The University of Nottingham

SCHOOL OF MATHEMATICAL SCIENCES

AUTUMN SEMESTER 2021-2022

MATH4063 - SCIENTIFIC COMPUTING AND C++

Coursework 1 - Released 8th November 2021, 10am

Your work should be submitted electronically via the MATH4063 Moodle page by the deadline indicated there. Since this work is assessed, your submission must be entirely your own work (see the University's policy on Academic Misconduct). Submissions up to five working days late will be subject to a penalty of 5% of the maximum mark per working day.

The marks for each question are given by means of a figure enclosed by square brackets, eg [20]. There are a total of 100 marks available for the coursework and it contributes 45% to the module. The marking rubric available on Moodle will be applied to each full question to further break down this mark.

You are free to name the functions you write as you wish, but bear in mind these names should be meaningful. Functions should be grouped together in `.cpp` *files and accessed in other files using correspondingly named* `.hpp` *files.*

All calculations should be done in double precision.

A single zip file containing your full solution should be submitted on Moodle. This zip file should contain three folders called `main`, `source` *and* `include`*, with the following files in them:*

main:

- q1d.cpp
- q2c.cpp
- q3b.cpp
- q4b.cpp

source:

- csr_matrix.cpp
- linear_algebra.cpp
- finite_difference.cpp

include:

- csr_matrix.hpp
- linear_algebra.hpp
- finite_difference.hpp

Hint: When using a C++ `struct` with header files, the whole struct needs to be defined fully in the header file, as well as in the corresponding `.cpp` file.

In this coursework you will build from scratch a 2D finite difference solver to solve the following PDE boundary value problem

$$-\Delta u + u = f(x,y), \quad (x,y) \in \Omega := (0,1)^2, \tag{1}$$
$$u(x,y) = 0, \quad (x,y) \in \partial\Omega, \tag{2}$$

where $f : \Omega \to \mathbb{R}$.

In order to solve this problem, you will first define a sparse matrix structure, then write subprograms to apply an iterative linear algebra solver and finally build and solve the linear system arising from the finite difference approximation of (1)-(2).

1. Matrices arising from the discretisation of partial differential equations using, for example, finite difference methods, are generally sparse in the sense that they have many more zero entries than nonzero ones. We would like to avoid storing the zero entries and only store the nonzero ones.

   A commonly employed sparse matrix storage format is the *Compressed Sparse Row* (CSR) format. Here, the nonzero entries of an $n \times n$ matrix are stored in a vector `matrix_entries`, the vector `column_no` gives the column position of the corresponding entries in `matrix_entries`, while the vector `row_start` of length $n+1$ is the list of indices which indicates where each row starts in `matrix_entries`. For example, consider the following

$$A = \begin{pmatrix} 8 & 0 & 0 & 2 \\ 0 & 3 & 0 & 1 \\ 0 & 0 & 4 & 0 \\ 2 & 1 & 0 & 7 \end{pmatrix} \longrightarrow \begin{array}{r} \texttt{matrix\_entries} = \begin{pmatrix} 8 & 2 & 3 & 1 & 4 & 2 & 1 & 7 \end{pmatrix} \\ \texttt{column\_no} = \begin{pmatrix} 0 & 3 & 1 & 3 & 2 & 0 & 1 & 3 \end{pmatrix} \\ \texttt{row\_start} = \begin{pmatrix} 0 & 2 & 4 & 5 & 8 \end{pmatrix} \end{array}$$

   Note, in the above, C++ indexing has been assumed, *i.e*, indices begin at 0.

   In fact, as the matrix above is symmetric, we could avoid storing the lower part of the matrix and store it instead as;

$$A = \begin{pmatrix} 8 & 0 & 0 & 2 \\ 0 & 3 & 0 & 1 \\ 0 & 0 & 4 & 0 \\ 2 & 1 & 0 & 7 \end{pmatrix} \longrightarrow \begin{array}{r} \texttt{matrix\_entries} = \begin{pmatrix} 8 & 2 & 3 & 1 & 4 & 7 \end{pmatrix} \\ \texttt{column\_no} = \begin{pmatrix} 0 & 3 & 1 & 3 & 2 & 3 \end{pmatrix} \\ \texttt{row\_start} = \begin{pmatrix} 0 & 2 & 4 & 5 & 6 \end{pmatrix} \end{array}$$

   (a) In `csr_matrix.cpp`, define a C++ `struct` to store a matrix in CSR format. In addition to `matrix_entries`, `column_no` and `row_start`, you should store the number of rows of the matrix explicitly and whether or not the matrix is being stored as a symmetric matrix.

