Assignment 3

Solving System of Linear Equations

Pedram Pasandide

Due Date: 30 November 2023

1 Introduction

A system of linear equations represents a collection of multiple equations involving multiple variables, where each equation is linear (first degree only) and can be expressed in the form:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m \end{aligned}$$

Here:

- x_1,x_2,...,x_n are the variables.
- a_{ij} are the coefficients of the variables.
- b_i are the constant terms.

The system is typically represented in matrix form as $Ax = b$, where:

- A is the matrix of coefficients.
- x is the column vector of variables.
- b is the column vector of constants.

Solving a system of linear equations involves finding values for the variables (x_1,x_2,...,x_n) that satisfy all the equations simultaneously. This could result in three scenarios:

Unique Solution: There is one set of values for the variables that satisfy all equations.

No Solution: The system has no set of values that satisfy all equations simultaneously (inconsistent system).

Infinite Solutions: The system has multiple sets of values that satisfy all equations (dependent system).

Take a look at the Appendix to see where we use a system of linear equations.

2 Programming the Solver

Take a look at the Appendix to see what methods we use to solve a system of linear equations. Choosing the method is up to you!

2.1 Implementation

Read the file `MatrixMarket.pdf` and then the section Appendix to understand Matrix Market Format. To solve a system of linear equations ($Ax = b$), we need to have the coefficient matrix (A) and the vector (b) to solve the system for the vector x. You will obtain the matrix A by reading the files with `.mtx` extension.

Files you have downloaded are:

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

2.1.1 Reading matrix A from a `.mtx` file (5 points)

Drive your own code to read the file `LFAT5.mtx` and save the matrix in **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`:

```
#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);
void spmv_csr(const CSRMatrix *A, const double *x, double *y);

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

/* <Here you can add the declaration of functions you need.>
<The actual implementation must be in functions.c>
Here what "potentially" you need:
1. A function called "solver" receiving const CSRMatrix A, double *b, double *x
and solving the linear system
2. A function called "compute_residual" to compute the residual like r=Ax-b.
This shows how much x values you found are accurate, but
printing the whole vector r might not be a good idea. So
3. A function called compute_norm to compute the norm of vector residual
*/


#endif
```

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

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "functions.h"

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

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

    CSRMatrix A;
    ReadMMtoCSR(filename, &A);

    // Initializing all the vector b (in Ax=b)
    double *b = (double *)malloc(A.num_rows * sizeof(double));
    // Set all elements of b to 1
    for (int i = 0; i < A.num_rows; ++i) {
     b[i] = 1.0;
    }

    // <The rest of your code goes here>

  }
```

I should 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 like:

`./main LFAT5.mtx`

Print out the values of `LFAT5.mtx` example, including row pointer, column index and values, and report them in `README.tex` file. I should be able to see something like what we had for `b1_ss.mtx` example in the Appendix.

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.

2.1.2 Sparse Matrix Multiplication (2 points)

To just get your hands on CSR format and how to use it, start with developing the function:

`spmv_csr(const CSRMatrix *A, const double *x, double *y)`

This computes the $y = A \times x$. In this function the type of input argument `A` is `CSRMatrix`. Also x and y are vectors. In section Checking the Result, we use this function to find the accuracy of solver you have developed. To make sure this function works, use small examples like `b1_ss.mtx` and `LFAT5.mtx`.

We know now why CSR format takes less memory size. CSR is also faster to be used compared with using the full matrix format. **Why?** This is something we talked about during our lectures and the computer architecture.

2.1.3 Solving $Ax = b$ (10 points)

Drive a code from **scratch** solving the a linear system $A \times x = b$ for vector x. Here, we assume that the elements of vector b is all equal to 1:

```
  // Initializing all the vector b (in Ax=b)
```

```
double *b = (double *)malloc(A.num_rows * sizeof(double));
// Set all elements of b to 1
for (int i = 0; i < A.num_rows; ++i) {
 b[i] = 1.0;
}
```

You can use any solver that you think it would work, but you have to write it's code from scratch. One of the methods is Cholesky decomposition, but it doesn't always give you the answer. Especially when the matrix A is large. Choosing the method is up to you.

Questions you have answer:

1. Is your solver able to get the results for `b1_ss.mtx`? Yes or no mention **how and why?!**
2. Is your solver able to get the results for

- `2cubes_sphere.mtx`,
- `ACTIVSg70K.mtx`,
- `tmt_sym.mtx`, and
- `StocF-1465.mtx`?

If not not **why?**

2.1.4 Checking the Result by Computing the norm or Residuals (3 points)

To measure the error or residual in finding x (where $Ax = b$), you can compute the difference between $A \times x$ and b. This difference represents the residual, which ideally should be close to zero when x is the correct solution.

The residual $r = A \times x - b$ is a way to gauge the accuracy of the solution x. If x is indeed a good solution, the residual should be close to zero. The computation would involve subtracting the result of the sparse matrix-vector multiplication ($A * x$ in section Sparse Matrix Multiplication) from the original vector b.

Printing the entire residual vector might indeed be cumbersome. To demonstrate the efficiency or magnitude of the residual, you can summarize it by computing some metrics, such as the norm or magnitude of the residual vector.

So far, every time I compile and run the code (let's say `./main LFAT5.mtx`), I must be able to see something like this:

