KENNESAW STATE
UNIVERSITY

CS 4732
MACHINE VISION

PROJECT 2

IMAGE ENHANCEMENT

INSTRUCTOR

Dr. Mahmut KARAKAYA

Michael Rizig

001008703

1. ABSTRACT

In this project, we are given 3 tasks to complete, image transformations, histogram equalization, and noise reduction. For transformations, we use log and power transformations to expand a narrow contrast range, and experiment with the gamma values 1, 1.5 and 2.2 on each power transformation to see how they affect the result. We found that different combinations of log constants and gamma values gave varying results on image clarity and balance in the contrast. For histogram equalization, we create a function based on the image's histogram distribution and attempt to normalize the histogram from a smaller spikelike shape to a even distribution. We found that transforming the data via division and manipulating the b values allowed us to find nearly perfect formulas to more equally distribute the histograms, leading to a much more balanced image. For the HSI color model, we utilized histogram equalization on the I element, and found that this method can help correct contrast issues in greyscale images and color images. Finally for noise reduction, we try both average and median filtering in 3x3, 5x5, and 7x7 filter sizes and compare the results from each run to determine which size best fits our input image. We found that in general, the larger the filter size, the greater the blur effect. We also found that median filtering works better to counter salt and pepper noise without gaining too much blur in the process.

To view all edits, changes, and see step by step revision history, view this project on my GitHub:

https://github.com/michaelrzg/CS4732-Projects

2. TEST RESULTS

2.1 Log transformation.

(Only a few selected images are used here to highlight the effect. All output images can be found in output>log folder in the zip submission)

For the log transformation, we used the formula found in the slides:

$$T(x) = y = \log(1+x) * c$$

in these runs we experimented with the c values: {10,20,30,50,70,90,110,120,150 } and found varying results.

Image 1a: Original Image 'univeristy.png. **This is the input image.**

Image 1b: y = log(1+x) * 20. Constant =20; we can see that the image actually gets a bit darker since our constant is too small.

İmage 1c: y = log(1+x) * 30. Constant =30; bumping up our c, we see slightly more details under the garbanzo and in shadier parts of our image.

İmage 1d: y = log(1+x) * 50. Constant =50; at this constant, we can see a lot more detail as compared to our input image.

Image 1e: y = log(1+x) * 70. Constant =70; this seems to be the best constant for our image, as it allows us to brighten the darker areas without completely washing out the whites.

Image 1f: = y = log(1+x) * 90. Constant =90; this constant seems too bright, giving the image a washed out look. Each run above this constant lead to even more washed out results.

(a)

(b)

(c)

(d)

(e)

(f)

Figure 1: (a) Original image (input/university.png), (b) y = log(1+x) * 20(output/log/logConstant-20.png), (c) y = log(1+x) * 30(output/log/logConstant-30.png) ,(d) y = log(1+x) * 50(output/log/logConstant-50.png), (e) y = log(1+x) * 70(output/log/logConstant-70.png), (f) y = log(1+x) * 90(output/log/logConstant-90.png)

2.2 Power transformation.

(Only a few selected images are used here to highlight the effect. All output images can be found in output>log folder in the zip submission)

For the power transformation, we used the formula found in the slides:

$$T(x) = y = c * x^{y} = c * x^{(y/gamma)}$$

For the power transformation, we found that values close to or above 1 darkened the image and values below 1 brightened the image, so we opted to continuously decrease the y value. Since our r^{y} term returns a value between (0,1), we used the constant 255 to scale it to our greyscale image. We utilize the y values {.9,.8,.7,.6,.5,.4) for each pass.
For gamma we divide our power by each gamma value within the range, {1,1.5,2.2}.

Image 2a: We begin with the original university.png. **This is the input image.**

Image 2b: y=.9 gamma = 1. We see once again that our image actually gets darker, meaning we need to lower our y value.

İmage 2c: y=.8, gamma =1. This image looks pretty similar to our input, no meaningful improvement.

İmage 2d: y=.6, gamma =1. This is where we start to see considerable improvement in our contrast. Our darkest points stay dark while comparatively lighter points scale up.

Image 2e: y=.4, gamma=1. At this point, the whites begin to get washed out, and the darker points lose their darkness, we know that at this gamma we've gone too low with our y value.

Image 2f: y=.9 gamma =1.5. Starting over and moving on to our next gamma, at .9 we see that our contrast is already dramatically changed for the better compared to gamma =1 at y=.9.

Image 2g: y=.8 gamma =1.5. At the next step, we see that the contrasts improve still, once again making a larger jump than our previous .8 at 1 gamma. This image looks the best in my opinion.

Image 2h: y=.6 gamma = 1.5; this step bumps the contrast up again, with the whites beginning to be overpowering.

Image 2i: y=.4 gamma = 1.5; At this point the contrast is way too bright, and the blacks are completely washed out.

Image 2j: y=.9 gamma = 2.2. We move on to the final gamma value 2.2, and see the same trend as before, with the image increasing in brightness dramatically with each step.

Image 2k: y=.8 gamma = 2.2; at .6 the image is as bright as .4 with the 1.5 gamma, meaning the curve for this graph would have a higher slope.

Image 2l: y=.6 gamma = 2.2; at this y we can clearly see the image is completely washed out, meaning we have gone way too far with our y value with this gamma.

Image 2m: y=.4 gamma = 2.2; once again we are completely washed out, and the image is unusable

