

**ChatGPT Citation:****Prompt:**

Here is the writeup: Your writeup should be written in LaTeX and included in a "homework-3/docs" directory (please include both the .tex and the .pdf files). Use the `\verb` command or `\verbatim` environment to format all code, including compilation commands.

Your writeup should include a discussion on the Compressed Sparse Row sparse matrix format. This discussion should include sections on the following:

Briefly explain how the `SparseMatrixCSR` structure represents a sparse matrix. Would you expect CSR or CSC be more efficient for a matrix-vector multiplication? Why?

Derive what the `SparseMatrixCSR` storage arrays should be (i.e., the value, column index, and row index arrays) for the following matrix:

In addition to the discussion, please include the following sections in your writeup:

Compilation: state the exact command or series of commands needed to compile your project. If you have any additional notes for your graders (i.e., that the code you submitted does not compile properly), please include them here.

Implementation: explain which Vector arithmetic operations you needed to implement to get the iteration to work.

Verification and Timing:

for each matrix type, report the number of iterations it took to reduce  $\text{norm}(r)$  to below  $1e-3$ , as well as the final value of  $\text{norm}(r)$ .

report the time spent computing the iteration for dense and sparse matrices.

**Output:**

```
% homework-3/docs/writeup.tex
```

```
\documentclass{article}
```

```
\usepackage{listings} % for code listings
```

```
\usepackage{graphicx} % for including images
```

```
\begin{document}
```

```
\title{Homework 3 Writeup}
```

```
\author{Your Name}
```

`\date{\today}`

`\maketitle`

`\section{Discussion on Compressed Sparse Row (CSR) Format}`

`\subsection{Representation of SparseMatrixCSR}`

The SparseMatrixCSR structure represents a sparse matrix using three arrays:

`\begin{itemize}`

`\item \textbf{Value Array (\_nzval)}:` Contains the nonzero values of the matrix.

`\item \textbf{Column Index Array (\_col\_index)}:` Contains the column indices corresponding to the nonzero values.

`\item \textbf{Row Index Array (\_row\_index)}:` Contains the starting index of each row in the value and column index arrays.

`\end{itemize}`

`\subsection{Efficiency of CSR for Matrix-Vector Multiplication}`

CSR is typically more efficient for matrix-vector multiplication compared to CSC. This is because matrix-vector multiplication involves iterating over the rows of the matrix, and CSR format allows for efficient access to the nonzero elements of each row.

`\subsection{Storage Arrays for Given Matrix}`

To derive the storage arrays for the given matrix:

`\[`  
 $A = \frac{1}{h^2}$  `\begin{bmatrix}`  
1 & -1 & 0 & \cdots & 0 \\  
-1 & 2 & -1 & \cdots & 0 \\  
\vdots & \vdots & \vdots & \ddots & \vdots \\  
0 & \cdots & 0 & -1 & 1  
`\end{bmatrix}`  
`\]`

The value array (`\_nzval`) would contain  $\frac{1}{h^2}$  for all diagonal entries (2) and  $-\frac{1}{h^2}$  for off-diagonal entries (-1). The column index array (`\_col\_index`)

would contain the corresponding column indices, and the row index array (`\_row\_index`) would contain the starting index of each row.

## `\section{Compilation}`

To compile the project, use the following commands:

```
\begin{verbatim}
$ cd /path/to/project
$ make
\end{verbatim}
```

Note: Additional notes for graders regarding compilation issues can be mentioned here.

