# 2PM3 Assignment 3

## Solving a System of Linear Equations in C

Pranav Chandrakumar

# 1 Introduction

In this document, I have outlined the results of my linear system solver program. Under Section 2.1, I have a sample of the correct conversion of a Matrix Market (MM) file to a Compressed Sparse Row (CSR) data type. I have also provided the results of a sample CSR matrix-vector multiplication under Section 2.2. In Section 2.3, I have included the results of the few systems my program was able to solve, and provided detailed reasoning as to why I was unable to find solutions for other systems. I have also included my functions which calculate the residual and norm and gave a brief description of how they were implemented in my solver in that Section. In Section 2.4 I have provided details as to how my `Makefile` works, as well visualizations of the CSR matricies from Section 2.3.

# 2 Results

## 2.1 Reading MM files into CSR format

My `ReadMMtoCSR` function is the function which enables me to read the Matrix Market data stored in the `.mtx` file and output the correct number of Non-Zero Entries, as well as the correct Row Pointers and Column Indexes. This function is critical for later function processes, as without correctly reading MM data into a CSR format, solving the system correctly will not be possible. Here is a sample output after reading `LFAT5.mtx`:

```
Number of non-zero entries: 30
Row Pointers: 0 1 2 3 5 7 9 11 14 17 19 21 24 27 30
Column Indexes: 0 1 2 0 3 0 4 1 5 2 6 3 4 7 3 4 8 5 9 6 10 7 8 11 7
    8 12 11 12 13
Values: 1.5709 12566400.0000 0.6088 -94.2528 15080.4480 0.7854
   3.1418 -6283200.0000 12566400.0000 -0.3044 0.6088 -7540.2240
   -94.2528 15080.4480 94.2528 0.7854 3.1418 -6283200.0000
   12566400.0000 -0.3044 0.6088 -7540.2240 -94.2528 15080.4480
   94.2528 0.7854 3.1418 94.2528 0.7854 1.5709
```

## 2.2 CSR Matrix Multiplication

Using my function `spmv_csr` I can multiply a matrix **A** by a column vector **x**, and can write the result into a column vector **y**. This function was not implemented in my system solver, as the matrix multiplication function I used for that accepted different argument. As the `spmv_csr` function was a function inherently supplied by the requirements of this assignments, I could not alter the number/type of arguments supplied to it, and thereby opted to use a different function for matrix multipication in my solver. This output is the output for multiplying the `b1_ss.mtx` CSR matrix by a column vector of **x** with all entries equal to 1.0.

```
Matrix Name: b1_ss.mtx
Number of non-zero entries: 15
The dimensions of the matrix: 7 by 7
Not all diagonal entries are non-zero. Jacobi approach will fail.
Result:
3.000000 -0.550000 -0.900000 -0.550000 0.964001 0.982363 0.992278
Program runtime: 0.0000000000000000
```

## 2.3 Results From Solved Systems

Table 1: Results with max number of Jacobi cycles = 10000 and Norm difference less than 1e-7

| Problem | Size | | Non-zero | CPU time (Sec) | Norm-Residual |
|---|---|---|---|---|---|
| | *Rows* | *Columns* | | | |
| LFAT5.mtx | 14 | 14 | 30 | 0.000000 | 0.000000 |
| LF10.mtx | 18 | 18 | 50 | 0.000000 | 0.037859 |
| ex3.mtx | 1821 | 1821 | 27253 | 1.633839 | -nan |
| jnlbrng1.mtx | 40000 | 40000 | 119600 | 1.652283 | 0.000000 |
| ACTIVSg70K.mtx | 69999 | 69999 | 154313 | 18.406089 | -nan |
| 2cubes_sphere.mtx | 101492 | 101492 | 874378 | 68.509645 | -nan |

Unfortunately, my solver proved to be unable to solve matrices which were substantially large. This is primarily due to the inefficiency in my `ReadMMtoCSR` function. Although it needs to be called only once, the manner in which I read data from an `.mtx` file proves to be highly inefficient. Furthermore, the `MatrixMultiply` function contributes a substantial amount to the program's run-time, as it needs to be called for every iteration of the solution method. Using `gprof`, below is the report for running the `jnlbrng1.mtx` system:

```
%    cumulative    self               self     total
time    seconds    seconds     calls    s/call    s/call   name
82.52       8.59      8.59         1      8.59      8.59   ReadMMtoCSR
12.58       9.90      1.31      1752      0.00      0.00
   MatrixMultiply
 2.40      10.15      0.25      1752      0.00      0.00   ComputeNorm
 2.40      10.40      0.25         1      0.25      1.82   Jacobi
 0.10      10.41      0.01         1      0.01      0.01   Symmetrify
 0.00      10.41      0.00         1      0.00      0.00   NumDiagonals
```

### 2.3.1 The Jacobi Method

The method I implemented for my solver in this program is the Jacobi method. This method is an iterative numerical method that converges to an answer by iterating off of previously generated results for the solution vector. The algorithm works off of the following process.

$$\begin{aligned} \mathbf{x_n}^0 &= \langle 0, 0, \ldots, 0_n \rangle \\ A \cdot \mathbf{x_n}^0 &= \mathbf{r_n}^0 \\ \mathbf{x_n}^1 &= \mathbf{b_n} - \mathbf{r_n}^0 \end{aligned}$$

Essentially, each element of the solution vector is solved for by the third line of the process above, and then those new elements comprise the next solution vector $\mathbf{x_n}$. This process repeats until a specified stop condition is reached, such as a condition on the difference in norms of sequential solution vectors, or the total number of iterations permitted by the algorithm. As can be seen in Table 1, I ran my program in a way such that the Jacobi method would stop if either 10000 iterations were reached, or the norm of $\mathbf{b_n} - \mathbf{r_n}$ is less than 1e-7. To calculate this difference in the norm, I implemented the function `ComputeNorm`, which was called the same number of times as `MatrixMultiply`, each time to check if the norm difference was sufficient to stop iterations. This approach also failed at solving few systems in Table 1. For systems which the result was `-nan`, the Jacobi method did not converge to a solution vector. This implies that these systems could not be solved by this approach (perhaps they aren't diagonally dominant), or there is an inherent flaw in my method implementation which prevents my algorithm from finding a solution for these systems.

Furthermore, I was unable to solve for `b1_ss.mtx` as it is not a matrix which has all non-zero entries in the diagonal. This is a limitation of the Jacobi method, as it only works for linear systems in which all the diagonal entries of the coefficient matrix are non-zeros, and are diagonally dominant.

## 2.4 Report and `Makefile`

Here is the `Makefile` for this assignment:

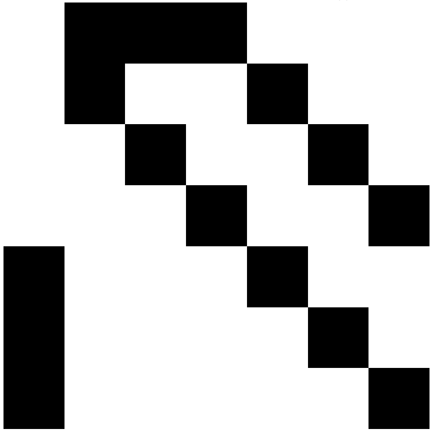
```
CC = gcc
CFLAGS = -Wall -Wextra -O3 #-O3 flag automatically slightly
   optimizes the code while compiling
LDFLAGS = -lm -lpng -pg#Need an extra set of flags for functions.c
   (instead of just including -lm in CFLAGS)
SOURCES = functions.h functions.c main.c
OBJECTS = $(SOURCES:.c =.o)
EXECUTABLE = SysOLE

all : $(EXECUTABLE)

$(EXECUTABLE):$(OBJECTS)
    $(CC) $(OBJECTS) -o $(EXECUTABLE) $(LDFLAGS)

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
clean :
    rm -f $(EXECUTABLE)
```

Standard `Makefile` format for this program. Run `make` in the terminal to create the executable file `SysOLE`. `-lm` flag is included due to usage of functions from the library `<math.h>` in the `functions.c` source file. `-O3` is included due to the inherent boost it provides to run-time efficiency. Source files are `functions.h`, and `functions.c`. `-pg` was included for `gprof` which was used in Section 2.3 to determine why my program was so inefficient at computing larger matrices. Running `make clean` in the terminal will remove only the executable files while retaining the source code.

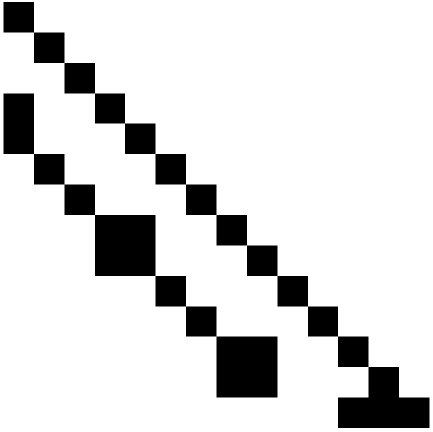
### 2.4.1 CSR Matrix Visualizations

Below are visualizations of the systems which were required to be solved in Section 2.3. Visualizations for `b1_ss.mtx`, `LFAT5.mtx`, `LF10.mtx`, and `ex3.mtx` were able to be generated. However, visualizations for `jnlbrng1.mtx`, `ACTIVISg70K.mtx`, `2cubes_sphere.mtx`, `tmt_sym.mtx`, `StocF-1465.mtx` were not generated. Visualizations for these four systems proved to be too computationally demanding, and the function used for visualization, `save_sparsity_pattern`, was unable to generate the required visualizations.

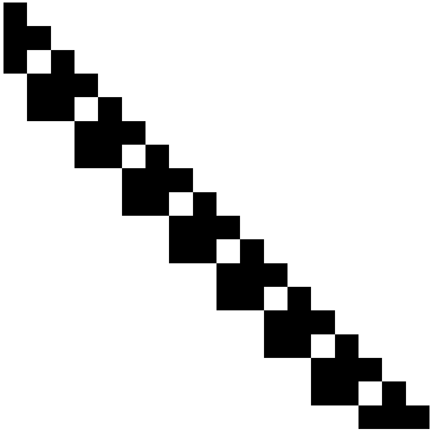
b1_ss.mtx. Rows and Columns = 7 x 7. #non-zero entries = 15



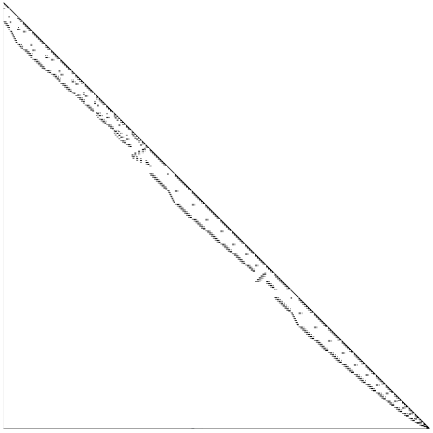
LFAT5.mtx. Rows and Columns = 14 x 14. #non-zero entries = 30



