Project 2

Part 1

Part1.1：Finite Difference Operator

1. Image Loading and Preprocessing
Load the PNG image from the specified path, check if there is an alpha channel, and remove it. Then, convert the RGB image to grayscale to simplify processing.

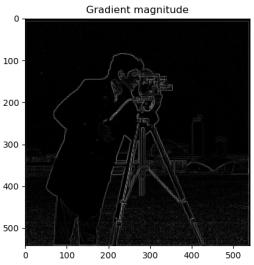
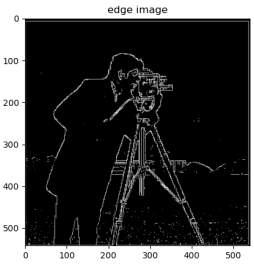
2. Define Finite Difference Operators
Use two simple difference operators:

- $Dx = [1, -1] D_x = [1, -1] Dx = [1, -1]$ to detect horizontal changes.
- $Dy = [1 - 1] D_y = \begin{bmatrix} 1 \\ -1 \end{bmatrix} Dy = [1 - 1]$ to detect vertical changes.

3. Convolution to Compute Gradients
Perform 2D convolution between the image and the D_x and D_y operators, computing gradients in the horizontal and vertical directions to detect local changes in the image.

4. Compute Gradient Magnitude
Use the formula $\sqrt{(\text{grad}_x^2 + \text{grad}_y^2)}$ to calculate the gradient magnitude, combining changes in both the x and y directions to highlight the edges in the image.

5. Binarization
Apply a threshold to the gradient magnitude image to perform binarization, marking pixels above the threshold as edges, resulting in the final edge-detected image. After sometimes try, I finally choose 0.2 to be the threhshold, which has the best performance.

Before	Gradient magnitude	edge image
		

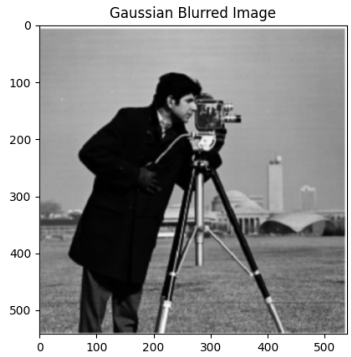
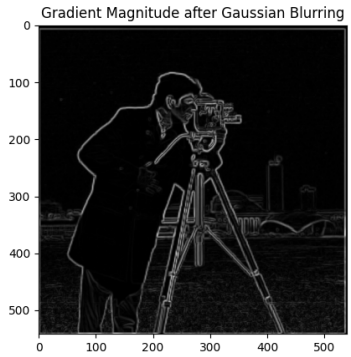
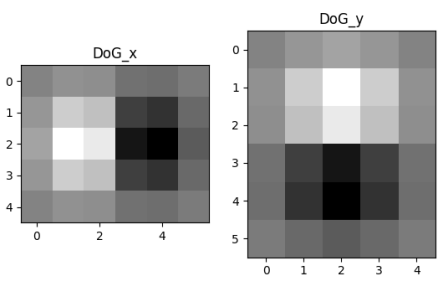
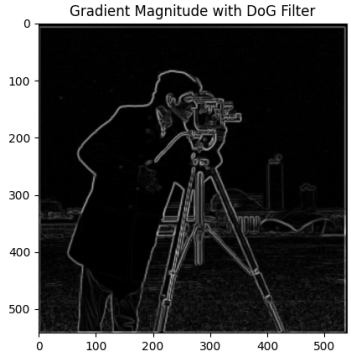
Part 1.2：Derivative of Gaussian (DoG) Filter

Algorithm Overview: Derivative of Gaussian (DoG) Filter

In this part, we use Gaussian smoothing to reduce image noise and combine it with a finite difference operator for edge detection. The process starts by applying Gaussian blur to smooth the image, followed by edge detection using a finite difference operator. We then optimize the process by combining the Gaussian smoothing and the difference operator into a single step using the **Derivative of Gaussian (DoG) filter**, which allows us to perform both operations with a single convolution.

