PSYCH 20A (PROF. FRANE) – LECTURE 9: IMAGES

GRAYSCALE IMAGES

A grayscale image may contain black, white, and any shade of gray. To represent a grayscale image in a computer, all you need is a matrix in which each value represents the amount of light at the given pixel position. Using a 0-to-1 range of values, 0 is black (no light), 1 is white (full strength light), .2 is a dark gray, .5 is a medium gray, .8 is a light gray, etc. Below, we define a 9 × 19 matrix called `crossImageMatrix`. It represents an image that is 9 pixels tall and 19 pixels wide, containing a 1-pixel thick white cross on a black background:

```
crossImageMatrix = [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                    0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0] ;
```

Below, we define a 16 × 13 matrix called `hImageMatrix`. It represents an image that is 16 pixels tall and 13 pixels wide, containing a light gray capital letter H on a white background:

```
hImageMatrix = [1   1   1   1   1   1   1   1   1   1   1   1   1
                1   1   1   1   1   1   1   1   1   1   1   1   1
                1   1  .8  .8   1   1   1   1   1  .8  .8   1   1
                1   1  .8  .8   1   1   1   1   1  .8  .8   1   1
                1   1  .8  .8   1   1   1   1   1  .8  .8   1   1
                1   1  .8  .8   1   1   1   1   1  .8  .8   1   1
                1   1  .8  .8   1   1   1   1   1  .8  .8   1   1
                1   1  .8  .8  .8  .8  .8  .8  .8  .8  .8   1   1
                1   1  .8  .8  .8  .8  .8  .8  .8  .8  .8   1   1
                1   1  .8  .8   1   1   1   1   1  .8  .8   1   1
                1   1  .8  .8   1   1   1   1   1  .8  .8   1   1
                1   1  .8  .8   1   1   1   1   1  .8  .8   1   1
                1   1  .8  .8   1   1   1   1   1  .8  .8   1   1
                1   1   1   1   1   1   1   1   1   1   1   1   1
                1   1   1   1   1   1   1   1   1   1   1   1   1] ;
```

A matrix representing pixels (such in our examples) is sometimes called a “bitmap” or a “raster.”

COLOR IMAGES

Your computer screen can be divided into a matrix of pixel positions. There are three light emitting diodes at each pixel position: a red one, a green one, and a blue one. All the colors your monitor can show are made by mixing the light from those three *color-channels*. Each color can be defined using a 3-value vector called an *RGB triplet*, which specifies the amount of red, green, and blue, respectively. For example, using a 0-to-1 range of values, [1 0 0] is a bright red (full strength red light, and no green or blue light). [.5 0 0] is a dark red (some red light, and no green or blue light), [1 1 0] is a bright yellow (red + green light make yellow), etc. Any RGB triplet with the same value in each color channel is a grayscale tone. For example, using a 0-to-1 range of values, [0 0 0] is black (no light on any channel), [1 1 1] is white (red + green + blue light make white), [.2 .2 .2] is a dark gray (essentially a low level of white light), etc.

To represent a color image in a computer, you can use a 3-page array called an *RGB array*. Page 1 is a matrix in which each value represents the amount of red light at the given pixel position; page 2 is a matrix in which each value represents the amount of green light at the given pixel position; and page 3 is a matrix in which each value represents the amount of blue light at the given pixel position.

Let's make a 9 × 19 × 3 RGB array called `crossRGB`., representing an image that's 9 pixels tall and 19 pixels wide. The image will contain a 1-pixel thick bright yellow cross on a black background. Thus, we want the color to be [1 1 0] (bright yellow) where the cross is, and [0 0 0] (black) everywhere else. The red layer looks like this:

```
crossRGB(:, :, 1) = [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0] ;
```

And the green layer looks the same as the red layer:

```
crossRGB(:, :, 2) = [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0] ;
```

And there's no blue light anywhere in the image, so the blue layer is all zeros:

```
crossRGB(:, :, 3) = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] ;
```

DISPLAYING AN IMAGE IN A FIGURE WINDOW

To display an image, first open a figure window using the `figure` function. You can specify a figure number (which can be any positive integer) as an input:

```
figure(1) % open figure 1 window
```

If you don't input a figure number, then Matlab automatically assigns the lowest figure number that isn't already in use.

