Image Processing Project

Modules Used

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
```

- `numpy`
  - Imported as: `np`
  - Purpose: Provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
  - Common Use: Mathematical computations, array manipulation, and handling numerical data.
- `PIL` **(Python Imaging Library)**
  - Specifically, the `Image` module is imported.
  - Purpose: Provides capabilities for opening, manipulating, and saving image files in various formats.
  - Common Use: Image processing tasks like resizing, filtering, or converting image formats.
- `matplotlib.pyplot`
  - Imported as: `plt`
  - Purpose: Used for creating static, interactive, and animated visualizations in Python.

    - Common Use: Plotting graphs, visualizing data distributions, and displaying images.

- `warnings`
    - Purpose: Provides a mechanism to control the display of warning messages.
    - Function Used: `filterwarnings("ignore")` disables warnings, ensuring that unnecessary messages don't clutter the output.

```
def convert_to_grayscale(image):

    image_arary = np.array(image)
    if len(image_arary.shape) == 3:
        grayscale = (
            0.2989 * image_arary[:, :, 0]
            + 0.5870 * image_arary[:, :, 1]
            + 0.1140 * image_arary[:, :, 2]
        )
        return grayscale.astype(np.uint8)

    else:
        return image_arary


# def Gray_scale(image_path):

# # convert it to grayscale
#     img = Image.open(image_path).convert('L')
#     return img
```

Purpose

The provided code converts a given image into grayscale format. Grayscale conversion is commonly used in image processing to simplify analysis by reducing the color information while retaining essential intensity details.

Function 1: `convert_to_grayscale`

Parameters

- `image`: A PIL Image object representing the input image.

Returns

- A 2D NumPy array representing the grayscale version of the input image.

Functionality

1. Converts the input image into a NumPy array.
2. Checks if the image has three color channels (indicating an RGB image).
    - If `True`, computes the grayscale image using the weighted sum formula: Grayscale=0.2989×R+0.5870×G+0.1140×B
    - These weights correspond to the human eye's sensitivity to different colors.
3. Converts the computed grayscale values to 8-bit unsigned integers (`uint8`) for compatibility with image formats.
4. If the image is already single-channel (e.g., grayscale), returns it unchanged.

Function 2

: `Gray_scale` (Commented Out)

Parameters

- `image_path`: A string representing the path to the input image file.

Returns

- A PIL Image object in grayscale mode (`L`).

Functionality

1. Opens the image from the specified file path.
2. Directly converts the image to grayscale using the `convert('L')` method of PIL.
    - The `L` mode indicates 8-bit pixels, black and white.

```
def apply_threshold(gray_scale_image):
    threshold = np.mean(gray_scale_image)

    binary_image = np.where(gray_scale_image >= threshold, 255,
    return binary_image.astype(np.uint8)
```

Purpose

This script demonstrates a complete image processing pipeline involving grayscale conversion, thresholding, and histogram calculation for visualizing pixel intensity distributions.

Functions

1. `apply_threshold`

Parameters

- `gray_scale_image`: A 2D NumPy array representing a grayscale image.

Returns

- A binary image as a 2D NumPy array where:
  - Pixel intensity is 255 (white) if the original pixel is greater than or equal to the threshold.
  - Pixel intensity is 0 (black) otherwise.

Functionality

1. Calculates the threshold value as the mean pixel intensity of the grayscale image.
2. Applies the threshold to generate a binary image using NumPy's `where` function.
3. Converts the binary image to `uint8` format for compatibility with image viewers and other image processing tools.

```
image = Image.open("./Screenshot 2024-11-27 181534.png")

grayscale = convert_to_grayscale(image)

binary = apply_threshold(grayscale)

histo_image = calculate_histogram(binary)

plt.figure(figsize=(10, 5))
plt.subplot(1, 4, 1)
plt.title("original Image")
plt.imshow(image)
plt.axis("off")

plt.subplot(1, 4, 2)
plt.title("Grayscale Image")
plt.imshow(grayscale, cmap="gray")
```

```
plt.axis("off")




plt.subplot(1, 4, 3)
plt.title("Thresholded Image")
plt.imshow(binary, cmap="gray")
plt.axis("off")
plt.show()


plt.figure(figsize=(10, 5))
plt.bar(range(256), histo_image, color="gray")
plt.title("Histogram")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.show()
```

Workflow

1. Grayscale Conversion

- The script uses the `convert_to_grayscale` function to transform an input image into grayscale.

2. Thresholding

- The `apply_threshold` function is applied to the grayscale image to produce a binary image.

3. Histogram Calculation

- The script calculates the histogram of the binary image using a presumed function `calculate_histogram`.

4. Visualization

- **Original Image**: Displays the original image.
- **Grayscale Image**: Displays the converted grayscale image.
- **Binary (Thresholded) Image**: Displays the binary image after thresholding.
- **Histogram**: Shows the pixel intensity distribution in a bar chart.

`simple_halftoning` Function

Purpose

The `simple_halftoning` function performs basic halftoning, a process of converting a grayscale image into a binary (black-and-white) image. The function uses a single intensity threshold to decide whether each pixel in the grayscale image should be represented as black or white.

```
def simple_halftoning(grayscale_image, threshold=128):

    binary_image = np.where(grayscale_image >= threshold, 255, (
    return binary_image.astype(np.uint8)
```

Parameters

- `grayscale_image`:

  A 2D NumPy array representing the grayscale image where pixel intensities range from 0 (black) to 255 (white).

  - Input should already be converted to grayscale.

- `threshold` *(optional)*:

An integer value (default: 128) used to separate the pixel intensities into two classes:

- **White (255)**: Pixels with intensity greater than or equal to the threshold.
- **Black (0)**: Pixels with intensity below the threshold.

Returns

- `binary_image`:A binary image as a 2D NumPy array, where:
  - Pixel values are either 255 (white) or 0 (black).
  - Data type is `uint8` for compatibility with image viewers and further processing.

Functionality

1. **Thresholding Logic**:
   - The function applies a threshold to each pixel in the grayscale image using NumPy's `where` function:
     - If `pixel_intensity >= threshold`, assign `255` (white).
     - Else, assign `0` (black).
2. **Data Type Conversion**:
   - Converts the binary image to `uint8` format for compatibility with standard image-processing libraries.

`Advanced_halftoning` Function

Purpose

The `Advanced_halftoning` function implements error-diffusion halftoning, a technique used to convert grayscale images into binary (black-and-white) images. It uses the Floyd-Steinberg dithering algorithm, which distributes the quantization error of a pixel to its neighboring pixels to achieve visually pleasing halftoning.

```
def Advanced_halftoning(grayscale_image):
    rows, cols = grayscale_image.shape
    output_image = grayscale_image.copy()

    for i in range(rows):
        for j in range(cols):
            old_pixel = output_image[i, j]
            new_pixel = 255 if old_pixel > 128 else 0
            output_image[i, j] = new_pixel

            error = old_pixel - new_pixel

            if j + 1 < cols:
                output_image[i, j + 1] += (7 / 16) * error
            if i + 1 < rows and j - 1 >= 0:
                output_image[i + 1, j - 1] += (3 / 16) * error
            if i + 1 < rows:
                output_image[i + 1, j] += (5 / 16) * error
            if i + 1 < rows and j + 1 < cols:
                output_image[i + 1, j + 1] += (1 / 16) * error

    return np.clip(output_image, 0, 255).astype(np.uint8)
```

Parameters

- `grayscale_image`:A 2D NumPy array representing the grayscale image, where pixel intensities range from 0 (black) to 255 (white).

Returns

- `output_image`:A binary (halftoned) image as a 2D NumPy array, with pixel values either 0 (black) or 255 (white). The output is clipped to the valid intensity range `[0, 255]` and converted to `uint8`.

Functionality

1. **Initialization**:
   - Copies the input grayscale image to `output_image` to avoid modifying the original image.
   - Retrieves the number of rows and columns of the input image.
2. **Thresholding**:
   - For each pixel, compares the pixel intensity to a fixed threshold (`128`):
     - If intensity > 128, sets the pixel to 255 (white).
     - Otherwise, sets it to 0 (black).
3. **Error Calculation**:
   - Computes the quantization error (`old_pixel - new_pixel`), which is the difference between the original and thresholded pixel values.
4. **Error Diffusion**:
   - Distributes the quantization error to neighboring pixels using the Floyd-Steinberg error diffusion formula:
     - Right neighbor: `(7/16) * error`
     - Bottom-left neighbor: `(3/16) * error`
     - Bottom neighbor: `(5/16) * error`
     - Bottom-right neighbor: `(1/16) * error`
5. **Clipping**:
   - Ensures pixel values remain within the valid range `[0, 255]` using `np.clip`.

`calculate_histogram` Function

Purpose

The `calculate_histogram` function computes the histogram of a grayscale image. A histogram represents the frequency distribution of pixel intensity values ranging from 0 (black) to 255 (white).

