Adavnced Research Computing: Practical session

1 Matrix multiplication

Write a program to accomplish the following:

- Take as user-input two arbitrary double-precision matrices from a text-file provided as a command-line argument.
- Multiply these two matrices together and output the solution to the terminal.
- You should not make use of pre-existing libraries (e.g. BLAS, Eigen, etc.) except for unit-testing.

For testing purposes, three files are provided in the `Practical` folder:

- `Matrix_Test_A_input.txt`: Two 3 3 matrices to be multiplied together.
- `Matrix_Test_B_bad_input.txt`: A 3 4 and a 3 3 matrix to be multiplied together. This clearly will not work.
- `Matrix_Test_C_large_input.txt`: A 64 96 and a 96 128 matrix to multiply together.

The file layout is as follows:

```
ROWS_1
COLS_1
A_00 A_01 A_02 A_03 ...
A_10 A_11 A_12 A_13 ...
...
A_N0 A_N1 ... A_NM
ROWS_2
COLS_2
B_00 B_01 B_02 B_03 ...
B_10 B_11 B_12 B_13 ...
...
B_N0 B_N1 ... B_NM
```

i.e. give the dimensions for each matrix, followed by the elements for each matrix. Numbers are separated using white-space.

1 1 Code running

In the standard case, your code would be run as:
`./multiply ./Matrix_Test_A_input.txt`

In order to save having to generate very large data-sets, your program should also accept matrix dimensions as options:

`./multiply 1024 512 512 2048`
which would generate two matrices of size 1024 512 and 512 2048, print them out, multiply them together,

and then print their product.

See `https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution` for how to generate reasonable random numbers. Try generating matrix elements in a small range including positive and negative number.

1 2 Optimization

You should make use of `valgrind` to try to optimize your code as far as possible. Points to note:

- Optimizations are only likely to become important for very large matrices, at least 100 100 if not 1000 1000 or larger.
- Be careful that your measured performance is not dominated by input/output.
- This level of optimization is difficult and involves a wide range of techniques, some of which are CPU and compiler dependent.
- As always, the optimal answer is an implementation of BLAS (or other external library).

1 3 Features

Your solution should have the following features:

- Make use of `CMake` to identify all compilers and other tools.
- Have all functions commented using Doxygen.
- Have unit-testing to ensure that invalid file-input (such as `Matrix_Test_B_bad_input.txt` above) is detected and does not result in seg-faults or other behaviour.

1 4 Suggestions

Some suggestions for how to lay out your code:

- Split your code into (at least) three files:

  `Main.C`: Handles command-line parameters, and calls input/multiply/output routines.

  `Multiply.C`: Contains functions to read input from a file, multiply two matrices, and output a result matrix.

  `Test.C`: Contains unit-tests written with Boost::Test or GoogleTest that test the functionality of at least the input and multiply functions.

- Separate out the input/multiplication/output functionality into three separate functions within `Multiply.C`.
- Write tests that are designed to fail. For example, the invalid `Matrix_Test_B_bad_input.txt` above should be handled via an exception being thrown.
- The unit-tests could make use of the BLAS or Eigen libraries to check that the result is correct. (Or, use simple matrices as input and calculate the result by hand.)

1 5 Other hints

- Decide on your data layout for matrices first (maybe based on BLAS for compatibility).
- The BLAS routine for multiplying two double-precision matrices together is `cblas_dgemm`. An example program (for `float`) can be found at: `https://www.gnu.org/software/gsl/doc/html/cblas.html examples`.
- Create trivial `struct`s corresponding to di erent exceptions/error cases.
- Write skeleton-functions that are well-documented, i.e. detail the input/output conditions carefully.
- Write a simple `CMakeLists.txt` that initially just compiles the source-code.
- Extend `CMake` to compile two separate executables for normal running and for unit-tests.
- Add support for `clang-format` and `clang-tidy` that raises errors if code is not well-structured.