ENPH 213 Laboratory 6, 2020

The objective of this week's laboratory is to solve a system of linear equations using Gaussian elimination with pivoting. To accomplish this efficiently, you should use matrix algebra.

Part A – Due at the end of the lab period

Task A1:

Write a function `pivot` with the following inputs/outputs in the following order:

INPUTS	
`m`	A matrix of arbitrary dimensions, representing a system of equations.

OUTPUT	
`mPivot`	A matrix with the same dimensions as `m`, representing an *equivalent* system of equations. The row containing the first column element with the largest **magnitude** should now be in the first row.

Make sure that you pivot the entire row, not just the first element. You can use the built-in Matlab `max` function to determine which row contains the maximum value in each column. Use the test function **test_pivot.m** from onQ to test your function.

Task A2

Write a function `elim` with the following inputs/outputs in the following order:

INPUTS	
`m`	A matrix of arbitrary dimensions, representing a system of equations.

OUTPUT	
`mElim`	A matrix with the same dimensions as `m`, representing an *equivalent* system of equations. Aside from the (row 1, column 1) element, the first column should be zeros.

Make sure to pivot first. Your function should return an error if all the elements in the first column are zero, this tells us that the system is under-determined. Recall the elementary row operations, which preserve the system of linear equations:

- Swap any two **rows**
- Multiply each element of any row by a constant
- Add or subtract one row from another

Recall that you can extract the ith row of a matrix `m` with the call `m(i,:)` where the colon indicates you are extracting all of the columns. Use the test function **test_elim.m** from onQ to test your function.

Task A3

Write a function `upTri` with the following inputs/outputs in the following order:

INPUTS	
`m`	A matrix of arbitrary dimensions, representing a system of equations.

OUTPUT	
`mUpTri`	A matrix with the same dimensions as `m`, representing an *equivalent* system of equations. The matrix should be in upper triangular form.

Your function should put the matrix `m` in upper triangular form by calling your `elim` function on progressively smaller submatrices. Use the test function **test_upTri.m** from onQ to test your function.

Task A4

Write a function `backSubs` with the following inputs/outputs in the following order:

INPUTS	
`AB`	A matrix of dimensions **(n, m)**, representing a system of equations.

OUTPUT	
`x`	A column vector of dimensions **(m-1, 1)**, representing the solution to the system of equations represented by the matrix `AB`.

The system of equations represented by the augmented matrix `AB=[A,B]` is:

$$\begin{bmatrix} \quad & \boldsymbol{A} & \quad \end{bmatrix} * \begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} B \end{bmatrix},$$

where the A matrix is the first matrix, and the B vector is vector of dependent values from you system of equations.

In your function, you need to check the following:

- Is the system of equations under-determined (*i.e.* whether A has more columns than rows)?
- Is the system of equations over-determined (*i.e.* whether A has fewer columns than rows)?
- If A is neither under-determined or over-determined, is it singular (det(A) = 0)?

If any of these checks are true, output an appropriate error message using Matlab's `error` function. Use the test function **test_backSubs.m** from onQ to test your function. You may also use matrix division `x=A\B` to check your work.

Task A5:

Write a function `inverse` with the following inputs/outputs in the following order:

INPUTS	
`m`	A square matrix of dimensions **(n, n)**.

OUTPUT	
`mInv`	A square matrix of dimensions **(n, n)**, the inverse of the matrix `m`.

Recall that the inverse of a matrix `m`, m^{-1}, is defined by:

$$\begin{bmatrix} & \boldsymbol{m} & \end{bmatrix} * \begin{bmatrix} & \boldsymbol{m^{-1}} & \end{bmatrix} = \begin{bmatrix} & \boldsymbol{I} & \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Solve for each column of the identity matrix separately using your function `backSubs`. For example, if `m` is a 3x3 matrix, the first column of `mInv` is given by `mInv(:,1) = backSubs(m,I(:,1))`. Use the test function **test_inverse.m** from onQ to test your function. You may also use the built-in Matlab function `inv` to check your work.

