Image Processing Project Documentation

➢ Introduction

This documentation provides an in-depth exploration of various image processing techniques, focusing on their implementation (How it works), advantages, disadvantages, and results for each algorithm

➢ The Algorithms that we will Cover:

1. Simple Pre-Processing :

-Simple Halftone
-Advanced Halftone
-Histogram
-Histogram Equalization

2. Basic Edge Detection Techniques:

-Sobel Operator
- Prewit Operator
-Kirsch Operator

3. Image Operations:

-Addition
-Subtraction
-Cut and Paste
-Invert

4. All Spatial Frequency Filters:

-High Pass Filter
-Low Pass Filter
-Median Filter

5. Advanced Edge Detection Techniques:

-Homogeneity Operator
-Difference Operator
-Range Operator
-Variance Operator
-Contrast Based
-Difference of Gaussians

6. Segmentation:

-Manual Segmentation
-Peak Segmentation
-Valley Segmentation
-Adaptive Segmentation

Simple Halftone

How it works

- Convert the image to grayscale if it is not already in the 'L' mode
- Convert the grayscale image to a NumPy array for easy manipulation
- Initialize a new array to store the halftoned image
- Loop through each pixel in the image:
  - If the pixel's intensity is less than 128, set the pixel value to 0 (black)
  - If the pixel's intensity is greater than or equal to 128, set the pixel value to 255 (white)
- Return the processed image as a new image from the halftoned array

Advantages

- Simple and fast: The thresholding technique is computationally inexpensive, making it suitable for real-time applications
- Effective for high-contrast images: Halftoning with a fixed threshold works well when the image has clear light and dark regions, making it easy to distinguish between them
- Reduction in data: By converting an image to a binary representation, the image size can be significantly reduced, which may be useful for certain applications like printing or data compression

Disadvantages

- Loss of detail: This method reduces the image to two colors (black and white), causing a loss of subtle gradients and image details
- Ineffective for complex images: Images with fine details or complex textures may not be well represented, leading to a "blocky" or overly simplistic appearance
- Fixed threshold limitation: A static threshold of 128 may not work well for all images, as lighting conditions or image content may require a dynamic threshold for better halftoning results

Result:

Simple Halftoning Result

Original Image

Halftoned Image

Advanced Halftone

How it works:

- The image is first converted to grayscale ('L' mode) if it is not already in that mode
- The image is then converted into a numpy array for easier manipulation of pixel values
- The algorithm iterates through each pixel of the image, comparing the pixel value to a predefined threshold (default is 128)
- If the pixel value is greater than or equal to the threshold, it is set to 255 (white); otherwise, it is set to 0 (black)
- The difference (error) between the original pixel value and the new pixel value is calculated
- The error is then distributed to neighboring pixels using the error diffusion method:
    - The pixel to the right receives 7/16 of the error
    - The pixel diagonally to the lower-left receives 3/16 of the error
    - The pixel directly below receives 5/16 of the error
    - The pixel diagonally to the lower-right receives 1/16 of the error
- After processing all pixels, the image array values are clipped to ensure they are within the 0-255 range
- Finally, the processed image array is converted back into a PIL image

Advantages:

- Preserves image details: By diffusing the error, this method can maintain more details in the image compared to simple thresholding
- Produces visually appealing results: The error diffusion technique reduces the harshness of halftoning and can create a more natural-looking result
- Effective for low-resolution displays: This technique can be especially useful for printing or displaying on devices with limited color resolution, such as printers or older screens

Disadvantages:

- Computationally expensive: The algorithm requires processing each pixel and distributing the error to multiple neighboring pixels, which can be time-consuming, especially for large images
- May introduce noise: The error diffusion can sometimes result in visible patterns or noise, especially in areas of the image with large uniform regions
- Limited to binary images: The output is always black and white, so it may not be suitable for applications requiring grayscale or color images

Result:


Advanced Halftoning Result
Original Image
Halftoned Image

Histogram

How it works

- The function begins by checking if the image is in grayscale ('L' mode). If not, it converts the image to grayscale
- A histogram is initialized as a list of 256 zeros, one for each possible pixel value in a grayscale image (0-255)
- The image is converted into a NumPy array for easier manipulation.
- The function iterates through each pixel in the flattened image array and increments the corresponding value in the histogram
- The histogram (pixel intensity counts) and the bin range (0-255) are then returned.

