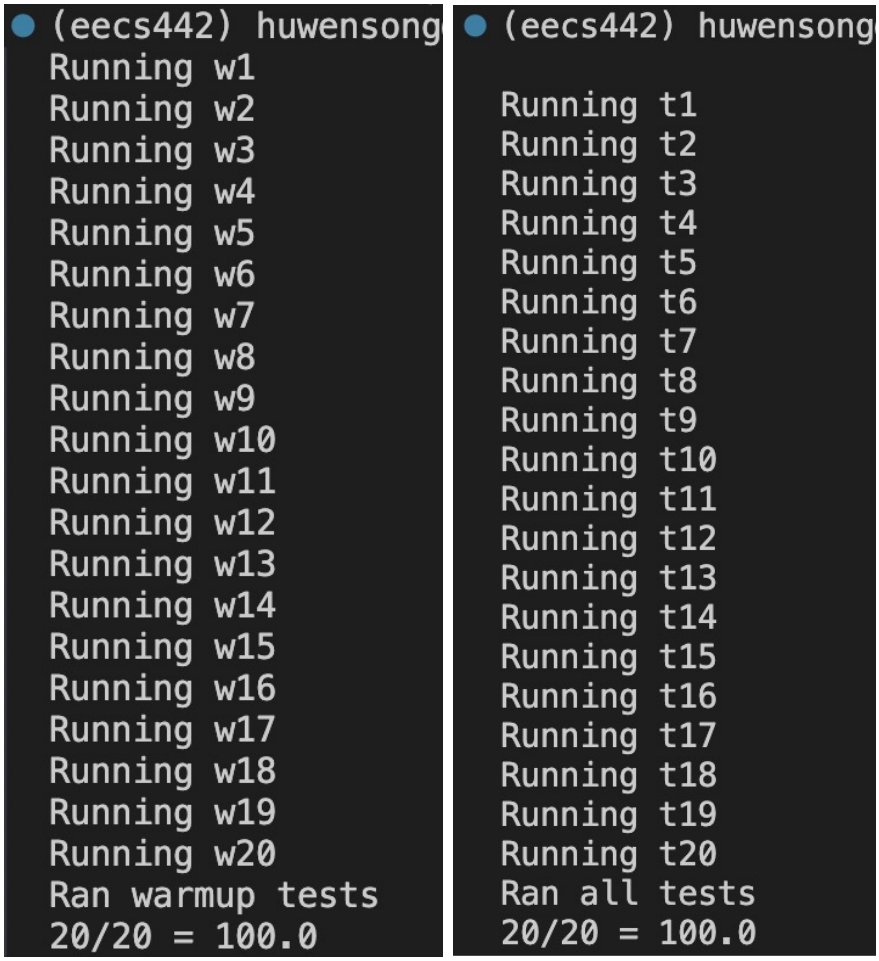
Wensong Hu
EECS
University of Michigan

January 31, 2024

HW1 — Numbers and Images

1 Numpy Intro

1.1 Warmups and Tests Code

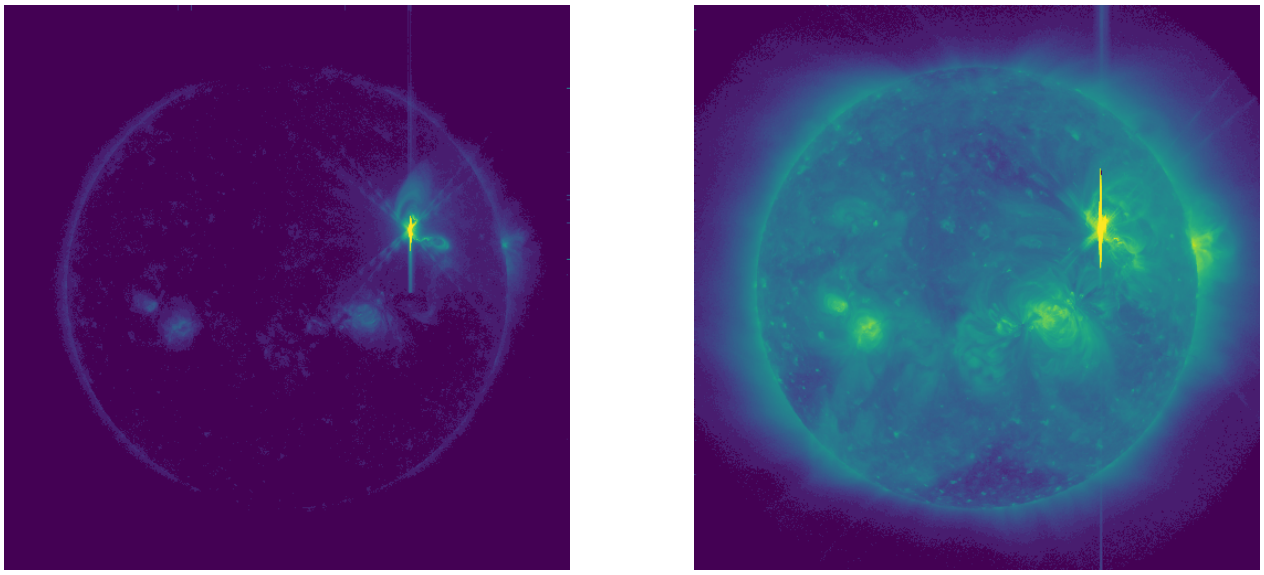


(a) warmups (b) tests

Figure 1: Termial outputs for run.py

2 Data Interpretation and Visualization