Part A: Submission List
When complete, please submit the following files to the **Lab 6: Part A** dropbox on onQ for marking:

- `pivot.m`
- `elim.m`
- `upTri.m`
- `backSubs.m`
- `inverse.m`
- Any other functions you wrote that the above files use

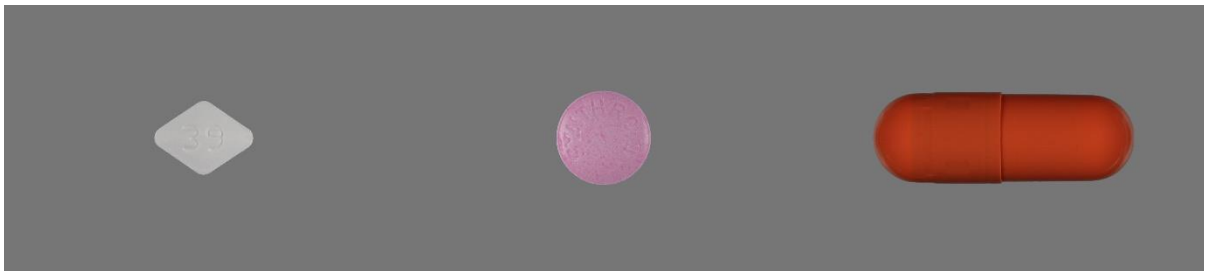
Part B: Due Sunday @ 9PM

The United States National Library of Medicine (NLM) has challenged you to program a “Pill Recognizer” (the challenge webpage can be found in [1]). Pharmaceutical tablets are regulated by the Food and Drug Administration (FDA) [2]:

“no drug product in solid oral dosage form may be introduced or delivered for introduction into interstate commerce unless it is clearly marked or imprinted with a code imprint that, in conjunction with the product's size, shape, and color, permits the unique identification of the drug product and the manufacturer or distributor of the product”

In plane English, tablets must have a unique combination of shape, color and imprinted code. Even with these unique identifiers, mislabeled and unidentified tablets remain a serious problem for doctors and their patients. Taking the wrong tablet and/or the wrong dose can result in adverse drug reactions and in extreme cases, can even cause death.

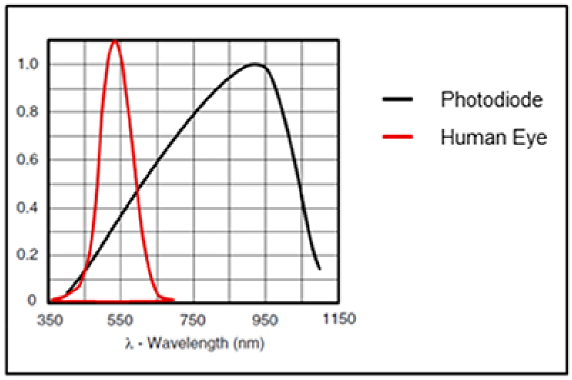
Your program will take as its input an image of an unknown tablet and provide as its output a measure of how circular the tablet is. Some example images from the NLM RxIMAGE database are shown below:



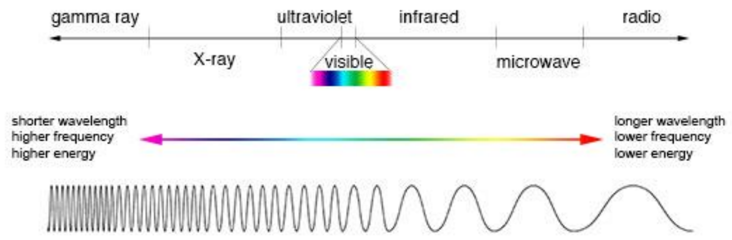
The shape of the tablet is one **feature** that can be used to **classify** the tablets and **recognize** which table is which. This would be useful for patients, to avoid taking the wrong tablet and/or dose, as well as pharmaceutical manufacturers, to perform quality insurance on their manufacturing lines (*e.g.* ensuring all of the tablets coming off the line are circular). Identifying the characters imprinted on a tablet is a more difficult problem and is outside the scope of ENPH 213. For those interested in this aspect of the problem and Optical Character Recognition (OCR) in general, refer to.

The aim of Part B for this week’s lab is to extract the edge points of a given tablet image. Next week, we will fit a circle to these edge points by solving a system of non-linear equations. The “goodness” of this fit will tell us how circular the tablet is. First, a primer on image processing:

Images are 2D arrays of intensity values generated by an image sensor. An image sensor is made up of an array of photosensitive elements that convert incident photons into electrons, where the build up of electric charge is proportional to the intensity of light. The accumulated charge is transferred from the sensor to a frame grabber, which converts the electrical signal from analog to digital. Each photosensitive element corresponds with a pixel in the final digital image. Standard cameras, like the one in your cellphone, use silicon semiconductors as the photosensitive elements. A typical responsivity curve for one of these sensors is shown below [3].