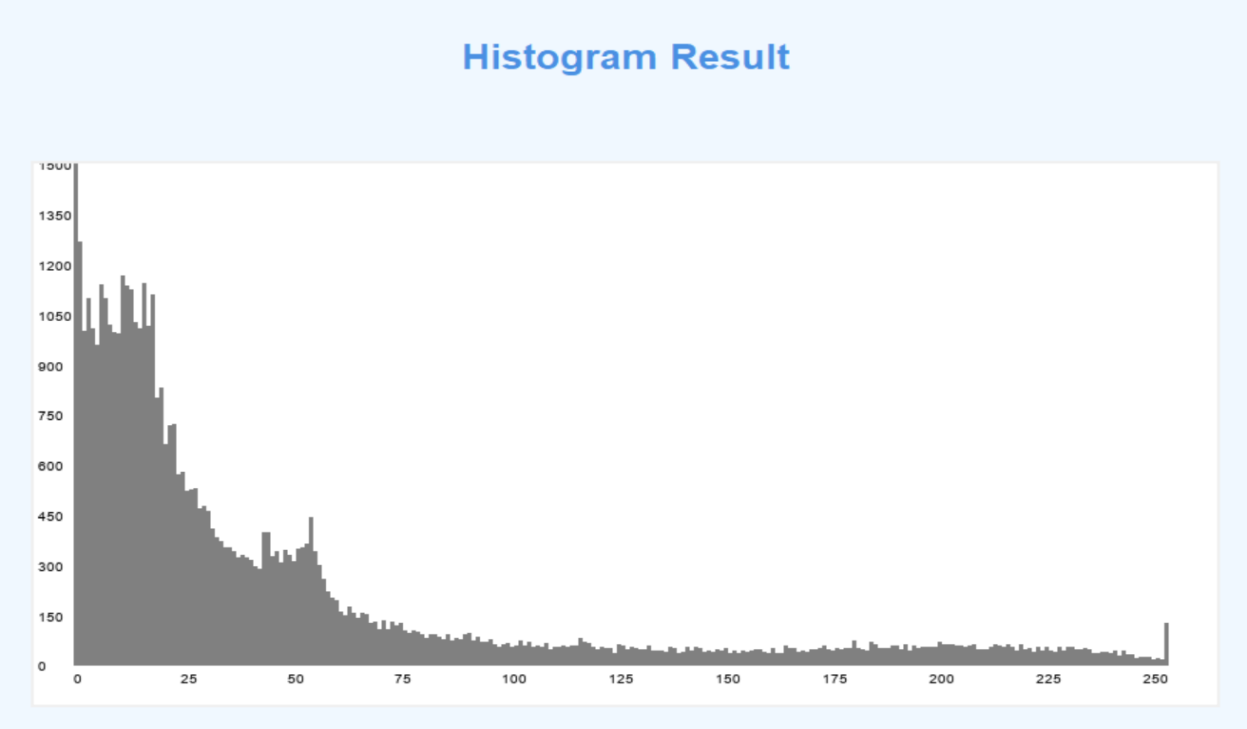
Advantages

- Simple and easy to implement for basic grayscale image histogram calculation.
- Efficient for small to medium-sized images
- Provides a quick understanding of pixel intensity distribution in an image, which can be used for further image analysis or enhancement

Disadvantages

- The function assumes that the image is grayscale, which may not be applicable for color images without additional handling
- Computationally inefficient for very large images, as it processes every pixel individually
- Does not account for other image formats or color models (e.g., RGB), limiting its versatility
- May require additional memory for large images due to the need to convert them into a NumPy array

Result:


Histogram Result
1350
1200
1050
900
750
600
450
300
150
0
0
25
50
75
100
125
150
175
200
225
250

Histogram Equalization

How it works:

- First, the image is checked to ensure it is in grayscale ('L') mode. If not, it is converted to grayscale
- The histogram of the image is calculated, which provides the frequency of each pixel intensity.
- A cumulative distribution function (CDF) is computed from the histogram. This CDF is normalized to fit within the range of 0 to 255
- The image array is flattened, and each pixel value is mapped using the normalized CDF to perform the equalization
- The equalized array is reshaped back to the original image dimensions, and a new image is created from the array
- The histogram of the equalized image is calculated and returned alongside the equalized image

Advantages:

- Enhances the contrast of the image, especially in cases where the original image has poor contrast
- Makes details in dark or bright areas more visible, improving the overall quality
- The method is simple and computationally efficient
- Works well for improving the visibility of features in grayscale images

Disadvantages:

- May result in over-enhancement, introducing noise in some regions of the image
- Can cause a loss of important details, especially in images with uniform intensity distribution
- It assumes a uniform distribution of pixel values, which may not be suitable for all images
- Can produce unnatural-looking results in some images, especially if the original image has a significant amount of contrast already

Result:

Original Image

Equalized Image

Sobel Operator

How it works

- The function first checks if the image is in grayscale ('L' mode). If not, it converts the image to grayscale
- The image is then converted into a NumPy array for easier processing
- Sobel operators for detecting edges in the horizontal (sobel_x) and vertical (sobel_y) directions are defined
- Two empty arrays grad_x and grad_y are created to store the gradients in the x and y directions
- A sliding window (3x3 region) is applied to each pixel in the image (ignoring the borders)
- For each pixel, the Sobel operator is applied by performing element-wise multiplication with the 3x3 region of the image, summing the results to calculate the gradients
- The magnitude of the gradient is computed using the formula sqrt(grad_x^2 + grad_y^2) for edge detection
- The result is clipped to a valid range (0 to 255) and converted back to a PIL image

Advantages

- Simple and efficient for detecting edges in an image, especially for detecting vertical and horizontal features
- It is easy to implement and computationally less expensive compared to more complex edge detection algorithms
- Helps in enhancing features like object boundaries and contours, which are useful for various image processing tasks such as object recognition and segmentation

Disadvantages

- The Sobel operator is sensitive to noise, which can result in false edges being detected.
- It only detects edges in the horizontal and vertical directions, so diagonal edges may not be as well-defined
- The output can sometimes be blurred, and additional techniques (like Gaussian filtering) might be required to reduce noise and improve result

Result:

Prewit Operator

How it works:

- First, the function checks if the input image is in grayscale ('L' mode). If not, it converts the image to grayscale
- The image is then converted into a NumPy array, and the height and width of the image are extracted
- Two Prewitt operator kernels are defined: one for detecting edges in the x-direction (prewitt_x) and one for detecting edges in the y-direction (prewitt_y)
- Empty arrays (grad_x and grad_y) are created to store the gradients in the x and y directions
- The function iterates through each pixel of the image (ignoring the borders) and extracts a 3x3 region around each pixel
- The gradients in both directions are calculated by multiplying the 3x3 region with the respective Prewitt operator kernels and summing the values
- The magnitude of the gradients is calculated by taking the square root of the sum of the squares of the gradients in both directions (grad_x and grad_y)
- The result is clipped to ensure pixel values stay within the range [0, 255]
- Finally, the result is converted back into an image format and returned

