

# Lab 2

## MATLAB Scripts

### Objectives:

- Gain familiarity with MATLAB scripts
- Learn standard matrix operations, accessing multi-dimensional arrays, and basic text I/O

### Assignment:

#### 1. Creating a “Green-Screen” Image – 35 points

For special effects shots in movies and local weather forecasters everywhere, objects are often filmed in front of a bright green screen. The reason for this is that the bold green color is not present in the object being filmed. Thus, a computer can superimpose the green screen image on a background image by looking at each pixel in the green screen image, comparing it to the color green, and overwriting the corresponding pixel in the background if it isn't green.

Write a script (i.e. m-file) that creates a green-screen image from the F/A-18 image *f18.jpg* supplied on Canvas. Since the sky is a fairly uniform color, the script will need to check each pixel in the image to see if it is sky blue and replace it with bright green if it is. Assume that a pixel is sky blue if the red content is between 69 and 98 inclusive, green is between 91 and 120 inclusive, and blue is between 141 and 169 inclusive. The green should be red=0, green=255, and blue=0. Once generated, the green-screen image must be displayed using the *image* command.

The original image and desired output are shown below.



## 2. Array Operations vs. Loops – 20 points

The goal of this problem is to compare the code structure of two programs which perform the same operations but through different means. One will use array/matrix features of MATLAB and the other will use iterative/nested loops.

For this problem, generate an image which is 250 x 250 pixels with gradually fading colors. To fade a color, we need to modify the RGB pixel values gradually throughout the (x,y) positions of the image matrix.

As mentioned above, we will examine two different ways of doing this. The first way is to use vectors and MATLAB's matrix multiplication features. Start by creating a vector which increases by 1 up to 250, and copying it (simply multiplying it by 1) through a second dimension 250 times. Consider the following matrix multiplication example:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ \dots \\ 250 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 2 & 2 & 2 & \dots & 2 \\ 3 & 3 & 3 & \dots & 3 \\ \dots & \dots & \dots & \dots & \dots \\ 250 & 250 & 250 & \dots & 250 \end{bmatrix}$$

We can perform this type of matrix multiplication in MATLAB simply using the “\*” (multiplication) operator, paying attention to the dimensions of the two vectors. Recall that when performing matrix multiplication, the inner dimensions of the two matrices (or vectors in this case) must match, e.g. a 250x1 multiplied by a 1x250 is a valid operation, yielding a 250x250 result matrix, but a 250x1 multiplied by a 250x1 is invalid.

To perform this operation in MATLAB, consider the following code:

```
>> [1;2;3]*[1 1 1]
```

```
ans =
```

```
1      1      1
2      2      2
3      3      3
```

We can simply extend both vectors in the code above to be 250 elements each. Try executing the following commands in the MATLAB command window. Note the use of the “'” (single quote) symbol after the first vector. This performs a transpose operation, effectively swapping the row and column dimensions. You should see an image (corresponding to variable “x”) with the top side shaded black, and fade to white on the bottom side:

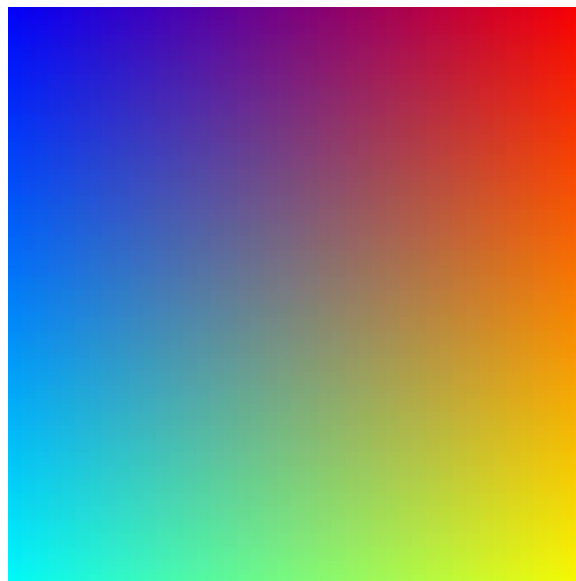
```
x = uint8(zeros(250,250,3)); %Create black box
x(:,:,1) = [1:250]'.*ones(1,250); %modify all red pixels
x(:,:,2) = [1:250]'.*ones(1,250); %modify all green pixels
x(:,:,3) = [1:250]'.*ones(1,250); %modify all blue pixels
image(x); axis square; axis off; %display image x
```

For the second method, use two for-loops (one placed within the other) to cycle through each pixel in the image and modify its RGB value. When using for-loops, we have access to a counter-variable which tracks the iteration number of each loop. Using the counters, we can update the RGB values as the loop travels through the image pixel-by-pixel. Consider the following code below; you should see the same image generated as before, now stored in variable “y”:

```
y = uint8(zeros(250,250,3)); %Create black box
for i=[1:250] %scan through rows
    for j=[1:250] %scan through columns
        y(i,j,1) = i; %update R-value of pixel at (i,j)
        y(i,j,2) = i; %update G-value of pixel at (i,j)
        y(i,j,3) = i; %update B-value of pixel at (i,j)
    end
end
image(y); axis square; axis off; %display image
```

Using these two concepts, create a three dimensional array that corresponds to the image shown below. The image is 250 pixels by 250 pixels. The red components of the pixels are increasing from 1 to 250 with an increment of 1, starting from left to right (all the right-most pixels have a red-value of 250 and all the left-most pixels have a red-value of 1). The green components of the pixels are also increasing from 1 to 250 with an increment of 1, but start from top to bottom. The blue components of the pixels increase in the same fashion, but start from right to left (opposite of red).

