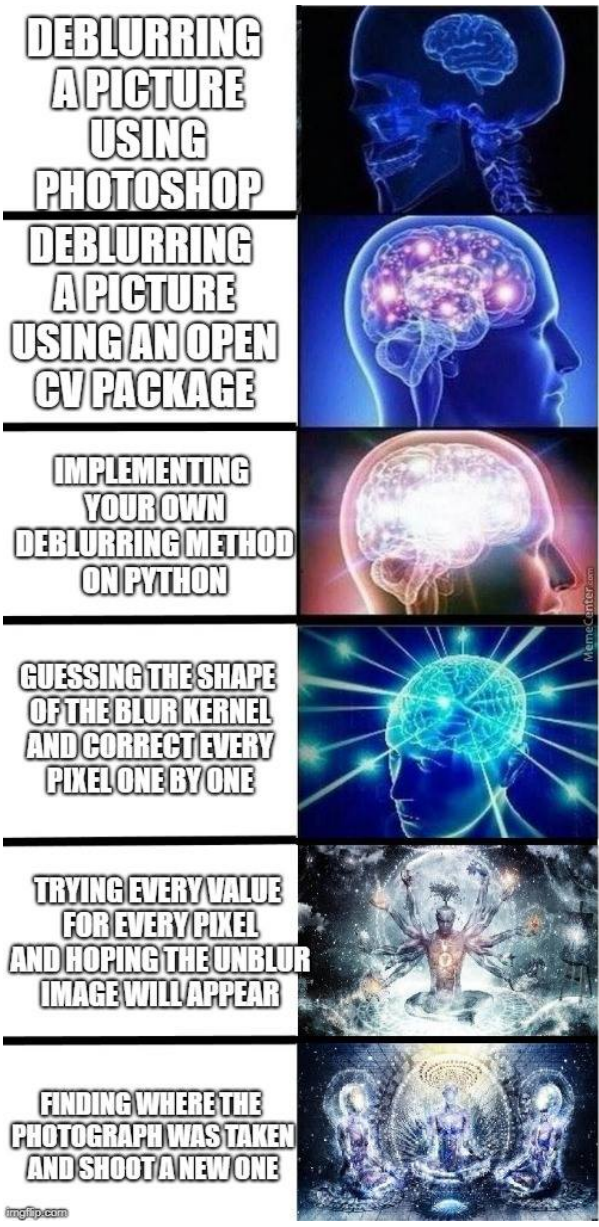
CIS*4720 Image Processing and Vision

Image Toolbox Documentation

Table of Contents

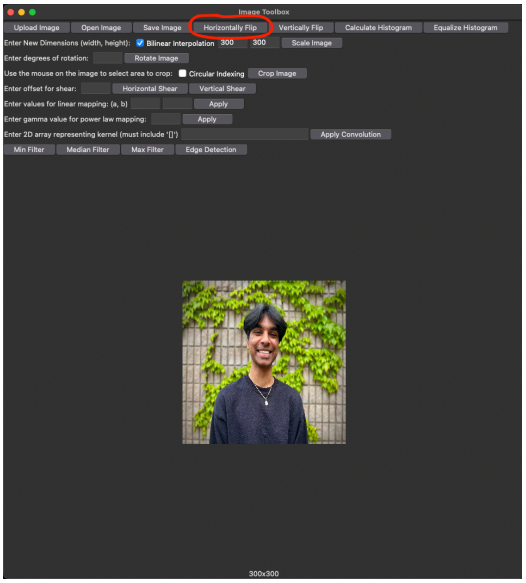


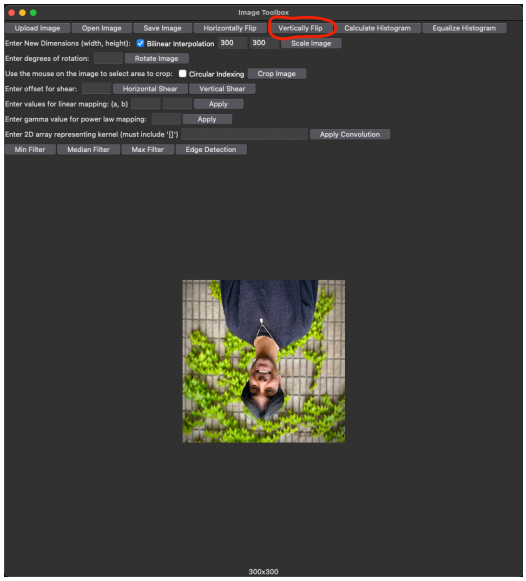
Section 1 - User's Manual

- It is assumed that you have uploaded an image by clicking the top left button named "Upload Image" before using any functionalities
- After uploading and every function used, the image will be automatically updated in the bottom half of the UI
- If necessary, you may click the "Open Image" button to view it in your preferred image viewer in case the UI is too small or insufficient to view the manipulated image