Algorithm:

1. **Image Loading and Grayscale Conversion:**
   The image is loaded and converted to grayscale to ensure we are working with a single-channel image.
2. **Gaussian Smoothing:**
   Using the `getGaussianKernel()` function from OpenCV, a 1D Gaussian kernel is generated. This 1D kernel is used to create a 2D Gaussian kernel via outer product. The 2D Gaussian kernel is then convolved with the grayscale image to produce a blurred version of the original image, reducing noise and smoothing the details.
3. **Convolution to Compute Image Gradients:**
   We convolve the blurred image with the finite difference operators DxD_xDx and DyD_yDy to compute the gradients in the horizontal and vertical directions, respectively. This step extracts edge information from the smoothed image.
4. **Compute Gradient Magnitude:**
   By combining the horizontal and vertical gradients, we compute the gradient magnitude of the image. The gradient magnitude highlights areas where changes are most significant, typically corresponding to edges in the image.
5. **Generate the Derivative of Gaussian (DoG) Filter:**
   To optimize the process, we convolve the Gaussian kernel with the difference operators to create the DoG filters. These filters allow us to perform both Gaussian smoothing and edge detection in a single convolution step. Finally, we apply the DoG filters to the image to compute the gradients and extract the edge information.

Gaussian Blurred Image	**Gradient Mgnitude after Gussian Blurring**
	
** DoG filter**	**Gradient Magnitude with DoG Filter**
	

Observed Differences:

- **Using only the difference operators**, the edge detection is more susceptible to noise, leading to irregular edges, especially when the original image contains high-frequency components where noise and edges are hard to distinguish.
- **After applying Gaussian smoothing**, the noise is reduced, resulting in smoother and more continuous edges, making the edge features more prominent and cleaner.
- **With the DoG filters**, the Gaussian smoothing and edge detection are combined into a single convolution, making the process more efficient while producing results similar to those of performing Gaussian smoothing followed by edge detection separately.

Part 2

Part 2.1: Image "Sharpening"

Algorithm:

1. **Image Loading and Grayscale Conversion**:

   The images are loaded and converted to grayscale, focusing the processing on brightness and edge information without interference from color.

2. **Gaussian Blur**:

   A Gaussian blur is applied to smooth the image, extracting its low-frequency components (smooth regions) while removing high-frequency details and noise.

3. **Extracting High-Frequency Components**:

   The original image is subtracted by the blurred image to obtain the high-frequency components, which contain details and edges.

4. **Sharpening (Unsharp Masking)**:

   The extracted high-frequency components are multiplied by a factor `alpha` and added back to the original image. This enhances the image's edges and details, resulting in a sharpened effect. The larger the `alpha` value, the stronger the sharpening effect.

5. **Brightness Matching**:

   Since sharpening can change the image's brightness, histogram matching (`exposure.match_histograms`) is used to adjust the sharpened image's brightness to match the original image, preventing the sharpened image from being too bright or too dark.

