

Sean Carda  
PID: A59009786  
ECE 253 – Image Processing  
October 17, 2021  
Homework 1

*Academic Integrity Policy: Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual to whom it is assigned, without unauthorized aid of any kind.*

*By including this in my report, I agree to abide by the Academic Integrity Policy mentioned above.*

Problem 1:

**Part i:**

*Code:*

```
# a)

# Create the matrix A.
A = np.array([[3, 9, 5, 1],
              [4, 25, 4, 3],
              [63, 13, 23, 9],
              [6, 32, 77, 0],
              [12, 8, 6, 1]])

# Create the matrix B.
B = np.array([[0, 1, 0, 1],
              [0, 1, 1, 0],
              [0, 0, 0, 1],
              [1, 1, 0, 1],
              [0, 1, 0, 0]])

# Create a matrix C by point-wise multiplying A and B.
C = np.multiply(A, B)

# Display the matrix C.
print("\nMatrix Result C:")
print(C)
```

*Output:*

```
Matrix Result C:
[[ 0  9  0  1]
 [ 0 25  4  0]
 [ 0  0  0  9]
 [ 6 32  0  0]
 [ 0  8  0  0]]
```

## **Part ii:**

### *Code:*

```
# b)

# Calculate the inner product between the 2nd and third rows of the resulting matrix C.
in_prod = np.dot(C[1, :], C[2, :])
print("\nResult of Inner Product:")
print(in_prod)
```

### *Output:*

```
Result of Inner Product:
0
```

## **Part iii:**

### *Code:*

```
# c)

# First, find the maximum and minimum values in the matrix C.
c_max = np.max(C)
c_min = np.min(C)

# Display the maximum and minimum to confirm that these are the desired values.
print("\nThe maximum of the Matrix C is: " + str(c_max))
print("The minimum of the Matrix C is: " + str(c_min))

# Now that the max and min have been found, use numpy to locate the indices of the maximum
# and minimum values.
index_max = np.where(C == c_max)
index_min = np.where(C == c_min)

# Print the indices that were found for the max and min values.
print("Max Indices: ")
print("row: " + np.array2string(index_max[0]))
print("col: " + np.array2string(index_max[1]))
print("Min Indices: ")
print("row: " + np.array2string(index_min[0]))
print("col: " + np.array2string(index_min[1]))
```

### *Output:*

```
The maximum of the Matrix C is: 32
The minimum of the Matrix C is: 0
Max Indices:
row: [3]
col: [1]
Min Indices:
row: [0 0 1 1 2 2 2 3 3 4 4 4]
col: [0 2 0 3 0 1 2 2 3 0 2 3]
```

## Problem 2:

### **Part i:**

*Code:*

```
# i)

# Establish the file path to the chosen image and read it into the parameter A.
file_path = r"astronaut.jpg"
A = cv2.imread(file_path)

# Show the original image.
cv2.imshow('Original Image A', A)
cv2.imwrite('Original_Image_A.jpg', A)
cv2.waitKey(0)
```

### **Part ii:**

*Code:*

```
# ii)

# Convert the color image A to a grayscale image.
B = cv2.cvtColor(A, cv2.COLOR_BGR2GRAY)

# Compute the max and min of the image to ensure that the values are between 0 and 255.
max = np.max(B)
min = np.min(B)
print("\nThe max value of the Grayscale image is " + str(max))
print("The min value of the Grayscale image is " + str(min))

# Show the grayscale image.
cv2.imshow('Grayscale Image B', B)
cv2.imwrite('Grayscale_Image_B.jpg', B)
cv2.waitKey(0)
```

*Grayscale Verification:*

```
The max value of the Grayscale image is 242
The min value of the Grayscale image is 5
```

### **Part iii:**

*Code:*

```
# iii)

# Now, create a matrix the same size as the image B filled with the value: 15.0
size = B.shape
constant_matrix = np.full([size[0], size[1]], 15, dtype=np.uint8)

# Add 15 to the grayscale image and display it.
C = B + constant_matrix

# Show the brightened image.
cv2.imshow('Image Increased by 15 Intensity Levels', C)
cv2.imwrite('Image_Increased_by_15_Intensity_Levels.jpg', C)
cv2.waitKey(0)
```

## **Part iv:**

*Code:*

```
# iv)

# Flip the image C along both the horizontal and vertical axis.
D = np.flip(B, axis = [0, 1])

# Show the flipped image.
cv2.imshow('Flipped Image', D)
cv2.imwrite('Flipped_Image.jpg', D)
cv2.waitKey(0)
```

## **Part v:**

*Code:*

```
# v)

# First, find the median of the matrix B.
B_median = np.median(B)

print("\nThe median of the image is: " + str(B_median))

# Then, once the matrix has been found, create a binary matrix where any value greater
# than the median to 0, and anything less than or equal to the median to 1.
E_binary = np.where(B > B_median, 0, 1)

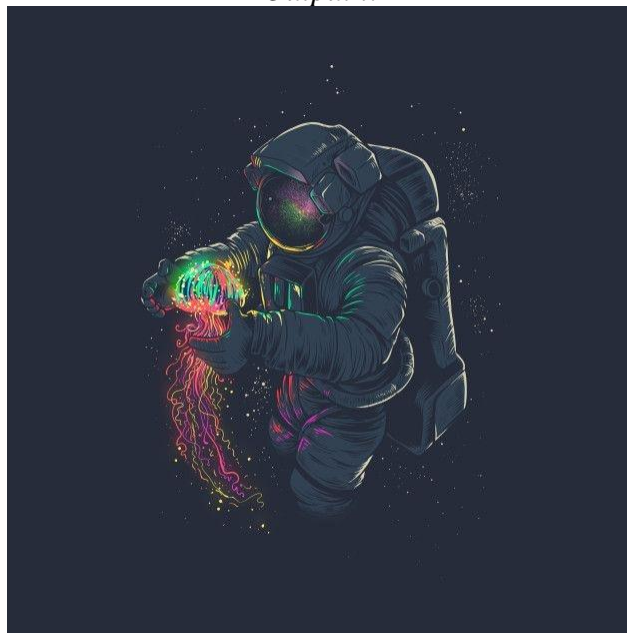
# For display purposes, convert the image to uint8 with intensities of 0 and 255.
E = np.uint8(255 * E_binary)

# Show the flipped image.
cv2.imshow('Binary Image E', E)
cv2.imwrite('Binary Image E.jpg', E)
cv2.waitKey(0)
```