Flipping Images

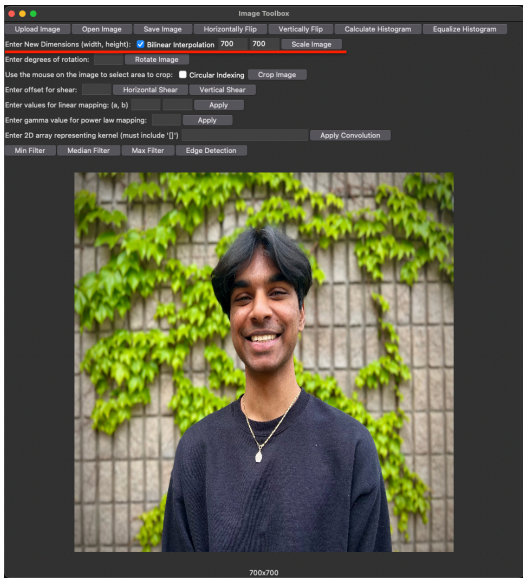
- **Horizontally Flip**
  - You can simply click on the "Horizontally Flip" button in the UI and the resulting image will be flipped in the left-right direction
- **Vertically Flip**
  - You can simply click on the "Vertically Flip" button in the UI and the resulting image will be flipped in the up-down direction

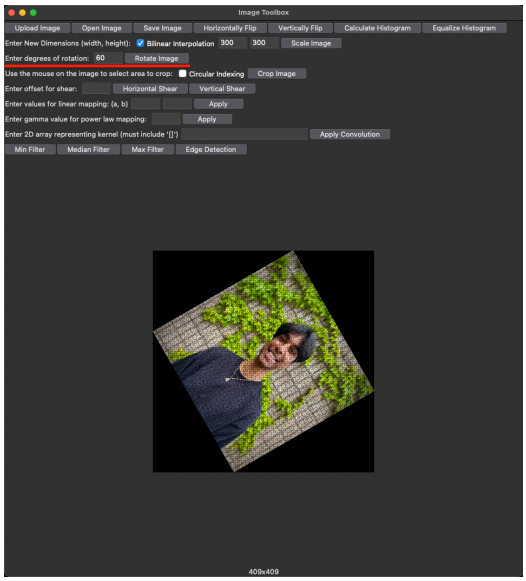




Scale

- You are expected to input two integer values representing the new width and height in pixels then click the "Scale Image" button
- By default, Nearest Neighbour Interpolation will be used, but you may click the checkbox to perform Bilinear Interpolation
- It will not perform other interpolation methods such as Bicubic Interpolation



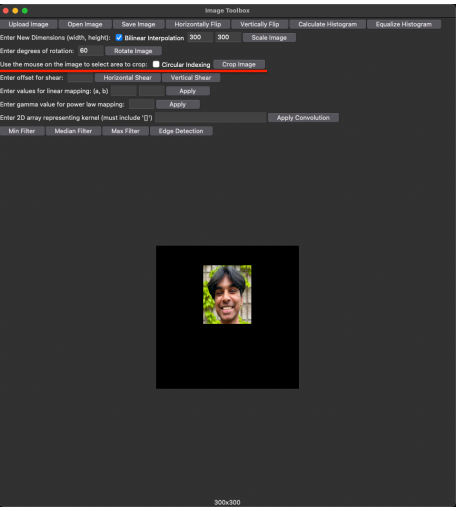


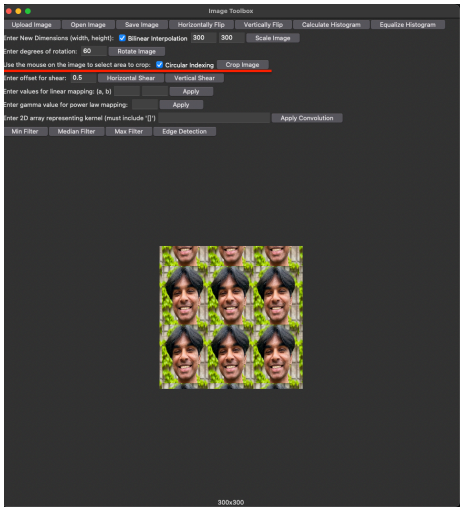
Rotate

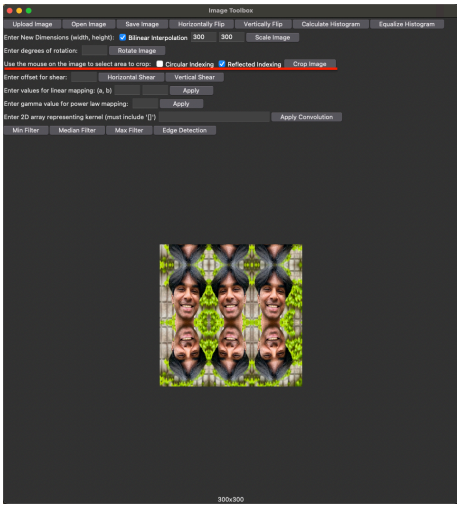
- You are expected to input one integer value representing the degrees that you would like to rotate the image by
- It will perform the necessary rotation as well as zero padding to make sure the image does not get cut off
- This bounding box of zero padding will reset as more rotations occur as to not infinitely make the image larger than necessary

