Image Processing and Modern Photo Editing

Emily Kaiser: 109322893
Katherine Simon: 109732851
Peter LeCavalier: 109223560

APPM 3310-002 Fall 2022

April 29, 2022

1 Abstract

The topic we are exploring in this paper is image processing, specifically examining convolution. We begin our analysis by delving into the simple matrix operations introduced in class and how they are applied to both grayscale and colored images with RGB layers, such as matrix addition/subtraction, rotation, and scalar multiplication. All of these methods are demonstrated on sample images. The next step of image processing we discuss is image convolution, which demonstrates a more complex application of matrix operations to photo editing. Convolution allows images to be blurred, sharpened, and aids us in a simpler process of edge detection in photos, as well as many other image processing techniques.

We then tie convolution and edge detection to photo editing that the general public uses every day, such as Snapchat filters and AirBrush. We find that comparable results can be produced by manual image processing via coding in Python and automatic filtering on a smartphone application. Using code allows us to highlight the specific efficiency of these techniques, and how the modern user can easily apply them. Complex matrix operations make photo editing possible but are not necessary to manually compute for a well-edited photo, due to developing user-friendly technology. The widespread access to photo editing allows more people to indulge in this creative process, while not requiring them to understand linear algebra or possess any coding skills. On the other hand, this ease of access creates the problem of users performing mindless alterations on photos; reality is rarely reflected by the edited images that are shared on social media and by news outlets.

2 Attribution

Peter LeCavalier wrote the code in Python for our experiment, performed photo operations and helped review mathematical results for paper; Katie Simon and Emily Kaiser analyzed the results and prepared the report.

3 Introduction

Before the rise of smartphones and social media, image processing was commonly only done by photographers, image processors, and those with computer programming skills. The modern ease of editing photos on a smartphone masks the complex processes that go into creating filters and applying them to images. Most of the mathematics of image processing techniques have been around for decades, but with the vast development of technology, these tools are readily available to be quickly used by the average consumer, not just professionals. We are interested in how computationally efficient these modern image processing techniques are, and the pros and cons

of this efficiency. Through this process, we hope to answer the following question: how is modern image editing made possible through complex matrix methods?

Any standard image editing is done through the application of matrix operations, so background knowledge of basic matrix operations and the structure of a digital photo are helpful to understanding the bulk of this paper. Digital images are a collection of matrices, where each pixel of the image is an entry in the matrix. When a camera produces a digital image, the size is typically measured in megapixels, where the higher the number of megapixels, the better the quality of the image. For example, an iPhone 12 Pro Max takes images with 2048x1536 pixels, translating to 12-megapixels (gsmarena). For gray scale images, each element of the image matrix is only one value, where that value corresponds to the intensity on a range from 0 to 255. However, for colored images, each element in the image matrix is a vector with three values corresponding to intensity of red, green, and blue. This three-element vector is called the RGB vector and it explains the color complexity of each pixel (Nag).

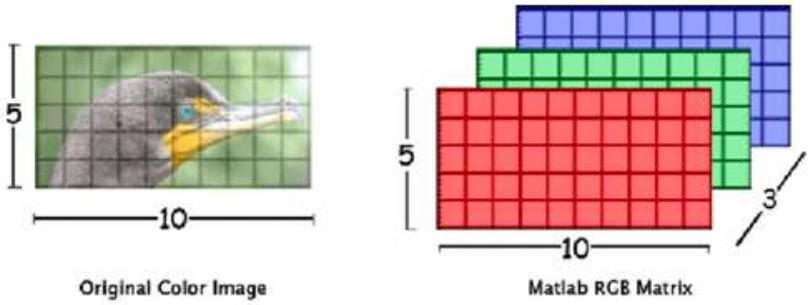


Figure 1 - RGB matrix visualization (Nag)

The scale of intensity has zero as the lack of light (black) and the highest value, 255, as the most intense version of the color (pure white, red, green, or blue). For example, pure red would be represented by the vector $[255, 0, 0]$ and pure purple would be represented by $[255, 0, 255]$.

We plan to discuss and apply a few matrix operations to a sample image as an introduction to image processing, such as matrix addition/subtraction, rotation, and multiplication. The next step of image processing we will discuss is image convolution: the process of transforming an image by applying a kernel over each pixel and surrounding pixels across the entire image. To dive deeper, we will connect these methods to photo editing that the general public uses every day, such as pre-installed iPhone editing and Instagram filters. The heart of our numerical computations will be using our own code and matrix operations to edit photos we have taken, comparing the results to the same photos edited on a smartphone, and highlighting the computational efficiency of smartphone editing.

