Exercise Sheet #9

Problem 1 (*Containers*) 10 Pts

This problem will treat different container types and how they perform compared to the generic array. Therefore we want to store N elements, e. g. we want to store the hexadecimal representation (i.e. a string) of all numbers in the set $\{0, N-1\}$. The types of storages we want to investigate are the container types vector, map, list and additionally the array.

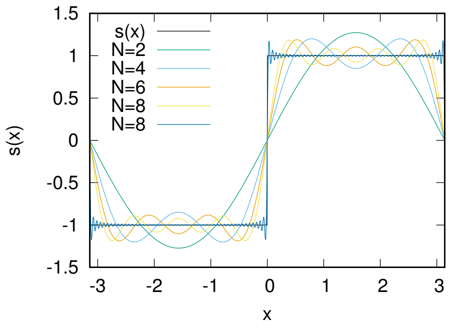
(a) Measure the time it needs to put all N elements into the respective storage for each type. Try with $N \in \{10^5, 10^6, 10^7, 10^8\}$, but keep an eye on your machine's memory. And do not forget to correctly delete the storages right after last usage. How can you speed up for the vector container? Can you estimate how much memory it should optimally take to store the N strings?

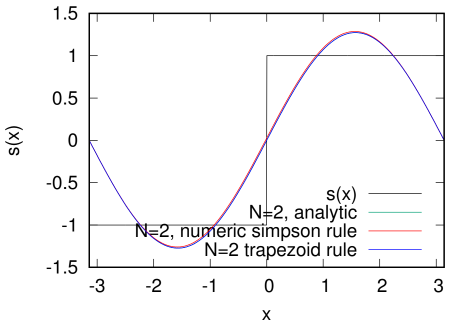
(b) Measure the access time, i. e. the time it needs to look up $K \sim 10^6$ random entries from the array, vector and map. Compare the average time needed to look up a single element for these storage types, using the highest possible N. Does this fit to what you know about the memory representation of the types?

(c) Measure the access time for list as well. Mind that this will be much slower, as you have to look up the entries by traversing the list manually, therefore use $K \sim 10^4$.

Iterators are a generalization of pointers and take care of pointer increment dynamics. Therefore, a program written with iterators can be easily adapted to different containers.

(d) Write a function `look_up_time` that takes as arguments a templated variable that is a pointer to a container, K, and N, and computes the average access time. This function has to be compatible with different containers (arrays, vectors and lists), so you should traverse the container. How does this affect performance? What is the problem if one were to use maps as well? **Hint**: You can use `auto` when instantiating iterators with containers' `begin` method (2 Pts)





Problem 2 (*Numeric Integration*) 10 Pts

Consider the square wave periodic function, defined by:

$$s(x) = \begin{cases} -1\,, & -\pi \le x < 0 \\ 1\,, & 0 \le x < \pi \end{cases}\,, \qquad s(x+2\pi) = s(x)\,. \tag{1}$$

(a) Calculate the Fourier series representation of the function analytically. Remember that, given an $f(x) = f(x+T)$ periodic function, the Fourier series expansion can be written as:

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{N\to\infty} \left(a_k \cos\frac{2\pi k x}{T} + b_k \sin\frac{2\pi k x}{T} \right)\,, \tag{2}$$

where the Fourier coefficients a_k and b_k are given by:

$$\begin{aligned} a_k &= \frac{2}{T}\int_{x_0}^{x_0+T} f(x)\cos\frac{2\pi k x}{T}\,\mathrm{d}x\,, \\ b_k &= \frac{2}{T}\int_{x_0}^{x_0+T} f(x)\sin\frac{2\pi k x}{T}\,\mathrm{d}x\,. \end{aligned} \tag{3}$$

(b) Compute the first ten pairs of Fourier coefficients, i. e. a_k and b_k for $k \in \{0, \ldots, 9\}$, by evaluating the integrals numerically. Do this using the Simpson rule as well as the trapezoid rule, which were both introduced in the lecture, evaluating the function at n points. Calculate errors between the analytically and numerically calculated coefficients as a function of n (let $n \in \{10, ..., 1000\}$, where n is the number of integration steps). Verify the error scaling given in the lecture by plotting this function in a log-log scale.

(c) Reproduce the plots above, considering the five partial sums $S_N(x)$ corresponding to $N \in \{2, 4, 6, 8, 100\}$. The Fourier series approximation of the square wave exhibits the so-called Gibbs phenomenon, which refers to the non-vanishing series expansion error close to the step. To verify this, plot the error $\varepsilon_N = |S_N(T/2N) - f(T/2N)|$ as a function of N. Verify that it does not converge to zero but a finite limit if you use exact coefficients. Does this also hold if you use numerically calculated coefficients (trapezoid & Simpson)?

Problem 3 (*Matrix Inversion using Gauss Elimination*) 5 Pts

In the lecture, you learned how to implement the Gauss elimination method to solve a linear set of equations. Now, given that finding the inverse of a $N \times N$ square matrix A is equivalent to finding the solution A^{-1} of the equation $AA^{-1} = I$, where I is the identity matrix, we can use the same Gauss method as for finding the solution $\mathbf{x}$ of a square linear system of equations $A\mathbf{x} = \mathbf{b}$. This can be seen by the fact that the matrix inversion problem is equivalent to solving N linear systems of size N, each of the form $AA_i^{-1} = I_i$, where A_i^{-1} and I_i is the i-th column of the inverse and the identity matrix, respectively.

Write a template function
`gaussInversion(double (&A)[N][N], double (&A_i)[N][N]){...}` that takes a matrix `A` by reference and writes the solution into `A_i`. You should also include appropriate error handling, i.e., return error messages if the matrix is not invertible. *Hint*: You could run the Gauss algorithm for each row, but this would be very inefficient. Rather, you should work with an augmented Matrix of the form $(A|I)$ and work on that.