Homework 01

Implementation of a sparse matrix class

Advanced Programming - SISSA, UniTS, 2024-2025

Pasquale Claudio Africa, Giuseppe Alessio D'Inverno

Due date: 31 Oct 2024

What is a sparse matrix?

Quoting from Wikipedia :

> A **sparse matrix** is a matrix in which most of the elements are zero. There is no strict definition regarding the proportion of zero-value elements for a matrix to qualify as sparse, but a common criterion is that the number of non-zero elements is roughly equal to the number of rows or columns.

The goal of this assignment is to create a C++ code implementing the concept of a sparse matrix while efficiently storing **only** the non-zero elements.

Storage schemes

There are several efficient ways to store a sparse matrix. In this homework, you are asked to implement and validate two popular storage schemes, namely the **Coordinate (COO)** and **Compressed Sparse Row (CSR)** formats.

Consider the following matrix as an example:

$$A = \begin{bmatrix} 0 & 0 & 3.1 & 0 & 4 \\ 0 & 0 & 5 & 0 & 7.4 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 6 & 0 \end{bmatrix}.$$

Coordinate (COO) format

A can be stored using three arrays of length *nnz* (number of non-zeros):

- An array `values` containing all the nonzero values.
- An array `rows` of integers containing their corresponding row indices.
- An array `cols` of integers containing their corresponding column indices.

For the example at hand, reading the matrix row-by-row, we have:

- `values  = [3.1, 4, 5, 7.4, 2, 6]`
- `rows    = [  0, 0, 1,   1, 3, 3]`
- `columns = [  2, 4, 2,   4, 1, 3]`

This expresses that entry 3.1 is stored in row 0, column 2, entry 5 is stored in row 1, column 2, and so on.

Compressed Sparse Row (CSR) format

This format employs again three arrays to represent A:

- An array `values` containing all the nonzero values (length *nnz*),
- An array `columns` of integers containing their column indices (length *nnz*),
- An array `row_idx` of integers with the cumulative number of nonzero entries up to the i-th row (excluded). The length of such array is $m + 1$, being m the number of rows of A. By convention, we assume that `row_idx[0] = 0`.

In such a way, the quantity `row_idx[i+1] - row_idx[i]` represents the number of nonzero elements in the i-th row.

For the example at hand, we get:

- `values  = [3.1, 4, 5, 7.4, 2, 6]`
- `columns = [2, 4, 2, 4, 1, 3]`
- `row_idx = [0, 2, 4, 4, 6]`

Code organization

- Separate class/function declarations and definitions in different files.
- Provide a `main.cpp` file to test and demonstrate the correctness of the proposed implementation.
- Document your code with comments to explain your design choices.

Compilation

Compile your code using the following compilation flags:

```
g++ -std=c++17 -Wall -Wpendantic main.cpp [other_files.cpp] -o sparse_matrix
```

Provide clear instructions on how to compile and run your code or, preferably, a working **compilation script**.

Interface

You are required to implement an *abstract* base class `SparseMatrix` that provides a public interface to perform the following operations:

1. Get the number of rows and the number of columns.
2. Get the number of nonzeros.
3. Read an entry of the matrix (e.g., `const double x = A(2, 3);`). If indices are out of bound, then throw an error.
4. Write an entry of the matrix (e.g., `A(2, 3) = 5.7;`). If indices are out of bound, then throw an error. If indices are compatible with the matrix size but the entry has not been allocated yet, either print an error or (**bonus**) allocate it.
5. Given a vector $\vec{x}$ of compatible size, compute the matrix-vector product $\vec{y} = A\vec{x}$.
6. Print the matrix to the standard output, in a convenient, readable format.
7. Implement other utilities you think are useful (if any).

Rules

- Use `std::vector<type>` or raw arrays to store data.
- Implement access and write matrix entries by overloading `operator()`.
- Implement the matrix-vector product by overloading `operator*`.
- You can use external resources for reference and learning, but acknowledge them in your code.
- Use meaningful names that reflect the behavior of each function or variable.
- Ensure proper `const` and non-`const` overloads in member functions for const-correctness.
- ⚠️ **Penalties will be assigned if your interface does not properly satisfy const-correctness.**
- **Tip**: start from simple cases, generalize later.

Implementation and validation

1. Derive from the base class `SparseMatrix` to implement classes for both COO and CSR storage schemes (e.g., `SparseMatrixCOO` and `SparseMatrixCSR`).
2. Implement the operations defined above for both storage schemes.
3. Provide utility functions to convert a matrix from COO format to CSR and vice versa.
4. **Bonus**: templatize your classes on the type of number stored by the matrix (e.g., `int` or `double`).

The `main.cpp` file should include tests to validate the correctness of your program. Here are some test ideas:

- If $v = \vec{1}$, then $\left(M\vec{v}\right)_i = \sum_{j=1}^{n_\text{cols}} M_{ij} v_j = \sum_{j=1}^{n_\text{cols}} M_{ij}$, i.e. the sum of i-th row.
- If $\vec{v} = \vec{e}_i$ (the i-th vector of the canonical basis), then $M\vec{e}_i$ returns the i-th column of M.
- Implement additional tests of your choice.