4 Mathematical Formulation

When beginning to work with image processing, we found basic functions like addition/subtraction, scalar multiplication, and rotation of images to be beneficial in understanding how matrix operations work on images and further how we can apply this to our main topic of interest, convolution. As we discussed above, each colored digital image is a

compilation of three matrices stacked on top of each other in order to represent the primary colors red, green, and blue. This makes basic matrix operations much more complex, since a new formula needs to be created to apply an operation to all three layers. However, this also means we can do a lot more with each digital image depending on how we want to alter it. We will discuss how each operation is conducted and then give an example of how we altered our own image with each operation throughout the paper.

4.1 Rotation

Rotation of an image uses a common rotation matrix but differs in application from a typical vector rotation. Multiplying an image by the rotation matrix will not rotate the image as we would expect. To rotate an entire image by a certain **θ** angle, each pixel of an image, with the vector $[x \quad y]^T$ denoting the pixel's coordinate location, must be moved to a new location within the image matrix (Berry). Multiplying each pixel's location by the rotation matrix by the appropriate angle **θ** will move the pixel to the correct corresponding location, as shown in Equation (1).

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix} \quad (1)$$

If the rotation angle is not a multiple of π/2, there will be a distortion in the image, which can be fixed by a process called antialiasing (Berry). For simplicity, we will demonstrate an image rotation of **θ** = k * π/2, as fixing rotation errors with antialiasing is beyond the scope of the analysis in this project.

Figure 2 - Original Images

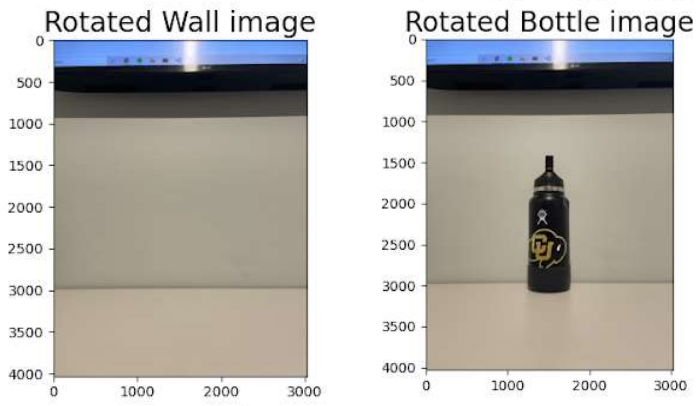


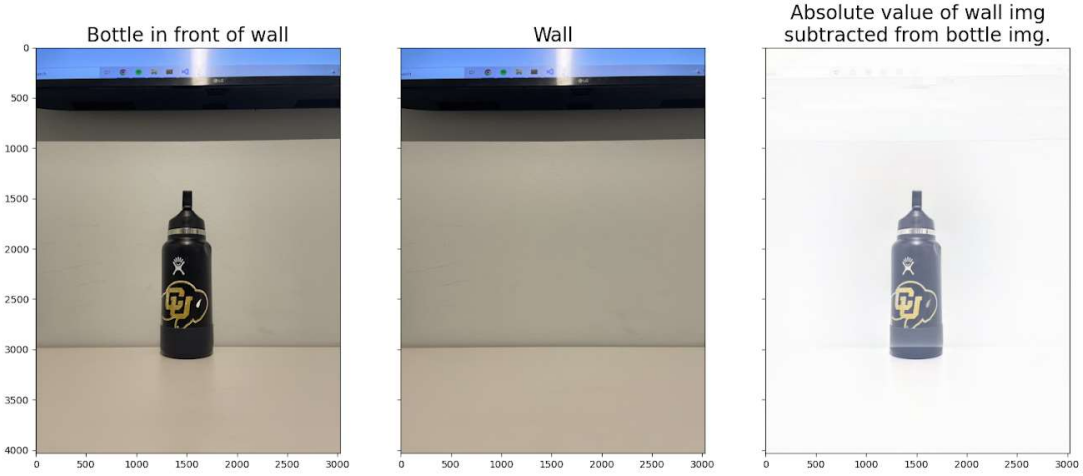
Figure 3 - Rotated Images

When applying the matrix methods to our own images in *Figure 2*, we started off by doing a rotation operation of $\theta = \frac{3\pi}{2}$, to get the images shown in *Figure 3*.

4.2 Addition/subtraction

Adding together two images is the same process as basic matrix addition. Each element in the new image matrix will be a vector composed of three RGB values; each value in the vector is added element-wise, so the new image has the additive combination of the RGB values of the other two images.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & a_{13} + b_{13} \\ a_{21} + b_{21} & a_{22} + b_{22} & a_{23} + b_{23} \\ a_{31} + b_{31} & a_{32} + b_{32} & a_{33} + b_{33} \end{bmatrix} \quad \text{(Joy) (2)}$$



(*Figure 3* again)

Figure 4 - Subtraction of images

Subtraction can be used in several contexts, such as removing the background from an image, as shown in *Figure 4*. Taking the absolute value of an image after subtracting the background image from the original completely isolates the foreground object with a white background.

