ADVANCED MATHEMATICAL PROGRAMMING

MTH399 / U15161 – Level 3

UNIVERSITY OF PORTSMOUTH

Spring Semester – First Coursework

Academic Session: 2010–2011

Instructions

a) **Deadline: Friday, March 23, at 3PM on Victory**

b) Compress all source code and result files into a zip file and upload it to the Victory assignment; **no printed copies are needed**.

c) **Make sure to label all materials with your reference number**.

d) This is the **first coursework (out of two), worth** 40% of the unit.

e) Complete **all three parts**.

f) Explain your solutions using comments in the code.

g) If you cannot complete a project, upload what you attempted to do.

h) Assignment must be undertaken **alone**.

1 Background information

Let $n \in \mathbb{N}$ and

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \in \mathrm{Mat}_n(\mathbb{R}), \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \in \mathbb{R}^n, \tag{1}$$

be a square $n \times n$ matrix and a column n-vector with real entries, respectively. **Gaussian elimination** is aimed at solving the system $A\boldsymbol{x} = \boldsymbol{b}$ for a vector of n unknowns $\boldsymbol{x}$, that is,

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\ & \vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n &= b_n \end{cases} \tag{2}$$

We start by portraying the system with an augmented matrix $(A \mid \boldsymbol{b})$, written in the form

$$\left(\begin{array}{cccc|c} a_{1,1} & a_{1,2} & \dots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & +\dots & a_{n,n} & b_n \end{array}\right) \tag{3}$$

The process in which to solve system (2) or, alternatively, (3), comprises two steps:

a) reducing (3) to a triangular system $\left(A^{(n)} \mid \boldsymbol{b}^{(n)}\right)$ of the sort

$$\left(\begin{array}{ccccc|c} a_{1,1}^{(1)} & a_{1,2}^{(1)} & a_{1,3}^{(1)} & \dots & a_{1,n}^{(1)} & b_1^{(1)} \\ 0 & a_{2,2}^{(2)} & a_{2,3}^{(2)} & \dots & a_{2,n}^{(2)} & b_2^{(2)} \\ 0 & 0 & a_{3,3}^{(3)} & \dots & a_{3,n}^{(3)} & b_3^{(3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a_{n,n}^{(n)} & b_n^{(n)} \end{array}\right) \tag{4}$$

having zeroes under the diagonal. This is the process of Gaussian elimination proper. It starts with $\left(A^{(1)} \mid \boldsymbol{b}^{(1)}\right) = (A \mid \boldsymbol{b})$ and, for every $k = 1, \ldots, n-1$,

$$i = k+1, \ldots, n : \quad \left\{ \begin{array}{rcll} b_i^{(k+1)} &=& b_i^{(k)} - \frac{a_{i,k}^{(k)}}{a_{k,k}^{(k)}} b_k^{(k)}, & \\ a_{i,j}^{(k+1)} &=& a_{i,j}^{(k)} - \frac{a_{i,k}^{(k)}}{a_{k,k}^{(k)}} a_{k,j}^{(k)}, & j = 1, \ldots, n \end{array} \right\} \tag{5}$$

$a_{k,k}^{(k)}$ is usually called the **pivot** at each step. At the end of the process, and except special cases mentioned below, A should have been reduced to an upper triangular form $U = A^{(n)}$ as in (4).

b) Once A has been transformed into the upper triangular matrix U,

$$U = A^{(n)} = \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{n,n} \end{pmatrix}, \tag{6}$$

all it takes to complete the process is solving system $U\boldsymbol{x} = \boldsymbol{b}^{(n)}$. This is sometimes called **backward substitution**: it starts with $x_n = \frac{b_n^{(n)}}{u_{n,n}}$ and proceeds as follows:

$$x_i = \frac{1}{u_{i,i}} \left(b_i^{(i-1)} - \sum_{j=i+1}^{n} u_{i,j} x_j \right), \qquad i = n-1, n-2, \ldots, 1. \tag{7}$$

Remarks 1.

