Assignment 3: Programming in C++

Chuyang Wang

November 2023

1 Codes

1.1 Main

```
#include <iostream>
#include <vector>
#include <cmath>
#include <chrono>
#include "dense_row_matrix.hpp"
#include "sparse_matrix_csr.hpp"
#include "vector.hpp"

// Define the norm function to calculate the Euclidean norm of a Vector
double norm(const Vector& v) {
    double sum = 0.0;
    for (int i = 0; i < v.length(); ++i) {
        sum += v(i) * v(i);
    }
    return std::sqrt(sum);
}

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

    // Initialize vector b
    Vector b(n + 1);
    for (int i = 0; i <= n; ++i) {
        b(i) = std::cos(M_PI * i * h);
    }

    // Initialize vector u
    Vector u(n + 1); // Assuming the Vector class has a constructor that takes size
    for (int i = 0; i <= n; ++i) {
```

```
    u(i) = 0.0; // Initialize with zeros
}

// Initialize A_dense
DenseRowMatrix A_dense(n + 1, n + 1);
double scale = 1.0 / (h * h);
for (int i = 0; i < n + 1; ++i) {
    if (i > 0) {
        A_dense(i, i - 1) = -scale;
    }
    A_dense(i, i) = 2.0 * scale;
    if (i < n) {
        A_dense(i, i + 1) = -scale;
    }
}

A_dense(0,0)=1/h/h;
A_dense(n,n)=1/h/h;

// Initialize A_sparse using the provided code
int nnz = 3 * (n - 1) + 4;
int *row_indices = new int[n + 2];
int *col_indices = new int[nnz];
double *values = new double[nnz];

//initialize values
for (int i = 1; i < n; ++i) {
    values[3 * i] = 2 / h / h;
    values[3 * i - 1] = -1 / h / h;
    values[3 * i + 1] = -1 / h / h;
}
values[0] = values[3 * n] = 1 / h / h;
values[1] = values[3 * n - 1] = -1 / h / h;

//initialize col_indicies
for (int i = 0; i < n; i++) {
    col_indices[3 * i + 1] = i + 1;
}

for (int i = 0; i <= n; i++) {
    col_indices[3 * i] = i;
}
for (int i = 1; i <= n; i++) {
    col_indices[3 * i - 1] = i - 1;
}
col_indices[0] = 0;
```

```
//initialize row_indices
for (int i = 1; i <= n; i++) {
    row_indices[i] = 3 * i - 1;
}
row_indices[0] = 0;
row_indices[n + 1] = 3 * n + 1;

// Create the SparseMatrixCSR object
SparseMatrixCSR A_sparse(n + 1, n + 1, nnz, values, col_indices, row_indices);

// Iteration for Dense Matrix
int iteration_dense = 0;
double norm_r_dense = 0.0;
auto start_dense = std::chrono::high_resolution_clock::now();
Vector r_dense = b - A_dense * u;
do {
    r_dense = b - A_dense * u;
    norm_r_dense = norm(r_dense);
    u = u + a * r_dense;
    iteration_dense++;
} while (norm_r_dense > 1e-3);

auto end_dense = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed_dense = end_dense - start_dense;
std::cout << "Dense Matrix: Iterations = " << iteration_dense << std::endl;
std::cout << "Dense Matrix: Time per iteration = "
          << (elapsed_dense.count() / iteration_dense) << " seconds" << std::endl;
std::cout << "Final value of norm(r) for Dense Matrix: " << norm_r_dense << std::endl;

// Reset vector u for Sparse Matrix
for (int i = 0; i <= n; ++i) {
    u(i) = 0.0; // Reinitialize with zeros
}

// Iteration for Sparse Matrix
int iteration_sparse = 0;
double norm_r_sparse = 0.0;
auto start_sparse = std::chrono::high_resolution_clock::now();
Vector r_sparse = b -  operator*(static_cast<const SparseMatrixCSR &>(A_sparse), u);
do {
     r_sparse = b -  operator*(static_cast<const SparseMatrixCSR &>(A_sparse), u);
    norm_r_sparse = norm(r_sparse);
    u = u + a * r_sparse;
    iteration_sparse++;
} while (norm_r_sparse > 1e-3 );
```

```
    auto end_sparse = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed_sparse = end_sparse - start_sparse;
    std::cout << "Sparse Matrix: Iterations = " << iteration_sparse << std::endl;
    std::cout << "Sparse Matrix: Time per iteration = "
              << (elapsed_sparse.count() / iteration_sparse) << " seconds" << std::endl;
    std::cout << "Final value of norm(r) for Sparse Matrix: " << norm_r_sparse << std::endl;

return 0;
}
```

