HOMEWORK 1

Due: Friday, July 3rd 2020, 11:59 PM

Formatting Reminder: The submitted work **MUST** follow the naming convention listed below.

- "**LastName_UID_HW_01_report.pdf**"
- "**LastName_UID_HW_01_code.zip**"
- "**LastName_UID_HW_01_main.m**"

Submit **two** separate files to the CCLE course website: (1) a pdf of your written report and (2) a .zip file containing all of the `MATLAB` files written for the assignment.

Use a switch statement to call for which problem to solve in your main script, ***DO NOT*** submit two separate m-files for two problems. Remember to use good coding practices by keeping your code organized, choosing suitable variable names, and commenting where applicable. Any of your `MATLAB` .m files should contain a few comment lines at the top to provide the name of the script, a brief description of the function of the script, and your name and UID. Any submission that does not follow the above-mentioned format will receive SEVERE point deductions!

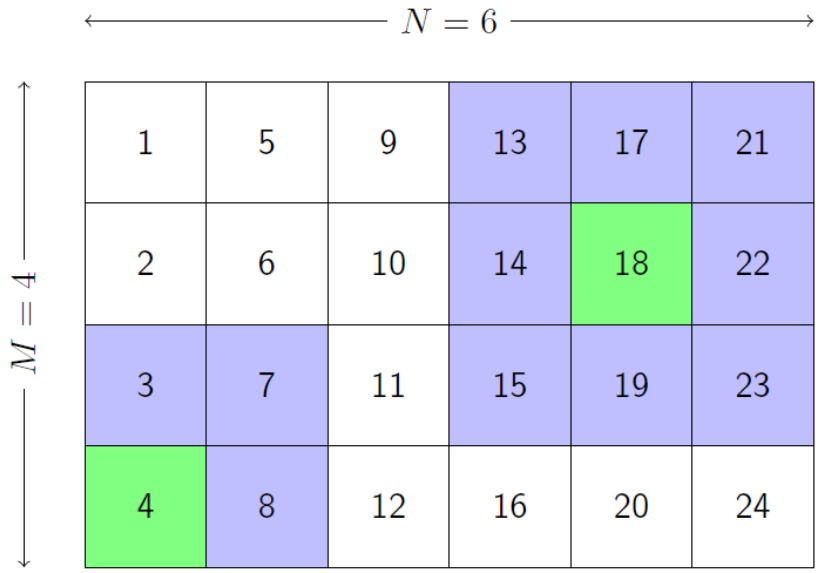
1. **Oblate spheroid calculations.**
   An *oblate spheroid* such as the Earth is obtained by revolving an ellipse about its minor axis. In everyday terms, it is the shape of a slightly compressed beach ball. The Earth's equatorial radius is about 20 km longer than its polar radius. The surface area of an oblate spheroid is given by:

$$A(r_1, r_2) = 2\pi \left(r_1^2 + \frac{r_2^2}{\sin(\gamma)} \ln \left(\frac{\cos(\gamma)}{1 - \sin(\gamma)} \right) \right) \tag{1}$$

   where r_1 is the equatorial radius, r_2 is the polar radius, and $\gamma = \arccos\left(\frac{r_2}{r_1}\right)$.

   Here we must enforce $r_2 < r_1$. Write a script that inputs the equatorial and polar radii and display both $A(r_1, r_2)$ and the approximation $4\pi\left((r_1 + r_2)/2\right)^2$. Apply the script to the Earth data (r_1, r_2) = (6378.137, 6356.752). Display 10 digits per answer to reveal the discrepancy.

2. **Neighbor Identification.**
Many engineering problems can be solved numerically by dividing a large, complicated geometry into a multitude of smaller easier-to-solve cells. The quantities represented in an individual cell (for example, temperature, velocity, and/or pressure) depend *only* on the values of those quantities stored at the cell's nearest neighbors. In this problem, we will write a script to identify all the neighbors of a given cell in a rectangular array. Consider the numbered setup shown below.



The neighbors of cells 4 and 18 identified on a linearly-indexed grid with M = 4 and N = 6. Two different sets of neighbors have been identified in the figure above: (1) cell `4` has neighbors `3`, `7`, `8`, and (2) cell `18` has neighbors `13`, `14`, `15`, `17`, `19`, `21`, `22`, and `23`. This configuration of cell s uses the concept of *linear indexing*, in which a single number (as opposed to two, e.g. (x, y)) is used to represent a cell's location in a 2D grid. In this problem, we will assume that cell numbering *always* starts in the upper-left corner, proceeds down the first column, then down the second column, etc., as shown.

(a) Begin by asking the user to input three separate values representing the number of rows in the array, M, the number of columns, N, and the cell for which we will be identifying neighbors, P. Your code must produce an error if either M or N is less than 2 or a non-integer value. Similarly your code must produce an error if the value of P is a non-integer or does not fall in the range of valid cell number, $1 \leq P \leq (M \times N)$.

(b) Once the values of M, N, and P have been collected and validated, we can begin identifying all the neighbors of P. You may find it helpful to write down in your pseudo-code the numerical rules and patterns that let us classify a cell, P, as being located on the walls (left, right, top, bottom), or corners (top left, top right, bottom left, bottom right). Note that the maximum number of neighbors is

8 (4 orthogonal + 4 diagonal), while the ones located on the walls each has 5 neighbors, and the ones located in the corners each has 3 neighbors.

(c) Print the neighbors of the user-input cell ID, P, to the command window. Use the formate shown below.

```
Cell ID:    4
Neighbors:  3  7  8
```

The neighbors needs to be in ascending order. Keep your logic as concise and organized as possible.

(d) Can a linear indexing scheme be used for a 3D grid? If so, explain the numbering convention you would employ, using examples to explain your convention if necessary. In 2D, we can classify every cell as either: (1) corner, (2) edge, or (3) interior. Extend this classification system for a 3D grid and define how many neighbors surround each of these different cell types.

Note: Your code should work for *any* valid values of M and N, do not rely on hard-coded logical test that presuppose a particular value for M and N, e.g. `if P == 24` to catch the bottom-right corner in the grid above. Keep everything defined in terms of M and N, and/or other derived quantities.