Crop

- Simply hover over the image display area in the UI and use a click and drag method to select the area to be cropped, after which, you may simply click the “Crop Image” button
- This will perform zero padding by default but you may click the checkbox to perform Circular Indexing on the cropped region or Reflected Indexing

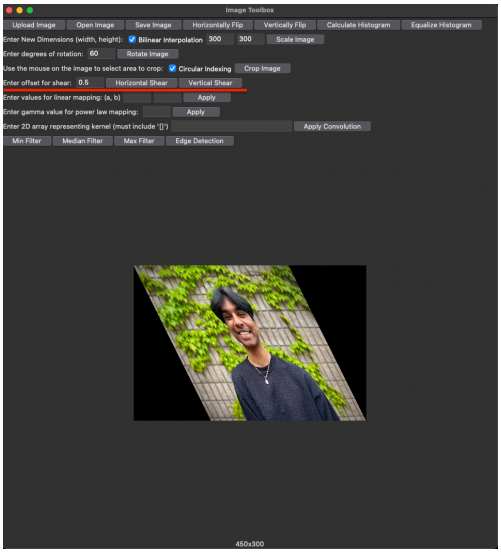


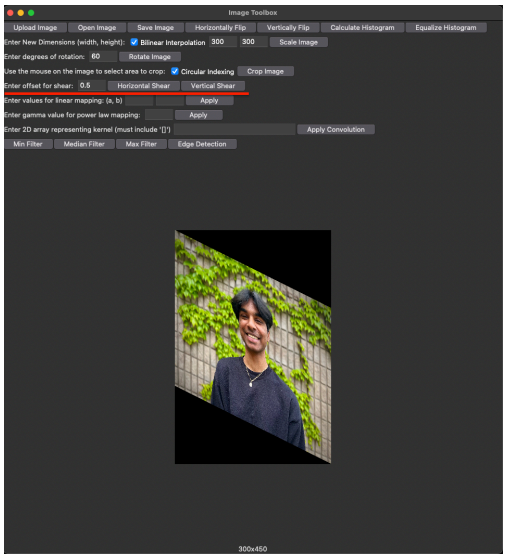




Shearing

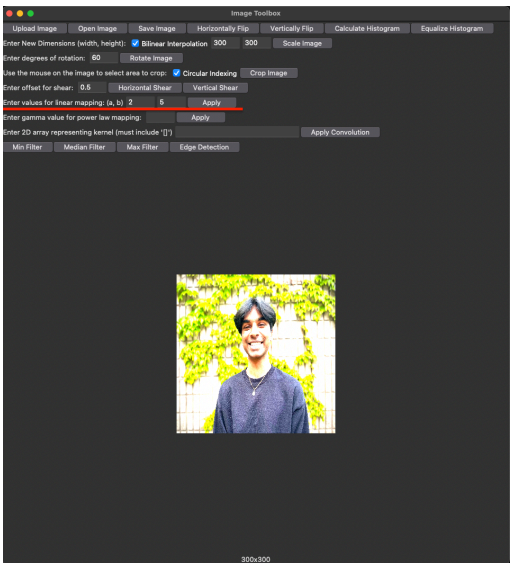
- You are expected to input a decimal or integer value which represents the offset you would like to shear by
- **Horizontal Shear**
  - You can simply click on the “Horizontal Shear” button in the UI and the resulting image will be sheared in the left-right direction
- **Vertical Shear**
  - You can simply click on the “Vertical Shear” button in the UI and the resulting image will be sheared in the up-down direction
- The program will perform zero padding to ensure the resulting image does not get cut off
- It will not perform circular or reflected indexing

