ENGR 105 – Introduction to Scientific Computing

Assignment 6

Due by 11:59 pm on Weds. 2/23/2022 on Gradescope

Problem 1 (20 pts): Create functions that perform the following operations on arrays without using loops.

i) Create a function `B = flipOddCol(A)` that that takes in any m x n matrix and flips every odd column. For instance, if column 1 of `A` was `[1; 2; 3; 4]`, then column 1 of `B` is `[4; 3; 2; 1]`. Column 2 of `A` would be unaffected.

ii) Create a function `B = diagEye(A)` that takes in any two dimensional matrix of size m x n and creates a new matrix, `B`, with a size min(m,n) x min(m,n) that has diagonal elements equivalent to the sum of the diagonal elements in `A`, but zeros in all other locations. The following demonstrates the functionality.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \quad \rightarrow \quad B = \begin{bmatrix} a_{11}+a_{22} & 0 \\ 0 & a_{11}+a_{22} \end{bmatrix}$$

iii) Create a function `B = colXchange(A)` that takes in any two dimensional matrix `A` of size m x n and outputs `B`, which is the same as `A` except that every pair of columns is swapped. For instance, if presented with a 3 by 5 matrix, columns 1 and 2 would be swapped, as would columns 3 and 4. Column 5 would remain unchanged.

iv) Create a function `B = stripes(n)` that takes in any scalar `n` and creates an *n* x *n* array where the odd rows are all zeros and the even rows are all ones.

v) Create a function `B = threshold(A,lob,hib)` that takes in any N-dimensional array `A` and returns array `B` which is the same as `A` except all entries less than `lob` (lower bound) are set to `lob` and larger than `hib` (higher bound) are set to `hib`. Both `lob` and `hib` are scalars and you may assume that `lob` is less than or equal to `hib`.

Upload all functions. Identify the names of all files associated with this problem in your README.txt.

Problem 2 (40 pts): The Game of Life[1] is a "cellular automaton" created by British mathematician John Conway in 1970. In its treatment here, the game takes in a 2-dimensional matrix that contains only values 1 (index is alive) or 0 (index is dead). The matrix is evolved to its next state of life via a simple set of rules. Similar algorithms have been used to predict, among other things, the geographic affiliation of voters.

The concept is as follows. We start with a matrix filled with 1's (alive) and 0's (dead). Then we evolve the matrix from its current state (`initialState`) to produce the next state (`finalState`) following these simple rules:

- Any live index with fewer than two live neighbors dies, as if caused by under-population.
- Any live index with two or three live neighbors lives on to the next generation.
- Any live index with more than three live neighbors dies, as if by over-population.
- Any dead index with exactly three live neighbors becomes a live cell, as if by reproduction.

Neighbors are considered to be indices adjacent to the index under consideration. That is, any rows and columns +/-1 index from the index being evaluated.

Consider the following 3x3 `initialState` matrix.

1	0	1
0	0	0
0	0	1

Calculating the value of (row, column) = (1, 1) in the next iteration in the Game of Life is shown visually below. The index under consideration (circled) has a value of 1 and its neighbors (shown in bold) of indices (1, 2), (2, 1), and (2, 2) are all zero. That is, the index has no live neighbors and a value of 1. Therefore, its value in the next iteration of the Game of Life is 0.

①	**0**	1
0	**0**	0
0	0	1

Now consider the value of index (2, 2). It's value is 0 (circled) and its neighboring indices are (1, 1), (1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2), and (3,3), which contain a total of three live neighbors. Therefore, the value of index (2, 2) in the next iteration of the Game of Life is 1. Note that the value of index (1, 1) is still 1 even though its value in the next state was determined to be 0. That is, the state of each index in the next state of the Game of Life is based on the values of its neighbors in the current state – in this case, the `initialState`.

[1] See https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life for a complete description and history of the Game of Life.

