Report:

Project Motivation:

Image processing methods are used in many different industries and various engineering disciplines to extract information, through analysis. Examples of different disciplines that use image processing are Civil Engineers, Environmental engineers, and Mechanical Engineers.

From an article by Ashwini A. Salunkhe published in 2022, It outlines the progress and many applications of image processing in Civil engineering. It talks about the application to automatically assess structural damage to buildings through a camera and machine learning. The future of image processing in civil engineering will continue to develop and improve through the implementation of more AI and machine learning. In the many different areas image analysis can be used will "save time and cost in some tedious jobs like material characterization in CDW, measuring strain, strength, and displacements."[1].

[1]Progress and Trends in Image Processing Applications in Civil Engineering: Opportunities and Challenges (hindawi.com)

As for Environmental Engineers, they are using historical aerial photos to analyze humanity's impact on different spaces and environments. More specifically "human industrial behavior"[2] and its effect on "sites through time"[2]. Allowing the comparison of images from different time periods, as they are affected by deforestation, urban sprawl, pollution, or habitat degradation, enabling them to devise sustainable strategies for land restoration and conservation. As technology advances and the environmental changes environmental engineers will find innovative ways to use image processing, from Environmental Forensics to simple enlarging minute details.

[2]Photogrammetry, Photointerpretation, and Digital Imaging and Mapping in Environmental Forensics - ScienceDirect

In a article authored by Marta Roberson, *How Image Recognition Is Transforming the Automobile Industry*, the advance of Image recognition is discussed and the implementations that it will make our cars safer, more efficient, and reliable. Mechanical Engineers will continue to implement image processing in the expanding Self driving industry.

[3]How Image Recognition Is Transforming the Automobile Industry - InnovationManagement

Project Overview and Methods:

In this section, we provide an overview of the core methods and techniques employed in our project for image processing. The primary focus of our project is on image blending using image pyramids. We will begin by outlining the essential concepts of image pyramids and how they serve as a fundamental tool in various image processing applications.

Image Pyramids

Image pyramids are a hierarchical multi-scale representation of an image. They play a vital role in computer vision, image processing, and graphics, enabling applications such as image blending, image registration, and object recognition. The pyramid structure consists of a sequence of images, each at a different scale, allowing for efficient and accurate processing.

Types of Image Pyramids

In our project, we utilize Gaussian and Laplacian pyramids:

1. **Gaussian Pyramid**: This pyramid is constructed by applying Gaussian smoothing to the input image, followed by subsampling. It helps in down sampling the image, reducing computational load while preserving essential image features.
2. **Laplacian Pyramid**: The Laplacian pyramid is derived from the Gaussian pyramid and is primarily used for image blending. It represents the difference between two consecutive levels of the Gaussian pyramid, capturing fine details in the image.

Image Blending Using Pyramids

Image blending is the process of combining two or more images into a single image seamlessly. Image pyramids offer a robust approach to achieve blending. The blending process involves the following steps:

3. **Source and Target Images**: We begin with two input images, a source image, and a target image. The source image typically contains the object or region to be blended into the target image.
4. **Gaussian Pyramids**: Gaussian pyramids are constructed for both the source and target images. These pyramids create multiple levels of images with varying resolutions.
5. **Masking**: We apply masks to both the source and target images. These masks define the regions of interest in each image. For example, in our project, the source image may have a mask to specify the region to be blended.
6. **Laplacian Pyramids**: Laplacian pyramids are generated for both the source and target images by taking the difference between consecutive levels of the Gaussian pyramid.
7. **Blending**: The Laplacian pyramids of the source and target images are combined, considering the respective masks, to create a blended pyramid.
8. **Reconstruction**: The blended pyramid is reconstructed to obtain the final blended image.

Our project focuses on the core aspects of these steps and demonstrates how they are implemented in Python. These techniques are valuable for various engineering applications, including image stitching, image compositing, and object removal.

In the following sections, we will delve into the design of our Python program, its functionality, and discuss the implementation of the above methods in detail:

9. **get_click_coordinates(image)**: This function allows you to interactively click on an image using a graphical interface to obtain the x and y coordinates of a mouse click. It returns the coordinates of the clicked point on the image.
10. **rgb_to_grayscale(image)**: This function converts a color image into a grayscale image. It uses the formula for luminance to calculate the grayscale values from the RGB channels.
11. **gaussian(gray)**: This function applies a Gaussian smoothing filter to a grayscale image. It converts the image with a 5x5 Gaussian kernel to reduce noise and create a smoothed version of the input image.
12. **upsample(subs image)**: This function upsamples (increases the resolution of) a given image by a factor of 2. It's used in image pyramid construction.
13. **subsample(image)**: This function down samples an image by a factor of 2, reducing its resolution. It is used in image pyramid construction, especially in creating lower-resolution levels.
14. **mask_source(image, x1, y1, x2, y2)**: This function creates a binary mask where a specified rectangular region is set to 0 (False), and the rest of the image is set to 1 (True). This mask is used to specify the region of interest in the source image.
15. **mask_target(image, x1, y1, x2, y2, horiz, vert)**: Like **mask_source**, this function creates a binary mask, but for the target image. It defines a rectangular region based on specified coordinates.
16. **mask_multiply(image, mask)**: This function multiplies each pixel of an image by the corresponding value in a binary mask. It effectively masks out (sets to zero) pixels outside the region of interest defined by the mask.
17. **gaussian_pyramid(image, pyramid_levels)**: This function generates a Gaussian image pyramid with the specified number of levels. It samples the image down and creates a series of blurred and lower-resolution images.
18. **laplacian_pyramid(gaussian_pyramid)**: This function takes a Gaussian image pyramid and computes the Laplacian pyramid. It represents the difference between consecutive levels of the Gaussian pyramid, capturing delicate details in the image.
19. **extract_face(image)**: This function extracts a face from a given image by finding the non-zero (non-black) region and cropping the image to this region's bounding box.

20. **paste_face(source_data, face, source_coords)**: This function pastes the extracted face back into a larger image at specified coordinates, effectively compositing the face into the image.