```
def calculate_histogram(grayscale_image):
    histogram = np.zeros(256, dtype=int)

    for value in grayscale_image.flatten():
        histogram[value] += 1

    return histogram
```

Parameters

- `grayscale_image`:A 2D NumPy array representing a grayscale image, where pixel intensities range from 0 to 255.

Returns

- `histogram`:A 1D NumPy array of size 256, where:
  - `histogram[i]` indicates the frequency of the pixel intensity value `i` in the grayscale image.

Functionality

1. **Initialization**:

- Creates a zero-filled NumPy array of size 256 to store frequency counts for each pixel intensity.

2. **Histogram Calculation**:
   - Flattens the 2D grayscale image into a 1D array using `flatten()`.
   - Iterates through each pixel value and increments the corresponding index in the histogram array.

`histogram_equalization` Function

Purpose

The `histogram_equalization` function enhances the contrast of a grayscale image by redistributing the pixel intensity values. This is achieved by using the histogram of the image to spread out the most frequent intensity values.

```
def histogram_equalization(grayscale_image):


    histogram = calculate_histogram(grayscale_image)


    cumulative_histogram = [0] * len(histogram)
    cumulative_histogram[0] = histogram[0]
    for i in range(1, len(histogram)):
        cumulative_histogram[i] = cumulative_histogram[i - 1] +


    dm = 256
    area = grayscale_image.shape[0] * grayscale_image.shape[1]


    normalized_cumulative_histogram = []
```

```
    for value in cumulative_histogram:
        normalized_value = round((dm - 1) * (value / area))
        normalized_cumulative_histogram.append(normalized_value)


    equalized_image = np.zeros_like(grayscale_image)
    rows, cols = grayscale_image.shape
    for i in range(rows):
        for j in range(cols):
            equalized_image[i, j] = int(normalized_cumulative_hi

    return (
        equalized_image.astype(np.uint8),
        histogram,
        cumulative_histogram,
        normalized_cumulative_histogram,
    )
```

Parameters

- `grayscale_image`:A 2D NumPy array representing the grayscale image, where pixel intensities range from 0 to 255.

Returns

- `equalized_image`:The contrast-enhanced image as a 2D NumPy array with pixel values in the range `[0, 255]`.
- `histogram`:The original histogram of the grayscale image.
- `cumulative_histogram`:The cumulative frequency of pixel intensities from the original histogram.
- `normalized_cumulative_histogram`:The scaled cumulative histogram, normalized to the intensity range `[0, 255]`.

Functionality

1. **Histogram Calculation**:
    - Calls `calculate_histogram` to get the frequency distribution of pixel intensity values.
2. **Cumulative Histogram**:
    - Computes the cumulative sum of the histogram to measure the distribution of pixel intensities.
3. **Normalization**:
    - Scales the cumulative histogram values to the range `[0, 255]`.
4. **Pixel Mapping**:
    - Maps each pixel intensity in the grayscale image to its corresponding value in the normalized cumulative histogram to create the equalized image.

`sobel_operator` Function

Purpose

The `sobel_operator` function applies the Sobel operator to a grayscale image to detect edges by calculating the gradient magnitude at each pixel. The function highlights regions of high intensity change, which correspond to edges in the image.

```
# def sobel_operator(grayscale_image, threshold=128):
#     Gx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
#     Gy = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

#     rows, cols = grayscale_image.shape
#     edge_image = np.zeros_like(grayscale_image)
```

```
#     for i in range(1, rows - 1):
#         for j in range(1, cols - 1):
#             region = grayscale_image[i - 1 : i + 2, j - 1 : j

#             gx = np.sum(Gx * region)
#             gy = np.sum(Gy * region)

#             magnitude = np.sqrt(gx**2 + gy**2)
#             #threshold += magnitude

#             edge_image[i, j] = 255 if magnitude > threshold el

#     return edge_image.astype(np.uint8)
```

Parameters

- `grayscale_image`:

  A 2D NumPy array representing the input grayscale image, where pixel values range from 0 to 255.

- `threshold` (default = 128):

  A scalar value used to determine whether a pixel is part of an edge. If the gradient magnitude at a pixel exceeds the threshold, the pixel is classified as part of an edge.

Returns

- `edge_image`:A binary image (2D NumPy array) where edge pixels have a value of 255 (white) and non-edge pixels have a value of 0 (black).

Functionality

1. **Gradient Kernels**:
   - The function uses the Sobel kernels `Gx` and `Gy` to compute horizontal and vertical gradients at each pixel.

2. **Gradient Calculation**:
    - For each pixel, the function extracts a 3×3 region centered on the pixel and computes:
        - Horizontal gradient (`gx`) using `Gx`.
        - Vertical gradient (`gy`) using `Gy`.
3. **Gradient Magnitude**:
    - The magnitude of the gradient is computed as:
      magnitude=gx2+gy2

      magnitude=gx2+gy2\text{magnitude} = \sqrt{g_x^2 + g_y^2}
4. **Edge Detection**:
    - If the gradient magnitude exceeds the specified threshold, the pixel is marked as an edge (255). Otherwise, it is marked as non-edge (0).

`sobel_operator` Function

Purpose

The `sobel_operator` function detects edges in a grayscale image using the Sobel method. It computes the gradient magnitude for each pixel, dynamically determines a threshold based on the average gradient magnitude, and generates a binary image where edges are highlighted.

```
def sobel_operator(grayscale_image):
    Gx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    Gy = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

    rows, cols = grayscale_image.shape
    edge_image = np.zeros_like(grayscale_image)
    magnitudes = []
```

```
    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            region = grayscale_image[i - 1 : i + 2, j - 1 : j +

            gx = np.sum(Gx * region)
            gy = np.sum(Gy * region)

            magnitude = np.sqrt(gx**2 + gy**2)
            magnitudes.append(magnitude)


    threshold = np.mean(magnitudes)


    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            region = grayscale_image[i - 1 : i + 2, j - 1 : j +

            gx = np.sum(Gx * region)
            gy = np.sum(Gy * region)

            magnitude = np.sqrt(gx**2 + gy**2)
            edge_image[i, j] = 255 if magnitude > threshold else

    return edge_image.astype(np.uint8)
```

Parameters

- `grayscale_image`:A 2D NumPy array representing the grayscale input image. Each pixel intensity should range from 0 to 255.

Returns

- `edge_image`:A binary image (2D NumPy array) where detected edges have a pixel intensity of 255 (white), and non-edge areas have a pixel intensity of 0 (black).

Functionality

1. **Gradient Kernels**:
   - Uses predefined Sobel kernels `Gx` (horizontal gradients) and `Gy` (vertical gradients) to compute gradients.
2. **Gradient Magnitude Calculation**:
   - Computes the gradient magnitude for each pixel using:

$$\text{magnitude} = \sqrt{g_x^2 + g_y^2}$$

3. **Dynamic Thresholding**:
   - Dynamically calculates the threshold as the mean gradient magnitude across all pixels.
4. **Edge Detection**:
   - Pixels with gradient magnitudes exceeding the threshold are marked as edges (255); others are marked as non-edges (0).

`prewitt_operator` Function

Purpose

The `prewitt_operator` function performs edge detection in a grayscale image using the Prewitt operator. It calculates the gradient magnitude for each pixel in the

image, determines a threshold dynamically based on the average gradient magnitude, and produces a binary edge map highlighting edges.

