ACSE-5: Advanced Programming Assignment

Implement an algorithm to solve the linear system **Ax**=**b**, where **A** is a real positive definite matrix, and **x** and **b** are vectors. The program which implements this will build upon the Matrix.cpp and Matrix.h libraries that we have constructed in class (as such **A** or $\mathbf{A}^{-1}$ must be of type Matrix, or a class derived from this, such as CSRMatrix, or your own custom subtype).

Your program must include a main method which loads/creates an example **A**, **x** and **b** (of any size greater than 10x10), and calls a method named "solver", that takes **A** and **b** as inputs and returns **x** as an output (these inputs/outputs may be passed by value, by reference, or as pointers, with the output as a return arguments or as an arguments to the function). *Any deviation from this strict argument list will be penalised.*

You may implement any type of linear solver, from direct solver to iterative solver, from easy to hard (Jacobi, Gauss-Seidel, LU factorisation, GMRES, multigrid, etc) as long as it can solve a linear system involving a generic positive definite matrix of a given size. This matrix may be stored dense, sparsely or in a banded format depending on your preference. If you chose to build multiple different types of solvers (e.g., a dense Jacobi and a dense Gauss-Seidel), there must still only be one main method that loads/creates **A**, etc, and then each different solver type should be called from that main method. Do not create a new main method (nor a new Visual Studio project) for each different solver. Also, if you have chosen to build multiple solver types using a single matrix format (e.g., a dense Jacobi and a dense Gauss-Seidel), do not create multiple versions of your Matrix.cpp and Matrix.h files. *Doing either of these things will be penalised.*

Do use methods/techniques that you have learned as part of this or previous courses. Try to incorporate programming techniques and styles that you have learned in class. Namely, inheritance, polymorphism, templates, smart pointers, loop ordering that respects cache hierarchies, simple loops that are easily vectorised, BLAS/LAPACK calls if desired. **Note:** this does not mean you can just call the linear solver routines in BLAS/LAPACK and write a wrapper around those; e.g., there is an LU decomposition for dense matrices in LAPACK, *do not just call that*. If you want to use BLAS/LAPACK routines limit yourself to using lower level routines, e.g., if you chose to build an LU decomposition you may wish to call daxpy from BLAS to help you do vector addition efficiently, etc.

Feel free to use techniques that are more advanced than those covered in class (!). Document your code (explain in the code what the different blocks of code are supposed to do), and do not copy code from the internet or other sources. Any algorithms that we have not covered in class must be referenced (including those you learnt in other courses in this MSc).

Your code will be compiled and executed as part of your evaluation. *If your code does not compile, or does not solve a positive definite linear system of any size correctly, you will be heavily penalised.* We will mark the projects based on: code structure and style (30/100), implementing of loading/writing/linear solver (30/100), user experience, creativity and execution (20/100), code documentation/commenting (10/100), and short report (10/100). To submit, follow these steps:

Step 1>> Register your team (of three people) in the GitHub classroom @ https://classroom.github.com/g/Qi0AX3Uk

Step 2>> Pick the linear system you would like to solve. This could be as simple as a random 10x10 matrix with large diagonals entries with random **b** vector.

Step 3>> Choose or design the linear solver to be implemented. Your solution method can be as simple as a Jacobi method on a matrix stored densely. A more challenging example could be a Cholesky factorisation on a sparse matrix. Implementing more challenging linear solvers will result in better marks for this assignment.

You can also chose to build multiple linear solvers. We should note however, that a correct, documented and commented assignment implementing simple linear solvers is preferable to a difficult to use, incorrect, badly implemented version of a hard algorithm.

Anything you could add to your short report showing that you understand the algorithm, the performance of your method (perhaps you might want to test the performance of your method as the matrix size increases and compare this to theoretical predictions, or compare the performance of different linear solvers) and any advantages/drawbacks to your approach would be advantageous. This includes design decisions that you might do differently if you had to do this assignment again.

Step 4>> Upload your code onto your GitHub classroom repository before midnight of **Sunday** February 7nd 2021. This should include: one main.cpp file including a main method and a solver method as described above, one or more .cpp and .h files containing your classes and any dependencies (like Matrix.h and Matrix.cpp), and any other additional header or source files you create (or MSCV project files). Make sure to upload all files required for your project to be compiled and executed.

Step 5>> Prepare a short report (no more than a 4 pages) describing your code (Word or PDF, single space 11pt).
Provide a short explanation of the design of your code, its structure, input/output, and how to execute it. Briefly describe the strengths and/or weaknesses of your chosen design of methods and code structure. Make sure to include the date, the name of your solver algorithm(s), programming team name, the names of each member in the group, link to your Github repo, and a rough breakdown of who did what in your group to the document.
If you have additional questions email: s.dargaville@imperial.ac.uk