Computer Assignment 2
Due Friday February 19th in lecture
You may work in groups of size 1-4 (one submission with 1-4 names on top)

In case you forgot, MatLab is available on computers in KAP 262 (the
computer lab of the Math Department). I believe it is also available at other
computer labs on the campus. If you prefer (many of you do), you can down-
load it on your own computer as well. See https://itservices.usc.edu/matlab/
for instructions on how to install the software on your own machine.

In the first assignment, we focused on using commands in the command
window. In this assignment, we will learn how to write our own functions
that we can then call from the command window.

Part 1.
To write a new function in MatLab, move your mouse to the top left
corner and then click on new → function. MatLab sets up the basic format
of where to put the inputs and outputs for our function (it also has a place
to put in comments for your code; anything after a percentage sign is ignored
when running the function). Suppose we wanted to write a function that
computes the trace of the matrix (MatLab already has a command that does
this, namely trace(A), so we are just doing this as a tutorial). We would
want to take a matrix A as an input and a scalar s as an output. We would
want to then sum all the diagonal entries.

One way we could this is to use a for loop, going from 1 to n, and ex-
tracting the corresponding diagonal entry on each iteration. The format for
a for loop is of the form for variable=first:increment:last.

So, for instance, for k=3:4:19 would loop through the values k = 3, 7, 11, 15, 19.
It is ok if the last element is not hit by the increment. For instance, for
k=3:4:20 would give the same values. If the increment is 1, then it can
be omitted. For instance, for k=5:9 would loop through the values of
k = 5, 6, 7, 8, 9. A for loop will perform all of the commands after it for
each value until it reaches an “end” statement.

For our trace program, the code might look something like:

- function s = mytrace(A)
- [m, n]=size(A); % This gives the dimensions of A, and stores the
  number of rows to m and the number of columns to n.
- s=0; % This initializes the sum to 0.
- for j=1:n
  - s=s+A(j,j);
end %This tells the program the for loop ends here.
- end %This tells the program the function ends here.

Now we can call this in the command window. For instance, try typ-
ing mytrace([1 2 3; 4 5 6; 7 8 9]) . It should return 15 as the an-
swer. If you wanted to store your answer as a variable, you could do
b=mytrace([1 2 3; 4 5 6; 7 8 9]) to store the value to the variable b. Adding a semicolon will suppress output but still perform the computation.

There is nothing to hand in for this part.

**Part 2.** Right now in our trace program we are assuming that the input matrix is a square. If we wanted to be safe, we could add a check of this to our code. To do this, we will use an if-else statement. “If” statements will only be performed if the condition immediately after it is satisfied; if it is, then it will perform everything until either an end or an else. Our code might look something like this.

- function s = mytrace(A)
- [m, n]=size(A);
- s=0;
- if m==n % this checks to see if the number of columns and rows match
- for j=1:n
- s=s+A(j,j);
- end
- else % This is what happens if they don’t match
- disp('Error: The matrix is not square!');
- end

Try the code and see what happens for square and nonsquare matrices. If you’re curious, you can type `edit trace` in the command window to see how the built in function was coded.

TO HAND IN: Write a function (call it whatever you’d like) that takes in an arbitrary rectangular matrix and computes the sum of squares of every entry in an odd row (rows 1, 3, 5, 7, ...). You may want to use a for loop within another for loop (one loop for the columns and the other for the rows); be sure to use a different variable for each loop. To check, the matrix

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]

should return a value of \(1^2 + 2^2 + 3^2 + 7^2 + 8^2 + 9^2 = 208\) in your function. Then try your code on a 15 × 15 magic square and write your output on your submission.

**Part 3.**

We’ve spent a couple lectures discussing the LU factorization. This can be a tedious computation, but with the help of a computer program we can simplify this process (in general, computers are great whenever we have a systematic algorithm to follow). To do this, we will need to be able to add a scalar multiple of one row to another in a matrix. The command

\[
A(j,:) = A(j,:) + m \times A(k,:)
\]
will add $m$ times row $k$ to row $j$ (the : symbol tells MatLab to consider all columns in that row).

TO HAND IN: Write a function function $[L,U] = \text{myLUfactor}(A)$ that computes the $LU$ factorization of a matrix. It should output the two matrices $L$ and $U$. A suggestion is to start with $L$ initialized to the identity matrix, and go column by column adding in the multipliers below the diagonal. Each time you add a multiplier, you can row adjust $A$ accordingly. At the end, you can just set $A$ to $U$. Try your function on a few examples from the book to verify it is working properly. Then compute the LU factorization of an $6 \times 6$ magic square and write (or print) the two matrices on your submission. Feel free to round decimal entries to two decimal places if you are handwriting.

Remark: For both part 1 and 2, you may wish to attach a printout of your code for partial credit if it is not working correctly, but this is not required. Just be aware that without a code printout, extra credit cannot be awarded.