4.3 Multiplication by a Scalar

Multiplying an image by a scalar also follows the same process as basic matrix scalar multiplication. Each element of the image matrix, across each value in each RGB vector, will be multiplied by the scalar. If the scalar is greater than one, the brightness of the image increases because RGB values increase. If the scalar is less than one, the brightness of the image will decrease.

$$c \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} c \cdot a_{11} & c \cdot a_{12} & c \cdot a_{13} \\ c \cdot a_{21} & c \cdot a_{22} & c \cdot a_{23} \\ c \cdot a_{31} & c \cdot a_{32} & c \cdot a_{33} \end{bmatrix} \quad \text{(Joy) (3)}$$

(*Figure 4* again)

Figure 5 - Darkened image

We multiplied *Figure 4* by a factor of 0.4 to decrease the brightness of the image. This scalar multiplication decreased the intensity values of every entry of the RGB vector for every pixel, giving us *Figure 5*.

4.4 Convolution

Convolution acts as a general-purpose filter effect for images that helps to blur, smooth, sharpen, or detect edges of an image depending on the desired outcome. Someone may want to enhance an entire image, such as enhancing a snapshot from security footage to better understand a crime scene, or zooming in on tiny imperfections and blurring only a certain area in order to get a desired final photo to post to various social media platforms. No matter the scale, matrix convolution operations are at work behind the scenes and are used every day by people anywhere from professionals to everyday social media users. The accessibility of image processing has greatly increased, yet so few people who do this on a daily basis actually understand the processes going on behind the apps on their smartphones or programs on their computer.

At its core, applying a convolution filter on a grayscale image is done by multiplying a pixel's and its neighboring pixels' intensity value by a matrix, called the kernel, and summing each product to produce a singular value in a modified matrix (Nag). The kernel matrix is "slid" across the image matrix we want to modify and multiplied by the corresponding entries in the image matrix. For example, with a standard 3x3 kernel, the kernel matrix is "slid" over every 3x3 block of pixels in the original image. The resulting values of each sum from the sliding kernel matrix will be the pixel values of the modified image that is the product of a blur, smooth, or sharpen convolution filter. *Figure 6* shows this process as if the matrix I is a grayscale image with the kernel K being the matrix that will apply the convolution filter to create the modified image, $I*K$.

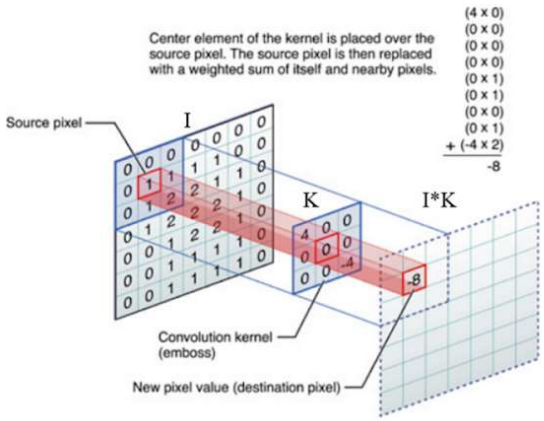


Figure 6 - Image convolution kernel example (Kim)

Applying a convolution to a colored image is a similar process, where the elements in matrix *I* are now RGB vectors, rather than singular intensity values. To apply a convolution, the kernel *K* is again slid over each 3x3 block of pixels, but the multiplication is done independently on each R, G, and B matrix. The summed R, G, and B values then are combined into an RGB vector, which becomes a pixel in the new convoluted image. The general expression of a convolution is represented in Equation (4):

$$g(x,y) = \omega \cdot f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x-dx, y-dy) \quad \text{(Kim) (4)}$$

Where $g(x,y)$ corresponds to the matrix *I*K* (the modified image), $f(x,y)$ corresponds to the matrix *I* (the original image), and ωcorresponds to the matrix *K* (the kernel) in *Figure 6.*

Different kernel matrices will have different effects on the image. There are standard forms of kernel matrices that correspond to many image effects, like the ones listed above. A sharpening kernel emphasizes differences in adjacent pixel values, which will make the image look more vivid. In contrast, a blurring kernel de-emphasizes differences in adjacent pixel values, which will make the image look less vivid. The identity kernel leaves an image unchanged. These standard 3x3 kernels are of the form (Powell):

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

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & c & -1 \\ 0 & -1 & 0 \end{bmatrix}$$
Sharpen Kernel

$$\begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix}$$
Blurring Kernel

The most common kernel sizes for simple image convolution are 3x3 and 5x5. The size of this can vary depending on the complexity and intensity of the desired convolution and can be above 5x5. A kernel matrix for the purpose of a standard convolution will have odd dimensions so that the image can be convoluted pixel by pixel.