*Verify Median:*

```
The median of the image is: 44.0
```

*Output i:*



*Output ii:*



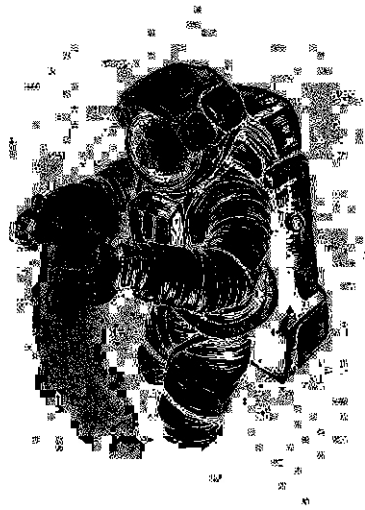
*Output iii:*



*Output iv:*



*Output v:*



### Problem 3:

#### Code:

```
#-----
# PROBLEM 3
#-----
cv2.destroyAllWindows()

def compute_norm_rgb_histogram(image):
    # Instantiate a collection of bins for each color channel in the image.
    bin_count = 32
    bins_red = np.zeros(bin_count)
    bins_green = np.zeros(bin_count)
    bins_blue = np.zeros(bin_count)

    # Grab the dimensions of the image for looping.
    size = image.shape

    # Grab each layer of the image and store it in a color-specific matrix. Since openCV
    # reads in images in BGR format, the indices 0 = Blue, 1 = Green, 2 = Red
    image_blue = image[:, :, 0]
    image_green = image[:, :, 1]
    image_red = image[:, :, 2]

    # Iterate through the image layers and sort them based on pixel intensity.
    for i in range(0, size[0]):
        for j in range(0, size[1]):
            # The mapping for this function maps every pixel into their respective bin except
            # for 255, which gets mapped to bin 32. So, if the bin is calculated to be greater
            # than 32, set the bin to 31.
            index_red = floor(image_red[i, j] * 32/255)
            if index_red >= 32:
                index_red = 31

            # Intensity cap for the green layer.
            index_green = floor(image_green[i, j] * 32/255)
            if index_green >= 32:
                index_green = 31

            # Intensity cap for the blue layer.
            index_blue = floor(image_blue[i, j] * 32/255)
            if index_blue >= 32:
                index_blue = 31

            # Using the bin values that were calculated previously, increase the histogram
            # value at that bin by 1.
            bins_red[index_red] += 1
            bins_green[index_green] += 1
            bins_blue[index_blue] += 1

    # Normalize the histograms for each respective color layer.
    bins_red /= sum(bins_red)
    bins_green /= sum(bins_green)
    bins_blue /= sum(bins_blue)

    # Concatenate the bins into one total histogram for the entire image.
    bins_total = np.concatenate((bins_red, bins_green, bins_blue))

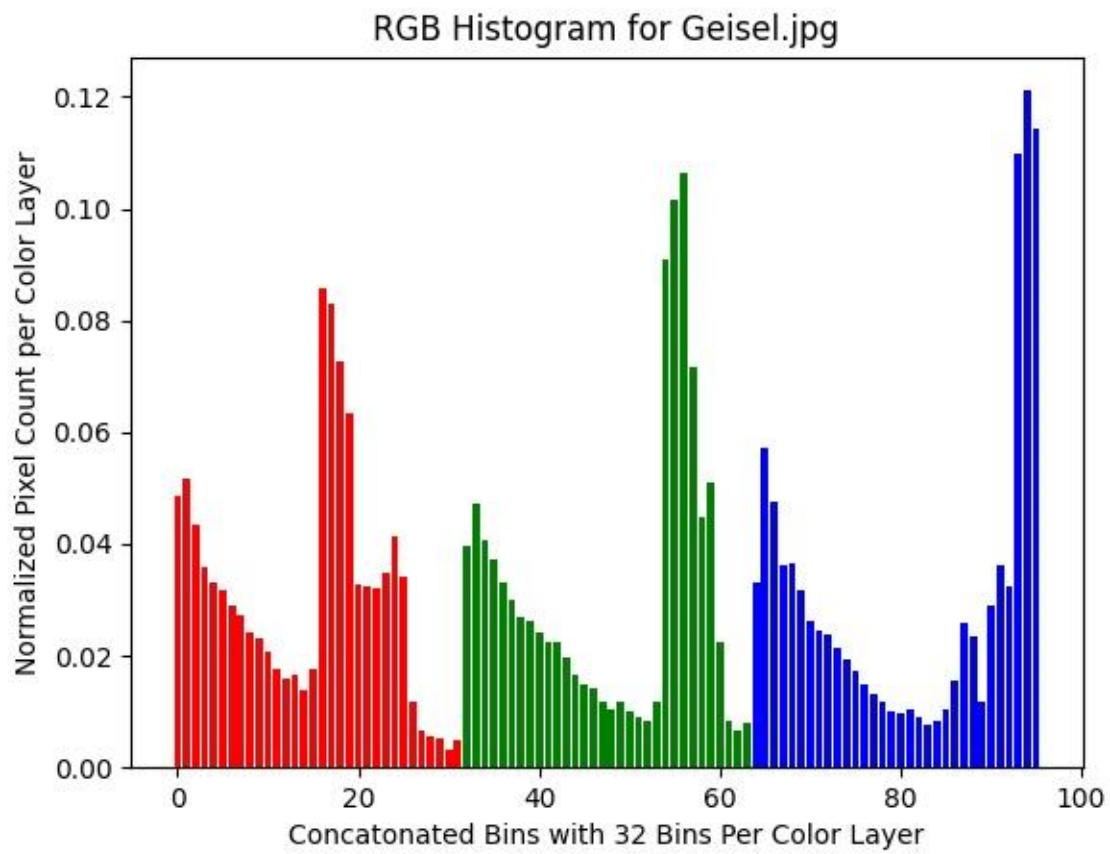
    # Finally, return the total histogram.
    return bins_total

# Read in the color jpg of Astronaut.
file_path = r"geisel.jpg"
A = cv2.imread(file_path)

# Create the histogram for the image.
print("Generating the histogram of " + file_path + "...")
A_hist = compute_norm_rgb_histogram(A)

# Print the histogram via matplotlib.
figure_1 = plt.figure(1)
color_labels = np.concatenate((32 * ['r'], 32 * ['g'], 32 * ['b']))
plt.bar(range(0, len(A_hist)), A_hist, color=color_labels)
plt.title('RGB Histogram for Geisel.jpg')
plt.xlabel('Concatonated Bins with 32 Bins Per Color Layer')
plt.ylabel('Normalized Pixel Count per Color Layer')
plt.show()
```