Advantages:

- Simple and easy to implement, making it suitable for basic edge detection tasks
- Effective at detecting horizontal and vertical edges in an image
- Computationally efficient for small images or applications where performance is not a primary concern

Disadvantages:

- Limited in its ability to detect more complex or diagonal edges compared to more advanced edge detection techniques
- Sensitive to noise, which may lead to false edge detections
- Does not perform well with images that have low contrast or gradual changes in intensity

Result:

Prewit Result

Original Image

Prewit Image

Kirsch Operator

How it works

- The input image is first converted to grayscale if it's not already in the 'L' mode (i.e., 8-bit pixels in grayscale)
- The image is then converted to a NumPy array for easier manipulation, and the height and width of the image are obtained
- The Kirsch operator uses eight different convolution kernels, each designed to highlight edges in various directions
- The kernels are applied to the image by sliding over the pixels and performing a convolution operation, which computes the weighted sum of pixel values in the neighborhood defined by the kernel
- The maximum gradient magnitude is taken for each pixel across all the kernel results, ensuring that the strongest edge response is selected
- The result is then clipped to the range [0, 255] to ensure pixel values are valid, and the processed image is converted back into a PIL Image object for output

Advantages

- The Kirsch operator can detect edges in multiple directions, providing a more comprehensive edge detection result than a single direction operator
- It highlights edges with higher contrast, making it useful for detecting strong boundaries within an image
- The operator is simple and computationally efficient, making it suitable for basic edge detection tasks

Disadvantages

- The Kirsch operator is sensitive to noise, meaning it may detect false edges in noisy images
- It may not perform well on images with low contrast or subtle edge details, as it relies on strong gradients to identify edges
- The convolution operation can be computationally expensive when applied to large images with multiple kernels

Result:

Kirsch Result

Original Image

Kirsch Image

High Pass Filter

How it works

- The function applies a High-Pass Filter to an input image to enhance edges and fine details
- It first ensures the image is in grayscale mode ('L') for simplicity in processing
- The image is converted into a NumPy array to perform mathematical operations efficiently
- A 3x3 high-pass filter kernel (mask_3x3_high_pass) is defined, emphasizing edge detection by enhancing areas with intensity changes
- The kernel is applied to the image using OpenCV's filter2D function, which performs convolution between the kernel and the image
- After convolution, the pixel values are clipped to the range [0, 255] to ensure valid image intensity values
- Finally, the processed NumPy array is converted back to an image format and returned

Advantages

- Enhances edges and sharpens details in the image, making it suitable for applications like object detection and feature extraction
- Simple and computationally efficient, making it practical for real-time applications
- Effective in isolating high-frequency components (e.g., edges, textures) while suppressing smooth areas

Disadvantages

- Can amplify noise, especially in images with significant background noise
- May cause halos or artifacts around edges due to the sharp transition in the filter kernel
- Less effective in handling images with subtle or low-contrast edges

Result:

Low Pass Filter

How it work

- The function applies a low-pass filter to an image, which is commonly used to reduce noise or blur an image
- It checks if the image is in grayscale mode ('L') and converts it if necessary
- The image is converted into a NumPy array for pixel-level operations
- A 3x3 low-pass filter kernel is defined, which assigns higher weight to the central pixel and distributes lower weights to its neighbors
- The filter is applied using OpenCV's filter2D function to perform convolution
- The resulting pixel values are clipped to the range [0, 255] to ensure valid grayscale values
- The filtered array is converted back to an image for output

Advantages

- Reduces high-frequency noise and smoothens the image, making it useful for preprocessing tasks like denoising
- Enhances uniform areas while minimizing abrupt intensity changes, helpful in some analytical applications
- Computationally efficient with a small kernel size, suitable for real-time image processing

Disadvantages

- May blur edges and fine details, which can degrade image sharpness
- Not effective in retaining important structural details in high-frequency regions
- Limited applicability in scenarios requiring preservation of edge details or texture

Result:

Median Filter

How it works:

- The function applies a median filter to the input image to reduce noise while preserving edges
- If the image is not in grayscale mode ('L'), it is converted to grayscale
- The image is converted into a NumPy array for pixel-level manipulation
- A sliding 3x3 window (neighborhood) is used to calculate the median value of the surrounding pixels for each pixel
- The median value replaces the center pixel of the window, effectively smoothing the image and reducing noise
- The filtered pixel values are clipped to the valid range [0, 255] to ensure proper image formatting
- The filtered array is converted back to an image for output

Advantages:

- Effectively removes salt-and-pepper noise from images
- Preserves edges better than other smoothing filters, such as mean or Gaussian filters
- Simple to implement and computationally efficient for small window sizes

Disadvantages:

- Performance can degrade for large window sizes, as it requires sorting values within the window for each pixel
- May not effectively handle large amounts of complex noise
- Can blur fine details in the image, especially in regions with high-frequency textures

Result:

Addition

How it works

- The function takes two images as input and ensures they are grayscale by converting them to mode 'L'
- If the sizes of the two images are different, the second image is resized to match the dimensions of the first image
- Both images are converted into NumPy arrays for pixel-wise operations, with their data type set to float32 for precise calculations
- A new empty NumPy array is created to store the pixel-wise sum of the two images
- The function iterates over each pixel position (height and width) of the images and computes the sum of corresponding pixel values from both images
- The resultant pixel values are clipped to the valid grayscale range (0–255) to avoid overflow or underflow.
- The final image is converted back to the Pillow format for compatibility and display

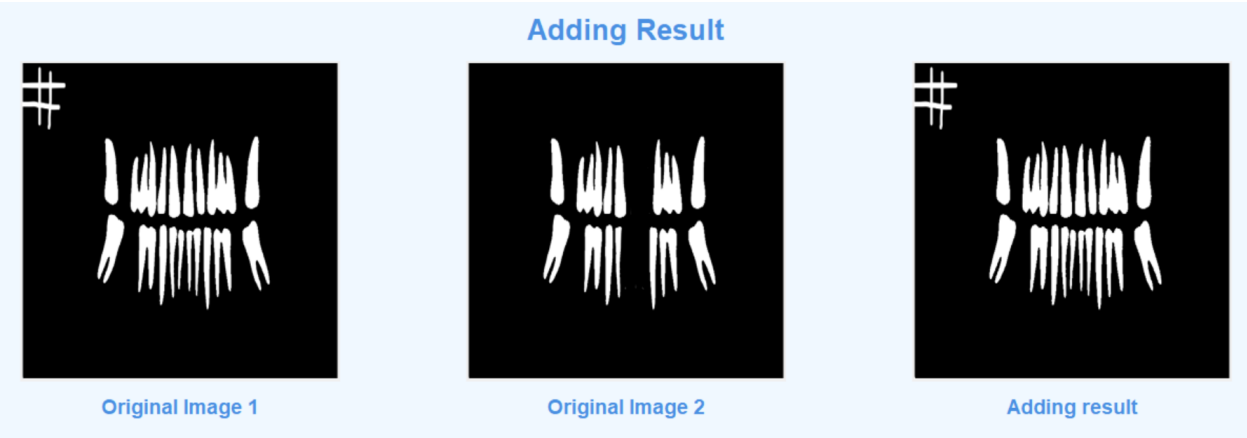
Advantages

- Enables pixel-by-pixel addition of two images, useful for blending and overlaying effects in image processing
- Handles grayscale conversion automatically, making the function robust to different input formats
- Prevents dimension mismatch by resizing images, ensuring compatibility during operations
- Uses clipping to maintain valid pixel intensity ranges, preserving image quality

Disadvantages

- Computationally expensive for large images due to nested loops for pixel-by-pixel addition
- Limited to grayscale images; it does not directly support color images (RGB or other formats)
- Potential loss of detail in the resultant image when resizing is applied to match dimensions
- No built-in mechanism for blending weights or controlling the intensity of addition

Result:



Subtraction

How it work

- The function performs pixel-wise subtraction between two images
- It ensures both images are in grayscale mode ('L') for uniform processing
- If the images differ in size, the second image is resized to match the first image's dimensions
- Both images are converted into NumPy arrays for numerical processing
- A new array is created to store the subtraction results
- Each pixel value of the first image is subtracted from the corresponding pixel value of the second image using nested loops
- The resulting values are clipped to ensure they fall within the valid range (0–255)
- Finally, the processed array is converted back into an image format and returned

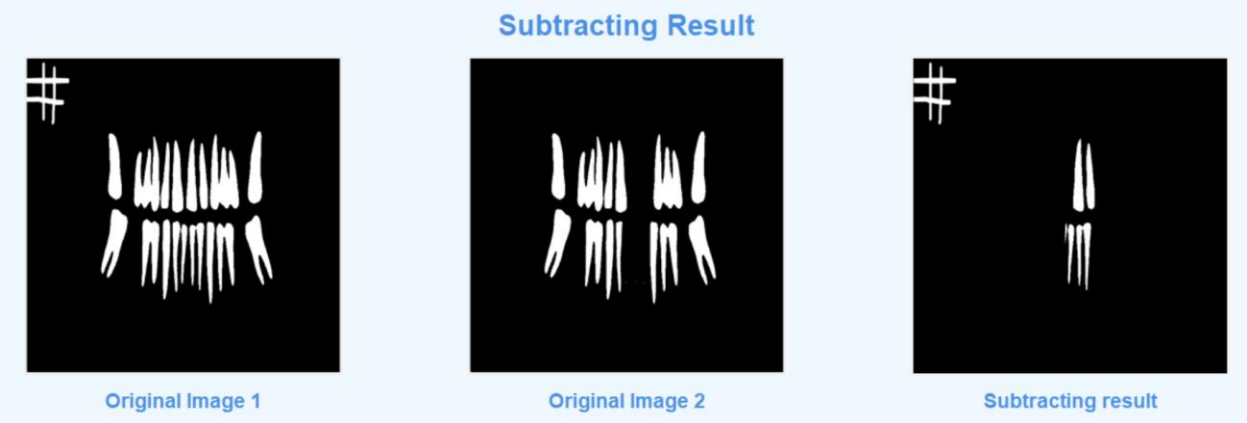
Advantages

- Allows detailed pixel-by-pixel comparison between two images
- Useful for highlighting differences between two images, such as changes or motion
- Supports resizing to handle images of different dimensions
- Operates in grayscale, making it computationally efficient

Disadvantages

- Limited to grayscale images; does not directly handle color images
- Slower for large images due to the use of nested loops for pixel processing
- Requires both images to be aligned properly; misaligned images can produce misleading results
- May lose information due to clipping of negative values in the subtraction process