For our purposes the y-axis in the above plot is arbitrary, as long as we know it is some measure of sensitivity or, in other words, the number of electrons generated by a photon of a given wavelength. As you can see, typical silicon sensors are sensitive not only in the visible range ~350-700nm but also into the near-infrared (up to ~1100 nm). This only samples a small portion of the electromagnetic spectrum, shown below [4]:



Not all images come from silicon sensors. Those of you that attended the talk by Francis Halzen on IceCube may remember him mentioning “multi-messenger astronomy”. This is the ability to capture images of astronomical events using gravitational wave, radio, optical, X-ray, gamma ray and neutrino detectors [5]. These telescopes and detectors are all generating **images**, a map of intensity over space. Closer to home, infrared cameras use sensors with responsivity curves covering the near to far infrared and are employed in a host of applications, including night vision googles.

Matlab handles images much the same way that it handles any other type of matrix data [6]. The images we will be working with are formatted as JPEGs but Matlab can handle a host of image formats. We will carry out the following image processing steps on the sample tablet images:

- Color to monochrome
- Resize image
- Gaussian filtering
- Edge detection

The first step in image processing with Matlab is to read the image into Matlab. We will use the command `imread` to the read a sample tablet image into Matlab. Make sure you have the sample tablet image files downloaded and in your current working directory. Run the following command:

```
tablet2 = imread('tablet2.jpg');
```

`tablet2` should now be a variable in your workspace, if you run the command:

```
size(tablet2)
```

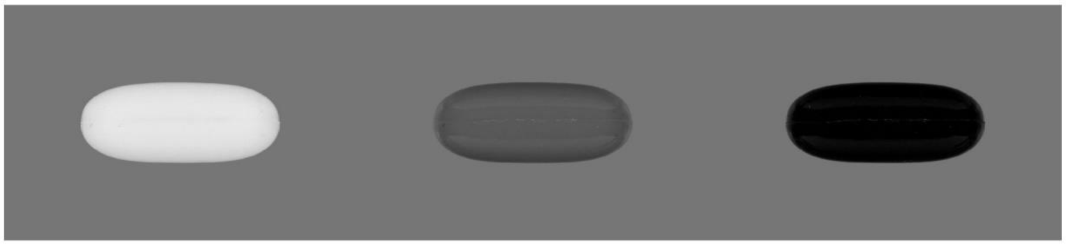
Matlab should return:

```
ans =

        1600          2400             3
```

The first two numbers tell us that the image resolution is 1600x2400 (fairly large). The last number tells us that the image is a color image rather than a monochrome or "black-and-white" image. This image is made up of three 1600x2400 arrays, one for each of red, green and blue color components. Using the command `imshow`, which displaces an image, you can easily see that any one of these arrays will appear like a monochrome image but the combination of all three yields a color image. Run the following commands:

```
imshow(tablet2(:, :, 1))
imshow(tablet2(:, :, 2))
imshow(tablet2(:, :, 3))
imshow(tablet2)
```

Since the sample tablet is orange, the tablet should appear black in the blue color component image (`tablet2(:, :, 3)`).



You will also notice that the data type is `uint8`, which means each of the red, blue and green color components for each pixel are allotted 8 bits of **bin depth**. This tells that there are 2^8 different reds, greens, and blues for each pixel. This allows for over 16 million different colors (combinations of red, green, and blue).

Since we are focusing on the shape rather than the color we can throw away the color information and work with monochrome images. What we want is a 1600x2400**x1** array of 8-bit intensity values from black, given a value of 0, to white, given a value of 255, with 254 different greys in between. On the surface, this seems quite simple. Why not just use an average of red, green and blue color components for each pixel? Well it turns out that the different detectors (called cones) in our eyes do not correspond directly with the red, green and blue filters on silicon photosensitive elements. For this reason, there is some debate over which color-to-monochrome conversion matches our perception [7]. For our purposes the built-in Matlab function `rgb2gray` will work just fine:

```
tablet2bw = rgb2gray(tablet2);

size(tablet2bw)

ans =

        1600        2400

imshow(tablet2bw)
```

It might have been more convenient to work with just the red color component for this tablet but our program needs to work for all tablet colors.

The next step is resizing our image, 1600x2400 pixels is much more than we are going to need to accurately detect the edges of the tablet. Use Matlab's built-in command `imresize` to reduce the resolution to 160x240, much more manageable.

