KENNESAW STATE
UNIVERSITY

CS 4732
MACHINE VISION

PROJECT 1

IMAGE RESOLUTION

INSTRUCTOR

Dr. Mahmut KARAKAYA

Michael Rizig
001008703

1. ABSTRACT

In this project, we are given 3 variables to mainipulate in images: Sampling, Qualtization, and Color Quantization. For the first file of this project (Sampling), we are given and 1024x1024 image and downsample the image into resolutions of 512x512, 256x256, 128x128, 64x64, and finally 32x32, then upsample the image step by step back to 1024. This downsampling was done by selecing the first pixel in every unique neighborhood of 4 and using that pixel in the output image for each level. By doing this, we half the veritcal and horizontal resolution by 2 each, giving us an output half the size as the input. In the second file (Quantization) we are downscaling the greylevel of an image from 8 bit (256 grey levels) down to 1 bit (2 grey levels), halving the numeber of levels in each step. To do this, we take the input greyscale levels (256) and divide it by 2 to the power of how many bits we want, giving us the grey level. We then take this ratio, take the floor of it, and multiply it back by the same ratio to give us the new lower count of greylevels. Finally for the third file (Color Quantizaton) we are essentially doing the same thing as greyscale downscaling, but for each color channel of the image (R,G,B).

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 Test Results for Downsampling

All images at each level downscale are included in zip file output>sampling. These are just a few selected images to highlight effect. (each step begins with original image)

Image 1a : Original Image at 1024x1024 resolution. **This is the input image.**

Image 1b: After 2 downsamples down to 256x256, we see that we lose detail in the finer aspects of the image (such as the whiskers)

İmage 1c: After 2 more downsamples, we get an image at 64x64, and at this point we see clear signs of artificating around the whiskers and eyes, and hairs around the ears.

İmage 1d: After another downsample, we arrive at an image at 32x32 and we can clearly see we have lost almsot all fine details, and are left with a very absrtact image of a cat.

2.2 Test Results for Upsampling

All images at each level upscale are included in zip file output>sampling. These are just a few selected images to highlight effect. (each step begins with original image)

Image 2a : We begin at the last downscaled image (32x32). **This is the input image for upsampling.**

Image 2b: After the first upsample up to 64x64, we see that although we have more pixels we are not gaining any new details in the image, and we essentally have the same image with slightly less blur.

İmage 2c: After 2 more upsamples, we get an image at 256x256, and at this point we can clearly see that upsampling the image without inserting any new pixels gives no added detail, just less blur.

İmage 2d: After another two more upsamples, we end with the final resolution of 1024x1024, and we can see that it is just as lacking in detail but we have more pixels lending to slightly less blur.