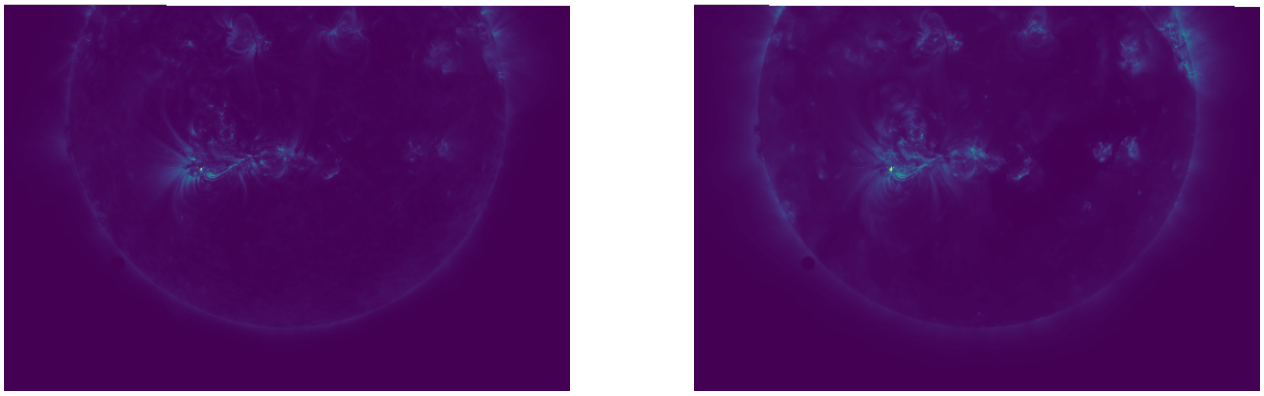
2.1 2 images from mysterydata2.npy



(a) 0th Channel (b) 4th Channel

Figure 2: Corrected The Mysterydata2.npy

2.2 2 images from mysterydata3.npy



(a) 2nd Channel (b) 3rd Channel

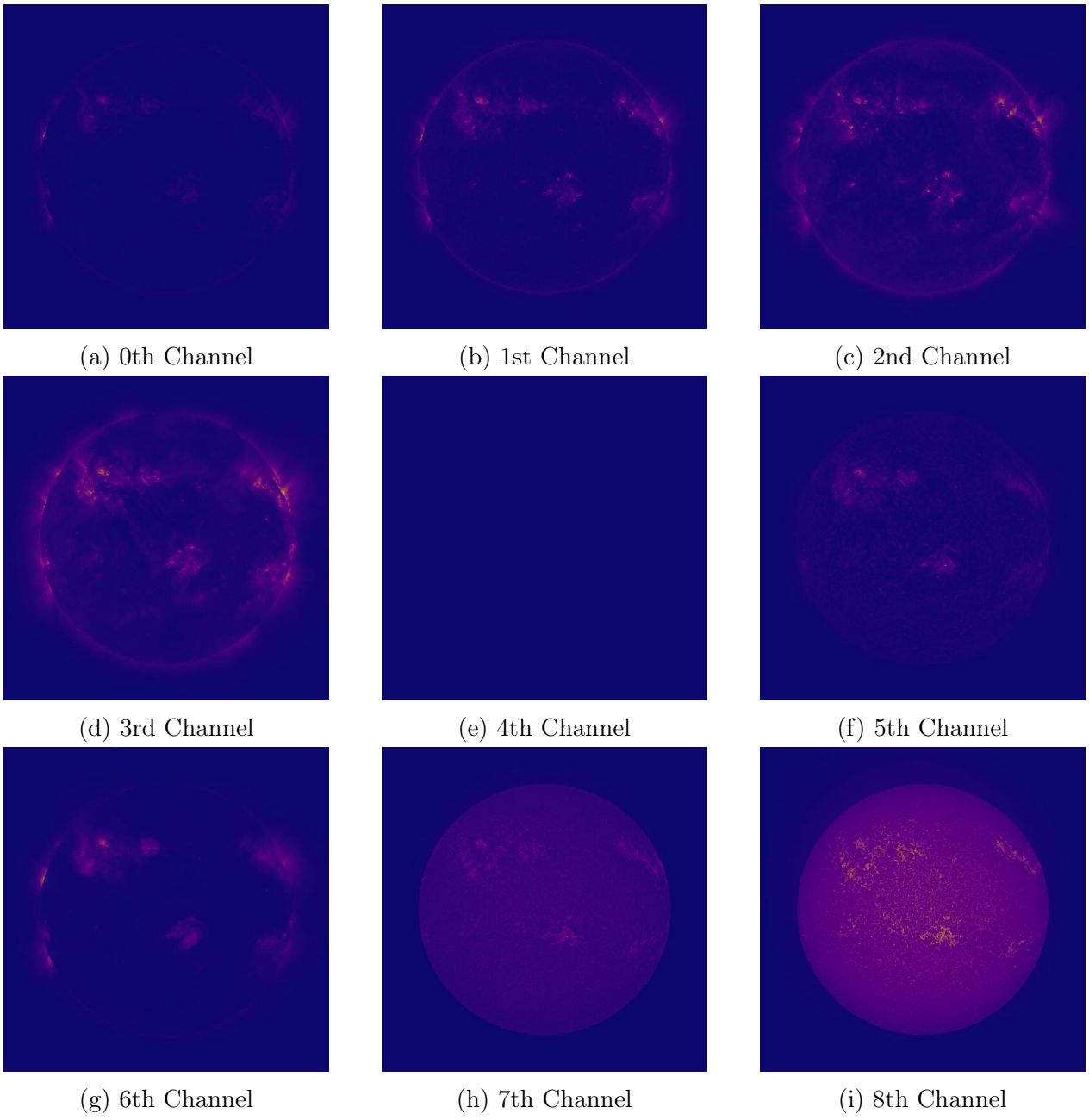
Figure 3: Corrected The Mysterydata3.npy