Discussion of Algorithm Design:

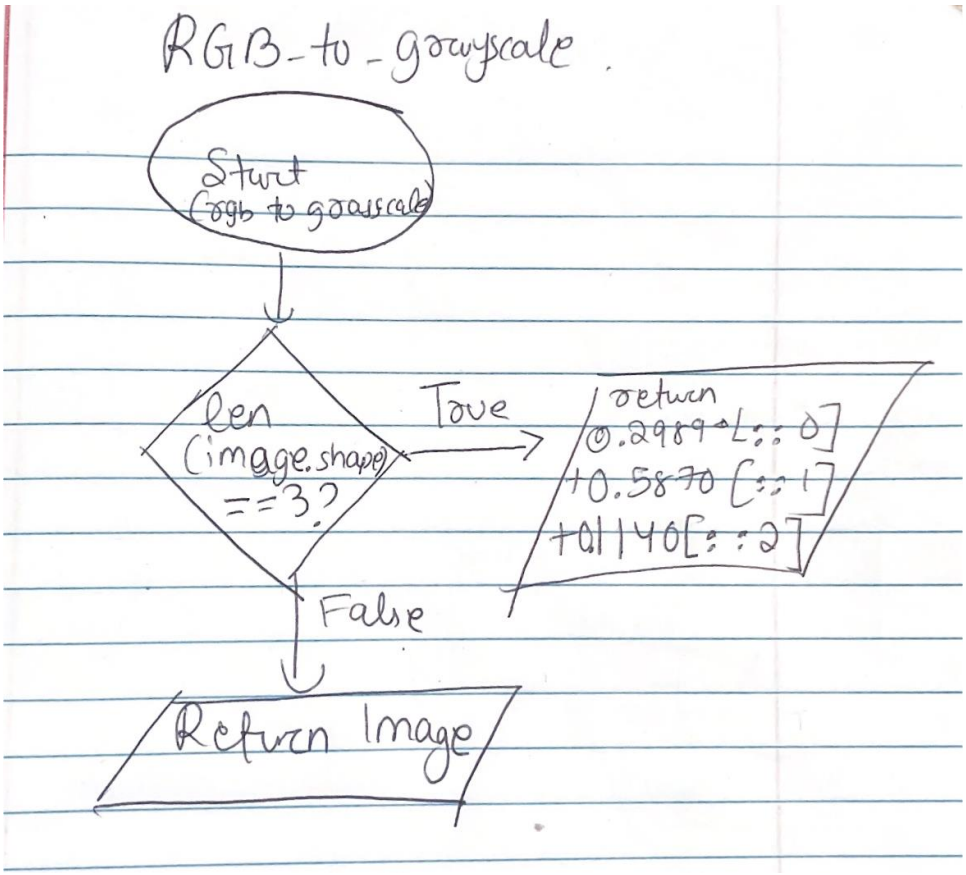
Each of these are described below in more detail.

As you read through each item consider the following: Which elements should go in the main program? Rather than having a main program. We had multiple functions so that we can check the error handling and edge handling in each of the functions.

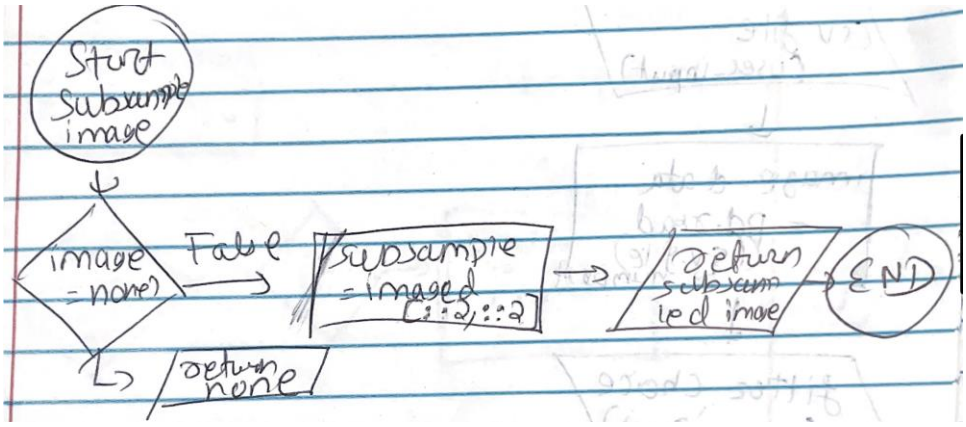
Which should be user-defined functions?

Below are the user defined functions and the flowchart related to them. These should be the user defined functions because it is how we segment each method and step in the processes.