   (b) In `csr_matrix.cpp`, write two C++ functions that will set up the matrix A from above in CSR format. The first function should set A with no assumption that is is symmetric, while the second should set it up as a symmetric matrix. Remember, if you are using dynamically allocated memory, then you should also have corresponding functions that will deallocate the memory you have set up.

   (c) In `csr_matrix.cpp`, write a C++ function that takes as input a matrix A stored in CSR format and a vector **x** and computes the product A**x**. Your function should be able to handle both the case where A is nonsymmetric and and where it is symmetric.

   (d) By setting a vector $\mathbf{x} = (6, 8, 2, 5)^{\top}$, write a test program in `q1d.cpp` to compute and print to the screen the product A**x**, where A is the matrix given above. Your code should test both scenarios, where A is stored as a nonsymmetric matrix and a symmetric one.

[30 marks]

2. Suppose we wish to find $\mathbf{x} \in \mathbb{R}^n$ such that

$$A\mathbf{x} = \mathbf{b}, \tag{3}$$

where A is a symmetric positive definite $n \times n$ matrix and $\mathbf{b}$ is an n–vector.

One algorithm for solving this problem is as follows: given an initial vector $\mathbf{x}_0$ and a termination tolerance tol, set

$$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0, \quad \mathbf{p}_0 = \mathbf{r}_0.$$

For $k = 0, 1, 2, ...$ repeat the following steps until the termination condition $||\mathbf{r}_k||_2 \leq tol$ is satisfied:

(i) $\alpha_k = \mathbf{r}_k^\top \mathbf{r}_k / \left(\mathbf{p}_k^\top A \mathbf{p}_k\right)$,

(ii) $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$,

(iii) $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k A \mathbf{p}_k$,

(iv) $\beta_k = \mathbf{r}_{k+1}^\top \mathbf{r}_{k+1} / \mathbf{r}_k^\top \mathbf{r}_k$,

(v) $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$.

(a) In `linear_algebra.cpp`, write a C++ function which implements the method above to find an approximation $\hat{\mathbf{x}}$ to

$$A\mathbf{x} = \mathbf{b},$$

where A is an $n \times n$ symmetric positive definite matrix, stored using the CSR format. The function should take tol as one of its inputs. The initial guess for the solution $\mathbf{x}_0$ should be set equal to the zero vector. The history of $||\mathbf{r}_k||_2$, including the iteration counter k (starting from $k = 0$) should be output as the method progresses.

Hint: Your function should make use of simpler functions, that you should write, implementing commonly used vector operations from the algorithm including, but not limited to,

- *a function that will compute the dot product of two vectors;*
- *a function that will compute the 2-norm of a vector;*
- *a function that will compute the linear combination $\mathbf{u} + a\mathbf{w}$ for a scalar a and two vectors $\mathbf{u}$ and $\mathbf{w}$.*

Your code should also make use of functions from Q1.

(b) In `csr_matrix.cpp`, write a C++ function that will read from a file a matrix already stored in CSR format and a vector. You may assume the file structures are as in `matrix1.dat` and `vector1.dat` on Moodle and you may use these data files to test your function.

(c) Write a test program in file `q2c.cpp` that will read in the matrix A from `matrix2.dat` and the vector $\mathbf{x}$ from `vector2.dat`, compute $\mathbf{b} = A\mathbf{x}$, then use your function from part (a) to find an approximation $\hat{\mathbf{x}}$ to $\mathbf{x}$. You should use a tolerance of 1×10^{-10} and finally print to the screen the error $||\mathbf{x} - \hat{\mathbf{x}}||_2$.

[30 marks]

3. Suppose that a uniformly spaced mesh has been set up, with $n+1 \geq 3$ nodes in each coordinate direction, so that nodes have coordinates (x_i, y_j) where

