Digital Image Processing: Project 1

Jacob Taylor Cassady

CECS 625:Digital Image Processing

9/18/19

Contents

[1. Introduction 2](#_Toc19743146)

[2. Approach 2](#_Toc19743147)

[2.1. Adaptive Histogram Equalization 2](#_Toc19743148)

[2.2. Adaptive Statistical Equalization 3](#_Toc19743149)

[3. Results 4](#_Toc19743150)

[3.1. Data 4](#_Toc19743151)

[3.2. Results 4](#_Toc19743152)

[3.2.1. Adaptive Histogram Equalization 5](#_Toc19743153)

[3.2.2. Adaptive Statistical Equalization 6](#_Toc19743154)

[4. Discussion 6](#_Toc19743155)

[5. Conclusion 7](#_Toc19743156)

[6. References 7](#_Toc19743157)

[7. Appendices 7](#_Toc19743158)

[7.1. FileManager.py 7](#_Toc19743159)

[7.2. main.py 7](#_Toc19743160)

# Introduction

Image transformations can be used to improve the perceptual quality of an image. These transformations are functions that take in an image as an input and return a new image as an output. The transformations come in a large variety, many with different benefits and approaches. An image, Fig0326.tif, was used to test different image transformations in an attempt to reveal hidden information. These transformations were designed as algorithms using the Python 3.7 and accompanying 3rd party libraries: NumPy, scipy, opencv, and matplotlib.

# Approach

Two transfer algorithms were developed in an attempt to reveal some underlying information in the given image. These algorithms are: Adaptive Histogram Equalization and Adaptive Statistical Equalization. Both algorithms utilize an adaptive approach that looks at a subset, or “neighborhood”, of pixels within an image when applying each transform. For an input pixel(i,j) the function returns a neighborhood centered at (i,j) and of size (m, n). This approach used a neighborhood of size (5,5) and utilized a padding of 0 pixel values for edge cases. The pertinent chunk of code for calculating the neighbors is shown in **Figure 1** below. Please refer to the appendices for a copy of the full source code.

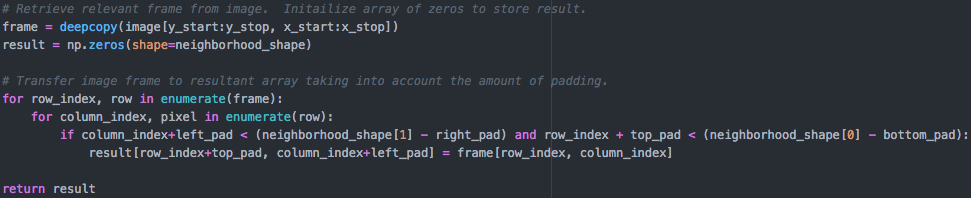


Figure 1 : get\_neighborhood() Pertinent Code Snippet

## Adaptive Histogram Equalization

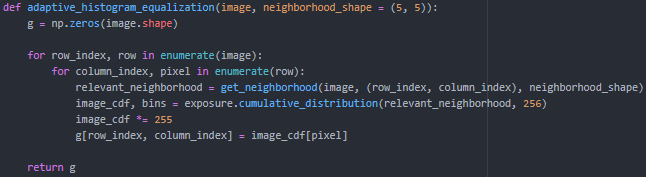
The Adaptive Histogram Equalization algorithm applies histogram equalization by iterating over neighborhoods and calculating the cumulative density function for each neighborhood. The center pixel value of the neighborhood is then used as an input in the neighborhood’s cumulative density function. The output of this function replaces the center pixel value from the input image within output image, g. For reference to the exact implementation of this algorithm, please refer to **Figure 2** below.

Figure 2 : Adaptive Histogram Equalization Algorithm

## Adaptive Statistical Equalization

The Adaptive Statistical Equalization algorithm references the statistics of a neighborhood surrounding pixel(i, j) of the input image to decide rather or not the pixel value should be enhanced by a factor of E in the output image g at the same pixel location (i, j).

The enhancement decision hinges on two comparisons. First off, the neighborhood mean must be less than or equal to a constant k0 \* the global mean of the image. Since k0 must be less than 1 and greater than 0, this means the neighborhood mean must be less than some percentage of the global mean to be enhanced.

The second comparison focuses on standard deviation. The pixel’s neighborhood’s standard deviation must be larger than or equal to a constant k1\* the global standard deviation while still being less than or equal to a constant k2 \* the global standard deviation. Again, since k1 and k2 must be less than 1 but greater than 0 these values create a range of pixel values to enhance. Variable creates a lower threshold while k2 creates the ceiling.

In other words, pixel (i,j) of input image f(i, j) is enhanced by E in output image g(i,j) if the neighborhood’s mean is less than (k0\*100)% of the global mean and the neighborhood’s standard deviation is greater than (k1\*100)% of the global standard deviation and less than (k2\*100)% of the global standard deviation. Consequently, areas of higher intensity are enhanced as well as those within a range of standard deviation. For reference to the exact implementation of this algorithm please see **Figure 3** below.