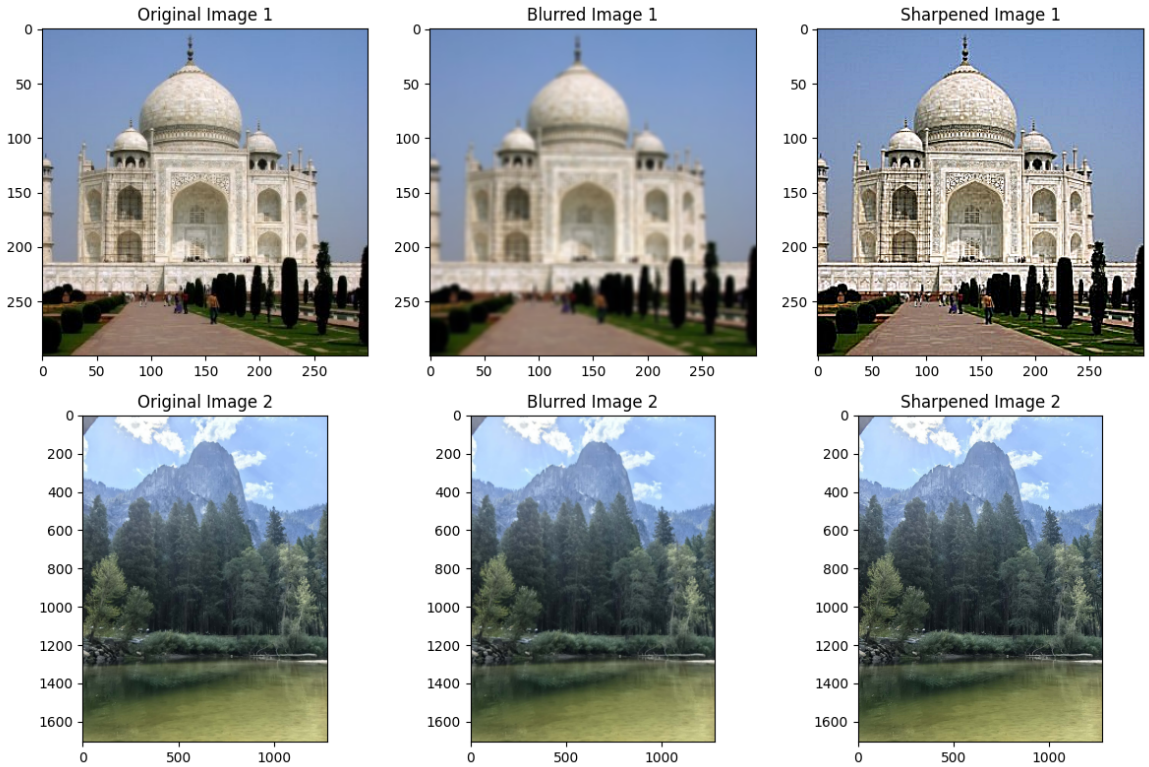
6. **Imgaes**

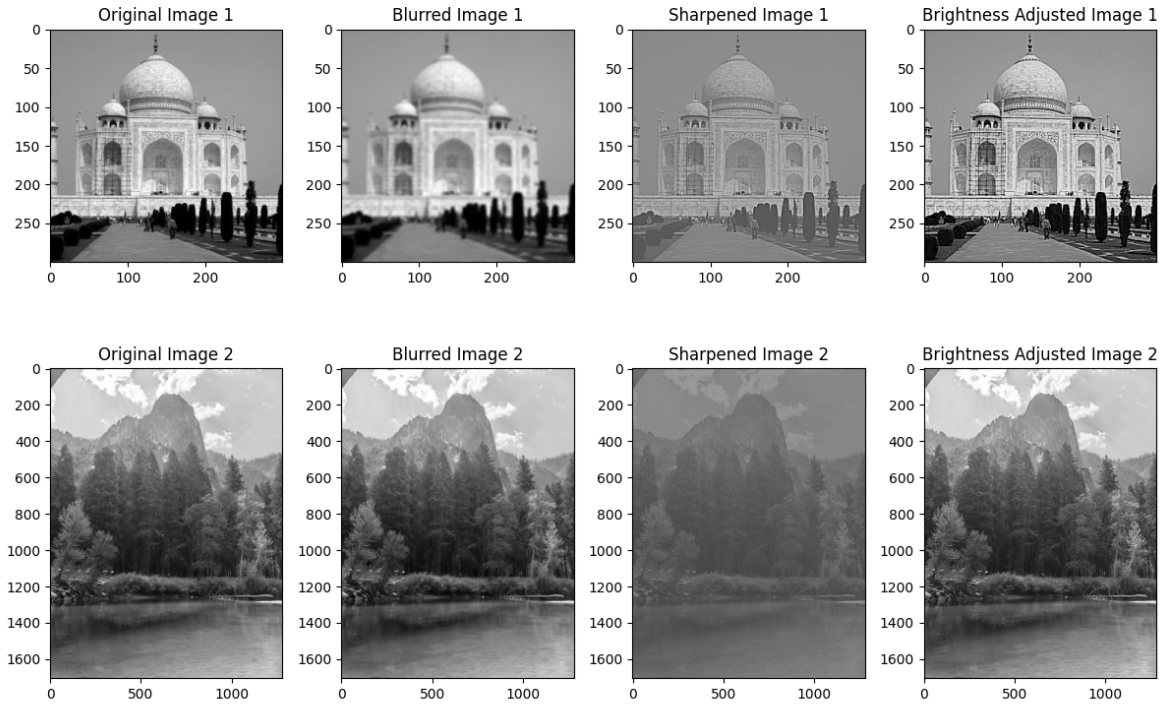
   Image1 and Image2 are the original clear pictures, of which picture 2 has a higher pixel. I blurred them, extracted the features and then sharpened them. Here's the result.