$$x_i = ih, \quad 0 \leq i \leq n,$$
$$y_j = jh, \quad 0 \leq j \leq n,$$

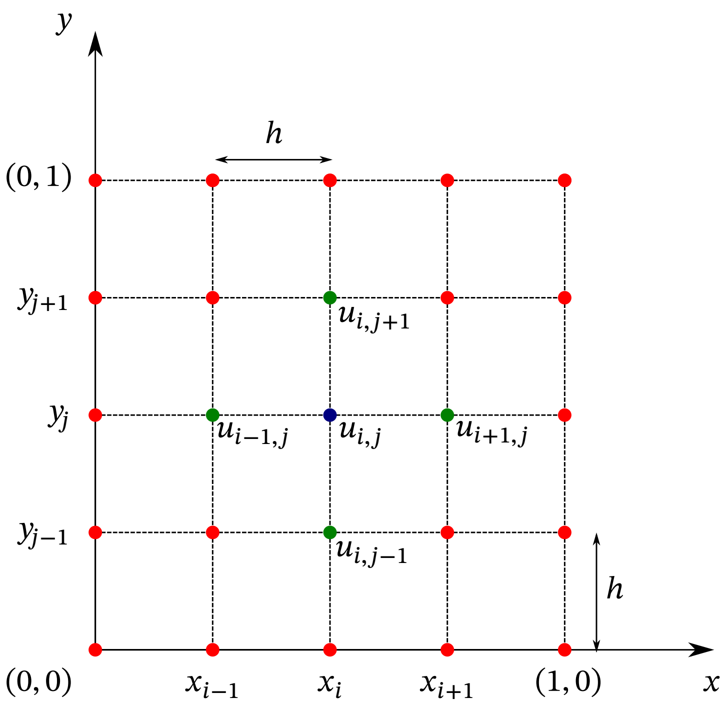
and

$$h = \frac{1}{n}.$$

Suppose further that $u_{i,j} \approx u(x_i, y_j)$ for $0 \leq i, j \leq n$. Then, the centred finite difference approximation to (1)-(2) is to find $\{u_{i,j}\}_{i,j=1}^{n-1}$ such that

$$-\frac{u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i,j+1} + u_{i+1,j}}{h^2} + u_{i,j} = f(x_i, y_j), \quad 1 \leq i, j \leq n-1. \tag{4}$$

The boundary condition is applied on the boundary so $u_{i,j} = 0$ for all i, j such that $(x_i, y_j) \in \partial\Omega$. A mesh with $n = 4$ is shown below.



With a re-indexing of the nodal points and renaming of the unknowns, so that

$$(\hat{x}_{(i-1)(n-1)+j-1}, \hat{y}_{(i-1)(n-1)+j-1}) = (x_i, y_j), \quad 1 \leq i, j \leq n-1$$
$$\hat{u}_{(i-1)(n-1)+j-1} = u_{i,j}, \quad 1 \leq i, j \leq n-1,$$
$$\hat{f}_{(i-1)(n-1)+j-1} = f(x_i, y_j), \quad 1 \leq i, j \leq n-1,$$

we can rewrite (4) as the matrix problem

$$A\mathbf{U} = \mathbf{F}, \tag{5}$$

where A is an $(n-1)^2 \times (n-1)^2$ symmetric, positive definite matrix, $\mathbf{U} = \{\hat{u}_k\}_{k=0}^{(n-1)^2-1}$ and $\mathbf{F} = \{\hat{f}_k\}_{k=0}^{(n-1)^2-1}$.

(a) In `finite_difference.cpp`, write a C++ function that will create matrix A in CSR format and the RHS vector **F**. This function should take as input $n \geq 2$ and a general RHS function f.

Hint: First find A by hand and be very careful of how the equations for unknowns adjacent to the boundary are set up. You may store the matrix either as if it is symmetric or not. The matrices in `matrix1.dat` and `matrix2.dat` are examples of A stored as symmetric matrices with $n = 3$ and $n = 9$, respectively.

(b) Write a main program in `q3b.cpp` that will use the function from part (a) to set up A and **F** and then call the linear solver from Q2(a) to find **U**, for $n = 16, 32, 64, 128, 256$. For the RHS function, use

$$f(x, y) = (2\pi^2 + 1)\sin(\pi x)\sin(\pi y).$$

In turn, this means the exact solution is given by

$$u(x, y) = \sin(\pi x)\sin(\pi y).$$

In addition to solving the approximate solution, for each value of n, your code should print to the screen

$$u_{n,\max} := \max_{k=0}^{(n-1)^2-1} \hat{u}_k.$$

Note, if you are unable to get the iterative solver from Q2 working, then you may create the matrix A as if it were a dense matrix (*i.e* store all the zero entries) and use the function `PerformGaussianElimination` from `gaussian_elimination.cpp` on Moodle to solve the system of equations. This will incur a small penalty. Note, an illustration of the use of `PerformGaussianElimination` can be found in the `main` program inside `gaussian_elimination.cpp`.

[30 marks]

4. (a) *Preconditioned* iterative methods attempt to solve the following modified version of (3)

$$P^{-1}A\mathbf{x} = P^{-1}\mathbf{b}, \tag{6}$$

where P (the preconditioner) is some easily inverted approximation to A, such that $P^{-1}A$ is also symmetric positive definite.

By duplicating and modifying your code from Q2(a), in `linear_algebra.cpp`, write a function to apply the algorithm from Q2 to the preconditioned system (6), where P is the **diagonal component** of A. This function should also output the history of $||\mathbf{r}_k||_2$ and the iteration counter k (starting from $k = 0$) as the method progresses.

(b) Write a test program called `q4b.cpp` that will run the same test problem as in Q3(b), but make use of the preconditioned iterative solver from part (a) to solve the linear system.

[10 marks]