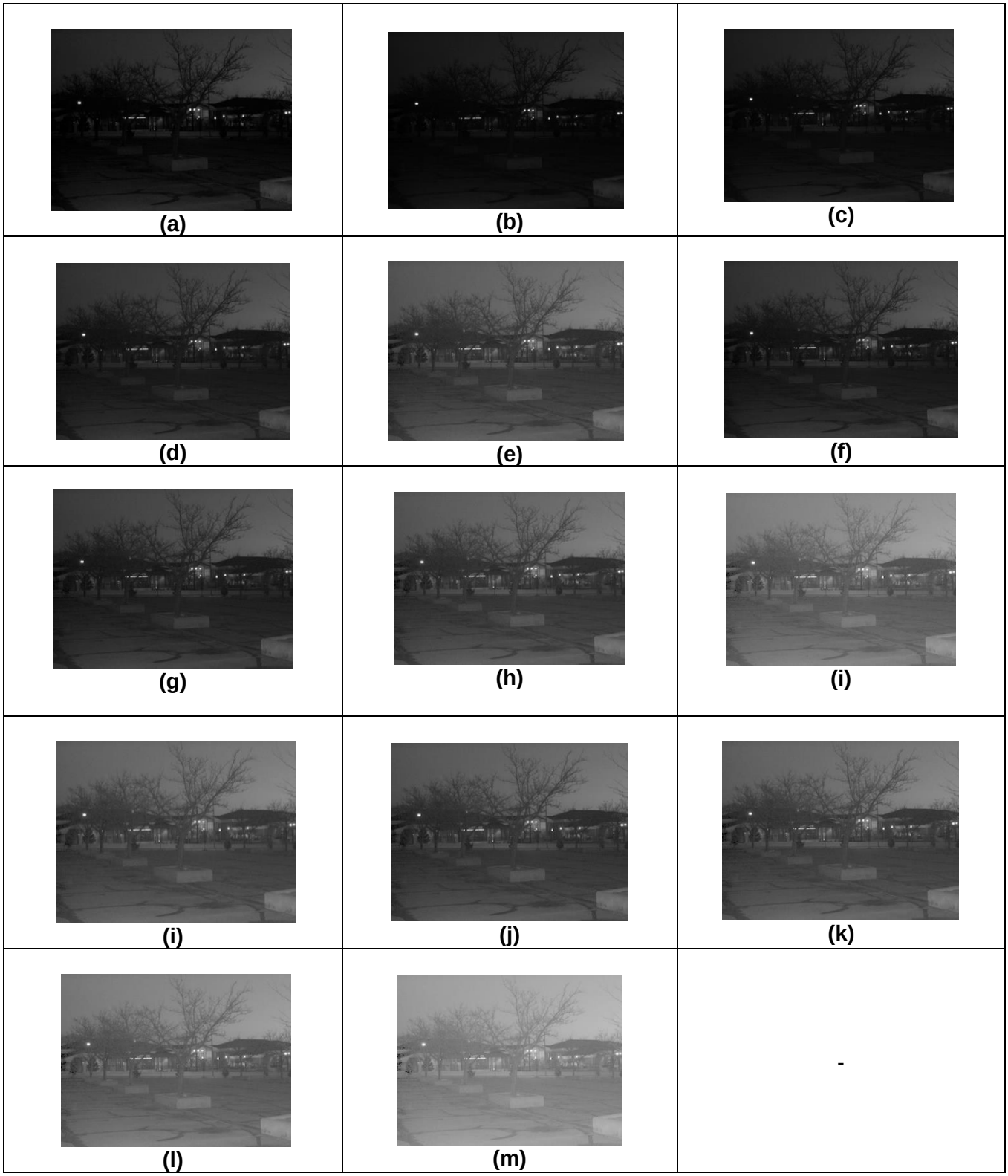


Figure 2: (a) original image (input/university.png), **(b)** y=.9 gamma=1 (output/power/gamma-1/yValue-0.9.png), **(c)** y=.8 gamma=1 (output/power/gamma-1/yValue-0.8.png), **(d)** y=.6 gamma=1 (output/power/gamma-1/yValue-0.6.png), **(e)** y=.4 gamma=1 (output/power/gamma-1/yValue-0.4.png), **(f)** y=.9 gamma=1.5 (output/power/gamma-1.5/yValue-0.9.png), (g) y=.8 gamma=1.5 (output/power/gamma-1.5/yValue-0.8.png), **(h)** y=.6 gamma=1.5 (output/power/gamma-1.5/yValue-0.6.png), **(i)** y=.4 gamma=1.5 (output/power/gamma-1.5/yValue-0.4.png), **(j)** y=.9 gamma=2.2 (output/power/gamma-2.2/yValue-0.9.png), **(k)** y=.8 gamma=2.2 (output/power/gamma-2.2/yValue-0.8.png), **(l)** y=.6 gamma=2.2 (output/power/gamma-2.2/yValue-0.6.png), **(m)** y=.4 gamma=2.2 (output/power/gamma-2.2/yValue-0.4.png)

2.3 Histogram Equalization (greyscale)

For the histogram equalization on the university.jpg image, we start by calculating the original image's histogram and observing the trends. Figure 3b shows this histogram. We notice that the distribution of data is focused around x=5 with a range of (0,50). To scale this to (0,255), all we need to do is create a function to convert each pixel in the original range to the new range. This can simply be done by converting the original range to a scale from 0 to 1, then multiplying that result by the range we want (255-0=255). So for each pixel we divide its grey-level by 50, then multiply that quotient by 255 to get our resulting image.

Image 3a: We begin with the original university.png. **This is the input image.**

Image 3b: This is our original image's histogram, computed via cv2's calcHist function and displayed via matplotlib.pyplot's plot function.

Image 3c: This is our output image after equalization. We can see that the image has an increased contrast ratio. We also see that histogram eq preserves our darker areas better than the log or power transformations did.

Image 3d: This is the histogram of our output image. We can see that while we still have a sharp spike, our values now span the entire range of the 8bit greyscale, rather than just (0,50).