1	**0**	**1**
0	⓪	**0**
0	**0**	**1**

You have been provided with two pieces of critical code – (1) `gameOfLife_start` and (2) `gameOfLife`. `gameOfLife_start` cleans up the workspace/closes any open figures, loads the initial state of the Game of Life matrix (feel free to change it), defines the number of iterations (feel free to change it), and calls `gameOfLife`. `gameOfLife` initializes a visualization of the incoming matrix, checks all indices of the incoming matrix to determine if the indices should be alive or dead in the next iteration, and updates the visualization for every iteration of the Game of Life.

You must create two functions that interact with `gameOfLife` – (1) `checkNeighbors` and (2) `indexState`.

`checkNeighbors` takes in the initial state of the matrix (a matrix) and the row and column of the index under consideration (both scalars). It reports the number of neighbors that are alive (a scalar), excluding the index under consideration. You will want to ensure that your `checkNeighbors` routine does not attempt to access matrix indices that do not exist (are out of bounds).

`indexState` takes in the value of the index of `initialState` under consideration (a scalar) and the number of live neighbors (a scalar) and decides, based on the Game of Life rule set, if the new state of the index should be alive (1) or dead (0).

The `checkNeighbors` and `indexState` functions you create will likely require use of the `size` command, `for` loops, decisions (`if`, `elseif`), Boolean expressions (`&&` and `||`), and, in some possible instances, the not (`~`) command.

You are also provided with a user interface to interactively design your own Game of Life matrices called `make_GOL_board`. This function can be run like a script (does not have any inputs or outputs), enables point-and-click builds of new Game of Life matrices or editing of existing matrices, and allows you to save the design to a custom filename in your working folder.

Several examples of Game of Life matrices are provided in the associated code folder with this assignment. Descriptions of these examples follow. These descriptions contain references to the Game of Life patterns described in the Wikipedia link above.

Test input `test_board_1.mat`: A 25 x 15 matrix that includes 2 spaceship gliders, 1 beacon, and 1 light-weight spaceship. The game collapses to still life blocks and blinker (period 2) oscillators by the ~90th generation.

Test input `test_board_2.mat`: A 75 x 75 matrix that includes a pulsar (period 3), 2 toads, 2 still life blocks, and a simkin gun that releases a glider every ~120 generations. The gliders that emerge from the simkin gun periodically collide with the pulsar and toads. The simulation has been observed to continuously change even up to the 500th generation.

Test input `test_board_3.mat`: A 30 x 20 matrix that contains a single block-laying switch engine. The game collapses to two blinkers (period 2) by the ~175th generation. A lot of interesting, some oscillating, and some semi-stable patterns emerge before the two blinkers appear.

As a bonus challenge: Feel free to design and upload any interesting Game of Life starting matrices that you wish to devise. Interesting designs that wow the TAs will receive extra credit!

Upload your functions as `checkNeighbors.m` and `indexState.m`. Upload any unique Game of Life boards that you designed. Identify the filenames in your README.txt.

Problem 3 (40 pts): Image kernels are often used to filter scientific images and during general image processing to sharpen images, blur images, perform edge detection, and achieve other similar image modifications.

The process involves convoluting the image data matrix and a smaller kernel matrix. The following describes the general process.

Consider the following 5 x 5 matrix, `M`, where each value represents a pixel color in an image. Larger values represent light gray or white and smaller values represent black or dark gray. The indices of the matrix will be referenced by the row and column index in the form `M(row,column)`. For example, `M(1,1)` would hold the value 101 and `M(3,2)` would hold the value 45.

M =

101	10	232	111	15	56	87
57	255	145	215	25	72	241
2	45	32	198	92	192	21
4	9	56	5	88	67	32
8	67	234	218	221	71	255

A kernel matrix is then convoluted with each image index and its surrounding indices to yield a filtered image. To achieve the convolution, the kernel matrix is centered over the index of interest in the image and the product of the overlapping kernel value and image values are summed to produce the filtered image pixel value. This website also provides a great visual interpretation and interactive interface to learn about kernel-based filters.