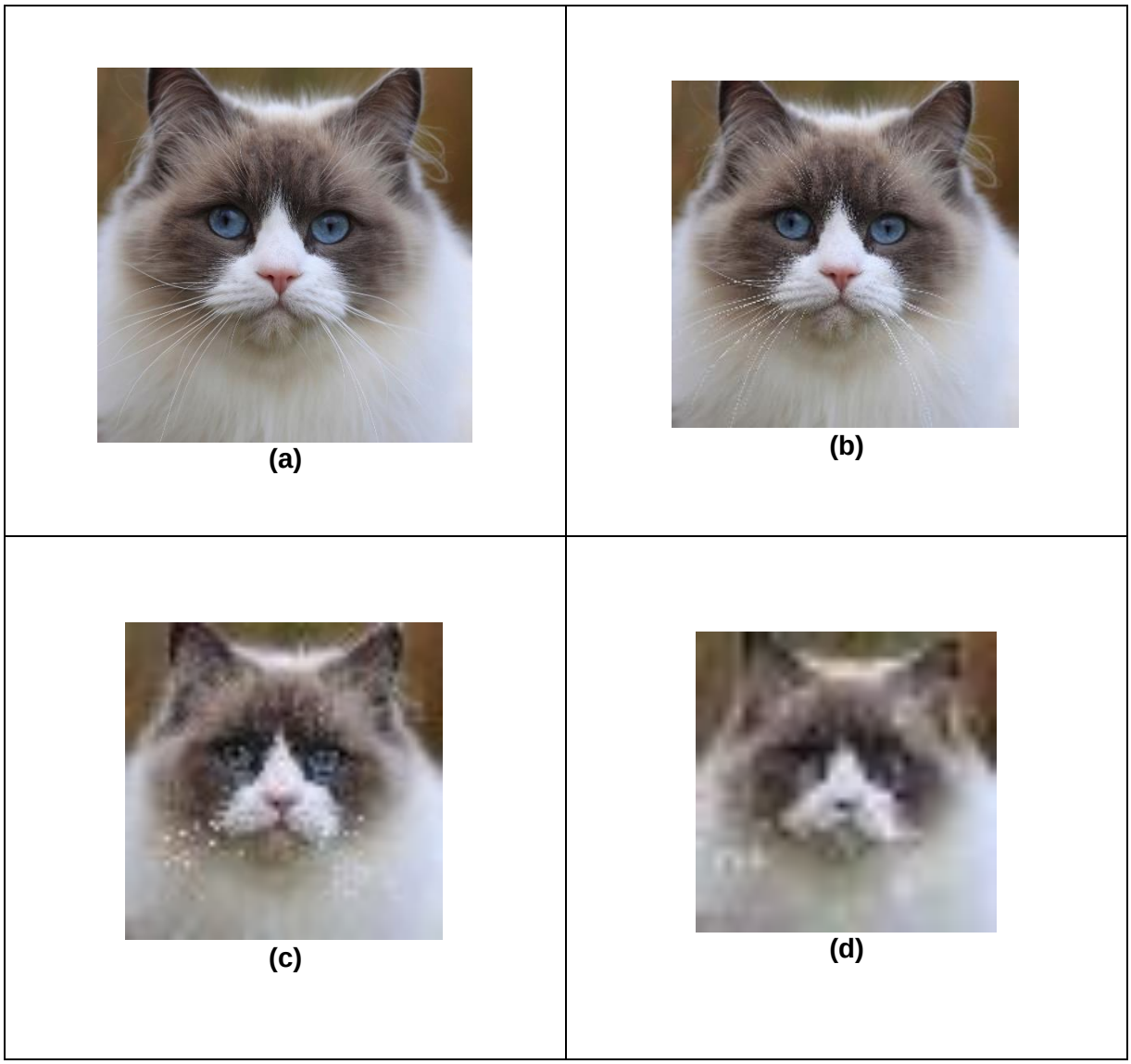

Figure 1: (a) Original cat image (input/cat.jpg),(b) cat downsampled to 256x256 pixels (output/sampling/downscale-2.jpg), (c) cat downscaled to 64x64 pixels (output/ sampling/downscale-4.jpg), (d) cat downscaled to 32x32 pixels (output/ sampling/downscale-5.jpg).