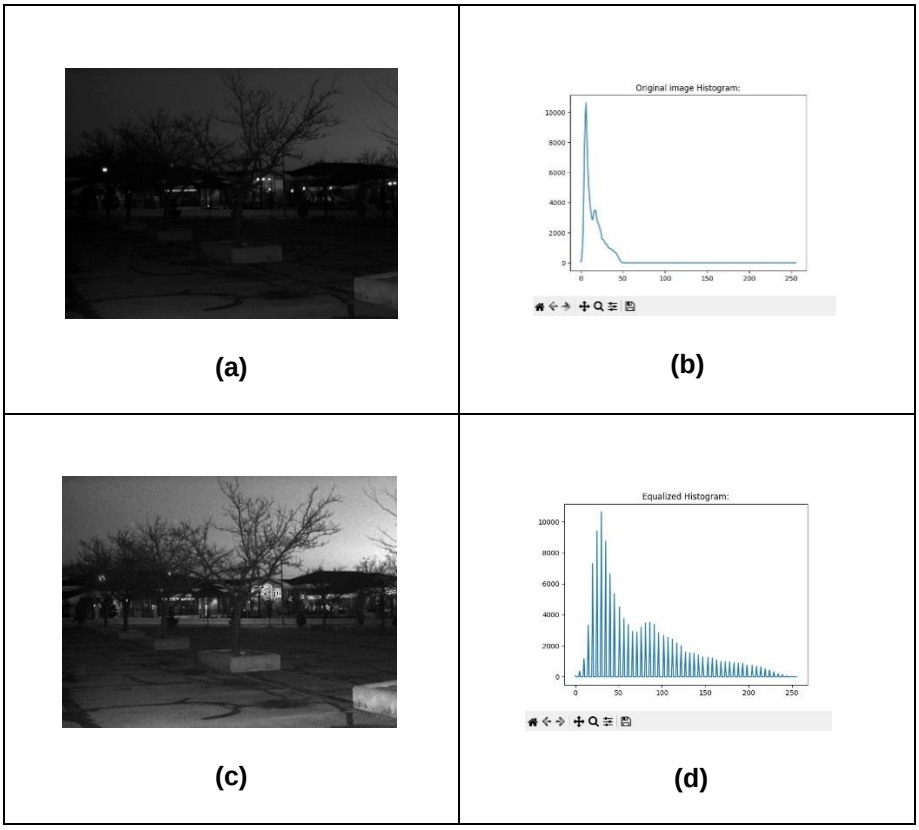


Figure 3: (a) Input image (input/university.png), **(b)** histogram of original image's grey-level distribution (output/hist/university/histogramBefore.jpeg), (c) The output university image after equalization (output/hist/university/uniEqualized.png), (d) histogram of original image's grey-level distribution after equalization (output/hist/university/histogramAfter.jpg)

2.4 Histogram Equalization (RGB)

For the histogram equalization for RGB on the university.jpg image, we start by calculating the original image's histogram for each color channel and observing the trends. We see that all three colors have a very similar original histogram, so the same equalization function should work for all three. Since our graph's focus range does not begin with 0 but rather 150, we need to do a little manipulation before equalization. We start by thresholding each pixel with some simple logic to determine if the pixel fits in our range. We simply check if a given pixel is below 150, and if so set that pixel to 0. If the pixel is greater than or equal to 150, we simply subtract the pixel's value from 150. This effectively moves our lower point of our range from 150 to 0, and makes our graph look similar to the previous greyscale histogram input graph. Now all we must do is divide each pixel by the length of the range (about 65) and scale it by 255 to get our equalized value. I've combined these two steps into a single function that runs on each pixel. These are the results:

Image 4a: We begin with the original sat_map.png. **This is the input image.**

Image 4b: This histogram plots the images B values, as we can see they are not equalized.

Image 4c: This histogram plots the images B values after equalization. We can see the values now fit the range better.

Image 4d: This is the original sat_map.png image with the B values adjusted. As we can see, the colors are now mainly blue, but as we continue adjusting channels the image will look much more viewable.

Image 4e: This histogram plots the images G values, as we can see they are not equalized.

Image 4f: This histogram plots the images G values after equalization. We can see the values now fit the range better.

Image 4g: This is the original sat_map.png image with the B and G values adjusted. As we can see, now that the green is adjusted, the red values which range from 150-215 are overpowering the image, and the image seems to be much more red.

Image 4h: This histogram plots the images R values, as we can see they are not equalized.

Image 4i: This histogram plots the images R values after equalization. We can see the values now fit the range better.

Image 4j: This is the original sat_map.png image with all 3 channels adjusted. We see that now the red levels are normalized, and the image looks as you would expect and colored satellite image to appear.

Image 4a: original satellite image (input/sat_map.png)

