A picture containing text, font, logo, graphics

Description automatically generated

Domain Filters

Youssef Mohamed, Mina Samir, Ahmed Hossam, Steven Hany

211777 212257 213581 210991

Faculty of Computer Science

Group B

Dr. Ehab Emam

May 19, 2023

TABLE OF CONTENTS

Introduction……………………….…………….………………………………………...1

Spatial Domain Filters ………………………….………………...………………………………2

Smoothing Spatial Filters……………………………………………...2

Median Filter…………………………………….2

Adaptive Median Filter……………………….…2

Averaging Filter…………………………………2

Gaussian Filter………………………………….2

Sharpening Spatial Filters………………...……………….………….2

Laplacian Operator……………………………….3

Unsharp Masking and High boost Filtering……….3

Roberts Cross-Gradient Operators………………….4

Sobel…………………………………………………5

Noise Filters……………………………………….………………….3

Impulse Noise……………………………………….6

Gaussian Noise……………………………………….7

Uniform Noise…………………………………………8

Transform / Frequency Domain Filters…….…………………….……………………………….8

Histogram Equalization………………………………………………………….9

Histogram Specification………………………………………………………….9

Fourier Transform………………………………………………………………10

Interpolation……………………………………………………………………...11

Introduction

Image processing plays a crucial role in various fields, including computer vision, medical imaging, surveillance, and multimedia applications. By manipulating and analyzing digital images, image processing techniques enable us to extract meaningful information, enhance image quality, and automate tasks that would otherwise be time-consuming or challenging for human perception. One fundamental aspect of image processing involves the application of filters to modify images and extract specific features or patterns. Filters can be categorized into spatial domain filters and frequency domain filters, each offering distinct advantages and applications. In this discussion, we will explore these different types of filters, their underlying principles, and their practical implications in image processing.

Spatial Domain Filters

Smoothing (Median Filter):

This code implements a median filter for image processing using the OpenCV library in Python. Let's go through the different parts of the code:

1. Importing Libraries: The code begins by importing the necessary libraries. **tkinter.filedialog** is used to display a file dialog for selecting an image file, **cv2** is the OpenCV library for image processing, and **numpy** is used for numerical computations.
2. Function Definitions:
   * **GetImagePath()**: This function opens a file dialog to select an image file and returns the path of the selected file.
   * **ReadImage(Path)**: Reads the image from the specified path using OpenCV's **imread()** function and returns the grayscale version of the image.
   * **SaveImage(Path, Image)**: Saves the given image to the specified path using OpenCV's **imwrite()** function.
   * **ShowImage(WindowTitle, Image)**: Displays the image with the specified window title using OpenCV's **imshow()** function. The image will be shown until a key is pressed.
   * **PadImageWithZeros(Image)**: Adds a zero border around the input image using NumPy's **zeros()** function, creating a padded image with the same size plus two extra rows and columns.
   * **Int32ToUint8(List)**: Converts a list of values to the **uint8** data type, which is commonly used for image pixel values. This is done using NumPy's **astype()** function.
   * **CalculateMedian(Kernel)**: Computes the median value of a given kernel (3x3 matrix) by flattening the kernel, sorting the flattened values, and returning the middle element if the number of elements is odd, or the average of the two middle elements if it is even.
   * **ApplyMedianFilter(PaddedImage)**: Applies the median filter to the padded image. It iterates over each pixel, extracts the surrounding 3x3 kernel, calculates the median value using **CalculateMedian()**, and assigns the median value to the corresponding pixel in the filtered image.
3. Image Processing:
   * **Path = GetImagePath()**: Calls the **GetImagePath()** function to obtain the path of the image file selected by the user.
   * **OriginalImage = ReadImage(Path)**: Reads the image from the specified path and assigns it to the **OriginalImage** variable.
   * **PaddedImage = PadImageWithZeros(OriginalImage)**: Adds zero padding to the original image, creating the padded image.
   * **FilteredImage = ApplyMedianFilter(PaddedImage)**: Applies the median filter to the padded image, obtaining the filtered image.
   * **FilteredImage = Int32ToUint8(FilteredImage)**: Converts the filtered image to the **uint8** data type.
   * **SaveImage('Output Images\Median Filter.png', FilteredImage)**: Saves the filtered image to the specified path.

Overall, this code allows you to select an image file, apply a median filter to it using a 3x3 kernel, and save the filtered image to a specified location.



