Assignment 3: Templates and the Standard Library

May 4, 2017

1 Background

In this assignment we introduce templates and the use of the Standard Template Library. Your task is two-fold: first you are going to rewrite the linear algebra library provided by the book to be aware of templates so that matrics can store float, double or integer values (or even complex values as for example std::complex<double>). Furthermore, we will introduce the more efficient row-major storage layout and store all elements in a `std::vector<T>`. In the second part you are going to extend the library by a new class: a sparse vector which can store very large vectors with millions of dimensions of which only a few are different from 0.

2 Exercise 1: Rewrite of Linear Algebra

Our current implementation provided by the book is *not* a prime example for good C++ coding practice. Amongst the issues are:

- The library does its own memory management. There is no reason for this, as std::vector is designed for handling dynamically sized arrays and we can simply rely on it instead of reimplementing our own solution.
- The library uses a very inefficient storage layout for matrices using the array-of-arrays approach. We should exchange that by the row-major storage layout. The row-major storage layout stores rows of a matrix so that entries are easily reachable for the processor. Assume we are given a matrix

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} .$$

The row-major format stores the entries continuously in one large chunk of memory:

$$s = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{bmatrix} .$$

This layout is slightly more complex to handle because every time we need to access an element A_{ij} we have to compute the index in the array s.

You are going to add these changes to the library:

- As you are going to make changes to the library provided, you should start by making regression tests. A regression test ensures that after the change you get the same numerical results as you got before and thus gives a good hint whether your changes introduced a bug. Create a program with a small test for each method of the library. You will have to modify these tests slightly when you rework the classes.
- Add a template parameter to `Matrix` and `Vector` so that you can store arbitrary values of type T instead of double.
- Work out on a piece of paper at what position of s the element A_{ij} in a matrix with n rows and m columns is.
- Replace the `mData` member of the `Vector` and `Matrix` classes by an std::vector<T> and adapt all methods to this change.
- Add a function `std::vector<T> const& getStorage() const` returning a const reference to `mData`.
- Remove all methods which can now be auto generated by the compiler.

3 Exercise 2: Sparse Vectors

Sparse vector and matrix operations are a corner stone for many scientific applications, for example finite element methods. Matrices are called sparse when the number of non-zero elements is tiny, often less than 1%. Sparse Vectors have a very simple format and can be implemented efficiently with standard tools provided by C++. An example for a sparse vector is

$$v = \begin{bmatrix} 0 & 0 & 5 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \end{bmatrix} .$$

This vector has dimensionality 10 but only two non-zer0 elements, so we can store it efficiently by storing it in the form of two arrays: one storing the indices of the nonzero elements and the other storing the value:

$$\begin{aligned} \text{indices} &= \begin{bmatrix} 2 & 6 \end{bmatrix} \\ \text{values} &= \begin{bmatrix} 5 & 3 \end{bmatrix} \end{aligned}$$

Indices are sorted in increasing order so that it is easy to look up whether an element exist or not. Your assignment is to implement the methods of the class `SparseVector`. To keep your implementation as simple and efficient as possible, you can assume that adding or subtracting two numbers will not lead to 0 (which means we might in the end store a few additional zeros depending on the input).

Hints:

- For `getValue` and `setValue`, you can use `std::lower_bound` to find efficiently the position of an index in an array. Note that the iterator might be pointing to the end of the range (the queried index is larger than any index stored) or might not point to the actual index (in which cae it points to the position of the next smaller element). To use this function the elements must be inserted in an ordered fashion.
- You can use `std::vector<T>::insert` to insert directly behind a given position.
- use a reference as `cppreference.com` to find simple usage examples
- Remember that the iterators provided by an `std::vector` behave almost exactly like a pointer. E.g subtracting two iterators gives you their distance. Therefore look up your pointer arithmetics!
- `std::vector` Is not a mathematical vector, you can not add or multiply values for example!