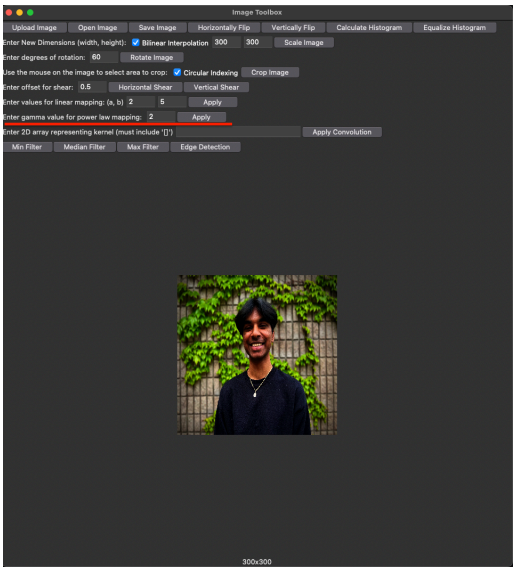




Linear & Power Law Mapping

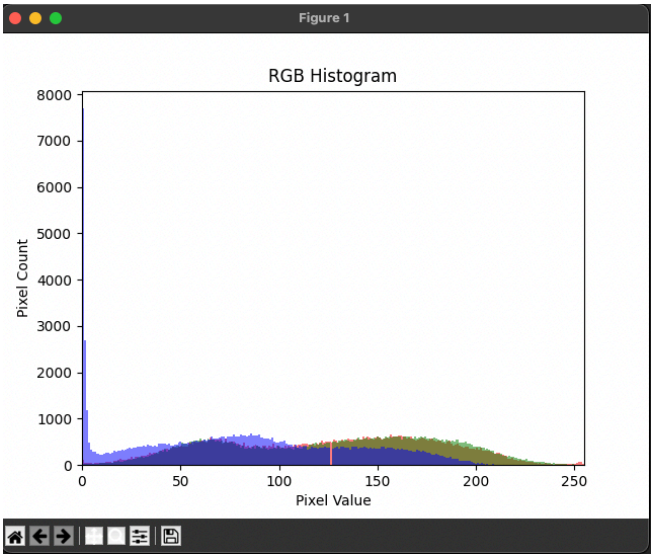
- **Linear Mapping**
  - You are expected to input decimal or integer values which represent “a” and “b” as contrast and brightness respectively
- **Power Law Mapping**
  - You are expected to input a decimal or integer value which represent the gamma
  - When gamma < 1, then the contrast in the darker regions of the image will increase
  - When gamma > 1, then the contrast in the darker regions of the image will decrease
- This will perform the mapping for both RGB and greyscale images
- If values are not entered for any field, it will use a safe default value that does not alter the image in any way

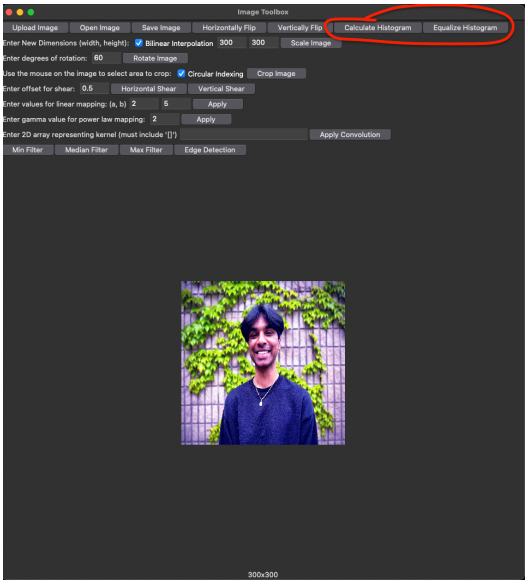


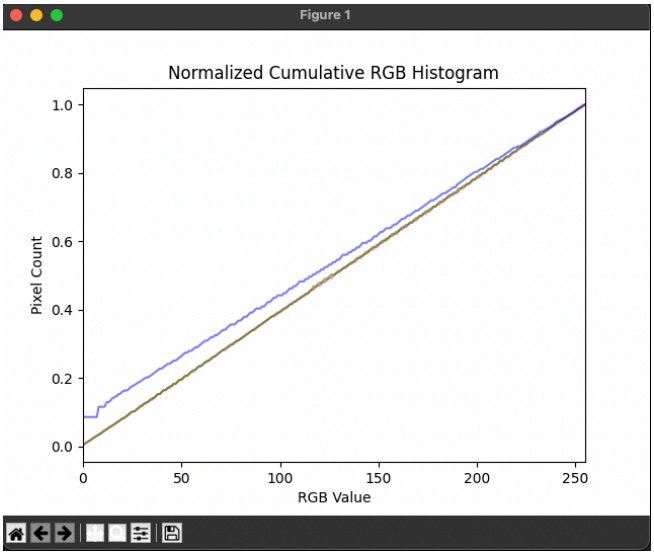


Histograms

- **Generate Histogram**
  - For RGB images, this will generate an RGB histogram which shows the distribution of pixels for each channel
  - For greyscale images, this will generate one channel histogram depicting the distribution of pixels with each grey level from 0-255
- **Equalize Histogram**
  - This will ensure a uniform distribution of pixel values and output the cumulative normalized histogram, after which, it will display the equalized image