Smoothing (Adaptive Median Filter):

This code implements an adaptive median filter for image processing using the OpenCV library in Python. Here's a breakdown of the code:

1. Importing Libraries: The code begins by importing the necessary libraries. It includes **askopenfilename** from **tkinter.filedialog** to display a file dialog for selecting an image file, **cv2** for image processing operations, and **numpy** for numerical computations.
2. Function Definitions:
   * **GetImagePath()**: Opens a file dialog to select an image file and returns the path of the selected file.
   * **ReadImage(Path)**: Reads the image from the specified path using OpenCV's **imread()** function and returns the grayscale version of the image.
   * **SaveImage(Path, Image)**: Saves the given image to the specified path using OpenCV's **imwrite()** function.
   * **ShowImage(WindowTitle, Image)**: Displays the image with the specified window title using OpenCV's **imshow()** function. The image will be shown until a key is pressed.
   * **PadImageWithZeros(Image)**: Adds a zero border around the input image using NumPy's **zeros()** function, creating a padded image with the same size plus two extra rows and columns.
   * **Int32ToUint8(List)**: Converts a list of values to the **uint8** data type, which is commonly used for image pixel values. This is done using NumPy's **astype()** function.
   * **CalculateMedian(Kernel)**: Computes the median value of a given kernel (subimage) by flattening the kernel, sorting the flattened values, and returning the middle element if the number of elements is odd or the average of the two middle elements if it is even.
   * **CalculateMax(Kernel)**: Computes the maximum value of a given kernel using NumPy's **max()** function.
   * **CalculateMin(Kernel)**: Computes the minimum value of a given kernel using NumPy's **min()** function.
   * **NumpyIgnoreError()**: Sets the NumPy error handling to ignore any overflow errors.
   * **ApplyAdaptiveMedianFilter(PaddedImage, SMax)**: Applies the adaptive median filter to the padded image. It iterates over each pixel and adapts the filter size based on the local image characteristics. It calculates the minimum, maximum, and median values within the current filter size and applies the adaptive median filtering algorithm to determine the filtered value for the pixel.
3. Image Processing:
   * **NumpyIgnoreError()**: Calls the **NumpyIgnoreError()** function to ignore any overflow errors during calculations.
   * **Path = GetImagePath()**: Calls the **GetImagePath()** function to obtain the path of the image file selected by the user.
   * **OriginalImage = ReadImage(Path)**: Reads the image from the specified path and assigns it to the **OriginalImage** variable.
   * **PaddedImage = PadImageWithZeros(OriginalImage)**: Adds zero padding to the original image, creating the padded image.
   * **SMax = 7**: Specifies the maximum filter size to be used in the adaptive median filter.
   * **FilteredImage = ApplyAdaptiveMedianFilter(PaddedImage, SMax)**: Applies the adaptive median filter to the padded image using the specified **SMax** value, obtaining the filtered image.
   * **FilteredImage = Int32ToUint8(FilteredImage)**: Converts the filtered image to the **uint8** data type.
   * **SaveImage('Output Images/Adaptive Median Filter.png', FilteredImage)**: Saves the filtered image to the specified path.

A picture containing outdoor, building, black and white, cloud

Description automatically generatedOverall, this code allows you to select an image file, apply an adaptive median filter to it with a variable filter size, and save the filtered image to a specified location. The adaptive median filter helps to effectively remove noise from the image while preserving edges and details.

Smoothing (Averaging Filter):

This code applies an averaging filter, also known as a box filter, to an image using OpenCV and NumPy libraries in Python. Let's break down the code:

1. Importing Libraries: The code begins by importing the necessary libraries. It includes **askopenfilename** from **tkinter.filedialog** to display a file dialog for selecting an image file, **cv2** for image processing operations, and **numpy** for numerical computations.
2. Function Definitions:
   * **GetImagePath()**: This function opens a file dialog to select an image file and returns the path of the selected file. It allows selecting PNG, BMP, JPEG, and JPG file types.
   * **ReadImage(Path)**: Reads the image from the specified path using OpenCV's **imread()** function and returns the grayscale version of the image by passing **0** as the second argument.
   * **SaveImage(Path, Image)**: Saves the given image to the specified path using OpenCV's **imwrite()** function.
   * **ShowImage(WindowTitle, Image)**: Displays the image with the specified window title using OpenCV's **imshow()** function. The image will be shown until a key is pressed.
   * **PadImageWithZeros(Image)**: Adds a zero border around the input image using NumPy's **zeros()** function, creating a padded image with the same size plus two extra rows and columns. The original image is assigned to the center of the padded image.
   * **Int32ToUint8(List)**: Converts a list of values to the **uint8** data type, which is commonly used for image pixel values. This is done using NumPy's **astype()** function.
   * **ApplyAveragingFilter(PaddedImage)**: Applies the averaging filter to the padded image. It iterates over each pixel in the padded image and calculates the local sum by considering a 3x3 neighborhood around each pixel. The average of the local sum is calculated as the local mean, and the pixel in the filtered image is updated with the local mean. The function returns the filtered image.
3. Image Processing:
   * **Path = GetImagePath()**: Calls the **GetImagePath()** function to obtain the path of the image file selected by the user.
   * **OriginalImage = ReadImage(Path)**: Reads the image from the specified path and assigns it to the **OriginalImage** variable.
   * **PaddedImage = PadImageWithZeros(OriginalImage)**: Adds zero padding to the original image, creating the padded image.
   * **FilteredImage = ApplyAveragingFilter(PaddedImage)**: Applies the averaging filter to the padded image, obtaining the filtered image.
   * **FilteredImage = Int32ToUint8(FilteredImage)**: Converts the filtered image to the **uint8** data type.
   * **SaveImage('Output Images\Averaging Filter.png', FilteredImage)**: Saves the filtered image to the specified path.