LF10.mtx. Rows and Columns = 18 x 18. #non-zero entries = 50



ex3.mtx. Rows and Columns = 1821 x 1821. #non-zero entries = 27253



### 2.4.2 Vtune

Below is a collection of the reports for running `b1_ss.mtx`, `LFAT5.mtx`, `LF10.mtx`, `ex3.mtx`, and `jnlberg.mtx` in order.

```
Top Hotspots
Function  Module  CPU Time  % of CPU Time(%)
--------  ------  --------  ----------------
Effective Physical Core Utilization: 4.0% (0.322 out of 8)
 | The metric value is low, which may signal a poor physical CPU
   cores
 | utilization caused by:
 |     - load imbalance
 |     - threading runtime overhead
 |     - contended synchronization
 |     - thread/process underutilization
 |     - incorrect affinity that utilizes logical cores instead of
   physical
 |       cores
 | Explore sub-metrics to estimate the efficiency of MPI and OpenMP
    parallelism
 | or run the Locks and Waits analysis to identify parallel
   bottlenecks for
 | other parallel runtimes.
 |
    Effective Logical Core Utilization: 2.2% (0.350 out of 16)
     | The metric value is low, which may signal a poor logical CPU
         cores
     | utilization. Consider improving physical core utilization as
         the first
     | step and then look at opportunities to utilize logical cores
        , which in
     | some cases can improve processor throughput and overall
        performance of
     | multi-threaded applications.
     |
Collection and Platform Info
    Application Command Line: ./SysOLE "b1_ss.mtx"
    Operating System: 5.15.133.1-microsoft-standard-WSL2 DISTRIB_ID
       =Ubuntu DISTRIB_RELEASE=22.04 DISTRIB_CODENAME=jammy
       DISTRIB_DESCRIPTION="Ubuntu 22.04.3 LTS"
    Computer Name: PC-PC
    Result Size: 3.6 MB
    Collection start time: 23:37:18 06/12/2023 UTC
    Collection stop time: 23:37:18 06/12/2023 UTC
    Collector Type: Driverless Perf per-process counting,User-mode
       sampling and tracing
    CPU
        Name: Intel(R) microarchitecture code named Tigerlake H
        Frequency: 2.304 GHz
        Logical CPU Count: 16
        Cache Allocation Technology
            Level 2 capability: not detected
            Level 3 capability: not detected
```

```
Top Hotspots
Function  Module  CPU Time  % of CPU Time(%)
--------  ------  --------  ----------------
Effective Physical Core Utilization: 8.0% (0.644 out of 8)
 | The metric value is low, which may signal a poor physical CPU
    cores
 | utilization caused by:
 |     - load imbalance
 |     - threading runtime overhead
 |     - contended synchronization
 |     - thread/process underutilization
 |     - incorrect affinity that utilizes logical cores instead of
    physical
 |       cores
 | Explore sub-metrics to estimate the efficiency of MPI and OpenMP
     parallelism
 | or run the Locks and Waits analysis to identify parallel
    bottlenecks for
 | other parallel runtimes.
 |
    Effective Logical Core Utilization: 4.1% (0.653 out of 16)
     | The metric value is low, which may signal a poor logical CPU
        cores
     | utilization. Consider improving physical core utilization as
        the first
     | step and then look at opportunities to utilize logical cores
       , which in
     | some cases can improve processor throughput and overall
        performance of
     | multi-threaded applications.
     |
Collection and Platform Info
    Application Command Line: ./SysOLE "LFAT5.mtx"
    Operating System: 5.15.133.1-microsoft-standard-WSL2 DISTRIB_ID
       =Ubuntu DISTRIB_RELEASE=22.04 DISTRIB_CODENAME=jammy
       DISTRIB_DESCRIPTION="Ubuntu 22.04.3 LTS"
    Computer Name: PC-PC
    Result Size: 3.6 MB
    Collection start time: 23:40:28 06/12/2023 UTC
    Collection stop time: 23:40:29 06/12/2023 UTC
    Collector Type: Driverless Perf per-process counting,User-mode
       sampling and tracing
    CPU
        Name: Intel(R) microarchitecture code named Tigerlake H
        Frequency: 2.304 GHz
        Logical CPU Count: 16
        Cache Allocation Technology
            Level 2 capability: not detected
            Level 3 capability: not detected
```

```
Top Hotspots
Function  Module  CPU Time  % of CPU Time(%)
--------  ------  --------  ----------------
Effective Physical Core Utilization: 8.6% (0.686 out of 8)
 | The metric value is low, which may signal a poor physical CPU
    cores
 | utilization caused by:
 |     - load imbalance
 |     - threading runtime overhead
 |     - contended synchronization
 |     - thread/process underutilization
 |     - incorrect affinity that utilizes logical cores instead of
    physical
 |       cores
 | Explore sub-metrics to estimate the efficiency of MPI and OpenMP
     parallelism
 | or run the Locks and Waits analysis to identify parallel
    bottlenecks for
 | other parallel runtimes.
 |
    Effective Logical Core Utilization: 4.1% (0.658 out of 16)
     | The metric value is low, which may signal a poor logical CPU
        cores
     | utilization. Consider improving physical core utilization as
        the first
     | step and then look at opportunities to utilize logical cores
       , which in
     | some cases can improve processor throughput and overall
        performance of
     | multi-threaded applications.
     |
Collection and Platform Info
    Application Command Line: ./SysOLE "LF10.mtx"
    Operating System: 5.15.133.1-microsoft-standard-WSL2 DISTRIB_ID
       =Ubuntu DISTRIB_RELEASE=22.04 DISTRIB_CODENAME=jammy
       DISTRIB_DESCRIPTION="Ubuntu 22.04.3 LTS"
    Computer Name: PC-PC
    Result Size: 3.6 MB
    Collection start time: 23:41:26 06/12/2023 UTC
    Collection stop time: 23:41:26 06/12/2023 UTC
    Collector Type: Driverless Perf per-process counting,User-mode sampling and tracing
    CPU
        Name: Intel(R) microarchitecture code named Tigerlake H
        Frequency: 2.304 GHz
        Logical CPU Count: 16
        Cache Allocation Technology
            Level 2 capability: not detected
            Level 3 capability: not detected
```

```
Top Hotspots
Function        Module     CPU Time  % of CPU Time(%)
--------------  ---------  --------  ----------------
MatrixMultiply  SysOLE       2.229s             86.7%
ReadMMtoCSR     SysOLE       0.140s              5.5%
ComputeNorm     SysOLE       0.100s              3.9%
Jacobi          SysOLE       0.090s              3.5%
__GI___qsort_r  libc.so.6    0.010s              0.4%
Effective Physical Core Utilization: 11.3% (0.907 out of 8)
 | The metric value is low, which may signal a poor physical CPU
    cores
 | utilization caused by:
 |     - load imbalance
 |     - threading runtime overhead
 |     - contended synchronization
 |     - thread/process underutilization
 |     - incorrect affinity that utilizes logical cores instead of
    physical
 |       cores
 | Explore sub-metrics to estimate the efficiency of MPI and OpenMP
     parallelism
 | or run the Locks and Waits analysis to identify parallel
    bottlenecks for
 | other parallel runtimes.
 |
    Effective Logical Core Utilization: 6.0% (0.952 out of 16)
     | The metric value is low, which may signal a poor logical CPU
         cores
     | utilization. Consider improving physical core utilization as
         the first
     | step and then look at opportunities to utilize logical cores
        , which in
     | some cases can improve processor throughput and overall
        performance of
     | multi-threaded applications.
     |
Collection and Platform Info
    Application Command Line: ./SysOLE "ex3.mtx"
    Operating System: 5.15.133.1-microsoft-standard-WSL2 DISTRIB_ID
      =Ubuntu DISTRIB_RELEASE=22.04 DISTRIB_CODENAME=jammy
      DISTRIB_DESCRIPTION="Ubuntu 22.04.3 LTS"
    Computer Name: PC-PC
    Result Size: 3.7 MB
    Collection start time: 23:42:51 06/12/2023 UTC
    Collection stop time: 23:42:54 06/12/2023 UTC
    Collector Type: Driverless Perf per-process counting,User-mode
      sampling and tracing
    CPU
        Name: Intel(R) microarchitecture code named Tigerlake H
        Frequency: 2.304 GHz
        Logical CPU Count: 16
        Cache Allocation Technology
            Level 2 capability: not detected
            Level 3 capability: not detected
```

### 2.4.3 gcov

```
Top Hotspots
Function             Module     CPU Time  % of CPU Time(%)
-------------------- ---------  --------  ----------------
ReadMMtoCSR          SysOLE      10.800s             82.6%
MatrixMultiply       SysOLE       1.670s             12.8%
ComputeNorm          SysOLE       0.300s              2.3%
Jacobi               SysOLE       0.210s              1.6%
__GI___isoc99_sscanf libc.so.6    0.050s              0.4%
[Others]             N/A          0.040s              0.3%
Effective Physical Core Utilization: 11.4% (0.914 out of 8)
 | The metric value is low, which may signal a poor physical CPU
    cores
 | utilization caused by:
 |     - load imbalance
 |     - threading runtime overhead
 |     - contended synchronization
 |     - thread/process underutilization
 |     - incorrect affinity that utilizes logical cores instead of
   physical
 |       cores
 | Explore sub-metrics to estimate the efficiency of MPI and OpenMP
     parallelism
 | or run the Locks and Waits analysis to identify parallel
   bottlenecks for
 | other parallel runtimes.
 |
    Effective Logical Core Utilization: 6.0% (0.964 out of 16)
     | The metric value is low, which may signal a poor logical CPU
        cores
     | utilization. Consider improving physical core utilization as
        the first
     | step and then look at opportunities to utilize logical cores
       , which in
     | some cases can improve processor throughput and overall
       performance of
     | multi-threaded applications.
     |
Collection and Platform Info
```

```
    Application Command Line: ./SysOLE "jnlbrng1.mtx"
    Operating System: 5.15.133.1-microsoft-standard-WSL2 DISTRIB_ID
      =Ubuntu DISTRIB_RELEASE=22.04 DISTRIB_CODENAME=jammy
      DISTRIB_DESCRIPTION="Ubuntu 22.04.3 LTS"
    Computer Name: PC-PC
    Result Size: 4.1 MB
    Collection start time: 23:44:59 06/12/2023 UTC
    Collection stop time: 23:45:12 06/12/2023 UTC
    Collector Type: Driverless Perf per-process counting,User-mode
      sampling and tracing
    CPU
        Name: Intel(R) microarchitecture code named Tigerlake H
        Frequency: 2.304 GHz
        Logical CPU Count: 16
        Cache Allocation Technology
            Level 2 capability: not detected
            Level 3 capability: not detected
```