```
def prewitt_operator(grayscale_image):
    Gx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
    Gy = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])

    rows, cols = grayscale_image.shape
    edge_image = np.zeros_like(grayscale_image)
    magnitudes = []

    # Compute gradient magnitudes
    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            region = grayscale_image[i - 1 : i + 2, j - 1 : j +

            gx = np.sum(Gx * region)
            gy = np.sum(Gy * region)

            magnitude = np.sqrt(gx**2 + gy**2)
            magnitudes.append(magnitude)


    threshold = np.mean(magnitudes)


    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            region = grayscale_image[i - 1 : i + 2, j - 1 : j +

            gx = np.sum(Gx * region)
            gy = np.sum(Gy * region)

            magnitude = np.sqrt(gx**2 + gy**2)
```

```
            edge_image[i, j] = 255 if magnitude > threshold else

    return edge_image.astype(np.uint8)
```

Parameters

- `grayscale_image`:A 2D NumPy array representing the input grayscale image. Pixel values should range from 0 (black) to 255 (white).

Returns

- `edge_image`:A binary image (2D NumPy array) where edges are highlighted with pixel values of 255 (white) and non-edge areas have pixel values of 0 (black).

Functionality

1. **Prewitt Kernels**:
   - The operator uses two 3×3 kernels:
     - `Gx`: For detecting horizontal edges (gradient in the x-direction).
     - `Gy`: For detecting vertical edges (gradient in the y-direction).
2. **Gradient Magnitude Calculation**:
   - For each pixel, the gradient magnitudes are computed by applying the Prewitt kernels using a sliding window approach:

$$\text{magnitude} = \sqrt{(G_x^2 + G_y^2)}$$

3. **Thresholding**:
   - A dynamic threshold is calculated by computing the average of all the gradient magnitudes across the image. If the magnitude at a pixel exceeds

this threshold, it is considered an edge (255), otherwise, it is marked as non-edge (0).

4. **Edge Detection**:
   - The output is a binary image where pixel intensities of 255 indicate edge pixels, and 0 represents non-edge pixels.

`kirsch_compass_masks` Function

Purpose

The `kirsch_compass_masks` function performs edge detection using the Kirsch operator with eight compass-direction masks (representing various orientations). It computes the maximum response from all directions for each pixel and highlights edges based on a dynamic threshold. It also outputs the direction of the detected edges.

```
def kirsch_compass_masks(grayscale_image, threshold=128):

    kirsch_masks = [
        np.array([[-3, -3, 5], [-3, 0, 5], [-3, -3, 5]]),  # Nor
        np.array([[-3, 5, 5], [-3, 0, 5], [-3, -3, -3]]),  # Nor
        np.array([[5, 5, 5], [-3, 0, -3], [-3, -3, -3]]),  # Eas
        np.array([[5, 5, -3], [5, 0, -3], [-3, -3, -3]]),  # Sou
        np.array([[5, -3, -3], [5, 0, -3], [5, -3, -3]]),  # Sou
        np.array([[-3, -3, -3], [5, 0, -3], [5, 5, -3]]),  # Sou
        np.array([[-3, -3, -3], [-3, 0, -3], [5, 5, 5]]),  # Wes
        np.array([[-3, -3, -3], [-3, 0, 5], [-3, 5, 5]]),  # Nor
    ]
```

```
    rows, cols = grayscale_image.shape
    edge_image = np.zeros_like(grayscale_image)
    direction_image = np.zeros_like(grayscale_image, dtype=int)

    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            region = grayscale_image[i - 1 : i + 2, j - 1 : j +

            responses = [np.sum(mask * region) for mask in kirs

            max_response = max(responses)
            max_direction = responses.index(max_response)

            edge_image[i, j] = 255 if max_response > threshold e
            direction_image[i, j] = max_direction

    return edge_image.astype(np.uint8), direction_image
```

Parameters

- `grayscale_image`:

  A 2D NumPy array representing the input grayscale image. Pixel values should range from 0 (black) to 255 (white).

- `threshold` (optional):

  A threshold value used to determine whether a pixel should be considered an edge. Default is 128. If the maximum response from the Kirsch masks exceeds this threshold, the pixel is marked as an edge (255); otherwise, it is marked as non-edge (0).

Returns

- `edge_image`:

A binary image (2D NumPy array) where edges are highlighted with pixel values of 255 (white) and non-edge areas have pixel values of 0 (black).

- `direction_image`:

  A 2D array of integers representing the direction of the detected edges, corresponding to one of the 8 compass directions. The values range from 0 to 7, where each number corresponds to a specific direction:

  - 0: North
  - 1: North-East
  - 2: East
  - 3: South-East
  - 4: South
  - 5: South-West
  - 6: West
  - 7: North-West

Functionality

1. **Kirsch Masks**:
   - The Kirsch operator uses eight 3×3 masks, each representing a direction (from North to North-West).
   - These masks detect edges in different orientations by convolving with the image.
2. **Gradient Calculation**:
   - For each pixel, the function calculates the response of each of the eight masks by performing element-wise multiplication and summing the results.
3. **Thresholding**:
   - The maximum response from all eight directions is compared against the given threshold. If the response exceeds the threshold, the pixel is

considered part of an edge.

4. **Edge Direction**:
    - The direction of the edge is determined by the mask that produced the maximum response. The index of this mask corresponds to one of the eight possible edge directions.
5. **Edge Image and Direction Image**:
    - The function returns two images:
        - `edge_image`: Binary image showing the detected edges.
        - `direction_image`: Image where each pixel indicates the direction of the edge.

`homogeneity_operator` Function

Purpose

The `homogeneity_operator` function is designed for edge detection by calculating the homogeneity of pixel neighborhoods in a grayscale image. The function compares each pixel's intensity with its local neighborhood, highlighting edges where there is a significant difference between the center pixel and its neighbors.

```
def homogeneity_operator(grayscale_image, threshold=128):

    rows, cols = grayscale_image.shape
    edge_image = np.zeros_like(grayscale_image)

    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            neighborhood = grayscale_image[i - 1 : i + 2, j - 1
```

```
            center_pixel = grayscale_image[i, j]

            differences = np.abs(neighborhood - center_pixel)

            max_difference = np.max(differences)

            edge_image[i, j] = 255 if max_difference > threshol

    return edge_image.astype(np.uint8)
```

Parameters

- `grayscale_image`:

  A 2D NumPy array representing the input grayscale image. Pixel values should range from 0 (black) to 255 (white).

- `threshold` (optional):

  A threshold value used to determine whether a pixel should be considered an edge. Default is 128. If the maximum difference between the center pixel and its neighbors exceeds this threshold, the pixel is considered part of an edge (255); otherwise, it is marked as non-edge (0).

Returns

- `edge_image`:A binary image (2D NumPy array) where edges are highlighted with pixel values of 255 (white) and non-edge areas have pixel values of 0 (black).

Functionality

1. **Neighborhood Comparison**:
   - For each pixel in the image (excluding the border), a 3×3 neighborhood is considered. The center pixel of this neighborhood is compared with its surrounding pixels.
2. **Difference Calculation**:

- The absolute differences between the center pixel and each of its neighbors are computed.

3. **Maximum Difference**:
   - The function finds the maximum absolute difference in the neighborhood, which reflects the strength of the edge in that region.
4. **Thresholding**:
   - If the maximum difference in the neighborhood exceeds the threshold, the pixel is considered part of an edge and is marked with a value of 255. Otherwise, the pixel is marked with 0.

`difference_operator` Function

Purpose

The `difference_operator` function detects edges in a grayscale image by calculating the absolute differences between diagonally and horizontally/vertically opposite pixels in a 3×3 neighborhood. The function identifies areas with the greatest intensity difference, which typically correspond to edges.

