main <file1.mtx> transpose 0
```

2.3 Report and `Makefile` (3 points)

Provide a `Makefile` with a detailed explanation that how your `Makefile` works. Your report **must** be in `README.tex` file. Other formats will not be accepted. Create the `README.pdf` file from `README.tex` and submit both formats. Create a section called "Appendix" in your report and include all your codes in the appendix. Fill out the following table for matrix multiplication and compare the CPU time with `.dot()` function in the given `CSR_test.ipynb` Python code. **Which one is faster? Your implementation in C or the Python function?**

Table 1: Matrix multiplication ($A \times B$) with CSR format

$A \times B$	A		B		CPU time (Sec)
	dim	nnz	dim	nnz	
n3c4-b1× Trec6	15×6	30	6×15	40	0.0000
2cubes_sphere× rm_101492rows					
tmt_sym× rm_726713rows					
StocF-1465× rm_1465137rows					

3 Submission on Avenue2L

Make sure you implement what ever we did in lectures, like profiling, documentation, and memory allocation. Submit on Avenue:

- `main.c`, `functions.c`, and `functions.h`.
- `Makefile`.

- `README.tex` and `README.pdf`.

4 Appendix

4.1 GSL Library

One of the ways to read matrix market format is using GSL library. You can set up GSL to validate if your code to read the matrix is correct. However, in this assignment **you are NOT allowed to use GSL functions and you must write the code from scratch to read a `.mtx` file**. To use GSL library, first you have to install it. Check if you have GSL installed using `gsl-config --version`. If not follow the steps:

1. Installation:
   - on Linux:
     `sudo apt-get update`
     `sudo apt-get install libgsl-dev`
   - on MacOS:
     `brew install gsl`

Tips! If GSL is installed in a non-standard location, you might need to set the `LD_LIBRARY_PATH` environment variable (on Linux) or `DYLD_LIBRARY_PATH` (on macOS). If you installed GSL in a custom location, add the following lines to `nano ~/.bashrc`:

- Linux: `export LD_LIBRARY_PATH=/path/to/gsl/lib:$LD_LIBRARY_PATH`
- MacOS: `export DYLD_LIBRARY_PATH=/path/to/gsl/lib:$DYLD_LIBRARY_PATH`

Replace `/path/to/gsl/lib` with the actual path to the GSL libraries. If the gsl-config tool is not in a directory that's already in your PATH, you might need to add that directory to your PATH. If the gsl-config tool is in a non-standard location, add its directory to your PATH:
`export PATH=/path/to/gsl/bin:$PATH`
After making the changes Save and Exit `bashrc`, then reload using `source ~/.bashrc`.

2. Compile and Run Code with GSL: Assuming you have a C file named `example.c` that includes `<gsl/gsl_spmatrix.h>`, you can compile and run your code as follows:

   `gcc main.c -o main -lgsl -lgslcblas`,

   where:

   - `-lgsl`: Links the GSL library, and
   - `-lgslcblas`: Links the CBLAS library, which GSL depends on.

Let's say we want to read the matrix `.mtx` format and keep it on the memory with CSR format. The following function receives the name of the file `filename` as a pointer, and reads the data in COO format and converts it into CSR format before getting rid of COO.

```c
#include <gsl/gsl_spmatrix.h>

gsl_spmatrix *ReadMMtoCSR(const char *filename)
{
  FILE *file = fopen(filename, "r");
  if (!file)
  {
    fprintf(stderr, "Failed to open file %s\n", filename);
    return NULL;
  }

  gsl_spmatrix *coo = gsl_spmatrix_fscanf(file);
  if (!coo)
  {
    fprintf(stderr, "Failed to read matrix from file %s\n", filename);
    fclose(file);
    return NULL;
  }

  // converting from COO format to CSR
  gsl_spmatrix *csr = gsl_spmatrix_crs(coo);

  gsl_spmatrix_free(coo);
  fclose(file);
  return csr;
}
```

To call `ReadMMtoCSR` function, you need to add the following code in your `main.c`:

```c
const char *filename = "b1_ss.mtx";
gsl_spmatrix *A = ReadMMtoCSR(filename);
```

- The pointer to rows can be called by `A->p[i]`, where `i` is the counter starting from zero to the maximum number of rows in matrix A (in `b1_ss.mtx` case it is seven).
- The column indexes can be accessed by `ja[i]=A->i[i]`, `i` from zero to number of non-zeros (in `b1_ss.mtx` case it is 7)
- The none zeros values, the third column in the file, can be accessed by `value[i]=A->data[i]`, `i` from zero to number of non-zeros (in `b1_ss.mtx` case it is 15)

If I print out these values for `b1_ss.mtx` based on CSR format, I get:

```
Number of non-zeros:15
Row Pointer: 0 3 5 7 9 11 13 15
Column Index: 1 2 3 1 4 2 5 3 6 0 4 0 5 0 6
Values: 1.0000 1.0000 1.0000 -1.0000 0.4500 -1.0000 0.1000 -1.0000 0.4500 -0.0360
    1.0000 -0.0176 1.0000 -0.0077 1.0000
```

Technically, in a part of this assignment, you will write a code from scratch to read the `.mtx` files and save them in CSR format doing exactly what GSL does.