Overall, this code allows you to select an image file, apply an averaging filter to it, and save the filtered image to a specified location. The averaging filter replaces each pixel with the average value of its surrounding neighborhood, resulting in a blurred effect and reducing image noise.



Smoothing (Gaussian Filter):

This code demonstrates a graphical user interface (GUI) for denoising an image using a Gaussian filter. Let's go through the code step by step:

1. Importing Libraries: The necessary libraries are imported, including **numpy** for numerical operations, **tkinter** for creating the GUI, **PIL** (Python Imaging Library) for image processing tasks, and **filedialog** from **tkinter** for opening file dialogs.
2. Gaussian\_Filter Function: This function generates a Gaussian filter kernel of a specified size and sigma value. It creates a meshgrid of coordinates using **np.mgrid**, computes the Gaussian values for each coordinate using the formula, and normalizes the kernel by dividing it by the sum of all values. The function returns the generated Gaussian kernel.
3. Denoise\_Image Function: This function performs the denoising operation on the input image using the Gaussian filter. It takes an image object, size, and sigma as parameters. The function first converts the image to grayscale using the **'L'** mode. Then, it converts the image to a NumPy array. It determines the height and width of the image array.

The function calls the **Gaussian\_Filter** function to obtain the Gaussian kernel with the specified size and sigma. It initializes an empty array called **Smoos\_Image** with dimensions based on the image size and kernel size.

The denoising operation is performed by iterating over the image pixels within the valid range, excluding the border pixels. For each pixel, a local neighborhood region is extracted using the size of the Gaussian filter kernel. The corresponding region in the image array is multiplied with the Gaussian kernel, and the element-wise product is summed. The result is assigned to the corresponding pixel location in **Smoos\_Image**.

After completing the denoising operation, the resulting **Smoos\_Image** array is converted back to an image object with a data type of **'uint8'** (unsigned 8-bit integer) using **Image.fromarray**. The denoised image is returned.

1. GF\_GUI Class: This class defines the GUI for the image denoising application. It takes the **master** (main window) as a parameter.
   * **\_\_init\_\_**: Initializes the GUI by setting the window title and creating buttons and labels for displaying images.
   * **Load\_Image**: This method is triggered when the "Browse" button is clicked. It opens a file dialog to select an image file. Once a file is selected, the method loads the image, updates the label to display the original image, and performs the denoising operation using the **Denoise\_Image** function. The denoised image is displayed in another label.
2. Main Program:
   * **root = Tk()**: Creates the main window of the GUI.
   * **gui = GF\_GUI(root)**: Initializes an instance of the **GF\_GUI** class.
   * **root.mainloop()**: Starts the main event loop for the GUI, allowing user interactions.

When running this code, the GUI window will appear with a "Browse" button. Clicking the button opens a file dialog to select an image file. Once an image is selected, the GUI will display the original image and the denoised image side by side.

The denoising is performed using a Gaussian filter with a specified size (**5** in the code) and sigma value (**1** in the code). Adjusting these parameters will affect the denoising result.



Sharpening (Laplacian Operator Filter):

This code demonstrates the application of a Laplacian operator filter on an image. Let's break down the code step by step:

1. Importing Libraries: The necessary libraries are imported, including **askopenfilename** from **tkinter.filedialog** for selecting an image file, **cv2** for image processing operations, and **numpy** for numerical computations.
2. GetImagePath Function: This function opens a file dialog using **askopenfilename** and allows the user to select an image file. It returns the path of the selected image file.
3. ReadImage Function: This function takes a file path as input and uses **cv2.imread** to read the image as a grayscale image (0 for grayscale mode). It returns the image data.
4. SaveImage Function: This function takes a file path and an image as input and uses **cv2.imwrite** to save the image to the specified path.
5. ShowImage Function: This function takes a window title and an image as input and displays the image using **cv2.imshow**.
6. WaitKey Function: This function waits until a key is pressed. It uses **cv2.waitKey(0)** for waiting.
7. PadImageWithZeros Function: This function takes an image as input and pads it with zeros by creating a new array with dimensions increased by 2 in both rows and columns. The original image is then placed in the center of the new array. The padded image is returned.
8. Int32ToUint8 Function: This function converts a list or array from **int32** data type to **uint8** data type using **astype(numpy.uint8)**. It returns the converted array.
9. CreateLaplacianOperatorKernel Function: This function prompts the user to choose a kernel for the Laplacian operator. It displays four different kernels and asks the user to enter a number corresponding to the desired kernel. Based on the user's input, the function returns the selected kernel. If an invalid input is provided, the function calls itself recursively until a valid input is entered.
10. ApplyLaplacianOperatorFilter Function: This function applies the Laplacian operator filter to the padded image. It first creates a copy of the padded image called **FilteredImage**. Then, it obtains the Laplacian operator kernel by calling the **CreateLaplacianOperatorKernel** function.