To show an image in the figure window you opened, use the `imshow` ("image show") function (you can input either an array or an image filename):

```
imshow(crossRGB) % show crossRGB image in current figure window
```

For RGB arrays, the `image` function is similar to `imshow`, but it labels the vertical and horizontal axes with numbers that count the rows and columns of pixels in the image:

```
image(crossRGB) % show crossRGB image in figure window and label axes
```

You can have multiple figure windows open at once. At any time, you can select a given figure window (making it the "current" figure window, and brining it to the front if it was behind some other window before) by using the figure function again:

```
figure(1) % select figure 1 window as current figure window
```

To place a title above the image, use the `title` function:

```
title('Yellow Cross Image') % place title above image
```

The font properties of the title can be customized using "name–value pairs" as additional input arguments. Input the name of the property you want to set, followed by the value you want to set it to. For example, here we set the font color to blue, the font size to 16, the font weight to bold, the font angle to italic, and the font name to Courier:

```
title('Yellow Cross Image', 'Color', 'blue', 'FontSize', 16, ...
       'FontWeight', 'bold', 'FontAngle', 'italic', 'FontName', 'Courier')
```

Matlab only recognizes a limited number of color names, but you can use an RGB triplet (in the 0-to-1 range) instead of a word. For example, to make the font dark blue:

```
title('Yellow Cross Image', 'Color', [0 0 .3])
```

To close specific figure windows, input the figure number (or a vector of figure numbers) to the `close` function:

```
close(1) % close the figure 1 window
```

To close all figure windows:

```
close all % close all figure windows
```

DISPLAYING MUTIPLE IMAGES IN A SINGLE FIGURE WINDOW

We can arrange multiple graphs in one figure by using the `subplot` function, which takes three inputs: the number of rows, the number of columns, and the position number of the current graph. The position number isn't counted down each column like when using linear indexing on a matrix. Instead, the position number is counted across each row. Thus, for example, in a 2 × 3 arrangement of graphs, the position numbers are:

```
1     2     3
4     5     6
```

The following code displays images on a 2 × 2 grid. The `crossImageMatrix` image is in the top-left corner (position 1), the `hImageMatrix` image is in the top-right corner (position 2), and the `crossRGB` image is in the bottom-left corner (position 3). There's nothing in the bottom-right corner (position 4):

```
figure(1) % open figure 1 window

subplot(2, 2, 1) % select position 1 in the 2x2 grid of images
imshow(crossImageMatrix) % show crossImageMatrix image at that position
title('White cross') % place title above that image

subplot(2, 2, 2) % select position 2 in the 2x2 grid of images
imshow(hImageMatrix) % show hImageMatrix image at that position
title('Light gray H') % place title above that image

subplot(2, 2, 3) % select position 3 in the 2x2 grid of images
imshow(crossRGB) % show crossRGB image at that position
title('Yellow cross') % place title above that image
```

Or to make that last image centered across positions 3 and 4, we could change the `subplot` command to:

```
subplot(2, 2, 3:4) % select positions 3 and 4 in the 2x2 grid of images
```

To add an overarching title to the whole figure, use the `sgtitle` ("suplot grid title") function. The font properties can be customized just like with the `title` function:

```
sgtitle('Some Simple Images', 'Color', [0 0 .3], 'FontSize', 24)
```

USING INTEGER RANGES OF VALUES

Instead of using the 0-to-1 range, you can use a 0-to-255 integer range in which 0 indicates no light and 255 indicates full strength light. To do that, the class of the array must be uint8 ("unsigned 8-bit integer"; unsigned means the values have no positive/negative sign, and "8-bit" means there are 2^8 = 256 possible values). To convert an image array from the 0-to1 range to the 0-to-255 integer range, we can multiply all the values by 255 and apply the `unit8` function:

```
crossRGBasUint8 = uint8(255*crossRGB) ; % convert 0-to-1 array to uint8
```