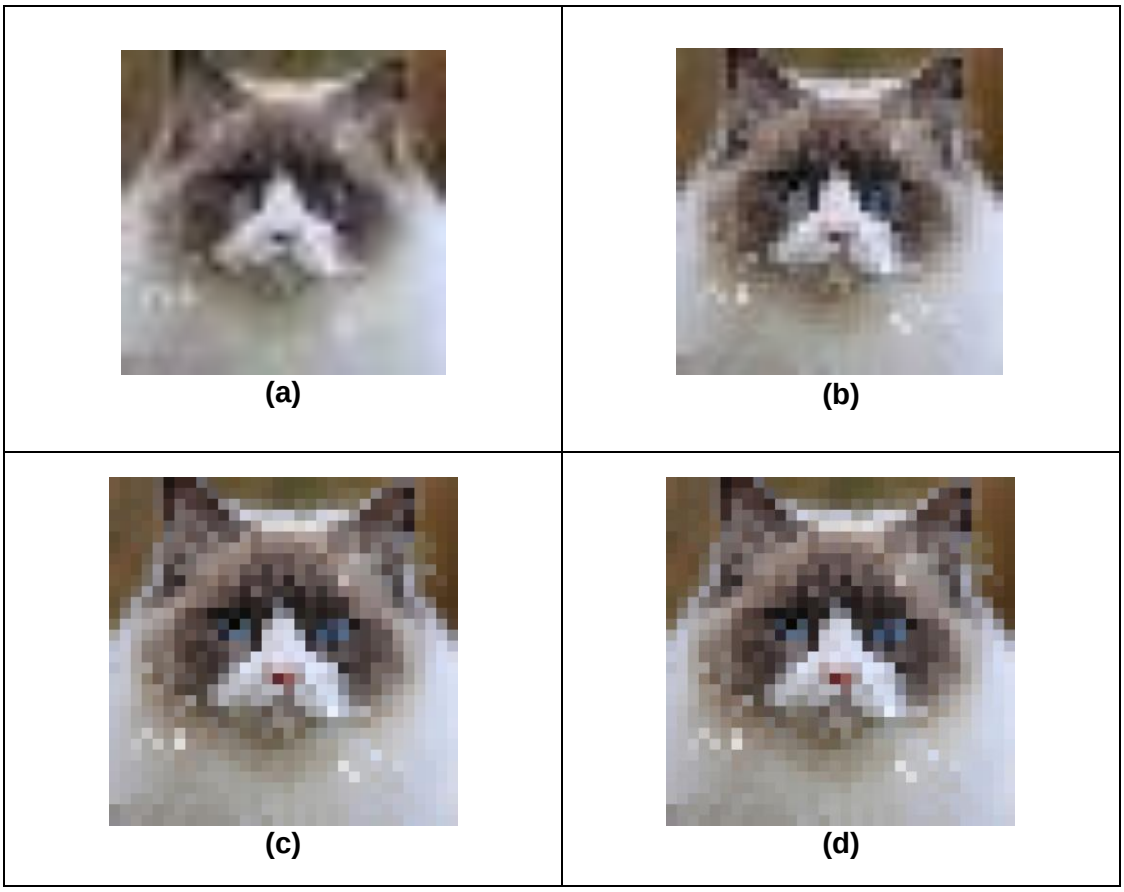

Figure 2: (a) Original cat image from downscaling (output/downscale-5.jpg),(b) cat upsampled to 64x64 (output/upscale-1.jpg), (c) cat upscaled to 256x256 pixels (output/upscale-3.jpg), (d) cat upscaled to 1024x1024 (output/upscale-5.jpg).

2.3 Test Results for Image Quantization (Greyscale)

All images at each grey level are included in zip file output>grey. These are just a few selected images to highlight effect. (each step begins with original image)

Image 3a : Original Greyscale image at 256 grey levels (8 bit) . **This is the input image.**

Image 3b: After the first quantization, we reduce the number of grey levels from 256 to 128 (8 bit to 7 bit) we see that we lose some detail in the background, but very minimal and easy to miss.

İmage 3c: After another reduction we take the image down from 128 to 64 grey levels (7 bit to 6 bit). We can see some considerable detail loss in the apple and background.

İmage 3d: After 2 more reductions, we take the image down from 64 to 16 grey levels (6 bit to 4 bit). We can see some considerable resolution loss.

İmage 3e: After another 2 reductions, we take the image from 16 to 4 grey levels (4 bit to 2 bit)

İmage 3f: finally, we take the image all the way to the minimum greyscale bits (1), utilizng only 2 levels.