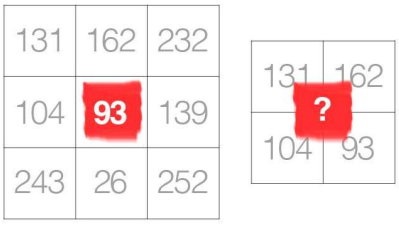


Figure 7 - Even dimension kernel (Sahoo)

As shown in *Figure 7*, a kernel with even dimension will not interpolate to a center pixel. This will result in distortion and will not produce the desired effect without extra manual input.

5 Examples and Numerical Results

We now use all of these processes to show how matrix operations contribute to applications of edge detection and convolution and their effect on modern photo editing.

5.1 Coding Summary

To process the imagery used for this report and apply image processing techniques, we used Python 3.9.7. We primarily used the Pillow, NumPy, and SciPy packages for parsing and processing images.

First, we had to load in and parse our images for use in Python, using the Pillow and NumPy packages (Appendix, lines 115-126). To apply basic addition and subtraction of matrices and scalars, as well as multiplication by scalars across matrices, the NumPy has efficient processing that can easily apply these operations across elements. Although these operations can be applied element by element in the matrix, this package is deemed extremely useful for making these operations as efficient as possible (Appendix, lines 146-150, 162-165). NumPy has methods to apply rotation to an image, however they don't highlight the matrix operations necessary to achieve a rotated image. To highlight the mathematics behind a typical image rotation, we created a function that leverages other NumPy functions such as matrix multiplication with the rotation matrix and assigning the new pixel values on the processed image (Appendix, lines 12-81, 129-132).

Although convolution is a process that involves a multitude of different matrix multiplications across an image, these convolutions can be done very efficiently in Python. The main purpose of the convolution code was to highlight the efficiency behind image processing techniques, so leveraging this package allowed us to highlight how quickly images can be processed with a modern computer (Appendix, lines 175-250). With a small amount of image pre and post processing, we are able to apply a convolution with any kernel of our choice. We added code in our examples to extract the runtime for each convolutional process (Appendix, lines 184-196, 224-234). This yielded a runtime of 0.614 seconds for the entire edge detection process, and a runtime of 1.08 seconds for the heavy gaussian smoothing process. As such, it is clear to see how easy it is for any personal consumer to apply these image processing techniques on their personal computer or smartphone.

A link to a GitHub repository containing all of the code, images, and generated images we used for this project can be found at the beginning of the Appendix.

5.2 Gaussian Edge Detection

Edge detection is a technique that has many applications across image processing, computer vision, and machine learning. Although there are several techniques for detecting edges in an image, one method uses a subtraction of a Gaussian-smoothed image from its original counterpart, as shown in Equation (5), where E is the edge-detected matrix, X is the original image, and G is the Gaussian-smoothed image (Efros).

$$E = X - G \quad (5)$$

This difference will generate negative values for detected edges, and more positive values for non-edge pixels. Effectively, this process finds the pixels in the image which are most affected by a smoothing effect, or equivalently, those which have the largest value gradient between neighboring pixels. We can then threshold these values using Equation (6) to generate a binary matrix, with edge-detected values being 1 and others being 0, where *E'(x,y)* is the value at pixel coordinate *(x,y)* in the threshold matrix, *E(x,y)* is the value at pixel coordinate *(x,y)* in the original edge detection matrix, and *t* is the value at which to threshold.

$$E'(x,y) = \begin{cases} 1, & E(x,y) < t \\ 0, & E(x,y) \geq t \end{cases} \quad (6)$$

Applying this process to an image with faces, for example, can efficiently and effectively extract facial features. As shown in *Figure 8* and *Figure 9*, this Gaussian edge detection performed with a 5x5 Gaussian kernel easily extracts the facial outline, eyes, nose, and mouth from an image.

Figure 8 - Gaussian edge detection

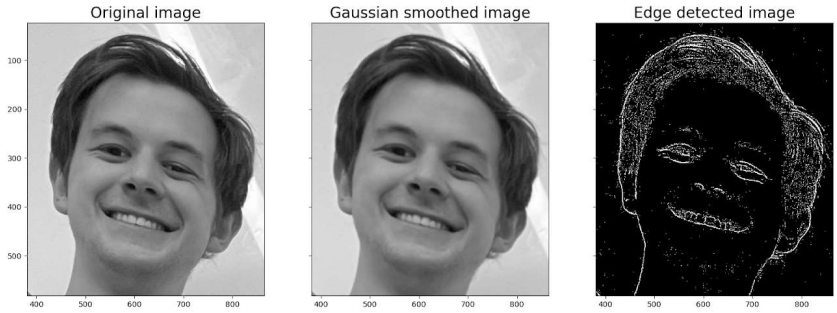


Figure 9 - Zoomed-in edge detection image, highlighting facial features

The processes shown in *Figure 8* and *Figure 9* can, for example, be used for facial recognition and detection. Popular smartphone applications such as Snapchat and Instagram likely use similar techniques to pick out facial features for filters and effects, where a pre-set filter within the app will automatically apply an effect onto regions specified by edge detection. Filters like these have effects that range from blurring acne or changing hair color, to fully transforming facial features to the point where the original image is unrecognizable.