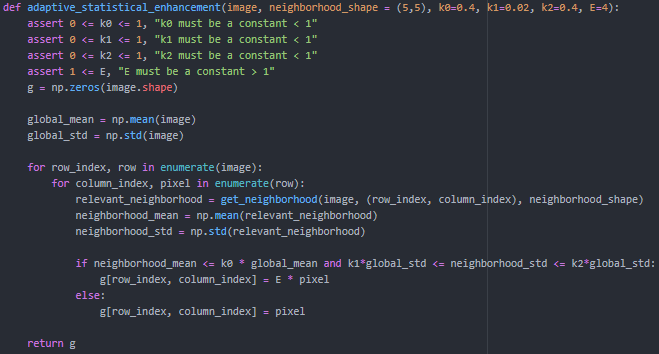


Figure 3 : Adaptive Stastical Equalization Algorithm

# Results

The algorithms described above were both implemented successfully. They produce different results with different tradeoffs between the two. Images in this section were generated using matplotlib with Python 3.7.

## Data

These algorithms were tested on one image. The input image tested was Fig0326.tif shown in **Figure 4** below. The image contains 5 black squares geometrically spaced. The squares look as though they have maximum intensity values while the surrounding space has minimum intensity values. There is little to no perceived noise in the image.

A screenshot of a computer

Description automatically generated

Figure 4 : Fig0326.tif Input

## Results

Four different representations of the input image described in section 3.1 of this document are shown in **Figure 5** below. On the far left, is the input image Fig0326.tif without any transformations. Next is the same image with histogram equalization applied without any adaptive implementation. Following is Fig0326.tif after Adaptive Histogram Equalization. Lastly, the far right shows the output image after Adaptive Statistical Equalization. The two latter examples reveal some hidden information in the images not exposed in the input image or from standard histogram equalization.

A picture containing crossword puzzle, text

Description automatically generated

Figure 5 : Equalization Algorithm Outputs

### Adaptive Histogram Equalization

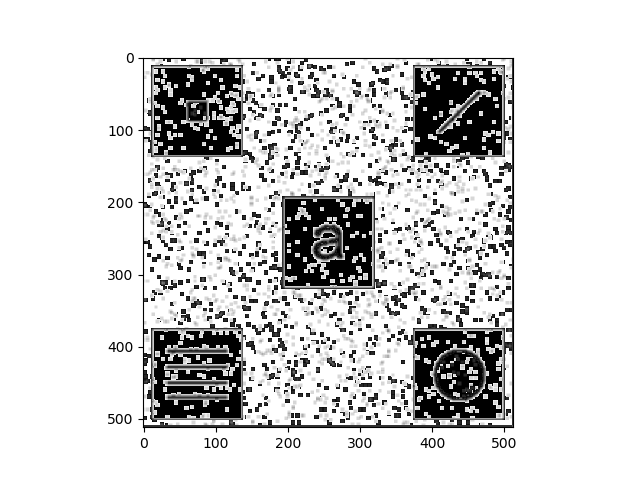
****

Figure 6 : Adaptive Histogram Equalization Fig0326.tif Output

### Adaptive Statistical Equalization

**A screenshot of a cell phone

Description automatically generated**

Figure 7 : Adaptive Statistical Equalization Fig0326.tif Output

# Discussion

The output of Adaptive Histogram Equalization reveals symbols inside of each of the black boxes. Neighborhood size for the output shown in **Figure 6** from section **3.2.1** was (5, 5). As you can see, some areas of the image gained perceptual clarity while adding a lot of increased noise in all areas of the image. This is a consequence of the algorithm’s failure to focus only on areas of interest like other algorithms such as Adaptive Statistical Equalization.

The output of Adaptive Statistical Equalization also reveals symbols inside of each of the black boxes but without adding the extra noise from the previously mentioned algorithm. This can be found in **Figure 7** from section **3.2.2**. There is no longer noise in the white areas of the image, as they do not meet the neighborhood mean cutoff with respect to the global mean. Although this algorithm produces less noise, certain symbols are tougher to perceive such as the ‘a’ found in the center box of **Figure 6**. This could be mitigated though as k1 and k2 can be tuned to create better perception with respect to the edges of the symbols.

It is important to note none of these symbols were exposed when applying standard Histogram Equalization. The hidden symbols might be masked when the image isn’t analyzed adaptively.

# Conclusion

In conclusion, both algorithms reveal underlying information in the input image that was not found using standard histogram equalization. The output image from Adaptive Histogram Equalization included much more noise than Adaptive Statistical Equalization but also did a better job of improving perception of the hidden symbol’s edges. These algorithms could be used together to gather information using Adaptive Histogram Equalization about the image’s underlying edges while using Adaptive Statistical Equalization to best tune to the found symbols.