## `\section{Implementation}`

To implement the iteration, the following Vector arithmetic operations were used:

```
\begin{itemize}
  \item Vector addition
  \item Scalar-vector multiplication
  \item Norm calculation
\end{itemize}
```

## `\section{Verification and Timing}`

### `\subsection{Dense Matrix}`

For the dense matrix representation:

```
\begin{itemize}
  \item Iterations: X
  \item Final norm(r): Y
  \item Time per iteration: Z microseconds
\end{itemize}
```

### `\subsection{Sparse Matrix}`

For the sparse matrix representation:

```

\begin{itemize}
  \item Iterations: X
  \item Final norm(r): Y
  \item Time per iteration: Z microseconds
\end{itemize}

\end{document}

```

**Prompt:**

Let  $n$  be some user-defined integer, and  $h = 1/n$ . Define the  $(n+1) \times (n+1)$  matrix  $A$   
 $A = (1/(h^2)) * [1, -1, \dots, -1, 2, -1, \dots, \dots, -1, 2, -1, \dots, \dots, -1, 1]$

If you have taken CMOR 304 or a similar class on numerical methods for partial differential equations (PDEs), you may recognize this as the second order finite difference discretization matrix for the Poisson equation under Neumann boundary conditions.

Let  $a = .5^2$ , and initialize  $u$  as a vector of all zeros. Let  $b$  be a vector whose  $i$ th entry is  $\cos(\pi * i * h)$  for  $i = 0, 1, 2, \dots, n$ . Write code to execute the iteration  $u += a * r$ , where  $r = b - A * u$ , until  $\text{norm}(r) < 1e-3$ . Report the number of iterations required, as well as the final value of  $\text{norm}(r)$  for  $n = 100$ . The iteration should converge in a few seconds for a dense matrix (and run slightly faster for a sparse matrix).

Use `#include<chrono>` to time your iteration using both dense and sparse matrix representations of  $A$ . Compute the time per iteration for both dense and sparse matrices. You do not need to average over multiple runs.

**Output:**

```

#include <iostream>
#include <cmath>
#include <Eigen/Dense>
#include <Eigen/Sparse>
#include <chrono>

```

```

using namespace Eigen;
using namespace std;

```

```

int main() {
  // Define parameters
  int n = 100;
  double h = 1.0 / n;
  double a = 0.5 * 0.5;

  // Initialize matrix A (sparse representation)

```

```

SparseMatrix<double> A(n + 1, n + 1);
vector<Triplet<double>> tripletList;

for (int i = 0; i <= n; ++i) {
    tripletList.push_back(Triplet<double>(i, i, 2.0));
    if (i > 0) {
        tripletList.push_back(Triplet<double>(i - 1, i, -1.0));
        tripletList.push_back(Triplet<double>(i, i - 1, -1.0));
    }
}

A.setFromTriplets(tripletList.begin(), tripletList.end());
A *= (1.0 / (h * h));

// Initialize vectors u and b
VectorXd u = VectorXd::Zero(n + 1);
VectorXd b = VectorXd::Zero(n + 1);

for (int i = 0; i <= n; ++i) {
    b(i) = cos(M_PI * i * h);
}

// Dense matrix representation of A for comparison
MatrixXd Adense = A;

// Perform iteration using sparse matrix
auto start_sparse = chrono::high_resolution_clock::now();

int iterations_sparse = 0;
double norm_r_sparse;

do {
    VectorXd r_sparse = b - A * u;
    u += a * r_sparse;
    norm_r_sparse = r_sparse.norm();
    iterations_sparse++;
} while (norm_r_sparse >= 1e-3);

auto end_sparse = chrono::high_resolution_clock::now();
auto duration_sparse = chrono::duration_cast<chrono::microseconds>(end_sparse -
start_sparse);

cout << "Sparse Matrix Iteration Results:" << endl;
cout << "Iterations: " << iterations_sparse << endl;

```

```

    cout << "Final norm(r): " << norm_r_sparse << endl;
    cout << "Time per iteration (sparse): " << duration_sparse.count() / iterations_sparse << "
microseconds" << endl;

    // Perform iteration using dense matrix for comparison
    auto start_dense = chrono::high_resolution_clock::now();

    int iterations_dense = 0;
    double norm_r_dense;

    do {
        VectorXd r_dense = b - Adense * u;
        u += a * r_dense;
        norm_r_dense = r_dense.norm();
        iterations_dense++;
    } while (norm_r_dense >= 1e-3);

    auto end_dense = chrono::high_resolution_clock::now();
    auto duration_dense = chrono::duration_cast<chrono::microseconds>(end_dense -
start_dense);

    cout << "\nDense Matrix Iteration Results:" << endl;
    cout << "Iterations: " << iterations_dense << endl;
    cout << "Final norm(r): " << norm_r_dense << endl;
    cout << "Time per iteration (dense): " << duration_dense.count() / iterations_dense << "
microseconds" << endl;

    return 0;
}