Result:



Cut and Paste

How It Works

- The function takes two grayscale images (image1 and image2), a position to cut (cut_position), and the size of the cut region (cut_size)
- If either image is not in grayscale, it is converted to grayscale using the .convert('L') method
- If the sizes of the two images differ, image2 is resized to match the dimensions of image1
- The function converts both images into NumPy arrays for pixel-wise manipulation
- It calculates the bounds of the region to be cut from image1 based on the specified position and size
- The cut region from image1 is extracted and pasted onto the corresponding region of image2
- The pixel values are clipped to the valid range (0–255) to ensure image integrity
- Finally, the modified array is converted back into an image using Image.fromarray() and returned

Advantages

- Provides a simple and efficient way to combine regions from two images
- Useful for tasks like blending images or generating synthetic data
- Handles images of different sizes by resizing them automatically
- Allows precise control over the position and size of the cut-and-paste region

Disadvantages

- Limited to grayscale images; additional adjustments are needed for colored images
- May produce unrealistic outputs if the pixel intensities or patterns of the two images are significantly different
- Lacks advanced blending techniques, which could lead to harsh edges in the cut-and-paste region
- Manual specification of cut size and position can be time-consuming for complex tasks

Result:

Invert

How it work

- The function first checks the mode of the input image. If it is not grayscale ('L'), it converts the image to grayscale using image.convert('L')
- The grayscale image is then converted into a NumPy array of type float32 to facilitate mathematical operations
- The pixel values are inverted using the formula 255 - pixel_value, which effectively reverses the intensity of each pixel
- The pixel values are clipped to ensure they remain within the range [0, 255]
- The inverted image array is then converted back to a PIL image using Image.fromarray

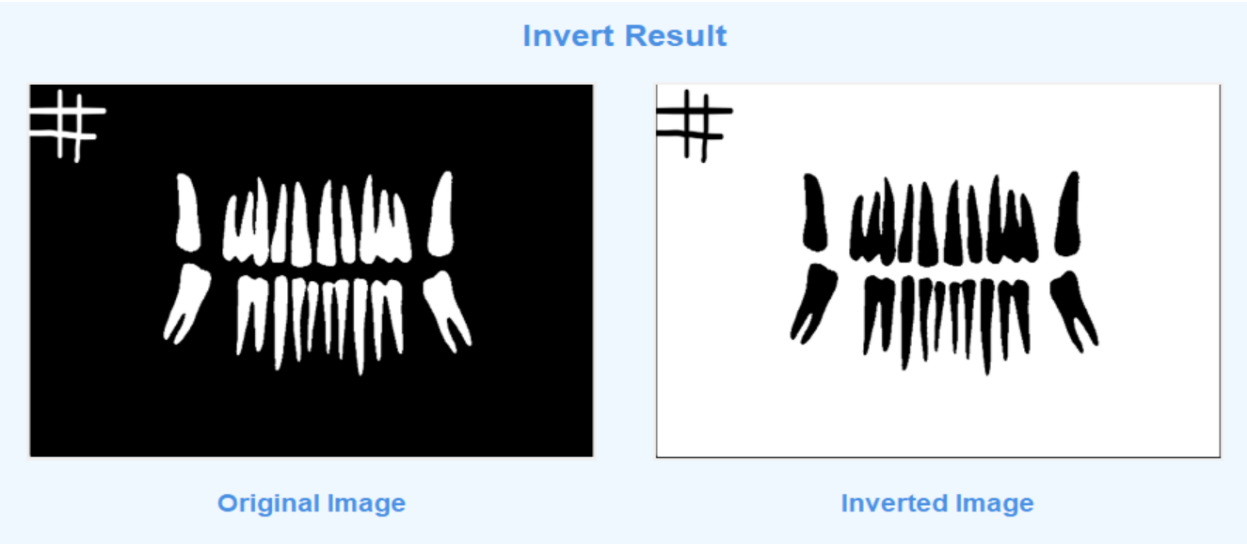
Advantages

- Works efficiently for grayscale images, making it suitable for image analysis tasks requiring contrast inversion
- Simple and easy-to-implement algorithm with minimal computational requirements
- Provides better visibility of features in some images by reversing intensity levels

Disadvantages

- Does not directly support color images; additional preprocessing is needed to handle colored input
- Limited application as the inversion effect may not be meaningful for all image types or processing tasks
- May introduce noise in certain scenarios if used on images with low contrast

Result:



Homogeneity Operator

How it works

- The function takes an image and converts it to grayscale ('L' mode) if it isn't already
- The grayscale image is converted into a NumPy array for pixel-level manipulation
- For each pixel (excluding the border pixels), the function calculates the absolute differences between the pixel value and its eight neighboring pixels
- The maximum difference among the neighbors is taken as the "homogeneity value" for that pixel
- If the homogeneity value exceeds the specified threshold, the pixel in the output image is assigned that value; otherwise, it is set to 0
- The resulting array is clipped to valid image intensity values (0–255) and converted back to an image format

Advantages

- Detects regions with high variability, making it effective for edge or texture detection
- Works well on grayscale images, which are computationally simpler than color images
- Provides control over sensitivity to changes in the image through the threshold parameter

Disadvantages

- Computationally intensive, especially for large images, due to the per-pixel neighborhood comparisons
- Can produce noisy results if the threshold is too low, requiring post-processing to clean the output
- Ignores the contextual relationship of pixels, which may limit its effectiveness in complex scenes