*Output:*





#### Problem 4:

#### Parts i & ii:

#### *Code:*

```
# First, import the two images that have been provided.
file_path = r"dog.jpg"
Dog = cv2.imread(file_path)

file_path = r"travolta.jpg"
Travolta = cv2.imread(file_path)

# Now define a function that will take an RGB image with green background as an input
# and will return the foreground elements of the image.
def chroma_key(image, tag):
    # First, extract the green layer from the image.
    layer_blue = image[:, :, 0]
    layer_green = image[:, :, 1]
    layer_red = image[:, :, 2]

    # Also, extract the size of the given image.
    size = image.shape

    # Now, creating the layer mask is done by setting all pixels not equal to this
    # value to 1, and all other pixel values to 0.
    if tag == 'travolta':
        mask = np.where((layer_green > 120) & (layer_red < 110), 0, 1)
    else:
        mask = np.where((layer_green > 87) & (layer_red < 105) & (layer_blue < 240), 0, 1)

    # For each layer in the original image, apply the mask.
    foreground_blue = np.uint8(layer_blue * mask)
    foreground_green = np.uint8(layer_green * mask)
    foreground_red = np.uint8(layer_red * mask)

    # To complete the foreground, instantiate an empty foreground array and
    # substitute the layers for the foreground layers computed previously.
    foreground = np.zeros([size[0], size[1], 3])
    foreground[:, :, 0] = foreground_blue
    foreground[:, :, 1] = foreground_green
    foreground[:, :, 2] = foreground_red

    # Finally, return the processed image.
    return [np.uint8(foreground), np.uint8(255 * mask)]

# i & ii)

# Show the travolta.jpg mask and foreground elements.
[Travolta_foreground, Travolta_mask] = chroma_key(Travolta, 'travolta')
cv2.imshow('Travolta Foreground Mask', Travolta_mask)
cv2.imwrite('Travolta_Foreground_Mask.jpg', Travolta_mask)
cv2.waitKey(0)

cv2.imshow('Travolta Foreground Elements', Travolta_foreground)
cv2.imwrite('Travolta_Foreground_Elements.jpg', Travolta_foreground)
cv2.waitKey(0)

# Show the dog.jpg mask and foreground elements.
[Dog_foreground, Dog_mask] = chroma_key(Dog, 'dog')
cv2.imshow('Dog Foreground Mask', Dog_mask)
cv2.imwrite('Dog_Foreground_Mask.jpg', Dog_mask)
cv2.waitKey(0)

cv2.imshow('Dog Foreground Elements', Dog_foreground)
cv2.imwrite('Dog_Foreground_Elements.jpg', Dog_foreground)
cv2.waitKey(0)
```



*Output i:*

*Travolta Mask*

*Dog Mask*

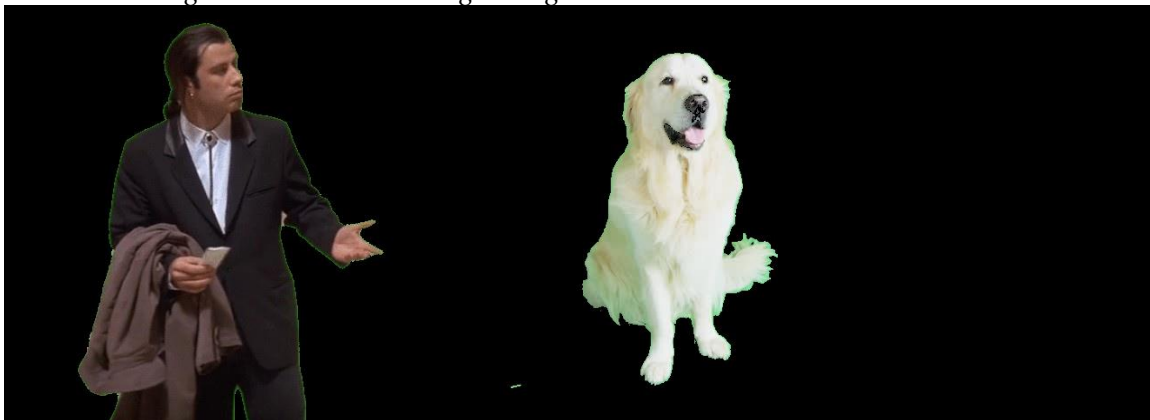


For this problem, it was simpler to find a suitable threshold range for the image of John Travolta since the range of green hues in the image was very small. Conversely, it was slightly more difficult to find a more suitable threshold range for the Dog image since the range of green hues differed quite broadly. Additionally, there appears to be a very small portion of the green screen that is not totally green in the image of the Dog, located on the floor. As a result, the thresholding that was adequate was not sufficient for removing this small remainder of the screen.