To convert an image array from the 0-to-255 integer range to the 0-to-1 range, we can apply the `double` function and divide by 255:

```
crossRGB = double(crossRGBasUint8) / 255 ; % convert uint8 array to 0-to-1
```

Or you can use the `im2double` function, which does the class-conversion and division in a single step:

```
crossRGB = im2double(crossRGBasUint8) ; % convert uint8 array to 0-to-1
```

The image will look the same regardless of which range you use. On most systems, each color channel is limited to 8 bits (meaning there are 2^8 = 256 possible levels of brightness). So even if you use the 0-to-1 range, Matlab essentially converts to an 8-bit integer scale when displaying the image anyway. But uint8 values take up less space in memory than double precision values.

If an image contains only pure black and pure white, then it can be represented as an array that contains only 0s and 1s. In that case, we can use logical values (which only require 1 bit, meaning there are 2^1 = 2 possible values):

```
crossImageMatrixAsLogical = logical(crossImageMatrix) ;
```

IMPORTING IMAGES

We can import an image from a file into an array by using the `imread` function. For example, here we import an image from a file called peppers.png (which comes with Matlab) into a class uint8 RGB array called `peppersRGB`:

```
peppersRGB = imread('peppers.png') ;
```

Most image files are formatted as 8-bit integer arrays, so they are imported as class uint8. For example, color image files typically use 8 bits per value on each of the 3 color channels (this format is called "true color," which has a "24 bit color depth" because there are 8 × 3 = 24 bits used to define each pixel). But in rare cases, images may be formatted using 16 bits per value, so they are imported as class uint16. And some black-and-white image files are formatted using 1 bit per value, so they are imported as logicals.

Special cases:
Some image files represent the image as two separate objects—an image matrix and a "color map"—rather as a single image array like we've been using. Some file formats (such as gif and tif) may contain multiple images stacked in a single file. Some image files include an "alpha" or transparency layer. See the documentation for how to import images in those cases.

SAVING IMAGES

An image array can be saved to a .mat file by using the `save` function, like any other object in the workspace.

Or you can export an image from an image array to an image file by using the `imwrite` function. If your array is double-precision, the exported data will typically be converted to an 8-bit integer scale. Matlab infers the format of the file from the extension you put in the filename, e.g.:

```
imwrite(crossRGB, 'crossRGB.png')
```

Or you can export all the contents of a figure window to an image file by using the `saveas` function—but use caution because the images may have been resized or otherwise altered to fit the figure window. The first input to the `saveas` function can be either the figure number or `gcf` ("get current figure," in which case the contents of the current figure window are saved):

```
saveas(gcf, 'someSimpleImages.pdf')
```

As always, if you want to save to a folder other than the present working directory, you can enter a full path rather than just the filename.

Saving in raster/bitmap file formats (such as bmp, jpg, png):
Bitmap file-formats are often used for photographic images. But for text, logos, and graphs, bitmap file-formats often make the image look too pixilated for professional non-web use, especially when enlarged or printed. And jpg is rarely used outside of the web, since it typically uses compression, which reduces quality.

Saving in vector-graphics compatible file formats (such as eps, pdf):
Vector-graphics compatible file formats can avoid the pixilation problem because they include instructions for drawing lines and curves so the lines and curves stay smooth even when the image is enlarged or printed. You can't use the `imwrite` function to save as an eps or pdf file, but you can put the image in a figure window and use the `saveas` function.

To save the contents of a figure window in color using the eps file format, add 'epsc' (which stands for "eps color") as the third input to the `saveas` function:

```
saveas(gcf, 'someSimpleImages.eps', 'epsc')
```

If your eps or pdf file looks pixilated after zooming in, even when viewing it in a vector-graphics compatible application, that may mean that the vector graphics weren't *rendered* properly before saving. In that case, try using the `set` function to set the renderer of the figure window to `'Painters'` before using `saveas` (caution: this may affect transparencies):

```
set(gcf, 'renderer', 'Painters')
```