Generate two separate m-files using the code in the above sections. The m-files must display the array as a picture using the *image* command.



## 3. Inputs &amp; Outputs – 35 points

For this problem, you will modify the script “circle\_demo.m” file, which draws a red shaded circle on a black background by modifying only the red color context (leaving the blue and green unaffected). Enhance the script so that it draws three shaded circles affecting only the red, green, or blue content respectively, each with different radii and x-y locations within the image. Also, the script must count the number of pixels used to draw each circle within the displayed image. To draw this image, there are 10 different values you need to collect beforehand: the dimension of the image, three radii values, and three sets of x-y locations.

The dimension of the image (assuming equal dimensions) and the radii of the three circles must be read from a file with the specified format. You may “hard code” the file name into the script, meaning that you must modify the filename in the code to read a different file. The format of the file is shown below, where XXX is a valid integer value. There are three examples of files with valid format and values on Canvas that you can/must use for testing. Assume that values and file format are valid (i.e. you don’t need to check for erroneous data/formatting).

Input text file format:

```
Dim: XXX
Red Radius XXX
Green Radius XXX
Blue Radius XXX
```

The x-y locations are obtained by prompting the user for the coordinates of the three circles with the “input” command. There must only be three prompts and the prompts to the user must be appropriate and useful, i.e. they should tell the user how to enter the data (specify the format). Assume that the values entered by the user are valid. An example of a user prompt is shown below.

```
Please enter red circle location [x,y]:
```

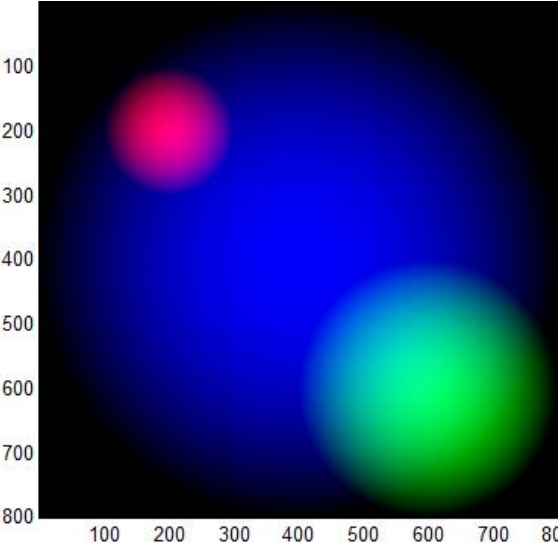
Once all the required information is gathered, the script must create the image with red, green, and blue shaded circles just as the sample script creates only a red circle. As mentioned above, the script must count the number of pixels used when creating each colored circle. The script must create an output file called “counts.txt” which contains a printed text output of the number of pixels in each circle that lie within the image, along with the location of the circle’s center. The format of the output file is given below, where XXX and YYY are the x-y locations of the circles (gathered from the user input) and ZZZ is the number of pixels counted during the creation of each circle:

Output text file format:

```
The red circle at (XXX,YYY) contains ZZZ pixels.
The green circle at (XXX,YYY) contains ZZZ pixels.
The blue circle at (XXX,YYY) contains ZZZ pixels.
```

Run the finished script on each of the three supplied test input files, specifying that the circles are in the center of each image.

The following is an example showing the inputs and outputs of this program. Again, the two inputs will be a text file and data entered interactively by the user. The outputs should be an image, and a text file with some calculated values inside.

<p>The input text file might look like:</p> <pre>Dim: 800 Red Radius 100 Green Radius 200 Blue Radius 400</pre>	<p>The user input might look like:</p> <pre>Red circle location: [x,y]=[200,200] Green circle location: [x,y]=[600,600] Blue circle location: [x,y]=[400,400]</pre>
<p>The output image for these inputs would be:</p> 	<p>The text output inside "counts.txt" would be:</p> <pre>The red circle at (200,200) contains 31397 pixels. The green circle at (600,600) contains 125609 pixels. The blue circle at (400,400) contains 502605 pixels.</pre>

### Deliverables:

Upload an electronic copy of the deliverables to Canvas per the lab submission requirements. Please note: **ALL** deliverables must be present in the single PDF report, and m-files must also be included as separate files with logical names.

#### Scoring:

- 10 points – Compliance with Submission Guidelines
- 35 points – m-file from Part 1
- 20 points – 2 m-files from Part 2
- 35 points – m-file and three copies of count.txt from Part 3