```
def difference_operator(grayscale_image, threshold=128):

    rows, cols = grayscale_image.shape
    edge_image = np.zeros_like(grayscale_image)

    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
```

```
            region = grayscale_image[i - 1 : i + 2, j - 1 : j +

            differences = [
                abs(region[0, 0] - region[2, 2]),
                abs(region[2, 0] - region[0, 2]),
                abs(region[1, 0] - region[1, 2]),
                abs(region[0, 1] - region[2, 1]),
            ]

            max_difference = max(differences)

            edge_image[i, j] = 255 if max_difference > threshol

    return edge_image.astype(np.uint8)
```

Parameters

- `grayscale_image`:

  A 2D NumPy array representing the input grayscale image. Pixel values should range from 0 (black) to 255 (white).

- `threshold` (optional):

  A threshold value to determine whether a pixel is part of an edge. Default is 128. If the maximum calculated difference exceeds this threshold, the pixel is considered part of an edge (255); otherwise, it is marked as a non-edge (0).

Returns

- `edge_image`:A binary image (2D NumPy array) where edges are highlighted with pixel values of 255 (white) and non-edge areas have pixel values of 0 (black).

Functionality

1. **Neighborhood Comparison**:

- For each pixel in the image (excluding the border), a 3×3 region of neighboring pixels is considered.

2. **Difference Calculation**:
   - The function calculates the absolute differences between diagonally opposite pixels (top-left and bottom-right, bottom-left and top-right), and horizontally and vertically opposite pixels (left-center and right-center, top-center and bottom-center).
3. **Max Difference**:
   - The maximum difference between the pairs of compared pixels is determined.
4. **Thresholding**:
   - If the maximum difference exceeds the threshold, the pixel is marked as an edge (255). Otherwise, it is marked as a non-edge (0).

`difference_of_gaussians` Function

Purpose

The `difference_of_gaussians` (DoG) function applies the Difference of Gaussians method to an input image, which is commonly used in image processing to enhance edges by subtracting two smoothed versions of the image. This technique is useful for edge detection and feature extraction.

```
kernel7 = np.array(
    [
        [0, 0, -1, -1, -1, 0, 0],
        [0, -2, -3, -3, -3, -2, 0],
        [-1, -3, 5, 5, 5, -3, -1],
        [-1, -3, 5, 16, 5, -3, -1],
```

```
        [-1, -3, 5, 5, 5, -3, -1],
        [0, -2, -3, -3, -3, -2, 0],
        [0, 0, -1, -1, -1, 0, 0],
    ]
)

kernel9 = np.array(
    [
        [0, 0, 0, -1, -1, -1, 0, 0, 0],
        [0, -2, -3, -3, -3, -3, -3, -2, 0],
        [0, -3, -2, -1, -1, -1, -2, -3, 0],
        [-1, -3, -1, 9, 9, 9, -1, -3, -1],
        [-1, -3, -1, 9, 19, 9, -1, -3, -1],
        [-1, -3, -1, 9, 9, 9, -1, -3, -1],
        [0, -3, -2, -1, -1, -1, -2, -3, 0],
        [0, -2, -3, -3, -3, -3, -3, -2, 0],
        [0, 0, 0, -1, -1, -1, 0, 0, 0],
    ]
)

def difference_of_guassians(image, kernel1, kernel2):

    smoothed1 = cv2.filter2D(image, -1, kernel1)
    smoothed2 = cv2.filter2D(image, -1, kernel2)


    dog_result = smoothed1 - smoothed2

    return np.clip(dog_result, 0, 255).astype(np.uint8),smoothe
```

Parameters

- `image`:

A 2D NumPy array representing the input grayscale image to which the Difference of Gaussians method will be applied.

- `kernel1`:

  A 2D NumPy array representing the first Gaussian kernel used for the first smoothing operation. The size and standard deviation of the kernel control the extent of smoothing.

- `kernel2`:

  A 2D NumPy array representing the second Gaussian kernel used for the second smoothing operation. Typically, this kernel has a different standard deviation or size than `kernel1` to detect different scales of features.

Returns

- `dog_result`:

  A 2D NumPy array representing the result of the Difference of Gaussians (DoG) applied to the input image. The result highlights edges by subtracting the two smoothed images.

- `smoothed1`:

  A 2D NumPy array representing the first smoothed image (image convolved with `kernel1`).

- `smoothed2`:

  A 2D NumPy array representing the second smoothed image (image convolved with `kernel2`).

Functionality

1. **Smoothing with Two Different Kernels**:
   - The function first applies two different Gaussian filters (kernels) to the image using `cv2.filter2D`. Each kernel smooths the image at different scales (depending on the kernel size or standard deviation).
2. **Difference of Gaussians**:

- The difference between the two smoothed images is computed (`smoothed1 - smoothed2`). This process highlights edges in the image because the difference emphasizes regions where there is a significant change in intensity between the two smoothed versions of the image.

3. **Clipping and Conversion**:
   - The result is clipped to the range [0, 255] to ensure pixel values are valid (as images use 8-bit values) and is then converted to `uint8` format for display or further processing.

`contrast_function` Function

Purpose

The `contrast_filter` function you are working on is designed to enhance the contrast of an image by combining edge detection with averaging. The steps include applying an edge mask (Laplacian filter), smoothing the image, and then calculating the contrast by dividing the edge-detected result by the smoothed image.

```
# def contrast_filter(image):
#     mask = edge_mask
#     smoothing_mask = np.ones((3,3)) / 9
#     output_edge = cv2.filter2D(image,-1,mask)
#     avg_output = cv2.filter2D(image,-1,smoothing_mask).astype(
#     avg_output += 1e-5
#     contrast_img = output_edge / avg_output
#     return contrast_img,output_edge,avg_output
```

```
# image = cv2.imread('Screenshot 2024-11-27 181534.png', cv2.IMR


# res_2,edg_out,avg_out = contrast_filter(image)


# plt.figure(figsize=(10, 5))



# plt.subplot(3, 2, 1)
# plt.title('Original Image')
# plt.imshow(image, cmap='gray')
# plt.axis('off')


# plt.subplot(3, 2, 2)
# plt.title('Contrast Filter')
# plt.imshow(res_2, cmap='gray')
# plt.axis('off')


# plt.subplot(3, 2, 3)
# plt.title('Output_Edge')
# plt.imshow(edg_out, cmap='gray')
# plt.axis('off')


# plt.subplot(3, 2, 4)
# plt.title('Avg_Output')
# plt.imshow(avg_out, cmap='gray')
# plt.axis('off')
```

```
# plt.tight_layout()
# plt.show()
```

Explanation

1. **Edge Detection with Laplacian Filter**:
   - The `edge_mask` is a Laplacian kernel, which is used to detect edges in the image. It emphasizes rapid intensity changes (edges) by subtracting the surrounding pixels from the center pixel.
2. **Smoothing**:
   - A simple averaging filter (`smoothing_mask`) is applied to the image to smooth it out. This is done to get the average intensity values of the image.
3. **Contrast Enhancement**:
   - The contrast is enhanced by dividing the result of the edge detection by the smoothed image. This helps to emphasize areas where the edges stand out compared to the overall image.

Steps Breakdown:

1. **Edge Detection**:

   The `edge_mask` uses a Laplacian kernel to detect edges. This filter subtracts the surrounding pixels from the center, highlighting areas of rapid intensity change.

2. **Smoothing**:

   The `smoothing_mask` applies a simple averaging filter (3×3 kernel) to smooth the image. This helps to obtain a blurred version of the image for the contrast calculation.

3. **Contrast Enhancement**:

   The contrast image is generated by dividing the edge detection result (`output_edge`) by the smoothed image (`avg_output`). This highlights the areas with stronger edges relative to the overall intensity.

4. **Visualization**:

   Using `matplotlib`, the function plots:

   - The original image
   - The enhanced contrast image (`Contrast Filter`)
   - The edge-detected image (`Output_Edge`)
   - The smoothed image (`Avg_Output`)

`contrast_based_edge_detection_with_edge_detector` Function

Purpose

The `contrast_based_edge_detection_with_edge_detector` function you've implemented applies edge detection based on the contrast between the smoothed image and its edges.