*Output ii:*

*Travolta Foreground*

*Dog Foreground*



### Part iii:

#### Code:

```
# Overlay the travolta and dog foreground images on a new image.
def overlay_image(foreground, background):
    # First, compute the size of the two images.
    back_size = background.shape
    fore_size = foreground.shape

    # Create a new image which will contain the background.
    new_image = background

    # Grab a section of the original image which will soon have the foreground elements
    # transposed on top of the section.
    overlay = background[back_size[0] - fore_size[0]: back_size[0], 0:fore_size[1], :]

    # On the section of the original image where the new foreground elements will be laid
    # on top of, store the pixel intensities of the foreground of the foreground elements
    # are not zero (i.e. where the green background used to be).
    overlay = np.where(foreground != 0, foreground, overlay)

    # Store the new section of the original image in the new image.
    new_image[back_size[0] - fore_size[0]:back_size[0], 0:fore_size[1], :] = overlay

    # Return the new image.
    return new_image

# iii)

# Load the image on which the foreground elements will be overlayed.
file_path = r"avengers.jpg"
Avengers = cv2.imread(file_path)

# For reference, show the original image.
cv2.imshow('Avengers', Avengers)
cv2.waitKey(0)

# For the travolta image.
travolta_is_an_avenger = overlay_image(Travolta_foreground, Avengers)
cv2.imshow('John Travolta Is An Avenger', travolta_is_an_avenger)
cv2.imwrite('John_Travolta_Is_An_Avenger.jpg', travolta_is_an_avenger)
cv2.waitKey(0)

# Python edits the original image after a computation has been done,
# so instantiate the original image again such that it is unmodified.
file_path = r"avengers.jpg"
Avengers = cv2.imread(file_path)

# For the dog image.
dog_is_an_avenger = overlay_image(Dog_foreground, Avengers)
cv2.imshow('Dog Is An Avenger', dog_is_an_avenger)
cv2.imwrite('Dog_Is_An_Avenger.jpg', dog_is_an_avenger)
cv2.waitKey(0)
```

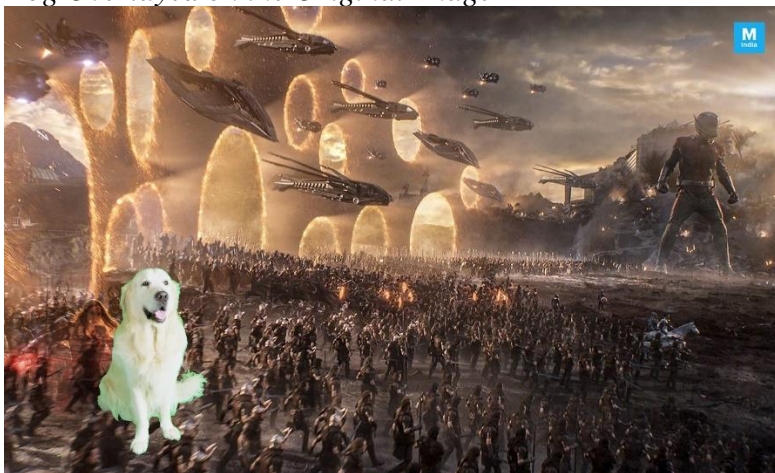
*Output:*  
*Original Image*



*Travolta Overlayed on the Original Image*



*Dog Overlayed on the Original Image*



## Problem 5:

### Part i:

One method of interpolation is "nearest-neighbor interpolation". This method of interpolation increases the size of an image by producing pixels with intensity values equal to their closest neighboring pixel. This is the simplest method of interpolation but is the least accurate of the three methods of interpolation presented.

A second method of interpolation is "bilinear interpolation". This method of interpolation increases the size of an image by generating pixels that are an average of the intensities of the four closest pixels. This method of interpolation is more accurate in terms of image quality than nearest-neighbor interpolation.

A third method of interpolation is "bicubic interpolation". This method of interpolation increases the size of an image by generating pixels that are an average of the intensities of the 16 nearest neighboring pixels. This is the most accurate of the three methods of interpolation due to the number of pixels that are being considered in generating a new pixel.