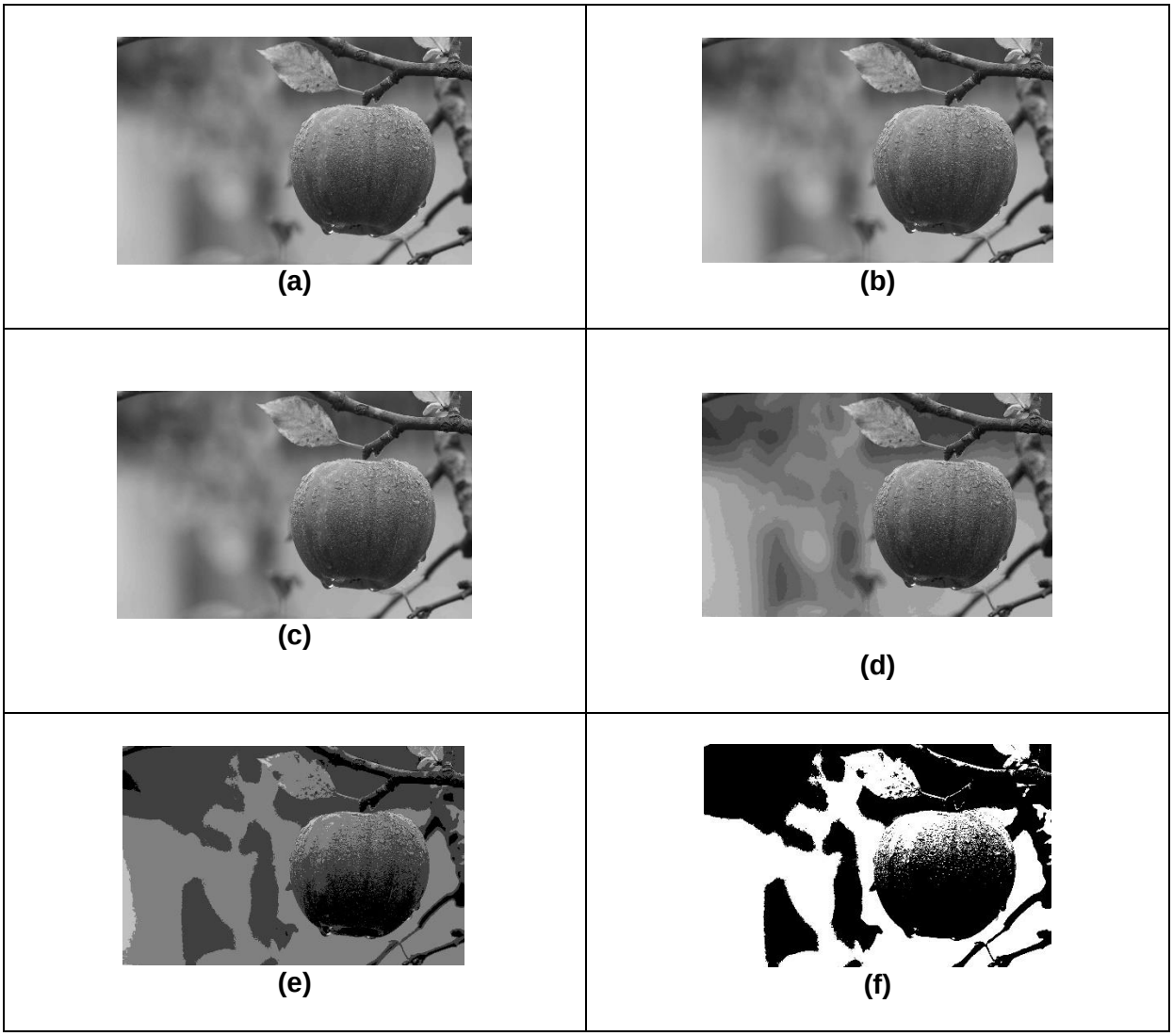

Figure 3: (a) Original Greyscale image at 256 grey levels (8 bit) (input/apple.jpeg),(b) apple reduced from 8 bit to 7 bit grey levels(output/grey/greyscale7-bits.jpeg), (c) apple reduced from 7 to 6 bits grey level(output/grey/greyscale6-bits.jpeg), (d) apple reduced to 4 bits greyscale (output/grey/greyscale4-bits.jpeg), (e) apple reduced to 2 bits (output/grey/greyscale2-bits.jpeg), (f) apple reduced to 1 bit (output/grey/greyscale1-bits.jpeg)

2.4 Test Results for Image Color Quantization

All images at each color level are included in zip file output>color. (each step begins with original image)

Image 4a : Original color image at 2^24 color levels (8bit R, 8-bit G, 8-bit B) . **This is the input image.**

Image 4b: After the first quantization, we reduce the number of colors from 2^24 to 2^12 (4-bit R, 4-bit G, 4-bit B). At this point the image is not too differnet, but slightly noticable.

İmage 4c: After the next quantizaton, we reduce the number of colors from 2^24 to 2^6 (2-bit R, 2-bit G, 2-Bit B). At this point, color distortion is very noticable and the image loses quality bascally everywhere.

İmage 4d: At the final quantization, we reduce the image from 2^24 down to 2^3 (1-bit R, 1-bit G, 1-bit B). At this point, all values that are not close to a solid primary color (R, G or B) are combined into black and white, or Cyan, Yellow, Magenta..

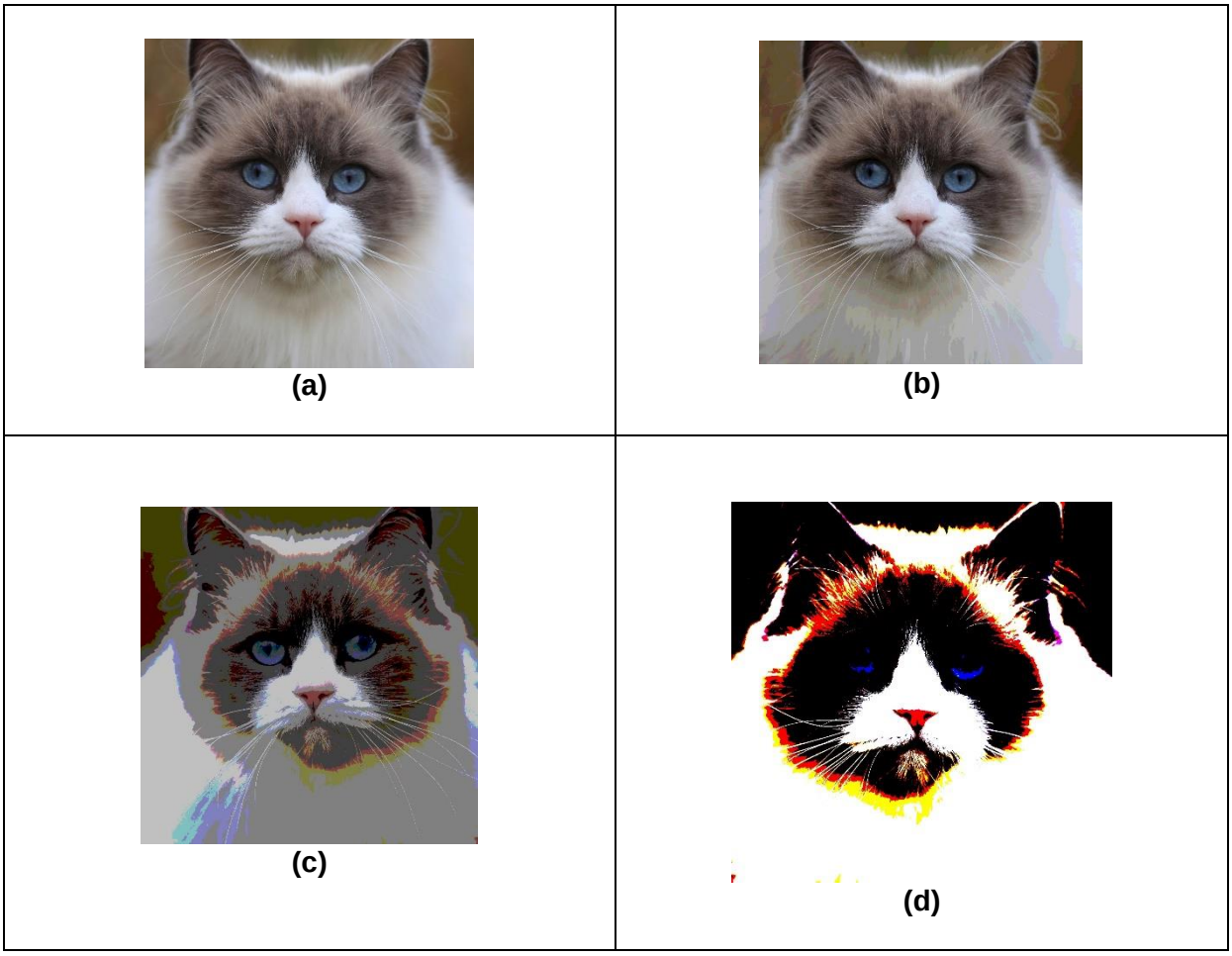

Figure 4: (a)Original Cat image, with 2^24 colors (input/cat.jpg),(b) cat reduced to 2^12 color levels(output/color/downcolor12bits.jpg), (c) cat reduced to 6 bit color(output/color/downcolor6bits.jpg), (d) cat redced to 3 bit color (output/color/downcolor3bits.jpg).

3. DİSCUSSİON

In this project, we learned many things about each of the tested factors of an image and how they effect the overall look of an image. For sampling, we can clearly see that

downsampling an image to a lower resolution loses a lot of detail. These details can not be regained by simply upscaing the image, and you would have to insert new pixels into the gaps to upscale an image. For greyscale quantization, we learn that as we decrease the grey levels of an image, we lose a lot of finer details in the background, and areas of the image with a lot lof closely colored pixels (such as the apple in our case). We also can note that we lose a lot of the grey resolution details and levels, which we cannot gain back. Finally, for the color quantization, we learn that as we lose levels of colors, our image resorts closer and close to a primary colors image, and by the very end we lose alot of details and see the image consists of only primary and combinaton colors (R,G,B, M, Y, C) and black/white. Given more time, it would be interesting to see how this effect works with images with dfferent aspect ratios, or extremely high resolutions (like 8K +). Overall, I've learned a lot from this project.

4. CODE

4.1 Code for Image Sampling (sampling.py)

```
# Michael Rizig
# Project 1: Digital Image Processing
# 001008703
# File 1: Sampling
# 6/4/2024

#nessessary imports:
from skimage import io
import matplotlib.pyplot as plottool
import numpy as np
import cv2

#read in image:
catImagePath = 'input/cat.jpg'
alqita = io.imread(catImagePath)

#obtain shape of image matrix:
shape = alqita.shape

#start by setting dowsample factor to 2
factor=2

#create output image as a numpy 3d 0's array:
xSampledImage = np.zeros((int(shape[0]/factor),int(shape[1]/factor),3),
dtype=np.uint8)
```

```
#print image dimentions:
print("Input image dimentions: ",shape)


#display original image:
plottool.imshow(alqita)
plottool.show()
#loop through to show 5 different scales:
for k in range(5):
    #print current dimensions:
    print("Current dimentions: ",xSampledImage.shape[0],
"x",xSampledImage.shape[1])

    #currentPixel=image[0,0]
    #iterating through the entire image
    for i in range(0,shape[0]-1,factor):
        for j in range(0,shape[1]-1,factor):
                # we are setting the output pixel to the given input pixel
                # we divide by the factor when accessing the output image becase
the output image is smaller, so a 1:1 access would cause out of bounds exception
                xSampledImage[int(i/factor),int(j/factor)]=alqita[i,j]

    #show result of current downscale:
    plottool.imshow(xSampledImage)
    plottool.show()
    #shift colors to rgb
    xSampledImage = cv2.cvtColor(xSampledImage, cv2.COLOR_BGR2RGB)
    cv2.imwrite(f'output/sampling/downscale-{k+1}.jpg',xSampledImage)
    #increaes downscale:
    factor=factor*2
    #reset output image:
    if(k!=4):
        xSampledImage = np.zeros((int(shape[0]/factor),int(shape[1]/factor),3),
dtype=np.uint8)

#now we upscale the image in the same fashon as before:
print("---end of downsampling, begining of upsampling---")

#save final downsampled image as new starting image
alqita = xSampledImage.copy()

#reset variables
shape = alqita.shape
factor=2
```

```
xSampledImage = np.zeros((int(shape[0]*factor),int(shape[1]*factor),3),
dtype=np.uint8)

for k in range(5):
    #print current dimensions:
    print("Current resolution: ",xSampledImage.shape[0],
"x",xSampledImage.shape[1])

    #currentPixel=image[0,0]
    # since we are upscaing, we need to mutliply the range by our factor to
ensure our output image has the correct number of pixels
    for i in range(0,shape[0]*factor,1):
        for j in range(0,shape[1]*factor,1):
                # since we are making a smaller image bigger, we need to divide
the equivalent pixel i and j value by the factor of incresae so we dont get a
memory segmentation falut trying to acccess the image
                # convert each value to integer incase we have an input image
that is not square and we get a float value after divison
                xSampledImage[i,j]=alqita[int(i/factor),int(j/factor)]

    #show result of current upscale:
    plottool.imshow(xSampledImage)
    plottool.show()
    #shift colors to rgb
    #output = cv2.cvtColor(output, cv2.COLOR_BGR2RGB)
    cv2.imwrite(f'output/sampling/upscale-{k+1}.jpg',xSampledImage)
    #increaes upscale:
    factor=factor*2
    #reset output image:
    if(k!=4):
        xSampledImage = np.zeros((int(shape[0]*factor),int(shape[1]*factor),3),
dtype=np.uint8)
```