Consider the following 3 x 3 kernel, which can be used as a low pass filter.

kernel =

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

The filtered value of `M(3,3)` would be computed as follows.

```
Mfilt(3,3) =  M(2,2) × kernel(1,1) + M(3,2) × kernel(2,1)
            + M(4,2) × kernel(3,1) + M(2,3) × kernel(1,2)
            + M(3,3) × kernel(2,2) + M(4,3) × kernel(3,2)
            + M(2,4) × kernel(1,3) + M(3,4) × kernel(2,3)
            + M(4,4) × kernel(3,3)
```

There will be situations where the kernel falls off the edge of the image matrix. In these cases, only kernel indices overlapping the image matrix need to be convoluted. Consider the filtered value of `M(1,1)` using the same low pass filter as before. The kernel indices `kernel(1,1)`, `kernel(2,1)`, `kernel(3,1)`, `kernel(1,2)`, and `kernel(1,3)` fall outside of the image matrix and should be excluded. The filtered value of `M(1,1)` would then be computed like the following.

```
Mfilt(1,1) = M(1,1) × kernel(2,2) + M(2,1) × kernel(3,2)
           + M(1,2) × kernel(2,3) + M(2,2) × kernel(3,3)
```

Of note, it is important that the kernel always operates on the original matrix indices, not the filtered matrix indices.

Devise a function that computes a filtered image matrix using the following function declaration.

```
function Mfilt = kernelFilter(M,kernel)
```

Input `M` is the original m x n image matrix, input `kernel` is a p x p kernel matrix assumed to have an equal and odd number of rows and columns, and output `Mfilt` is the filtered matrix.

Your function should be programmed to accommodate `M` of any size greater than the size of `kernel` and any square `kernel` with 3 or more rows and columns. Additionally, your function should be home-brew; do not use any built-in convolution functions provided by Mathworks/MATLAB.

Test your function using a script by filtering the provided image `testImage.mat` with the smoothing kernel followed by an outline kernel defined below.

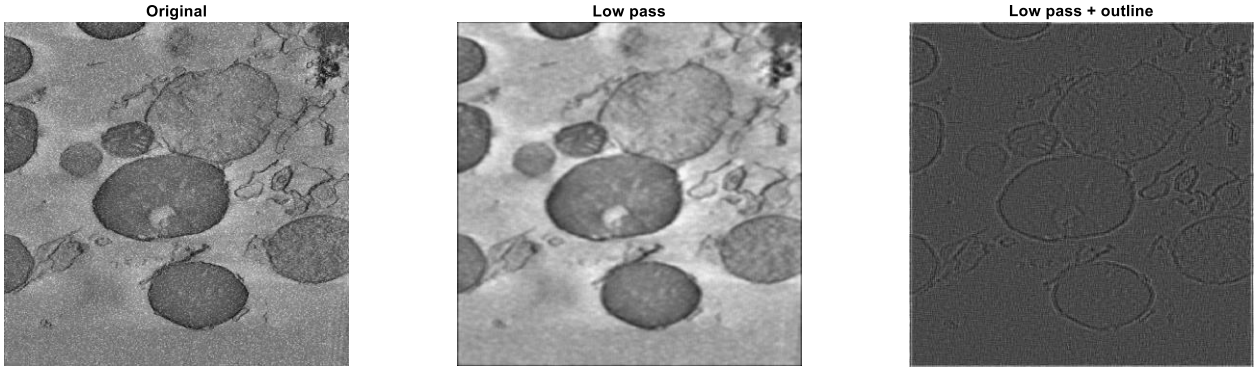
The smoothing (low pass) kernel.

1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25

The outline kernel.

-1	-1	-1
-1	8	-1
-1	-1	-1

Your results should look like the following.



The three plots were produced on separate figures using code like the following. Note that this code just produces the “Original” image.

```
figure
imagesc(testImage)
colormap gray
axis tight square off
title('Original')
```

Upload your function, script, and jpgs of the original image, image subjected to the low pass filter, and image subjected to the low pass filter and outline filter. Identify all filenames in your README.txt.