### Part ii:

#### *Code:*

```
# Function which accepts an image and a scaling factor and computes the nearest-neighbor, bilinear,
# and bicubic interpolations of the the given image.
def perform_resize(image, factor):
    # Grab the size of the given image.
    size = image.shape

    # Using the factor provided, compute the necessary dimensions of the new image set.
    new_size = [round(size[0] * factor), round(size[1] * factor)]

    # For each interpolation method, use openCV to compute the resized images using interpolation.
    nearest = cv2.resize(image, [new_size[1], new_size[0]], interpolation=cv2.INTER_NEAREST)
    linear = cv2.resize(image, [new_size[1], new_size[0]], interpolation=cv2.INTER_LINEAR)
    cubic = cv2.resize(image, [new_size[1], new_size[0]], interpolation=cv2.INTER_CUBIC)

    # Once the image set has been computed, store all of the images into an album of images.
    image_set = np.zeros([new_size[0], new_size[1], 11])
    image_set[:, :, 0:3] = nearest
    image_set[:, :, 4:7] = linear
    image_set[:, :, 8:11] = cubic

    # Return the image set.
    return np.uint8(image_set)

# Function to display the image set for an interpolated image set.
def show_image_set(image_set, factor, name):
    # Determine the shape of the given image set.
    size = image_set.shape

    # Attribute appropriate titles and file names for the images.
    title = "Nearest Factor of " + str(factor) + " " + name
    file = "Nearest_Factor_of_" + str(factor) + "_" + name + ".jpg"

    # Show and save a cropped version the nearest-neighbor image.
    image = image_set[:, :, 0:3]
    image_cropped = image[round(size[0] * 4/9):round(size[0] * 7/9), round(size[1] * 4/9):round(size[1] * 7/9)]
    cv2.imshow(title, image_cropped)
    cv2.imwrite(file, image_cropped)
    cv2.waitKey(0)
```



```

# Show and save a cropped version the bilinear image.
title = "Linear Factor of " + str(factor) + " " + name
file = "Linear_Factor_of_" + str(factor) + "_" + name + ".jpg"
image = image_set[:, :, 4:7]
image_cropped = image[round(size[0] * 4/9):round(size[0] * 7/9), round(size[1] * 4/9):round(size[1] * 7/9)]
cv2.imshow(title, image_cropped)
cv2.imwrite(file, image_cropped)
cv2.waitKey(0)

# Show and save a cropped version the bicubic image.
title = "Cubic Factor of " + str(factor) + " " + name
file = "Cubic_Factor_of_" + str(factor) + "_" + name + ".jpg"
image = image_set[:, :, 8:11]
image_cropped = image[round(size[0] * 4/9):round(size[0] * 7/9), round(size[1] * 4/9):round(size[1] * 7/9)]
cv2.imshow(title, image_cropped)
cv2.imwrite(file, image_cropped)
cv2.waitKey(0)

# For ease of viewing, close all of the open windows.
cv2.destroyAllWindows()

# ii)

# Create an array of image file paths to lookup during the loop.
file_paths = [r"gondor.jpg", r"rohan.jpg", r"tolkien.jpg"]
names = ["Gondor", "Rohan", "Mordor"]

# Loop through the file paths in the array to simplify the amount of code to write.
# For each file path, display the downsampled version of the image.
for i in range(0, 3):
    A = cv2.imread(file_paths[i])
    cv2.imshow('Original Image', A)
    cv2.waitKey(0)

    downsample_1 = perform_resize(A, 0.3)
    show_image_set(downsample_1, 0.3, names[i])

    downsample_2 = perform_resize(A, 0.5)
    show_image_set(downsample_2, 0.5, names[i])

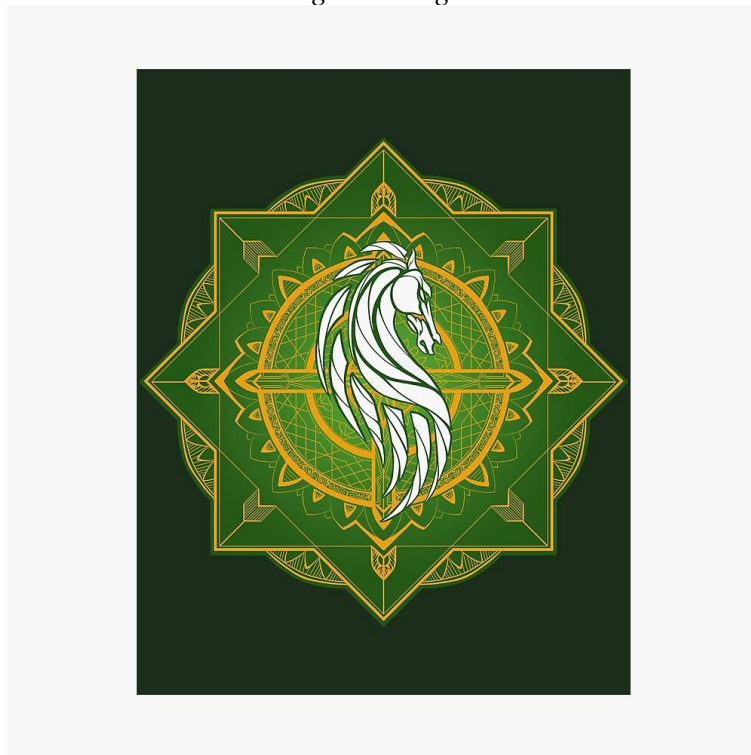
    downsample_3 = perform_resize(A, 0.7)
    show_image_set(downsample_3, 0.7, names[i])

