Preliminary Examination
Department of Scientific Computing

Linear Algebra

1.

a) Suppose that the real n by n matrix A is *strictly* upper triangular, that is, the lower triangle and the diagonal are zero. The eigenvalues of an upper triangular matrix appear on the diagonal, so we know that the only eigenvalue for A is zero. Every matrix has at least one (nonzero!) eigenvector. What is one eigenvector of A?

The only eigenvector for the matrix A is $(1, 0, 0, 0, ..., 0)$. This is because row $n-1$ of the eigenvalue equation $A*x = 0*x$ actually reads $A_{n-1,n} * x_n = 0$, meaning that x_n must be zero. Therefore, row $n-2$ will imply that x_{n-1} is 0, and as we go backwards, we end up with row 1, which will imply that x_2 must be zero. This leaves x_1 free, and we may take it to be 1 (or any nonzero value.)

Simply stating the correct eigenvector is acceptable, though deprecated.

b) Suppose that B is an n by n matrix, and that we have computed the matrix X whose columns are eigenvectors of B, and the diagonal matrix Λ, whose diagonal entries are the corresponding eigenvalues. Write the matrix equation which relates B, X and Λ. If we wish to compute the value of B^{100}, one way is simply to carry out 99 matrix multiplications. However, suppose that the eigenvector matrix X is an orthogonal matrix, and describe a formula for computing B^{100} that is much cheaper.

The matrix equation is $B * X = X * \Lambda$.

Let us get an expression for Z by post-multiplying by X' (which is the same as X^{-1} since X is orthogonal): $B = X * \Lambda * X'$. Notice that, if we wish to square B, we could instead write: $B^2 = X * \Lambda * X' * X * \Lambda * X'$, which looks more complicated, until we realize that $X' * X = I$ meaning we actually have $B^2 = X * \Lambda^2 * X'$. This trick works again if we compute B^3, and it's easy to see that $B^{100} = X * \Lambda^{100} * X'$. Since Λ is a diagonal matrix, computing Λ^{100} is trivial, and we end up computing B^{100} with really only one or two matrix multiplications.

c) Suppose that C is an n by n upper triangular matrix of 1's, and consider the problem of determining a solution of the linear system $C*x = b$ for the unknown vector x, given the vector b. Write down the sequence of (scalar) equations that you would use to determine each entry of x, taking into acount the special form of this matrix.

The first equation is $x_1 + x_2 + ... + x_n = b_1$, and the very last equation is $x_n = b_n$. To solve the system, we simply work backwards: $x_n = b_n$, then $x_{n-1} = b_{n-1} - x_n$, ..., all the way to $x_1 = b_1 - x_2 - x_3 - ... - x_n$.

Parallel Programming - MPI

2. A hole has been drilled into the earth to an underground reservoir of oil. A pipeline has been created, consisting of 100 consecutive stations, with station 1 down there in the reservoir, and station 100 up at the surface. There are 25 million barrels of oil in the reservoir, and it is desired to bring at least 24 million barrels of oil to the surface.

The oil is moved by pumps from one station to the next higher one. Each pump has the property that, over the course of a day, it can move 15% of the oil at that station to the next higher station. There is a pump at every station except the last, where the oil is collected in a gigantic tub.

We measure oil in millions of barrels, so $U(1)$ is initially 25, and we want to pump until $U(100)$ is at least 24. A simplified model allows us to start from a list $U()$ of the amount of oil at each station at the beginning of a day, and determine $V()$, the amount at the end of the day:

$$\begin{aligned} V(1) &= 0.85 * U(1) \\ V(2) &= 0.15 * U(1) + 0.85 * U(2) \\ &... \\ V(99) &= 0.15 * U(98) + 0.85 * U(99) \\ V(100) &= 0.15 * U(99) + 1.00 * U(100) \end{aligned}$$

(Note that U(100) has a coefficient of 1 in the last equation!)

a) Write a program that will:

1. initialize the first entry of U to be 25, the remaining entries zero;
2. compute V, the updated value of U for the end of the day;
3. if V(100) is at least 24, stop, printing the number of days;

4. Otherwise, set U = V and return to step 2.

Your code may be in pseudocode, or a standard computer language, and you do not need to run it.

```
u(1) = 25;
u(2:100) = 0;

while ( u(100) < 24 )

  v(1:99) = 0.85 * u(1:99)
  v(100)  =        u(100)
  v(2:100) = v(2:100) + 0.15 * u(1:99)

  u(1:100) = v(1:100)
```

b) Rewrite your program from part a) to use MPI. Assume that you only have two MPI processes available. Write your program in such a way that you can hope it will run about twice as fast as your original program. Your program should also measure and report the program execution time. Again, you do not need to run the program.

You are free to present your program as a complete or partial program in a standard language, or in pseudocode. However, your answer must specify all important MPI issues, including initialization and shutdown. For instance, instead of a full call to **MPI_Send()**, it would be clear enough to say *"Use MPI_Send() to send the value of* **i** *to process* **0***"*.

If we split the 100 vector entries into two sets of 50, then we only have two communication problems. On each step, 15% of U(50) should get transferred to U(51). But now U(51) is sitting on process 1, and is called U(1). So on each step, we have to send the current value of U(50) from process 0 to process 1. Moreover, for the stopping test, process 1 will know that U(100) (now renamed U(50)) has hit 24, because it computes it, but this information must be sent to process 0, so it will also know when it is time to stop.

```
if ( id == 0 )
  u(1) = 25;
  u(2:50) = 0;
else
```

```
  u(1:50) = 0;

while ( true )

  if ( id == 0 )

    set temp1 = u(50)
    use MPI_Send to send temp1 to process 1.

    v(1:50) = 0.85 * u(1:50);
    v(2:50) = v(2:50) + 0.15 * u(1:49);

    u(1:50) = v(1:50);

    use MPI_Receive to receive temp2 from process 1.
    if 24 <= temp2 ) then exit

  else if ( id == 1 )

    use MPI_Receive to receive temp1 from process 0;

    v(1:49) = 0.85 * u(1:49);
    v(50) = u(50);

    v(1) = v(1) + 0.25 * temp1;
    v(2:50) = v(2:50) + 0.15 * u(1:49);

    u(1:50) = v(1:50);

    set temp2 = u(50);
    use MPI_Send to send temp2 to process 0.
    if ( 24 <= temp2 ) then exit
```

Parallel Programming - OpenMP

3. Subdivide the unit square into 4 equal squares of side $\frac{1}{2}$. Consider arrays $x()$ and $y()$ which contain the coordinates of n=10,000 points. We wish to count the number of points that lie within each of the 4 squares.

That is, we want an array $c()$ of length 4, so that the first entry counts the number of points that fall within the first square, and so on. We might expect that the sum of the entries in c is exactly 10,000, but it might actually be less (or possibly even slightly more!).

a) Write a pseudocode program that determines and prints c. Your pseudocode should be written in such a way that an intelligent programmer could use it as a blueprint.

b) Write a partial program, in C or Fortran, that carries out the same computation as in part a), but does so in parallel, using OpenMP with 4 threads. Include statements that would allow your program to report the execution time for the calculation. You do not have to declare your variables, or worry about the details of print statements or where the x and y arrays come from. The important features of your answer will be the sensible use and ordering of loops, and the correct use and placement of OpenMP directives.

c) Now suppose that you have subdivided the unit square into 100x100 = 10,000 equal subsquares, and you have the coordinates of 7 points. Suppose we wish to create an array $d()$ of length 7, so that each entry of $d()$ lists the index of the subsquare containing the corresponding point. Write a partial program in C or Fortran that carries out this task, using OpenMP with 100 threads.