   To see the difference clearer, I exchange the imgaes into gray imgaes as well.

Core Steps:

- **Gaussian Blur**: Separates the low-frequency (smooth) parts of the image.
- **High-Frequency Extraction**: Subtracts the blurred image from the original to isolate details.
- **High-Frequency Weighting**: Enhances the sharpened effect by adding back weighted high-frequency components.
- **Brightness Matching**: Ensures the brightness and contrast of the sharpened image remain consistent with the original.





Sharpen blurry image

1. **Gaussian Blur**:

The image is smoothed using a Gaussian blur. This acts as a low-pass filter that removes high-frequency components such as noise and fine details.

2. **Extracting High-Frequency Components**:

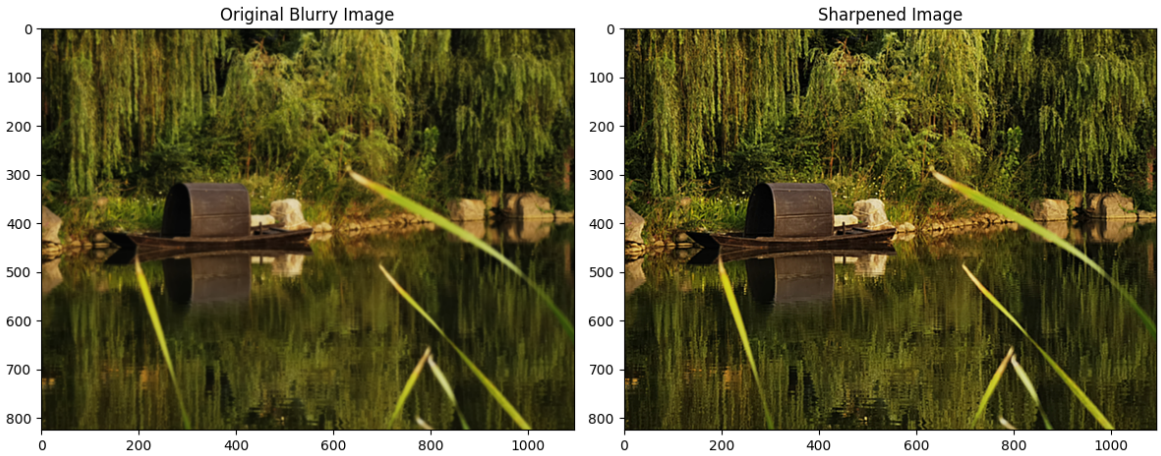
The high-frequency components (edges and details) are obtained by subtracting the blurred image from the original image. This isolates the sharp details that were removed by the blur.

3. **Sharpening**:

The high-frequency components are multiplied by a factor `alpha` and added back to the original image using `cv2.addWeighted()`. This step enhances the edges and details, making the image sharper. The `alpha` value controls the intensity of the sharpening effect.

4. **Display**:

Finally, the original blurry image and the sharpened image are displayed side by side for comparison.



Part 2.2: Hybrid Images

This part focuses on generating a hybrid image by combining the low-frequency content of one image with the high-frequency content of another image, and then analyzing its frequency spectrum using Fourier transforms. Here's a brief explanation of the algorithm:

1. Image Alignment

The `align_images` function is used to align two images:

- **Manual Point Selection**: Users select key points in both images to align the center, scale, and rotation.
- **Center Alignment**: The `align_image_centers` function aligns the centers of the two images by recalculating the center of the selected key points.
- **Image Rescaling**: The `rescale_images` function adjusts the scale of the images, making sure both images have the same scale.
- **Image Rotation**: The `rotate_im1` function rotates one image based on the relative angle between the key points.
- **Size Matching**: The `match_img_size` function ensures that both images have the same size by trimming or padding the edges as needed.

2. Hybrid Image Generation

The `hybrid_image` function is responsible for generating the hybrid image through the following steps:

- **Low-Frequency Part**: The first image (`im1`) is blurred using a Gaussian filter to extract its low-frequency content.
- **High-Frequency Part**: The second image (`im2`) is also blurred, and the result is subtracted from the original image to retain the high-frequency information.
- **Combining Frequencies**: The low-frequency and high-frequency parts are added together to create the hybrid image. The result is clipped to ensure pixel values remain within a valid range using `np.clip`.

