HOMEWORK 2

Due: Tuesday, April 17, 2018, 11:55 pm

0. **Reading Assignment.**
   Read section 1.2 of ITC

1. **The Lunar Phase Calculator.**
   In this problem, you will construct a script that returns the lunar phase for a user-specified date. We can quickly estimate the lunar phase by knowing two things: (i) the number of days between the user-input date and a known new-moon date (in this case, we'll use the new moon that occurred on January 1, 1900) and (ii) the number of days in each lunar revolution. Your program should:

   (a) Take 3 separate command window inputs for a given date in the following format: `MMM DD YYYY` (you can assume and enforce that the month input must be capitalized). Ensure that your code requests the inputs in the order shown below and that the inputs are all taken as a string, which maximizes our opportunities for error checking (i.e., `input('...', 's')`).

```
Please enter the month as MMM (e.g. JAN):
Please enter the day as DD (e.g. 01):
Please enter the year as YYYY (e.g. 2000):
```

   (b) Check the validity of the input and return an error if necessary. If the user inputs a non-numeric value or doesn't follow the formatting specification (e.g., `YYYY` as `19a6` or `DD` as `7` vs. `07`) or inputs a date that doesn't exist (e.g., `JAN 32 2000`, `FEB 29 2014`, or `MAR .4 -198`), return an error message to the command window *using the MATLAB built-in* `error` *function*. Design your program to work for year values from 1900 to 9999. Be sure to take leap years into account when performing your checks (e.g., `FEB 29 2017` should produce an error, but `FEB 29 2016` should not).

   (c) Calculate the elapsed number of days since the new moon. To easily calculate the elapsed number of days since January 1, 1900, we'll first convert the user-input date into its corresponding Julian Day Number, J, an astronomical quantity indicating a continuous integer count of days past 4713 B.C.

$$J = \text{day} + \left\lfloor \frac{153m+2}{5} \right\rfloor + 365y + \left\lfloor \frac{y}{4} \right\rfloor - \left\lfloor \frac{y}{100} \right\rfloor + \left\lfloor \frac{y}{400} \right\rfloor - 32045$$

   where

$$y = \text{year} - a + 4800$$
$$m = \text{month} + 12a - 3$$

   and the offset value, a, is set to 1 for January and February, and 0 for all other months. Once the Julian Day Number of the input date is calculated, the elapsed number of days can be calculated with a simple subtraction, $\Delta J = J_{input} - J_0$. For reference, January 1, 1900 has a Julian Day Number $J_0 = 2415021$.

   Note: The $\lfloor ... \rfloor$ brackets indicate rounding a value "down" to the nearest integer. For example, $\lfloor 5.75 \rfloor = 5$.

(d) Calculate the lunar phase. With the elapsed number of days, we can determine the percent illumination of the moon, L.

$$L = \sin\left(\pi\frac{(\Delta J \mod T)}{T}\right)^2$$

where $0 \leq L \leq 1$ (Note: $L = 1$ indicates a full moon, i.e., 100 percent illumination, while $L = 0$ indicates a new moon, i.e., 0 percent illumination) and T is the number of days in each lunar revolution, $T = 29.530588853$.

Note: The $\mod$ notation indicates the modulus after division. For example, $10 \mod 4 = 2$.

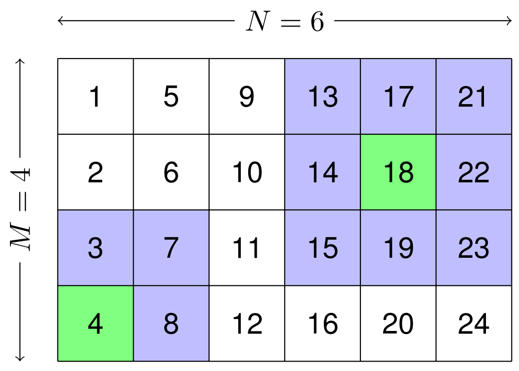
(e) Print the percent illumination to the command window using exactly the formatting shown below (values shown are for illustration purposes only; if correct, your code should match the formatting shown, but *not* the illumination value itself).

```
JAN 15 1900:
Illumination = 95.0 percent
Waxing
```

The third line of output should display "Waxing" (illuminated part of moon getting bigger) if $(\Delta J \mod T)/T < 0.5$ and "Waning" (illuminated part getting smaller) otherwise. Note the scaling on the value of L. Always display `DD` as two characters and `YYYY` as four, including leading zeros if necessary. Using your model, what is the date of the next full moon? What assumptions are we making in this problem, and how could this method be extended and improved in the future?

Note: Although you are not allowed to turn in a solution which uses a built-in MATLAB date function or variable, feel free to use it for debugging purposes while testing your calculations.

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'll 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 above graphic: (1) cell `4` has neighbors `3, 7,` and `8`, and (2) cell `18` has neighbors `13, 14, 15, 17, 19, 21, 22,` and `23`. This configuration of cells 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'll 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'll 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 doesn't fall in the range of valid cell numbers, $1 \leq P \leq M \times N$.

(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 study the lecture slides and write down the numerical rules and patterns that let us classify a cell, P, as being located on the left wall, bottom wall, etc. Note that although the maximum number of neighbors is 8 (4 orthogonal + 4 diagonal) not every cell will have this many; non-corner edge cells will have only 5 neighbors, while corner cells will have only 3.

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

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

The neighbors don't need to be presented in any sorted order; instead, focus on keeping your logic as concise and organized as possible. Keep in mind, there's no requirement that you print the neighbors "all at once". Depending on how you arrange your algorithm, you may find it easier to manage multiple, partial print statements that display one or more neighbors as soon as they're found.

(d) Can a linear indexing scheme be used for a 3D grid ($L \times M \times N$)? 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 your different cell types.

Note: Your code should work for *any* valid values of M and N; don't rely on hard-coded logical tests 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.

3. **Classifying Cubic Functions.**
Complete problem 1.2.11 from ITC. *You do not need to include a write up for Problem 3 in your report!* A few other notes about this problem:

(a) You can assume and code that the inputs will be taken as doubles.

(b) All numeric outputs use the format specifier `%10.6f`.

(c) Print the values of `a, b, c, d` by printing each value on a new line with the variable name and the value itself formatted as `'%10.6f'`.

Example 1 (`a = 1.0, b = -6.0, c = 11.0, d = -6.0`):

```
a =   1.000000
b =  -6.000000
c =  11.000000
d =  -6.000000
```

Print the results to the command window exactly as shown in the examples below.

Example 1 (`a = 1.0, b = -6.0, c = 11.0, d = -6.0`):

```
r1 =   2.577350
q(r1) =  -0.384900
r2 =   1.422650
q(r2) =   0.384900
Function q is simple.
```

Example 2 (`a = 1.0, b = -6.0, c = 11.0, d = 94.0`):

```
r1 =   2.577350
q(r1) =  99.615100
r2 =   1.422650
q(r2) = 100.384900
Function q is NOT simple.
```

Example 3 (`a = 1.0, b = 0.0, c = 1.0, d = 0.0`):

```
Monotone
```

Note: Remember, turn in the code only for Problem 3. Do not use the MATLAB symbolic math toolbox or any other root-finding function (e.g., `fzero`) in your final solution.

Using the naming convention presented in the syllabus, 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. 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.