The function iterates over the pixels of the padded image (excluding the border) and applies the Laplacian operator by taking the element-wise product of the kernel and the corresponding neighborhood in the padded image. The sum of the element-wise products is computed and assigned to the corresponding pixel in **FilteredImage**.

The resulting values in **FilteredImage** are clipped between 0 and 255 using **numpy.clip** to ensure they are within the valid range for image pixel values.

Finally, the filtered image is returned.

1. Main Program:
   * **Path = GetImagePath()**: The user is prompted to select an image file using the **GetImagePath** function, and the path of the selected image is stored in the **Path** variable.
   * **OriginalImage = ReadImage(Path)**: The image is read using the **ReadImage** function, and the grayscale image data is stored in the **OriginalImage** variable.
   * **PaddedImage = PadImageWithZeros(OriginalImage)**: The original image is padded with zeros using the **PadImageWithZeros** function, and the padded image is stored in the **PaddedImage** variable.
   * **FilteredImage = ApplyLaplacianOperatorFilter(PaddedImage)**: The Laplacian operator filter is applied to the padded image using the **ApplyLaplacianOperatorFilter** function, and the filtered image is stored in the **FilteredImage** variable.
   * **FilteredImage = Int32ToUint8(FilteredImage)**: The filtered image is converted from **int32** data type to **uint8** data type using the **Int32ToUint8** function.
   * **SaveImage('Output Images/Laplacian Operator Filter.png', FilteredImage)**: The filtered image is saved as a PNG file with the specified path and file name.

The code allows the user to select an image, applies the Laplacian operator filter to the image, and saves the filtered image as a PNG file.



Sharpening (Unsharp Masking and High boost Filtering):

The provided code demonstrates two different approaches for performing unsharp masking and high boost filtering on an image using different filters: median filter and Gaussian filter. Let's break down each code segment separately:

1. Median Filter Approach:
   * The code begins by importing the necessary libraries and functions.
   * **GetImagePath()** function prompts the user to select an image file and returns the selected file's path.
   * **ReadImage(Path)** function reads the image from the specified path as a grayscale image (0 for grayscale mode) using **cv2.imread**.
   * **SaveImage(Path, Image)** function saves the given image to the specified path using **cv2.imwrite**.
   * **ShowImage(WindowTitle, Image)** function displays the given image with the specified window title using **cv2.imshow**.
   * **PadImageWithZeros(Image)** function pads the input image with zeros by creating a new array with dimensions increased by 2 in both rows and columns. The original image is then placed in the center of the new array. The padded image is returned.
   * **Int32ToUint8(List)** function converts a list or array from **int32** data type to **uint8** data type using **astype(numpy.uint8)**.
   * **CalculateMedian(Kernel)** function calculates the median value of a given kernel (3x3 neighborhood) by flattening the kernel, sorting the values, and finding the middle value.
   * **ApplyMedianFilter(PaddedImage)** function applies the median filter to the padded image. It iterates over the pixels of the padded image, extracts the 3x3 kernel around each pixel, calculates the median value of the kernel using **CalculateMedian**, and assigns the median value to the corresponding pixel in the filtered image. The filtered image is then returned.
   * **ApplyUnsharpMaskingAndHighboostFilteringFilter(OriginalImage, K)** function applies unsharp masking and high boost filtering to the original image. It first pads the original image with zeros using **PadImageWithZeros**, applies the median filter to the padded image using **ApplyMedianFilter**, converts the filtered image to **uint8** using **Int32ToUint8**, computes the mask by subtracting the filtered image from the padded image, and finally sharpens the image by adding K times the mask to the padded image. The sharpened image is returned.
   * The code prompts the user to enter the weight of the mask (K) and stores it in the variable **K**.
   * Based on the value of K, the code either applies the unsharp masking and high boost filtering or bypasses it.
   * The sharpened image is then saved using **SaveImage** function.
2. Gaussian Filter Approach:
   * The code begins by importing the necessary libraries and functions.
   * **GetImagePath()** function prompts the user to select an image file and returns the selected file's path.
   * **ReadImage(Path)** function reads the image from the specified path as a grayscale image (0 for grayscale mode) using **cv2.imread**.
   * **SaveImage(Path, Image)** function saves the given image to the specified path using **cv2.imwrite**.
   * **ShowImage(WindowTitle, Image)** function displays the given image with the specified window title using **cv2.imshow**.
   * **Int32ToUint8(List)** function converts a list or array from **int32** data type to **uint8** data type using **astype(numpy.uint8)**.
   * **CreateGaussianKernel(Size, Sigma)** function creates a Gaussian kernel of the specified size and standard deviation (Sigma). The kernel values are calculated using the Gaussian function and normalized by dividing by the sum of all kernel values. The created kernel is returned.
   * **ApplyGaussianFilter(OriginalImage)** function applies a Gaussian filter to the original image. It creates a Gaussian kernel using **CreateGaussianKernel**, pads the original image with zero borders using **cv2.copyMakeBorder**, iterates over each pixel in the original image, convolves the kernel with the corresponding neighborhood in the padded image, and assigns the weighted average of the neighborhood to the corresponding pixel in the filtered image. The filtered image is then returned.
   * **ApplyUnsharpMaskingAndHighboostFilteringFilter(OriginalImage, K)** function applies unsharp masking and high boost filtering to the original image. It first applies the Gaussian filter to the original image using **ApplyGaussianFilter**, converts the filtered image to **uint8** using **Int32ToUint8**, computes the mask by subtracting the filtered image from the original image, and finally sharpens the image by adding K times the mask to the original image. The sharpened image is returned.
   * The code prompts the user to enter the weight of the mask (K) and stores it in the variable **K**.
   * Based on the value of K, the code either applies the unsharp masking and high boost filtering or bypasses it.
   * The sharpened image is then saved using **SaveImage** function.