### 2.4.4 gcov

Below is a collection of the 3 gcov reports for `functions.c` (as it's the file that has the most computational processes) for running `b1_ss.mtx`, `LFAT5.mtx`, and `LF10.mtx`, in order.

```
-:    0:Source:functions.c
        -:    0:Graph:functions.gcno
        -:    0:Data:functions.gcda
        -:    0:Runs:1
        -:    1:#include <stdlib.h>
        -:    2:#include <stdio.h>
        -:    3:#include <string.h>
        -:    4:#include <ctype.h>
        -:    5:#include <math.h>
        -:    6:#include <png.h>
        -:    7:#include "functions.h"
        -:    8:
        1:    9:void ReadMMtoCSR(const char *filename, CSRMatrix *
          matrix)
        -:   10:{
        -:   11:    FILE *source;
        1:   12:    source = fopen(filename, "r");
        -:   13:
        1:   14:    if (source == NULL)
        -:   15:    {
    #####:   16:        printf("Error in opening file\n");
    #####:   17:        exit(0);
        -:   18:    }
        -:   19:
        -:   20:    char test_line[256];
        1:   21:    int ignore = 0;
        1:   22:    int line_num = 0;
        1:   23:    int ignore_first_line = 0;
        -:   24:    int *all_rows;
        -:   25:    int *all_cols;
        -:   26:    double *all_values;
        -:   27:
       30:   28:    while (fgets(test_line, sizeof(test_line),
         source) != NULL)
        -:   29:    {
      160:   30:        for (int i = 0; i < strlen(test_line); i++)
         // Checks if the line is a valid line
        -:   31:        {
      144:   32:            if (!isdigit(test_line[i]) && (
        test_line[i] != '.') && (test_line[i] != ' ') && (
        test_line[i] != '\n') && (test_line[i] != '-'))
        -:   33:            {
       13:   34:                ignore++; // Adds one to a "checker
          value" which indicates the current line is a line to be
         ignored
       13:   35:                break;
        -:   36:            }
        -:   37:        }
        -:   38:
       29:   39:        if (ignore == 1) // Condition to check if
         the line is a line to be ignored
        -:   40:        {
       13:   41:            ignore--;
        -:   42:        }
       16:   43:        else if (ignore == 0 && line_num == 0) //
```

```
Condition to check if the line is the first line of "important" values (#Rows, #Columns, #Non-zero entries)
        -:   44:        {
        1:   45:            sscanf(test_line, "%d %d %d", &matrix->num_rows, &matrix->num_cols, &matrix->num_non_zeros);
        -:   46:
        1:   47:            all_rows = (int *)malloc(matrix->num_non_zeros * sizeof(int));
        1:   48:            all_cols = (int *)malloc(matrix->num_non_zeros * sizeof(int));
        1:   49:            all_values = (double *)malloc(matrix->num_non_zeros * sizeof(double));
        -:   50:
        1:   51:            if (all_rows == NULL || all_cols == NULL || all_values == NULL)
        -:   52:            {
    #####:   53:                printf("Memory allocation failed\n");
    #####:   54:                return;
        -:   55:            }
        -:   56:            //printf("%d %d %d\n", matrix->num_rows, matrix->num_cols, matrix->num_non_zeros);
        1:   57:            line_num++;
        -:   58:        }
        -:   59:
       29:   60:        if (ignore == 0 && line_num >= 1 && ignore_first_line != 0) // Executes for all lines that contain the non-zero values and their indices
        -:   61:        {
       15:   62:            sscanf(test_line, "%d %d %lf", &all_rows[line_num - 1], &all_cols[line_num - 1], &all_values[line_num - 1]);
        -:   63:            // printf("%d %d %.16f\n", all_rows[line_num - 1], all_cols[line_num - 1], all_values[line_num - 1]);
       15:   64:            line_num++;
        -:   65:        }
       14:   66:        else if (ignore == 0 && line_num >= 1 && ignore_first_line == 0) // Don't assign first line values
```

```
      to actual non-zero values arrays
        -:   67:        {
        1:   68:            ignore_first_line++;
        -:   69:        }
        -:   70:    }
        -:   71:
        1:   72:    matrix->row_ptr = (int *)calloc(matrix->
      num_rows + 1, sizeof(int)); // Allocate +1 to
      accommodate for num_non_zeros
        1:   73:    matrix->col_ind = (int *)malloc(matrix->
      num_non_zeros * sizeof(int));
        1:   74:    matrix->csr_data = (double *)malloc(matrix->
      num_non_zeros * sizeof(double));
        -:   75:
        1:   76:    if (matrix->row_ptr == NULL || matrix->col_ind
      == NULL || matrix->csr_data == NULL)
        -:   77:    {
    #####:   78:        printf("Memory allocation failed\n");
    #####:   79:        return;
        -:   80:    }
        -:   81:
        1:   82:    matrix->row_ptr[matrix->num_rows + 1] = matrix
      ->num_non_zeros; // Sets the last element to number of
      nonzeros
        -:   83:
       16:   84:    for (int i = 0; i < matrix->num_non_zeros; i++)
        -:   85:    {
       15:   86:        matrix->row_ptr[all_rows[i]]++;
        -:   87:    }
        -:   88:
        9:   89:    for (int i = 0; i <= matrix->num_rows; i++)
        -:   90:    {
        8:   91:        matrix->row_ptr[i] += matrix->row_ptr[i -
      1];
        -:   92:    }
        -:   93:
        1:   94:    int indexes[matrix->num_non_zeros];
        1:   95:    int index_val = 0;
        9:   96:    for (int i = 0; i <= matrix->num_rows; i++)
        -:   97:    {
      128:   98:        for (int j = 0; j < matrix->num_non_zeros; j++)
        -:   99:        {
      120:  100:            if (all_rows[j] == i)
        -:  101:            {
       15:  102:                indexes[index_val] = j;
       15:  103:                index_val++;
        -:  104:            }
        -:  105:        }
        -:  106:    }
        -:  107:
       16:  108:    for (int i = 0; i < matrix->num_non_zeros; i++)
        -:  109:    {
       15:  110:        matrix->col_ind[i] = all_cols[indexes[i]] - 1;
       15:  111:        matrix->csr_data[i] = all_values[indexes[i]];
        -:  112:    }
        -:  113:
        -:  114:    // Print Statements
        -:  115:    // Uncomment these print statements for the same output as the first part of the assignment requires.
        -:  116:    /*
        -:  117:    printf("Number of non-zero entries: %d\n", matrix->num_non_zeros);
        -:  118:    printf("Row Pointers: ");
        105:  179:            if (i == row_indexes[j])
        -:  180:            {
       15:  181:                y[i] += x[A->col_ind[j]] * A->csr_data[j];
        -:  182:            }
        -:  183:        }
        -:  184:    }
        -:  185:
        1:  186:    printf("Result: \n");
        8:  187:    for (int i = 0; i < A->num_cols; i++)
        -:  188:    {
```

```
        7:  189:            printf("%f ", y[i]);
        -:  190:        }
        1:  191:        printf("\n");
        1:  192:}
        -:  193:
        1:  194:int NumDiagonals(const CSRMatrix *A)
        1:  195:{
        1:  196:        int row_indexes[A->num_non_zeros];
        1:  197:        int row_value = 0;
        1:  198:        int row_index_index = 0;
        -:  199:
        8:  200:        for (int i = 1; i < A->num_cols + 1; i++)
        -:  201:        { // Creating a new array which contains all
           the row indices for all non zero values
       22:  202:            for (int j = 0; j < (A->row_ptr[i] - A->
          row_ptr[i - 1]); j++)
        -:  203:            {
       15:  204:                row_indexes[row_index_index] =
          row_value;
       15:  205:                row_index_index++;
        -:  206:            }
        7:  207:            row_value++;
        -:  208:        }
        -:  209:
        1:  210:        int num_diagonal_entries = 0;
       16:  211:        for (int i = 0; i < A->num_non_zeros; i++)
        -:  212:        {
       15:  213:            if (row_indexes[i] == A->col_ind[i])
        -:  214:            {
        6:  215:                num_diagonal_entries++;
        -:  216:            }
        -:  217:        }
        1:  218:        return num_diagonal_entries;
        -:  219:}
        -:  220:
    #####:  221:void Symmetrify(const CSRMatrix *A, int *
       symmetrified_rows, int *symmetrified_col, double *
       symmetrified_values, const int num_diagonal_entries)
    #####:  222:{
```

```
    #####:  223:    int row_indexes[A->num_non_zeros];
    #####:  224:    int row_value = 0;
    #####:  225:    int row_index_index = 0;
        -:  226:
    #####:  227:    for (int i = 1; i < A->num_cols + 1; i++)
        -:  228:    { // Creating a new array which contains all the row indices for all non zero values
    #####:  229:        for (int j = 0; j < (A->row_ptr[i] - A->row_ptr[i - 1]); j++)
        -:  230:        {
    #####:  231:            row_indexes[row_index_index] = row_value;
    #####:  232:            row_index_index++;
        -:  233:        }
    #####:  234:        row_value++;
        -:  235:    }
        -:  236:
    #####:  237:    int all_row_indexes[2 * A->num_non_zeros - num_diagonal_entries];
    #####:  238:    int all_col_indexes[2 * A->num_non_zeros - num_diagonal_entries];
    #####:  239:    double symmetric_all_values[2 * A->num_non_zeros - num_diagonal_entries];
    #####:  240:    int pos = 0;
        -:  241:
    #####:  242:    for (int i = 0; i < A->num_non_zeros; i++)
        -:  243:    {
    #####:  244:        all_row_indexes[i] = row_indexes[i];
    #####:  245:        all_col_indexes[i] = A->col_ind[i];
    #####:  246:        symmetric_all_values[i] = A->csr_data[i];
    #####:  247:        if (row_indexes[i] != A->col_ind[i])
        -:  248:        {
    #####:  249:            all_row_indexes[A->num_non_zeros + pos] = A->col_ind[i];
    #####:  250:            all_col_indexes[A->num_non_zeros + pos] = row_indexes[i];
    #####:  251:            symmetric_all_values[A->num_non_zeros + pos] = A->csr_data[i];
    #####:  252:            pos++;
        -:  253:        }
        -:  254:    }
        -:  255:
    #####:  256:    int test_if_in_order = 2 * A->num_non_zeros -
       num_diagonal_entries;
    #####:  257:    int switched_row_index = 0;
    #####:  258:    int switched_col_index = 0;
    #####:  259:    double switched_value = 0.0;
        -:  260:
    #####:  261:    while (test_if_in_order != 0)
        -:  262:    { // Arranging rows
    #####:  263:        for (int i = 0; i < 2 * A->num_non_zeros -
       num_diagonal_entries; i++)
        -:  264:        {
    #####:  265:            if ((i != 2 * A->num_non_zeros -
       num_diagonal_entries - 1) && (all_row_indexes[i + 1] ==
       all_row_indexes[i]))
        -:  266:            {
    #####:  267:                test_if_in_order--;
        -:  268:            }
    #####:  269:            else if ((i != 2 * A->num_non_zeros -
       num_diagonal_entries - 1) && (all_row_indexes[i + 1] -
       all_row_indexes[i] == 1))
        -:  270:            {
    #####:  271:                test_if_in_order--;
        -:  272:            }
    #####:  273:            else if ((i != 2 * A->num_non_zeros -
       num_diagonal_entries - 1) && (all_row_indexes[i + 1] -
       all_row_indexes[i] < 0))
        -:  274:            {
    #####:  275:                switched_row_index =
       all_row_indexes[i + 1];
    #####:  276:                switched_col_index =
       all_col_indexes[i + 1];
    #####:  277:                switched_value =
       symmetric_all_values[i + 1];
    #####:  278:                all_row_indexes[i + 1] =
       all_row_indexes[i];
    #####:  279:                all_col_indexes[i + 1] =
       all_col_indexes[i];
    #####:  280:                symmetric_all_values[i + 1] =
       symmetric_all_values[i];
    #####:  281:                all_row_indexes[i] =
       switched_row_index;
    #####:  282:                all_col_indexes[i] =
       switched_col_index;
    #####:  283:                symmetric_all_values[i] =
       switched_value;
        -:  284:            }
        -:  285:        }
    #####:  286:        if (test_if_in_order != 0)
        -:  287:        {