To understand how Gaussian filtering and Sobel edge detection work we need to go on a brief mathematical interlude [8]. Many image filters are **convolutions.** The convolution of the continuous function $f(x)$ with the continuous function $g(x)$ is expressed as:

$$[f * g](x) = \int_{-\infty}^{\infty} f(y)g(x-y)\,dy$$

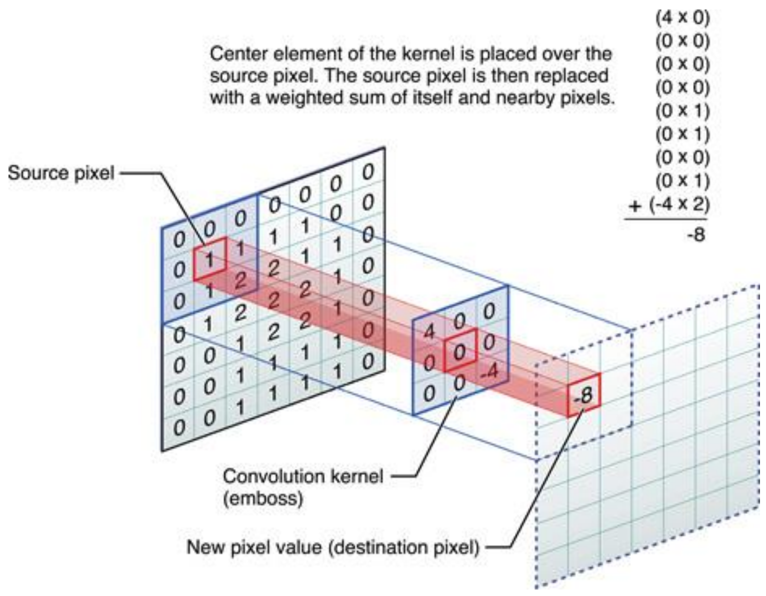
First off, convince yourself that the function $g(x-y)$ is identical to $g(x)$ but shifted to the **right** by y. Second thing, the product of the functions $f(y)$ and $g(x-y)$ is a measure of their overlap. If they don't overlap at all their product is zero, if they overlap quite a bit their product is large. Wolfram has a great animation of this, found in . For discrete functions we just replace the integral with a sum:

$$[f * g](n) = \sum_{i=-\infty}^{\infty} f(i)g(n-i)$$

In the 2D discrete case, we have two indices rather than one, one for each of our axes:

$$[f * g](m,n) = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} f(i,j) g(m-i, n-j)$$

Instead of shifting our function along one axis, we have to shift it over two. In image processing convolution is performed using a **convolution kernel**. A convolution kernel is a matrix of numbers. The values in the matrix dictate the effect of the convolution, *i.e.* the difference between the original and processed images. Performing a convolution on an image can be thought of as moving a matrix (the convolution kernel) across a much large matrix of numbers (the image). An example is shown below [9]:


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

This operation is then repeated for every pixel in the image. The convolution kernel is shifted such that it is centered over the pixel. A weighted sum of the kernel values and the original pixel values is carried out. The result is the new pixel value. As you can probably guess the kernel:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

results in no change to the original image. None of the neighbouring pixel affect the new pixel value, there is a one-to-one correspondence between original and new pixel values. One last point regarding the image boundaries. The boundaries of the original image are effectively padded with zeros when the kernel extends beyond the original image boundaries (when performing the weighted sum for any elements in the first or last row or column for example).

Task B1:

Write a function `imFilter` with the following inputs/outputs in the following order:

INPUTS	
`imOrig`	Original image matrix with dimensions **(m, n)**
`kernel`	Convolution kernel with dimensions **(p, q)**, where **p is less than m** and **q is less than n** and **both p and q are odd.**

OUTPUT	
`imNew`	Filtered image matrix with dimensions equal to those of the original image matrix, **(m, n).**

For each pixel multiply the pixel and its neighbours by the convolution kernel `kernel`, and sum over the resulting values to find the corresponding element in the new filtered image. You will need to correct the range of intensity values and enforce that your output image matrix is of the type `uint8` using the following commands at the end of your function:

```
imNew = abs(imNew);
imNew = imNew - min(min(imNew));
imNew = (imNew/max(max(imNew))).*255;
imNew = uint8(imNew);
```

Make sure that you supress all of your outputs with semicolons. Any submissions that fail to do so will not be marked. Use the test function **test_imFilter.m** from onQ to test your function.

