University of Nottingham
School of Mathematical Sciences

G14SCC SCIENTIFIC COMPUTATION AND C++

Submission Date: Monday, 27th November 2017 **Assessed Coursework 2**

The following questions are to be used for the coursework assessment in the module G14SCC.
A single zip or tar file containing your solution should be submitted through the module webpage on Moodle as per the instructions on the Coursework Submission form.
The style and efficiency of your programs is important. A barely-passing solution will include attempts to write programs which include some of the correct elements of a solution. A borderline distinction solution will include working code that neatly and efficiently implements the relevant algorithms, and that shows evidence of testing.
An indication is given of the approximate weighting of each question by means of a figure enclosed by square brackets, eg [12].

All calculations should be done in double precision.

1. Matrices arising from the discretisation of partial differential equations using, for example, finite element or finite difference methods, are generally sparse in the sense that they have many more zero entries than 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

$$A = \begin{pmatrix} 8 & 0 & 0 & 2 \\ 0 & 3 & 0 & 1 \\ 0 & 0 & 4 & 0 \\ 2 & 1 & 0 & 7 \end{pmatrix} \quad \longrightarrow \quad \begin{aligned} \texttt{matrix\_entries} &= \begin{pmatrix} 8 & 2 & 3 & 1 & 4 & 2 & 1 & 7 \end{pmatrix} \\ \texttt{column\_no} &= \begin{pmatrix} 1 & 4 & 2 & 4 & 3 & 1 & 2 & 4 \end{pmatrix} \\ \texttt{row\_start} &= \begin{pmatrix} 1 & 3 & 5 & 6 & 9 \end{pmatrix} \end{aligned}$$

   (a) In a file `sp_matrix_multiply.cpp` write a C++ function called `sp_matrix_multiply` which computes the product of an $n \times n$ matrix A stored in CSR format and a vector x. The function should take `matrix_entries`, `column_no`, `row_start` (for A), x and n as input and return as output the product Ax. [10]

   (b) Setting the vector $x = (6, 8, 2, 5)^\top$, write a test program called `q1b.cpp` to compute the product Ax, where A is the matrix given above. Note that all arrays should be dynamically allocated and deleted after use. [5]

2. The Conjugate Gradient (CG) method is an algorithm for solving the matrix problem: find $x \in \mathbb{R}^n$ such that

$$Ax = b,$$

   where $A = \{A_{i,j}\}_{i,j=1}^n$ is a symmetric positive definite $n \times n$ matrix and b is an n–vector.

   The CG method is defined as follows: given an initial vector x_0 and a termination tolerance tol, set

$$r_0 = b - Ax_0, \quad p_0 = r_0.$$

   For $k = 0, 1, 2, \ldots$ repeat the following steps until the termination condition $\|r_k\|_2 \leq tol$ is satisfied:

(i) $\alpha_k = r_k^{\mathrm{T}} r_k / \left(p_k^{\mathrm{T}} A p_k\right)$,

(ii) $x_{k+1} = x_k + \alpha_k p_k$,

(iii) $r_{k+1} = r_k - \alpha_k A p_k$,

(iv) $\beta_k = r_{k+1}^{\mathrm{T}} r_{k+1} / r_k^{\mathrm{T}} r_k$,

(v) $p_{k+1} = r_{k+1} + \beta_k p_k$.

(a) In a file `dot_product.cpp` write a C++ function called `dot_product` which calculates the dot product of two vectors. [2]

(b) In a file `copy_vec.cpp` write a C++ function called `copy_vec` which makes a copy of a double precision vector. [2]

(c) In a file `vec_plus_scalar_times_vec.cpp` write a C++ function called `vec_plus_scalar_times_vec` which performs the following operation:

$$x = y + \alpha z,$$

where x, y, and z are n–vectors, and α is a real number. [2]