```
def contrast_based_edge_detection_with_edge_detector(image, thre

    smoothing_mask = (1 / 9) * np.array([[1, 1, 1], [1, 1, 1],

    smoothed_image = cv2.filter2D(image, -1,smoothing_mask)

    edge_detector_mask = np.array([[-1, 0, -1], [0, 4, 0], [-1,

    edge_response = cv2.filter2D(smoothed_image,-1, edge_detect

    edges = np.where(edge_response > threshold, 255, 0)
```

```
    return edges.astype(np.uint8)
```

Function Explanation:

1. **Smoothing the Image:**
   - You first apply a smoothing filter (average filter) to the image. The filter is a simple 3×3 kernel where each element is `1/9`, which gives the average of the surrounding pixels.
2. **Edge Detection:**
   - After smoothing, you use a Laplacian edge detector mask (`edge_detector_mask`) to highlight areas of rapid intensity changes (edges). This mask applies a convolution operation to the smoothed image to detect edges.
3. **Thresholding the Edges:**
   - The edge response is thresholded. If the response value exceeds the specified threshold, the corresponding pixel is set to 255 (white), indicating an edge. Otherwise, it is set to 0 (black), meaning no edge.
4. **Return the Result:**
   - The function returns a binary edge map, where edges are white (255) and non-edges are black (0).

Explanation of Each Part:

1. **Smoothing with `smoothing_mask`:**

   The `smoothing_mask` is a 3×3 averaging filter that blurs the image. This helps in reducing noise and unnecessary details, which allows the edge detector to focus on the major edges in the image.

2. **Edge Detection with `edge_detector_mask`:**

   The `edge_detector_mask` is a Laplacian kernel that highlights edges in the image. It computes the second derivative of the image and emphasizes areas where intensity changes rapidly.

3. **Thresholding:**
   After obtaining the edge response, you apply a threshold (`threshold=15` by default) to determine which pixels should be considered as part of the edges. If the edge response is greater than the threshold, the pixel is classified as an edge.

`variance_mask` Function

Purpose

The `variance_mask` function you've implemented calculates the variance of pixel intensities in a local neighborhood for each pixel in the image. Variance is a measure of how spread out the pixel values are in a local region, and it can help detect edges or regions of high variation in an image.

Explanation of the `variance_mask` Function:

1. **Neighborhood Calculation:**
   - For each pixel in the image (except the borders), the function calculates the variance of a 3×3 neighborhood around the pixel.
2. **Mean Calculation:**
   - For each 3×3 neighborhood, the mean pixel intensity is calculated.
3. **Variance Calculation:**
   - The variance is calculated as the average of squared differences between each pixel in the neighborhood and the mean.
4. **Return Output:**

- The output is an image where each pixel represents the variance of its neighborhood in the original image.

```
# def variance_mask(image):
#     output_image = np.zeros_like(image)
#     h, w = image.shape
#     for i in range(1, h-1):
#         for j in range(1, w-1):
#             neighborhood = image[i-1:i+2, j-1:j+2]
#             mean = np.mean(neighborhood)
#             var = np.sum((neighborhood - mean) ** 2) / 9
#             output_image[i, j] = var
#     return output_image

# image = cv2.imread("./Screenshot 2024-11-27 181534.png", cv2.I


# var_img = variance_mask(image)


# plt.figure(figsize=(10, 5))



# plt.subplot(1, 2, 1)
# plt.title('Original Image')
# plt.imshow(image, cmap='gray')
# plt.axis('off')


# plt.subplot(1, 2, 2)
# plt.title('Variance Image')
# plt.imshow(var_img, cmap='gray')
# plt.axis('off')
```

```
# plt.tight_layout()
# plt.show()
```

Improvements:

1. **Data Type for Output Image:**
   - The output image is now initialized with `dtype=np.float32` to ensure that we store the floating-point values from the variance calculation properly, which might be important for subsequent processing steps.
2. **Efficiency in Variance Calculation:**
   - Instead of using `np.sum()` for variance, we directly use `np.mean()` for the squared differences, which can be a bit more efficient.

Result:

The output will be two images:

1. **Original Image:** The grayscale image you loaded.
2. **Variance Image:** An image where each pixel represents the variance of a 3×3 neighborhood in the original image. High variance regions (edges or textured areas) will appear brighter, while low variance regions (smooth areas) will appear darker.

`variance_edge_detector` Function

Purpose:

`variance_edge_detector` function implements a simple edge detection method based on the variance of pixel intensities within a local region around each pixel. The idea is that areas with higher variance (such as edges or textured regions) will be

highlighted, while smooth areas will be suppressed.

```
## For GUI
def variance_edge_detector(image, region_size=3, threshold=50):
    pad = region_size // 2

    padded_image = np.pad(image, pad, mode="constant", constant_

    rows, cols = image.shape
    edge_image = np.zeros_like(image)

    for i in range(pad, rows + pad):
        for j in range(pad, cols + pad):

            region = padded_image[i - pad : i + pad + 1, j - pad

            mean_intensity = np.mean(region)

            variance = np.mean((region - mean_intensity) ** 2)

            edge_image[i - pad, j - pad] = 255 if variance > thr

    return edge_image.astype(np.uint8)
```

Explanation of the `variance_edge_detector` Function:

1. **Padding the Image:**
    - The image is padded to handle border pixels. The padding size is calculated as half the size of the region (`region_size // 2`).
2. **Iterating over the Image:**
    - The function loops over each pixel in the image (after padding) and extracts a local region of size `region_size x region_size` around the current

pixel.

3. **Variance Calculation:**
    - For each region, the mean intensity is calculated, and then the variance of the pixel intensities within the region is computed.
4. **Edge Detection:**
    - If the variance within the region exceeds the specified `threshold`, the corresponding pixel in the `edge_image` is set to 255 (indicating an edge), otherwise, it is set to 0.

`range_mask` Function

Purpose

The `range_mask` function computes the "range" (the difference between the maximum and minimum pixel intensities) within a local neighborhood of each pixel in the image. This is useful for edge detection, as areas with higher intensity variation (such as edges or textured regions) will have a higher range value.

```
# def range_mask(image):
#     output_image = np.zeros_like(image)
#     h, w = image.shape
#     for i in range(1, h-1):
#         for j in range(1, w-1):
#             neighborhood = image[i-1:i+2, j-1:j+2]
#             range_v = np.max(neighborhood) - np.min(neighborh
#             output_image[i,j] = range_v
#     return output_image
```

```
# image = cv2.imread("./Screenshot 2024-11-27 181534.png", cv2.I


# range_img = range_mask(image)


# plt.figure(figsize=(10, 5))



# plt.subplot(1, 2, 1)
# plt.title('Original Image')
# plt.imshow(image, cmap='gray')
# plt.axis('off')


# plt.subplot(1, 2, 2)
# plt.title('Range Image')
# plt.imshow(range_img, cmap='gray')
# plt.axis('off')



# plt.tight_layout()
# plt.show()
```

Explanation of the `range_mask` Function:

1. **Neighborhood Extraction:**

- For each pixel in the image (excluding the borders), a 3×3 neighborhood is extracted around the pixel.

2. **Range Calculation:**
   - The "range" of the neighborhood is calculated as the difference between the maximum and minimum pixel values within the neighborhood. This is a measure of intensity variation in the local region.
3. **Resulting Mask:**
   - The resulting value (range) is assigned to the corresponding pixel in the `output_image`. This produces an image that highlights regions with high intensity variation.

`range_mask` Function

Purpose:

The `range_edge_detector` function detects edges based on the intensity range of pixel neighborhoods. The method works by calculating the range (the difference between the maximum and minimum pixel values) within a region around each pixel. If the range exceeds a specified threshold, the pixel is marked as an edge (255); otherwise, it is set to 0.