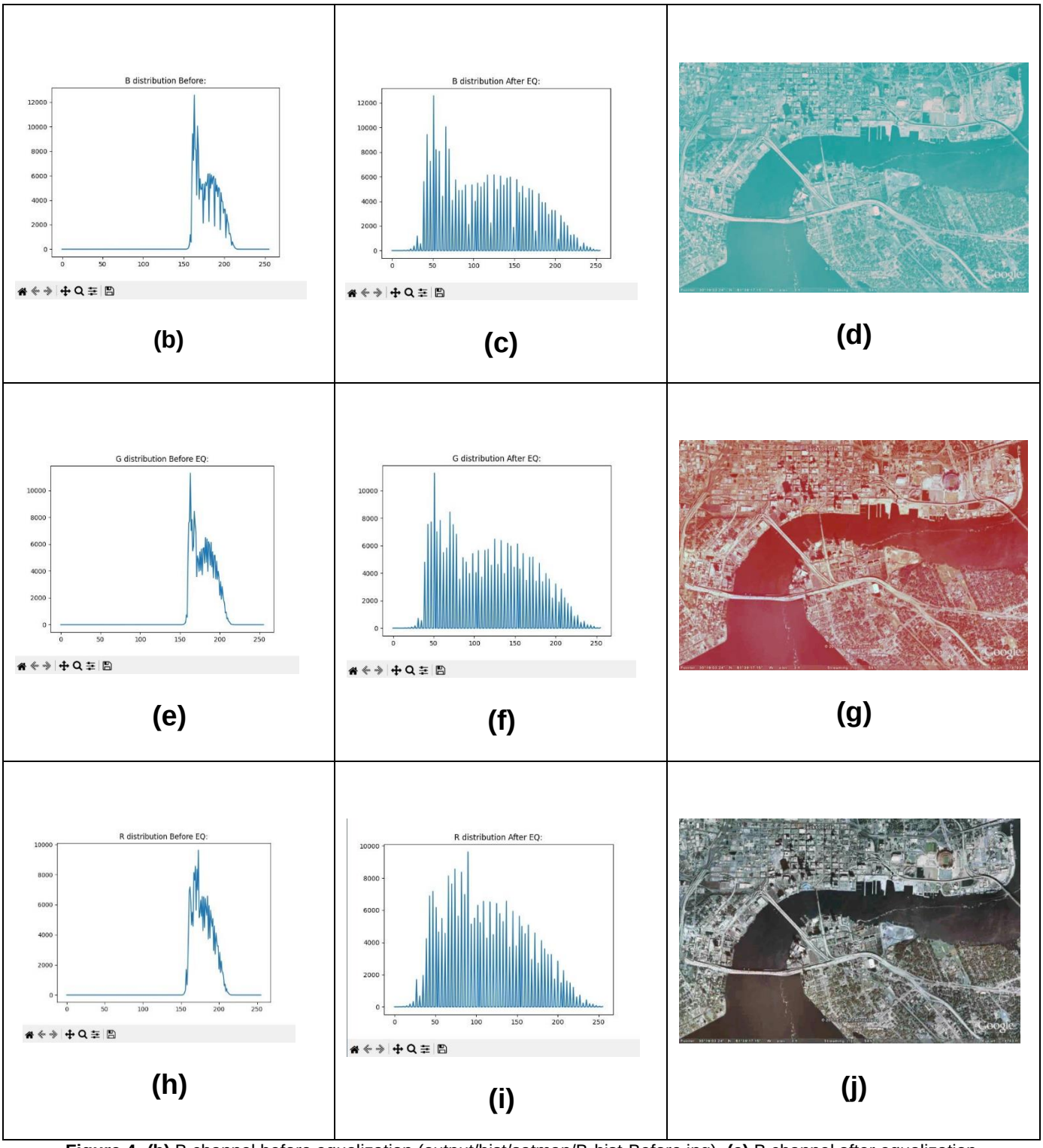


Figure 4: (b) B channel before equalization (output/hist/satmap/B-hist-Before.jpg), **(c)** B channel after equalization (output/hist/satmap/B-hist-After.jpg), **(d)** satmap image after B channel correction (output/hist/satmap/B-corrected.png) **(e)** G channel before equalization (output/hist/satmap/G-hist-Before.jpg), **(f)** G channel after equalization (output/hist/satmap/G-hist-after.jpg), **(g)** satmap image after B and G correction (output/hist/satmap/G-corrected.png),**(h)** R channel before equalization (output/hist/satmap/R-hist-Before.jpg), **(i)** R channel after equalization (output/hist/satmap/R-hist-after.jpg), **(j)** satmap after all 3 corrected (output/hist/satmap/R-Corrected.png)

2.5 Histogram Equalization (HSI)

The HSI color model, or Hue, Saturation, and Intensity model is another color model that is useful for color description and intensity. For this equalization, we follow the same steps as the RGB, but instead of manipulating the R, G, and B values, we covert the color model from RGB to HSI, and manipulate the I, or Intensity value, which in essence is simply the average of the R,G,B values of each pixel. We normalize the I histogram, and convert the image back to rgb. Doing this, we found that the color intensity and contrast issues of the original satmap image are corrected.

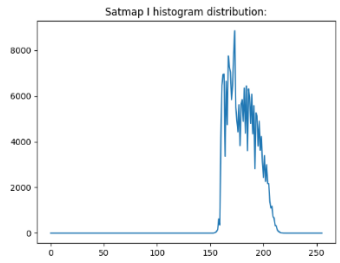
Image 5a: We begin with the original sat_map.png. **This is the input image.**

Image 5b: This histogram plots the images I values (Intensity = 1/3 (R+G+B)) and as we can see they are not equalized.

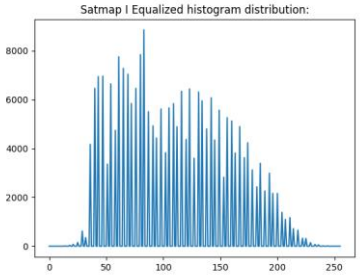
Image 5c: This histogram plots the images I values after equalization, utilizing the same equalization function as 2.4. We can see the values now fit the range better.

Image 5d: This is the original sat_map.png image converted back to RGB with the I values adjusted. As we can see, the image seems more saturated, and the blacks look more intense. We also see more contrast in the lighter vs the darker areas of the image.

(a)



(b)



(c)

(d)

Figure 5: (a) original sat-map image (input/sat_map.png), **(b)** original image's histogram distribution for I value (output/hist/HSI/OriginalHist.png), **(c)** normalized histogram distribution for I value (output/hist/HSI/CorrectedHist.png), corrected image converted back to rgb (output/hist/HSI/correctedSatMap.png)

2.6 Noise Reduction (Average Filtering)

The concept of average filtering is pretty simple. We take a given filter size and pass that sized matrix over each 'target' image pixel. We then average every pixel in the filter and use that average value as our target pixels value. This technique is one way to deal with noise reduction, but as you will see in the next page, median filtering performs better. We try filter sizes 3x3, 5x5, and 7x7

Image 6a: We begin with the original noisy_atrium.png. **This is the input image.**

Image 6b: Filter size: 3x3. At this size we don't see too much blur effect, but we see that the salt and pepper is not eliminated, only slightly reduced.

Image 6c: Filter size: 5x5. At this size we see more noticeable blur effect, and we see that the salt and pepper noise is not really affected much more than before, but it is more blurred into the background.

Image 6d: Filter size: 7x7. At this size we see considerable blurring, and the noise is blended into the image. Overall, we see that this method did not work very well.