1.2 Makefile

```
CC = g++
CPPFLAGS = -I./include
SRC_DIR = ./src
OBJ_DIR = ./obj
BIN_DIR = ./bin

INC_FILES := $(wildcard $(INC_DIR)/*.hpp)
SRC_FILES := $(wildcard $(SRC_DIR)/*.cpp)
OBJ_FILES := $(patsubst $(SRC_DIR)/%.cpp, $(OBJ_DIR)/%.o, $(SRC_FILES))

.PHONY: all clean main

all: main

main: $(OBJ_FILES) main.cpp
mkdir -p $(BIN_DIR)
$(CC) $^ $(CPPFLAGS) -o $(BIN_DIR)/main

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.cpp $(INC_FILES)
mkdir -p $(OBJ_DIR)
$(CC) -c $< -o $@ $(CPPFLAGS)

clean:
rm -f $(BIN_DIR)/main $(OBJ_DIR)/*.o *~
```

1.3 Abstract matrix

1.3.1 abstract_matrix.hpp

```
#include "vector.hpp"
```

```
#ifndef _ABSTRACTMATRIX
#define _ABSTRACTMATRIX

class AbstractMatrix{
public:
  AbstractMatrix(int m, int n){
    _rows = m;
    _columns = n;
  }
  int num_rows()const{ return _rows; };
  int num_columns()const{ return _columns; };

  virtual double operator()(int row, int column) const{
    throw std::logic_error("`double operator()` not implemented"
   "for AbstractMatrix");
  }

  virtual double & operator()(int row, int column){
    throw std::logic_error("`double & operator()` not implemented"
   "for AbstractMatrix");
  }

private:
  int _rows;
  int _columns;
};

#endif

Vector operator*(AbstractMatrix & A, Vector & x);
```

1.3.2 abstract_matrix.cpp

```
#include "abstract_matrix.hpp"

Vector operator*(AbstractMatrix & A, Vector & x){
  Vector out(x.length());
  for (int i = 0; i < A.num_rows(); ++i){
    out(i) = 0.0;
    for (int j = 0; j < A.num_rows(); ++j){
      out(i) += A(i,j) * x(j);
```

```
    }
  }
  return out;
}
```

1.4 Vector

1.4.1 vector.hpp

```
#include<iostream>

#ifndef _VECTOR
#define _VECTOR

class Vector{

public:
  // constructor
  Vector(int length){
    _length = length;
    _data = new double [length];
  }

  // destructor
  ~Vector(){ delete [] _data; }

  // copy constructor
  Vector(const Vector & copy_from);

  // copy move constructor
  Vector & operator=(const Vector & copy_from);

  int length() const{ return _length; };

  void print(std::string variable_name);

  double & operator()(const int index) const {
    return _data[index];
  }

  Vector & operator+=(Vector x);

private:

  int _length;
```

```
  double * _data;

};

#endif

Vector operator+(const Vector & x, const Vector & y);
Vector operator-(const Vector & x, const Vector & y);
Vector operator*(double x, const Vector & y);
Vector operator*(const Vector & x, double y);
```