3. Fourier Spectrum Analysis

The `display_fft` function is used to visualize the frequency spectrum of the images:

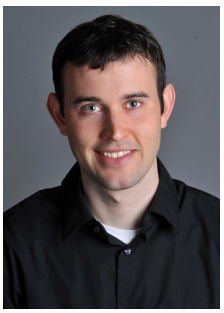
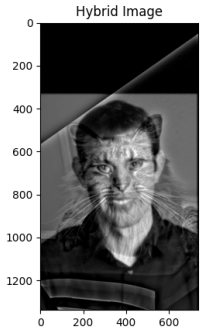
- **Fourier Transform**: The `np.fft.fft2` function computes the 2D Fourier transform of the image, and `fftshift` shifts the zero-frequency component to the center of the spectrum.
- **Displaying the Spectrum**: The magnitude of the Fourier transform is computed, and logarithmic scaling is applied to avoid large value ranges. The result is then visualized as a grayscale image representing the frequency distribution.

Summary

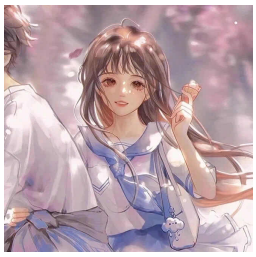
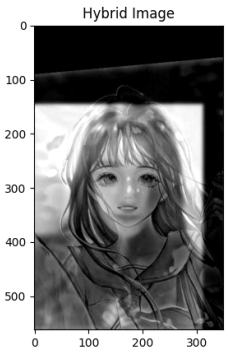
- **Image Alignment**: Key points are manually selected by the user to align the images in terms of center, scale, and rotation.
- **Hybrid Image Generation**: The low-frequency information from one image is combined with the high-frequency information from another to produce a hybrid image.
- **Fourier Spectrum Analysis**: Fourier transforms are used to analyze the frequency characteristics of the images, and the results are displayed as grayscale frequency spectrum images.

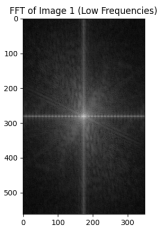
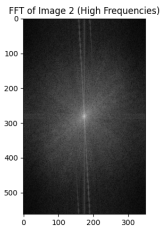
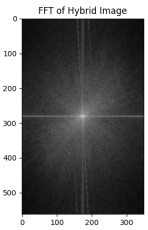
Results

- **The result of running a given image:**

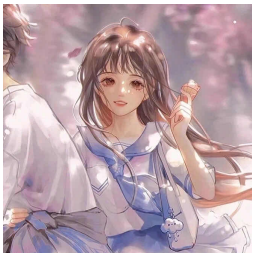
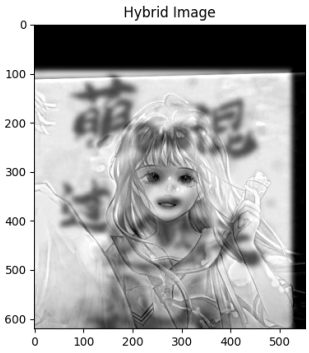
Low Frequency	High Frequency	Result
		

- **Result with Fourier frequency analysis spectrum:**

Low Frequency	High Frequency	Result
		

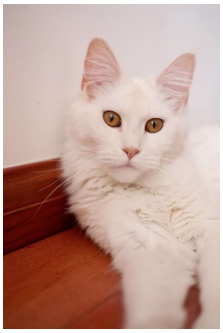
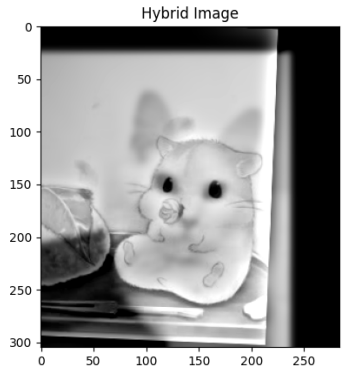
FFT of Image 1 (Low Frequencies)	FFT of Image 2 (High Frequencies)	FFT of Hybrid Image
		

- **Some other atemps:**

Low Frequency	High Frequency	Result
		

- **Failed attemps :**

We are not able to recognize the low frequency image of the cat even if we stand really far away. That might because the cat's head is not very clear.