5.3 Blurring/Smoothing Convolution of Acne

One common use for convolution is a blurring or smoothing effect. This can be achieved using several different convolution kernels across an image, although a common one is called a Gaussian kernel. This kernel is generated by applying a Gaussian distribution on the element values of the kernel matrix based on their x and y coordinates. If we call the center entry in the kernel matrix (x,y) = (0,0), we can generate a Gaussian smoothing kernel using Equation (7), where *G(x,y)* represents the Gaussian kernel value at an *x,y* coordinate, and σ is the standard deviation of the Gaussian distribution.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad \text{(Efros) (7)}$$

This can be applied in several contexts, one of which is for smoothing imperfections in images. *Figure 10* shows how a 61x61 Gaussian kernel can be used to smooth acne scars.

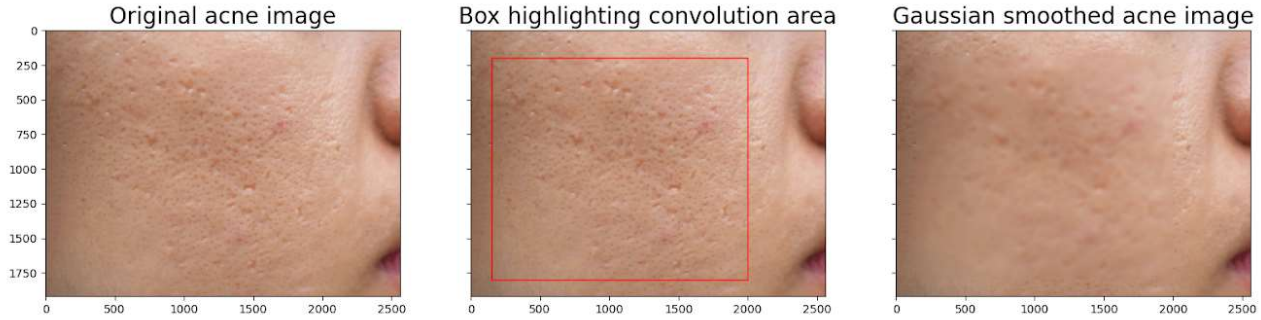


Figure 10 - Manual smoothing of acne scars (PanOxyl)

5.4 Airbrush Application on Face Image

We can compare the Gaussian smoothing convolution to editing done in a smartphone application, AirBrush. There are thousands of applications widely available that can produce comparable results, showing how widespread image alteration is available to the general public. These types of applications mask the complex mathematical processes that go into editing an image: we smoothed the image on the smartphone with just a swipe, while manually coding the smoothing effect took more time, effort, and knowledge of the mathematics behind the effect. *Figure 11* shows the result of smoothing the acne scars with AirBrush, a free app available on the Apple and Google Play stores.

Figure 11 - Automatic smoothing of acne scars

Having photo editing apps at the touch of a finger is beneficial because it allows for more people to edit their photos; photo editing is no longer done only by a few people with select skills and resources. This opens up more opportunities for people to present photos in the way they choose, which can be an outlet for creative processes and freedom. The widespread ownership of smartphones brings these complicated mathematical processes to the hands of the general public and allows anyone to perform the operations we described in this paper, without any knowledge of what those processes are. Additionally, most photo editing apps have sliding scales of how intense filters and other basic editing techniques are applied. Not only can the effect be applied to any photo but also in any way to any part of a photo. This further allows for the editor to customize their photos to the precision they desire.

The widespread access of photo editing, while providing creative opportunities, can result in mindless alterations and a distorted perception of reality; most images posted on social media or published in news outlets are edited in some way. It is difficult to discern what images are real, or what images are altered because everyone has access to simple editing software. Some applications are able to perform automatic edits to dispose of “undesirable” traits, extending this problem further. Not only does this process mask the mathematics behind image processing, but even the editor is not aware of how many edits are being applied to their own photos.

6 Discussion and Conclusion

Through this research, we learned that a digital image itself is a matrix or a collection of matrices. Simple matrix operations can be performed on grayscale images in the same way as on regular matrices, while colored images add extra dimension to these operations. The three element RGB vector that exists for every entry in a colored image matrix complicates simple matrix operations: each operation must be applied to all three corresponding layers of the image. Beyond simple matrix operations, we dove into convolution, which uses multiplication and addition, but is done in a pattern that fully alters the composition of a matrix. We used convolution to then create a form of edge detection, composed of subtraction and a Gaussian blur convolution.