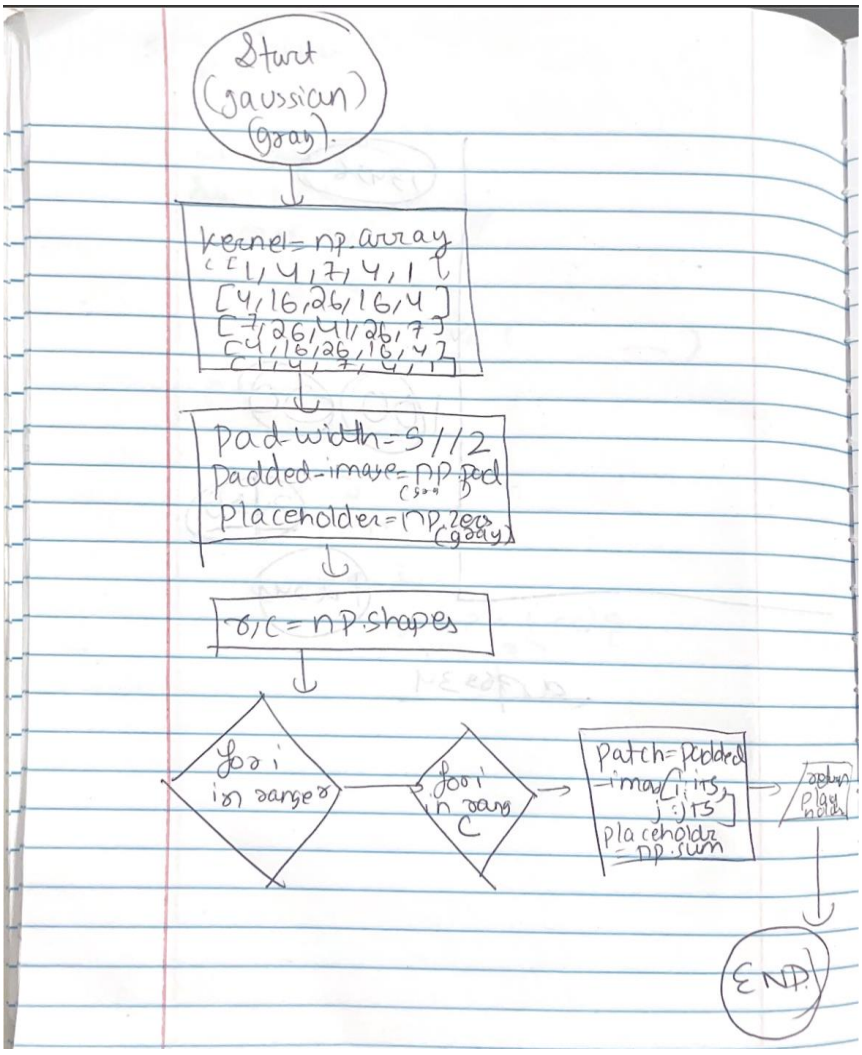
1: RGB_to_grayscale



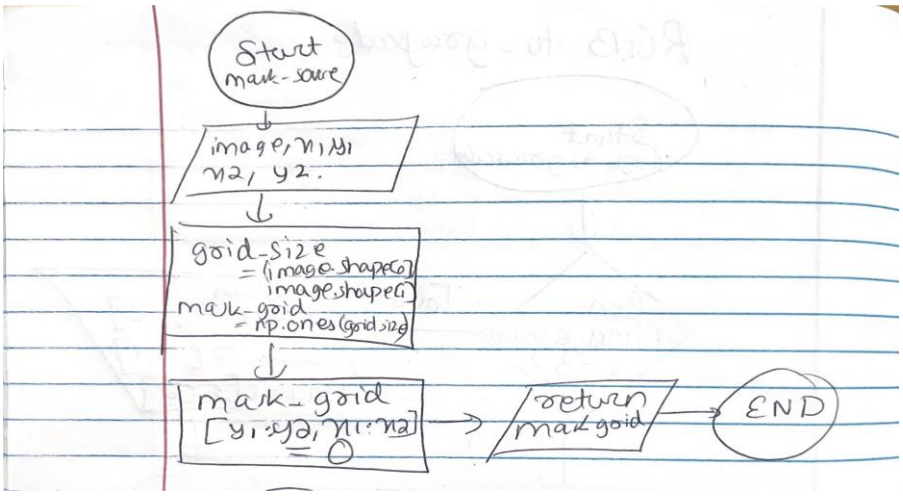
2: Subsampling



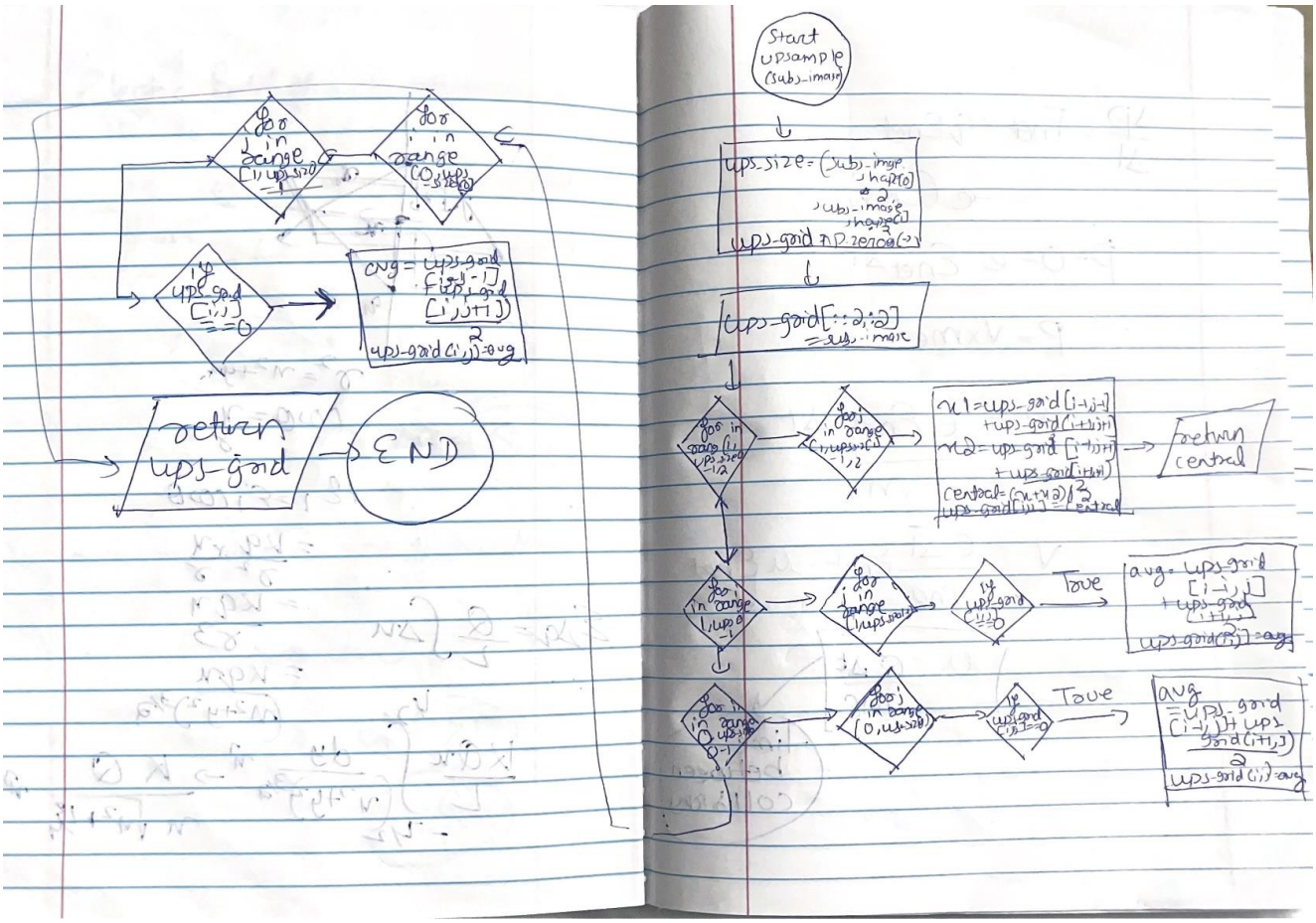
3: Gaussian filter flowchart