The third and fourth test kernels in the test function **test_imFilter.m** are Gaussian blurring or smoothing filters for 3x3 and 5x5 neighbourhoods. Now all we need is a convolution kernel that will result in an image matrix with high intensities near the edges and low intensities everywhere else. An edge in an image is defined by a sudden change in intensity. This sounds a lot like the local derivative. The slope of the intensity at a pixel tells us how likely it is that that pixel is part of an edge. Recall from previous labs that we approximated the slope of a 1D function as:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{h}$$

Also called the central difference formula. The Sobel operator performs a similar operation along both axes of the image, in the "right" direction and the "down" direction at every pixel. The gradient approximations are then combined like vectors to produce a gradient magnitude for each pixel in the original image.

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \text{imOrig}$$

$$S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * \text{imOrig}$$

$$\text{imNew} = \sqrt{(S_x)^2 + \left(S_y\right)^2}$$

Task B2:

Write a function `tabletEdgeDetection` with the following inputs/outputs in the following order:

INPUTS	
`tabletImgFilename`	Filename of the sample tablet image, string, *e.g. 'tablet2.jpg'*

OUTPUT	
`tabletImgSobel`	Filtered image matrix with dimensions equal to those of the original image matrix, **(m, n).**

Your function should perform the following image processing operations in this order:

- Read the image using `imread`
- Convert the image to monochrome using `rgb2gray`
- Resize the image to be 160x240 pixels using `imresize`
- Subtract the background pixel value and then convert the resulting pixel values back into type `double`, my solution included the following commands:

```
        tabletImgBW = double(tabletImgBW);
        tabletImgBW = abs(tabletImgBW - tabletImgBW(1, 1));
        tabletImgBW = tabletImgBW - min(min(tabletImgBW));
        tabletImgBW = (tabletImgBW/max(max(tabletImgBW))).*255;
        tabletImgBW = uint8(tabletImgBW);
```

- Perform a 5x5 Gaussian blurring operation using your function `imFilter` and the fourth test kernel give in **test_imFilter.m**
- Perform a Sobel edge detection operation, you will need to use your `imFilter` function twice
- Binarize the image using `imbinarize` with a threshold of 0.65

Once you have performed the two Sobel convolutions, you will need to convert the results into type `double`, my solution included the following commands:

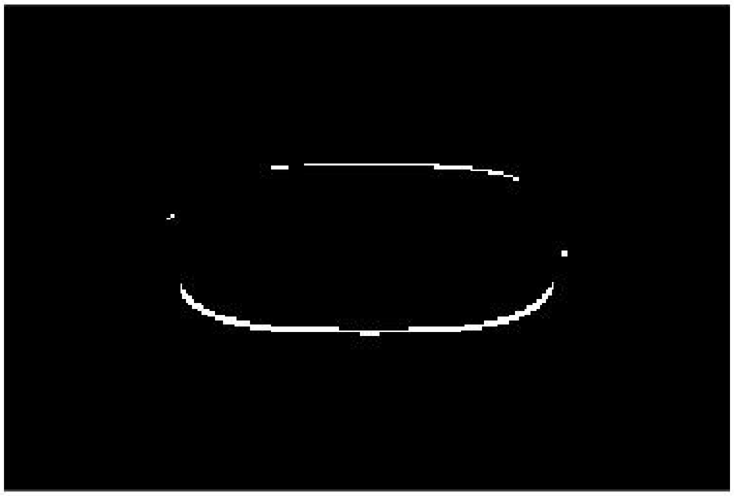
```
tabletImgSX = double(tabletImgSX);
tabletImgSY = double(tabletImgSY);

tabletImgSobel = sqrt(tabletImgSX.^2 + tabletImgSY.^2);

tabletImgSobel = abs(tabletImgSobel);
tabletImgSobel = tabletImgSobel - min(min(tabletImgSobel));
tabletImgSobel = (tabletImgSobel/max(max(tabletImgSobel))).*255;
tabletImgSobel = uint8(tabletImgSobel);
```

where `tabletImgSX` and `tabletImgSY` are the results of the two Sobel convolutions.