```
The matrix name: LFAT5.mtx
The dimension of the matrix:14 by 14
Number of non-zeros:30
CPU time taken solve Ax=b: 0.000009
Residual Norm: 0.000000
```

The time printed out here is only the time taken to solve $Ax = b$.

2.2 Report and `Makefile`

Provide a `Makefile` to compile your code, and in the report with details tells us 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. Prove a table like for different problems:

Table 1: Results

Problem	size		non-zeros	CPU time (Sec)	Residual
	row	*column*			
LFAT5.mtx	14	14	30	0.000009	0
LF10.mtx					
ex3.mtx					
jnlbrng1.mtx					
ACTIVSg70K.mtx					
2cubes_sphere.mtx					
tmt_sym.mtx					
StocF-1465.mtx					

You can use LaTex file format that I have sent you. In the report you must include the mathematical algorithm showing how the linear solve works. For example if you are using Cholesky decomposition, mention only the algorithm not the numerical example. All the analysis with VTune, gcov, and figures plotting the matrix A, must be included in your report file. Plotting the matrix A for all problems like:

Visualizing the sparsity pattern of a matrix refers to displaying a graphical representation of the matrix where the positions of non-zero elements are highlighted, typically against a background of zero elements. This visualization aids in understanding the density and structure of non-zero values in a sparse matrix.

Here's a description and possible approaches you can take to visualize the sparsity pattern of a matrix:

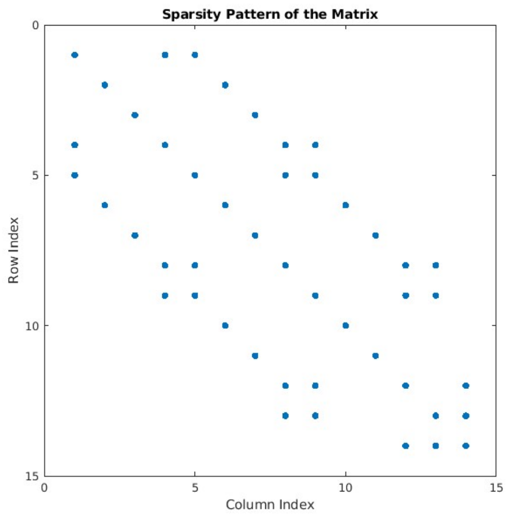

Figure 1: Visualize sparsity pattern of matrix LFAT5.mtx

Visualizing the sparsity pattern of a matrix, such as matrix A, involves creating a plot or image where:

- Each row represents a row in the matrix.
- Each column represents a column in the matrix.
- Non-zero elements are marked or highlighted, while zero elements are left blank.
- The position and density of non-zero elements are displayed, providing insights into the matrix structure.

Possible Ways to Visualize:

1. Using C with `libpng`: You can employ C programming with the `libpng` library to generate an image representing the sparsity pattern. The library you must include is `#include <png.h>` and compile the code with `-lpng` flag.
2. Linking to Python for Visualization: Alternatively, you can use Python, a language widely used for data visualization. Extract the sparsity pattern from the matrix data. Utilize Python libraries like matplotlib to create a plot. Plot the matrix, customizing markers or colors for non-zero elements. Save the plot as an image file.

Plot the sparsity pattern for all the problems mentioned in Table 1. To avoid losing any points, make sure you include your code and the results in the report.

2.3 Solving for Larger Systems- Bonus (+2 points)

Driving an algorithm to solve larger systems is more challenging. Solve the following systems:

- `2cubes_sphere.mtx`,
- `ACTIVSg70K.mtx`,
- `tmt_sym.mtx`, and
- `StocF-1465.mtx`.

Mention the algorithm you have used, and include your results in your report.

3 Submission

Make sure you implement what ever we did in lectures, like profiling, documentation, and memory allocation. **Submit On GitLab following files:**

- `main.c`, this is the main body of your code.
- `functions.c`, the actual implementation of the functions are here.
- `functions.h`, the declarations are only here.
- **Both** `README.tex` and `README.pdf`

Your directory on GitLab must be like `MT2MP3>A3>all_the_files_here`.

Make sure to add, commit and push your files before the due date. You can verify your submission via web browser by inspecting the contents of your submission repo. If you have any question about the procedure of submitting in this way please feel free to email Alireza Daeijavad (daeijava@mcmaster.ca) with the subject "2MP3 Assignment 3".

4 Appendix

4.1 Where we have System of Linear Equations?

Systems of linear equations arise in various fields, and here are some examples:

- Optimization Problems:

  Linear Programming: In operations research and economics, linear programming involves optimizing a linear objective function subject to linear equality and inequality constraints.

  Portfolio Optimization: In finance, solving for optimal investment strategies involves formulating systems of equations based on risk-return trade-offs, considering constraints on assets.

- Engineering Applications:

  Circuit Analysis: Electrical engineers solve systems of equations to analyze and design electrical circuits, determining voltage and current distributions.

  Structural Analysis: Civil and mechanical engineers use linear equations to model stresses and strains in structures, helping optimize designs.

- Differential Equations:

  Control Systems: Engineers use differential equations to model and control systems like robotics or automatic control systems.

  Physics and Chemistry: Modeling physical or chemical phenomena, like heat diffusion or reaction rates, often involves systems of differential equations.

- Machine Learning and Data Science:

  Regression Analysis: Solving for coefficients in regression models involves solving systems of equations, where relationships between variables are linear.

  Solving Linear Systems in Data: Matrix operations underlie various algorithms, including solving linear systems in data processing or image processing tasks.