2.3 colorMapArray()

```python
import os
import numpy as np
import matplotlib.pyplot as plt
import cv2
import pdb


def colormapArray(name, X, colors):
    """
    Basically plt.imsave but return a matrix instead

    Given:
        a HxW matrix X
        a Nx3 color map of colors in [0,1] [R,G,B]
    Outputs:
        a HxW uint8 image using the given colormap. See the Bewares
    """
    vmax = np.nanmax(X)
    vmin = np.nanmin(X)
    N = colors.shape[0]
    X = ((N - 1) * (X - vmin) / (vmax - vmin)).astype(np.uint8)
    data_return = np.zeros((X.shape[0], X.shape[1], 3))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            data_return[i, j, :] = colors[X[i, j]]

    plt.imsave(name, np.log1p(data_return))
    data_return = ((N - 1) * data_return).astype(np.uint8)
    return data_return


if __name__ == "__main__":
    # Example
    colors = np.load("mysterydata/colors.npy")
    data1 = np.load("mysterydata/mysterydata.npy")
    dataNone = [plt.imsave(f"vis_1_{i}.png", data1[:, :, i]) for i in range(9)]

    # Question 1
    data2 = np.load("mysterydata/mysterydata2.npy")
    data2 = np.log1p(data2)
    dataNone = [plt.imsave(f"vis_2_{i}.png", data2[:, :, i]) for i in range(9)]

    # Question 2
    data3 = np.load("mysterydata/mysterydata3.npy")
    # print(np.mean(np.isfinite(data3)))
    # print(np.mean(np.isnan(data3)))
    dataNone = [plt.imsave(f"vis_3_{i}.png", data3[:, :, i], vmin = np.nanmin(data3[:, :, i]), vmax = np.nanmax(data3[:, :, i])) for i in range(9)]

```

```
    #Question 3 and 4
    data4 = np.load("mysterydata/mysterydata4.npy")
    data4_colormap = [colormapArray(f"vis_4_{i}.jpg", data4[:, :, i],
colors) for i in range(9)]
```

2.4 9 images from mysterydata4.npy



(a) 0th Channel (b) 1st Channel (c) 2nd Channel

(d) 3rd Channel (e) 4th Channel (f) 5th Channel

(g) 6th Channel (h) 7th Channel (i) 8th Channel

Figure 4: Mysterydata4.npy plot

3 Lights on a Budget

3.1 Naive Approach

3.1.1 Implement quantize(v,palette)

```
def quantize(v, palette):
    """
    Given a scalar v and array of values palette,
    return the index of the closest value
    """
    palette = np.array(palette)
    v = np.array(v)
    idx = np.argmin(np.abs(palette.reshape(-1, 1) - v.reshape(1, -1)),
   axis=0).astype(np.uint8)
    return idx
```

3.1.2 Implement quantizeNaive(IF,palette)

```
def quantizeNaive(IF, palette):
    """Given a floating-point image return quantized version (Naive)"""
    # quantizing multiple
    IF = np.array(IF)
    palette = np.array(palette)
    quantizedIdx = np.zeros_like(IF, dtype=np.uint8)
    H, W = IF.shape
    for i in range(H):
        for j in range(W):
            idx = quantize(IF[i, j], palette)
            quantizedIdx[i, j] = idx
    return quantizedIdx
```

3.1.3 Quantize Runtime

The reason why applying to gallery is slow is that the image size is much larger than gallery200, which increases a lot of computation cost.

