Image Processing Project Report

Amr Khaled Abdeltawab, ID: 224363
Ahmed Mohamed Mounir, ID: 224643
John Medhat, ID: 224343
Mohamed Mostafa, ID: 223911

May 21, 2024

Contents

Chapter 1

Introduction

This project implements a comprehensive suite of image processing algorithms using Python, complemented by a graphical user interface (GUI) built with Tkinter. The primary objective is to provide a robust toolset for various image processing tasks, including blurring filters, noise addition, edge detection, histogram equalization, interpolation, and coding techniques such as Huffman and Golomb coding. This report details the implemented algorithms, the mathematical foundations underpinning them, and the corresponding code.

In the early stages of the project, the team members, Amr Khaled Abdeltawab, Ahmed Mohamed Mounir, John Medhat, and Mohamed Mostafa, dedicated significant time to studying the mathematical principles underlying each algorithm. This foundational understanding was crucial for effectively translating mathematical concepts into functional code.

To begin, the team conducted a thorough literature review, exploring textbooks, academic papers, and online resources to gain a deep understanding of the mathematical models used in image processing. The study included: - **Blurring Filters**: Understanding convolution operations and kernel functions for averaging, Gaussian, median, min, and max filters. - **Noise Addition**: Exploring statistical models, particularly Gaussian noise and salt-and-pepper noise, and their effects on image data. - **Edge Detection**: Delving into gradient-based methods such as the Sobel, Laplacian, and Roberts Cross operators to detect edges by identifying intensity changes. - **Histogram Equalization**: Learning about cumulative distribution functions (CDFs) and their role in enhancing image contrast. - **Interpolation**: Studying nearest neighbor interpolation to effectively resize images while maintaining visual quality. - **Coding Techniques**: Investigating lossless compression methods like Huffman and Golomb coding, focusing on their algorithms and efficiency.

Regular meetings were held to discuss findings, share insights, and collaboratively solve challenges. This collaborative approach ensured a well-rounded understanding of the concepts and fostered a supportive environment for learning and problem-solving.

The process of converting mathematical equations into code required careful attention to detail and iterative testing. Python, with its rich set of libraries such as NumPy and SciPy, provided the necessary tools for implementing complex mathematical operations. Tkinter was chosen for the GUI due to its simplicity and ease of integration with Python.

In summary, this project not only enhanced the team's technical skills in Python programming and GUI development but also deepened their understanding of the mathematical foundations of image processing. The resulting application serves as a practical demonstration of these concepts, providing users with a powerful tool for exploring and applying various image processing techniques.

Chapter 2

Algorithms and Mathematics

2.1 Blurring Filters

Blurring filters are used to smooth images by reducing noise and detail. These filters are often employed to preprocess images for further analysis or to achieve a specific visual effect.

2.1.1 Blur Filter

A blur filter smooths the image by averaging the pixels within a neighborhood.

Mathematical Representation:

$$I'(x, y) = \frac{1}{N} \sum_{(i,j) \in \mathcal{N}} I(x + i, y + j)$$

where $\mathcal{N}$ is the neighborhood of pixel (x, y) and N is the number of pixels in the neighborhood.

Code:

```
def blur_image(image):
    image_array = np.array(image)
    if len(image_array.shape) == 2:
        image_array = np.stack((image_array,) * 3, axis=-1)

    kernel_size = 5
    kernel = np.ones((kernel_size, kernel_size)) / kernel_size ** 2
    blurred_image_array = np.copy(image_array)

    pad_size = kernel_size // 2
    padded_image = np.pad(image_array, ((pad_size, pad_size),
    (pad_size, pad_size), (0, 0)), mode='edge')

    for i in range(image_array.shape[0]):
        for j in range(image_array.shape[1]):
            for k in range(3):
                blurred_image_array[i, j, k] = np.sum(padded_image[i:i +
                kernel_size, j:j + kernel_size, k] * kernel)
    return Image.fromarray(blurred_image_array.astype('uint8'))
```

2.1.2 Gaussian Filter

A Gaussian filter smooths the image using a Gaussian function, reducing noise and detail.

Mathematical Representation:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Code:

```
def gaussian_filter(image, sigma):
    image_array = np.array(image.convert('L'))
    gaussian_blur = GaussianBlur(image_array)
    kernel = gaussian_blur.gaussian_kernel(sigma)
    blurred_image_array = gaussian_blur.apply_blur(kernel)
    return Image.fromarray(blurred_image_array)
```

2.1.3 Median Filter

A median filter is used to reduce noise in an image by replacing each pixel value with the median value of the intensities in its neighborhood.

Mathematical Representation:

$$I'(x, y) = \text{median}\{I(x + i, y + j) \,|\, (i, j) \in \mathcal{N}\}$$

Code:

```
def median_filter(image, window_size):
    image_array = np.array(image.convert('L'))
    filtered_image_array = np.zeros_like(image_array)

    half_window = window_size // 2
    height, width = image_array.shape

    for y in range(height):
        for x in range(width):
            y1 = max(y - half_window, 0)
            y2 = min(y + half_window + 1, height)
            x1 = max(x - half_window, 0)
            x2 = min(x + half_window + 1, width)
            neighborhood = image_array[y1:y2, x1:x2]
            median_value = np.median(neighborhood)
            filtered_image_array[y, x] = median_value

    return Image.fromarray(filtered_image_array)
```

2.1.4 Min Filter

A min filter replaces each pixel value with the minimum value of the intensities in its neighborhood.

