MATLAB Programming NYCU Fall 2024 LAB#02 2024/9/19

Exercises in advanced indexing and program control.

1. Draw a filled circle (an exercise for logical indexing; use no loop)

(a) Make a square matrix **A** of size **nxn**. Make **n** an odd number.

(b) Compute the "distances" of all the elements to the center element. Store these in a "distance matrix" **D**, also of size **nxn**. Note: Function **meshgrid** is useful here.

(c) For a given radius **r** (**r** > 0; **r** can be a floating-point number), set **A(ii,jj)** to 1 if **D(ii,jj)<r**, and 0 otherwise. Then just print out **A**. Example below for **n**=7 and **r**=2.5:

```
0     0     0     0     0     0     0
0     0     1     1     1     0     0
0     1     1     1     1     1     0
0     1     1     1     1     1     0
0     1     1     1     1     1     0
0     0     1     1     1     0     0
0     0     0     0     0     0     0
```

2. Histogram of images (an exercise of loops and logical arrays)

Note: You can use **one level** of loop.

The histogram of an image is the distribution of gray scales of its pixels, and contains information useful for tasks such as contrast enhancement and segmentation. While MATLAB already provides functions for computing histograms, you will implement the task yourself.

First, load an image using

```
IM = rgb2gray( imread( image-file-name ) );
```

This gives you a 2D **uint8** array representing the image. You can use

```
figure;  imshow(IM);
```

to see the image.

Allocate a vector of size 256 to store the histogram, as the pixel values are 0~255.

Use a loop over the possible pixel values (i.e., 0~255), count the number of pixels with that value, and store it in your vector. There are several ways to count the elements in an array that satisfy a condition. For example, the expression **sum(A(:)==0)** will return the number of elements in **A** that are zero.

Compare your histogram with the histogram returned by the toolbox function **imhist(IM, 256)**.

3. Histogram equalization (an exercise in advanced indexing)

First, you have to normalize the histogram (make it sum to one) computed in the previous task.

This is a standard approach of image contrast enhancement. Without explaining the its theory (you can learn that in courses on image processing), let's say it computes a mapping of pixel values:

```
w(v) = uint8( sum( H(1:v+1) ) * 255 )
```

Here **H** is the normalized histogram, **v** is the original pixel value, and **w(v)** is its mapped new pixel value.

You can use a loop to compute the vector **w** with slight deduction. For full credit, you can compute **w** in one statement by using **cumsum** on **H**.

Now you can apply the mapping to the image in one statement. Check the last point in slide#8 of set#01, with **A** being the mapping vector and **B** being the (adjusted) input image. (Note: The adjustment is required because pixel values are 0~255 and array indices have to be 1~256. As a result, you need to use **(double(IM)+1)**, instead of **IM** itself, as **B** in the expression **A(B)** in the slide.)

Once you're done, you can use **imshow** to display the original and the contrast-adjusted images. (An example is shown here: Left: original; right: after histogram equalization.)