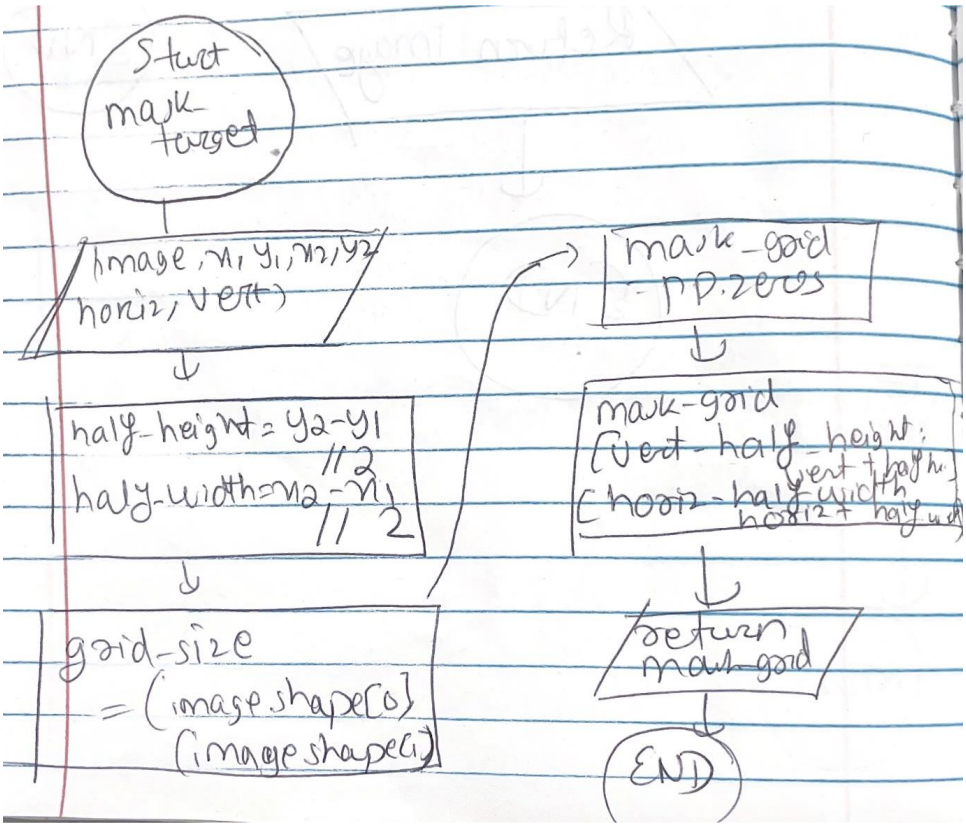
4: Mask_source



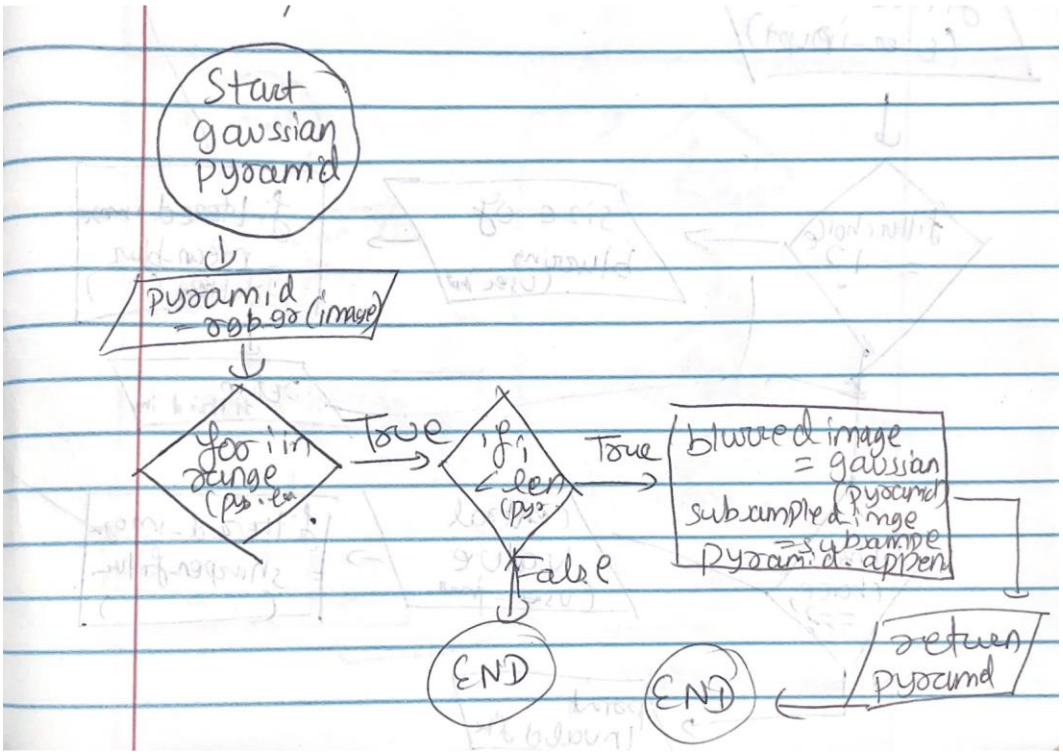
5: Upsampling



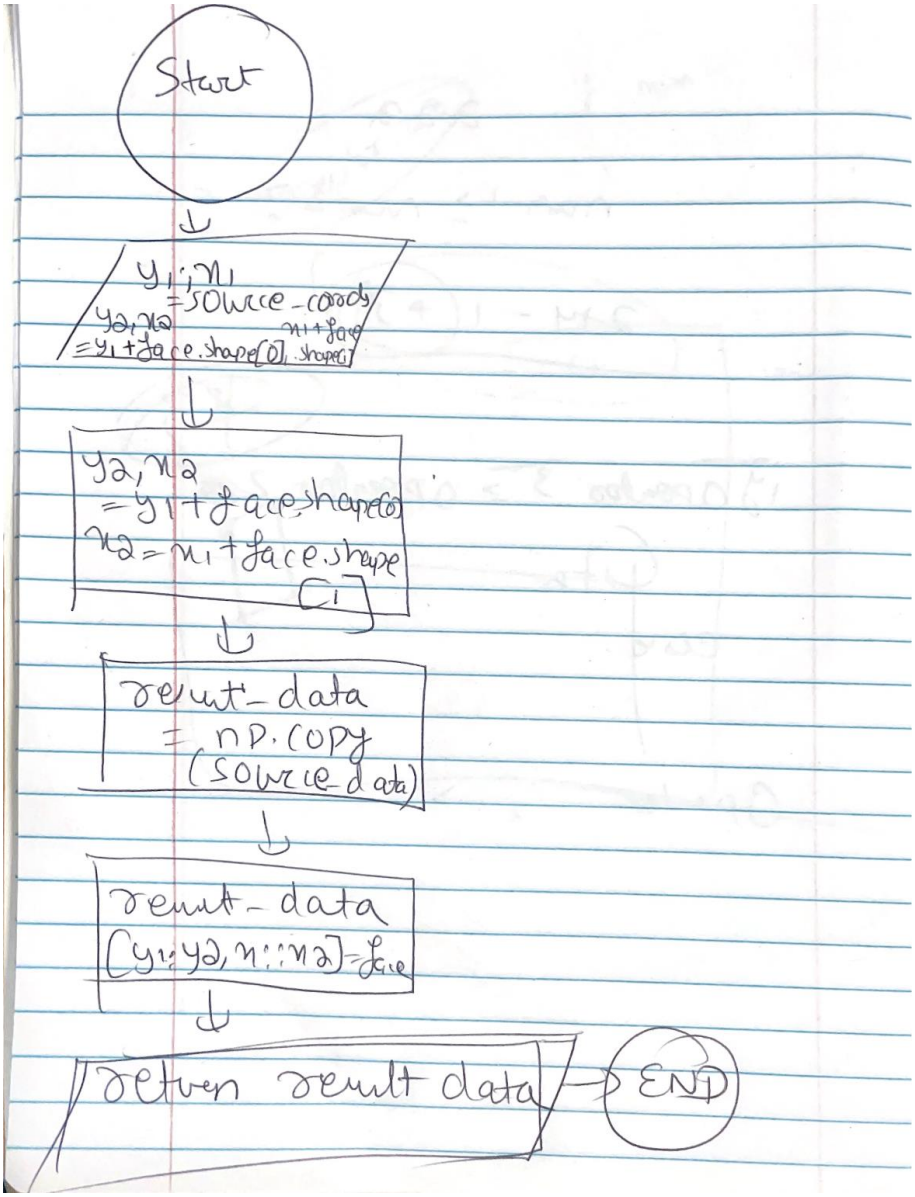
6: Mask_Target



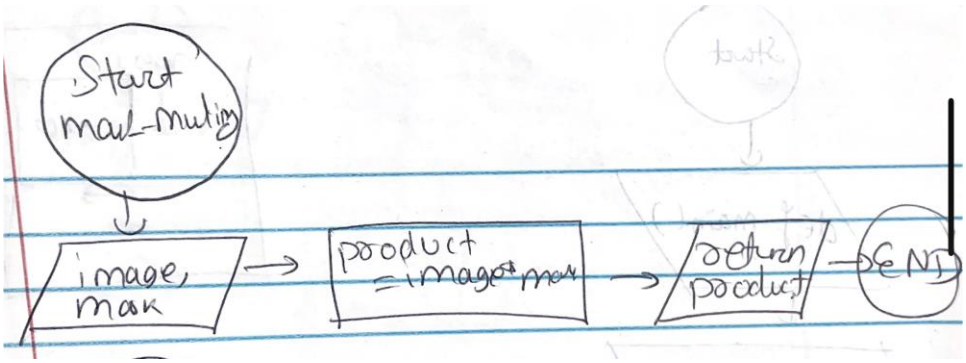