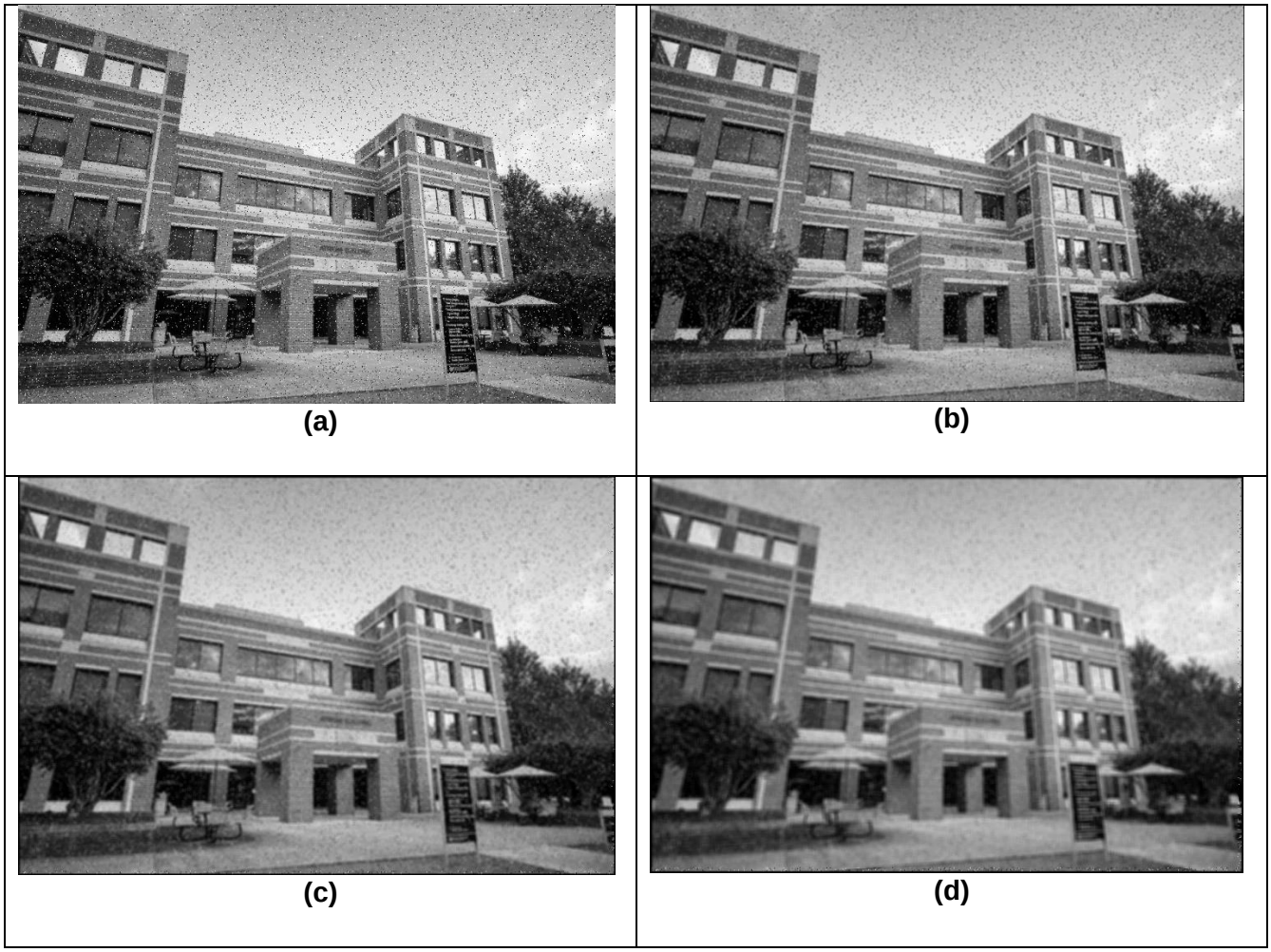


Figure 6: (a) Original Input Image with salt and pepper noise (input/noisy_atrium.png),**(b)**Filter size:3x3. We see increase in blue with little to no noise reduction (output/reduction/average/filtersize-3.png), **(c)** Filter size:5x5. We see significant increase in blur with some noise reduction (output/reduction/average/filtersize-5.png)**(d)** Filter size: 7x7. We see significant increase in blur with some very noticeable noise reduction, but at the heavy cost of image clarity (output/reduction/average/filtersize-7.png)

2.7 Noise Reduction (Median Filtering)

The concept of median filtering is not too different from average filtering but has one key difference. As before, we take a given filter size and pass that size filter over each 'target' pixel. The difference is, instead of averaging, we take the median (technically just another type of average) and use that for the target pixel instead. As you can see from the results below, this method works better than Average filtering.

Image 7a: We begin with the original noisy_atrium.png. **This is the input image.**

Image 7b: Filter size: 3x3. At this size we don't see too much blur effect, but we see that the salt and pepper noise is essentially eliminated in this one pass besides a handful of outliers.

Image 7c: Filter size: 5x5. At this size we see more noticeable blur effect, but the noise is completely gone.

Image 7d: Filter size: 7x7. At this size we only see an increase in blur with no noticeable effect with noise reduction. At this point it is clear that we should have stopped at size 3x3 or 5x5.