(d) In a file `cg.cpp` write a C++ function called `cg` which implements the CG method to solve

$$Ax = b,$$

where A is an $n \times n$ symmetric positive definite matrix, stored using the CSR format. The function should take `matrix_entries`, `column_no`, `row_start` (for A), b, n, and the convergence tolerance *tol* as input and return as output the solution x. The initial guess for the solution x_0 should be set equal to zero. The history of $\|r_k\|_2$, including the iteration counter k (starting from $k = 0$) should be output as the method progresses. [20]

(e) Using the stopping criterion $\|r_k\|_2 \leq 10^{-10}$, in a file called `q2e.cpp` write a C++ program to compute the solution to the matrix problem $Ax = b$ defined in Question 1 using the CG method (b is the vector determined in 1(b)). Report the history of k and $\|r_k\|_2$ as the method progresses, and output your final solution. [4]

3. Consider the following problem: find u such that

$$-u'' + u = 1, \quad x \in (0, 1),$$

subject to the boundary conditions

$$u(0) = u(1) = 0.$$

To define a finite difference method for the numerical approximation of the above differential equation, we subdivide $[0, 1]$ into N subintervals $[x_i, x_{i+1}]$, $i = 0, \ldots, N-1$, by the points

$$x_i = ih, \quad i = 0, \ldots, N, \qquad h = 1/N, \quad N \geq 2.$$

Then, writing U_i to denote the approximation to $u(x_i)$, $i = 0, \ldots, N$, we introduce the finite difference method: find $U = [U_0, U_1, \ldots, U_N]^{\top}$, such that

$$-\frac{U_{i-1} - 2U_i + U_{i+1}}{h^2} + \frac{U_{i-1} + 4U_i + U_{i+1}}{6} = 1,$$

for $i = 1, \ldots, N-1$, where $U_0 = 0$ and $U_N = 0$.

By rewriting the above finite difference scheme in terms of a matrix system, in a file `q3.cpp` write a C++ program which solves the resulting system of equations using the CG method implemented in question 2; here, the matrix should be stored in CSR format. Compute the finite difference approximation on the sequence of meshes, with $h = 1/8$, $1/16$, and $1/32$. Note that all arrays used to store the matrix A in CSR format, as well as the right-hand side vector and the solution vector should be dynamically allocated and deleted after use. For each mesh, set *tol*$= 10^{-10}$ and output the following quantities:

- The history of the k and $\|r_k\|_2$ as the CG method progresses.
- The finite difference solution U.
- ℓ_∞–norm of the error in the finite difference approximation to u, evaluated at the mesh points, on each of the meshes employed.

[20]

4. In practice the convergence of the CG method may be very slow for large matrix problems; to accelerate the convergence a so-called preconditioned variant of the CG method is generally implemented. The idea here is to replace $Ax = b$ by another symmetric positive definite system with the same solution. In practice, the preconditioned CG algorithm requires knowledge of a symmetric positive definite matrix M which represents an approximation to A^{-1}. Given M, the Preconditioned CG method is defined as follows: given an initial vector x_0 and a termination tolerance tol, set

$$r_0 = b - Ax_0, \quad z_0 = Mr_0, \quad p_0 = z_0.$$

For $k = 0, 1, 2, \ldots$ repeat the following steps until the termination condition $\|r_k\|_2 \leq tol$ is satisfied:

(i) $\alpha_k = z_k^{\mathrm{T}} r_k / \left(p_k^{\mathrm{T}} A p_k\right)$,
(ii) $x_{k+1} = x_k + \alpha_k p_k$,
(iii) $r_{k+1} = r_k - \alpha_k A p_k$,
(iv) $z_{k+1} = M r_{k+1}$,
(v) $\beta_k = z_{k+1}^{\mathrm{T}} r_{k+1} / z_k^{\mathrm{T}} r_k$,
(vi) $p_{k+1} = z_{k+1} + \beta_k p_k$.

