Matrices in C

Exercise 1: Matrix addition

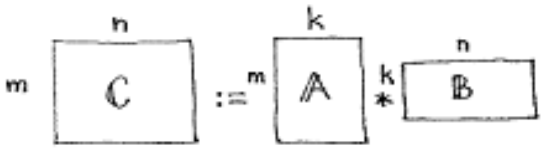
Write a C program, that adds two $m \times n$ matrices A and B, and stores the result in matrix C. Check the correctness of your program, e.g. by comparing your results with the same operation in MATLAB, Python (numpy), or any other application you are familiar with.
Obtain some timing results (use e.g. the C function `clock()` for your timings), for different values of m and n.
Note: to get meaningful results, you need to measure the same event several times (e.g. by adding a loop around it), and then take the average. As a rule of thumb, a total run time of 3-5 seconds should be achieved, to rule out side effects.

Exercise 2: Matrix times vector

Write a C program, that multiplies a $m \times n$ matrix A with a vector b (of length n), and store the result in the vector c. As in (1) above, use an external application to check the correctness of your code.
Obtain some timing results (use e.g. the C function `clock()` for your timings), for different values of m and n.

Exercise 3: Matrix multiplication

Write a C program, that multiplies a $m \times k$ matrix A with a $k \times n$ matrix B, and store the result in the $m \times n$ matrix C.



Hint: Use the conventions from the figure above to specify the dimensions of your matrices. This will make life easier later on.
As a starting point, you should choose matrices of a reasonable small size, e.g. $m = 3, n = 2, k = 5$. Initialize A and B according to

$$A(r, s) = 10.0 * r + s$$
$$B(r, s) = 20.0 * r + s$$

$r, s = 1, 2, ...$, and your result should look like this:

$$C = \begin{pmatrix} 4165.0 & 4230.0 \\ 7215.0 & 7330.0 \\ 10265.0 & 10430.0 \end{pmatrix}.$$

As in (1) above, you can use an external application to check the correctness of your code, or use the recipe described above.
Obtain some timing results (use e.g. the C function `clock()` for your timings), for different values of m, n and k.

Exercise 4: Dynamic allocation of matrices in C

If you have not done, yet, implement the matrix multiplication code above with dynamic allocation of the matrix-memory, such that your program can handle matrices of arbitrary sizes. See the lecture notes on 'Matrix computations' for an example code for allocation and de-allocation of matrices.

Exercise 5: Calling the BLAS Level 3 routine DGEMM

Note: This exercise is meant as a preparation to the first assignment!

After this exercise you should be able to write a program that calls the BLAS level 3 subroutine *dgemm()*, to multiply two matrices. You will need this in the first assignment, mainly for two reasons: to check your results against the results from the library routine, and to compare the performance.
The *dgemm()* routine is a general purpose routine for matrix multiplication (and addition), and it implements the formula

$$C = \alpha * A * B + \beta * C$$

We will restrict ourselves here to the case $C = A * B$, i.e. $\alpha = 1$ and $\beta = 0$, as illustrated in the figure above.
Implement a C program (or a subroutine), that calls `dgemm()` to do the matrix multiplication. There are several alternative implementations that you can choose from: CBLAS, BLAS with native C interface, and the original FORTRAN-based BLAS version. Below, you can find a short description of the interfaces, and how compile and link with the different libraries.
Common to all the interfaces is the fact, that there is a number of arguments (BLAS: 13, CBLAS: 14), that you have to provide in the function call — so please be careful when setting up the call. Take a look at the lecture notes, to read about the definitions of terms like *'leading dimension'*, etc.

Using `cblas_dgemm()`

Using the CBLAS interface is the most *"natural"* way for a C/C++ programmer to call BLAS functions, since it uses native C datatypes in the function calls. In addition to the standard BLAS functions, there is usually one additional function call argument, `CBLAS_ORDER`, that describes the ordering of the matrix data passed in. This can either be `CblasRowMajor`, i.e. the natural C ordering, or `CblasColMajor`, i.e. for matrices that use column-major ordering (like in FORTRAN).
Since there is no man pages for CBLAS available (at least on our systems), we provide a PDF file (`cblas_cinterface.pdf`) with a reference guide to CBLAS on DTU Learn.
On our systems, CBLAS is part of the ATLAS library:

- you need to include the `cblas.h` header file
- you need to link against the ATLAS library, i.e. use the `-L /usr/lib64/atlas -lsatlas` linker option.

Calling the native (FORTRAN) BLAS `dgemm()`

This is added for the sake of completeness, in case you want to (or are in a situation, where you have to) use the native BLAS interface.
BLAS routines like *dgemm()* are FORTRAN-style routines. If you call FORTRAN BLAS routines from a C-language program you must follow the FORTRAN-style calling conventions:

a) Pass variables by address as opposed to passing by value.

b) Be sure to pass data FORTRAN-style, i.e. data stored column-major rather than row-major order. (see note (*) below)

c) You have to make sure that the addresses passed to the routines point to contiguous memory - this is something you must have in mind when implementing matrices of arbitrary size, i.e. dynamic allocation at runtime.

d) You need to add a trailing underscore to the function name, i.e. you need to call `dgemm_()`.

(*) There are several ways to accomplish this, e.g. storing the data in column-major order, converting the data before calling dgemm(), let dgemm() do the work (check the argument list!), etc. Since we will later on use row-major order - i.e. the 'native' way in C/C++, you shouldn't waste any time to implement a column-major implementation.
Hint: There is another way of avoiding an ordering change in this case — remember your Linear Algebra class and the rules for the product of transposed matrices.