    #####:  288:            test_if_in_order = 2 * A->num_non_zeros
        - num_diagonal_entries - 1;
        -:  289:        }
        -:  290:    }
        -:  291:
    #####:  292:    while (test_if_in_order != 2 * A->num_non_zeros
        - num_diagonal_entries)
        -:  293:    {
        #####:  294:        for (int i = 0; i < 2 * A->
           num_non_zeros - num_diagonal_entries; i++)
        -:  295:        {
    #####:  296:            if ((all_row_indexes[i + 1] !=
       all_row_indexes[i]) || (i == 2 * A->num_non_zeros -
       num_diagonal_entries - 1))
        -:  297:            {
    #####:  298:                test_if_in_order++;
        -:  299:            }
    #####:  300:            else if ((all_col_indexes[i + 1] -
       all_col_indexes[i] >= 1))
        -:  301:            {
    #####:  302:                test_if_in_order++;
        -:  303:            }
    #####:  304:            else if ((all_col_indexes[i + 1] -
       all_col_indexes[i] < 0))
        -:  305:            {
    #####:  306:                switched_col_index =
       all_col_indexes[i + 1];
    #####:  307:                switched_value =
       symmetric_all_values[i + 1];
    #####:  308:                all_col_indexes[i + 1] =
       all_col_indexes[i];
    #####:  309:                all_col_indexes[i] =
       switched_col_index;
        -:  310:            }
        -:  311:        }
    #####:  312:        if (test_if_in_order != 2 * A->
       num_non_zeros - num_diagonal_entries)
        -:  313:        {
    #####:  314:            test_if_in_order = 0;
        -:  315:        }
        -:  316:    }
        -:  317:
    #####:  318:    for (int i = 0; i < 2 * A->num_non_zeros -
       num_diagonal_entries; i++)
        -:  319:    {
    #####:  320:        symmetrified_rows[i] = all_row_indexes[i];
    #####:  321:        symmetrified_col[i] = all_col_indexes[i];
    #####:  322:        symmetrified_values[i] =
       symmetric_all_values[i];
        -:  323:    }
    #####:  324:}
        -:  325:
    #####:  326:void MatrixMultiply(const CSRMatrix *A, const int *
       all_rows, const int *all_cols, const double *all_values,
       const double *x, double *result, int num_diagonals)
        -:  327:{
        -:  328:    // Initialize result vector to zero
    #####:  329:    for (int i = 0; i < A->num_rows; i++)
        -:  330:    {
    #####:  331:        result[i] = 0.0;
        -:  332:    }
        -:  333:
        -:  334:    // Perform matrix-vector multiplication
    #####:  335:    for (int i = 0; i < A->num_rows; i++)
        -:  336:    {
```

```
#####:  337:        for (int j = 0; j < 2 * A->num_non_zeros -
   num_diagonals; j++)
    -:  338:        {
#####:  339:            if (i < all_rows[j])
    -:  340:            {
#####:  341:                break;
    -:  342:            }
#####:  343:            else if (i == all_rows[j])
    -:  344:            {
#####:  345:                result[i] += x[all_cols[j]] *
   all_values[j];
    -:  346:            }
    -:  347:        }
    -:  348:    }
#####:  349:}
-:  350:
    1:  351:void Jacobi(const CSRMatrix *A, double *b, double *
       x, int num_iterations)
    -:  352:{
    -:  353:
    1:  354:    int num_diagonal_entries = NumDiagonals(A);
    1:  355:    if (num_diagonal_entries != A->num_cols)
    -:  356:    {
    1:  357:        printf("Not all diagonal entries are non-
       zero. Jacobi approach will fail.\n");
    8:  358:        for (int i = 0; i < A->num_cols; i++)
    -:  359:        { // Initial guess for x is all 1 for
       simple matrix multiplication.
    7:  360:            x[i] = 1.0;
    -:  361:        }
    1:  362:        spmv_csr(A, x, b);
    1:  363:        return;
    -:  364:    }
#####:  365:    int AllRows[2 * A->num_non_zeros -
   num_diagonal_entries];
#####:  366:    int AllCols[2 * A->num_non_zeros -
   num_diagonal_entries];
#####:  367:    double AllVals[2 * A->num_non_zeros -
   num_diagonal_entries];
```

```
    #####:  368:    Symmetrify(A, AllRows, AllCols, AllVals, num_diagonal_entries);
    #####:  369:    double diagonal_entries[num_diagonal_entries]; // A list of all diagonal entries, used for algebra.
    #####:  370:    int index_of_diagonal_entry = 0; // Need this for program logic
        -:  371:
    #####:  372:    for (int i = 0; i < 2 * A->num_non_zeros - num_diagonal_entries; i++)
        -:  373:    {
    #####:  374:        if (AllRows[i] == AllCols[i])
        -:  375:        {
    #####:  376:            diagonal_entries[index_of_diagonal_entry] = AllVals[i];
    #####:  377:            index_of_diagonal_entry++;
        -:  378:        }
        -:  379:    }
        -:  380:
        -:  381:    /*
        -:  382:    for (int i = 0; i < 2 * num_diagonal_entries; i++)
        -:  383:    {
        -:  384:        printf("%.16f\n", diagonal_entries[i]);
        -:  385:    }
        -:  386:    */
        -:  387:
    #####:  388:    double result[A->num_cols]; //Result vector of A*x from Jacobi method
    #####:  389:    double norm_diff = 1.0; //Initialize and norm_diff
        -:  390:
    #####:  391:    for (int i = 0; i < A->num_cols; i++)
        -:  392:    { // Initial guess for x is all zeros.
    #####:  393:        x[i] = 0.0;
        -:  394:    }
    #####:  395:    for (int i = 0; i < num_iterations; i++)
        -:  396:    {
    #####:  397:        if (norm_diff < 1e-7) //Condition to check if norm difference is sufficient to stop iterations
        -:  398:        {
    #####:  399:            break;
        -:  400:        }
    #####:  401:        MatrixMultiply(A, AllRows, AllCols, AllVals, x, result, num_diagonal_entries);
    #####:  402:        for (int i = 0; i < A->num_cols; i++)
        -:  403:        {
    #####:  404:            x[i] = (b[i] - result[i] + diagonal_entries[i] * x[i]) / diagonal_entries[i]; //Algebra, this is based off of the numerical process for the Jacobi method
        -:  405:        }
    #####:  406:        norm_diff = ComputeNorm(A, b, result);
        -:  407:        //printf("%f\n", norm_diff);
        -:  408:    }
        -:  409:
        -:  410:    // This prints out the components of the solution vector
        -:  411:    /*
        -:  412:    printf("Result:\n");
        -:  413:    for (int i = 0; i < A->num_cols; i++)
        -:  414:    {
        -:  415:        printf("%f ", x[i]);
        -:  416:    }
        -:  417:    printf("\n");
        -:  418:    */
        -:  419:
    #####:  420:    printf("Residual Norm: %.6f\n", norm_diff);
        -:  421:
        -:  422:}
        -:  423:
    #####:  424:void ComputeResidual(const CSRMatrix *A, double *b, double *Ax)
    #####:  425:{
    #####:  426:    double residual[A->num_cols];
    #####:  427:    printf("Residual: [");
    #####:  428:    for (int i = 0; i < A->num_cols; i++)
        -:  429:    {
    #####:  430:        residual[i] = b[i] - Ax[i];
    #####:  431:        printf("%f, ", residual[i]);
        -:  432:    }
    #####:  433:    printf("]\n");
    #####:  434:}
        -:  435:
    #####:  436:double ComputeNorm(const CSRMatrix *A, double *b,
       double *Ax)
    #####:  437:{
    #####:  438:    double residual[A->num_cols];
    #####:  439:    double norm = 0.0;
    #####:  440:    for (int i = 0; i < A->num_cols; i++)
        -:  441:    {
    #####:  442:        residual[i] = b[i] - Ax[i];
    #####:  443:        norm += residual[i] * residual[i];
        -:  444:    }
    #####:  445:    norm = sqrt(norm);
    #####:  446:    return norm;
        -:  447:}
        -:  448:
    #####:  449:void save_sparsity_pattern(const CSRMatrix *A,
       const char *filename)
        -:  450:{
    #####:  451:    int width = A->num_cols * 10;  // Adjust
       dimensions for larger image
    #####:  452:    int height = A->num_rows * 10; // Adjust
       dimensions for larger image
    #####:  453:    png_bytep *row_pointers = (png_bytep *)malloc(
       height * sizeof(png_bytep));
    #####:  454:    if (row_pointers == NULL)
        -:  455:    {
    #####:  456:        fprintf(stderr, "Memory allocation failed\n
       ");
    #####:  457:        return;
        -:  458:    }
    #####:  459:    for (int i = 0; i < height; i++)
        -:  460:    {
    #####:  461:        row_pointers[i] = (png_byte *)malloc(width
       * sizeof(png_byte));
    #####:  462:        if (row_pointers[i] == NULL)
```

```
        -:  463:        {
    #####:  464:            fprintf(stderr, "Memory allocation failed\n");
    #####:  465:            return;
        -:  466:        }
        -:  467:    }
    #####:  468:    for (int i = 0; i < height; i++)
        -:  469:    {
    #####:  470:        for (int j = 0; j < width; j++)
        -:  471:        {-:  410:    // This prints out the components of the solution vector
        -:  411:    /*
        -:  412:    printf("Result:\n");
        -:  413:    for (int i = 0; i < A->num_cols; i++)
        -:  414:    {
        -:  415:        printf("%f ", x[i]);
        -:  416:    }
        -:  417:    printf("\n");
        -:  418:    */
        -:  419:
    #####:  420:    printf("Residual Norm: %.6f\n", norm_diff);
        -:  421:
        -:  422:}
        -:  423:
    #####:  424:void ComputeResidual(const CSRMatrix *A, double *b, double *Ax)
    #####:  425:{
    #####:  426:    double residual[A->num_cols];
    #####:  427:    printf("Residual: [");
    #####:  428:    for (int i = 0; i < A->num_cols; i++)
        -:  429:    {
    #####:  430:        residual[i] = b[i] - Ax[i];
    #####:  431:        printf("%f, ", residual[i]);
        -:  432:    }
    #####:  433:    printf("]\n");
    #####:  434:}
        -:  435:
    #####:  436:double ComputeNorm(const CSRMatrix *A, double *b, double *Ax)
    #####:  437:{
    #####:  438:    double residual[A->num_cols];
    #####:  439:    double norm = 0.0;
    #####:  440:    for (int i = 0; i < A->num_cols; i++)
        -:  441:    {
    #####:  442:        residual[i] = b[i] - Ax[i];
    #####:  443:        norm += residual[i] * residual[i];
        -:  444:    }
    #####:  445:    norm = sqrt(norm);
    #####:  446:    return norm;
        -:  447:}
        -:  448:
    #####:  449:void save_sparsity_pattern(const CSRMatrix *A, const char *filename)
        -:  450:{
    #####:  451:    int width = A->num_cols * 10;  // Adjust dimensions for larger image
    #####:  452:    int height = A->num_rows * 10; // Adjust dimensions for larger image
    #####:  453:    png_bytep *row_pointers = (png_bytep *)malloc(height * sizeof(png_bytep));
    #####:  454:    if (row_pointers == NULL)
        -:  455:    {
    #####:  456:        fprintf(stderr, "Memory allocation failed\n");
    #####:  457:        return;
        -:  458:    }
    #####:  459:    for (int i = 0; i < height; i++)
        -:  460:    {
    #####:  461:        row_pointers[i] = (png_byte *)malloc(width * sizeof(png_byte));
    #####:  462:        if (row_pointers[i] == NULL)
        -:  463:        {
    #####:  464:            fprintf(stderr, "Memory allocation failed\n");
    #####:  465:            return;
        -:  466:        }
        -:  467:    }
    #####:  468:    for (int i = 0; i < height; i++)
        -:  469:    {
    #####:  470:        for (int j = 0; j < width; j++)
        -:  471:        {
        #####:  472:            row_pointers[i][j] = 255; // Initialize to white (255 = white in grayscale)
        -:  473:            // Set a larger block for non-zero elements
    #####:  474:            if (i % 10 == 0 && j % 10 == 0)
        -:  475:            {
    #####:  476:                for (int k = 0; k < 10; k++)
        -:  477:                {
    #####:  478:                    for (int l = 0; l < 10; l++)
        -:  479:                    {
    #####:  480:                        if ((i + k) < height && (j + l) < width)
        -:  481:                        {
    #####:  482:                            row_pointers[i + k][j + l] = 255; // Adjust the block color if needed
        -:  483:                        }
        -:  484:                    }
        -:  485:                }
        -:  486:            }
        -:  487:            // Check if the element at (i, j) is non-zero