6: Gaussian pyramid



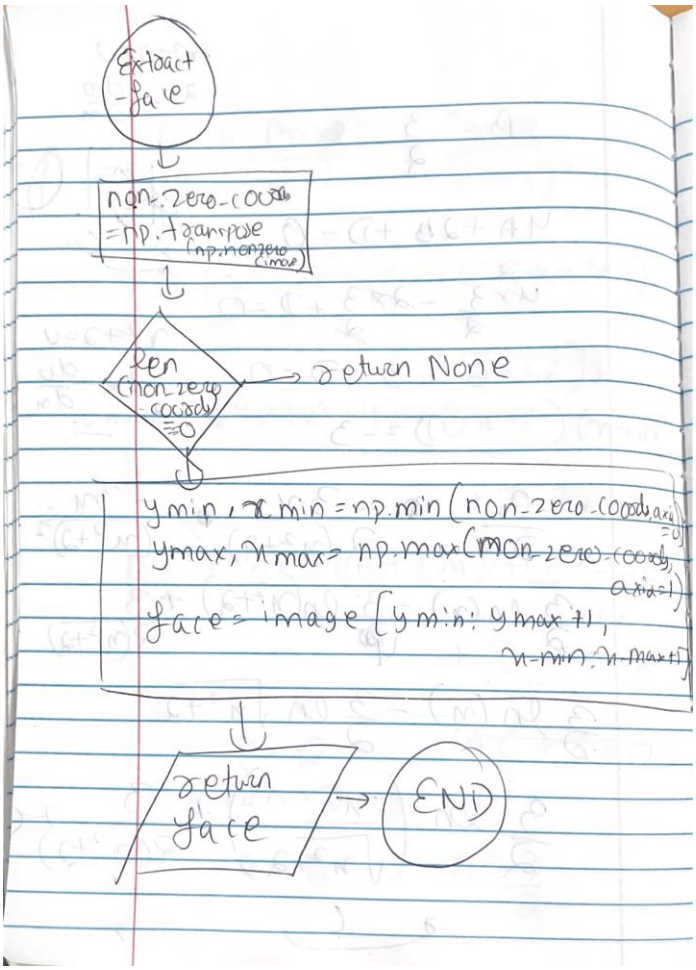
7: Laplacian Pyramid



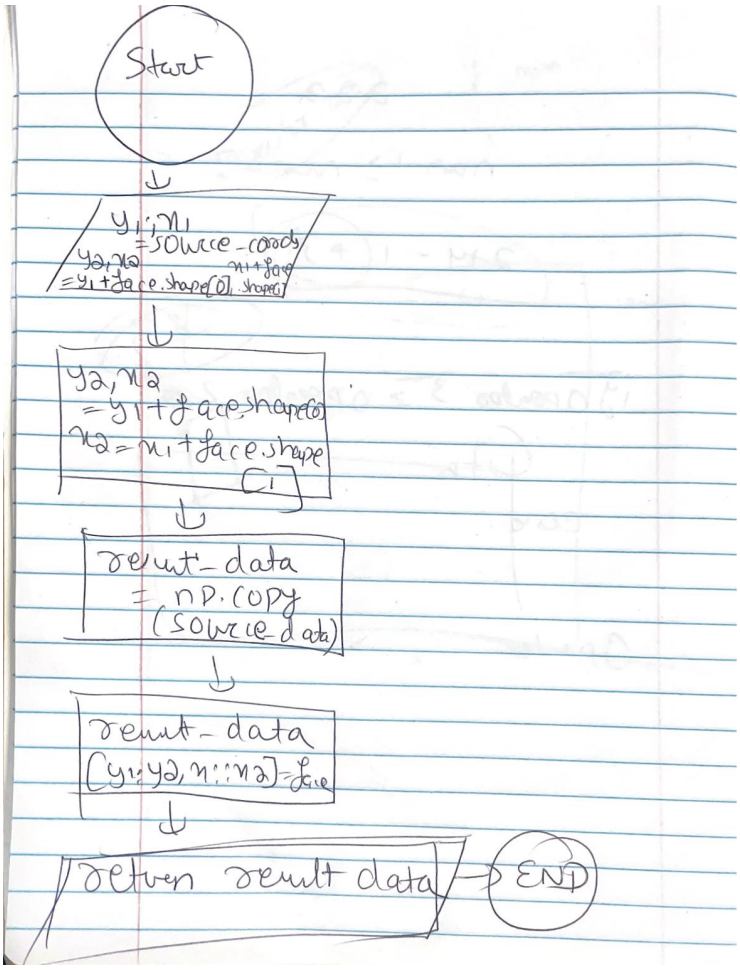
8: Mask_Multiply



9: Extract Face



10: Paste face



Where might loops be useful?