Low Frequency	High Frequency	Result
		

Part 2.3 : Gaussian and Laplacian Stacks

Algorithms:

1. **Gaussian Stack**:
    - The Gaussian stack is created by applying Gaussian blur to the image at each level without downsampling. The 0th level is the original image, and each subsequent level is a more blurred version of the previous one.
    - Steps:
        - Start with the original image and apply Gaussian blur to generate the next level.
        - Repeat until the desired number of levels is reached, keeping the image size constant.
2. **Laplacian Stack**:
    - The Laplacian stack is generated by subtracting each level of the Gaussian stack from the next one, capturing the image's edge and detail information. The last level of the Laplacian stack is the lowest-resolution image in the Gaussian stack.
    - Steps:
        - Subtract the next Gaussian level from the current one to get the Laplacian image for that level.
        - The last level of the Laplacian stack is the last (most blurred) image from the Gaussian stack.
3. **Processing Color Images**:
    - The color image is split into three channels: red, green, and blue. Each channel is processed independently to generate Gaussian and Laplacian stacks. After processing, the channels are merged back to form a color image.
    - Steps:
        - Split the image into RGB channels.
        - Generate Gaussian and Laplacian stacks for each channel.
        - Merge the channels back together for each level of the stack.
4. **Image Normalization**:
    - Since the pixel values in the Laplacian images may exceed the usual range, the code normalizes each image's intensity to the range [0, 255] for proper visualization.
    - Steps: Use `cv2.normalize` to scale the pixel values of the image to the 0-255 range and convert the image to `uint8`.

5. **Display Stacks**:
    - The code uses `matplotlib` to display the Gaussian and Laplacian stacks in a single figure. Each level of the stack is shown side-by-side for comparison.
    - Steps:
        - Normalize the images for visualization.
        - Display both the Gaussian and Laplacian stack side by side using subplots.

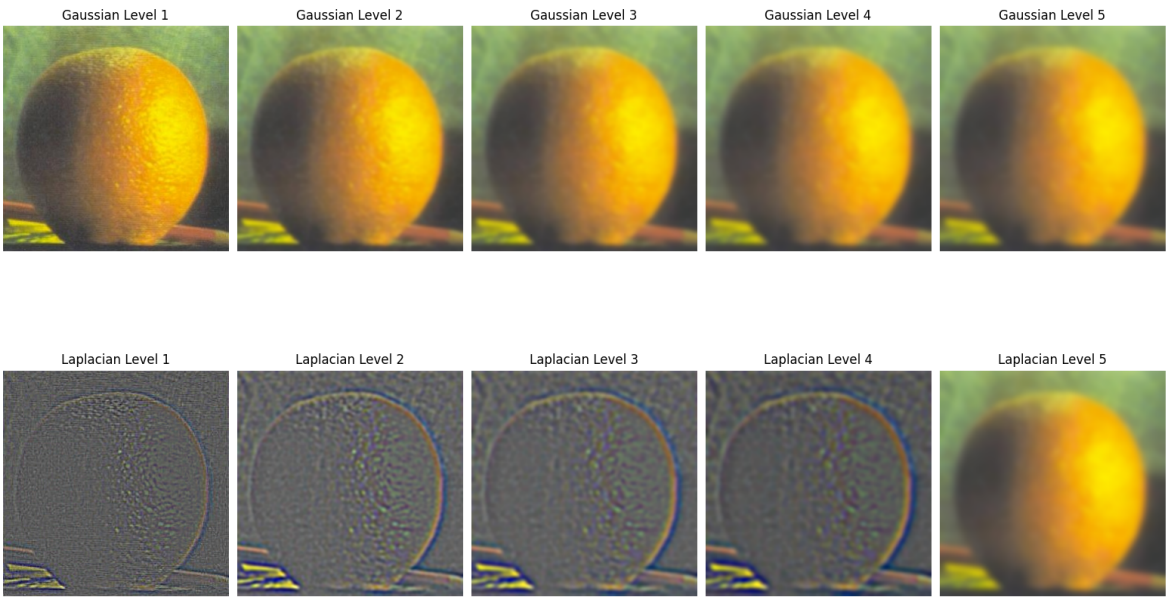
Workflow:

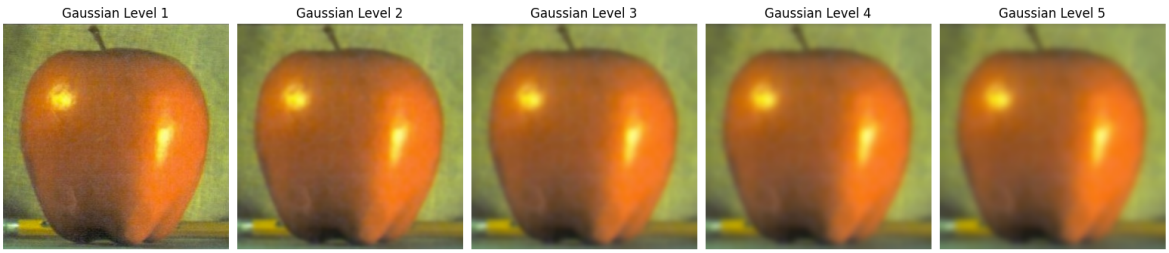
1. **Load Image**:
    - The code loads the image in color, converts it to float format, and switches from BGR to RGB for correct color representation.
2. **Generate Stacks**:
    - The `process_color_image` function is used to generate the Gaussian and Laplacian stacks for each channel of the image.
3. **Display Results**:
    - The `display_stacks` function is used to show the Gaussian and Laplacian stacks in a single figure.

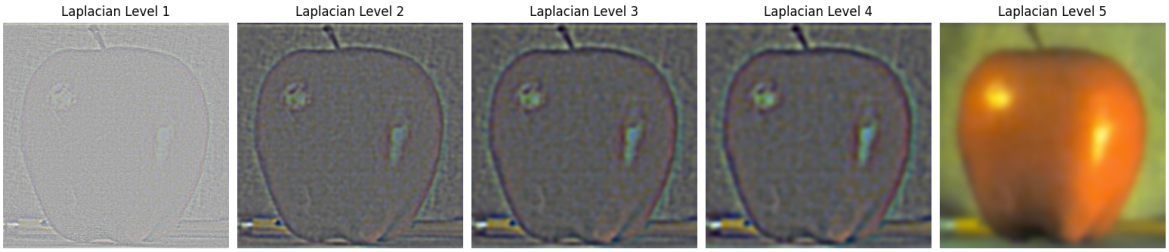
Summary:

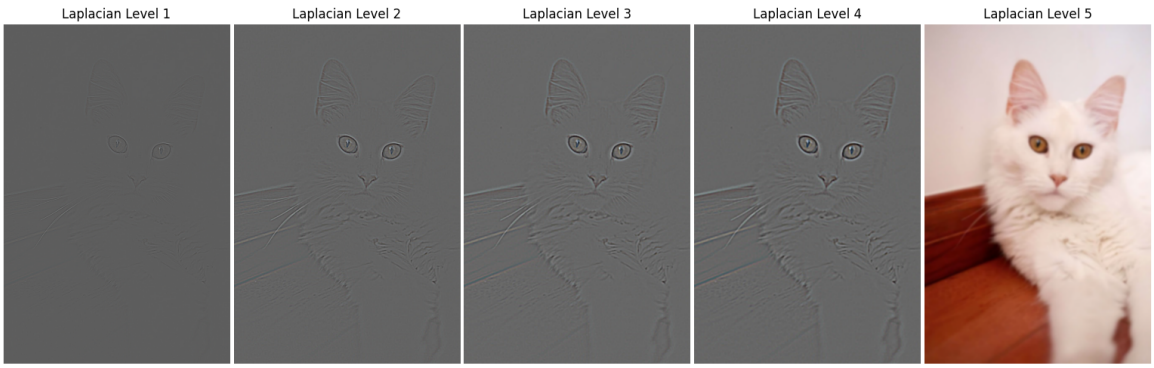
- **Gaussian Stack**: Creates multiple progressively blurred versions of the image, maintaining the same size.
- **Laplacian Stack**: Captures the edge and detail information at each level by subtracting adjacent Gaussian levels.
- **Color Processing**: Splits the image into RGB channels, processes them individually, and then merges the results.
- **Normalization**: Ensures all images are displayed with correct brightness and contrast by rescaling intensities.

