Projekt 2: "fast" Matrix-Matrix multiplication

The standard algorithm to multiply two matrices A, $B \in \mathbb{R}^{n \times n}$ has complexity $\mathcal{O}(n^3)$. Volker Strassen published in 1969 the *Strassen-Algorithm* that reduced this complexity (i.e., the number of arithmetic operations) to $\mathcal{O}(n^{\log_2(7)})$.
The Strassen algorithm is as follows. For simplicity, let $n = 2^m$. The goal is to compute

$$C = AB. \tag{1}$$

The matrices are decomposed into 4 blocks of size 2^{m-1}

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \text{ und } C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \tag{2}$$

The standard multiplication can be written as

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}. \tag{3}$$

Another, equivalent, representation is obtained in terms of the matrices

$$
\begin{aligned}
M_1 &:= (A_{11} + A_{22})(B_{11} + B_{22}) && \text{(4a)}\\
M_2 &:= (A_{21} + A_{22})B_{11} && \text{(4b)}\\
M_3 &:= A_{11}(B_{12} - B_{22}) && \text{(4c)}\\
M_4 &:= A_{22}(B_{21} - B_{11}) && \text{(4d)}\\
M_5 &:= (A_{11} + A_{12})B_{22} && \text{(4e)}\\
M_6 &:= (A_{21} - A_{11})(B_{11} + B_{12}) && \text{(4f)}\\
M_7 &:= (A_{12} - A_{22})(B_{21} + B_{22}) && \text{(4g)}
\end{aligned}
$$

and the observation that

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7,\\
C_{12} &= M_3 + M_5,\\
C_{21} &= M_2 + M_4, \quad \text{und}\\
C_{22} &= M_1 - M_2 + M_3 + M_6.
\end{aligned}
$$

The multiplication in (4) is done recursively until only 1×1 matrices have to be multiplied.

1. Check the correctness of the above algorithm and prove its complexity.

2. Program the following functions for square matrices of size 2^m of type **float** in C or C++:

a) the standard algorithm for the matrix-matrix multiplication

b) the Strassen algorithm for the matrix-matrix multiplication

The function for the Strassen algorithm is recursive and it shouldn't allocate memory. Therefore, it should have a fourth argument with *work space*, e.g.,

```
float * A    = new float[n*n]; FillWithSomething(A);
float * B    = new float[n*n]; FillWithSomething(B);
float * C    = new float[n*n];
float * Work = new float[n*n];  // don't use 'new' again
ComputeStrassen(n,n,A,B,C,Work); // A*B=C using 'Work'
```

Document your source code sufficiently. Test your code with matrices for small m until you are convinced that your code works properly. Test first the function for the standard multiplication using, if necessary, `matlab`. Test your implementation of the Strassen algorithm by comparing with the standard multiplication. Document your test examples and results.

3. The complexity and the run-time behavior of the two algorithms are to be compared. Write a program that measure the CPU-time needed for the algorithms of Problem 2. Use the function clock(),

```
#include <ctime>
...
  clock_t start = clock();
  ComputeStrassen(...);
  clock_t end = clock();

  took = difftime(end,start)*1000.0/CLOCKS_PER_SEC;
  std::cout << "Took " << took << " [millisec]" << std::endl;
```

Write a program that inputs m from the command line and generates two matrices A and B of size $n = 2^m$ with random entries in $[0, 1]$. Measure the times needed:

```
  clock_t sStrassen = clock();
  ComputeStrassen(...);
  clock_t eStrassen = clock();

  clock_t sStandard = clock();
  ComputeStandard(...);
  clock_t eStandard = clock();

  std::cout << "time needed for n = " << n << std::endl
            << "Strassen: "
            << difftime(eStrassen,sStrassen)/CLOCKS_PER_SEC
            << " [sec]" << std::endl
            << "Standard: "
            << difftime(eStandard,sStandard)/CLOCKS_PER_SEC
            << " [sec]" << std::endl;
```

Call your program with $m = 6, \ldots, 11$ (or larger if your memory permits it). Collect the timings in a table. Plot the timings using `matlab`. What do you observe?

hint: Do *not* use the optimization flags of the compiler as this may improve the timings significantly for certain values.