1.4.2 vector.cpp

```
#include "vector.hpp"
#include <cmath>
Vector & Vector::operator+=(Vector x){
  for (int i = 0; i < x.length(); ++i){
    _data[i] += x(i);
  }
  return *this; // pointer to the instance
}

Vector & Vector::operator=(const Vector & copy_from){
  if (_length != copy_from.length()){
    std::cout << "ERROR: vectors are not the same length";
    std::cout << std::endl;
  }
  for (int i = 0; i < _length; ++i){
    _data[i] = copy_from(i);
  }

  return *this;
}

Vector::Vector(const Vector & copy_from){
  _length = copy_from.length();
  _data = new double [_length];
  for (int i = 0; i < _length; ++i){
    _data[i] = copy_from(i);
  }
}

void Vector::print(std::string variable_name){
  std::cout << variable_name << " = " << std::endl;
  for (int i = 0; i < _length; ++i){
```

```
    std::cout << _data[i] << std::endl;
  }
}

Vector operator+(const Vector & x, const Vector & y){
  Vector out(x.length());
  for (int i = 0; i < out.length(); ++i){
    out(i) = x(i) + y(i);
  }
  return out;
}

Vector operator-(const Vector & x, const Vector & y){
  return (x + -1.0 * y);
}

Vector operator*(double x, const Vector & y){
  Vector out(y.length());
  for (int i = 0; i < out.length(); ++i){
    out(i) = x * y(i);
  }
  return out;
}
```

1.5 Dense Row Matrix

1.5.1 dense_row_matrix.hpp

```
#include "abstract_matrix.hpp"

class DenseRowMatrix : public AbstractMatrix {
public:
    DenseRowMatrix(int m, int n);
    ~DenseRowMatrix();

    double operator()(int row, int column) const override;
    double &operator()(int row, int column) override;

private:
    double *_data;
};
```

1.5.2 dense_row_matrix.cpp

```
#include "dense_row_matrix.hpp"
```

```
#include <stdexcept>

DenseRowMatrix::DenseRowMatrix(int m, int n) : AbstractMatrix(m, n) {
    _data = new double[m * n](); // Initialize with zeros
}

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

double DenseRowMatrix::operator()(int row, int column) const {
    if (row >= num_rows() || column >= num_columns()) {
        throw std::out_of_range("Index out of bounds");
    }
    return _data[row * num_columns() + column];
}

double &DenseRowMatrix::operator()(int row, int column) {
    if (row >= num_rows() || column >= num_columns()) {
        throw std::out_of_range("Index out of bounds");
    }
    return _data[row * num_columns() + column];
}
```

1.6 Sparse Matrix CSR

1.6.1 sparse_matrix_csr.hpp

```
#ifndef SPARSE_MATRIX_CSR_HPP
#define SPARSE_MATRIX_CSR_HPP

#include "abstract_matrix.hpp"

class SparseMatrixCSR : public AbstractMatrix {
public:
    SparseMatrixCSR(int m, int n, int nnz, const double *values, const int *col_indices, co

    ~SparseMatrixCSR();

    const double *get_nzval() const { return _nzval; }
    const int *get_col_index() const { return _col_index; }
    const int *get_row_index() const { return _row_index; }

    double operator()(int row, int column) const override;
```

```
private:
    double *_nzval;
    int *_col_index;
    int *_row_index;
};

Vector operator*(const SparseMatrixCSR &A, const Vector &x);

#endif // SPARSE_MATRIX_CSR_HPP
```

1.6.2 sparse_matrix_csr.cpp

```
#include "sparse_matrix_csr.hpp"
SparseMatrixCSR::SparseMatrixCSR(int m, int n, int nnz, const double *values, const int *col
    : AbstractMatrix(m, n), _nzval(nullptr), _col_index(nullptr), _row_index(nullptr) {
    _nzval = new double[nnz];
    _row_index = new int[m + 1];
    _col_index = new int[nnz];

    for (int i = 0; i < nnz; ++i) {
        _nzval[i] = values[i];
        _col_index[i] = col_indices[i];
    }

    for (int i = 0; i <= m; ++i) {
        _row_index[i] = row_indices[i];
    }
}

SparseMatrixCSR::~SparseMatrixCSR()
{
    delete[] _nzval;
    delete[] _col_index;
    delete[] _row_index;
}


double SparseMatrixCSR::operator()(int row, int column) const {
    int row_start = _row_index[row];
    int row_end = _row_index[row + 1];
    for (int i = row_start; i < row_end; ++i) {
        if (_col_index[i] == column) {
            return _nzval[i];
        }
    }
```

```
    return 0.0; // Default value for missing entries
}


Vector operator*(const SparseMatrixCSR& A, const Vector& x) {
    const double* nzval = A.get_nzval();
    const int* row_index = A.get_row_index();
    const int* col_index = A.get_col_index();

    Vector result(A.num_rows());
    for (int i = 0; i < A.num_rows(); ++i) {
        int row_start = row_index[i];
        int row_end = row_index[i + 1];
        double sum = 0.0;
        for (int j = row_start; j < row_end; ++j) {
            int col = col_index[j];
            sum += nzval[j] * x(col);
        }
        result(i) = sum;
    }
    return result;
}
```