    #####:  488:            for (int k = A->row_ptr[i / 10]; k < A->row_ptr[(i / 10) + 1]; k++)
        -:  489:            {
    #####:  490:                if (A->col_ind[k] == (j / 10))
        -:  491:                {
    #####:  492:                    row_pointers[i][j] = 0; // Set to black
    #####:  493:                    break;
        -:  494:                }
        -:  495:            }
        -:  496:        }
        -:  497:    }
    #####:  498:    FILE *fp = fopen(filename, "wb");
    #####:  499:    if (!fp)
        -:  500:    {
```

```
    #####:  501:        fprintf(stderr, "Error opening file for
       writing\n");
    #####:  502:        return;
        -:  503:    }
    #####:  504:    png_structp png_ptr = png_create_write_struct(
       PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    #####:  505:    if (!png_ptr)
        -:  506:    {
    #####:  507:        fprintf(stderr, "Error creating PNG write
       struct\n");
    #####:  508:        fclose(fp);
    #####:  509:        return;
        -:  510:    }
    #####:  511:    png_infop info_ptr = png_create_info_struct(
       png_ptr);
    #####:  512:    if (!info_ptr)
        -:  513:    {
    #####:  514:        fprintf(stderr, "Error creating PNG info
       struct\n");
    #####:  515:        png_destroy_write_struct(&png_ptr, NULL);
    #####:  516:        fclose(fp);
    #####:  517:        return;
        -:  518:    }
    #####:  519:    png_set_IHDR(png_ptr, info_ptr, width, height,
       8, PNG_COLOR_TYPE_GRAY, PNG_INTERLACE_NONE,
       PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);
    #####:  520:    png_set_rows(png_ptr, info_ptr, row_pointers);
    #####:  521:    png_set_filter(png_ptr, 0, PNG_FILTER_NONE);
    #####:  522:    png_init_io(png_ptr, fp);
    #####:  523:    png_write_png(png_ptr, info_ptr,
       PNG_TRANSFORM_IDENTITY, NULL);
    #####:  524:    fclose(fp);
    #####:  525:    png_destroy_write_struct(&png_ptr, &info_ptr);
    #####:  526:    for (int i = 0; i < height; i++)
        -:  527:    {
    #####:  528:        free(row_pointers[i]);
        -:  529:    }
    #####:  530:    free(row_pointers);
        -:  531:}
```

```
-:    0:Source:functions.c
    -:    0:Graph:functions.gcno
    -:    0:Data:functions.gcda
    -:    0:Runs:1
    -:    1:#include <stdlib.h>
    -:    2:#include <stdio.h>
    -:    3:#include <string.h>
    -:    4:#include <ctype.h>
    -:    5:#include <math.h>
    -:    6:#include <png.h>
    -:    7:#include "functions.h"
    -:    8:
    1:    9:void ReadMMtoCSR(const char *filename, CSRMatrix *
       matrix)
    -:   10:{
    -:   11:    FILE *source;
    1:   12:    source = fopen(filename, "r");
    -:   13:
    1:   14:    if (source == NULL)
    -:   15:    {
#####:   16:        printf("Error in opening file\n");
#####:   17:        exit(0);
    -:   18:    }
    -:   19:
    -:   20:    char test_line[256];
    1:   21:    int ignore = 0;
    1:   22:    int line_num = 0;
    1:   23:    int ignore_first_line = 0;
    -:   24:    int *all_rows;
    -:   25:    int *all_cols;
    -:   26:    double *all_values;
    -:   27:
   49:   28:    while (fgets(test_line, sizeof(test_line),
      source) != NULL)
    -:   29:    {
  538:   30:        for (int i = 0; i < strlen(test_line); i++)
      // Checks if the line is a valid line
    -:   31:        {
  512:   32:            if (!isdigit(test_line[i]) && (test_line[i] != '.') && (test_line[i] != ' ') && (test_line[i] != '\n') && (test_line[i] != '-'))
    -:   33:            {
   22:   34:                ignore++; // Adds one to a "checker value" which indicates the current line is a line to be ignored
   22:   35:                break;
    -:   36:            }
    -:   37:        }
    -:   38:
   48:   39:        if (ignore == 1) // Condition to check if the line is a line to be ignored
    -:   40:        {
   22:   41:            ignore--;
    -:   42:        }
   26:   43:        else if (ignore == 0 && line_num == 0) // Condition to check if the line is the first line of "important" values (#Rows, #Columns, #Non-zero entries)
    -:   44:        {
    1:   45:            sscanf(test_line, "%d %d %d", &matrix->num_rows, &matrix->num_cols, &matrix->num_non_zeros);
    -:   46:
    1:   47:            all_rows = (int *)malloc(matrix->num_non_zeros * sizeof(int));
    1:   48:            all_cols = (int *)malloc(matrix->num_non_zeros * sizeof(int));
    1:   49:            all_values = (double *)malloc(matrix->num_non_zeros * sizeof(double));
    -:   50:
    1:   51:            if (all_rows == NULL || all_cols == NULL || all_values == NULL)
    -:   52:            {
#####:   53:                printf("Memory allocation failed\n");
#####:   54:                return;
    -:   55:            }
    -:   56:            //printf("%d %d %d\n", matrix->num_rows, matrix->num_cols, matrix->num_non_zeros);
    1:   57:            line_num++;
    -:   58:        }
    -:   59:
    48:   60:        if (ignore == 0 && line_num >= 1 && ignore_first_line != 0) // Executes for all lines that contain the non-zero values and their indices
    -:   61:        {
   30:   62:            sscanf(test_line, "%d %d %lf", &all_rows[line_num - 1], &all_cols[line_num - 1], &all_values[line_num - 1]);
    -:   63:            // printf("%d %d %.16f\n", all_rows[line_num - 1], all_cols[line_num - 1], all_values[line_num - 1]);
   30:   64:            line_num++;
    -:   65:        }
   18:   66:        else if (ignore == 0 && line_num >= 1 && ignore_first_line == 0) // Don't assign first line values to actual non-zero values arrays
    -:   67:        {
    1:   68:            ignore_first_line++;
    -:   69:        }
    -:   70:    }
    -:   71:
    1:   72:    matrix->row_ptr = (int *)calloc(matrix->num_rows + 1, sizeof(int)); // Allocate +1 to accommodate for num_non_zeros
    1:   73:    matrix->col_ind = (int *)malloc(matrix->num_non_zeros * sizeof(int));
    1:   74:    matrix->csr_data = (double *)malloc(matrix->num_non_zeros * sizeof(double));
    -:   75:
    1:   76:    if (matrix->row_ptr == NULL || matrix->col_ind == NULL || matrix->csr_data == NULL)
    -:   77:    {
#####:   78:        printf("Memory allocation failed\n");
#####:   79:        return;
    -:   80:    }
    -:   81:
    1:   82:    matrix->row_ptr[matrix->num_rows + 1] = matrix
```

```
           ->num_non_zeros; // Sets the last element to number of
           nonzeros
        -:   83:
       31:   84:    for (int i = 0; i < matrix->num_non_zeros; i++)
        -:   85:    {
       30:   86:        matrix->row_ptr[all_rows[i]]++;
        -:   87:    }
        -:   88:
       16:   89:    for (int i = 0; i <= matrix->num_rows; i++)
        -:   90:    {
       15:   91:        matrix->row_ptr[i] += matrix->row_ptr[i -
          1];
        -:   92:    }
        -:   93:
        1:   94:    int indexes[matrix->num_non_zeros];
        1:   95:    int index_val = 0;
       16:   96:    for (int i = 0; i <= matrix->num_rows; i++)
        -:   97:    {
      465:   98:        for (int j = 0; j < matrix->num_non_zeros;
         j++)
        -:   99:        {
      450:  100:            if (all_rows[j] == i)
        -:  101:            {
       30:  102:                indexes[index_val] = j;
       30:  103:                index_val++;
        -:  104:            }
        -:  105:        }
        -:  106:    }
        -:  107:
       31:  108:    for (int i = 0; i < matrix->num_non_zeros; i++)
        -:  109:    {
       30:  110:        matrix->col_ind[i] = all_cols[indexes[i]] -
           1;
       30:  111:        matrix->csr_data[i] = all_values[indexes[i
          ]];
        -:  112:    }
        -:  113:
        -:  114:    // Print Statements
        -:  115:    // Uncomment these print statements for the
```

```
same output as the first part of the assignment requires.
    -:  116:    /*
    -:  117:    printf("Number of non-zero entries: %d\n", matrix->num_non_zeros);
    -:  118:    printf("Row Pointers: ");
    -:  119:    for (int i = 0; i <= matrix->num_rows; i++)
    -:  120:    {
    -:  121:        printf("%d ", matrix->row_ptr[i]);
    -:  122:    }
    -:  123:    printf("\n");
    -:  124:
    -:  125:    printf("Column Indexes: ");
    -:  126:    for (int i = 0; i < matrix->num_non_zeros; i++)
    -:  127:    {
    -:  128:        printf("%d ", matrix->col_ind[i]);
    -:  129:    }
    -:  130:    printf("\n");
    -:  131:
    -:  132:    printf("Values: ");
    -:  133:    for (int i = 0; i < matrix->num_non_zeros; i++)
    -:  134:    {
    -:  135:        printf("%.4f ", matrix->csr_data[i]);
    -:  136:    }
    -:  137:    printf("\n\n");
    -:  138:    */
    -:  139:
    -:  140:
    -:  141:    // Print statements required by the sample output for later components of the assignment
    1:  142:    printf("Matrix Name: %s\n", filename);
    1:  143:    printf("Number of non-zero entries: %d\n", matrix->num_non_zeros);
    1:  144:    printf("The dimensions of the matrix: %d by %d\n", matrix->num_rows, matrix->num_cols);
    -:  145:
    -:  146:
    1:  147:    free(all_rows);
    1:  148:    free(all_cols);
```

```
        1:  149:    free(all_values);
        -:  150:
        1:  151:    fclose(source);
        -:  152:}
        -:  153:
    #####:  154:void spmv_csr(const CSRMatrix *A, const double *x,
       double *y) // Caluclating A*x and writing the result to y.
       IMPORTANT: This function does not consider if a matrix is
       symmetrical or not.
Therefore, for matricies like b1_ss.mtx, this function is called
   and only prints out the result of matrix multiplication with x[i
   ] = 1.0.
        -:  155:{
    #####:  156:    for (int i = 0; i < A->num_rows; i++)
        -:  157:    { // Initialize all elements of y to 0
    #####:  158:        y[i] = 0.0;
        -:  159:    }
        -:  160:
    #####:  161:    int row_indexes[A->num_non_zeros];
    #####:  162:    int row_value = 0;
    #####:  163:    int row_index_index = 0;
        -:  164:
    #####:  165:    for (int i = 1; i < A->num_cols + 1; i++)
        -:  166:    { // Creating a new array which contains all
           the row indices for all non zero values
    #####:  167:        for (int j = 0; j < (A->row_ptr[i] - A->
       row_ptr[i - 1]); j++)
        -:  168:        {
    #####:  169:            row_indexes[row_index_index] =
       row_value;
    #####:  170:            row_index_index++;
        -:  171:        }
    #####:  172:        row_value++;
    -:  173:    }
        -:  174:
    #####:  175:    for (int i = 0; i < A->num_rows; i++)
        -:  176:    { // Vector x HAS to equal the number of
           columns of A, otherwise we can't do a matrix
           multiplication
    #####:  177:        for (int j = 0; j < A->num_non_zeros; j++)
        -:  178:        {
    #####:  179:            if (i == row_indexes[j])
        -:  180:            {
    #####:  181:                y[i] += x[A->col_ind[j]] * A->
       csr_data[j];
        -:  182:            }
        -:  183:        }
        -:  184:    }
        -:  185:
    #####:  186:    printf("Result: \n");
    #####:  187:    for (int i = 0; i < A->num_cols; i++)
        -:  188:    {
    #####:  189:        printf("%f ", y[i]);
        -:  190:    }
    #####:  191:    printf("\n");
    #####:  192:}
        -:  193:
        1:  194:int NumDiagonals(const CSRMatrix *A)
        1:  195:{
        1:  196:    int row_indexes[A->num_non_zeros];
        1:  197:    int row_value = 0;
        1:  198:    int row_index_index = 0;
        -:  199:
       15:  200:    for (int i = 1; i < A->num_cols + 1; i++)
        -:  201:    { // Creating a new array which contains all
           the row indices for all non zero values
       44:  202:        for (int j = 0; j < (A->row_ptr[i] - A->
          row_ptr[i - 1]); j++)
        -:  203:        {
       30:  204:            row_indexes[row_index_index] =
          row_value;
       30:  205:            row_index_index++;
        -:  206:        }
       14:  207:        row_value++;
        -:  208:    }
        -:  209:
        1:  210:    int num_diagonal_entries = 0;
       31:  211:    for (int i = 0; i < A->num_non_zeros; i++)