In both approaches, the code utilizes OpenCV (cv2) for image manipulation, NumPy for array operations, and Tkinter's file dialog to prompt the user for selecting an image file. The functions are used to read an image, apply filters, convert data types, display images, and save the processed images.



Sharpening (Roberts Cross-Gradient Operators):

The provided code demonstrates the application of the Roberts Cross Gradient Operators filter to an image. Here's a breakdown of the code:

1. Importing Libraries:
   * The code imports the necessary libraries, including **askopenfilename** from **tkinter.filedialog**, **cv2** from OpenCV, and **numpy**.
2. Defining Functions:
   * **GetImagePath()** function opens a file dialog using **askopenfilename** and allows the user to select an image file. It returns the path of the selected file.
   * **ReadImage(Path)** function reads the image from the specified path as a grayscale image (0 for grayscale mode) using **cv2.imread**. It returns the read image.
   * **SaveImage(Path, Image)** function saves the given image to the specified path using **cv2.imwrite**.
   * **ShowImage(WindowTitle, Image)** function displays the given image with the specified window title using **cv2.imshow**.
   * **WaitKey()** function waits for a keyboard event. In this case, it waits indefinitely until a key is pressed. It is used in conjunction with **cv2.imshow**.
   * **ApplyRobertsCrossGradientOperatorsFilter(OriginalImage)** function applies the Roberts Cross Gradient Operators filter to the original image. It performs the following steps:
     + Creates two kernels for the Roberts Cross operators: **Gx** for horizontal gradients and **Gy** for vertical gradients.
     + Pads the original image with reflections at the borders using **cv2.copyMakeBorder** to handle edge cases.
     + Initializes an empty filtered image array.
     + Iterates over each pixel in the original image, calculating the horizontal and vertical gradient responses using element-wise multiplication between the neighborhood around the pixel and the respective gradient operators.
     + Computes the gradient magnitude by taking the square root of the squared Gx and Gy responses.
     + Clips the gradient magnitude values between 0 and 255 using **numpy.clip** to ensure they fit within the valid intensity range.
     + Assigns the computed gradient magnitude to the corresponding pixel in the filtered image array.
     + Finally, it returns the filtered image.
3. Image Processing:
   * The code calls **GetImagePath()** to prompt the user for an image file and stores the path in the variable **Path**.
   * **ReadImage(Path)** is then called to read the image from the specified path, and the resulting image is stored in the variable **OriginalImage**.
   * The **ApplyRobertsCrossGradientOperatorsFilter** function is applied to the **OriginalImage**, and the resulting filtered image is stored in the variable **FilteredImage**.
   * **SaveImage** is used to save the filtered image to the specified path.

The code essentially loads an image, applies the Roberts Cross Gradient Operators filter to it, and saves the filtered image. The Roberts Cross operator helps detect edges in the image by calculating the gradient magnitude.