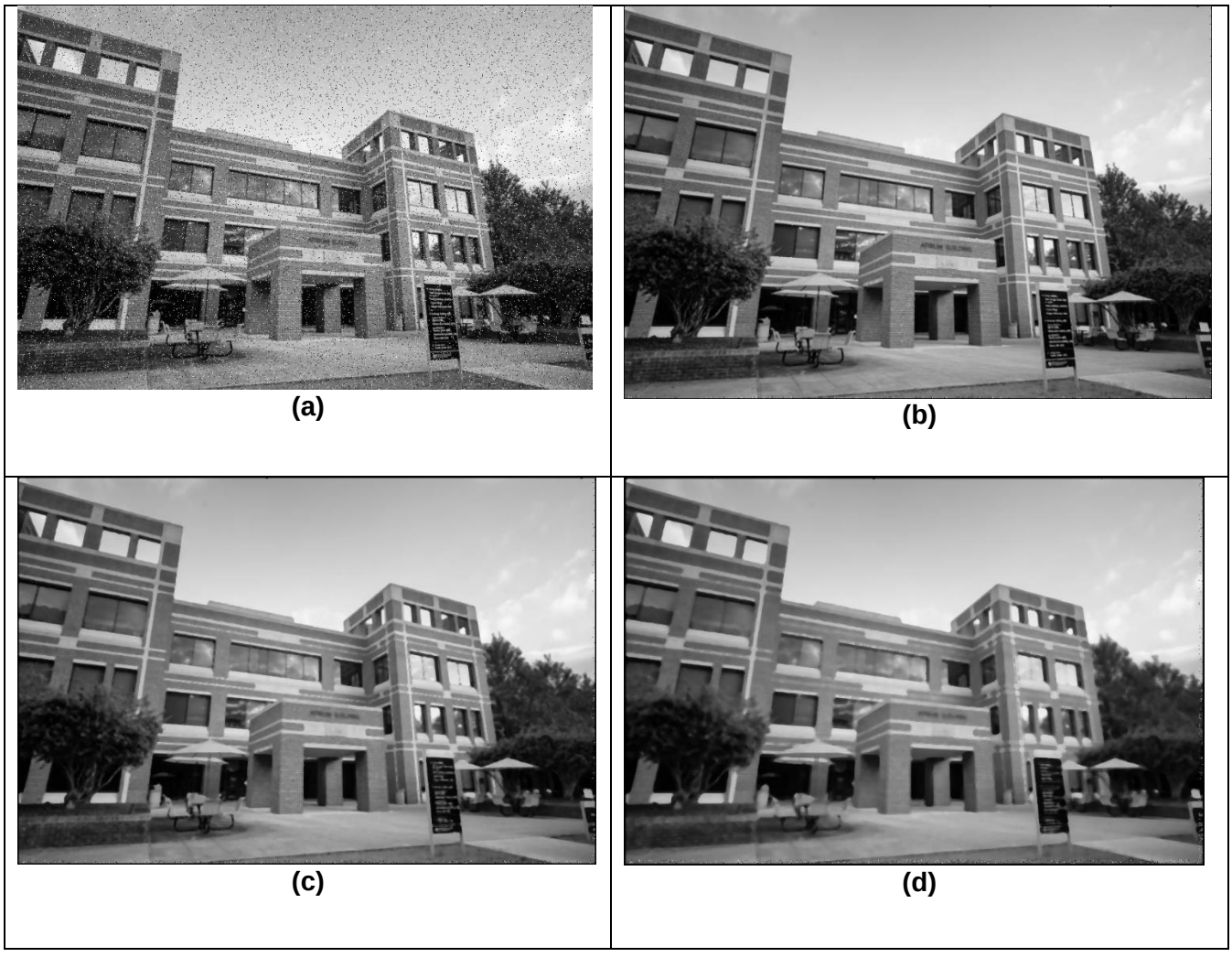


Figure 7: (a) Original Input Image with salt and pepper noise (input/noisy_atrium.png),**(b)**Filter size:3x3. We see a very slight increase in blur but a significant reduction in noise(output/reduction/median/filtersize-3.png), **(c)** Filter size:5x5. We see slightly more blur but basically no noise(output/reduction/median /filtersize-5.png)**(d)** Filter size: 7x7. We see noticeable blue around sharp edges in the image, but the noise is completely removed. (output/reduction/median/filtersize-7.png)

2.8 Discussion

In this project, we tackled log and power transformers, histogram manipulation and equalization for both greyscale and color images, and noise reduction. For log transformations, we learned that varying the constant factor can lead to a large range of results, and finding the right factor for an image is important. In the power transformations, we found that the in general, lower the y power is, the brighter the image is. By using 255 as our scaling constant, and dividing our power by several gamma values, we found that the gamma plays a large role in how the contrast of the image is perceived. In the greyscale histogram equalization, we found that by using a formula to normalize the data into a range (0,1) and scaling it to 8-bit greyscale (0,255), we can take an image and expand its contrast range to better see details in the image. We also saw that in the color equalization, by manipulating the histograms for each color then equalizing the colors for the satmap image we went from an almost greyscale looking washed out image, to a full color satellite image you would expect to see on google maps. Finally, in the noise reduction, we learned that in general as we increased our filter size, we see better noise reduction but a significant increase in blur effect each step up. We also see that the median filtering is much better at handling noise reduction than average filtering, with the 3x3 median filter being more effective than even the 7x7 average filter while minimizing how much blur we take on. Given more time, it would be nice to test how these filtering and histogram techniques can be used on a wider variety of images and how it can be applied to real world projects.

3. CODES

3.1 Code for log and power transformations (Transformations.py)

```
# Michael Rizig
# Project 2: Image Enhancement
# 001008703
# File 1: Transformation
# 6/11/2024

#nessessary imports
from skimage import io
import matplotlib.pyplot as plottool
import cv2
import math

#define image path
uniPath = 'input/university.png'

#create image object with skimage
image = io.imread(uniPath)

#color correction
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)

#define an output image as copy of input
out = image.copy()
```

```
# display input image
plottool.imshow(image)
plottool.show()

# log transformation with varying constant factors:
constant = [20,30,50,70,90,110]
#outer loop goes through each log constant
for k in range(6):

    # loop through each image pixel
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            #using the formula found in the slides :
            # s = T(r) = c * Log(1 + r) where r = greylevel
            # along with varying constants, we can see how the log transformation
maps the narrow
            # range found in this image to a wider range
            f = math.log10(1+ image[i][j][1]) *constant[k]

            #set pixel to this grey value
            out[i][j] = [f,f,f]
    #display results for each log run
    plottool.imshow(out)
    plottool.title(f'y = log(1+ x) * {constant[k]}')
    plottool.show()

    #save results with informative name
    cv2.imwrite(f'output/log/logConstant-{constant[k]}.png',out)

    #initilize/reset output image
    out=image.copy()

# power law transformation with varying gamma and y valyes:
# from the slides, we know that expanding narrow ranges utilizes a smaller y
pwer, while narrowing utilized a larger y
# for this reason we are using smaller and smaller y values to see effect
yValue = [.9,.8,.7,.6,.5,.4]
#we define a few gamma levels to see difference
gamma = [1,1.5,2.2]

#outer loop goes through each gamma
for l in range(3):
    # k loop goes through each y value
    for k in range(6):
```

```
        # i,j for each pixel in image
        for i in range(image.shape[0]):
            for j in range(image.shape[1]):
                # using the formula from the lecture slides:
                # s = c * r^y
                # we apply this fomula with varying y values, and adjust for
gamma by dividing each y value by a gamma level
                # we use the c values 255 to scale the range from 0,1 given by
the power to 0,255
                f = math.pow(image[i][j][1]/255,(yValue[k]/gamma[l]))*255

                #set pixel grey level to function output
                out[i][j] = [f,f,f]
        #display this runs results
        plottool.imshow(out)
        plottool.title( f'y={yValue[k]}, gamma={gamma[l]}')
        plottool.show()
        # save results with name contianing gamma and y value used
        cv2.imwrite(f'output/power/gamma-{gamma[l]}/yValue-{yValue[k]}.png',out)
        #reset output image for next run
        out=image.copy()
```