```

```
        -:  212:    {
       30:  213:        if (row_indexes[i] == A->col_ind[i])
        -:  214:        {
       14:  215:            num_diagonal_entries++;
        -:  216:        }
        -:  217:    }
        1:  218:    return num_diagonal_entries;
        -:  219:}
        -:  220:
        1:  221:void Symmetrify(const CSRMatrix *A, int *
           symmetrified_rows, int *symmetrified_col, double *
           symmetrified_values, const int num_diagonal_entries)
        1:  222:{
        1:  223:    int row_indexes[A->num_non_zeros];
        1:  224:    int row_value = 0;
        1:  225:    int row_index_index = 0;
        -:  226:
       15:  227:    for (int i = 1; i < A->num_cols + 1; i++)
        -:  228:    { // Creating a new array which contains all
           the row indices for all non zero values
       44:  229:        for (int j = 0; j < (A->row_ptr[i] - A->
          row_ptr[i - 1]); j++)
        -:  230:        {
       30:  231:            row_indexes[row_index_index] =
          row_value;
       30:  232:            row_index_index++;
        -:  233:        }
       14:  234:        row_value++;
       -:  235:    }
        -:  236:
        1:  237:    int all_row_indexes[2 * A->num_non_zeros -
           num_diagonal_entries];
        1:  238:    int all_col_indexes[2 * A->num_non_zeros -
           num_diagonal_entries];
        1:  239:    double symmetric_all_values[2 * A->
           num_non_zeros - num_diagonal_entries];
        1:  240:    int pos = 0;
        -:  241:
       31:  242:    for (int i = 0; i < A->num_non_zeros; i++)
```

```
        -:  243:    {
       30:  244:        all_row_indexes[i] = row_indexes[i];
       30:  245:        all_col_indexes[i] = A->col_ind[i];
       30:  246:        symmetric_all_values[i] = A->csr_data[i];
       30:  247:        if (row_indexes[i] != A->col_ind[i])
        -:  248:        {
       16:  249:            all_row_indexes[A->num_non_zeros + pos]
           = A->col_ind[i];
       16:  250:            all_col_indexes[A->num_non_zeros + pos]
           = row_indexes[i];
       16:  251:            symmetric_all_values[A->num_non_zeros +
           pos] = A->csr_data[i];
       16:  252:            pos++;
        -:  253:        }
        -:  254:    }
        -:  255:
        1:  256:    int test_if_in_order = 2 * A->num_non_zeros -
           num_diagonal_entries;
        1:  257:    int switched_row_index = 0;
        1:  258:    int switched_col_index = 0;
        1:  259:    double switched_value = 0.0;
        -:  260:
       31:  261:    while (test_if_in_order != 0)
        -:  262:    { // Arranging rows
     1410:  263:        for (int i = 0; i < 2 * A->num_non_zeros -
        num_diagonal_entries; i++)
        -:  264:        {
     1380:  265:            if ((i != 2 * A->num_non_zeros -
        num_diagonal_entries - 1) && (all_row_indexes[i + 1] ==
        all_row_indexes[i]))
        -:  266:            {
      642:  267:                test_if_in_order--;
        -:  268:            }
      738:  269:            else if ((i != 2 * A->num_non_zeros -
         num_diagonal_entries - 1) && (all_row_indexes[i + 1] -
         all_row_indexes[i] == 1))
        -:  270:            {
      390:  271:                test_if_in_order--;
        -:  272:            }
      348:  273:            else if ((i != 2 * A->num_non_zeros -
   num_diagonal_entries - 1) && (all_row_indexes[i + 1] -
   all_row_indexes[i] < 0))
        -:  274:            {
      318:  275:                switched_row_index =
   all_row_indexes[i + 1];
      318:  276:                switched_col_index =
   all_col_indexes[i + 1];
      318:  277:                switched_value =
   symmetric_all_values[i + 1];
      318:  278:                all_row_indexes[i + 1] =
   all_row_indexes[i];
      318:  279:                all_col_indexes[i + 1] =
   all_col_indexes[i];
      318:  280:                symmetric_all_values[i + 1] =
   symmetric_all_values[i];
      318:  281:                all_row_indexes[i] =
   switched_row_index;
      318:  282:                all_col_indexes[i] =
   switched_col_index;
      318:  283:                symmetric_all_values[i] =
   switched_value;
        -:  284:            }
        -:  285:        }
       30:  286:        if (test_if_in_order != 0)
        -:  287:        {
       29:  288:            test_if_in_order = 2 * A->num_non_zeros
     - num_diagonal_entries - 1;
        -:  289:        }
        -:  290:    }
        -:  291:
        2:  292:    while (test_if_in_order != 2 * A->num_non_zeros
      - num_diagonal_entries)
        -:  293:    {
       47:  294:        for (int i = 0; i < 2 * A->num_non_zeros -
    num_diagonal_entries; i++)
        -:  295:        {
       46:  296:            if ((all_row_indexes[i + 1] !=
     all_row_indexes[i]) || (i == 2 * A->num_non_zeros -
```

```
       num_diagonal_entries - 1))
    -:  297:            {
   14:  298:                test_if_in_order++;
    -:  299:            }
   32:  300:            else if ((all_col_indexes[i + 1] -
      all_col_indexes[i] >= 1))
    -:  301:            {
   32:  302:                test_if_in_order++;
    -:  303:            }
#####:  304:            else if ((all_col_indexes[i + 1] -
   all_col_indexes[i] < 0))
    -:  305:            {
#####:  306:                switched_col_index =
   all_col_indexes[i + 1];
#####:  307:                switched_value =
   symmetric_all_values[i + 1];
#####:  308:                all_col_indexes[i + 1] =
   all_col_indexes[i];
#####:  309:                all_col_indexes[i] =
   switched_col_index;
    -:  310:            }
    -:  311:        }
    1:  312:        if (test_if_in_order != 2 * A->
       num_non_zeros - num_diagonal_entries)
    -:  313:        {
#####:  314:            test_if_in_order = 0;
    -:  315:        }
    -:  316:    }
    -:  317:
   47:  318:    for (int i = 0; i < 2 * A->num_non_zeros -
      num_diagonal_entries; i++)
    -:  319:    {
   46:  320:        symmetrified_rows[i] = all_row_indexes[i];
   46:  321:        symmetrified_col[i] = all_col_indexes[i];
   46:  322:        symmetrified_values[i] =
      symmetric_all_values[i];
    -:  323:    }
    1:  324:}
    -:  325:
     1248:  326:void MatrixMultiply(const CSRMatrix *A, const int *all_rows, const int *all_cols, const double *all_values, const double *x, double *result, int num_diagonals)
        -:  327:{
        -:  328:    // Initialize result vector to zero
    18720:  329:    for (int i = 0; i < A->num_rows; i++)
        -:  330:    {
    17472:  331:        result[i] = 0.0;
        -:  332:    }
        -:  333:
        -:  334:    // Perform matrix-vector multiplication
    18720:  335:    for (int i = 0; i < A->num_rows; i++)
        -:  336:    {
   433056:  337:        for (int j = 0; j < 2 * A->num_non_zeros - num_diagonals; j++)
        -:  338:        {
   431808:  339:            if (i < all_rows[j])
        -:  340:            {
    16224:  341:                break;
        -:  342:            }
   415584:  343:            else if (i == all_rows[j])
        -:  344:            {
    57408:  345:                result[i] += x[all_cols[j]] * all_values[j];
        -:  346:            }
        -:  347:        }
        -:  348:    }
     1248:  349:}
        -:  350:
        1:  351:void Jacobi(const CSRMatrix *A, double *b, double *x, int num_iterations)
        -:  352:{
        -:  353:
        1:  354:    int num_diagonal_entries = NumDiagonals(A);
        1:  355:    if (num_diagonal_entries != A->num_cols)
        -:  356:    {
    #####:  357:        printf("Not all diagonal entries are non-zero. Jacobi approach will fail.\n");
    #####:  358:        for (int i = 0; i < A->num_cols; i++)
```

```
        -:  359:            { // Initial guess for x is all 1 for simple matrix multiplication.
    #####:  360:                x[i] = 1.0;
        -:  361:            }
    #####:  362:            spmv_csr(A, x, b);
    #####:  363:            return;
        -:  364:        }
        1:  365:        int AllRows[2 * A->num_non_zeros - num_diagonal_entries];
        1:  366:        int AllCols[2 * A->num_non_zeros - num_diagonal_entries];
        1:  367:        double AllVals[2 * A->num_non_zeros - num_diagonal_entries];
        1:  368:        Symmetrify(A, AllRows, AllCols, AllVals, num_diagonal_entries);
        1:  369:        double diagonal_entries[num_diagonal_entries]; // A list of all diagonal entries, used for algebra.
        1:  370:        int index_of_diagonal_entry = 0; // Need this for program logic
        -:  371:
       47:  372:        for (int i = 0; i < 2 * A->num_non_zeros - num_diagonal_entries; i++)
        -:  373:        {
       46:  374:            if (AllRows[i] == AllCols[i])
        -:  375:            {
       14:  376:                diagonal_entries[index_of_diagonal_entry] = AllVals[i];
       14:  377:                index_of_diagonal_entry++;
        -:  378:            }
        -:  379:        }
        -:  380:
        -:  381:        /*
        -:  382:        for (int i = 0; i < 2 * num_diagonal_entries; i++)
        -:  383:        {
        -:  384:            printf("%.16f\n", diagonal_entries[i]);
        -:  385:        }
        -:  386:        */
        -:  387:
```

```
        1:  388:    double result[A->num_cols]; //Result vector of A*x from Jacobi method
        1:  389:    double norm_diff = 1.0; //Initialize and norm_diff
        -:  390:
       15:  391:    for (int i = 0; i < A->num_cols; i++)
        -:  392:    { // Initial guess for x is all zeros.
       14:  393:        x[i] = 0.0;
        -:  394:    }
     1249:  395:    for (int i = 0; i < num_iterations; i++)
        -:  396:    {
     1249:  397:        if (norm_diff < 1e-7) //Condition to check if norm difference is sufficient to stop iterations
        -:  398:        {
        1:  399:            break;
        -:  400:        }
     1248:  401:        MatrixMultiply(A, AllRows, AllCols, AllVals, x, result, num_diagonal_entries);
    18720:  402:        for (int i = 0; i < A->num_cols; i++)
        -:  403:        {
    17472:  404:            x[i] = (b[i] - result[i] + diagonal_entries[i] * x[i]) / diagonal_entries[i]; //Algebra, this is based off of the numerical process for the Jacobi method
        -:  405:        }
     1248:  406:        norm_diff = ComputeNorm(A, b, result);
        -:  407:        //printf("%f\n", norm_diff);
        -:  408:    }
        -:  409:
        -:  410:    // This prints out the components of the solution vector
        -:  411:    /*
        -:  412:    printf("Result:\n");
        -:  413:    for (int i = 0; i < A->num_cols; i++)
        -:  414:    {
        -:  415:        printf("%f ", x[i]);
        -:  416:    }
        -:  417:    printf("\n");
        -:  418:    */
        -:  419:
        1:  420:    printf("Residual Norm: %.6f\n", norm_diff);
        -:  421:
        -:  422:}
        -:  423:
    #####:  424:void ComputeResidual(const CSRMatrix *A, double *b, double *Ax)
    #####:  425:{
    #####:  426:    double residual[A->num_cols];
    #####:  427:    printf("Residual: [");
    #####:  428:    for (int i = 0; i < A->num_cols; i++)
        -:  429:    {
    #####:  430:        residual[i] = b[i] - Ax[i];
    #####:  431:        printf("%f, ", residual[i]);
        -:  432:    }
    #####:  433:    printf("]\n");
    #####:  434:}
        -:  435:
     1248:  436:double ComputeNorm(const CSRMatrix *A, double *b, double *Ax)
     1248:  437:{
     1248:  438:    double residual[A->num_cols];
     1248:  439:    double norm = 0.0;
    18720:  440:    for (int i = 0; i < A->num_cols; i++)
        -:  441:    {
    17472:  442:        residual[i] = b[i] - Ax[i];
    17472:  443:        norm += residual[i] * residual[i];
        -:  444:    }
     1248:  445:    norm = sqrt(norm);
     1248:  446:    return norm;
        -:  447:}
        -:  448:
    #####:  449:void save_sparsity_pattern(const CSRMatrix *A, const char *filename)
        -:  450:{
    #####:  451:    int width = A->num_cols * 10;  // Adjust dimensions for larger image
    #####:  452:    int height = A->num_rows * 10; // Adjust dimensions for larger image