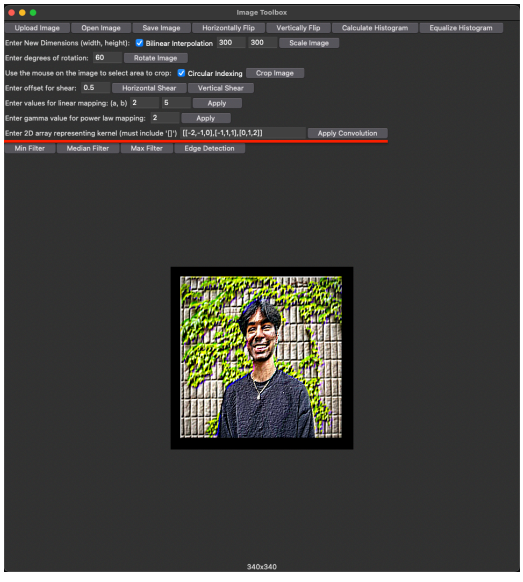






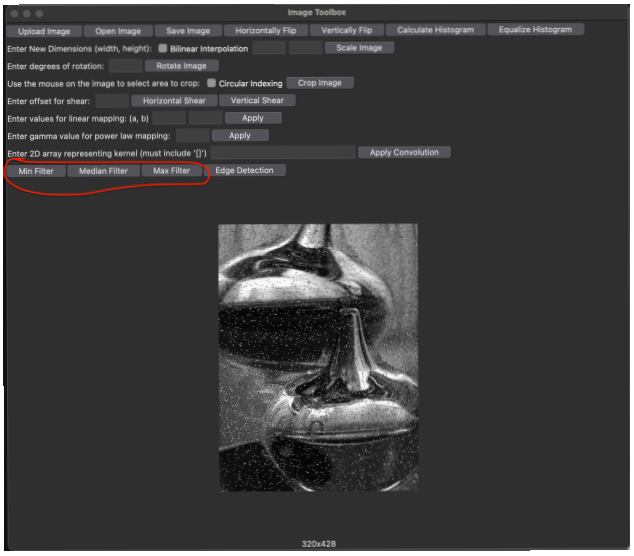
Convolution

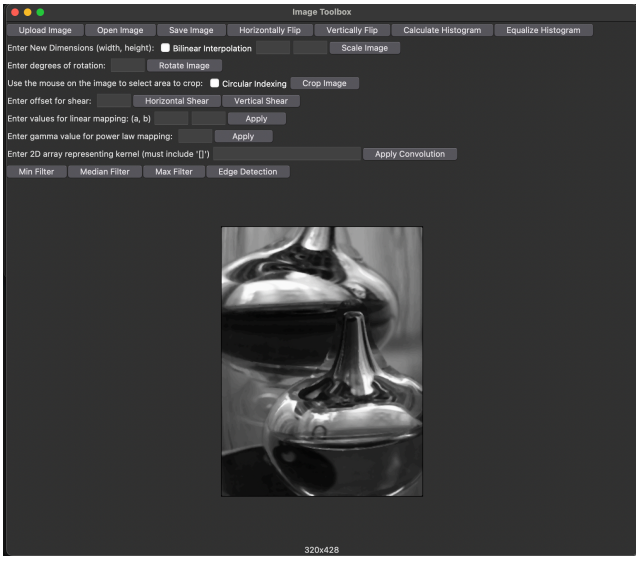
- You are expected to input an odd MxN 2D array consisting of integer values to convolve with the image pixels, as an example: [[1,1,1],[1,1,1],[1,1,1]]
- You may have spaces within the numbers and commas but not between the brackets themselves
- It will perform some zero padding to deal with the borders of the image
- It will not perform circular or reflected indexing for the borders of the image
- It will not accept decimal numbers for the kernel values



Non-Linear Filtering

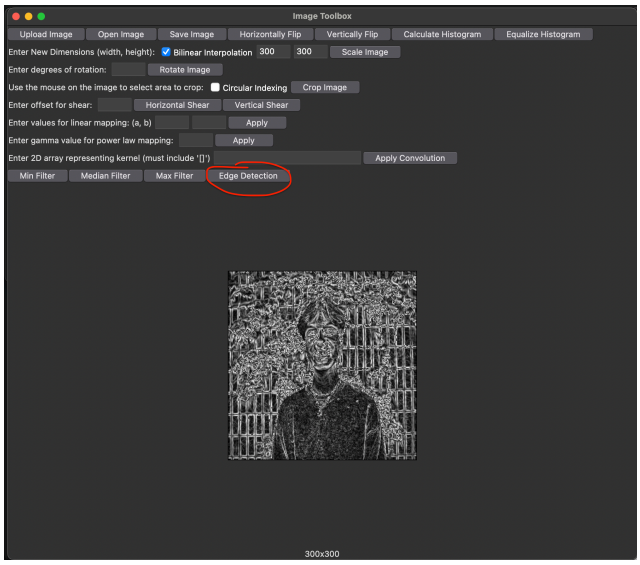
- **Min Filter**
  - This will apply a min filter on both RGB and greyscale images, therefore taking away any salt noise in the subject
- **Median Filter**
  - This will apply a median filter on both RGB and greyscale images, therefore taking away both salt and pepper noise in the subject
- **Max Filter**
  - This will apply a max filter on both RGB and greyscale images, therefore taking away any pepper noise in the subject
- For RGB images, they will most likely have RGB noise which a Median Filter will remove





Edge Detection

- This button, when clicked, will detect all edges in the image and output those edges to the user
- It will perform this for both RGB and greyscale images