2 Commands and outputs

```
make

mkdir -p ./bin
g++ obj/abstrct_matrix.o obj/dense_row_matrix.o obj/sparse_matrix_csr.o obj/vector.o main.cp

bin/main

Dense Matrix: Iterations = 18342
Dense Matrix: Time per iteration = 0.000142293 seconds
Final value of norm(r) for Dense Matrix: 0.000999652
Sparse Matrix: Iterations = 18342
Sparse Matrix: Time per iteration = 6.01251e-06 seconds
Final value of norm(r) for Sparse Matrix: 0.000999652
```

3 discussion on the Compressed Sparse Row sparse matrix format

3.1 Briefly explain how the SparseMatrixCSR structure represents a sparse matrix.

The `SparseMatrixCSR` structure is an efficient representation of a sparse matrix, where most of the elements are zero. It utilizes the Compressed Sparse Row (CSR) format for storing the matrix.

The CSR format is characterized by three main components:

- **Values Array**: A one-dimensional array that stores all the non-zero values of the matrix sequentially row by row.
- **Column Indices Array**: This array stores the column indices corresponding to each non-zero element in the values array.
- **Row Start Indices Array**: It indicates the starting index in the values array for each row of the matrix. The size of this array is one more than the number of rows, with the last element representing the total count of non-zero elements.

The CSR format is highly efficient for matrices with a large number of zero elements. It significantly reduces the memory requirement by storing only non-zero elements and their respective positions. Moreover, operations like matrix-vector multiplication can be performed more efficiently using this format.

In summary, the `SparseMatrixCSR` structure, leveraging the CSR format, provides an effective solution for handling and processing large sparse matrices, especially in applications where memory efficiency and fast computational performance are crucial.

3.2 Would you expect CSR or CSC be more efficient for a matrix-vector multiplication? Why?

For matrix-vector multiplication, the efficiency of using CSR (Compressed Sparse Row) versus CSC (Compressed Sparse Column) format depends on the specific operation:

- In the case of multiplying a sparse matrix by a dense vector ($Ax = b$), **CSR format is generally more efficient**. This efficiency arises because CSR aligns with the row-wise access pattern of the operation, facilitating sequential and cache-friendly memory access.
- Conversely, for operations like multiplying a dense vector by a sparse matrix ($xA = b$), the **CSC format is more efficient** as it aligns with the column-wise access pattern required in such operations.

Overall, the choice between CSR and CSC formats for matrix-vector multiplication largely hinges on the nature of the multiplication (row-wise vs. column-wise) and the sparsity structure of the matrix involved.

3.3 Derive what the SparseMatrixCSR storage arrays should be

Given a matrix, its Compressed Sparse Row (CSR) representation consists of three arrays: *values*, *column indices*, and *row start indices*.

Values Array

This array contains all the non-zero values of the matrix listed in a row-major order:

$$\text{values} = [8, 3, 7, 1, 5, 8, 6, 1, 9]$$

Column Indices Array

This array contains the column indices corresponding to each non-zero value in the matrix:

$$\text{col\_indices} = [0, 1, 0, 2, 4, 0, 2, 0, 1]$$

Row Start Indices Array

This array contains the indices where each row starts in the *values* array, with an extra element at the end indicating the total count of non-zero elements:

$$\text{row\_start\_indices} = [0, 2, 5, 7, 9]$$

4 For graduate students