1. There will be matrices for which straight Gaussian elimination will not be possible at some step, unless an additional process of row exchange called **pivoting** is used. For instance,
$$A = \begin{pmatrix} 1 & 2 & 4 & 8 \\ 0 & 0 & -3 & 1 \\ 0 & 1 & 2 & -1 \\ 0 & 0 & 0 & 15 \end{pmatrix}$$
will cause a division by zero, unless rows 2 and 3 are swapped before Gaussian elimination; same applies at some step of the elimination process to
$$B = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 5 & 7 & 13 \\ 7 & 6 & 1 & 4 \end{pmatrix};$$
check this yourselves by hand.

2. Row pivoting may not be *necessary*, but it will still be *useful*, if done properly (see Parts 2 and 3) in order to reduce the overall numerical error. The process in this case consists of choosing the element $a_{i,k}^{(k)}$ below (or equal to) the pivot with the largest absolute value, swapping rows i and k and then performing step k of the elimination.

3. A further reason why system solution might not be possible using this method, is that the matrix be not invertible, meaning $\det A = 0$.

Assume we wish to invert A. More generally, assume we want to solve $AX = B$ where, as opposed to the previous system, the unknown and the right-hand terms are now matrices instead of vectors. This implies solving n linear systems instead of one:
$$A\boldsymbol{x}_1 = \boldsymbol{b}_1, \quad A\boldsymbol{x}_2 = \boldsymbol{b}_2, \quad \dots \quad A\boldsymbol{x}_n = \boldsymbol{b}_n,$$
where $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$ (resp. $\boldsymbol{b}_1, \dots, \boldsymbol{b}_n$) are the column vectors of X (resp. B). Gaussian elimination yields an augmented $n \times 2n$ matrix
$$\left(A^{(n)} \mid B^{(n)}\right) = \left(A^{(n)} \mid \boldsymbol{b}_1^{(n)}\, \boldsymbol{b}_2^{(n)} \cdots \boldsymbol{b}_n^{(n)}\right), \tag{8}$$
and backward substitution is then performed n times in order to obtain the solution columns $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$:
$$A^{(n)}\boldsymbol{x}_1 = \boldsymbol{b}_1^{(n)}, \quad A^{(n)}\boldsymbol{x}_2 = \boldsymbol{b}_2^{(n)}, \quad \dots \quad A^{(n)}\boldsymbol{x}_n = \boldsymbol{b}_n^{(n)}. \tag{9}$$
If $B = \mathrm{Id}_n$, X is nothing but the inverse matrix.

Let us now describe a more useful procedure, especially in the solution of multiple systems with the same matrix. An *LU* **decomposition** of A is any expression of the matrix as a product of a lower triangular matrix L (having 1's in its diagonal) and an upper triangular matrix U:
$$A = LU = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{2,1} & 1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 \end{pmatrix} \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{pmatrix} \tag{10}$$
Thanks to the following Theorem and its Corollary, Gaussian elimination can be used in this setting.

Theorem 2. *If A admits Gauss* ***without a need for pivoting*** *(i.e. no pivot is equal to zero at any stage of the Gaussian elimination process), then there exists a unique LU decomposition (10) of A. Moreover, the entries of L below the diagonal are precisely the coefficients arising from Gaussian elimination, and the matrix U is exactly the resulting matrix (6) from the elimination process:*
$$l_{i,k} = \frac{a_{i,k}^{(k)}}{a_{k,k}^{(k)}}, \quad u_{i,j} = a_{i,j}^{(i)}.$$

Corollary 3. *Given* ***any invertible matrix*** *A, there exists a set of three matrices:*

a) permutation matrix P (e.g. exactly one 1 entry in each row and column and 0's elsewhere),

b) lower-triangular L with 1's in its diagonal, and

c) upper-triangular U,

such that $PA = LU$. This is called the LUP ***decomposition*** *of A.*

Assume we need to solve system $A\boldsymbol{x} = \boldsymbol{b}$. This implies $LU\boldsymbol{x} = P\boldsymbol{b}$, hence here is a proposed new method based on matrix decompositions:

I We first find the LUP decomposition $PA = LU$ for A.

II Secondly, we solve $L\boldsymbol{y} = \boldsymbol{v} = P\boldsymbol{b}$ for $\boldsymbol{y}$ using the so-called **forward substitution**:

$$y_1 = v_1, \qquad y_i = v_i - \sum_{j=1}^{i-1} l_{i,j} y_j, \quad i = 2, \dots, n. \tag{11}$$

III Thirdly, we solve $U\boldsymbol{x} = \boldsymbol{y}$ for $\boldsymbol{x}$.