Section 2 - Technical Discussion

Flipping Images

- **Horizontally Flip** *def horizontalFlip(self)*
  - This operation loops through the image pixels halfway to the width and swaps the current *(i, j)* pixel with the *(width - i - 1, j)* pixel
  - In other words, we are replacing the current pixel value with the opposite ends pixel value
- **Vertically Flip** *def verticalFlip(self)*
  - Similar to horizontally flipping, this operation loops through the image pixels halfway to the height and swaps the current *(i, j)* pixel with the *(i, height - j - 1)* pixel

Scale *def scale(self, newWidth, newHeight, option)*

- **Nearest Neighbour Interpolation**
  - At the start, I create a new image with the *newWidth* and *newHeight*
  - This operation will then calculate ratios to use as scaling factors which is done by *width / newWidth* and *height / newHeight*
  - After this, we will loop through the new image pixels and calculate the new *(i, j)* pixel by multiplying each pixel by the corresponding ratio, after which, we put the new pixel in that position
- **Bilinear Interpolation**
  - This operation will loop through new dimensions of the image
  - For each new position, calculates its updated position in the original image based on the ratio of the new dimensions to the original dimensions
  - Identifies the nearest four pixels in the original image, i.e., the pixels that form a square around the updated position
  - Calculates the weights or coefficients (alpha and beta) for each pixel based on their distance to the updated position
  - Computes the interpolated value for each colour channel (RGB or greyscale) using the four surrounding pixels and their weights
  - Assigns the interpolated colour value to the corresponding pixel in the new image

- **Bilinear interpolation** (do it for x and y)
  - $\overline{f}(x, y_0) = (1 - s) * f(x_0, y_0) + s * f(x_0 + 1, y_0)$
  - $\overline{f}(x, y_0 + 1) = (1 - s) * f(x_0, y_0 + 1) + s * f(x_0 + 1, y_0 + 1)$
  - $\overline{f}(x, y) = (1 - s) * \overline{f}(x, y_0) + t * \overline{f}(x, y_0 + 1)$
  - $s = x - x_0 \wedge t = y - y_0$

Rotate *def rotate(self, degrees)*

- Since rotating has the potential to cut off the image, the *newWidth* and *newHeight* is calculated using *width * cosTheta + height * sinTheta* and *height * cosThea + width * sinTheta* respectively
- Since we want to rotate the image with respect to its center point, we subtract this point from each location so the center point will be (0,0) before applying the equation
- In the last step, we make sure we only consider the pixels which are in bounds since sometimes, the new coordinate value will go over the size of the original image

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Crop *def crop(self, x, y, w, h, circCrop, reflectCrop)*

- When the user clicks and drags on the image, the first click coordinate is recorded and the last mouse move coordinate is recorded which gives us the start x, start y, end x, end y coordinates without needed to input these values separately
- A rectangle is also drawn onto the image to indicate to the user which part of the image will be cropped
- By default, this operation will perform zero padding on the cropped region which is done by looping through the image, if the pixels are in the start x, start y, end x, and end y coordinates, then we put the same pixels onto the new image
- If the pixels are outside of that which means its the outside of the user selected rectangle, we turn those pixels black
- If the user chose circular indexing, we first calculate the tile width and height based on the user rectangle, then use the formula *(i - x) % (tileSizeWidth) + x* and *(j - y) % (tileSizeHeight) + y* to place the new pixel when it is out of bounds
- If the user chose reflected indexing, we check 6 cases depending on where we are in the image to reflect it properly, these cases are:
  - If i < *x: newI = x - (i - x)*
  - If i >= *w: newI = w + (w - i) - 1*
  - Else*: newI = i*
  - If j < *y: newJ = y - (j - y)*
  - If j >= *h: newJ = h + (h - j) - 1*
  - Else*: newJ = j*
- Then we make sure the reflected index values are within bounds by doing *newI = max(0, min(newI, width - 1))* and *newJ = max(0, min(newJ, height - 1))*

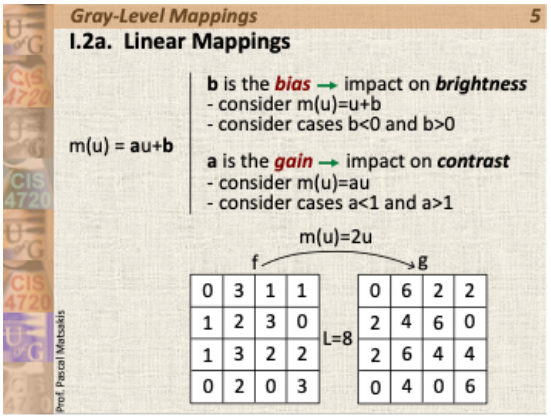
Shearing

- **Horizontal Shear** *def horizontalShear(self, offset)*
  - Since this operation has the potential of cutting off the image width-wise, we will calculate the maximum width based on the user offset which is done by *newWidth = width + abs(int(height * offset))*
  - When we construct our new image, we will have a bounding box consisting of zero padding
  - To perform horizontal shearing, we will loop through the image and use the formula on the width pixels *newI = i + int(j * offset)*