All of these methods led up to the comparison of manual convolution vs automatic photo editing applications used on a smartphone. The results of this process show that despite the complex techniques that go into editing an image, similar results can be achieved by editing a photo manually and editing it with a simple smartphone application. There are many layers of

mathematical operations that go into each edit or filter applied to an image that people who commonly edit photos do not know about. At the end of the day, image processing is made possible by matrix operations.

To extend our research, we could explore more types of convolutions and how we can replicate their effects with smartphone apps. For example, we demonstrated the sharpening kernel, but never applied this to an image of our own using matrices or a smartphone application. We also could dive deeper into the concept of edge detection. Edge detection in image processing is a field of its own used for facial recognition, license plate recognition, passport information extraction, or assistive technology for blind and visually impaired users, among many other applications. We could further explore these common applications using our created method for edge detection, as well as how other methods of edge detection, not formed with convolution, compare. It would be interesting to see how the methods in this project can be extended to broader applications.

Sources

Andonie, R. (1991, October 15). *Gaussian Smoothing By Optimal Iterated Uniform Convolutions*. Retrieved April 22, 2022, from https://www.cwu.edu/faculty/sites/cts.cwu.edu.faculty/files//users/142/documents/Gaussian%20Smoothing_96.pdf.

Apple iPhone 12 pro max. Apple iPhone 12 Pro Max - Full phone specifications. (n.d.). Retrieved April 26, 2022, from https://www.gsmarena.com/apple_iphone_12_pro_max-10237.php

Berry, N. (n.d.). *Rotating Images*. Rotating images. Retrieved April 26, 2022, from https://datagenetics.com/blog/august32013/index.html

Efros, A. (2005, September). *Convolution and edge detection - Carnegie Mellon University*.Convolution and Edge Detection . Retrieved April 22, 2022, from http://graphics.cs.cmu.edu/courses/15-463/2005_fall/www/Lectures/convolution.pdf.

Joy, Dan. *Image Processing through Linear Algebra - Infokara*. http://infokara.com/gallery/78-jan-3530.pdf.

Kim, Sung. *Applications of Convolution in Image Processing with Matlab*. 20 Aug. 2013, http://kiwi.bridgeport.edu/cpeg585/ConvolutionFiltersInMatlab.pdf.

Mohamed, Allali. “Linear Algebra and Image Processing.” *Taylor & Francis*, 7 June 2010, https://www.tandfonline.com/doi/full/10.1080/00207391003675133.

Nag, H. (2020, June 30). *Applications of linear algebra in image filters [part I]-operations*. Medium. Retrieved April 21, 2022, from https://medium.com/swlh/applications-of-linear-algebra-in-image-filters-part-i-operations-aeb64f236845#:~:text=Convolution%20is%20the%20process%20of,similarly%20denoted%20by%20%E2%80%9C*%E2%80%9D.

PanOxyl. “Acne Scar” *PanOxyl*, 9 Mar. 2020, https://www.panoxyl.com/acne-scars/.
Powell, V. (n.d.). *Image kernels explained visually*. Explained Visually. Retrieved April 26, 2022, from https://setosa.io/ev/image-kernels/

Sahoo, S. “Odd vs. Even-sized filters” *Medium* (2018, November 29). *Deciding optimal filter size for cnns*.. Retrieved April 21, 2022, from https://towardsdatascience.com/deciding-optimal-filter-size-for-cnns-d6f7b56f9363.

Appendix-Code

GitHub link for our code: https://github.com/pele6150/matrix_methods_image_processing

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from tqdm import tqdm
from scipy.signal import convolve2d, convolve
from cv2 import copyMakeBorder, BORDER_REFLECT

import math
import timeit

def rotation(im, theta):
    # im as a np array, theta in degrees
    theta = np.radians(theta)
    # Set up the rotation matrix to be applied to each pixel
    rot_mat = np.array([[np.cos(theta), -np.sin(theta)],[np.sin(theta), np.cos(theta)]])

    # max of each axis
    y_max = im.shape[0] - 1
    x_max = im.shape[1] - 1

    # Calculate the lower and upper bounds across each axis when rotation is applied
    bounds = []
    bounds.append(np.matmul(rot_mat, [0, 0]))
    bounds.append(np.matmul(rot_mat, [0, y_max]))
    bounds.append(np.matmul(rot_mat, [x_max, 0]))
    bounds.append(np.matmul(rot_mat, [x_max, y_max]))

    bounds = np.array(bounds)

    # low and high bounds here - (x, y) for each
    low_bounds = np.min(bounds, axis=0)
    high_bounds = np.max(bounds, axis=0)
```