```
# For GUI
def range_edge_detector(image, region_size=3, threshold=50):
    pad = region_size // 2

    padded_image = np.pad(image, pad, mode="constant", constant_
```

```
    rows, cols = image.shape
    edge_image = np.zeros_like(image)

    for i in range(pad, rows + pad):
        for j in range(pad, cols + pad):

            region = padded_image[i - pad : i + pad + 1, j - pad

            intensity_range = np.max(region) - np.min(region)

            edge_image[i - pad, j - pad] = 255 if intensity_rang

    return edge_image.astype(np.uint8)
```

Explanation of the `range_edge_detector` Function:

1. **Padding:**
   - To avoid issues with borders, the image is padded by a specified region size. This ensures that neighborhoods can be extracted from the edges of the image.
2. **Neighborhood Extraction:**
   - For each pixel (excluding the padded borders), a neighborhood region of the specified size is extracted.
3. **Intensity Range Calculation:**
   - The range of pixel intensities within the neighborhood is calculated. The range is the difference between the maximum and minimum pixel values in the neighborhood.
4. **Edge Detection:**
   - If the intensity range exceeds the threshold, the pixel is marked as part of an edge (255); otherwise, it is set to 0.

Explanation of Key Parameters:

- `region_size`: The size of the neighborhood used to calculate the intensity range. A 3×3 region is the default, but you can adjust it to capture larger neighborhoods.
- `threshold`: The minimum intensity range required to classify a pixel as an edge. You can adjust this threshold depending on the image and how sensitive you want the edge detection to be.
- `padded_image`: Padding is applied to the image to ensure that neighborhood regions can be extracted even for border pixels.

Visualization of Results:

1. **Original Image**: Shows the grayscale version of the image.
2. **Edge Image**: Displays the output of the edge detection, where edges are highlighted as white pixels (255) and the background is black (0).

`high_pass_filter` Function

Purpose
The
`high_pass_filter` function is designed to enhance the high-frequency components of an image, which can help sharpen edges and highlight fine details.

```
def high_pass_filter(image):

    high_pass_mask = np.array([[0, -1, 0], [-1, 5, -1], [0, -1,


    filtered_image = cv2.filter2D(image, -1,high_pass_mask)

    filtered_image = np.clip(filtered_image, 0, 255)
```

```
    return filtered_image.astype(np.uint8)
```

Explanation:

1. **High-Pass Mask:**
   - The `high_pass_mask` used here is a **sharpening filter**. It highlights rapid changes in pixel intensity (edges) by subtracting the surrounding pixels' values from the center pixel.
   - The kernel is a 3×3 matrix with a center value of 5 and surrounding values of -1, which emphasizes edges and reduces the lower-frequency components (smooth areas).
2. **Filter Application:**
   - The `cv2.filter2D()` function is used to apply the mask (filter) to the image. This function performs a convolution operation between the image and the filter, modifying the pixel values based on the surrounding pixels.
3. **Clipping Values:**
   - After filtering, the pixel values may go beyond the valid range of 0 to 255 (due to convolution). The `np.clip()` function ensures that the values stay within this range.
4. **Return Type:**
   - The output is cast to `np.uint8` to ensure it is in the correct data type for image display and further processing.

`low_pass_filter` Function

Purpose

The `low_pass_filter` function is designed to apply a smoothing or blurring effect to an image by using a **low-pass filter**. This type of filter reduces high-frequency noise

and sharp edges, producing a smoother result.

```
def low_pass_filter(image):

    low_pass_mask = (1 / 6) * np.array([[0, 1, 0], [1, 2, 1], [0

#     filtered_image = manual_convolution(image, low_pass_mask)
    filtered_image = cv2.filter2D(image, -1,low_pass_mask)

    filtered_image = np.clip(filtered_image, 0, 255)

    return filtered_image.astype(np.uint8)
```

Explanation:

1. **Low-Pass Mask:**
    - The `low_pass_mask` here is a **Gaussian-like kernel** designed to blur the image. It's a weighted average filter, where the center value has the highest weight, and the surrounding values decrease in weight.
    - The kernel you've defined is:

      ```
      Copy code
      [[0, 1, 0],
       [1, 2, 1],
       [0, 1, 0]]
      ```

        - This kernel performs a smoothing operation by averaging pixel values in a neighborhood, giving more weight to the central pixel and less to the surrounding pixels.

    - The total weight sums to 6, and by multiplying by `1/6`, the average operation is normalized.

2. **Filter Application:**
    - The function `cv2.filter2D()` applies the kernel (filter) to the image using **convolution**. This means the filter is moved across the image, and for each region, the weighted average of the neighborhood pixels is calculated.
3. **Clipping Values:**
    - After the filtering operation, the pixel values may exceed the valid range of `[0, 255]` due to the convolution operation. The `np.clip()` function ensures that all values are within this range.
4. **Return Type:**
    - The function returns the filtered image with pixel values cast to `np.uint8`, which is the standard data type for image arrays.

Key Features of the Low-Pass Filter:

1. **Smoothing/Blurring Effect:**
    - This filter blurs the image by averaging pixel values in a neighborhood, effectively reducing high-frequency noise and sharp transitions between pixel intensities.
2. **Kernel Characteristics:**
    - The center pixel has the largest weight (2), and the surrounding pixels have a weight of 1, which gives more importance to the central pixel when calculating the output.

`median_filter` Function

Purpose

he

`median_filter` function you've implemented is designed to apply a **median filter** to an image. The median filter is commonly used for noise reduction, particularly in removing **salt-and-pepper noise** from images.

```
def median_filter(image, region_size=3):

    pad = region_size // 2

    padded_image = np.pad(image, pad, mode="constant", constant_

    rows, cols = image.shape
    filtered_image = np.zeros_like(image)

    for i in range(pad, rows + pad):
        for j in range(pad, cols + pad):

            region = padded_image[i - pad : i + pad + 1, j - pad

            median_value = np.median(region)

            filtered_image[i - pad, j - pad] = median_value

    return filtered_image.astype(np.uint8)
```

Explanation:

1. **Padding:**
    - The `region_size` parameter defines the size of the neighborhood or window used for filtering (typically a square). The function pads the image with zeros on all sides to handle border pixels.

- Padding is done by `np.pad(image, pad, mode="constant", constant_values=0)` where `pad` is half the region size.

2. **Sliding Window:**
   - The function iterates over every pixel in the image. For each pixel, it extracts a **square region** of size `region_size x region_size`, centered around the pixel.
   - For every region, the function computes the **median** of all pixel values in that region.
3. **Median Calculation:**
   - `np.median(region)` computes the median value of the region (the central value when the values are sorted).
   - The pixel value at the center of the region is replaced by this median value, which helps in removing outliers (like noise).
4. **Return Type:**
   - The resulting filtered image is returned after casting it to `np.uint8` (standard 8-bit unsigned integer format for image data).

`add_image(img1,img2)` Function

```
# def add_images(img1, img2):

#     # Ensure both images are of the same size by resizing img2
#     img2_resized = cv2.resize(img2, (img1.shape[1], img1.shape

#     height, width = img1.shape
#     new_img = np.zeros((height, width), dtype=np.uint8)

#     for i in range(height):
```

```
#         for j in range(width):
#             # Add pixel values and clamp between 0 and 255
#             new_img[i, j] = max(0, min(img1[i, j] + img2_resiz

#     return new_img


# img1 = cv2.imread(IMAGE_PATH, cv2.IMREAD_GRAYSCALE)
# img2 = cv2.imread(IMAGE_PATH_2, cv2.IMREAD_GRAYSCALE)


# if img1 is None or img2 is None:
#     raise FileNotFoundError("One or both image files could not


# result_image = add_images(img1, img2)

# plt.figure(figsize=(15, 5))


# plt.subplot(1, 3, 1)
# plt.title('Image 1')
# plt.imshow(img1, cmap='gray')
# plt.axis('off')


# plt.subplot(1, 3, 2)
# plt.title('Image 2')
# plt.imshow(img2, cmap='gray')
# plt.axis('off')


# plt.subplot(1, 3, 3)
# plt.title('Added Image')
# plt.imshow(result_image, cmap='gray')
# plt.axis('off')
```

```
# plt.tight_layout()
# plt.show()
```

Function Explanation:

1. **Resizing:**
   - It ensures that both images (`img1` and `img2`) have the same dimensions by resizing `img2` to match the size of `img1`. This is important because you can only perform pixel-wise operations on images of the same size.
2. **Pixel-wise Addition:**
   - The function iterates over each pixel of the images, adds the corresponding pixel values from both images, and ensures that the sum is clipped to be between 0 and 255 (the valid range for image pixel values).
3. **Handling Overflow:**
   - The pixel values are clamped between `0` and `255` using `max(0, min(value, 255))` to avoid overflow, which can occur if the sum exceeds the maximum value (255 for 8-bit images).
4. **Return:**
   - The resulting image is returned after processing all pixels.