Results:









Part 2.4 : Multiresolution Blending

Key Concepts:

1. **Gaussian Pyramid**: This is a series of increasingly blurred and downsampled versions of an image. Each level of the pyramid represents the image at a lower resolution.
2. **Laplacian Pyramid**: This pyramid captures the details at different scales by subtracting the Gaussian pyramid's levels. Each level shows the image's fine details, and it's created by subtracting the upsampled version of a lower-level Gaussian pyramid from the current Gaussian level.
3. **Blending**: The blending is performed using an input mask that defines the regions to blend between two images. The blending occurs at each level of the Laplacian pyramids using the

Gaussian pyramid of the mask to smooth transitions.

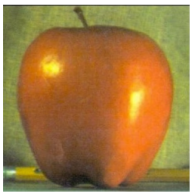
Algorithm Steps:

1. Gaussian Pyramid Creation
    - The `gaussian_stack` function creates a Gaussian pyramid for a given image by repeatedly applying Gaussian blur and downsampling.
    - The number of levels (`num_levels`) and the blur strength (`sigma`) are defined as parameters.
2. Laplacian Pyramid Creation
    - The `laplacian_stack` function computes the Laplacian pyramid by subtracting the upsampled version of a lower-resolution Gaussian level from the current Gaussian level. This captures the details (edges, textures) at each level.
3. Processing Color Images
    - The `process_color_image` function processes color images by applying the Gaussian and Laplacian pyramid construction on each color channel (R, G, B) separately.
4. Multiresolution Blending
    - The `multiresolution_blending` function blends two images (`image1` and `image2`) based on a given mask (`mask`).
    - First, Gaussian and Laplacian pyramids are created for both images and the mask.
    - At each level of the pyramids, the blending is performed using the formula: blended=laplacian1×(1−mask)+laplacian2×mask\text{blended} = \text{laplacian1} \times (1 - \text{mask}) + \text{laplacian2} \times \text{mask}blended=laplacian1×(1−mask)+laplacian2×mask
    - Finally, the blended image is reconstructed by successively upsampling and adding the blended pyramid levels from the Laplacian pyramids.

Result:

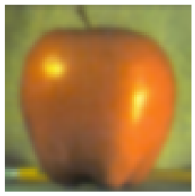
- The final blended image is produced after combining the Laplacian pyramids of both images using the mask pyramid. The result is a smooth transition between the two images at multiple scales.
- The code visualizes the blended image at the end using `matplotlib`.

Gaussian Level 1

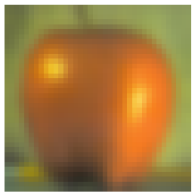


Gaussian Level 2

Gaussian Level 3



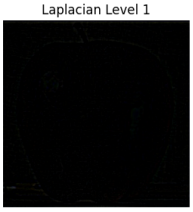
Gaussian Level 4



Gaussian Level 5

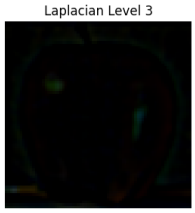
Gaussian Level 6

Laplacian Level 1



Laplacian Level 2

Laplacian Level 3



Laplacian Level 4

Laplacian Level 5

Laplacian Level 6

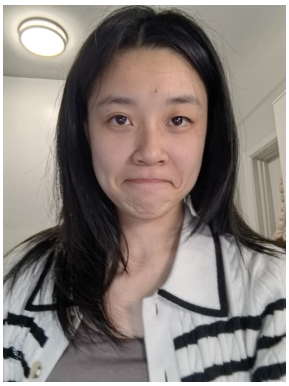
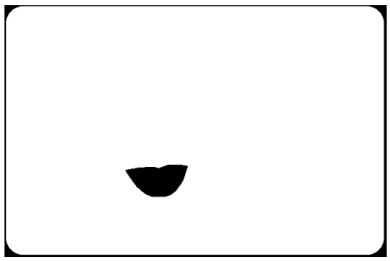
image 1	image 2	result
		Blended Image 

Image 1

Image 2

Inverted Mask



Simple Blend

Multiresolution Blended Image