An advantage of matrix decompositions is that they make it easy to solve multiple systems having the same matrix. Indeed, to solve the matrix system $AX = B$, we only need to perform Gaussian elimination to the matrix (to obtain L and U), not to the columns $\boldsymbol{b}_1, \dots, \boldsymbol{b}_n$ of B:

$$LU\boldsymbol{x}_1 = P\boldsymbol{b}_1, \quad LU\boldsymbol{x}_2 = P\boldsymbol{b}_2, \quad \dots \quad LU\boldsymbol{x}_n = P\boldsymbol{b}_n. \tag{12}$$

Hence, all we need to do is perform forward-backward substitution on every system.

2 Assignment

Part 1. (35 marks) You will solve linear systems using Gaussian elimination without pivoting. Your project should contain, at least, the following:

a) a function `gauss` having at least three inputs: A, $\boldsymbol{b}$, and an output file, and transforming the first two inputs into $A^{(n)}$ and $\boldsymbol{b}^{(n)}$, or else returning an error message if straight Gaussian elimination was not possible (see Comments). For each step $k \to k+1$ of the elimination process, $\left(A^{(k+1)} \mid \boldsymbol{b}^{(k+1)}\right)$ (or an exit message if the step was unfeasible) will be written down on the output file in a well-structured manner.

b) a function `backward` having at least three inputs: an upper triangular matrix U, a vector $\boldsymbol{v}$, and the same output `ofstream` file as `gauss`, and returning the solution vector $\boldsymbol{x}$ to $U\boldsymbol{x} = \boldsymbol{v}$.

c) a function named `norm` computing the vector norm of a given vector $\boldsymbol{v} \in \mathbb{R}^n$. You can use any of the common (and equivalent) Hölder norms available on $\mathbb{R}^n$, e.g.

$$\|\boldsymbol{v}\|_2 := \sqrt{v_1^2 + \cdots + v_n^2}, \qquad \|\boldsymbol{v}\|_1 := \sum_{i=1}^{n} |v_i|, \qquad \|\boldsymbol{v}\|_\infty := \max_{1 \le i \le n} |v_i|.$$

d) a function `check`, having inputs $A, \boldsymbol{x}, \boldsymbol{b}, \mathrm{tol}$, which will compute the norm of the difference vector, $\|A\boldsymbol{x} - \boldsymbol{b}\|$. If such norm is smaller than tol, we will accept the solution as valid; otherwise, chances are there is a mistake somewhere in your project.

e) a subroutine `matrix_invert` designed to invert matrices. You can adapt your previous function `gauss` into a new one called `matrixgauss` for (8), so as to replace the entries of $\boldsymbol{b}^{(k)}$ in (5) by the entries of a matrix $B^{(k)}$. (9) will then be performed within `matrix_invert`.

f) a function named `matrixcheck` checking the validity of `matrix_invert` on $AX - \mathrm{Id}_n$, namely that $\|AX - \mathrm{Id}_n\| < \mathrm{tol}$ for an adequate matrix norm $\|\cdot\|$, e.g. the Frobenius, sub-1 or sub-∞ norm:

$$\|B\|_F := \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |b_{i,j}|^2}, \qquad \|B\|_1 := \max_{1\le j\le n} \sum_{i=1}^{n} |b_{i,j}|, \qquad \|B\|_\infty := \max_{1\le i\le n} \sum_{j=1}^{n} |b_{i,j}|.$$

A function named `matrixnorm` could take care of this.

Comments:

a) You must include sample result files for particular matrices and vectors A, $\boldsymbol{b}$, *along with the checked residues* $\|A\boldsymbol{x} - \boldsymbol{b}\|$ *and* $\|AX - \mathrm{Id}_n\|$.

b) Your project must comprise separate header (`*.h`) and `.cpp` files.

c) Include your results for sample matrices in which Gaussian elimination is stopped because pivoting was *necessary* at some stage.

d) A further reason why system solution might not be possible using your method, is that the matrix be not invertible, meaning $\det A = 0$. Your project will still return an error message such as the one in the previous item – namely, that for some step k there is a pivot $a^{(k)}_{k,k}$ having an absolute value below tol, hence making the division in (5) numerically inadequate. However, there is still the question of whether the matrix chosen failed the process because it was non-invertible, or simply out of a poor choice of row order (such as in examples A and B in the previous item). Make sure to find examples of both cases. Include said examples among the result files.

e) Make sure to pass your output file by reference as an argument of your function; never declare output files globally.

f) Needless to say, everything described above has been written for entries ranging from 1 to n but this might not be possible or practical in your project; adapt the above computations accordingly.

g) Feel free to use STL `vector` containers instead of arrays if you wish.