```

```
#####:  453:    png_bytep *row_pointers = (png_bytep *)malloc(height * sizeof(png_bytep));
#####:  454:    if (row_pointers == NULL)
    -:  455:    {
#####:  456:        fprintf(stderr, "Memory allocation failed\n");
#####:  457:        return;
    -:  458:    }
#####:  459:    for (int i = 0; i < height; i++)
    -:  460:    {
#####:  461:        row_pointers[i] = (png_byte *)malloc(width * sizeof(png_byte));
#####:  462:        if (row_pointers[i] == NULL)
    -:  463:        {
#####:  464:            fprintf(stderr, "Memory allocation failed\n");
#####:  465:            return;
    -:  466:        }
    -:  467:    }
#####:  468:    for (int i = 0; i < height; i++)
    -:  469:    {
#####:  470:        for (int j = 0; j < width; j++)
    -:  471:        {
#####:  472:            row_pointers[i][j] = 255; // Initialize to white (255 = white in grayscale)
    -:  473:            // Set a larger block for non-zero elements
#####:  474:            if (i % 10 == 0 && j % 10 == 0)
    -:  475:            {
    #####:  476:                for (int k = 0; k < 10; k++)
    -:  477:                {
#####:  478:                    for (int l = 0; l < 10; l++)
    -:  479:                    {
#####:  480:                        if ((i + k) < height && (j + l) < width)
    -:  481:                        {
#####:  482:                            row_pointers[i + k][j + l] = 255; // Adjust the block color if needed
    -:  483:                        }
    -:  484:                    }
    -:  485:                }
    -:  486:            }
    -:  487:            // Check if the element at (i, j) is
       non-zero
#####:  488:            for (int k = A->row_ptr[i / 10]; k < A
   ->row_ptr[(i / 10) + 1]; k++)
    -:  489:            {
#####:  490:                if (A->col_ind[k] == (j / 10))
    -:  491:                {
#####:  492:                    row_pointers[i][j] = 0; // Set
   to black
#####:  493:                    break;
    -:  494:                }
    -:  495:            }
    -:  496:        }
    -:  497:    }
#####:  498:    FILE *fp = fopen(filename, "wb");
#####:  499:    if (!fp)
    -:  500:    {
#####:  501:        fprintf(stderr, "Error opening file for
   writing\n");
#####:  502:        return;
    -:  503:    }
#####:  504:    png_structp png_ptr = png_create_write_struct(
   PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
#####:  505:    if (!png_ptr)
    -:  506:    {
#####:  507:        fprintf(stderr, "Error creating PNG write
   struct\n");
#####:  508:        fclose(fp);
#####:  509:        return;
    -:  510:    }
#####:  511:    png_infop info_ptr = png_create_info_struct(
   png_ptr);
#####:  512:    if (!info_ptr)
    -:  513:    {
#####:  514:        fprintf(stderr, "Error creating PNG info
   struct\n");
```

```
    #####:  515:        png_destroy_write_struct(&png_ptr, NULL);
    #####:  516:        fclose(fp);
    #####:  517:        return;
        -:  518:    }
    #####:  519:    png_set_IHDR(png_ptr, info_ptr, width, height,
       8, PNG_COLOR_TYPE_GRAY, PNG_INTERLACE_NONE,
       PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);
    #####:  520:    png_set_rows(png_ptr, info_ptr, row_pointers);
    #####:  521:    png_set_filter(png_ptr, 0, PNG_FILTER_NONE);
    #####:  522:    png_init_io(png_ptr, fp);
    #####:  523:    png_write_png(png_ptr, info_ptr,
       PNG_TRANSFORM_IDENTITY, NULL);
    #####:  524:    fclose(fp);
    #####:  525:    png_destroy_write_struct(&png_ptr, &info_ptr);
    #####:  526:    for (int i = 0; i < height; i++)
        -:  527:    {
    #####:  528:        free(row_pointers[i]);
        -:  529:    }
    #####:  530:    free(row_pointers);
        -:  531:}
```

```
-:    0:Source:functions.c
        -:    0:Graph:functions.gcno
        -:    0:Data:functions.gcda
        -:    0:Runs:1
        -:    1:#include <stdlib.h>
        -:    2:#include <stdio.h>
        -:    3:#include <string.h>
        -:    4:#include <ctype.h>
        -:    5:#include <math.h>
        -:    6:#include <png.h>
        -:    7:#include "functions.h"
        -:    8:
        1:    9:void ReadMMtoCSR(const char *filename, CSRMatrix *
           matrix)
        -:   10:{
        -:   11:    FILE *source;
        1:   12:    source = fopen(filename, "r");
        -:   13:
        1:   14:    if (source == NULL)
        -:   15:    {
    #####:   16:        printf("Error in opening file\n");
    #####:   17:        exit(0);
        -:   18:    }
        -:   19:
        -:   20:    char test_line[256];
        1:   21:    int ignore = 0;
        1:   22:    int line_num = 0;
        1:   23:    int ignore_first_line = 0;
        -:   24:    int *all_rows;
        -:   25:    int *all_cols;
        -:   26:    double *all_values;
        -:   27:
       69:   28:    while (fgets(test_line, sizeof(test_line),
          source) != NULL)
        -:   29:    {
      997:   30:        for (int i = 0; i < strlen(test_line); i++)
          // Checks if the line is a valid line
        -:   31:        {
```

```
      946:   32:            if (!isdigit(test_line[i]) && (test_line[i] != '.') && (test_line[i] != ' ') && (test_line[i] != '\n') && (test_line[i] != '-'))
        -:   33:            {
       17:   34:                ignore++; // Adds one to a "checker value" which indicates the current line is a line to be ignored
       17:   35:                break;
        -:   36:            }
        -:   37:        }
        -:   38:
       68:   39:        if (ignore == 1) // Condition to check if the line is a line to be ignored
        -:   40:        {
       17:   41:            ignore--;
        -:   42:        }
       51:   43:        else if (ignore == 0 && line_num == 0) // Condition to check if the line is the first line of "important" values (#Rows, #Columns, #Non-zero entries)
        -:   44:        {
        1:   45:            sscanf(test_line, "%d %d %d", &matrix->num_rows, &matrix->num_cols, &matrix->num_non_zeros);
        -:   46:
        1:   47:            all_rows = (int *)malloc(matrix->num_non_zeros * sizeof(int));
        1:   48:            all_cols = (int *)malloc(matrix->num_non_zeros * sizeof(int));
        1:   49:            all_values = (double *)malloc(matrix->num_non_zeros * sizeof(double));
        -:   50:
        1:   51:            if (all_rows == NULL || all_cols == NULL || all_values == NULL)
        -:   52:            {
    #####:   53:                printf("Memory allocation failed\n");
    #####:   54:                return;
        -:   55:            }
        -:   56:            //printf("%d %d %d\n", matrix->num_rows, matrix->num_cols, matrix->num_non_zeros);
        1:   57:            line_num++;
        -:   58:        }
        -:   59:
        -:  178:        {
    #####:  179:            if (i == row_indexes[j])
        -:  180:            {
    #####:  181:                y[i] += x[A->col_ind[j]] * A->
       csr_data[j];
        -:  182:            }
        -:  183:        }
        -:  184:    }
        -:  185:
    #####:  186:    printf("Result: \n");
    #####:  187:    for (int i = 0; i < A->num_cols; i++)
        -:  188:    {
    #####:  189:        printf("%f ", y[i]);
        -:  190:    }
    #####:  191:    printf("\n");
    #####:  192:}
        -:  193:
        1:  194:int NumDiagonals(const CSRMatrix *A)
        1:  195:{
        1:  196:    int row_indexes[A->num_non_zeros];
        1:  197:    int row_value = 0;
        1:  198:    int row_index_index = 0;
        -:  199:
       19:  200:    for (int i = 1; i < A->num_cols + 1; i++)
        -:  201:    { // Creating a new array which contains all
           the row indices for all non zero values
       68:  202:        for (int j = 0; j < (A->row_ptr[i] - A->
          row_ptr[i - 1]); j++)
        -:  203:        {
       50:  204:            row_indexes[row_index_index] =
          row_value;
       50:  205:            row_index_index++;
        -:  206:        }
       18:  207:        row_value++;
        -:  208:    }
        -:  209:
        1:  210:    int num_diagonal_entries = 0;
       51:  211:    for (int i = 0; i < A->num_non_zeros; i++)
        -:  212:    {
       50:  213:        if (row_indexes[i] == A->col_ind[i])
        -:  214:        {
       18:  215:            num_diagonal_entries++;
        -:  216:        }
        -:  217:    }
        1:  218:    return num_diagonal_entries;
        -:  219:}
        -:  220:
        1:  221:void Symmetrify(const CSRMatrix *A, int *
           symmetrified_rows, int *symmetrified_col, double *
           symmetrified_values, const int num_diagonal_entries)
        1:  222:{
        1:  223:    int row_indexes[A->num_non_zeros];
        1:  224:    int row_value = 0;
        1:  225:    int row_index_index = 0;
        -:  226:
       19:  227:    for (int i = 1; i < A->num_cols + 1; i++)
        -:  228:    { // Creating a new array which contains all
           the row indices for all non zero values
       68:  229:        for (int j = 0; j < (A->row_ptr[i] - A->
          row_ptr[i - 1]); j++)
        -:  230:        {
       50:  231:            row_indexes[row_index_index] =
          row_value;
       50:  232:            row_index_index++;
        -:  233:        }
       18:  234:        row_value++;
        -:  235:    }
        -:  236:
        1:  237:    int all_row_indexes[2 * A->num_non_zeros -
           num_diagonal_entries];
        1:  238:    int all_col_indexes[2 * A->num_non_zeros -
           num_diagonal_entries];
        1:  239:    double symmetric_all_values[2 * A->
           num_non_zeros - num_diagonal_entries];
        1:  240:    int pos = 0;
        -:  241:
       51:  242:    for (int i = 0; i < A->num_non_zeros; i++)
        -:  243:    {
       50:  244:        all_row_indexes[i] = row_indexes[i];
       50:  245:        all_col_indexes[i] = A->col_ind[i];
       50:  246:        symmetric_all_values[i] = A->csr_data[i];
       50:  247:        if (row_indexes[i] != A->col_ind[i])
        -:  248:        {
       32:  249:            all_row_indexes[A->num_non_zeros + pos]
           = A->col_ind[i];
       32:  250:            all_col_indexes[A->num_non_zeros + pos]
           = row_indexes[i];
       32:  251:            symmetric_all_values[A->num_non_zeros +
           pos] = A->csr_data[i];
       32:  252:            pos++;
        -:  253:        }
        -:  254:    }
        -:  255:
        1:  256:    int test_if_in_order = 2 * A->num_non_zeros -
           num_diagonal_entries;
        1:  257:    int switched_row_index = 0;
        1:  258:    int switched_col_index = 0;
        1:  259:    double switched_value = 0.0;
        -:  260:
       51:  261:    while (test_if_in_order != 0)
        -:  262:    { // Arranging rows
     4150:  263:        for (int i = 0; i < 2 * A->num_non_zeros -
        num_diagonal_entries; i++)
        -:  264:        {
     4100:  265:            if ((i != 2 * A->num_non_zeros -
        num_diagonal_entries - 1) && (all_row_indexes[i + 1] ==
        all_row_indexes[i]))
        -:  266:            {
     2290:  267:                test_if_in_order--;
        -:  268:            }
     1810:  269:            else if ((i != 2 * A->num_non_zeros -
        num_diagonal_entries - 1) && (all_row_indexes[i + 1] -
        all_row_indexes[i] == 1))
        -:  270:            {
```

```
  850:  271:                test_if_in_order--;
    -:  272:            }
  960:  273:            else if ((i != 2 * A->num_non_zeros - num_diagonal_entries - 1) && (all_row_indexes[i + 1] - all_row_indexes[i] < 0))
    -:  274:            {
  910:  275:                switched_row_index = all_row_indexes[i + 1];
  910:  276:                switched_col_index = all_col_indexes[i + 1];
  910:  277:                switched_value = symmetric_all_values[i + 1];
  910:  278:                all_row_indexes[i + 1] = all_row_indexes[i];
  910:  279:                all_col_indexes[i + 1] = all_col_indexes[i];
  910:  280:                symmetric_all_values[i + 1] = symmetric_all_values[i];
  910:  281:                all_row_indexes[i] = switched_row_index;