3.2 Code for histogram equalization (greyscale) (Histogram-EQ.py)

```
# Michael Rizig
# Project 2: Image Enhancement
# 001008703
# File 2: Histogram Equalization
# 6/12/2024

#Import nessessary tools
import matplotlib.pyplot as plottool
import cv2
from skimage import io

#equalizaion function created to match the histogram for this image

def equalizationFunc(pixel):
    # to find formula, first i analized the histogram distrobution of the graph
    # at first glance, the graph is mostly centered around the range 0,50,
meaning the max-min gives us a range of about 50.
```

```
    # by dividing each pixel value by 50, we get its ratio, and by then
multiplying that ratio by 255, we get its corrected mapped value
    # the formula below does this
    return 255 * (pixel/50)

# image path
mapPath = 'input/university.png'

#read in image
university = io.imread(mapPath)

#color corect image from bgr
university = cv2.cvtColor(university,cv2.COLOR_BGR2RGB)

#define range and max value for histogram tool
greyrange = [0,256]
maxsize= [256]

# utilize cv2 to draw historgram data
hist = cv2.calcHist([university], [0], None, maxsize, greyrange)

#define output image and initilize it to university image
out= university.copy()

#display original image
plottool.imshow(out)
plottool.title("Original image:")
plottool.show()

#display original image's histogram distrobution
plottool.plot(hist)
plottool.title("Original image Histogram:")
plottool.show()

#print(out.shape)
#loop through the input image and apply the equalizaton function to each pixel's
grey value, saving the value in the output image
for k in range(0,university.shape[0]):
    for p in range(0,university.shape[1]):
        #apply the formula to each pixel
        out[k][p] = equalizationFunc(university[k][p])


#calculate new histogram
hist = cv2.calcHist([out], [0], None, maxsize, greyrange)
```

```
#display histrogram
plottool.plot(hist)
plottool.title("Equalized Histogram:")
plottool.show()

#display corrected image
plottool.imshow(out)
plottool.title("Equalized Image:")
plottool.show()

#save image
cv2.imwrite('output/hist/university/uniEqualized.png',out)
```

3.3 Code for histogram equalization (color) (Histogram-color.py)

```
# Michael Rizig
# Project 2: Image Enhancement
# 001008703
# File 3: Histogram Color Equalization
# 6/12/2024

# nessesary imports
import cv2
from skimage import io
import matplotlib.pyplot as plottool

#define the equalization funciton for this image
# finding a function for this image was more challanging than the previous uni
image
# by analizing the histogram we can see that the graph centers around the range
(165,220) and has a range with of around 65
# we also see that there are a few outlier pixels in the extremes that would give
us issues
def equalizationFunc(pixel):
    # to account for low outliers that would give weird artifacts in the output,
I use a simple if statement
    if pixel <150:
        #if the pixel is below our main range, set it equal to 0 to prevent
strange artifacting
        pixel =0
    else:
```

```
        #else we subtract the pixel from out lower bound so that we dont lose any
colors on the high end, and that we convert the problem back to a simple range
expansion in one directoin
        pixel -=150
    # by using the same concept from the previous histogram eq file, we divide
each pixel by our range, and scale the (0,1) result to (0,255) by multiplying
    return 255 * (pixel/65)

#define our color range and max color value for the histogram
colorrange = [0,256]
maxsize= [256]
#as usual we set path and use skimage to open file
satpath = 'input/sat_map.png'
satImage = io.imread(satpath)
#print(satImage[0][0])

#define out output image


satImage = cv2.cvtColor(satImage,cv2.COLOR_BGR2RGB)
out = satImage.copy()
#sshow our input image before processing
plottool.imshow(satImage)
plottool.show()

#calculate histogram for first channel (B)
hist = cv2.calcHist([satImage], [0], None, maxsize, colorrange)
#plot histrogram to show our original image's B channel
plottool.plot(hist)
plottool.title("B distribution Before:")
plottool.show()

#histogram correction for B
for k in range(0,satImage.shape[0]):
    for p in range(0,satImage.shape[1]):
        out[k][p][0] = equalizationFunc(satImage[k][p][0])

#show output image after our B channel correction
plottool.imshow(out)
plottool.show()
out = cv2.cvtColor(out,cv2.COLOR_BGR2RGB)

cv2.imwrite('output/hist/satmap/B-Corrected.png',out)
#show equalized historgram for B channel after processing
hist = cv2.calcHist([out], [0], None, maxsize, colorrange)
```

```
plottool.plot(hist)
plottool.title("B distribution After EQ:")
plottool.show()

hist = cv2.calcHist([out], [1], None, maxsize, colorrange)
plottool.plot(hist)
plottool.title("G distribution Before EQ:")
plottool.show()
#histogram correction for G
for k in range(0,satImage.shape[0]):
    for p in range(0,satImage.shape[1]):
        out[k][p][1] = equalizationFunc(satImage[k][p][1])

#show output image after G channel histogram equalization
plottool.imshow(out)
plottool.show()
out = cv2.cvtColor(out,cv2.COLOR_BGR2RGB)
cv2.imwrite('output/hist/satmap/G-Corrected.png',out)

# show equalized histogram for g values
hist = cv2.calcHist([out], [1], None, maxsize, colorrange)
plottool.plot(hist)
plottool.title("G distribution After EQ:")
plottool.show()

hist = cv2.calcHist([out], [2], None, maxsize, colorrange)
plottool.plot(hist)
plottool.title("R distribution Before EQ:")
plottool.show()
#correct R values
for k in range(0,satImage.shape[0]):
    for p in range(0,satImage.shape[1]):
        out[k][p][2] = equalizationFunc(satImage[k][p][2])

#show resulting image
plottool.imshow(out)
plottool.show()
out = cv2.cvtColor(out,cv2.COLOR_BGR2RGB)

cv2.imwrite('output/hist/satmap/R-Corrected.png',out)


#show resulting histo
hist = cv2.calcHist([out], [2], None, maxsize, colorrange)
plottool.plot(hist)
```

```
plottool.title("R distribution After EQ:")
plottool.show()
```