3.1.4 Intensity Values vs Palette Values

Yes, low intensity value correspond to low palette value. the result of algo_fn() is index for palette, and the plot command will map 0 to lowest intensity map, 1 to highest intensity map.

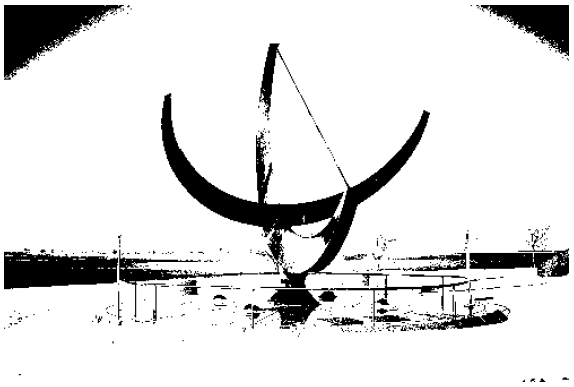
3.1.5 Two input/output pairs: aep.jpg + your choice

(a) Original

(b) Naive Method

Figure 5: aep.jpg with Naive method

(a) Original



(b) Naive Method

Figure 6: cosmos.jpg with Naive method

3.2 Floyd-Steinberg

3.2.1 quantizeFloyd()

```
def quantizeFloyd(IF, palette):
    """
    Given a floating-point image return quantized version (Floyd-Steinberg
    )
    """
    IF = np.array(IF)
    palette = np.array(palette)
    quantizedIdx = np.zeros_like(IF, dtype=np.uint8)
    H, W = IF.shape
    for i in range(H):
        for j in range(W):
```

```
            oldValue = IF[i, j]
            colorIndex = quantize(oldValue, palette)
            quantizedIdx[i, j] = colorIndex
            newValue = palette[colorIndex]
            error = oldValue - newValue
            if j + 1 < W:
                IF[i, j + 1] += error * 7/16
            if i + 1 < H:
                IF[i + 1, j - 1] += error * 3/16
                IF[i + 1, j] += error * 5/16
            if i + 1 < H and j + 1 < W:
                IF[i + 1, j + 1] += error * 1/16
    return quantizedIdx
```

3.2.2 Why does ditering work?

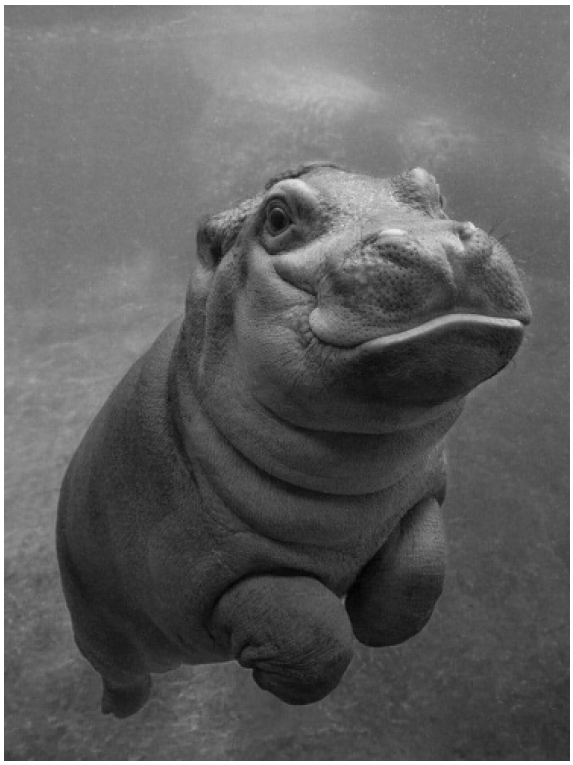
Dithering works by blending colors through pixels. When viewed from a distance or with reduced visual acuity, merge together to create the illusion of a wider palette of colors and smoother gradients.

3.2.3 3 results from gallery/ including aep.jpg

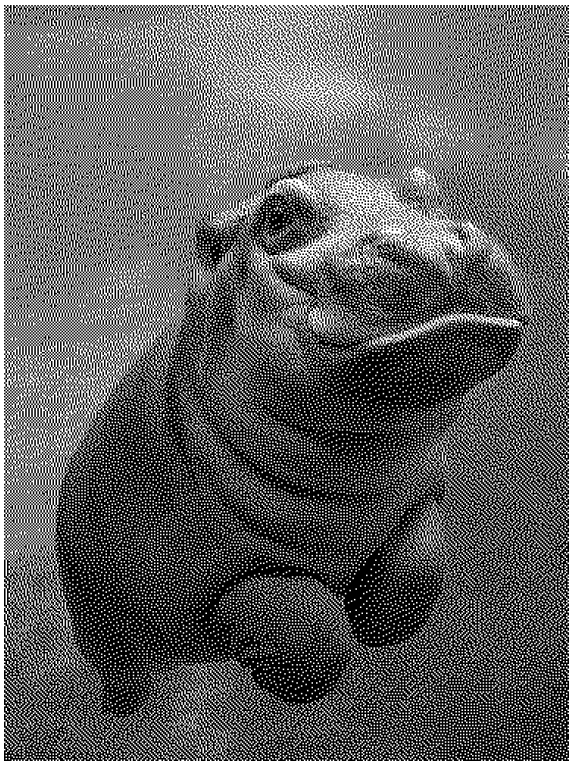
(a) Original

(b) Floyd dithering

Figure 7: aep.jpg with Floyd dithering



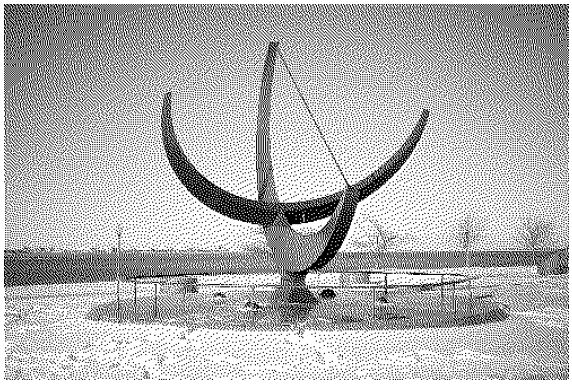
(a) Original



(b) Floyd dithering

Figure 8: babyhippo.jpg with Floyd ditering

(a) Original



(b) Floyd dithering

Figure 9: cosmos.jpg with Floyd ditering

3.3 Resizing Images

3.3.1 resizeToSquare()

```
def resizeToSquare(I, maxDim):
    """Given an image, make sure it's no bigger than maxDim on either side
    """
```

```
    H, W, C = I.shape
    if np.max((H, W)) > maxDim:
        down_ratio = maxDim / np.max((H, W))
        H_new = (H * down_ratio).astype(int)
        W_new = (W * down_ratio).astype(int)
        # pdb.set_trace()
        I = cv2.resize(I, (W_new, H_new), interpolation=cv2.INTER_LINEAR)
    return I
```

3.4 Handling Color

3.4.1 rewrite quantize()

```
def quantize(v, palette):
    """
    Given a scalar v and array of values palette,
    return the index of the closest value
    """
    palette = np.array(palette)
    v = np.array(v)
    idx = np.argmin(np.abs(palette.reshape(-1, 1) - v.reshape(1, -1)),
   axis=0).astype(np.uint8)
    return idx

def quantizeNaive(IF, palette):
    """Given a floating-point image return quantized version (Naive)"""
    # quantizing multiple
    IF = np.array(IF)
    palette = np.array(palette)
    quantizedIdx = np.zeros_like(IF, dtype=np.uint8)
    if len(IF.shape) < 3:
        H, W = IF.shape
    else:
        H, W, C = IF.shape
    for i in range(H):
        for j in range(W):
            idx = quantize(IF[i, j], palette)
            quantizedIdx[i, j] = idx
    return quantizedIdx
```

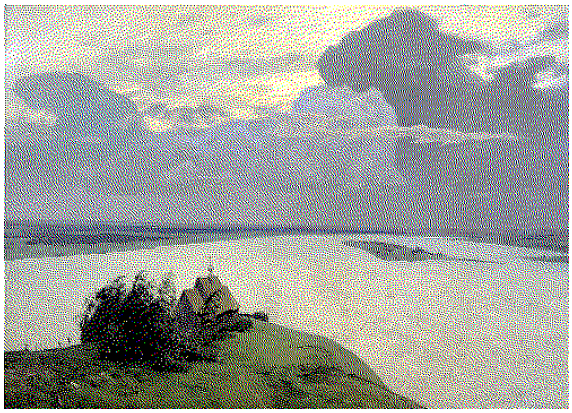
3.4.2 rewrite quantizeFloyd()

```
def quantizeFloyd(IF, palette):
    """
    Given a floating-point image return quantized version (Floyd-Steinberg
   )
    """
    IF = np.array(IF)
    palette = np.array(palette)
    quantizedIdx = np.zeros_like(IF, dtype=np.uint8)
    if len(IF.shape) < 3:
        H, W = IF.shape
```

```
    else:
        H, W, C = IF.shape
    for i in range(H):
        for j in range(W):
            oldValue = IF[i, j]
            colorIndex = quantize(oldValue, palette)
            quantizedIdx[i, j] = colorIndex
            newValue = palette[colorIndex]
            error = oldValue - newValue
            if j + 1 < W:
                IF[i, j + 1] += error * 7/16
            if i + 1 < H:
                IF[i + 1, j - 1] += error * 3/16
                IF[i + 1, j] += error * 5/16
            if i + 1 < H and j + 1 < W:
                IF[i + 1, j + 1] += error * 1/16
    return quantizedIdx
```

3.4.3 4 results

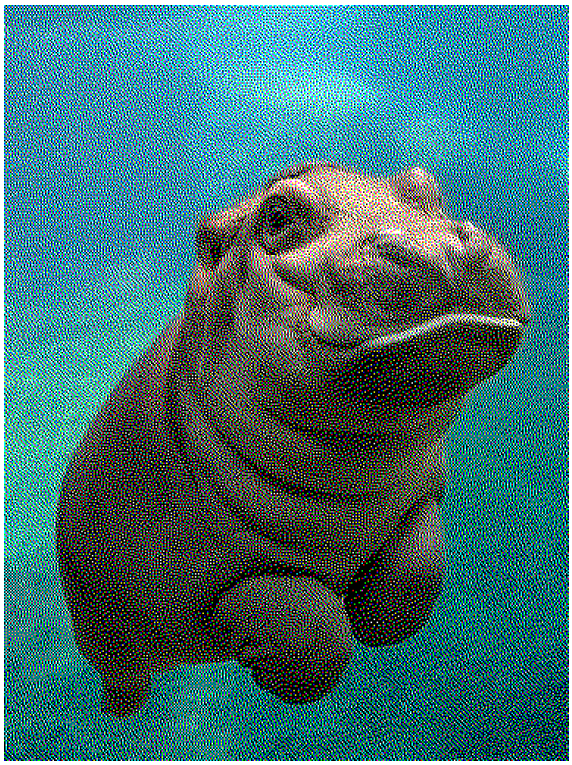
(a) Original



(b) Floyd dithering

Figure 10: Colored aep.jpg with Floyd dithering

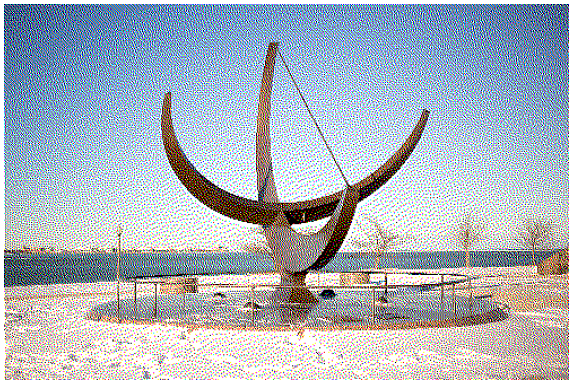
(a) Original



(b) Floyd dithering

Figure 11: Colored babyhippo.jpg with Floyd dithering

(a) Original



(b) Floyd dithering

Figure 12: Colored cosmos.jpg with Floyd dithering

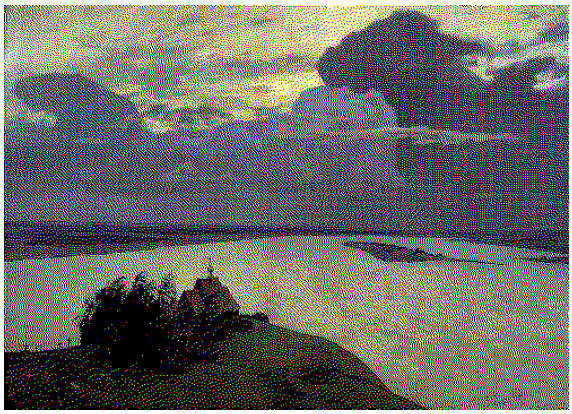
(a) Original

(b) Floyd dithering

Figure 13: Colored cossacks.jpg with Floyd dithering

3.5 Gamma Correction

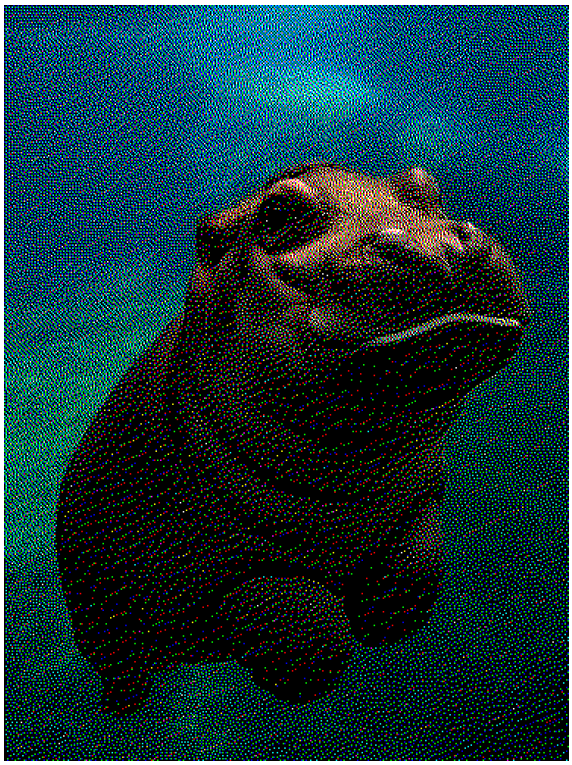
(a) Original



(b) Floyd dithering (Gamma correction)

Figure 14: Colored aep.jpg with Floyd dithering (Gamma correction)

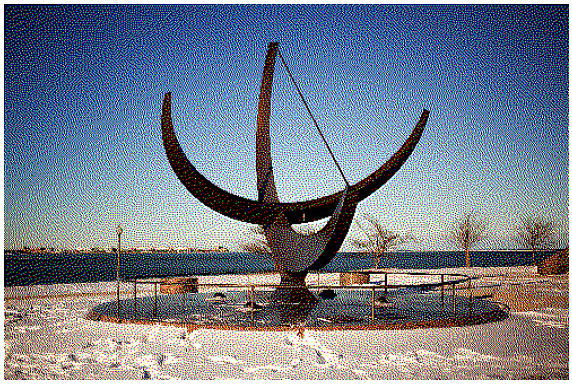
(a) Original



(b) Floyd dithering (Gamma correction)

Figure 15: Colored babyhippo.jpg with Floyd dithering (Gamma correction)

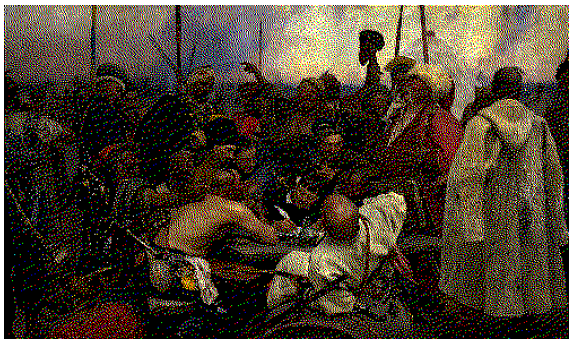
(a) Original



(b) Floyd dithering (Gamma correction)

Figure 16: Colored cosmos.jpg with Floyd dithering (Gamma correction)

(a) Original



(b) Floyd dithering (Gamma correction)

Figure 17: Colored cossacks.jpg with Floyd dithering (Gamma correction)

4 Colorspaces

4.1 RGB Plots

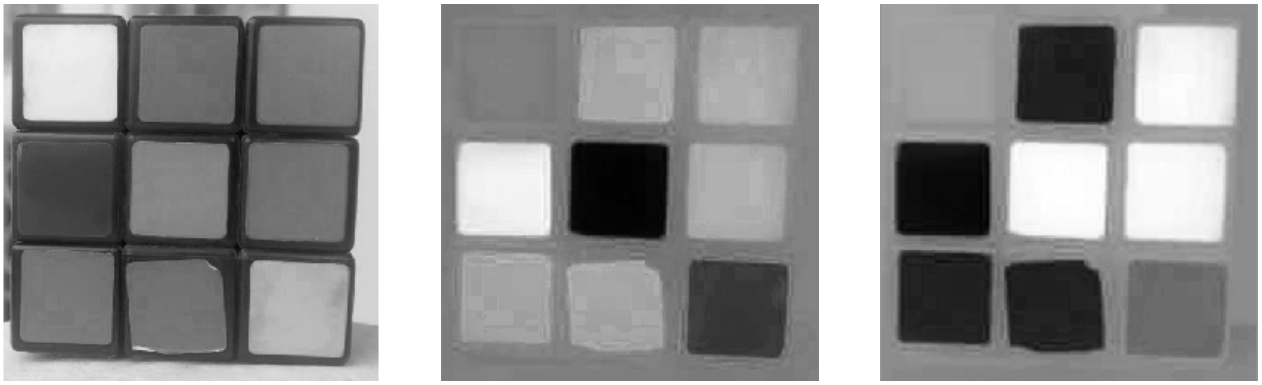
```
import numpy as np
import matplotlib.pyplot as plt
import cv2
import pdb
import argparse
import os

if __name__ == "__main__" :
    print(f"Runing $ python rubik.py <source> <target> <cvttolab> \nPlease
    give the input. For <cvttolab>, 0 for use RGB, 1 for use LAB")

    parser = argparse.ArgumentParser()
    parser.add_argument('source')
    parser.add_argument('target')
    parser.add_argument('cvttolab', type=int, help="0 for RGB, 1 for LAB")
    args = parser.parse_args()

    if not os.path.exists(args.target):
        os.mkdir(args.target)

    # get image names
    images = [fn for fn in os.listdir(args.source) if fn.endswith(".png")]
    images.sort()

    # process images
    result = {}
    for image in images:
        if image not in result:
            result[image] = []

        I = cv2.imread(os.path.join(args.source, image))

        # RGB
        if not args.cvttolab:
```

```
            for i in range(I.shape[2]):
                result[image].append(I[:, :, i])
        # LAB
        else:
            I_LAB = cv2.cvtColor(I, cv2.COLOR_RGB2LAB)
            for i in range(I_LAB.shape[2]):
                result[image].append(I_LAB[:, :, i])

    # show images
    color = 'RGB'
    if args.cvttolab:
        color = 'LAB'

    for image in result:
        [plt.imsave(f'result/{image}_{color}_{i}.png',image_channel, cmap='gray') for i, image_channel in enumerate(result[image])]

    print(f'Image saved!')
```

4.2 LAB Plots

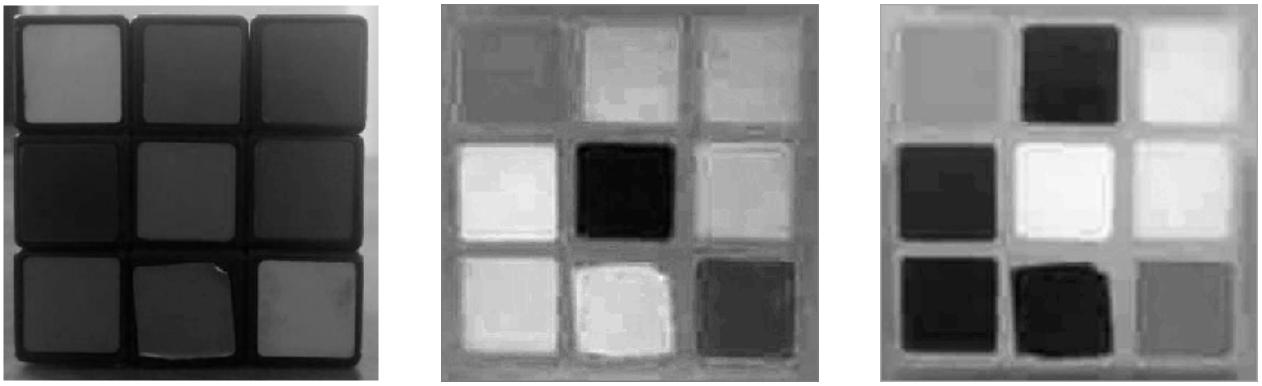
Shown in Section 4.1.

4.3 RGB vs LAB



(a) L channel (b) A channel (c) B channel

Figure 18: Rubik outdoor.png in LAB color space

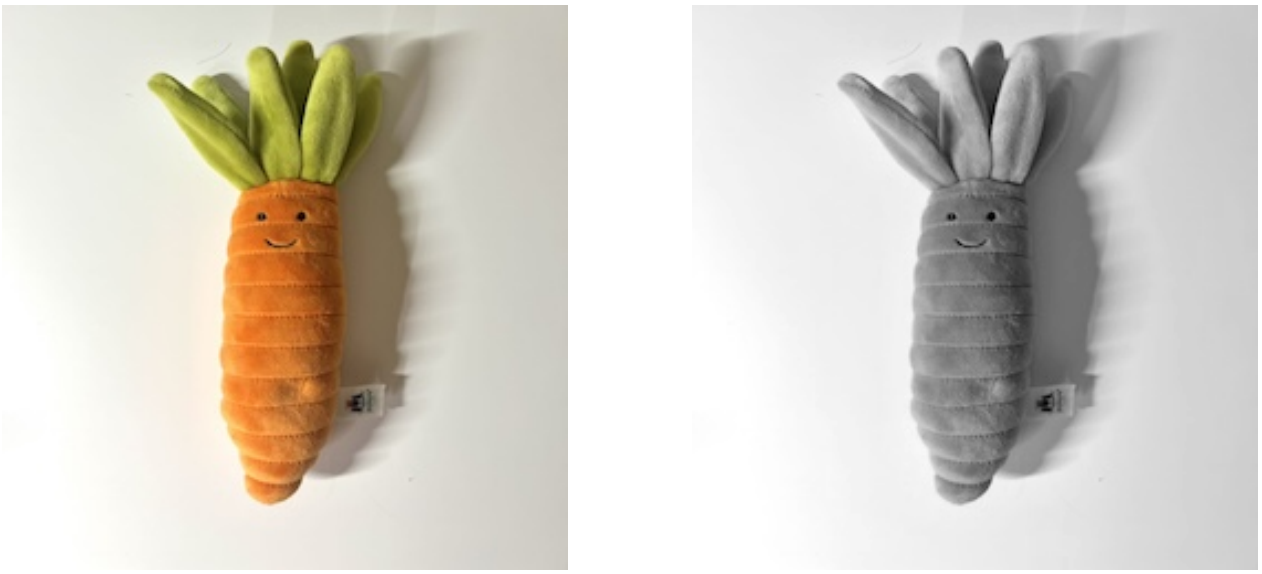


(a) L channel (b) A channel (c) B channel

Figure 19: Rubik indoor.png in LAB color space

The LAB color space is better at separating illuminance compared to the RGB color space. LAB is designed to be more perceptually uniform, meaning a change of the same amount in a color value should result in about the same amount of visual change. L Channel Represents luminance, ranging from black to white. This channel separates the lightness information from the color information. In contrast, for RGB, changes in luminance are not separated from changes in color. Adjusting the brightness of an image in RGB affects the color values directly and can alter the perceived hue and saturation.

4.4 Two images and their Luminance plot



(a) Original (b) L channel

Figure 20: Carrot(Bright)

(a) Original

(b) L channel

Figure 21: Carrot(Dark)

Submitted by Wensong Hu on January 31, 2024.