Sharpening (Sobel Operator):

The provided code is a Python program that implements a GUI application for applying image filters using the Sobel operator. Here's a breakdown of the code:

1. The code imports necessary modules and libraries: **tkinter**, **filedialog** from **tkinter**, **Image** and **ImageTk** from the PIL library, **numpy**, **cv2** (OpenCV), **math**, **os**, and **random**.
2. The code defines a class **Apply\_IP\_Filter** that represents the GUI application.
3. The **\_\_init\_\_** method initializes the application window and sets up various GUI elements such as labels, buttons, frames, etc.
4. The **Load\_Image** method is triggered when the "Load Image" button is clicked. It opens a file dialog to select an image file and then displays the loaded image in the GUI.
5. The **Apply\_Filter** method applies the Sobel operator to the loaded image. It convolves the image with Sobel x and y kernels to compute the gradient magnitude at each pixel. Pixels with magnitudes above a certain threshold are set to 255 (white), and others are set to 0 (black).
6. The **Apply\_Noise** method applies noise to the loaded image. It iterates over each pixel of the image and adds random noise using a Gaussian distribution. The standard deviation of the noise is obtained from the user-entered value. The resulting noisy image is stored in the **Noise\_Image** variable.
7. The **ResetWindow** method is triggered when the "Reset" button is clicked. It closes the current window and restarts the application.
8. The program creates an instance of the **Apply\_IP\_Filter** class and starts the GUI application.

Overall, this code creates a simple GUI application for loading an image, applying noise to the image, and then applying the Sobel operator to detect edges in the noisy image.



Noise (Impulse):

This code demonstrates an impulse noise filter using the median filter algorithm. Here’s how it works:

1. The **Image** class is defined, which takes an image path as input and loads the image using OpenCV's **imread** function.
2. The **show\_image** method displays the original image using **cv2.imshow**.
3. The **generate\_salt\_and\_pepper\_noise** method adds salt and pepper noise to the image. It takes a threshold parameter that determines the amount of noise to be added. The method creates a copy of the original image and iterates over each pixel. For each pixel, a random value between 0 and 1 is generated. If this random value is below **threshold / 2**, the pixel is set to maximum intensity (salt noise). If the random value is above **1 - threshold / 2**, the pixel is set to minimum intensity (pepper noise). The method returns the noisy image.
4. The **median\_filter** method applies the median filter to the noisy image. It takes the noisy image and the kernel size as parameters. The method creates a copy of the noisy image and iterates over each pixel, excluding the image borders. For each pixel, a neighborhood around it is extracted based on the kernel size. The median value of the neighborhood is calculated, and the pixel is set to this median value. The method returns the filtered image.
5. An instance of the **Image** class is created, passing the path of the input image.
6. The **show\_image** method is called to display the original image.
7. The **generate\_salt\_and\_pepper\_noise** method is called with a threshold value of 0.05 to add salt and pepper noise to the image. The noisy image is assigned to the variable **Noise\_img**.
8. The **median\_filter** method is called with the **Noise\_img** and a kernel size of 3 to apply the median filter. The filtered image is assigned to the variable **Filtered\_img**.
9. Matplotlib is used to display the noisy image and the filtered image separately. **plt.imshow** is used to display the images, **plt.title** sets the titles of the plots, and **plt.colorbar** adds a colorbar to the plots. The noisy image and filtered image are displayed one by one using **plt.show**.

The result is the display of the noisy image and the filtered image, allowing comparison between the two and demonstrating the effectiveness of the impulse noise filter.



Noise (Gaussian):

This code represents a GUI application for applying a Gaussian noise filter to an input image. Here's how it works:

1. The necessary modules and libraries are imported, including Tkinter for the GUI, PIL for image processing, NumPy for array manipulation, and OpenCV for computer vision operations.
2. The **Enhance\_Image** class is defined, which represents the main application window. It takes the **root** window as a parameter.
3. The GUI elements are created and positioned within the application window using the **Label**, **Button**, and **Entry** widgets from Tkinter.
4. The **Load\_Image** method is called when the "Load Image" button is clicked. It opens a file dialog to select an image file and loads the image using PIL. The loaded image is resized and displayed in the application window.
5. The **Apply\_Noise** method is called when the "Apply Filter" button is clicked. It applies the Gaussian noise filter to the loaded image. The standard deviation (**sd**) value is obtained from the user-entered entry field. The method iterates over each pixel of the image and creates a 3x3 kernel around it. It calculates the Gaussian value based on the standard deviation and generates noise using the **random.gauss** function. The pixel value in the kernel is updated with the generated noise. The filtered image is then displayed in the application window.
6. The **ResetWindow** method is called when the "Reset" button is clicked. It destroys the current window and opens a new instance of the application.
7. The **root** window is created, and an instance of the **Enhance\_Image** class is created, passing the **root** window as a parameter. The application window is displayed by entering the Tkinter event loop using **root.mainloop()**.

Note: The code assumes that there is a file named "Gaussian.py" in the same directory, which is opened when the "Reset" button is clicked.

Overall, this code provides a simple GUI application for loading an image, applying a Gaussian noise filter, and displaying the filtered image using Tkinter and PIL.