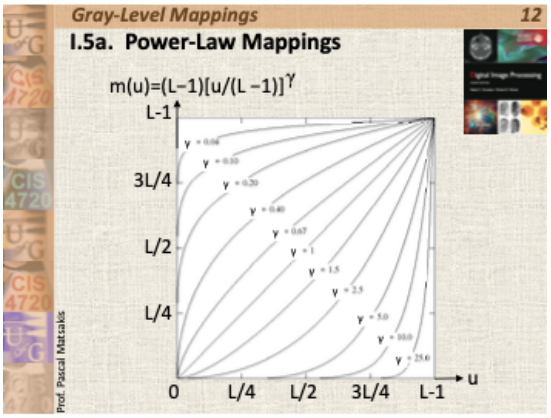
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- **Vertical Shear** *def verticalShear(self, offset)*
  - Similar to horizontal shearing, we will calculate the new height by *newHeight = height + abs(int(width * offset))*
  - To perform vertical shearing, we will loop through the image and use the formula on the height pixels *newJ = j + int(i * offset)*

Linear & Power Law Mapping



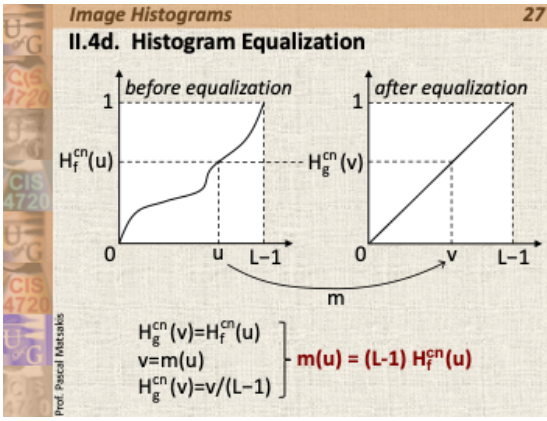
- **Linear Mapping** *def linearMapping(self, a, b)*
  - This operation will first check the values the user inputted and if not inputted, will default to safe values for *a* and *b* which are 1 and 0 respectively
  - Then we will loop through the image and if the image is RGB, we will apply the linear equation to each of the 3 channels and put that pixel into the new image
  - If the image is greyscale, we only apply the equation one time and put that pixel into the new image

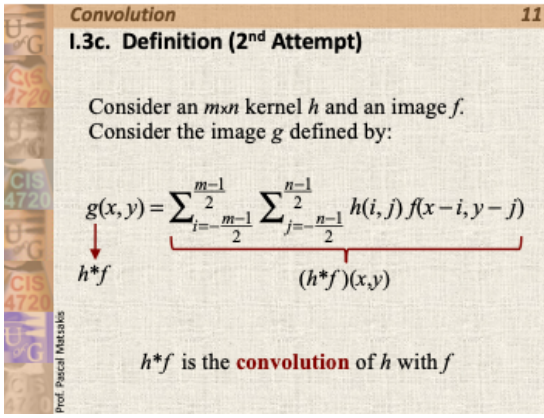


- **Power Law Mapping** *def powerLawMapping(self, gamma)*
  - The user must first enter a gamma value which will default to a safe value of 1 if not inputted
  - Similar to linear mapping, we apply this equation for all 3 channels if the image is RGB and only once for a greyscale image

Histograms

- **Generating a histogram** *def generateHistogram(self)*
  - This operation utilizes Matplotlib to generate the figure with an x axis of pixel values ranging from 0..255 and a y axis of pixel counts after converting the image into an array of pixel values
  - Along with greyscale images, it will generate the histogram for RGB by generating separate histograms for each channel and combining them into one figure that is colour coded by using *numpy.histogram()* and *plt.hist()*
- **Histogram Equalization** *def generateEqualizedHistogram(self)*
  - To equalize our histogram, we first generate a normalized histogram by dividing each value in our histogram array by the total amount of pixels. then generate a cumulative normalized histogram
  - For each pixel, we set a new intensity value which results in equalized pixel values from the equation *m(u) = Hcnf(u) * (L-1)*
  - To show that the histogram was equalized, we generate the new cumulative normalized histogram which will be a linear graph, and we also apply the new equalized pixels to the image which performs a statistical enhancement to it
  - For RGB images, each channel becomes a linear graph instead of just one line