Loops are useful in programming for repetitive tasks or for the collection of data. In the code, loops are used in several places to perform tasks that need to be repeated many times. For example, every filter needs to be applied across the array and through the data. More specifically our code uses loops in the following functions.

Nested Loops for Image Processing:

In the gaussian function, there are nested loops used to iterate through the rows and columns of an image to apply a Gaussian filter to each pixel. These loops process the entire image pixel by pixel to perform convolution.

Iterating Through Pyramid Levels:

In the gaussian_pyramid and laplacian_pyramid functions, loops are used to iterate through different levels of an image pyramid. These loops construct the pyramid by applying operations at each level.

Iterating Through Masked Images in Pyramids:

In the code that processes masked images for source and target, loops are used to iterate through the pyramid levels and apply masks to each level of the pyramid.

Iterating Through Image Coordinates for Blending:

In the add_images function, loops are used to iterate through the source image's coordinates and blend it with the target image. These loops ensure that the source image is correctly positioned within the target image.

Displaying Images in Pyramids:

When displaying images at different pyramid levels, a loop is used to iterate through the pyramid and display each level of the pyramid. This is done for both the source and target images.

Iterating Through the Pixels of a Blended Image:

After blending the source and target images, a loop is used to iterate through the resulting image's pixels and display the combined image for each pyramid level.

In summary, loops are useful in various parts of the code to process images, construct image pyramids, apply masks, blend images, and display the results. They enable repetitive operations and help automate image processing tasks.

Which parts does your team need to program and which parts do NumPy or matplotlib have built-in functions for?

Our group has to code the following tasks:

User Interaction and Input:

Collecting user input for source and target image filenames, pyramid levels, and coordinates is something you would program.

Custom Image Processing Functions:

Your team has written custom functions for operations such as applying Gaussian filters, subsampling, upsampling, creating masks, and blending images at different pyramid levels.

Control Flow and Logic:

The high-level logic of the program, including decisions, loops, and the organization of operations, is your responsibility.

Iterating Through Pyramid Levels and Displaying Images:

Loops are used to iterate through pyramid levels, and you program the code to display images at various levels of the pyramid.

On the other hand, NumPy and Matplotlib have built-in functions for the following:

Image Loading and Display:

NumPy and Matplotlib functions are used for loading and displaying images. For example, plt.imread() is used to load images, and plt.imshow() is used for image display.

Basic Image Operations:

NumPy is used for basic image operations, such as converting RGB images to grayscale and padding images.

Gaussian Filter:

NumPy is used for matrix operations to apply a Gaussian filter to grayscale images.

Subsampling and Upsampling:

NumPy is utilized to efficiently subsample and upsample images by manipulating the image data directly.

Padding Images:

NumPy is used to pad images with zeros, allowing for proper convolution and image processing.

Array Manipulations:

NumPy is employed for various array manipulations and mathematical operations on image data.

References:

[1] Salunkhe, Ashwini A. "Progress and Trends in Image Processing Applications in Civil Engineering: Opportunities and Challenges." *Hindwawi*, 5 May 2022, www.hindawi.com/journals/ace/2022/6400254/. Accessed 18 Oct. 2023.

[2] Ebert, James I. "Photogrammetry, Photointerpretation, and Digital Imaging and Mapping in Environmental Forensics." *Science Direct*, 2022, www.sciencedirect.com/science/article/pii/B9780124046962000035. Accessed 18 Oct. 2023.

[3] Robertson, Marta. "How Image Recognition Is Transforming the Automobile Industry." *Innovation Management*, 04 Oct 2019, innovationmanagement.se/2019/10/04/how-image-recognition-is-transforming-the-automobile-industry/. Accessed 18 Oct. 2023.

Appendices:

a. User manual

User Manual for Image Blending Code

Table of Contents

1. Introduction

This code is designed to blend two images seamlessly using image pyramids. Use the manual to guide you through the installation, and usage of the code.

2. Installation and Requirements

Before using the code, check you have installed the following:

Python

NumPy

Matplotlib

Make sure you have the required Python packages installed (NumPy and Matplotlib) and two images you want to blend.

3. Downloading and Running the Code

Download the code.

Open a terminal or command prompt.

4. Using the Image Blending Code

4.1. Input Image Selection

When you run the code, it will prompt you to enter the file of your Target, then define region and enter Source image, and then you will be prompted to select region (be sure to use a jpg). Make sure to provide valid image file paths.

4.2. Defining the Source and Target Regions

After selecting the source and target images, you will be prompted to define the regions in these images. Use the interactive interface to click on the desired regions for blending. On the

Target select the top left corner of the desired area and the bottom right corner of the desired area. On the Source select the middle of the desired area.

4.3. Setting Pyramid Levels

Specify the number of pyramid levels you want to use for the blending process. A higher number of levels may capture more details but require more time.

4.4. Running the Blending Process

The code will run the image blending process and display the final blended image. Be sure to close the image window to proceed or run the function again.

5. Understanding the Functions

This section explains the functions used in the code. It's essential to understand these functions for a deeper comprehension of the code's functionality.

It has been explained in detail under the methods at Project Overview.

6. Troubleshooting and Tips

Ensure that you have the required dependencies installed.

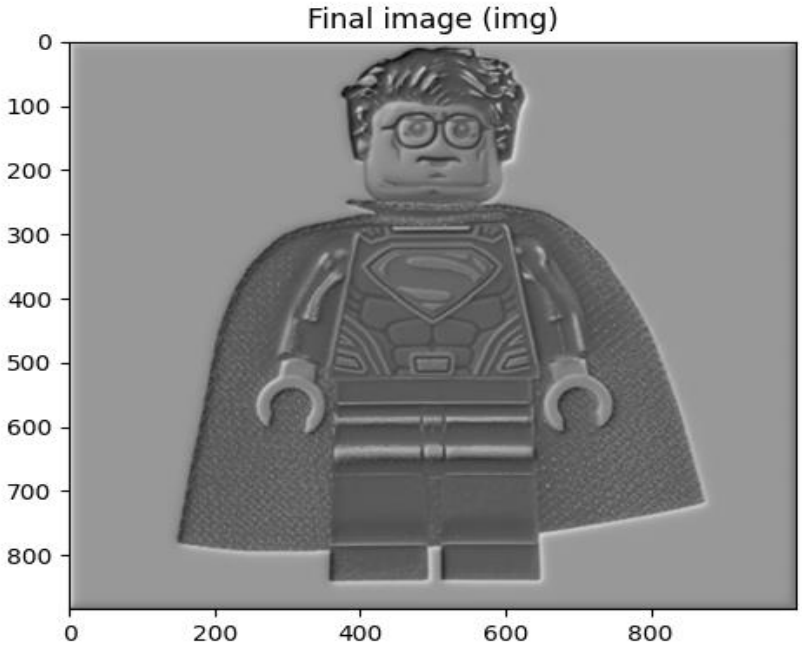
Verify that the image file paths provided are correct.