3.4 Code for histogram equalization (HSI)

```
  # Michael Rizig

# Project 2: Image Enhancement
# 001008703
# File 5: HSI Equalization
# 6/16/2024

from skimage import io
import cv2
import matplotlib.pyplot as plottool

def equalizationFunc(pixel):
    # to account for low outliers that would give weird artifacts in the output,
I use a simple if statement
    if pixel <150:
        #if the pixel is below our main range, set it equal to 0 to prevent
strange artifacting
        pixel =0
    else:
        #else we subtract the pixel from out lower bound so that we dont lose any
colors on the high end, and that we convert the problem back to a simple range
expansion in one directoin
        pixel -=150
    # by using the same concept from the previous histogram eq file, we divide
each pixel by our range, and scale the (0,1) result to (0,255) by multiplying
    return 255 * (pixel/70)
#define path
mapPath = 'input/sat_map.png'
# read in image using skimage.io
satMap = io.imread(mapPath)

#show original image
plottool.imshow(satMap)
plottool.show()
#convert colors to hsv color scale
satMap = cv2.cvtColor(satMap,cv2.COLOR_BGR2HSV)

#calculate initial histogram for hsi [i] element
```

```
hist = cv2.calcHist([satMap],[2],None, [256],[0,256])

#display original histogram
plottool.plot(hist)
plottool.title("Satmap I histogram distribution:")
plottool.savefig('output/hist/HSI/OriginalHist.png')
plottool.show()

#define output image
out = satMap.copy()

#histogram correction for I
for k in range(0,satMap.shape[0]):
    for p in range(0,satMap.shape[1]):
        out[k][p][2] = equalizationFunc(satMap[k][p][2])

#calculate new corrected histogram
hist = cv2.calcHist([out],[2],None, [256],[0,256])

#plot new histogram
plottool.plot(hist)
plottool.title("Satmap I Equalized histogram distribution:")
plottool.savefig('output/hist/HSI/CorrectedHist.png')
plottool.show()

#show image after I histogram correction
out = cv2.cvtColor(out,cv2.COLOR_HSV2BGR)
cv2.imwrite('output/hist/HSI/correctedSatMap.png',out)
plottool.imshow(out)
plottool.show()
```

3.5 Code for noise reduction (noiseReduction.py)

```
# Michael Rizig
# Project 2: Image Enhancement
# 001008703
# File 4: Noise Reduction via Average Filtering & Median Filtering
# 6/12/2024

# import nessesary libs
import cv2
from skimage import io
import matplotlib.pyplot as plottool
import statistics
```

```
# below function defines the average filter, which takes in the coordinates x, y
of the pixel value, and the size of the desired filter.
def averageFilter(x,y,size):
    #init a sum varible to 0
    s =0
    #loop through the filter size. Since we want the filter to be centered at our
x,y we need to modify the range of our loop
    # by dividing the filter size by 2, then going to the left and right by the
quotent, we are essentially centering the filter
    # ive done this by setting the range from -(size of filter)/2 to (size of
filter)/2 +1, then taking floor of these values
    # for example , with filter size 3, we get range -1,2, meaning our filter
will span:
    # [i-1,j-1] [i-1,j] [i-1,j+1]
    # [i,  j-1] [i,  j] [i,  j+1]
    # [i+1,j-1] [i+1,j] [i+1,j+1]
    for i in range(-int(size/2),int(size/2)+1):
       for j in range(-int(size/2),int(size/2)+1):
        #for each value we add its weight to the sum (all values share the same
weight in this implementation)
        s+= padded[x+i][y+j][0] / size**2
    #finally return the average
    return s

#below function defines the medianfilter, which take the same range of value
def medianFilter(x,y,size):
    # instead of a sum, we define a set for all values
    s =[]

    #loop through the filter size. Since we want the filter to be centered at our
x,y we need to modify the range of our loop
    # by dividing the filter size by 2, then going to the left and right by the
quotent, we are essentially centering the filter
    # ive done this by setting the range from -(size of filter)/2 to (size of
filter)/2 +1, then taking floor of these values
    # for example , with filter size 3, we get range -1,2, meaning our filter
will span:
    # [i-1,j-1] [i-1,j] [i-1,j+1]
    # [i,  j-1] [i,  j] [i,  j+1]
    # [i+1,j-1] [i+1,j] [i+1,j+1]
    for i in range(-int(size/2),int(size/2)+1):
       for j in range(-int(size/2),int(size/2)+1):
        #instead of adding like we did before, we are simply appending each value
to the list
        s.append( padded[x+i][y+j][0])
```

```
    #sort the values (the easy way)
    s.sort()
    #and finally return the median of the values
    return statistics.median(s)

#define filter sizes
filter_size = [3,5,7]
# define image path
noisyAtriumPath = 'input/noisy_atrium.png'
# import image using skimage
noisyAtrium = io.imread(noisyAtriumPath)
#adjust colors from bgr to rgb
noisyAtrium = cv2.cvtColor(noisyAtrium,cv2.COLOR_BGR2RGB)

#print(noisyAtrium.shape)

#display original image
plottool.imshow(noisyAtrium)
plottool.show()

#this loop applies the average noise filter
for i in filter_size:

    #start by padding the image, which is one simple way described in the
lectures to avoid out of bounds errors
    #padding by the floor of half the filter size: since center of filter is
always an image pixel, we only need 1/2 of filter size out each direction
    padding = int(i/2)
    # use cv2 built in tool to add padding
    padded =
cv2.copyMakeBorder(noisyAtrium,padding,padding,padding,padding,cv2.BORDER_CONSTAN
T,value=[0,0,0])
    # finally, initialize out output image for this filter
    out = padded.copy()

    #this loop goes through each pixle and applies the filter
    for k in range (padding,noisyAtrium.shape[0]-padding):
        for j in range(padding,noisyAtrium.shape[1]-padding):
            out[k][j] = averageFilter(k,j,i)

    #save the image with the appropriate informative name
    cv2.imwrite(f'output/reduction/average/filtersize-{i}.png',out)

    #finally show each image and how the filter smoothing effected it
```

```
    plottool.imshow(out)
    plottool.show()

# this loop works essentially the same way but with the median filtering
for i in filter_size:

    #create padding for image
    padding = int(i/2)
    padded =
cv2.copyMakeBorder(noisyAtrium,padding,padding,padding,padding,cv2.BORDER_CONSTAN
T,value=[0,0,0])
    out = padded.copy()

    #apply filter
    for k in range (padding,noisyAtrium.shape[0]-padding):
        for j in range(padding,noisyAtrium.shape[1]-padding):
            out[k][j] = medianFilter(k,j,i)
    #save and display
    cv2.imwrite(f'output/reduction/median/filtersize-{i}.png',out)
    plottool.imshow(out)
    plottool.show()
```