Mathematical Representation:

$$I'(x, y) = \min\{I(x + i, y + j) \,|\, (i, j) \in \mathcal{N}\}$$

Code:

```
def min_filter(image, window_size):
    width, height = image.size
    filtered_image = Image.new("L", (width, height))
    half_window = window_size // 2
    for y in range(height):
        for x in range(width):
            window = []
            for dy in range(-half_window, half_window + 1):
                for dx in range(-half_window, half_window + 1):
                    px = min(max(x + dx, 0), width - 1)
                    py = min(max(y + dy, 0), height - 1)
                    window.append(image.getpixel((px, py)))
            min_pixel = min(window)
            filtered_image.putpixel((x, y), int(min_pixel))
    return filtered_image
```

2.1.5 Max Filter

A max filter replaces each pixel value with the maximum value of the intensities in its neighborhood.

Mathematical Representation:

$$I'(x, y) = \max\{I(x + i, y + j) \,|\, (i, j) \in \mathcal{N}\}$$

Code:

```
def max_filter(image, window_size):
    width, height = image.size
    filtered_image = Image.new("L", (width, height))
    half_window = window_size // 2
    for y in range(height):
        for x in range(width):
            window = []
            for dy in range(-half_window, half_window + 1):
                for dx in range(-half_window, half_window + 1):
                    px = min(max(x + dx, 0), width - 1)
                    py = min(max(y + dy, 0), height - 1)
                    window.append(image.getpixel((px, py)))
            max_pixel = max(window)
            filtered_image.putpixel((x, y), int(max_pixel))
    return filtered_image
```

2.2 Noise Addition

Noise addition techniques simulate different types of noise in the image.

2.2.1 Gaussian Noise

Gaussian noise is added to the image with a specified mean and variance.

Mathematical Representation:

$$N(x,y) = I(x,y) + \mathcal{N}(0,\sigma^2)$$

where $\mathcal{N}(0,\sigma^2)$ is a Gaussian noise with mean 0 and variance σ^2.

Code:

```
def add_gaussian_noise(image, mean=0, variance=0.01):
    noise = np.random.normal(mean, math.sqrt(variance), image.shape)
    noisy_image = image + noise * 255
    return np.clip(noisy_image, 0, 255).astype('uint8')
```

2.2.2 Salt and Pepper Noise

Salt and pepper noise randomly changes some of the pixels to black or white.

Mathematical Representation:

$$N(x,y) = \begin{cases} 0 & \text{with probability } p/2 \\ 255 & \text{with probability } p/2 \\ I(x,y) & \text{with probability } 1-p \end{cases}$$

Code:

```
def salt_and_pepper_noise(image, prob):
    output = image.copy()
    probs = np.random.random(output.shape[:2])
    output[probs < (prob / 2)] = 0
    output[probs > 1 - (prob / 2)] = 255
    return output
```

2.3 Edge Detection

Edge detection is a fundamental tool in image processing used to identify points in a digital image at which the image brightness changes sharply.

2.3.1 Sobel Operator

The Sobel operator is used for edge detection by calculating the gradient of the image intensity.

Mathematical Representation: The Sobel operator uses two 3x3 kernels which are convolved with the original image to calculate the gradient magnitude and direction. The kernels are defined as:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The gradient magnitude and direction are then calculated as:

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

Code:

```
def sobel_operator(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 = image.shape
    gradient_magnitude = np.zeros((rows, cols))
    gradient_direction = np.zeros((rows, cols))

    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            region = image[i - 1:i + 2, j - 1:j + 2]
            gx = np.sum(Gx * region)
            gy = np.sum(Gy * region)
            gradient_magnitude[i, j] = np.sqrt(gx ** 2 + gy ** 2)
            gradient_direction[i, j] = np.arctan2(gy, gx)

    return gradient_magnitude, gradient_direction
```

2.3.2 Laplacian Operator

The Laplacian operator is used to find areas of rapid change in images, which often correspond to edges.

Mathematical Representation: The Laplacian operator uses the following kernel:

$$L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

The Laplacian image is obtained by convolving this kernel with the image.

Code:

```
def laplacian_operator(image):
    image_array = np.array(image.convert('L'))
    kernel = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
    laplacian_image_array = np.copy(image_array)

    pad_size = kernel.shape[0] // 2
    padded_image = np.pad(image_array, pad_size, mode='edge')

    for i in range(image_array.shape[0]):
        for j in range(image_array.shape[1]):
            laplacian_image_array[i, j]
            = np.sum(padded_image[i:i + kernel.shape[0], j:j
            + kernel.shape[1]] * kernel)

    laplacian_image_array = np.clip(laplacian_image_array, 0, 255)
    .astype('uint8')
    return Image.fromarray(laplacian_image_array)
```

2.3.3 Roberts Cross-Gradient Operators

The Roberts Cross-Gradient Operators are used to perform edge detection by calculating the gradient of the image intensity using diagonal differences.