```

*Original Image 1:*

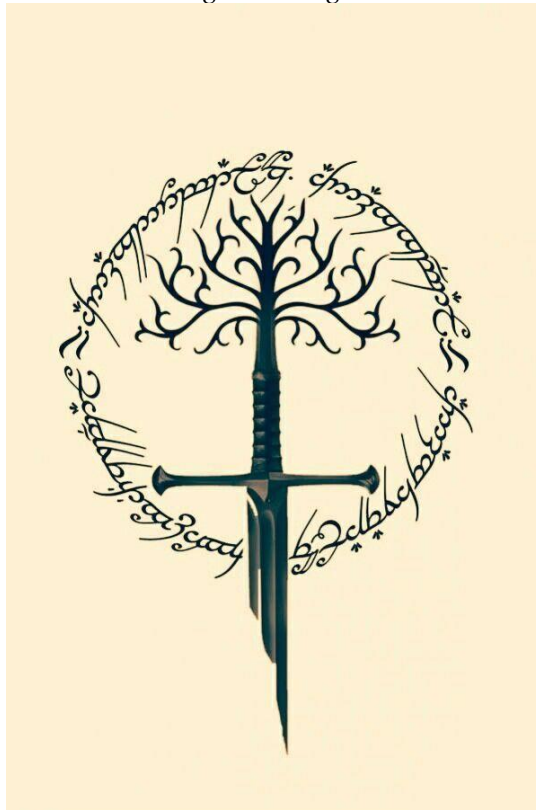


*Original Image 2:*






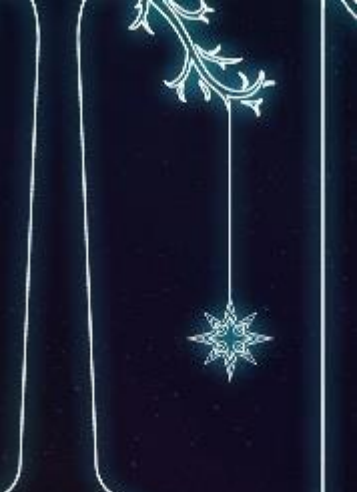







*Original Image 3:*











*Image 1 Down-sampling*

Size	Nearest-Neighbor	Bilinear	Bicubic
0.3			
0.5			
0.7			

When examining the effect of the interpolation methods for downsizing, it is clear that the bicubic interpolation method is superior to the others in terms of quality. However, this only seems to hold for certain scaling factors. For example, the difference in quality between bilinear and bicubic when subject to a scaling factor of 0.3 is almost non-differentiable. The difference in quality between bilinear and bicubic is a little more apparent when the scaling factor is closer to 1.0. The two images look almost exactly alike. However, the quality is still superior to the nearest-neighbor interpolation method regardless of scaling factor.

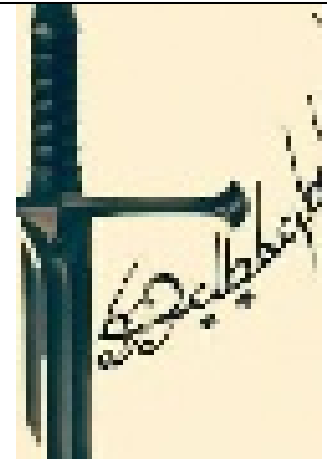
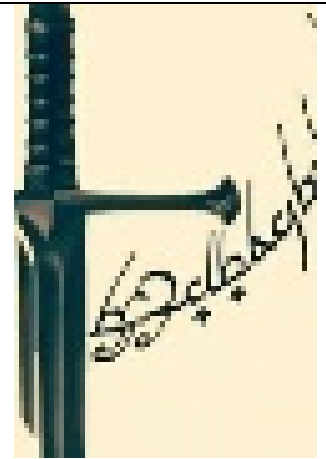




*Image 2 Down-sampling*

Size	Nearest-Neighbor	Bilinear	Bicubic
0.3			
0.5			
0.7			

The effects of the interpolation techniques on this image are comparable to image 1. However, some of the finer details are more prominent in the bicubic interpolation images. Once

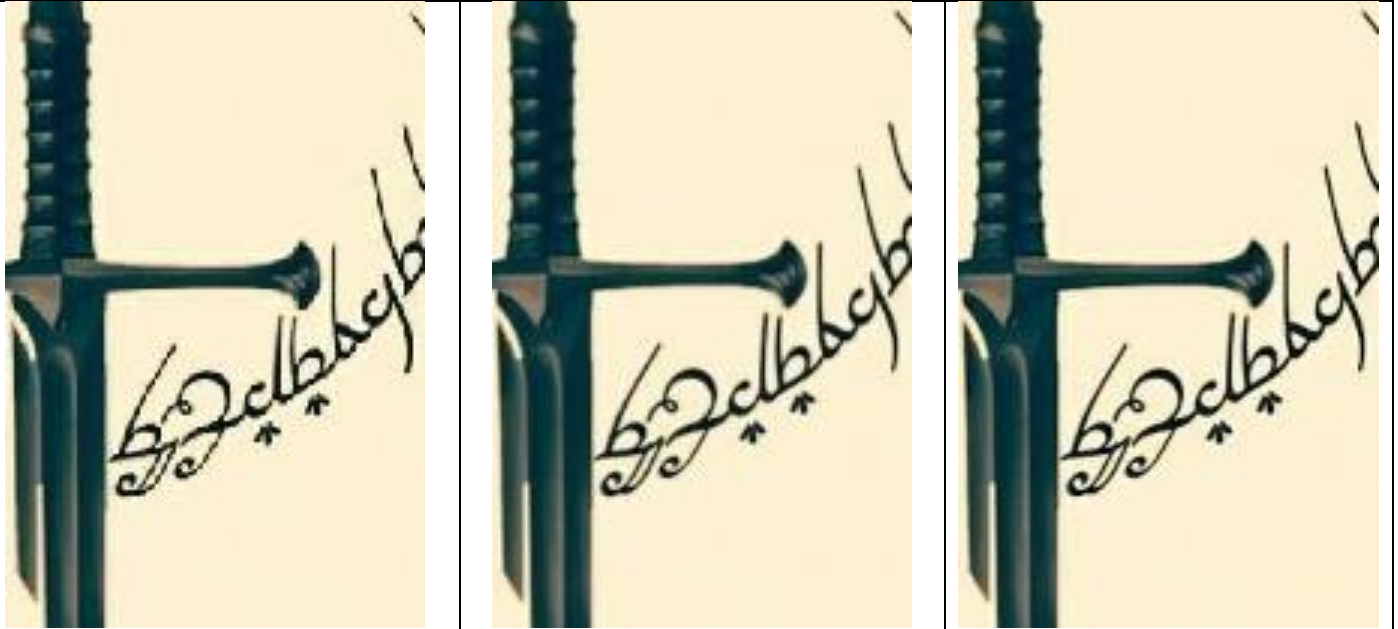
again, it can be concluded that the bicubic interpolation method is superior to the nearest-neighbor and the bilinear methods.