If you encounter issues, refer to error messages and the troubleshooting section in the code.

7. Conclusion

This user manual should help you use the Image Blending Code effectively. Some photos examples have been posted in the appendix for further clarity.

The following images are an example, here the target image is a Superman image, and the source image is a Clark Kent image. Using a mask level of 3, you get the following image:



Project Management Plan:

Plan for distribution of work, Yash will handle most of the code and other members will support where they can. Work together on the report and coimmunicate.,

Contributions-

Adrian: Handled communication and assisted with the initial code, and the Project overview, Project management plan and Project motivation.

Yash M: Wrote the entirety of the code and fixed multiple issues we ran into. Led in planning the code and educated the rest of the group the necessary python concepts.

Tommy: Contributes to the point referencing and debugging of an incorrect gray scaling issue, and Gaussian filter.

Sergio: Assisted with the Up sampling and input statements. On the report portion of the project, he was responsible for the User manual and shorter flowcharts.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
#getting the coordinates from the images
def get_click_coordinates(image):
x_click, y_click = None, None


def onclick(event):
nonlocal x_click, y_click
x_click, y_click = event.xdata, event.ydata
plt.close()


plt.imshow(image, cmap='gray')
cid = plt.connect('button_press_event', onclick)
plt.show()
plt.disconnect(cid)
```

```
x = int(x_click) if x_click is not None else None
y = int(y_click) if y_click is not None else None


return x, y


# Function to convert an RGB image to grayscale
def rgb_to_grayscale(image):
if len(image.shape) == 3:
return 0.2989 * image[:, :, 0] + 0.5870 * image[:, :, 1] + 0.1140 * image[:, :, 2]
else:
return image


# Function to apply a Gaussian filter to a grayscale image
def gaussian(gray):
kernel = np.array([
[1, 4, 7, 4, 1],
[4, 16, 26, 16, 4],
[7, 26, 41, 26, 7],
[4, 16, 26, 16, 4],
[1, 4, 7, 4, 1]
]) / 273


pad_width = 5 // 2
padded_image = np.pad(gray, pad_width, mode='constant')
placeholder = np.zeros_like(gray)


r, c = np.shape(gray)
```

```
for i in range(r):
for j in range(c):
patch = padded_image[i:i+5, j:j+5]
placeholder[i, j] = np.sum(patch * kernel)


return placeholder


#Function to upscale an image
def upsample(subs_image):
ups_size = (subs_image.shape[0] * 2, subs_image.shape[1] * 2)
ups_grid = np.zeros(ups_size)


ups_grid[::2, ::2] = subs_image


for i in range(1, ups_size[0] - 1, 2):
for j in range(1, ups_size[1] - 1, 2):
x1 = (ups_grid[i-1, j-1] + ups_grid[i+1, j-1]) / 2
x2 = (ups_grid[i-1, j+1] + ups_grid[i+1, j+1]) / 2
central = (x1 + x2) / 2
ups_grid[i, j] = central


for i in range(1, ups_size[0] - 1):
for j in range(1, ups_size[1] - 1):
if ups_grid[i, j] == 0:
avg = (ups_grid[i, j-1] + ups_grid[i-1, j] + ups_grid[i, j+1] + ups_grid[i+1, j]) / 4
```

```
ups_grid[i, j] = avg


for i in range(1, ups_size[0] - 1):
for j in range(0, ups_size[1]):
if ups_grid[i, j] == 0:
avg = (ups_grid[i-1, j] + ups_grid[i+1, j]) / 2
ups_grid[i, j] = avg


for i in range(0, ups_size[0]):
for j in range(1, ups_size[1] - 1):
if ups_grid[i, j] == 0:
avg = (ups_grid[i, j-1] + ups_grid[i, j+1]) / 2
ups_grid[i, j] = avg


return ups_grid


# Function to create a binary mask for a source image
def mask_source(image, x1, y1, x2, y2):
grid_size = (image.shape[0], image.shape[1])
mask_grid = np.ones(grid_size, dtype='bool')
mask_grid[y1:y2, x1:x2] = 0
return mask_grid


# Function to create a binary mask for a target image
def mask_target(image, x1, y1, x2, y2, horiz, vert):
half_height = (y2 - y1) // 2
half_width = (x2 - x1) // 2
```

```
grid_size = (image.shape[0], image.shape[1])
mask_grid = np.zeros(grid_size, dtype='bool')
mask_grid[vert-half_height:vert+half_height,horiz-half_width:horiz+half_width]=1
return mask_grid


# Function to multiply an image with a binary mask
def mask_multiply(image, mask):
product = image * mask
return product


# Function to downscale an image by a factor of 2
def subsample(image):
if image is None:
return None
subsampled_image = image[::2, ::2]
return subsampled_image


#Input and load the source and target images
source_image_file = input("Enter the name of the target image file: ")
if source_image_file:
source_image = plt.imread(source_image_file)
source_range_location_1 = get_click_coordinates(source_image)
source_range_location_2 = get_click_coordinates(source_image)
if source_range_location_1 is not None and source_range_location_2 is not None:
x1, y1 = source_range_location_1
x2, y2 = source_range_location_2
else:
print("Failed to obtain target image coordinates.")
```

```
exit()


target_image_file = input("Enter the name of the source image file: ")
if target_image_file:
target_image = plt.imread(target_image_file)
target_location = get_click_coordinates(target_image)
if target_location is not None:
horiz, vert = target_location
else:
print("Failed to obtain source image coordinates.")
exit()


pyramid_levels = int(input("Enter the number of pyramid levels: "))


# Create Gaussian pyramids for the source and target images
def gaussian_pyramid(image, pyramid_levels):
pyramid = [rgb_to_grayscale(image)]


for i in range(pyramid_levels):
if i < len(pyramid):
blurred_image = gaussian(pyramid[i])
subsampled_image = subsample(blurred_image)
pyramid.append(subsampled_image)
else:
break # Break the loop if the pyramid reaches the desired number of levels
```

```
return pyramid


# Create Laplacian pyramids for the source and target images
def laplacian_pyramid(gaussian_pyramid):
pyramid_levels = len(gaussian_pyramid)
laplacian = []


for i in range(1,pyramid_levels):
upsampled_image = upsample(gaussian_pyramid[i])
upsampled_image = gaussian(upsampled_image)
expanded_image = np.pad(upsampled_image, ((1, 0), (1, 0)), mode='constant') # Adjust padding if necessary
difference = gaussian_pyramid[i-1] - expanded_image[:gaussian_pyramid[i-1].shape[0], :gaussian_pyramid[i-1].shape[1]]
laplacian.append(difference)