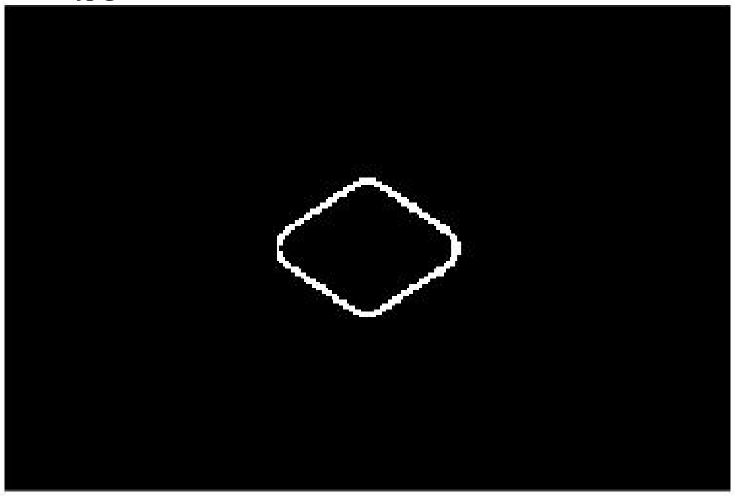
Use the test function **test_tabletEdgeDetection.m** from onQ to test your function.

Your final Sobel images for the three sample tablet images should look like:

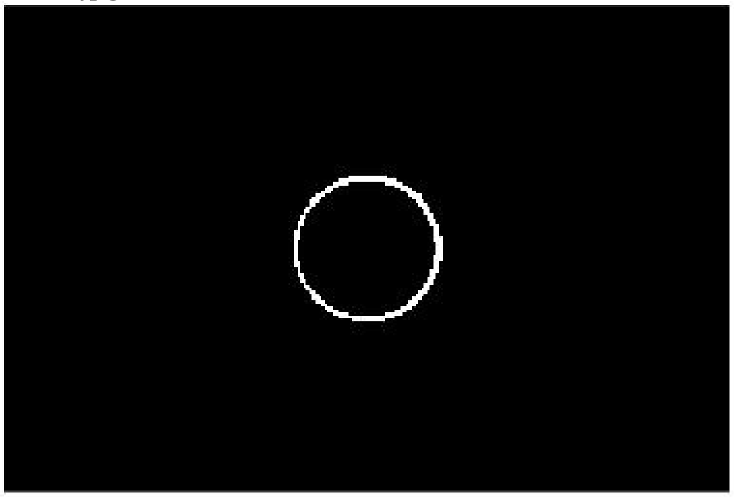
'tablet1.jpg'



'tablet2.jpg'



'tablet3.jpg'



Part B: Submission List

When complete, please submit the following files to the **Lab 6: Part B** dropbox on onQ for marking:

- `imFilter.m`
- `tabletEdgeDetection.m`
- Any other functions you wrote that the above files use (**including the image files**)

[1] U.S. National Library of Medicine, “Pill Image Recognition Challenge: Submission Instructions,” 2016. [Online]. Available: https://pir.nlm.nih.gov/challenge/submission.html. [Accessed: 19-Feb-2019].

[2] United States Food and Drug Administration, “CFR - Code of Federal Regulations Title 21,” *Code of Federal Regulations*, 2018. [Online]. Available: https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfcfr/CFRSearch.cfm?fr=206.10. [Accessed: 19-Feb-2019].

[3] B. Baker, “How to Optimize Display Brightness for Low Power in Real Time,” *DigiKey*, 2018. [Online]. Available: https://www.digikey.dk/en/articles/techzone/2018/jul/how-to-optimize-display-brightness-for-low-power-in-real-time. [Accessed: 19-Feb-2019].

[4] NASA, “Electromagnetic Spectrum - Introduction,” 2013. [Online]. Available: https://imagine.gsfc.nasa.gov/science/toolbox/emspectrum1.html. [Accessed: 19-Feb-2019].

[5] T. I. IceCube Collaboration *et al.*, “Multimessenger observations of a flaring blazar coincident with high-energy neutrino IceCube-170922A.,” *Science*, vol. 361, no. 6398, 2018.

[6] R. C. Gonzalez, R. E. (Richard E. Woods, and S. L. Eddins, *Digital Image processing using MATLAB®*. Gatesmark Publishing, 2009.

[7] M. Cadik, “Perceptual Evaluation of Color-to-Grayscale Image Conversions.” [Online]. Available: http://cadik.posvete.cz/color_to_gray_evaluation/. [Accessed: 19-Feb-2019].

[8] K. F. (Kenneth F. Riley, M. P. (Michael P. Hobson, and S. J. (Stephen J. Bence, *Mathematical methods for physics and engineering*, 3rd ed. Cambridge University Press, 2006.

[9] Apple Inc., “Performing Convolution Operations,” 2016. [Online]. Available: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html. [Accessed: 19-Feb-2019].