Noise (Uniform):

This code represents a GUI application for applying uniform noise to an input image. Here's how it works:

1. The necessary modules and libraries are imported, including Tkinter for the GUI, PIL for image processing, NumPy for array manipulation, and OpenCV for computer vision operations.
2. The **Enhance\_Image** class is defined, which represents the main application window. It takes the **root** window as a parameter.
3. The GUI elements are created and positioned within the application window using the **Label**, **Button**, and **Entry** widgets from Tkinter.
4. The **Load\_Image** method is called when the "Load Image" button is clicked. It opens a file dialog to select an image file and loads the image using PIL. The loaded image is resized and displayed in the application window.
5. The **Apply\_Noise** method is called when the "Apply Filter" button is clicked. It applies uniform noise to the loaded image. The range of noise values (**RangeA** and **RangeB**) is obtained from the user-entered entry fields. The method iterates over each pixel of the image and creates a 3x3 kernel around it. If the pixel value falls within the specified range, a random value within that range is generated as noise and assigned to the central pixel of the kernel. The filtered image is then displayed in the application window.
6. The **ResetWindow** method is called when the "Reset" button is clicked. It destroys the current window and opens a new instance of the application.
7. The **root** window is created, and an instance of the **Enhance\_Image** class is created, passing the **root** window as a parameter. The application window is displayed by entering the Tkinter event loop using **root.mainloop()**.

this code provides a simple GUI application for loading an image, applying uniform noise within a specified range, and displaying the filtered image using Tkinter and PIL.

Frequency Domain (Histogram Equalization):

This code performs histogram equalization on a grayscale image using OpenCV, NumPy, and Matplotlib. Here's how it works:

1. The **Main** class is defined, which represents the main processing logic. It takes the path of the image file as a parameter when initialized.
2. The **Read\_Image** method reads the image using **cv2.imread()** with the **cv2.IMREAD\_GRAYSCALE** flag, which loads the image as grayscale. The image is stored in the **Image\_List** variable and displayed using the **Show\_Image** method.
3. The **Histogram\_Equalization** method performs histogram equalization on the loaded image. It first initializes an array **Nk** with 256 elements, each representing the count of pixels with a specific intensity level (0-255) in the image. This count is obtained by iterating over each pixel of the image and incrementing the corresponding element in **Nk**.
4. The cumulative distribution function (CDF) is calculated by summing up the counts in **Nk**. The CDF represents the accumulated distribution of pixel intensities in the image.
5. The CDF is then normalized to span the full range of pixel intensities (0-255). This is achieved by dividing each CDF value by the total number of pixels in the image and multiplying it by 255.
6. A new image **Equalized\_Image** is created with the same dimensions as the original image. Each pixel value in **Equalized\_Image** is replaced with its corresponding normalized CDF value obtained from **CDF\_Normalized**.
7. The **Equalized\_Image** is converted to the **uint8** data type, which is the expected data type for grayscale images, using **np.uint8()**.
8. The **Show\_Image** method is called to display the original image and the equalized image using Matplotlib's **imshow()** function. The title for each image is specified, and the images are displayed without axis labels.
9. The path of the image file is assigned to the **Path** variable, and an instance of the **Main** class is created, passing the path as a parameter.
10. The **Read\_Image** method is called to read and display the original image.
11. The **Histogram\_Equalization** method is called to perform histogram equalization and display the equalized image.

This code demonstrates how to perform histogram equalization on a grayscale image, enhancing its contrast and improving its visual appearance.

A close-up of a tower

Description automatically generated with low confidence

Frequency Domain (Histogram Specification):

This code performs histogram specification on an input image. Histogram specification is a technique used to modify the histogram of an image to match a desired histogram.

Here's a breakdown of the code:

1. The **Image** class is defined, which takes the path of the input image as a parameter in its constructor.
2. The **show\_image** method displays the original image before any modification.
3. The **cal\_tot** method calculates the total number of pixels in the image.
4. The **intensity\_dist** method computes the intensity distribution of the image by counting the occurrences of each intensity level.
5. The **show\_intensity\_dist** method displays a plot showing the intensity distribution of the image.
6. The **unifrom\_dist** method creates a uniform distribution with equal probabilities for each intensity level.
7. The **histogram\_specification** method performs the histogram specification. It first calculates the cumulative distribution function (CDF) of the input image's histogram and normalizes it. Then it calculates the CDF of the desired histogram (uniform distribution) and normalizes it as well.
8. The intensities of the input image are mapped to the desired histogram using the CDFs. The closest intensity index in the desired CDF is found for each intensity value in the input image using **np.argmin(np.abs(desired\_cdf\_normalized - input\_cdf\_normalized[intensity]))**.
9. The method returns the matched image as an 8-bit unsigned integer array.
10. An instance of the **Image** class is created with the path of the input image.
11. The **show\_image** method is called to display the original image.
12. The **cal\_tot** method is called to calculate the total number of pixels.
13. The **intensity\_dist** method is called to compute the intensity distribution.
14. The **unifrom\_dist** method is called to create the desired uniform distribution.
15. The **histogram\_specification** method is called to perform the histogram specification and obtain the matched image.
16. The matched image is displayed using **plt.imshow**.

A close-up of a tower

Description automatically generated with low confidence

Frequency Domain (Forward Fourier and Inverse Fourier):

This code performs the forward Fourier transform and inverse Fourier transform on an image using the NumPy and OpenCV libraries. The resulting magnitude spectrum and the reconstructed image after inverse Fourier transform are displayed using matplotlib.

Here is a breakdown of the code:

1. The code defines a class named **Main** that takes a path to an image as input.
2. The **Read\_Image** method reads the image from the specified path and stores it in grayscale format.
3. The **Fourier** method performs the forward Fourier transform on the image using nested loops. It calculates the complex-valued Fourier coefficients for each frequency component.
4. The **Shift** method shifts the Fourier coefficients to rearrange the frequency components. This is done to center the low-frequency components.
5. The **Magnitude\_Spectrum** variable computes the logarithm of the magnitude of the shifted Fourier coefficients, which represents the magnitude spectrum.
6. The **Inverse\_Fourier** method performs the inverse Fourier transform on the shifted Fourier coefficients using nested loops. It calculates the sum of complex exponentials for each spatial position to reconstruct the image.
7. The **Magnitude** variable computes the absolute value of the inverse Fourier-transformed coefficients.
8. The **Magnitude\_Normalized** variable normalizes the magnitude values between 0 and 1.
9. The **Inverse\_Fourier\_Uint8** variable converts the normalized magnitude values to a range of 0-255 to obtain a grayscale image.
10. The **Show\_Image** method displays the original image, the magnitude spectrum, and the reconstructed image side by side using matplotlib.

To use this code, you need to provide the path to an image file in the **Path** variable. After executing the code, it will display the original image, the magnitude spectrum, and the reconstructed image.

A picture containing text, sketch, black and white, building

Description automatically generatedA picture containing text, black

Description automatically generated

Frequency Domain (Interpolation):

This code is for performing image interpolation using different methods, such as nearest neighbor interpolation, bilinear interpolation, and bicubic interpolation. Here's a breakdown of the code:

1. The code imports the necessary libraries: **cv2** for image processing, **numpy** for numerical computations, and **matplotlib** for plotting.
2. The **Image** class is defined, which takes a path to an image as input and initializes the **image** attribute by reading the image using OpenCV's **imread** function.
3. The **show\_image** method displays the original image using **cv2.imshow** and waits for a key press before closing the window.
4. The **cal\_tot** method calculates the total number of pixels in the image and stores it in the **tot\_pxls** attribute.
5. The **intensity\_dist** method computes the intensity distribution of the image. It counts the occurrences of each intensity value and calculates the corresponding probability.
6. The **show\_intensity\_dist** method plots the intensity distribution using **matplotlib.pyplot.plot** and displays the length of the intensity counts array, the counts themselves, and the probabilities.
7. The **nearest\_neighbor\_interpolation** method performs image scaling using the nearest neighbor interpolation technique. It takes a scale factor as input and computes the dimensions of the scaled image. Then, it iterates over each pixel of the scaled image and assigns the corresponding pixel value from the original image.
8. The **bilinear\_interpolation** method performs image scaling using bilinear interpolation. It calculates the dimensions of the scaled image and iterates over each pixel of the scaled image. For each pixel, it determines the four nearest pixels in the original image and computes the interpolated value based on the distances and intensities of these pixels.
9. The **bicubic\_interpolation** method performs image scaling using bicubic interpolation. It calculates the dimensions of the scaled image and iterates over each pixel of the scaled image. For each pixel, it calculates the coordinates in the original image and determines the surrounding 4x4 neighborhood. It applies bicubic interpolation to compute the interpolated value.
10. The **bicubic\_weight** method calculates the weight for bicubic interpolation based on the given input **x**.
11. An instance of the **Image** class is created with the path to the image file.
12. The original image is displayed using the **show\_image** method.
13. The total number of pixels is calculated using the **cal\_tot** method.
14. The intensity distribution is computed using the **intensity\_dist** method.
15. The intensity distribution is displayed using the **show\_intensity\_dist** method.
16. Image scaling is performed using one of the interpolation methods (nearest neighbor, bilinear, or bicubic). The scaled image is stored in the **scaled\_image** variable.
17. The scaled image is displayed using **cv2.imshow** and waits for a key press before closing the window.

Overall, this code demonstrates different methods of image interpolation and provides visualizations of the intensity distribution of the image.

A picture containing outdoor, sky, black and white, building

Description automatically generated