# The last level of the Laplacian pyramid is the same as the last level of the Gaussian pyramid
laplacian.append(gaussian_pyramid[-1])


return laplacian


# Mask and manipulate image pyramids
gaussian_pyr = gaussian_pyramid(source_image, pyramid_levels)
laplacian_pyr = laplacian_pyramid(gaussian_pyr)
gaussian_pyr2 = gaussian_pyramid(target_image,pyramid_levels)
laplacian_pyr2 = laplacian_pyramid(gaussian_pyr2)
# Display images in Gaussian pyramid
```

```
source_masked_pyramid = []
x1_source, y1_source, x2_source, y2_source = source_range_location_1[0], source_range_location_1[1], source_range_location_2[0], source_range_location_2[1]


for i, laplacian_level in enumerate(laplacian_pyr[:pyramid_levels]):
# Apply the source mask to the source Laplacian level
source_masked_image = mask_source(laplacian_level, x1_source, y1_source, x2_source, y2_source)
source_masked_image = mask_multiply(laplacian_level, source_masked_image)
source_masked_pyramid.append(source_masked_image)


# Update source coordinates for the next level (divide by 2) if not the last level
if i <= pyramid_levels - 1:
x1_source //= 2
y1_source //= 2
x2_source //= 2
y2_source //= 2


target_masked_pyramid = []
half_height = (y2 - y1) // 2
half_width = (x2 - x1) // 2


# Calculate target coordinates
x1_target = horiz - half_width
x2_target = horiz + half_width
y1_target = vert - half_height
y2_target = vert + half_height
```

```
for i, laplacian_level in enumerate(laplacian_pyr2[:pyramid_levels]):
# Apply the target mask to the target Laplacian level
target_masked_image = mask_target(laplacian_level, x1_target, y1_target, x2_target, y2_target, horiz, vert)
target_masked_image = mask_multiply(laplacian_level,  target_masked_image)
target_masked_pyramid.append(target_masked_image)


# Update target coordinates for the next level (divide by 2) if not the last level
if i <= pyramid_levels - 1:
x1_target //= 2
y1_target //= 2
x2_target //= 2
y2_target //= 2
horiz //= 2 # Update horizontal coordinate for the target mask
vert //= 2 # Update vertical coordinate


def extract_face(image):
# Find the coordinates of non-zero pixels in the image
non_zero_coords = np.transpose(np.nonzero(image))
if len(non_zero_coords) == 0:
return None # No non-zero pixels found, no face to extract
# Calculate the bounding box of the non-zero pixels
y_min, x_min = np.min(non_zero_coords, axis=0)
y_max, x_max = np.max(non_zero_coords, axis=0)
# Extract the face region
face = image[y_min:y_max+1, x_min:x_max+1]
```

```
return face



x1, y1 = source_range_location_1
print(x1,y1)


def paste_face(source_data, face, source_coords):
y1, x1 = source_coords


# Ensure the face and the destination region have the same dimensions
y2, x2 = y1 + face.shape[0], x1 + face.shape[1]
# Create a copy of the source data for manipulation
result_data = np.copy(source_data)


# Paste the face into the specified region
result_data[y1:y2,x1:x2] = face


return result_data



for i in range(pyramid_levels):
face = extract_face(target_masked_pyramid[i])
if face is not None:
source_masked_pyramid[i] = paste_face(source_masked_pyramid[i], face, (y1//(2**i),
x1//(2**i)))
```

```
img = gaussian_pyr[i]
img2 = upsample(img)
img3 = gaussian(img2)
mask2 = mask_source(img3, x1//2**(i-1), y1//2**(i-1), x2//2**(i-1), y2//2**(i-1))
final_image = mask_multiply(img3,mask2)


x1,y1 = source_range_location_1
x2,y2 = source_range_location_2
horiz,vert = target_location
img4 = gaussian_pyr2[i]
img5 = upsample(img4)
img6 = gaussian(img5)
mask3 = mask_target(img6, x1//2**(i-1), y1//2**(i-1), x2//2**(i-1), y2//2**(i-1),
horiz//2**(i-1), vert//2**(i-1))
final_image2 = mask_multiply(img6,mask3)


x1, y1 = source_range_location_1
final_image3 = extract_face(final_image2)
final_image4 = paste_face(final_image,final_image3,(y1//2**(i-1),x1//2**(i-1)))



# Resize final_image4 and source_masked_pyramid[i-1] to match minimum dimensions
min_height = np.minimum(final_image4.shape[0], source_masked_pyramid[i-1].shape[0])
min_width = np.minimum(final_image4.shape[1], source_masked_pyramid[i-1].shape[1])


resized_final_image4 = final_image4[:min_height, :min_width]
resized_imgN = source_masked_pyramid[i-1][:min_height, :min_width]
```

```
final_image4 = np.array(final_image4)
source_masked_pyramid[i-1] = np.array(source_masked_pyramid[i-1])


min_height = min(final_image4.shape[0], source_masked_pyramid[i-1].shape[0])
min_width = min(final_image4.shape[1], source_masked_pyramid[i-1].shape[1])


resized_final_image4 = np.zeros_like(source_masked_pyramid[i-1])
resized_final_image4[:min_height, :min_width] = final_image4[:min_height, :min_width]


image1 = resized_final_image4+ source_masked_pyramid[i-1]
image2 = upsample(image1)
image3 = gaussian(image2)


image3 = np.array(image3)
source_masked_pyramid[i-2] = np.array(source_masked_pyramid[i-2])
min_height = min(image3.shape[0], source_masked_pyramid[i - 2].shape[0])
min_width = min(image3.shape[1], source_masked_pyramid[i - 2].shape[1])


resized_Image3 = np.zeros_like(image3)
resized_Image3[:min_height, :min_width] = source_masked_pyramid[i-2][:min_height,
:min_width]
```

```
Image4 = image3 + source_masked_pyramid[i-2][:min_height, :min_width]
image5 = upsample(Image4)
Image6 = gaussian(image5)



# Resize Image6 to match source_masked_pyramid[i - 3] dimensions
min_height = min(Image6.shape[0], source_masked_pyramid[i - 3].shape[0])
min_width = min(Image6.shape[1], source_masked_pyramid[i - 3].shape[1])


resized_Image6 = np.zeros_like(source_masked_pyramid[i - 3])
resized_Image6[:min_height, :min_width] = Image6[:min_height, :min_width]


x = upsample(source_masked_pyramid[0])
x2 = gaussian(x)
Image7 = Image6 + x


plt.imshow(Image7, cmap='gray')
plt.title('Final image (img)')
plt.show()


print(source_masked_pyramid[2].shape)
print(source_masked_pyramid[1].shape)
print(source_masked_pyramid[0].shape)
```