```
def add_imagess(image):
    image_copy = image.copy()
    added_image = image + image_copy
    return np.clip(added_image, 0, 255).astype(np.uint8)


def subtract_imagess(image):
    image_copy = image.copy()
    subtracted_image = image_copy - image
```

```
    return np.clip(subtracted_image, 0, 255).astype(np.uint8)


def invert_image(image):
    invert_image = 255 - image
    return invert_image.astype(np.uint8)
```

1. `add_imagess(image)`:

This function adds an image to its copy (i.e., `image + image_copy`), which is essentially doubling the pixel intensity values.

- **What it does**:
  - The image is copied using `image.copy()`.
  - The original image and its copy are added together, and pixel values are clamped between 0 and 255 using `np.clip` to avoid overflow.
- **Issues**:
  - The pixel values can become larger than 255 after addition. Clamping the values using `np.clip(added_image, 0, 255)` ensures that the output image maintains valid pixel values.

2. `subtract_imagess(image)`:

This function subtracts the original image from its copy (i.e., `image_copy - image`), which might highlight differences between the two images.

- **What it does**:
  - The function copies the image and subtracts the original image from its copy. This will highlight areas of contrast where the image has non-zero values.
  - `np.clip` ensures that negative values are set to 0 and values greater than 255 are clamped.

3. `invert_image(image)`:

This function inverts the image by subtracting each pixel value from 255.

- **What it does**:
  - The inversion is performed using `255 - image`, where each pixel value is flipped to its opposite in the 8-bit color scale.
  - This effectively converts black to white, white to black, and inverts all other pixel values in the image.

Explanation:

- **add_imagess**: Adds an image to its copy and clips the result to the range `[0, 255]` to avoid overflows.
- **subtract_imagess**: Subtracts the original image from its copy, with clipping to ensure pixel values stay within valid bounds.
- **invert_image**: Inverts the pixel values by subtracting them from 255.

`manual_histogram_segmentation` Function

Purpose:

The function segments an image based on manually provided intensity thresholds. This is often used to isolate specific regions of interest in an image based on pixel intensity levels. The image is divided into multiple segments, where each segment corresponds to a specific range of intensity values.

```
def manual_histogram_segmentation(image, thresholds):

    thresholds = sorted(thresholds)
    segmented_image = np.zeros_like(image)

    for i, threshold in enumerate(thresholds):
        segmented_image[image <= threshold] = (i + 1) * 50
        image = np.where(image > threshold, image, 0)
```

```
        segmented_image[image > 0] = (len(thresholds) + 1) * 50

        return segmented_image
```

Parameters:

1. `image` **(numpy.ndarray):**
    - Type: `numpy.ndarray`
    - Description: The grayscale image that is to be segmented. It should be a 2D array where pixel values range from 0 to 255.
2. `thresholds` **(list or array-like):**
    - Type: `list` or `array-like`
    - Description: A list of intensity thresholds used to segment the image. Each threshold value defines a cut-off point between different intensity levels. The function processes these thresholds in ascending order to create segments.
    - Example: `[100, 150, 200]` would segment the image into four regions, one for pixels with intensity values ≤ 100, one for pixels with intensity values between 100 and 150, one for pixels between 150 and 200, and one for pixels > 200.

Returns:

- `segmented_image` **(numpy.ndarray):**
    - Type: `numpy.ndarray`
    - Description: The segmented image where pixel values represent different segments. Each segment is assigned a unique intensity value, which is calculated by the index of the threshold.
    - The final `segmented_image` contains pixel values representing the intensity levels that correspond to the thresholds defined.

Functionality:

1. The function starts by sorting the given `thresholds` in ascending order. This ensures that the segmentation occurs in a logical order based on intensity values.
2. A new image (`segmented_image`) is created with the same shape as the input image, initialized to zeros (i.e., a blank image).
3. For each threshold in the sorted list:
    - All pixels in the original image that are less than or equal to the current threshold are assigned a segment value. The segment value is calculated as `(i + 1) * 50`, where `i` is the index of the threshold. This ensures each segment has a unique value.
    - Pixels greater than the current threshold are retained for further processing with subsequent thresholds.
4. After processing all thresholds, pixels that are above the final threshold are assigned a special segment value, calculated as `(len(thresholds) + 1) * 50`.

Function: `histogram_peak_thresholding(image)`

Purpose:

This function performs **histogram-based thresholding** by identifying two peaks in the histogram (background and object), then calculates a threshold based on the average of these peaks. The image is then thresholded into a binary image using the calculated threshold. This method is commonly used for image segmentation, where you want to separate foreground and background.

```
def histogram_peak_thresholding(image):


    histogram = np.zeros(256, dtype=int)
```

```
    for pixel in image.ravel():  # Flatten the image into a 1D a
        histogram[pixel] += 1


    peaks = []
    for i in range(1, len(histogram) - 1):
        if histogram[i] > histogram[i - 1] and histogram[i] > hi
            peaks.append(i)


    peaks = sorted(peaks, key=lambda x: histogram[x], reverse=Tr


    if len(peaks) < 2:
        raise ValueError("Not enough peaks found in histogram!")
    background_peak = peaks[0]
    object_peak = peaks[1]


    threshold = (background_peak + object_peak) // 2


    binary_image = ((image > threshold) * 255).astype(np.uint8)

    return binary_image, threshold, histogram, background_peak,


grayscale_image = cv2.imread(IMAGE_PATH, cv2.IMREAD_GRAYSCALE)


thresholded_img, threshold, histogram, bg_peak, obj_peak = histo


plt.figure(figsize=(15, 5))
```

```
plt.subplot(1, 3, 1)
plt.title("Original Image")
plt.imshow(grayscale_image, cmap="gray")


plt.subplot(1, 3, 2)
plt.title("Histogram with Peaks")
plt.plot(histogram, color="black")
plt.axvline(x=bg_peak, color="blue", linestyle="--", label=f"Ba
plt.axvline(x=obj_peak, color="red", linestyle="--", label=f"Ob
plt.axvline(x=threshold, color="green", linestyle="--", label=f'
plt.legend()


plt.subplot(1, 3, 3)
plt.title(f"Thresholded Image (T={threshold})")
plt.imshow(thresholded_img, cmap="gray")

plt.show()
```

Parameters:

1. `image` **(numpy.ndarray):**
    - **Type**: `numpy.ndarray`
    - **Description**: The grayscale image to be thresholded. It should be a 2D array where pixel values range from 0 to 255. The function expects an image in grayscale format (single-channel).

Returns:

- `binary_image` **(numpy.ndarray):**
    - **Type**: `numpy.ndarray`
    - **Description**: A binary image where pixels are either 0 (background) or 255 (object) based on the threshold.

- `threshold` **(int):**
  - **Type**: `int`
  - **Description**: The calculated threshold value used for binary thresholding.
- `histogram` **(numpy.ndarray):**
  - **Type**: `numpy.ndarray`
  - **Description**: The histogram of pixel intensities in the input image. It shows the frequency of each pixel intensity value from 0 to 255.
- `background_peak` **(int):**
  - **Type**: `int`
  - **Description**: The intensity value corresponding to the background peak in the histogram.
- `object_peak` **(int):**
  - **Type**: `int`
  - **Description**: The intensity value corresponding to the object peak in the histogram.

How It Works:

1. **Histogram Calculation:**
   - The function starts by calculating the histogram of the input image, which represents the frequency of pixel intensities.
2. **Peak Detection:**
   - The algorithm searches for local maxima (peaks) in the histogram. A peak is identified if the pixel intensity is greater than both its neighboring intensity values.
3. **Sorting Peaks:**
   - The peaks are sorted by their frequency, and the two most significant peaks are selected as the background and object peaks.
4. **Threshold Calculation:**

- The threshold is calculated as the average of the two peaks (`background_peak` and `object_peak`).

5. **Binary Thresholding:**
   - The input image is then thresholded: pixels greater than the threshold are set to 255 (object), and pixels less than or equal to the threshold are set to 0 (background).
6. **Output:**
   - The function returns the binary thresholded image, the threshold value, the histogram, and the background and object peaks.

Function: `manual_peak_detection(histogram)`

Purpose:

This function identifies the **local maxima (peaks)** in a histogram, which are the points where the intensity value is higher than its neighboring values. Peaks are important for image segmentation, where identifying regions with significant differences in intensity can aid in separating objects from the background.