Result:

Difference Operator

How it works

- The function processes an image using a difference operator technique, which calculates the intensity differences between neighboring pixels to highlight edges or transitions
- The input image is first converted to grayscale ('L' mode) if it is not already, ensuring the operation works on a single channel
- The image is converted into a NumPy array for pixel-wise manipulation, and an empty array of the same size is created to store the results
- For each pixel (excluding the border pixels), four directional differences are calculated based on the pixel's neighbors:
  - Top-left to bottom-right
  - Top-center to bottom-center
  - Top-right to bottom-left
  - Left-center to right-center
- The maximum difference among these four values is computed for each pixel and assigned to the result image if it exceeds the specified threshold. Pixels below the threshold are set to 0.
- The processed image is clipped to ensure values remain within the range of 0–255 and converted back to an image format.

Advantages

- Highlights significant transitions or edges, making it effective for edge detection tasks
- Simple and computationally lightweight, suitable for applications that require basic image analysis
- Flexible, as the threshold value can be adjusted to control sensitivity to differences

Disadvantages

- Sensitive to noise, as small variations in intensity can result in false edges
- Does not account for texture or patterns, focusing only on pixel intensity differences
- Limited to grayscale images; additional preprocessing is needed for color images
- May miss edges or details in complex regions with subtle intensity transitions

Result:

Variance Operator

How it work

- The function starts by converting the input image to grayscale (if it isn't already) since variance calculation is usually performed on single-channel images
- The image is converted into a NumPy array for efficient mathematical operations
- The dimensions of the image are used to prepare an output array initialized to zeros
- A nested loop iterates through each pixel of the image (excluding the border pixels)
- For each pixel, a 3x3 neighborhood around it is extracted
- The mean of the neighborhood is calculated
- The variance is computed by summing the squared differences between each pixel in the neighborhood and the mean, then dividing by the total number of pixels (9 for a 3x3 window)
- The variance value is assigned to the corresponding pixel in the output array
- Finally, the output array is converted back to an image and returned

Advantages

- Captures the local variability or texture of the image, highlighting areas with high variance (e.g., edges or textured regions)
- Useful for texture-based image analysis and feature extraction in computer vision tasks
- Can enhance patterns and structural details that may not be visible in the original image

Disadvantages

- Computationally intensive due to the nested loops and per-pixel variance calculation, especially for large images
- Sensitive to noise, as small fluctuations in pixel intensity can result in exaggerated variance values
- Border pixels are excluded from processing, which can lead to incomplete analysis in some cases

Result:

Variance Result

Original Image

Variance Image

Range Operator

How it works

- The range_operator function processes an input image by first checking its color mode and converting it to grayscale ('L' mode) if necessary
- The image is then converted into a NumPy array for easier manipulation and calculations
- The height and width of the image are obtained to define the boundaries for pixel processing
- A new array (output) is created to store the processed values
- The function iterates through the image, excluding the borders, and for each pixel, it considers a 3x3 neighborhood of pixels around it
- For each neighborhood, the range value is calculated as the difference between the maximum and minimum pixel values in the neighborhood
- The calculated range value is assigned to the corresponding position in the output array
- After processing all the pixels, the output array is converted back into an image and returned

Advantages

- Simple and computationally efficient, especially for small images
- Highlights local variations in the image, making it useful for edge detection in certain contexts
- Can be applied to any grayscale image, as it operates based on pixel intensities

Disadvantages

- The method is sensitive to noise, as the range value may vary significantly in noisy areas
- Only works on local neighborhoods and does not take into account the broader context of the image
- May not capture finer details or complex structures as effectively as more advanced edge detection methods

Result:

Contrast Based Operator

How it works

- First, the image is checked for its color mode. If the image is not in grayscale ('L'), it is converted into grayscale
- The image is then converted to a NumPy array with the data type set to np.float32 for precision in calculations
- An edge detection mask (a simple Laplacian filter) is applied to the image using the cv2.filter2D function. This mask highlights edges by applying a convolution operation
- A smoothing mask is created using a 3x3 kernel filled with equal values to blur the image slightly. This helps in calculating the average pixel values
- The smoothing mask is applied to the image to get a blurred version of the image
- To avoid division by zero errors, a very small value (1e-10) is added to the average image output
- The contrast edges are calculated by dividing the edge-detected image by the smoothed image
- Finally, the result is converted back to a PIL image

Advantages

- Can effectively highlight edges in images by utilizing contrast-based methods, which can improve the clarity of image features
- Works well in extracting prominent features while minimizing noise
- The method is simple and computationally efficient, making it suitable for quick edge detection tasks
- The approach uses a smoothing filter to reduce noise, which can make the edge detection more robust

Disadvantages

- The method may produce less accurate results for images with low contrast or for very noisy images, as the edge detection depends on the contrast between adjacent regions
- The simple kernel used for edge detection might not capture finer details in the image, especially for complex images with intricate features
- For images with high-frequency noise, the method may enhance the noise along with the edges
- The result might not be as precise or detailed as more advanced edge detection techniques, such as Canny or Sobel

Result:


Contrast Based Result
Original Image
Contrast Based Image

Difference of Gaussians

How it works:

- The function difference_of_gaussians applies a Difference of Gaussians (DoG) technique to an input image
- The image is first converted to grayscale (if it is not already in grayscale) using image.convert('L')
- Two separate filters are applied to the image using custom convolution masks (mask_7x7 and mask_9x9) with the OpenCV filter2D function. These filters perform edge detection with different kernel sizes
- The resulting images from both filters (image_7x7 and image_9x9) are subtracted from each other to highlight the difference between the two, emphasizing edge features
- The result is clipped to ensure pixel values stay within the valid range (0-255), and then converted back into an 8-bit unsigned integer format
- Finally, the function returns three images: the difference of Gaussians image (dog), the result from the 7x7 mask, and the result from the 9x9 mask

Advantages:

- Helps in detecting edges and fine details in an image by enhancing the contrast between regions of different intensities
- The use of two different mask sizes (7x7 and 9x9) can highlight both fine and broader details, making the method adaptable to various features in images
- The Difference of Gaussians method can suppress noise while enhancing significant edges in the image, making it effective for tasks such as feature detection
- The method is computationally efficient compared to other edge-detection techniques like the Canny edge detector

Disadvantages:

- The method might not be suitable for images with complex backgrounds or subtle edges, as the difference between the Gaussian filters can be too harsh
- The results depend heavily on the choice of kernel sizes, and selecting an inappropriate kernel size may lead to loss of important image details or excessive noise
- It is sensitive to the input image's quality and resolution; lower-quality images may result in unsatisfactory edge detection

Result:

Original Image

DOG Image

Manual Segmentation

How it works:

- The function first checks if the image is in grayscale ('L' mode). If not, it converts the image to grayscale
- It then converts the image to a NumPy array with np.float32 to perform numerical operations
- A new array segmented_image is created with the same shape as the input image, initialized to zeros
- The function applies the thresholding operation, where pixel values within the specified low_threshold and high_threshold are set to 255 (white), indicating that those pixels are part of the segmented area
- The resulting segmented image is clipped to ensure that all values are within the valid range of 0 to 255
- Finally, the NumPy array is converted back to a PIL Image and returned as the output

Advantages:

- Simple and easy to implement, providing a basic yet effective way to segment an image based on intensity values
- It allows for control over the range of values to segment by using customizable low_threshold and high_threshold parameters
- Useful for applications where objects of interest have distinct intensity ranges

Disadvantages:

- The method assumes a fixed threshold range, which may not work well for images with varying lighting conditions or contrast
- It may not work well for more complex images with subtle transitions in intensity
- The method is sensitive to noise, as any noise within the threshold range will also be included in the segmentation

Result:

Manual Segmentation Result

Original Image

Manual Segmentation Image

Peak Segmentation

How it works:

- The image is first checked to ensure it is in grayscale ('L') mode. If it is not, it is converted to grayscale
- The image is then converted into a NumPy array for easy manipulation of pixel values
- A histogram of the image is computed using the calculate_histogram() function
- The peaks in the histogram are identified and sorted using the find_peaks_and_sort_them() function
- If fewer than two peaks are found, the low and high thresholds are set to 0 and 255, respectively, meaning no segmentation will occur
- If two or more peaks are detected, the calculate_low_and_high_thresholds() function determines the low and high thresholds based on the peak positions
- The image is then segmented by setting pixel values within the threshold range (low to high) to 255 (white), while all other pixels are set to 0 (black)
- The resulting segmented image is returned as a new image object

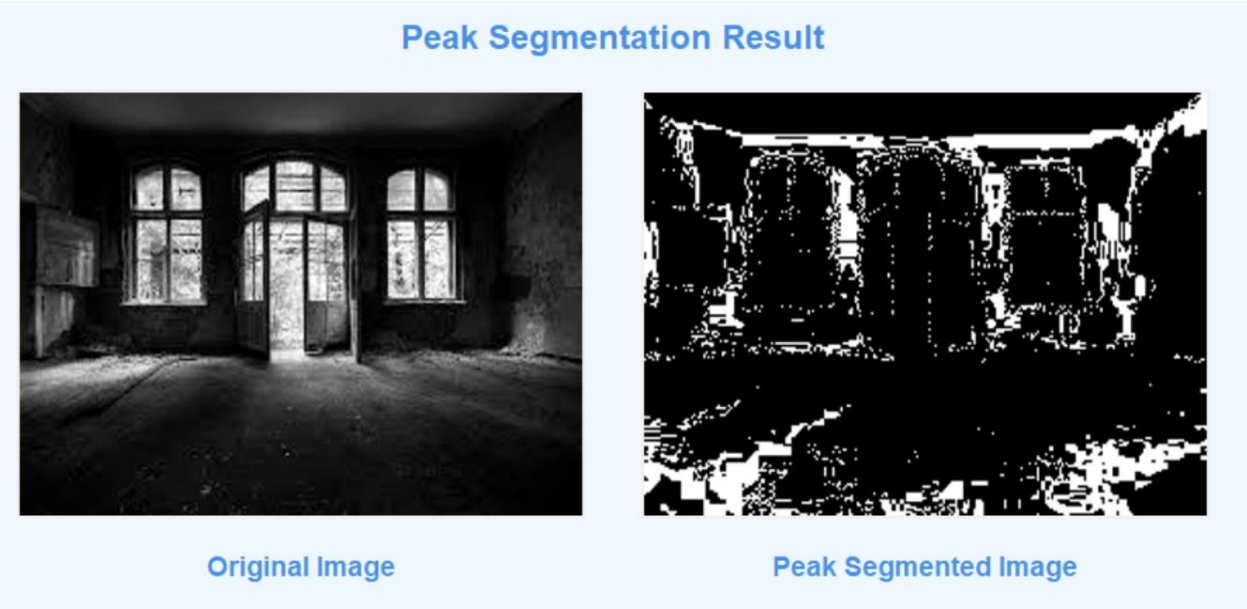
Advantages:

- Simple and effective for images with clear and distinct intensity peaks in the histogram.
- Does not require any manual intervention, making it an automated method
- Can easily be applied to various grayscale images with minimal setup

Disadvantages:

- Struggles with images that have a noisy or uneven histogram, as it may fail to detect two distinct peaks
- Less effective on images where the object and background have similar intensities, leading to poor segmentation
- Assumes that the image contains two primary regions, which might not always be the case

Result:


Peak Segmentation Result
Original Image
Peak Segmented Image

Valley Segmentation

How it works:

- The function first checks if the image is in grayscale ('L' mode). If not, it converts it to grayscale
- The image is then converted into a NumPy array for easier processing
- A histogram of pixel intensities is calculated from the image
- Peaks in the histogram are identified and sorted to find significant intensity levels
- If there are fewer than two peaks, the image is segmented with the entire intensity range (0 to 255)
- If there are more than two peaks, the function looks for a valley point between the peaks and uses it to determine a low and high threshold for segmentation
- The image is segmented by setting all pixel values within the threshold range to 255 (white), while others remain black (0)
- The segmented image is returned as a PIL image

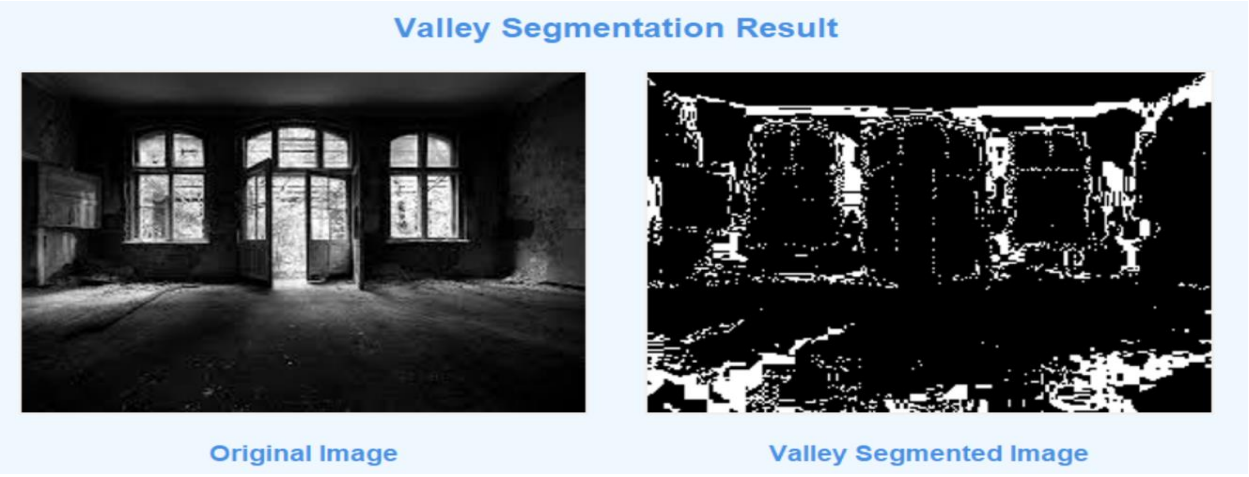
Advantages:

- Simple and effective method for thresholding based on histogram characteristics
- Can adapt to varying image content if peaks and valleys are well defined in the histogram
- Does not require complex parameters or manual tuning if histogram features are prominent

Disadvantages:

- Assumes that there are distinct valleys in the histogram, which may not always be the case, especially for noisy or low-contrast images
- Limited in handling complex images with overlapping intensity distributions, where multiple segmentation thresholds may be needed
- Sensitive to the quality of the histogram and the presence of noise in the image

Result:



Adaptive Segmentation

How it works:

- The function starts by checking if the image is in grayscale ('L') mode. If not, it converts the image to grayscale
- It converts the image into a NumPy array for further processing
- A histogram of pixel intensity values is calculated using the calculate_histogram function
- Peaks in the histogram are identified and sorted using the find_peaks_and_sort_them function
- The find_valley_points function is used to determine the valley points between the peaks, which helps in segmenting the image
- The low and high thresholds for segmentation are determined based on the valley points using the valley_high_low function
- A first pass of segmentation is performed by setting pixel values between the low and high thresholds to 255 (white)
- The means of the background and object regions are calculated in the first pass segmented image using calculate_means
- New peak indices are calculated based on the background and object means
- Valley points are recalculated based on these new peaks, and new low and high thresholds are determined
- A second pass of segmentation is performed with the new thresholds, generating the final segmented image
- The function returns the final segmented image as a PIL image object

Advantages:

- Adaptive histogram thresholding adjusts thresholds based on the image content, making it suitable for images with varying lighting conditions
- It can effectively segment images with clear foreground and background contrasts, providing better segmentation results compared to fixed thresholding methods
- The method improves segmentation accuracy by performing multiple passes to refine the threshold values
- It’s a non-parametric method, meaning it doesn't require prior knowledge of the image, such as the number of objects or their intensity ranges

Disadvantages:

- The method may not perform well for images with very low contrast or complex backgrounds, leading to poor segmentation results
- It can be computationally expensive due to multiple passes through the image and histogram calculations
- The choice of peaks and valleys may not always correspond well to the true object boundaries, especially in images with overlapping intensities
- The approach might struggle with images that have uneven lighting or noise, requiring additional preprocessing to work effectively

Result: