MICHIGAN ENGINEERING
UNIVERSITY of MICHIGAN ■ COLLEGE of ENGINEERING

MATLAB “write some code” practice

ENGR 101 – Fall 2017

Announcements

1. First C++ lecture on Wednesday

2. Check CANVAS → Google Drive → Resources → Sample_Exams for old MATLAB exams

3. I will stay after lecture for office hours, behind Stamps Auditorium

BONUS

Check* CANVAS → Google Drive → Resources *for Exam Reviews notes

EXTRA BONUS

Check* CANVAS → Google Drive → Resources → MATLAB *for more notes

Lecture Goals

MICHIGAN ENGINEERING
UNIVERSITY of MICHIGAN ■ COLLEGE of ENGINEERING

- **Exam "write some code" practice**
  - **You will receive 2 questions from previous exams**
  - **You will spend the first half of the lecture answering these questions**
    - **You may use your laptops during the lecture to test your code**
    - **You may work with your neighbours**
  -
  - **Answer code will be discussed during the second half**

TIPS

- **Try using a "bottoms-up" design where you start with individual features, such as "how does one delete a row with MATLAB?"**
- **Test these tasks interactively, and collect the working code fragments into your final function**

WHILE you are coding

MICHIGAN ENGINEERING
UNIVERSITY of MICHIGAN ■ COLLEGE of ENGINEERING

https://www.youtube.com/watch?v=4I_NYya-WWg&index=1&list=RD4I_NYya-WWg&nohtml5=False

Answer code will be placed in 00_Todays_Lecture after lecture

Question 1 – Write some code (25 points)

When performing experiments, there are often "bad measurements" that are included in the recorded data sets. One reason may be that a digital sensor was set off when sampled and provided a nonsense data point, such as temperature = -9999F. Proper data analysis techniques must recognize and remove poor quality measurements prior to reporting any results.

Your task is to write a MATLAB function that takes a series of meteorological observations and 1) removes poor quality data, and then returns 2) a corrected set of data with 3) the average value, it's standard deviation, and the maximum and minimum values.

The data that acts as "input" to your function will be a 2-dimensional array where each column consists of a different meteorological measurement (column 1 is temperature, column 2 is pressure, and succeeding columns are humidity, dew point, wind speed, etc.). The experiment gathered a large set of these same meteorological parameters over time, with different sets recorded in succeeding rows. All "bad measurements" have the value -9999. in that particular (row, column) data point. For example, the first 4 weather reports of the 5 parameters noted above are:

```
57.   29.46  64.   45.   11.
58.   -9999. 62.   46.   10.
59.   29.47  62.   -9999. 16.
61.   29.47  56.   48.   12.
```

Your function, `function [cout, cave, catd, cmax, cmin] = CleanUp(inp)` must perform the following operations:

1) clean up the input data, `inp`, returning `cout` which will not contain any rows with bad data; using the data above in our example, the return must be

```
57.   29.46  64.   45.   11.
61.   29.47  56.   48.   12.
```

2) return row vectors for `cave`, `catd`, `cmax`, and `cmin` that operate on the cleaned up data columns and determine i) the mean (`cave`), ii) it's standard deviation (`catd`), iii) the maximum value (`cmax`), and iv) the minimum value (`cmin`) of each column; in our example

```
cave = [59.   29.465  60.   46.5   11.5]
catd = [2.824 0.0071 5.6569 2.1213 0.7071]
cmax = [61.   29.47  64.   48.    12.]
cmin = [57.   29.46  56.   45.    11.]
```

Note, the formulae for `cave` and `catd` are the usual definitions for average and standard deviation. For example, for temperature,

average = (sum of all temperatures) / (the number of temperatures) = (57.+61.)/2. = 59.

standard deviation =

square root ((sum of squared difference between temperature and average) / (number – 1))

$$= \sqrt{\frac{(57.-59.)^2+(61.-59.)^2}{(2.-1.)}} = 2.824$$

Write your MATLAB function in the space provided on the next page.

Q1

- **Recall MATLAB functions**

  - To locate a `-9999.` in matrix `inp`, use
    `(inp == -9999.)` to set cells to `TRUE`

  - The `any` function is `TRUE` when a cell is non `0`
    - For a matrix, `any` works on columns
  - To convert rows to columns, use `inp'`

  - Deleting a row in MATLAB

```
z(2,:) = []  % deletes row 2
```

Q1

- **Putting these together,**

```
rinp = inp';
lbad = any(rinp == -9999.); % lbad=[0 1 1 0]
inp(lbad,:) = []; % lbad is a logical array
cout = inp;
```

- **In other words,**
  - **find the columns which contain any of the bad measurements in the transpose of the original matrix**
  - **delete the rows in the original matrix: these correspond to the columns in the transposed matrix**

Q1

- Then, `cout` contains the "good" data
- The summary return for temperature (col 1) is then

```
cave = mean(cout(:,1));
cstd = std(cout(:,1));
cmax = max(cout(:,1));
cmin = min(cout(:,1));
```

and for all parameters

```
cave = mean(cout);
cstd = std(cout);
cmax = max(cout);
cmin = min(cout);
```

Q1

- **Putting it all together,**

```
function [cout, cave, cstd, cmax, cmin] = CleanUp2(inp)

rinp = inp';
lbad = any(rinp == -9999.);
inp(lbad,:) = [];
cout = inp;

cave = mean(cout);
cstd = std(cout);
cmax = max(cout);
cmin = min(cout);

end
```

Q1

- **Alternatively for the first part, using iteration,**

```
[r c] = size(inp);
n = 0;
for i = 1:r
 lbad = inp(i,:) == -9999.;
 if sum(lbad) == 0
  n = n + 1;
  cout(n,:) = inp(i,:);
 end
end
```

Q1

- **And for the second part,**

```
for i = 1:c
  cave(i) = sum(cout(:,i))./n;
  cstd(i) = sqrt(sum((cout(:,i)-cave(i)).^2)/
(n-1));
  cmax(i) = max(cout(:,i));
  cmin(i) = min(cout(:,i));
end
```

Question 2 – Write some code (15 points per each function)

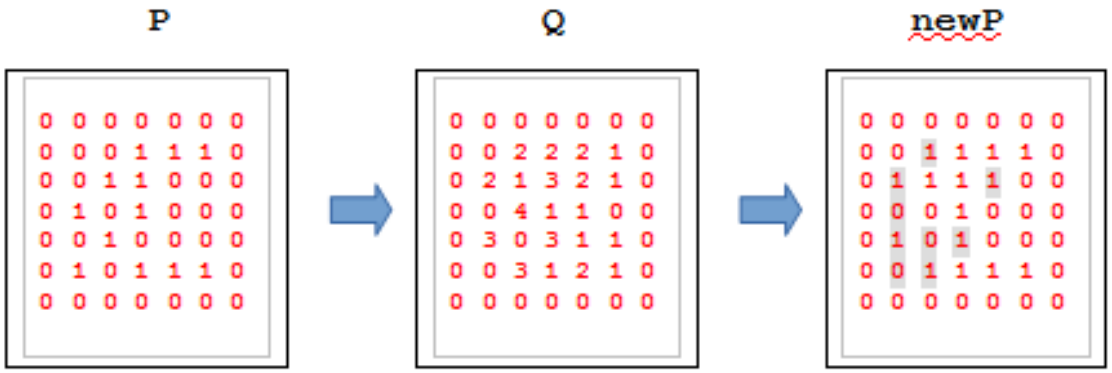
For this question, you must write two ***vectorized*** (no if selection or for/while iteration statements) MATLAB functions to aid in simulating a bacterial culture growing in a Petri dish. The original distribution of bacteria is represented by the matrix `P`, where each location in `P` contains either a 1 (occupied by a bacterium) or a 0 (unoccupied). The size of `P` is `NxN` where `N` is greater than 2. **The edges of `P` are always unoccupied (0) and correspond to rows 1 and `N` and columns 1 and `N`.**

Your task is to write two functions, the first called `neigbourCount`, the second named `petriStep`. Each takes a single input parameter, `P`, which is the `NxN` matrix of integers. For both functions, the input matrix contains only the values 0 or 1 representing cells that are unoccupied or occupied by a bacterium. The `neighbourCount` function must determine how many adjacent locations (above, below, left, and right) to this cell are occupied and store that number (either 0, 1, 2, 3, or 4) in the corresponding cell location of a matrix, `Q`. The return, `Q`, from function `neigbourCount`, will also be a matrix of size `NxN` with edge locations also set to 0.

The second function, `petriStep`, must first call the function `neighbourCount` to gain access to the output `Q` matrix. Using the information from both the `P` and `Q` matrices, the `petriStep` function must calculate the return matrix, `newP`, which contains the next generation of bacteria culture in the Petri dish, according to the following rules:

- unoccupied, non-edge locations in `P` that have 2 or 3 neighbouring occupied cells are set to occupied (1) in the corresponding location in `newP`
- occupied, non-edge locations in `P` that have 0 or 4 neighbouring occupied cells are set to unoccupied (0) in the corresponding location in `newP`
- all the remaining locations in `newP` that are not covered by the previous two cases are set equal to their corresponding locations in `P`

The following is an example of an initial matrix, `P`. `Q` is the matrix returned by the `neighbourCount` function using `P` as the input parameter. Finally, `newP` is the matrix returned from the `petriStep` function which also uses `P` as the only input parameter. Locations in `newP` that differ from `P` due to changes by running the `petriStep` function are highlighted in gray.



Q2 – `neighbourCount` function

- **Recall MATLAB functions**
  - **The size of a matrix is given by `size(P)` so `zeros(size(P))` is a same-size matrix**
  - **The last row or column in an array is `end`**
    - **So, `end-1` is the next to last row or column**

Q2 – `neighbourCount` function

- **And if an arbitrary cell is at `(a,b)` ...**
  - **The upper neighbour is at cell `(a-1,b)`**
  - **The lower neighbour is at cell `(a+1,b)`**
  - **The left neighbour is at cell `(a,b-1)`**
  - **The right neighbour is at cell `(a,b+1)`**

Q2 – `neighbourCount` function

- **Putting it all together**

```
function [Q] = neighbourCount(P)

Q = zeros(size(P));
Q(2:end-1,2:end-1) = P(1:end-2, 2:end-1) + … % above
                     P(3:end,   2:end-1) + … % below
                     P(2:end-1, 1:end-2) + … % left
                     P(2:end-1, 3:end);      % right
end
```

Q2 – `petriStep` function

- **Next generation rules**
  - **Unoccupied in P, with 2 or 3 neighbours in Q**

```
(P == 0) & (Q == 2 | Q == 3)
```

  - **Occupied in P, with 0 or 4 neighbours in Q**

```
(P == 1) & (Q == 0 | Q == 4)
```

  - **And the rest**
    - **If you start with**

```
newP = P
```

      then "the rest" will retain their starting designations

Q2 – `petriStep` function

- **Putting it together**

```
function [newP] = petriStep(P)

Q = neighbourCount(P);
newP = P;
newP((P == 0) & (Q == 2 | Q == 3)) = 1;
newP((P == 1) & (Q == 0 | Q == 4)) = 0;

end
```

Good Luck with the ENGR101 exam and with all your exams