```
34
35      # Floor/ceiling the bounds to make sure out-of-bounds doesn't happen
36      if low_bounds[0] < 0:
37          left_shift = -int(np.floor(low_bounds[0]))
38      else:
39          left_shift = -int(np.ceil(low_bounds[0]))
40      if low_bounds[1] < 0:
41          up_shift = -int(np.floor(low_bounds[1]))
42      else:
43          up_shift = -int(np.ceil(low_bounds[1]))
44
45      up_upper = high_bounds[1] - y_max
46      if up_upper < 0:
47          up_upper = int(np.floor(up_upper))
48      else:
49          up_upper = int(np.ceil(up_upper))
50      left_upper = high_bounds[0] - x_max
51      if left_upper < 0:
52          left_upper = int(np.floor(left_upper))
53      else:
54          left_upper = int(np.ceil(left_upper))
55
56      new_y_size = up_shift + up_upper + im.shape[0]
57      new_x_size = left_shift + left_upper + im.shape[1]
58
59      # Set up a new matrix with each entry being the new pixel placement
60      new_px_mat = np.ones((new_y_size, new_x_size, im.shape[2])).astype(int) + 254
61
62      # Make a meshgrid of coordinate values to apply rotation to
63      xs, ys = np.meshgrid(np.arange(im.shape[1]), np.arange(im.shape[0]))
64      idxs = np.dstack((xs, ys)).reshape((xs.shape[0], xs.shape[1], 2))
65      idxs = idxs.reshape((idxs.shape[0] * idxs.shape[1], 2))
66      idxs = np.swapaxes(idxs, 0, 1)
67
68      # Rotate the indices by matrix multiplying by the rot. matrix
69      rot_idxs = np.matmul(rot_mat, idxs)
70      rot_idxs[0] = np.clip(rot_idxs[0] + left_shift, 0, new_x_size - 1)
71      rot_idxs[1] = np.clip(rot_idxs[1] + up_shift, 0, new_y_size - 1)
72      rot_idxs = np.swapaxes(rot_idxs, 0, 1).astype(int)
73
```

```
74      # Loop through each of the pixels and set the new image
75      for y in range(len(im)):
76          for x in range(len(im[y])):
77              new_x, new_y = rot_idxs[y*len(im[y]) + x].astype(int)
78              # shift the pixel values to the location on the new image
79              # clip to avoid out-of-bounds indexing
80              new_px_mat[new_y, new_x] = im[y, x]
81      return new_px_mat
82
83  # Taken from https://stackoverflow.com/a/12201744
84  # Converts an rgb image into grayscale
85  def rgb2gray(rgb):
86      return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])
87
88  # Create a gaussian smoothing kernel with a specific size
89  def gaussian_kernel(size=5):
90      # radius of the kernel - used to conveniently get the right index
91      radius = int(np.floor(size / 2))
92      # sigma for gaussian distribution
93      sigma = size/4.
94      #coefficient to go in front of each exp
95      coef = (1/(2*math.pi*sigma**2))
96
97      # make an array of coordinates
98      # doing it this way will make matrix math a lot more efficient
99      xs, ys = np.meshgrid(np.arange(-radius, radius+1), np.arange(-radius, radius+1))
100     x2_plus_y2 = np.square(xs) + np.square(ys)
101     # calculate the gaussian at each pixel
102     kernel = coef*np.exp(-(x2_plus_y2)/(2*sigma**2))
103
104     #normalize the kernel
105     kernel = kernel / np.sum(kernel)
106     return kernel
107
108 def box_kernel(size=5):
109     # Creates a kernel for box blurring
110     # Matrix has all matching values, with a sum of 1.
111     kernel = np.ones((size, size))
112     kernel = kernel / np.sum(kernel)
113     return kernel
```

```
114
115 #Loading in images
116 im_filenames = ["./images/wall.JPG", "./images/wall_bottle.JPG",
    "./images/acne_scars.jpg", "./images/us_smiling.jpg"]
117 ims = []
118 for i in im_filenames:
119     cur_im = np.array(Image.open(i)).astype(int)
120     ims.append(cur_im)
121
122 # Make variables for each
123 wall_arr = ims[0]
124 bottle_arr = ims[1]
125 acne_arr = ims[2]
126 smile_arr = ims[3]
127
128
129 # ------ Rotation
130 # Apply a 270 degree rotation on the images
131 rot_wall_arr = rotation(wall_arr, 90)
132 rot_bottle_arr = rotation(bottle_arr, 90)
133
134 # Plot everything
135 fig, axs = plt.subplots(2, 2)
136 axs[0][0].imshow(wall_arr)
137 axs[0][0].set_title("Original Wall image", fontsize=20)
138 axs[0][1].imshow(rot_wall_arr)
139 axs[0][1].set_title("Rotated Wall image", fontsize=20)
140 axs[1][0].imshow(bottle_arr)
141 axs[1][0].set_title("Original Bottle image", fontsize=20)
142 axs[1][1].imshow(rot_bottle_arr)
143 axs[1][1].set_title("Rotated Bottle image", fontsize=20)
144 # ------ END Rotation
145
146 # ------ Subtraction
147 # Take the absolute value of the subtraction between the images
148 # and clip values to stay in 0-255
149 subtract = 255 - np.abs(np.subtract(rot_bottle_arr, rot_wall_arr))
150 subtract = np.clip(subtract, 0, 255)
151
152 # Plot everything
```