  910:  282:                all_col_indexes[i] = switched_col_index;
  910:  283:                symmetric_all_values[i] = switched_value;
    -:  284:            }
    -:  285:        }
   50:  286:        if (test_if_in_order != 0)
    -:  287:        {
   49:  288:            test_if_in_order = 2 * A->num_non_zeros - num_diagonal_entries - 1;
    -:  289:        }
    -:  290:    }
    -:  291:
    2:  292:    while (test_if_in_order != 2 * A->num_non_zeros - num_diagonal_entries)
    -:  293:    {
   83:  294:        for (int i = 0; i < 2 * A->num_non_zeros - num_diagonal_entries; i++)
    -:  295:        {
```

```
   82:  296:                if ((all_row_indexes[i + 1] !=
      all_row_indexes[i]) || (i == 2 * A->num_non_zeros -
      num_diagonal_entries - 1))
    -:  297:                {
   18:  298:                    test_if_in_order++;
    -:  299:                }
   64:  300:                else if ((all_col_indexes[i + 1] -
      all_col_indexes[i] >= 1))
    -:  301:                {
   64:  302:                    test_if_in_order++;
    -:  303:                }
#####:  304:                else if ((all_col_indexes[i + 1] -
   all_col_indexes[i] < 0))
    -:  305:                {
#####:  306:                    switched_col_index =
   all_col_indexes[i + 1];
#####:  307:                    switched_value =
   symmetric_all_values[i + 1];
#####:  308:                    all_col_indexes[i + 1] =
   all_col_indexes[i];
#####:  309:                    all_col_indexes[i] =
   switched_col_index;
    -:  310:                }
    -:  311:            }
    1:  312:            if (test_if_in_order != 2 * A->
       num_non_zeros - num_diagonal_entries)
    -:  313:            {
#####:  314:                test_if_in_order = 0;
    -:  315:            }
    -:  316:        }
    -:  317:
   83:  318:        for (int i = 0; i < 2 * A->num_non_zeros -
      num_diagonal_entries; i++)
    -:  319:        {
   82:  320:            symmetrified_rows[i] = all_row_indexes[i];
   82:  321:            symmetrified_col[i] = all_col_indexes[i];
   82:  322:            symmetrified_values[i] =
      symmetric_all_values[i];
    -:  323:        }
```

```
        1:  324:}
        -:  325:
    10000:  326:void MatrixMultiply(const CSRMatrix *A, const int *
      all_rows, const int *all_cols, const double *all_values,
      const double *x, double *result, int num_diagonals)
        -:  327:{
        -:  328:    // Initialize result vector to zero
   190000:  329:    for (int i = 0; i < A->num_rows; i++)
        -:  330:    {
   180000:  331:        result[i] = 0.0;
        -:  332:    }
        -:  333:
        -:  334:    // Perform matrix-vector multiplication
   190000:  335:    for (int i = 0; i < A->num_rows; i++)
        -:  336:    {
  7970000:  337:        for (int j = 0; j < 2 * A->num_non_zeros -
    num_diagonals; j++)
        -:  338:        {
  7960000:  339:            if (i < all_rows[j])
        -:  340:            {
   170000:  341:                break;
        -:  342:            }
  7790000:  343:            else if (i == all_rows[j])
        -:  344:            {
   820000:  345:                result[i] += x[all_cols[j]] *
     all_values[j];
        -:  346:            }
        -:  347:        }
        -:  348:    }
    10000:  349:}
    -:  350:
        1:  351:void Jacobi(const CSRMatrix *A, double *b, double *
          x, int num_iterations)
        -:  352:{
        -:  353:
        1:  354:    int num_diagonal_entries = NumDiagonals(A);
        1:  355:    if (num_diagonal_entries != A->num_cols)
        -:  356:    {
    #####:  357:        printf("Not all diagonal entries are non-
       zero. Jacobi approach will fail.\n");
    #####:  358:        for (int i = 0; i < A->num_cols; i++)
        -:  359:        { // Initial guess for x is all 1 for
           simple matrix multiplication.
    #####:  360:            x[i] = 1.0;
        -:  361:        }
    #####:  362:        spmv_csr(A, x, b);
    #####:  363:        return;
        -:  364:    }
        1:  365:    int AllRows[2 * A->num_non_zeros -
           num_diagonal_entries];
        1:  366:    int AllCols[2 * A->num_non_zeros -
           num_diagonal_entries];
        1:  367:    double AllVals[2 * A->num_non_zeros -
           num_diagonal_entries];
        1:  368:    Symmetrify(A, AllRows, AllCols, AllVals,
           num_diagonal_entries);
        1:  369:    double diagonal_entries[num_diagonal_entries];
           // A list of all diagonal entries, used for algebra.
        1:  370:    int index_of_diagonal_entry = 0;
           // Need this for program logic
        -:  371:
       83:  372:    for (int i = 0; i < 2 * A->num_non_zeros -
          num_diagonal_entries; i++)
        -:  373:    {
       82:  374:        if (AllRows[i] == AllCols[i])
        -:  375:        {
       18:  376:            diagonal_entries[
          index_of_diagonal_entry] = AllVals[i];
       18:  377:            index_of_diagonal_entry++;
        -:  378:        }
        -:  379:    }
        -:  380:
        -:  381:    /*
        -:  382:    for (int i = 0; i < 2 * num_diagonal_entries; i
           ++)
        -:  383:    {
        -:  384:        printf("%.16f\n", diagonal_entries[i]);
        -:  385:    }
        -:  386:    */
        -:  387:
        1:  388:    double result[A->num_cols]; //Result vector of A*x from Jacobi method
        1:  389:    double norm_diff = 1.0; //Initialize and norm_diff
        -:  390:
       19:  391:    for (int i = 0; i < A->num_cols; i++)
        -:  392:    { // Initial guess for x is all zeros.
       18:  393:        x[i] = 0.0;
        -:  394:    }
    10001:  395:    for (int i = 0; i < num_iterations; i++)
        -:  396:    {
    10000:  397:        if (norm_diff < 1e-7) //Condition to check if norm difference is sufficient to stop iterations
        -:  398:        {
    #####:  399:            break;
        -:  400:        }
    10000:  401:        MatrixMultiply(A, AllRows, AllCols, AllVals, x, result, num_diagonal_entries);
   190000:  402:        for (int i = 0; i < A->num_cols; i++)
        -:  403:        {
   180000:  404:            x[i] = (b[i] - result[i] + diagonal_entries[i] * x[i]) / diagonal_entries[i]; //Algebra, this is based off of the numerical process for the Jacobi method
        -:  405:        }
    10000:  406:        norm_diff = ComputeNorm(A, b, result);
        -:  407:        //printf("%f\n", norm_diff);
        -:  408:    }
        -:  409:
        -:  410:    // This prints out the components of the solution vector
        -:  411:    /*
        -:  412:    printf("Result:\n");
        -:  413:    for (int i = 0; i < A->num_cols; i++)
        -:  414:    {
        -:  415:        printf("%f ", x[i]);
        -:  416:    }
        -:  417:    printf("\n");
        -:  418:    */
        -:  419:
        1:  420:    printf("Residual Norm: %.6f\n", norm_diff);
        -:  421:
        -:  422:}
        -:  423:
    #####:  424:void ComputeResidual(const CSRMatrix *A, double *b, double *Ax)
    #####:  425:{
    #####:  426:    double residual[A->num_cols];
    #####:  427:    printf("Residual: [");
    #####:  428:    for (int i = 0; i < A->num_cols; i++)
        -:  429:    {
    #####:  430:        residual[i] = b[i] - Ax[i];
    #####:  431:        printf("%f, ", residual[i]);
        -:  432:    }
    #####:  433:    printf("]\n");
    #####:  434:}
        -:  435:
    10000:  436:double ComputeNorm(const CSRMatrix *A, double *b, double *Ax)
    10000:  437:{
    10000:  438:    double residual[A->num_cols];
    10000:  439:    double norm = 0.0;
   190000:  440:    for (int i = 0; i < A->num_cols; i++)
        -:  441:    {
   180000:  442:        residual[i] = b[i] - Ax[i];
   180000:  443:        norm += residual[i] * residual[i];
        -:  444:    }
    10000:  445:    norm = sqrt(norm);
    10000:  446:    return norm;
        -:  447:}
        -:  448:
    #####:  449:void save_sparsity_pattern(const CSRMatrix *A, const char *filename)
        -:  450:{
    #####:  451:    int width = A->num_cols * 10;  // Adjust dimensions for larger image
    #####:  452:    int height = A->num_rows * 10; // Adjust
       dimensions for larger image
    #####:  453:    png_bytep *row_pointers = (png_bytep *)malloc(
       height * sizeof(png_bytep));
    #####:  454:    if (row_pointers == NULL)
        -:  455:    {
    #####:  456:        fprintf(stderr, "Memory allocation failed\n
       ");
    #####:  457:        return;
        -:  458:    }
    #####:  459:    for (int i = 0; i < height; i++)
        -:  460:    {
    #####:  461:        row_pointers[i] = (png_byte *)malloc(width
       * sizeof(png_byte));
    #####:  462:        if (row_pointers[i] == NULL)
        -:  463:        {
    #####:  464:            fprintf(stderr, "Memory allocation
       failed\n");
    #####:  465:            return;
        -:  466:        }
        -:  467:    }
    #####:  468:    for (int i = 0; i < height; i++)
        -:  469:    {
    #####:  470:        for (int j = 0; j < width; j++)
        -:  471:        {
    #####:  472:            row_pointers[i][j] = 255; // Initialize
        to white (255 = white in grayscale)
        -:  473:            // Set a larger block for non-zero
           elements
    #####:  474:            if (i % 10 == 0 && j % 10 == 0)
        -:  475:            {
    #####:  476:                for (int k = 0; k < 10; k++)
        -:  477:                {
    #####:  478:                    for (int l = 0; l < 10; l++)
        -:  479:                    {
    #####:  480:                        if ((i + k) < height && (j
       + l) < width)
        -:  481:                        {
    #####:  482:                            row_pointers[i + k][j +
        1] = 255; // Adjust the block color if needed
        -:  483:                        }
        -:  484:                    }
        -:  485:                }
        -:  486:            }
        -:  487:            // Check if the element at (i, j) is
           non-zero
    #####:  488:            for (int k = A->row_ptr[i / 10]; k < A
       ->row_ptr[(i / 10) + 1]; k++)
        -:  489:            {
    #####:  490:                if (A->col_ind[k] == (j / 10))
        -:  491:                {
    #####:  492:                    row_pointers[i][j] = 0; // Set
       to black
    #####:  493:                    break;
        -:  494:                }
        -:  495:            }
        -:  496:        }
        -:  497:    }
    #####:  498:    FILE *fp = fopen(filename, "wb");
    #####:  499:    if (!fp)
        -:  500:    {
    #####:  501:        fprintf(stderr, "Error opening file for
       writing\n");
    #####:  502:        return;
        -:  503:    }
    #####:  504:    png_structp png_ptr = png_create_write_struct(
       PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    #####:  505:    if (!png_ptr)
        -:  506:    {
    #####:  507:        fprintf(stderr, "Error creating PNG write
       struct\n");
    #####:  508:        fclose(fp);
    #####:  509:        return;
        -:  510:    }
    #####:  511:    png_infop info_ptr = png_create_info_struct(
       png_ptr);
    #####:  512:    if (!info_ptr)
        -:  513:    {
```

```
  #####:  514:        fprintf(stderr, "Error creating PNG info
    struct\n");
  #####:  515:        png_destroy_write_struct(&png_ptr, NULL);
  #####:  516:        fclose(fp);
  #####:  517:        return;
      -:  518:    }
  #####:  519:    png_set_IHDR(png_ptr, info_ptr, width, height,
    8, PNG_COLOR_TYPE_GRAY, PNG_INTERLACE_NONE,
    PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);
  #####:  520:    png_set_rows(png_ptr, info_ptr, row_pointers);
  #####:  521:    png_set_filter(png_ptr, 0, PNG_FILTER_NONE);
  #####:  522:    png_init_io(png_ptr, fp);
  #####:  523:    png_write_png(png_ptr, info_ptr,
    PNG_TRANSFORM_IDENTITY, NULL);
  #####:  524:    fclose(fp);
  #####:  525:    png_destroy_write_struct(&png_ptr, &info_ptr);
  #####:  526:    for (int i = 0; i < height; i++)
      -:  527:    {
  #####:  528:        free(row_pointers[i]);
      -:  529:    }
  #####:  530:    free(row_pointers);
      -:  531:}
```