Each step of the preconditioned conjugate gradient algorithm requires the computation of the n–vector $z_k = (z_{k,1}, z_{k,2}, \ldots, z_{k,n})^\top$ which satisfies

$$z_k = M r_k,$$

where $r_k = (r_{k,1}, r_{k,2}, \ldots, r_{k,n})^\top$ is the residual vector at the kth iterate and M is a symmetric positive definite matrix which represents an approximation to A^{-1}. Very often, M cannot be constructed explicitly; instead, simply the action of M applied to r_k is computed. For example, this may be done by applying a small number of iterations of a (cheap) iterative solver to the linear system

$$A z_k = r_k.$$

Of course, if this problem is solved exactly, then the preconditioned conjugate gradient algorithm will terminate in one step. Thereby, only a few iterations may need to be applied in order to obtain an effective preconditioner.

A simple example of such an algorithm is the symmetric Gauss-Seidel method, which is outlined below:

- $z_k^0 = 0$.

- For $l = 0$ until convergence do
  - for $j = 1, 2, \ldots, n$:

$$z_{k,j}^{l+1/2} = \frac{1}{A_{j,j}} \left[r_{k,j} - \sum_{m<j} A_{j,m} z_{k,m}^{l+1/2} - \sum_{m>j} A_{j,m} z_{k,m}^{l} \right]$$

  - for $j = n, n-1, \ldots, 1$:

$$z_{k,j}^{l+1} = \frac{1}{A_{j,j}} \left[r_{k,j} - \sum_{m<j} A_{j,m} z_{k,m}^{l+1/2} - \sum_{m>j} A_{j,m} z_{k,m}^{l+1} \right]$$

(a) In a file `symmetric_gauss_siedel.cpp` write a C++ function called `symmetric_gauss_siedel` which implements the above symmetric Gauss-Seidel method to solve the problem

$$Ax = b,$$

where A is an $n \times n$ matrix, stored using the CSR format. Setting the initial guess vector $x^0 = \mathbf{0}$, the algorithm should terminate when *either* a maximum number of iterations $n_{\max}$ have been computed, or when the error between successive iterates satisfies

$$\max_{1 \le i \le n} |x_i^{k+1} - x_i^k| \le tol_{GS}.$$

The function should take `matrix_entries`, `column_no`, `row_start` (for A), b, n, the convergence tolerance tol_{GS}, $n_{\max}$, and the Boolean variable `output_errors` (which determines whether the error between successive iterations is output to the console), as input and return as output the solution x. [12]

(b) Setting $tol_{GS} = 10^{-10}$ and $n_{\max} = 20$ in a file called `q4b.cpp` write a C++ program to compute the solution to the matrix problem $Ax = b$ defined in Question 1 using the symmetric Gauss-Seidel method (b is the vector determined in 1(b)). Report the history of k and $\max_{1 \le i \le n} |x_i^{k+1} - x_i^k|$ as the method progresses, and output your final solution. [4]

(c) In a file `pcg.cpp` write a C++ function called `pcg` which implements the preconditioned conjugate gradient method, where the preconditioner is based on employing the symmetric Gauss-Seidel method. The function should take `matrix_entries`, `column_no`, `row_start` (for A), b, n, the convergence tolerance tol, tol_{GS}, and $n_{\max}$ as input and return as output the solution x. The initial guess for the solution x_0 should be set equal to zero. The history of $\|r_k\|_2$, including the iteration counter k (starting from $k = 0$) should be output as the method progresses. Note that the output from the symmetric Gauss-Seidel preconditioner should be suppressed, i.e., set `output_errors=false`. [15]

(d) To test `pcg.cpp` in a file `q4.cpp` repeat the computations undertaken in question 3 with $tol = 10^{-10}$, $n_{\max} = 10$, and $tol_{GS} = 10^{-20}$. Here you should report the history of $\|r_k\|_2$, including the iteration counter k (starting from $k = 0$), and the ℓ_∞–norm of the error in the finite difference approximation to u, evaluated at the mesh points, on each of the meshes employed (for brevity, there is no need to output the finite difference solution U). Comment on your results. [4]