# References

NumPy Documentation - <https://docs.scipy.org/doc/>

Matplotlib Documentation - <https://matplotlib.org/3.1.1/contents.html>

Scipy Documentation - <https://www.scipy.org/>

OpenCV Documentation - <https://docs.opencv.org/2.4/>

# Appendices

## FileManager.py

import cv2

class FileManager():

@staticmethod

def read\_grayscale\_image(image\_path):

return cv2.imread(image\_path, 0)

## main.py

import sys

import cv2

from copy import deepcopy

import numpy as np

import matplotlib.pyplot as plt

from skimage import exposure

from scipy.stats import norm

from FileManager import FileManager

def display\_image(image):

# Display image:

plt.imshow(image, cmap='gray')

plt.show()

def adaptive\_histogram\_equalization(image, neighborhood\_shape = (5, 5)):

g = np.zeros(image.shape)

for row\_index, row in enumerate(image):

for column\_index, pixel in enumerate(row):

relevant\_neighborhood = get\_neighborhood(image, (row\_index, column\_index), neighborhood\_shape)

image\_cdf, bins = exposure.cumulative\_distribution(relevant\_neighborhood, 256)

image\_cdf \*= 255

g[row\_index, column\_index] = image\_cdf[pixel]

return g

def adaptive\_statistical\_enhancement(image, neighborhood\_shape = (5,5), k0=0.4, k1=0.02, k2=0.4, E=4):

assert 0 <= k0 <= 1, "k0 must be a constant < 1"

assert 0 <= k1 <= 1, "k1 must be a constant < 1"

assert 0 <= k2 <= 1, "k2 must be a constant < 1"

assert 1 <= E, "E must be a constant > 1"

g = np.zeros(image.shape)

global\_mean = np.mean(image)

global\_std = np.std(image)

for row\_index, row in enumerate(image):

for column\_index, pixel in enumerate(row):

relevant\_neighborhood = get\_neighborhood(image, (row\_index, column\_index), neighborhood\_shape)

neighborhood\_mean = np.mean(relevant\_neighborhood)

neighborhood\_std = np.std(relevant\_neighborhood)

if neighborhood\_mean <= k0 \* global\_mean and k1\*global\_std <= neighborhood\_std <= k2\*global\_std:

g[row\_index, column\_index] = E \* pixel

else:

g[row\_index, column\_index] = pixel

return g

def get\_neighborhood(image, pixel\_location, neighborhood\_shape):

# Retrieve data from inputs

rows = len(image[:, 0])

columns = len(image[0, :])

pixel\_row, pixel\_column = pixel\_location

# Initialize variables dependent on if statements

left\_pad = 0

right\_pad = 0

top\_pad = 0

bottom\_pad = 0

# Compute image\_frame WRT neighborhood\_shape and pixel location

mid\_to\_right = neighborhood\_shape[0] // 2

mid\_to\_top = neighborhood\_shape[1] // 2

y\_start = int(pixel\_row-mid\_to\_right-1)

if y\_start < 0:

top\_pad = y\_start \* -1 -1

y\_start = 0

y\_stop = int(pixel\_row+mid\_to\_right+1)

if y\_stop > columns - 1:

bottom\_pad = -(y\_stop - columns - 1)

y\_stop = columns

x\_start = int(pixel\_column-mid\_to\_top-1)

if x\_start < 0:

left\_pad = x\_start \* -1 -1

x\_start = 0

x\_stop = int(pixel\_column+mid\_to\_top+1)

if x\_stop > rows:

right\_pad = x\_stop - rows - 1

x\_stop = rows

# Retrieve relevant frame from image. Initailize array of zeros to store result.

frame = deepcopy(image[y\_start:y\_stop, x\_start:x\_stop])

result = np.zeros(shape=neighborhood\_shape)

# Transfer image frame to resultant array taking into account the amount of padding.

for row\_index, row in enumerate(frame):

for column\_index, pixel in enumerate(row):

if column\_index+left\_pad < (neighborhood\_shape[1] - right\_pad) and row\_index + top\_pad < (neighborhood\_shape[0] - bottom\_pad):

result[row\_index+top\_pad, column\_index+left\_pad] = frame[row\_index, column\_index]

return result

if \_\_name\_\_ == "\_\_main\_\_":

args = sys.argv

if(len(args) != 2):

print("Command Line Arguments should follow the format:")

print("python main.py [relative\_image\_path]")

else:

image\_path = args[1]

image = FileManager.read\_grayscale\_image(image\_path)

equalized\_image = cv2.equalizeHist(image)

ahe\_image = adaptive\_histogram\_equalization(image)

ase\_image = adaptive\_statistical\_enhancement(image)

# stack images side by side

res = np.hstack((image, equalized\_image, ahe\_image, ase\_image))

display\_image(res)