4.2 Code for Image Quantization (Quantization.py)

```
# Michael Rizig
# Project 1: Digital Image Processing
# 001008703
# File 2: Quantization
# 6/6/2024

# imports
from skimage import io
```

```
import matplotlib.pyplot as plottool
import cv2

# this function takes in desired bit count and a given pixels grey value
# and returns a conversion into the passed in bitcount greyscale value
def filter(bits, x):
    # base case: if image is being converted into binary, simply check and return
1 or 0
    if bits==1:
        if x>127:
            return 255
        return 0
    # for every other case: divide 256 (starting value) by number of greyscale
levels desired
    val = 256 / 2**bits
    # by dividing current pixel by ratio of 256/greyscale levels and convering
from float to integer, we are essentially taking the floor of that value
    # by multiplying it by the original value, we lose any deviation from a even
multiple, giving us n=bits possible outputs, properly scaled
    # example: x=126; bits=4
    # val = 256/16 = 16
    # this means we only have 16 possible values for greyscale, those being :
0,16,32,48,64,80,96,112,128,144 etc
    # because of this fact, we are mimicing an n-bit greyscale by only displaying
2**n grey levels
    # 126 / 16 = 7.875; convering to int, 7.875 = 7
    # finally multiplying by 16 gives us 112.
    return val * int(x*(1/val))

# define path for image
path = 'input/apple.jpeg'
# read in image into memory
image = io.imread(path)
# convert image from brg to rgb
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)
# create output image
out = image.copy()
# display initial image
print("Initial image: ")
plottool.imshow(image)
plottool.show()

for divisor in range (7,0,-1):
    for k in range(0,image.shape[0]):
        for j in range(0, image.shape[1]):
```

```
            v = filter(divisor,int(image[k][j][0]))
            out[k][j] = [v,v,v]
    print("Image greyscaled down from ", divisor+1 , " bits to " , divisor, "
bits: ")
    # display current image
    plottool.imshow(out)
    plottool.show()
    # save current image with unique name
    cv2.imwrite(f'output/grey/greyscale{divisor}-bits.jpeg',out)
    # reset out image to original for next pass
    out = image.copy()
```

4.3 Code for Image Color Quantization (ColorQuantization.py)

```
# Michael Rizig
# Project 1: Digital Image Processing
# 001008703
# File 3: Color Quantization
# 6/8/2024

# imports
from skimage import io
import matplotlib.pyplot as plottool
import cv2
# this function takes in desired bit count and a given pixels grey value from
file 2
# we are repurposing it for this file to change color levels
def filter(bits, x):
 # base case: if pixel is being converted into binary, simply check and return 1
or 0
    if bits==1:
        if x>127:
            return 255
        return 0
    val = 256 / 2**bits
    # by dividing current pixel by ratio of 256/levels and convering from float
to integer, we are essentially taking the floor of that value
    # by multiplying it by the original value, we lose any deviation from a even
multiple, giving us n=bits possible outputs, properly scaled
    # example: x=126; bits=4
    # val = 256/16 = 16
    # this means we only have 16 possible values for greyscale, those being :
0,16,32,48,64,80,96,112,128,144 etc
```

```
    # because of this fact, we are mimicing an n-bit greyscale by only displaying
2**n grey levels
    # 126 / 16 = 7.875; convering to int, 7.875 = 7
    # finally multiplying by 16 gives us 112.
    return val * int(x*(1/val))

#grab inital image same way as files 1 and 2
path = 'input/cat.jpg'
baseImage = io.imread(path)

#show initial image
plottool.imshow(baseImage)
plottool.show()

#create output image
out = baseImage.copy()

#color shift if needed
#cv2.cvtColor(baseImage,cv2.COLOR_BGR2RGB)

#define a divisor vector to contain each level of colors we desire
divisor = [12,6,3]

#loop through colors
for k in range (0,3):
    #loop through image pixels
    for i in range(baseImage.shape[0]):
        for j in range(baseImage.shape[1]):
            #loop through each color channel
            for pixel in range(0,3):
                #update each channel to fit given number of colors for each
(divisor/3 since we have 3 colors)
                out[i][j][pixel] = filter(divisor[k]/3,baseImage[i][j][pixel])
    print('Image color levels from ',divisor[k]*2, " to " , divisor[k])
    #plot image
    plottool.imshow(out)
    #display image
    plottool.show()
    #convert colors
    out = cv2.cvtColor(out, cv2.COLOR_BGR2RGB)
    #save image with appropriate name
    cv2.imwrite(f'output/color/downcolor{divisor[k]}bits-{k}.jpg',out)
    # restore output image to original quality for next itteration
    out=baseImage.copy()
```