```
def manual_peak_detection(histogram):
    peaks = []

    for i in range(1, len(histogram) - 1):
        if histogram[i] > histogram[i - 1] and histogram[i] > hi
            peaks.append(i)

    return peaks
def valley_based_segmentation(image):
```

```
    histogram = calculate_histogram(image)

    all_peaks = manual_peak_detection(histogram)

    prominent_peaks = sorted(all_peaks, key=lambda x: histogram

    if len(prominent_peaks) < 2:
        raise ValueError("Not enough peaks detected to calculate
    peak1, peak2 = prominent_peaks[:2]

    start, end = min(peak1, peak2), max(peak1, peak2)
    print(f"Selected Peaks -> Peak 1: {peak1}, Peak 2: {peak2},

    valley_range = histogram[start:end + 1]
    if len(valley_range) == 0:
        raise ValueError("Invalid valley range. Check peaks and
    valley = np.argmin(valley_range) + start
    print(f"Valley Detected: {valley}")

    segmented_image = np.zeros_like(image)
    segmented_image[image <= valley] = 50
    segmented_image[image > valley] = 255

    return segmented_image, histogram, (peak1, peak2), valley
```

Parameters:

1. `histogram` **(numpy.ndarray):**
    - **Type**: `numpy.ndarray`
    - **Description**: The input histogram, which is an array representing the frequency of pixel intensities in an image. The histogram values correspond to the intensity levels (0–255).

Returns:

- `peaks` **(list of int):**

- **Type**: `list`
- **Description**: A list containing the indices of the peaks in the histogram. Peaks are detected where the intensity value is greater than both its neighboring values.

How It Works:

1. The function iterates over the histogram (excluding the first and last values).
2. It checks for peaks by comparing each intensity value with its neighbors (previous and next values).
3. If an intensity value is greater than both neighbors, it is identified as a peak, and its index is added to the list.

Function: `valley_based_segmentation(image)`

Purpose:

This function performs **valley-based segmentation** of an image using the histogram. It identifies two prominent peaks in the histogram and finds the valley between them. The valley is then used to segment the image into foreground and background.

Parameters:

1. `image` **(numpy.ndarray):**
   - **Type**: `numpy.ndarray`
   - **Description**: The grayscale image to be segmented. It should be a 2D array where pixel values range from 0 to 255.

Returns:

- `segmented_image` **(numpy.ndarray):**
  - **Type**: `numpy.ndarray`
  - **Description**: The segmented binary image where pixels are assigned values of 50 for the background (pixels ≤ valley) and 255 for the foreground (pixels > valley).

- `histogram` **(numpy.ndarray):**
  - **Type**: `numpy.ndarray`
  - **Description**: The histogram of pixel intensities in the input image.
- `peaks` **(tuple of int):**
  - **Type**: `tuple`
  - **Description**: A tuple containing the indices of the two prominent peaks identified in the histogram.
- `valley` **(int):**
  - **Type**: `int`
  - **Description**: The intensity value at the valley point (local minimum) between the two peaks.

How It Works:

1. **Calculate Histogram:** The histogram of pixel intensities is computed for the input image.
2. **Peak Detection:** The function detects peaks in the histogram using the `manual_peak_detection` function. It sorts the peaks by their frequency and selects the two most prominent peaks.
3. **Valley Detection:** The valley is found by identifying the local minimum between the two peaks.
4. **Segmentation:** The image is segmented based on the valley:
   - Pixels with intensities less than or equal to the valley are assigned a value of 50 (background).
   - Pixels with intensities greater than the valley are assigned a value of 255 (foreground).

Function: `adaptive_histogram_segmentation(image)`

Purpose:

This function performs **adaptive histogram-based image segmentation** by first detecting prominent peaks in the image's histogram, identifying a valley between two peaks, and using this valley to segment the image into foreground and background. It then refines the segmentation by calculating the mean pixel intensities of the background and object regions, and adjusts the threshold for better segmentation.

```
def adaptive_histogram_segmentation(image):


    histogram = calculate_histogram(image)


    all_peaks = manual_peak_detection(histogram)


    prominent_peaks = sorted(all_peaks, key=lambda x: histogram

    if len(prominent_peaks) < 2:
        raise ValueError("Not enough peaks detected for adaptive

    peak1, peak2 = prominent_peaks[:2]


    start, end = min(peak1, peak2), max(peak1, peak2)
    valley = np.argmin(histogram[start:end + 1]) + start


    segmented_image = np.zeros_like(image)
    segmented_image[image <= valley] = 50  # Background
    segmented_image[image > valley] = 255  # Object
```

```
    background_mean = np.mean(image[segmented_image == 50])
    object_mean = np.mean(image[segmented_image == 255])


    new_threshold = (background_mean + object_mean) // 2


    final_segmented_image = np.zeros_like(image)
    final_segmented_image[image <= new_threshold] = 50  # Backgr
    final_segmented_image[image > new_threshold] = 255  # Object

    return {
        "final_segmented_image": final_segmented_image,
        "histogram": histogram,
        "all_peaks": all_peaks,
        "prominent_peaks": (peak1, peak2),
        "valley": valley,
        "background_mean": background_mean,
        "object_mean": object_mean,
        "new_threshold": new_threshold,
    }
```

Parameters:

1. `image` **(numpy.ndarray):**
    - **Type**: `numpy.ndarray`
    - **Description**: The input image, expected to be a 2D grayscale image (pixel values range from 0 to 255).

Returns:

- **A dictionary containing the following:**
    - `final_segmented_image` **(numpy.ndarray):**
        - **Type**: `numpy.ndarray`

- **Description**: The final segmented image where pixels are assigned values of 50 for the background and 255 for the object based on the new threshold.

- `histogram` **(numpy.ndarray):**
  - **Type**: `numpy.ndarray`
  - **Description**: The histogram of pixel intensities in the input image.
- `all_peaks` **(list):**
  - **Type**: `list`
  - **Description**: A list of all detected peaks in the histogram.
- `prominent_peaks` **(tuple):**
  - **Type**: `tuple`
  - **Description**: A tuple containing the indices of the two most prominent peaks in the histogram.
- `valley` **(int):**
  - **Type**: `int`
  - **Description**: The intensity value at the valley point between the two peaks.
- `background_mean` **(float):**
  - **Type**: `float`
  - **Description**: The mean pixel value of the background region (pixels with intensity less than or equal to the valley).
- `object_mean` **(float):**
  - **Type**: `float`
  - **Description**: The mean pixel value of the object region (pixels with intensity greater than the valley).
- `new_threshold` **(int):**
  - **Type**: `int`

- **Description**: The new threshold value, calculated as the mean of the background and object mean values.

How It Works:

1. **Histogram Calculation:** The function first computes the histogram of pixel intensities for the input image.
2. **Peak Detection:** It uses the `manual_peak_detection` function to find all the peaks in the histogram.
3. **Prominent Peaks:** The peaks are sorted, and the two most prominent peaks are selected.
4. **Valley Detection:** The function identifies the valley (local minimum) between the two prominent peaks.
5. **Initial Segmentation:** The image is segmented using the detected valley, assigning pixels less than or equal to the valley as background (50), and pixels greater than the valley as the object (255).
6. **Refinement with Adaptive Thresholding:**
   - The mean pixel intensity values of the background and object regions are calculated.
   - A new threshold is computed as the average of these two mean values.
7. **Final Segmentation:** The image is re-segmented using the new adaptive threshold.