```

### **Prompt:**

Programming portion of assignment

### **Output:**

Abstract Matrix.hpp:

```

#ifndef ABSTRACT_MATRIX_HPP
#define ABSTRACT_MATRIX_HPP

```

```

class AbstractMatrix {
public:
    virtual ~AbstractMatrix() {}

```

```
    virtual double operator()(int row, int column) const = 0;
    // Other virtual functions if needed
};
```

```
#endif // ABSTRACT_MATRIX_HPP
```

Vector.hpp:

```
#ifndef VECTOR_HPP
#define VECTOR_HPP
```

```
class Vector {
    // Define your Vector class if not provided in the provided code
};
```

```
#endif // VECTOR_HPP
```

Denserowmatrix.hpp;

```
#ifndef DENSE_ROW_MATRIX_HPP
#define DENSE_ROW_MATRIX_HPP
```

```
#include "AbstractMatrix.hpp"
```

```
class DenseRowMatrix : public AbstractMatrix {
private:
    double* _data;
    int _rows, _columns;

public:
    DenseRowMatrix(int rows, int columns);
    ~DenseRowMatrix();

    double& operator()(int row, int column) const override;
    // Other member functions if needed
};
```

```
#endif // DENSE_ROW_MATRIX_HPP
```

Sparsematrix.csr.hpp:

```
#ifndef SPARSE_MATRIX_CSR_HPP
#define SPARSE_MATRIX_CSR_HPP
```

```

#include "AbstractMatrix.hpp"

class SparseMatrixCSR : public AbstractMatrix {
private:
    double* _nzval;
    int* _col_index;
    int* _row_index;
    int _rows, _columns, _nonzeros;

public:
    SparseMatrixCSR(int rows, int columns, int nonzeros, int* row_indices, int* col_indices,
double* values);
    ~SparseMatrixCSR();

    double operator()(int row, int column) const override;
    double* get_nzval() const;
    int* get_col_index() const;
    int* get_row_index() const;
    // Other member functions if needed
};

#endif // SPARSE_MATRIX_CSR_HPP

```

Denserowmatrix.cpp:

```

#include "DenseRowMatrix.hpp"

```

```

DenseRowMatrix::DenseRowMatrix(int rows, int columns) : _rows(rows), _columns(columns) {
    _data = new double[rows * columns](); // Initialize to zero
}

```

```

DenseRowMatrix::~DenseRowMatrix() {
    delete[] _data;
}

```

```

double& DenseRowMatrix::operator()(int row, int column) const {
    return _data[row * _columns + column];
}
// Implement other member functions if needed

```

Sparsematrix.csr.cpp:

```
#include "SparseMatrixCSR.hpp"
```

```
SparseMatrixCSR::SparseMatrixCSR(int rows, int columns, int nonzeros, int* row_indices, int* col_indices, double* values)
```

```
    : _rows(rows), _columns(columns), _nonzeros(nonzeros) {  
        // Implement the conversion to CSR format  
    }
```

```
SparseMatrixCSR::~SparseMatrixCSR() {  
    // Implement the destructor to free allocated memory  
}
```

```
double SparseMatrixCSR::operator()(int row, int column) const {  
    // Implement accessing the value in CSR format  
}
```

```
double* SparseMatrixCSR::get_nzval() const {  
    return _nzval;  
}
```

```
int* SparseMatrixCSR::get_col_index() const {  
    return _col_index;  
}
```

```
int* SparseMatrixCSR::get_row_index() const {  
    return _row_index;  
}
```

```
// Implement other member functions if needed
```