- Economics and Social Sciences:

  Input-Output Models: Economic models use linear equations to represent economic interactions between sectors.

  Social Network Analysis: Linear equations are used in modeling relationships, influence, or information flow in social networks.

- Cryptography:

  Cryptanalysis: Solving systems of linear equations is fundamental in some methods of breaking cryptographic algorithms.

These examples showcase the broad applicability of systems of linear equations in diverse fields, demonstrating their importance in modeling, analysis, and optimization across various domains.

4.2 What Solvers We Have?

There are various solvers available to solve linear systems of equations, each with its own advantages, disadvantages, and suitability based on the characteristics of the system. Here's a brief overview of some common solvers:

- Direct Methods:

  Gaussian Elimination: This is a basic method to solve linear systems by transforming the augmented matrix to reduced row-echelon form.

  LU Decomposition: It factors the matrix into the product of a lower triangular matrix (L) and an upper triangular matrix (U), enabling efficient solving of systems of equations.

- Iterative Methods:

  Jacobi Method: It's an iterative method that solves for each variable by approximating using previous values.

  Gauss-Seidel Method: An improvement over Jacobi, it uses updated values as soon as they are available.

  Successive Over-Relaxation (SOR): Builds on Gauss-Seidel by using a relaxation factor to speed up convergence.

  Conjugate Gradient Method: Efficient for symmetric positive-definite matrices, it minimizes the error over the space of A.

  GMRES (Generalized Minimal Residual Method): Iterative method suitable for non-symmetric matrices, minimizing the residual.

- Sparse Solvers:

  Sparse LU Factorization: Specialized LU decomposition for sparse matrices, exploiting their structure.

  Cholesky Factorization: Efficient for symmetric positive-definite sparse matrices.

  Sparse Iterative Solvers: Methods like Conjugate Gradient, GMRES, or BiCGSTAB, specifically designed for large sparse matrices.

- Matrix Factorization and Decomposition:

  QR Decomposition: Decomposes a matrix into an orthogonal matrix and an upper triangular matrix.

  SVD (Singular Value Decomposition): Decomposes a matrix into singular vectors and singular values, useful for least squares and data compression.

The choice of solver depends on various factors:

- Matrix Properties: Symmetric, positive-definite, sparse, etc.
- Accuracy Required: Some methods might converge faster but provide less accurate results.
- Memory and Computational Efficiency: Some solvers might be more memory-efficient or suitable for parallelization.
- Special Structures: Exploiting specific matrix structures (e.g., sparse matrices) with specialized solvers can be more efficient.

For real-world applications, considering the characteristics of the problem (matrix size, sparsity, symmetry, etc.) and the trade-offs between accuracy, computational complexity, and memory usage is crucial in selecting the most appropriate solver.

4.3 Matrix Market Format

First, read the file `MatrixMarket.pdf` to understand what a Matrix Market Format is. In general, to save a matrix which has many zeros and to skip saving zeros to lower the amount of memory taken for saving the matrix. This can be really important if we are dealing with Sparse Matrix. There are many ways to read and save these matrices like using GSL. Using this library, we can read a Matrix Market Format with `.mtx` extension in

- Coordinate Storage (COO)
- Compressed Sparse Column (CSC)
- Compressed Sparse Row (CSR)

Check out a small example of a `.mtx` named `b1_ss.mtx`. The following matrix is a 7×7 size matrix (49 elements), with only 15 non-zero values. Saving this matrix in double precision (64bits double in my machine) on memory, takes 49×64bit memory size. In this case the would waste of memory for saving all zero value.

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

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

```
%%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 actual 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 is values corresponding to its 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 to save the same matrix with this format into the memory is $2\times15\times32+15\times64$ which is about 40% less memory compared with saving the full matrix format. This value can be significantly lower if the matrix is really large and sparse.

In the future (in this assignment you are NOT allowed to use GSL), if you want to use GSL library to read `.mtx` first you have to install it. Let's say we want to read the matrix `.mtx` format and keep it on memory with CSR format. If you have GSL installed on you OS, you should be able to compile and run your code by including `#include <gsl/gsl_spmatrix.h>` library at the top of your `main.c`. The following function receives the name of the file `filename` as a pointer, and reads the data in COO format and saves it into CSR format.

```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`:

```
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)
- And 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)

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 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.