*Image 3 Down-sampling*

Size	Nearest-Neighbor	Bilinear	Bicubic
0.3			
0.5			



0.7



Surprisingly, there is little difference in quality between several of the images above, except for a couple of the nearest-neighbor interpolated images. It could be argued that the bilinear and bicubic images look almost exactly alike. Thus, it is difficult to reach a conclusion about which interpolation technique worked better. It can be concluded, however, that the nearest-neighbor method is the worst of the three. When scaled by a factor of 0.3, the text of the nearest-neighbor image is hardly legible (despite the fact that it is a fictional language).

### Part iii:

Code:

```
# iii)




# Loop through the file paths in the array to simplify the amount of code to write.
# For each file path, display the upsampled version of the image.
for i in range(0, 3):
    A = cv2.imread(file_paths[i])

    upsample_1 = perform_resize(A, 1.5)
    show_image_set(upsample_1, 1.5, names[i])




    upsample_2 = perform_resize(A, 1.7)
    show_image_set(upsample_2, 1.7, names[i])


    upsample_3 = perform_resize(A, 2)
    show_image_set(upsample_3, 2, names[i])
```

*Image 1 Up-sampling*

Size	Nearest Neighbor	Bilinear	Bicubic
1.5			

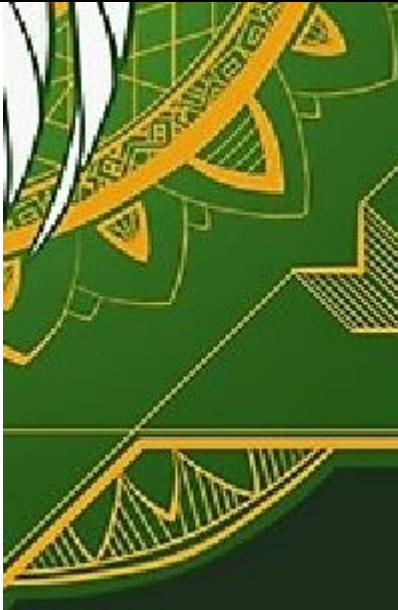







Size	Nearest Neighbor	Bilinear	Bicubic
1.7			



Size	Nearest Neighbor	Bilinear	Bicubic
2.0			

For this image, it can be concluded that bicubic and bilinear share very similar qualities, except that the bicubic images do a better job of displaying the finer qualities present in the original image. Similar to down-sampling, bicubic seems to be the strongest method of interpolation, and nearest-neighbor interpolation appears to be the weakest.

Image 2 Up-sampling

Size	Nearest Neighbor	Bilinear	Bicubic
1.5			
1.7			



Size	Nearest Neighbor	Bilinear	Bicubic
2.0			

For this image, the distinction in quality between the three methods is much more pronounced. The bicubic interpolation method preserves the finer details of the image which also preserving the image's original color. For example, in the scale factor 2.0 row, the lines on the bilinear image are darker than the original, but the fine detail is somewhat preserved. Once again, the strongest interpolation method is bicubic, and the weakest is nearest-neighbor.

*Image 3 Up-sampling*

Size	Nearest Neighbor	Bilinear	Bicubic
1.5			
1.7			

Size	Nearest Neighbor	Bilinear	Bicubic
2.0			

For image 3, the effects of the different interpolation techniques are also very apparent. The contrast in quality between the nearest-neighbor interpolation and the other two methods is starkly different. The nearest-neighbor images look very pixelated, without much smoothing at all. The bilinear and bicubic images look much smoother, but it could be argued that the methods had the same quality on the image. It is possible that the text in the bicubic images is slightly darker compared to the bilinear, but the difference is too minimal to tell. Overall, the bicubic interpolation method produces the highest quality images, and the nearest-neighbor interpolation method produces the lowest quality images.



## **Part iv:**

### *Code:*

```
# iv)

# Generate the downsampled image sets for each interpolation method.
image_set_gondor = perform_resize(cv2.imread(r"gondor.jpg"), 0.1)
image_set_rohan = perform_resize(cv2.imread(r"rohan.jpg"), 0.1)
image_set_tolkien = perform_resize(cv2.imread(r"tolkien.jpg"), 0.1)

# Method set for identifying which image was downsampled with a particular interpolation
# method.
method_set = ['nearest', 'linear', 'cubic']

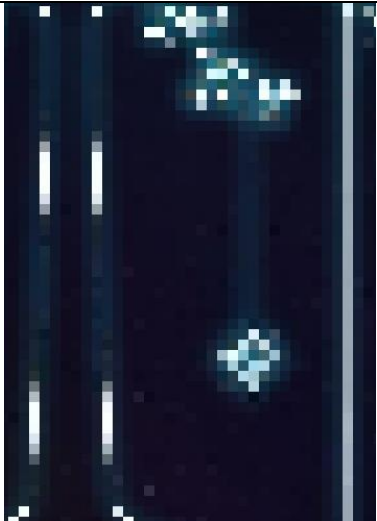



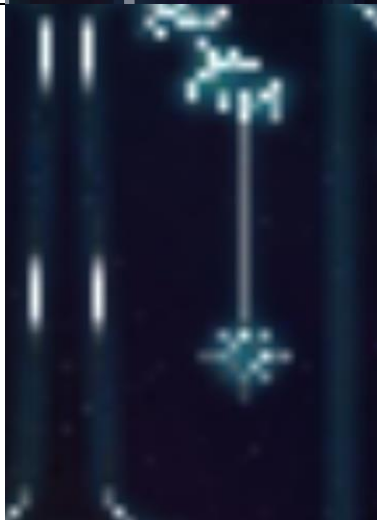
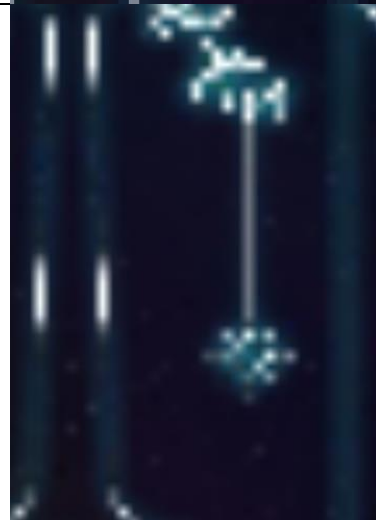



# Upsample the Gondor image set.
for i in range(0, 3):
    current_image = image_set_gondor[:, :, (4*i):(4*i + 3)]
    upsample_gondor = perform_resize(current_image, 10)
    show_image_set(upsample_gondor, 10, ("Gondor_" + method_set[i]))

# Upsample the Rohan image set.
for i in range(0, 3):
    current_image = image_set_rohan[:, :, (4*i):(4*i + 3)]
    upsample_rohan = perform_resize(current_image, 10)
    show_image_set(upsample_rohan, 10, ("Rohan_" + method_set[i]))

# Upsample the Tolkien image set.
for i in range(0, 3):
    current_image = image_set_tolkien[:, :, (4*i):(4*i + 3)]
    upsample_tolkien = perform_resize(current_image, 10)
    show_image_set(upsample_tolkien, 10, ("Tolkien_" + method_set[i]))
```

