Lab 04

Profiling, debugging and class templates

Paolo Joseph Baioni

October 31, 2025

Recalls

Introduction to exercise 1

- The implemented matrix class is organized as **column-major**, *i.e.* $A(i,j) =$ `data[i + j * rows()]`, conversion from 1d to 2d indexing is performed by the utility method `sub2ind`.
- Access to elements is implemented both in `const` and non-`const` versions, by overloading `operator()`.
- Data is private, *getter methods* expose what is needed to the user, both `const` and non-`const` versions are provided.
- Naive implementation of matrix-matrix multiplication is slow because it has low *data locality*, simply transposing the left matrix factor improves performance significantly[1].
- The `#include <ctime>` header provides timing utilities, `tic()` and `toc(x)` macros start and stop the timer.

[1] See *M. Kowarschik, C. Weiß. (2002). Lecture Notes in Computer Science. 213-232. DOI: 10.1007/3-540-36574-5_10* for further details.

Follow-up of Lab 3

Exercise 1.3

- Include the `Eigen/Dense` header.
- Use the `Eigen::Map` template class to wrap the matrix data and interpret it as `Eigen::MatrixXd`.
- Compare the execution speed with respect to the previous implementations for varying matrix size, such as `msize=500` and `msize=5000`.

Profiling and memory cheking

Exercise 1.4

Going back to Ex. 1.2, perform the subsequent analysis:

- coverage (`lcov`)
- memcheck (`valgrind`)
- profile (`valgrind, kcachegrind`)

Debugging
Exercise 2

The program `integer-list` in the directory `02-bug` has:

- a compile error;
- a run-time error;
- a memory leak;
- a potential memory leak that is not captured by the `main`.

Find all the issues and fix them.

Get help by using `gdb` and `valgrind`.

Implement a `C++` template class to evaluate derivatives of any order of a given function (callable object) at a given point using recursive backward/forward first-order finite difference formulas.

- A $N \times N$ sparse matrix is a matrix whose number of non-zero elements N_{nz} is $O(N)$.
- The average number m of non-zero elements per row (or column) is constant w.r.to the matrix size.
- If the majority of matrix entries is 0, *i.e.* if $m \ll N$ it is convenient to store only the non-zero elements.
- The matrix-vector product (which is the basic ingredient of Krylov solvers) costs $O(N)$ rather than $O(N^2)$.

Class templates

Sparse matrix formats

Some (slightly revisited) classical data structures for sparse matrices

$$A = \begin{bmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ 0 & 0 & -1 & 4 \end{bmatrix}$$

COO (coordinates) or AIJ:

```
std::vector<double>       A{4, -1, -1, ...};
std::vector<unsigned int> I{0,  0,  1, ...};
std::vector<unsigned int> J{0,  1,  0, ...};
```

triplet vector:

```
std::vector<std::tuple<unsigned int, unsigned int, double>>
  t{{0, 0, 4}, {0, 1, -1}, {1, 0, -1}, ...};
```

CSR (Compressed Sparse Row) or CRS (Compr. Row Storage) or Yale:

```
std::vector<double>       val{4, -1, -1, 4, -1, -1, 4, -1, -1, 4};
std::vector<unsigned int> col_idx{0, 1, 0, 1, 2, 1, 2, 3, 2, 3};
std::vector<unsigned int> row_ptr{0, 2, 5, 8, 10}; // n_rows + 1.
```

Class templates

Sparse matrix typical operations

- Insertion:
  `A[i][j] = x;`
- Deletion:
  `A[i].erase(j);` or `A.erase(i, j);`
- Random access:
  `x = A[i][j]; A[i][j] += y;`
- Sequential traversing:

```
for (row : A)
  {
    for (column : row)
      std::cout << column.value << " ";
    std::cout << std::endl;
  }
```

- Matrix-vector multiplication:
  `std::vector<double> y = A * x;`

The C++ standard is very permissive for the implementation of new containers, but:

- STL containers have non-virtual destructors!
- C.35: A base class destructor should be either public and virtual, or protected and non-virtual.

```
class Base {
public:
  ~Base { do_something(); }; // Non-virtual.
}

class Derived : public Base {
public:
  MyComplexType member;
  ~Derived { member.clear(); ... }
}

Base *var = new Derived;
delete var; // Calls var::~Base() but not var::~Derived()!
```

- Implement a `C++` class to represent a sparse matrix inheriting from suitable STL containers.
- Simplify random access, allocation, entry increment, sequential traversing.
- (Optional) Refactor the code: instead of inheriting, make the actual STL container a private class member. Which operators should be added for the code to compile?

Implement the following methods:

```
/// Convert row-oriented sparse matrix to AIJ format.
void
aij(std::vector<double> &        a,
    std::vector<unsigned int> &i,
    std::vector<unsigned int> &j);

/// Convert row-oriented sparse matrix to CSR format.
void csr(std::vector<double> &        a,
         std::vector<unsigned int> &col_ind,
         std::vector<unsigned int> &row_ptr);

/// Stream operator.
friend std::ostream &
operator<<(std::ostream &stream, Class templates}{Sparse matrix exercise}
           sparse_matrix &M);

/// Sparse matrix increment.
void sparse_matrix::operator+=(sparse_matrix &other);

/// Compute matrix-vector product.
friend std::vector<double>
operator*(sparse_matrix &M,
          const std::vector<double> &x);
```