Part 2. (50 marks) Let us now add pivoting to the picture. *Naturally, the next step would be slightly modifying the* `gauss` *and* `matrixgauss` *routines by adding the process of pivoting into their main loops. However, we are going to bypass this step and go directly to* LUP *decomposition.*

As advanced in Remark 1, *you will perform row pivoting (namely, through the use of* P*) even if it is not extremely necessary.* Your project should contain, at least, the following:

a) a function delivering the LUP decomposition of a given matrix

```
int LUP ( double **a, int n, int *index, ...  );
```

having at least three inputs: A, n, and an index vector keeping track of what rows or columns you need to exchange in L and U (summarising P or, in other words, the process of pivoting). Pivoting itself consists of the act of checking, for each value $k = 1, \dots, n-1$, which is the entry in the column below and including pivot $a^{(k)}_{k,k}$ with the largest absolute value. If it turns out not to be the pivot, but another element below it, say $a^{(k)}_{\tilde{k},k}$, you need to swap rows $\tilde{k}$ and k, amend vector `index` appropriately, and perform step k of the Gaussian elimination. Hence, using `index` in clever ways will spare you the effort of declaring a whole matrix `P`.

Upon exiting the function, matrix `a` should contain the entries of L under the diagonal, and the entries of U elsewhere. *If the matrix is not invertible, your function should be able to detect this fact and report it to the function calling it.*

b) the same function `backward` as in Part 1, or a similar one called `LUPbackward`.

c) a function `LUPforward ( double **L, double *v, int n, ... );` carrying out forward substitution (11) on *any* lower-triangular system $L\boldsymbol{y} = \boldsymbol{v}$ with diagonal 1's.

d) a subroutine `LUPmatrix_invert` designed to **invert matrices using** LUP **decomposition**, as described in (12) above.

e) functions `norm`, `check`, `matrixnorm` and `matrixcheck` as in Part 1.

Comments:

a) Use separate header and `.cpp` files.

b) It is strongly advised to maintain an output file as in Part 1 and write everything down on it.

c) Make sure you try your program with different systems $A\boldsymbol{x} = \boldsymbol{b}$ and different matrix inverses A^{-1} and include the result files in the assignment dropbox along with the rest of the files.

Part 3. (15 marks) Let us compare the routines `matrix_invert` (Gaussian elimination without pivoting) and `LUPmatrix_invert` (Gaussian elimination with row pivoting) in the process of inverting a given matrix. For increasing values of the number of rows and columns n of matrix A_n, you must compute inverses X_1 and X_2 (using `matrix_invert` and `LUPmatrix_invert` respectively) and fill in a file with three columns:

n	$\|A_n X_1 - \mathrm{Id}_n\|$	$\|A_n X_2 - \mathrm{Id}_n\|$
$\cdots$	$\cdots$	$\cdots$
$\cdots$	$\cdots$	$\cdots$

(13)

At the end of each file, write a comment on your interpretation of the amounts shown. Concerning the choice of matrix A_n, you may use matrices having *pseudo-random* entries. You are advised to create a pseudo-random matrix $A_{n_{\max}}$ with large-enough dimension (say, $n_{\max} = 20, 30, 50\ldots$) and create a table such as (13) for the submatrices A_n given by the first n rows and columns of $A_{n_{\max}}$, for $n = 2, \ldots, n_{\max}$.

Alternatively, you may create a whole brand-new pseudo-random matrix A_n for every value of n.

Comments:

a) You may want to create a specific function of the sorts of `double **random ( int n_max );` to generate $A_{n_{\max}}$.

b) Since the matrix is pseudo-random, the question remains on whether $A_{n_{\max}}$ is amenable to Gaussian elimination without pivoting. Think of ways of modifying $A_{n_{\max}}$ accordingly before creating the table.

c) **Repeat the process for different collections of pseudo-random matrices** (plainly put: create more than one table (13)).