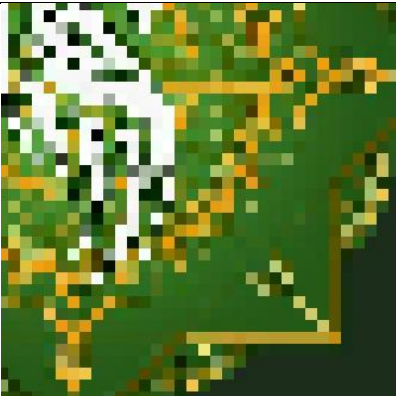

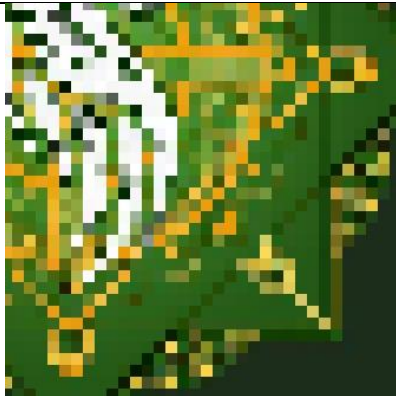


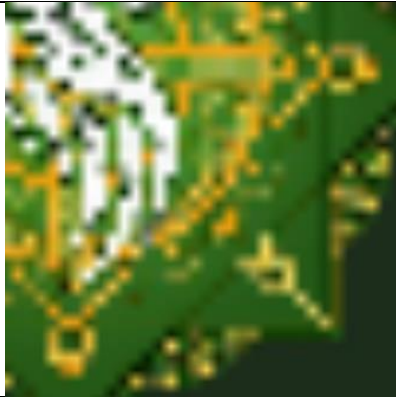



Output:

Image 1 Sampling Method Combinations

Up-Sampling	Down-sampling		
	Nearest-Neighbor	Bilinear	Bicubic
Nearest-Neighbor			
Bilinear			
Bicubic			

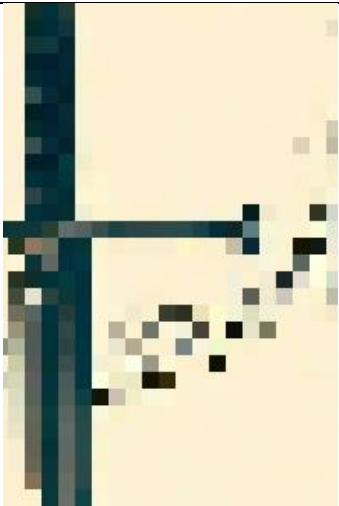








For image 1, it is difficult to tell which combination of down-sampling and up-sampling has the highest quality. It is clearly not the images in which down-sampling has been performed with nearest-neighbor interpolation. The resulting images are very blocky, with very little detail preserved from the original. The highest quality image could be those in which down-sampling was done with **bilinear** interpolation and up-sampling done with **bicubic** interpolation. While the details are still not highly preserved, you can tell when compared to the original image what was supposed to be displayed. I would have expected the combination bicubic-bicubic to preserve the quality of the original image, but this does not appear to be the case for this image.

Image 2 Sampling Method Combinations:

Up-Sampling	Down-sampling		
	Nearest-Neighbor	Bilinear	Bicubic
Nearest-Neighbor			
Bilinear			
Bicubic			

These results are not at all surprising. The details contained in the original image have mostly disappeared from the resampled images. What is surprising is that perhaps the highest quality combination of interpolation methods is the image in which both down-sampling and up-sampling were done with **bilinear** interpolation. I believe this to be case since most of the original colors are preserved in the image. While the detail does not match the original, it is clear what the primary colors used to be in the image as well as what the primary geometric structures were.

*Image 3 Sampling Method Combinations*

Up-Sampling	Down-sampling		
	Nearest-Neighbor	Bilinear	Bicubic
Nearest-Neighbor			
Bilinear			
Bicubic			

Once again, it is difficult to determine which combination of sampling techniques has produced the highest quality image. None of the text which appears in the original image is recognizable in any of the sampling combinations. The colors on the sword also vary quite a bit. The highest quality combination of techniques is perhaps the image in which down-sampling was done with **bilinear** interpolation and up-sampling was done with **bicubic** interpolation. I believe that this image best preserves the original colors of the sword, and the text, while still illegible, is very clearly wrapping around the sword in a quarter-circle.