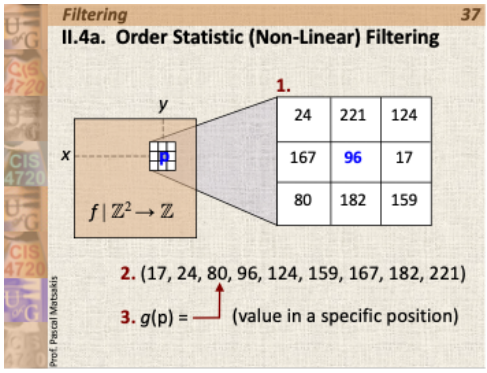




Convolution *def convolution(self, kernel)*

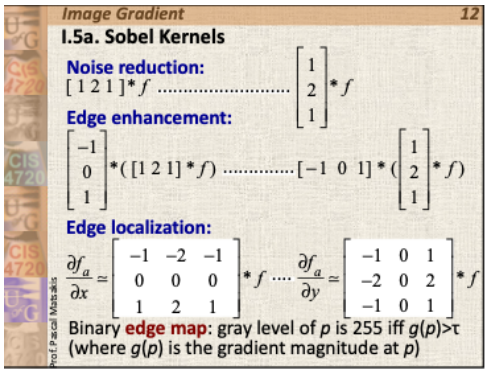
- This operation will first parse the user inputted kernel and also calculate the shape of this kernel
- It also ensures the kernel is an odd kernel to make sure the kernel is valid
- Once error checking is done, it will create a new image with zero padding as to deal with the borders effectively and start looping through this new image
- We reset a pixel array at each iteration of the image since the kernel needs to be applied to each pixel individually
- The next nested loop will loop through the actual kernel itself and calculate each pixel by doing *pixelSum[0] += int(currentPixel[0] * kernel[i][j]* and similar for [1] and [2] pixels
- After iterating through the kernel, it will place this new pixel value in the new image and repeat for the rest

Non-Linear Filtering



- **Min Filter** *def minFilter(self)*
  - We first loop through the image ignoring the edge pixels and then run another nested loop in order to get the neighbourhood pixels (chessboard)
  - We reset a pixel array after each iteration of the image and append to it the neighbourhood pixels, after which, the new pixel we take is the minimum of this pixel array which is of size 9
  - For RGB images, we reset 3 arrays corresponding to each channel and take the minimum of each channel for the new pixel value
- **Median Filter** *def medianFilter(self)*
  - The only difference with this from Min Filter, is that once we get the neighbourhood pixels, we sort this array of 9 values and take the middle element for the new pixel value
  - For RGB images, we take the middle element for each channel after sorting each
- **Max Filter** *def maxFilter(self)*
  - The only difference with this from the other filters, is that once we get the neighbourhood pixels, we take the max value and use this for the new pixel value
  - For RGB images, we take the max element for each channel

Edge Detection *def edgeDetection(self)*



- For detecting edges, we utilize the *Sobel Kernels*
- After converting the image to greyscale, we loop through the image and truncate the borders
- For each pixel, we are calculating the gradient sums for x and y by doing *numpy.sum(kernelX * imageArr[i - 1: i+ 2, j - 1 : j + 2]* and *numpy.sum(kernelY * imageArr[i - 1 : i + 2, j - 1 : j + 2]*
- Then we calculate the gradient magnitude by doing *numpy.sqrt(gradientX^2 + gradientY^2)*
- This magnitude is what we use for the new pixel value

Section 3 - Discussion of Results and Future Work

- **Testing**
  - I used a variety of images to test, some being greyscale, RGB, JPEG formats, PNG formats, and more
  - For most operations, I used a built in function that performed the same operation and compared the results with my own implementation
    - For example, .flip(), .scale(), .rotate(), etc, were all pre-existent methods that I used to compare with
    - There were also methods such as .convolve(), .filter() to test my implementations of most filtering operations
  - For Non-Linear filtering, I used images that contained salt and pepper noise to make sure the filters got rid of them and for RGB images, chose one with RGB noise
  - To make sure my histogram was equalized correctly, I made sure to generate the cumulative normalized histogram for the image and ensured it was a straight line
- **Strengths**
  - My program performs all necessary image processing functionalities without any major bugs in plus extra features
    - These extra features include:
      - Circular and Reflected indexing on cropping
      - RGB image processing in addition to greyscale
      - Bilinear Interpolation on scale
      - An easy to use Graphical User Interface
      - Additional geometric spatial transformations such as Shearing
      - Edge Detection using the Sobel kernels
      - An intuitive an easy to use cropping interface
      - Live image updates with respect to operations using lambda functions
- **Weaknesses**
  - When rotating an image, the image will have some aliasing which happens due to multiplying by sin and cos on the integer values then rounding these real numbers in order to put the pixel value
  - Convolution currently does not work with decimal values
  - Larger images take more time to do complex operations such as edge detection and other filtering methods
- **Future Work**
  - In the future, I would add the following functionalities:
    - Bicubic Interpolation on scale
    - Circular and Reflected indexing on rotation and shearing
    - An interactive website in addition to a desktop application
    - Edge detection using Laplacian method
    - Option for users to see separate normalized and cumulative histograms
    - Advanced image processing functions such as Object Recognition and Detection
    - Refactor certain operations to be more efficient as to take less time on larger images