```
153 fig, axs = plt.subplots(1, 3, sharey=True, sharex=True)
154 axs[0].imshow(rot_bottle_arr)
155 axs[1].imshow(rot_wall_arr)
156 axs[2].imshow(subtract)
157 axs[0].set_title("Bottle in front of wall", fontsize=20)
158 axs[1].set_title("Wall", fontsize=20)
159 axs[2].set_title("Absolute value of wall img\nsubtracted from bottle img.", fontsize=20)
160 # ------ END Subtraction
161
162 # ------ Scalar Multiplication
163
164 # Scalar multiply the image
165 darker_im = (0.4 * subtract).astype(int)
166
167 # Plot everything
168 fig, axs = plt.subplots(1, 2, sharey=True, sharex=True)
169 axs[0].imshow(subtract)
170 axs[1].imshow(darker_im)
171 axs[0].set_title("Water bottle subtracted image", fontsize=20)
172 axs[1].set_title("Scalar multiplied image (x0.4)", fontsize=20)
173 # ------ END Scalar Multiplication
174
175 # ------ Edge detection
176 # Turn the image into grayscale
177 smile_gray = rgb2gray(smile_arr)
178
179 # Create a gaussian kernel
180 kernel_size = 5
181 kernel_radius = int(np.floor(kernel_size / 2))
182 smooth_55_kernel = gaussian_kernel(kernel_size)
183
184 # This times the processing
185 start = timeit.default_timer()
186
187 # Apply a gaussian smoothing to the image
188 smooth_smile = convolve2d(smile_gray, smooth_55_kernel, boundary="symm")
189
190 # Crop the image to remove padding during the convolution process
191 smooth_smile = smooth_smile[kernel_radius:-kernel_radius,
    kernel_radius:-kernel_radius]
```

```
192
193 # Subtract the smoothed image from the original and threshold
194 edge = smile_gray - smooth_smile
195 edge = edge < -5
196 stop = timeit.default_timer()
197 print(f"Edge detection time: {stop - start} seconds")
198
199 # Plot everything
200 fig, axs = plt.subplots(1, 3, sharey=True, sharex=True)
201 axs[0].imshow(smile_gray, cmap="gray")
202 axs[1].imshow(smooth_smile, cmap="gray")
203 axs[2].imshow(edge, cmap="gray")
204 axs[0].set_title("Original image", fontsize=20)
205 axs[1].set_title("Gaussian smoothed image", fontsize=20)
206 axs[2].set_title("Edge detected image", fontsize=20)
207 # ------ END Edge detection
208
209 # ------ Gaussian Smoothing
210 # Set the kernel size/radius
211 kernel_size = 61
212 kernel_radius = int(np.floor(kernel_size / 2))
213 # Generate a gaussian kernel (expand_dims adds an extra dimension for convolution)
214 smooth_55_kernel = np.expand_dims(gaussian_kernel(kernel_size), axis=-1)
215
216 # Focus on a specific portion of the image
217 smooth_acne = np.copy(acne_arr)
218 acne_crop = smooth_acne[200:1800, 150:2000]
219
220 # Add borders to the image for convolution, reflecting the borders
221 acne_crop = copyMakeBorder(acne_crop, kernel_radius, kernel_radius, kernel_radius,
    kernel_radius, BORDER_REFLECT)
222
223 # Time the processing
224 start = timeit.default_timer()
225
226 # Apply gaussian smoothing to the image
227 smooth_acne_crop = convolve(acne_crop, smooth_55_kernel).astype(int)
228
229 # Remove padding after convolution
230 smooth_acne_crop = smooth_acne_crop[kernel_size - 1:-kernel_size + 1, kernel_size -
```

```
    1:-kernel_size + 1]
231
232 # Replace the section of the image with the convolved area
233 smooth_acne[200:1800, 150:2000] = smooth_acne_crop
234 stop = timeit.default_timer()
235 print(f"Gaussian Smoothing timing: {stop - start} seconds")
236
237 # Create a rectangle to draw over the smoothing area
238 rect = Rectangle((150,200),1850,1600, edgecolor='r', facecolor="none")
239
240 # Plot everything
241 fig, axs = plt.subplots(1, 3, sharey=True, sharex=True)
242 axs[0].imshow(acne_arr)
243 axs[1].imshow(acne_arr)
244 axs[1].add_patch(rect)
245 axs[2].imshow(smooth_acne)
246
247 axs[0].set_title("Original acne image", fontsize=20)
248 axs[1].set_title("Box highlighting convolution area", fontsize=20)
249 axs[2].set_title("Gaussian smoothed acne image", fontsize=20)
250 # ------ END Gaussian Smoothing
251
252 # Show all the plots
253 plt.show()
```