Mathematical Representation: The Roberts Cross operator uses the following kernels:

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

The gradient magnitude is then calculated as:

$$G = \sqrt{G_x^2 + G_y^2}$$

Code:

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

    image = image.astype(np.int32)  # Convert to int32 to avoid overflow
    rows, cols = image.shape
    gradient_magnitude = np.zeros((rows, cols))

    for i in range(rows - 1):
        for j in range(cols - 1):
            gx = image[i, j] - image[i + 1, j + 1]
            gy = image[i + 1, j] - image[i, j + 1]
            gradient_magnitude[i, j] = np.sqrt(gx ** 2 + gy ** 2)

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

2.4 Histogram Equalization

Histogram equalization enhances the contrast of the image by spreading out the most frequent intensity values.

Mathematical Representation: The cumulative distribution function (CDF) is calculated as:

$$CDF(i) = \sum_{j=0}^{i} \frac{h(j)}{n}$$

where h is the histogram and n is the total number of pixels. The new pixel value is given by:

$$I'(x, y) = \text{round}(CDF(I(x, y)) \times (L - 1))$$

Code:

```
def histogram_equalization(image):
    histogram = np.zeros(256, dtype=int)
    for value in image.flatten():
        histogram[value] += 1

    cdf = np.cumsum(histogram).astype(float)
    cdf = (cdf - cdf.min()) * 255 / (cdf.max() - cdf.min())
    cdf = cdf.astype('uint8')

    equalized_image = cdf[image]
    return equalized_image
```

2.4.1 Histogram Specification

Histogram specification, also known as histogram matching, transforms the pixel intensity values of the source image to match the histogram of the reference image.

Mathematical Representation: Given a source image with histogram h_s and a reference image with histogram h_r, the transformation function T can be defined as:

$$T(x) = F_r^{-1}(F_s(x))$$

where F_s and F_r are the CDFs of the source and reference images respectively.

Code:

```
def histogram_specification(source_img, reference_img):
    source = source_img.flatten()
    reference = reference_img.flatten()

    source_hist, bins = np.histogram(source, bins=256, range=[0, 256]
    , density=True)
    source_cdf = np.cumsum(source_hist) * 255
    source_cdf = (source_cdf - source_cdf.min()) / (source_cdf.max()
    - source_cdf.min())

    reference_hist, bins = np.histogram(reference, bins=256, range=[0, 256]
    , density=True)
    reference_cdf = np.cumsum(reference_hist) * 255
    reference_cdf = (reference_cdf - reference_cdf.min())
    / (reference_cdf.max() - reference_cdf.min())

    lookup_table = np.zeros(256)
    g_j = 0
    for g_i in range(256):
        while g_j < 256 and source_cdf[g_i] > reference_cdf[g_j]:
            g_j += 1
        lookup_table[g_i] = g_j

    source = np.clip(source, 0, 255).astype(np.uint8)
    specified_img = lookup_table[source].reshape(source_img.shape)
    .astype(np.uint8)
    return specified_img
```

2.5 Interpolation

Interpolation is used to resize the image.

2.5.1 Nearest Neighbor Interpolation

In nearest neighbor interpolation, the value of each pixel in the resized image is assigned the value of the nearest pixel in the original image.

Mathematical Representation:

$$I'(x,y) = I(\text{round}(x \cdot \frac{M}{M'}), \text{round}(y \cdot \frac{N}{N'}))$$

Code:

```
def nearest_neighbor_interpolation(image, new_width, new_height):
    orig_height, orig_width = image.shape[:2]
    row_ratio = orig_height / new_height
    col_ratio = orig_width / new_width
    new_image = np.zeros((new_height, new_width, image.shape[2]),
    dtype=image.dtype)

    for new_row in range(new_height):
        for new_col in range(new_width):
            orig_row = int(new_row * row_ratio)
            orig_col = int(new_col * col_ratio)
            new_image[new_row, new_col] = image[orig_row, orig_col]

    return new_image
```

2.6 Coding Techniques

Coding techniques are used to compress the image data.

2.6.1 Huffman Coding

Huffman coding is a lossless data compression algorithm that assigns variable-length codes to input characters based on their frequencies.

Mathematical Representation: The frequency of each pixel value is used to build a binary tree with the shortest codes assigned to the most frequent values.

Code:

```
class HuffmanNode:
    def __init__(self, char, freq):
        self.freq = freq
        self.char = char
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(frequencies):
    heap = [HuffmanNode(char, freq) for char, freq in frequencies.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        merged = HuffmanNode(None, node1.freq + node2.freq)
        merged.left = node1
        merged.right = node2
        heapq.heappush(heap, merged)

    return heap[0]

def generate_codes(node, prefix="", codebook={}):
    if node:
        if node.char is not None:
            codebook[node.char] = prefix
        generate_codes(node.left, prefix + "0", codebook)
        generate_codes(node.right, prefix + "1", codebook)
    return codebook

def huffman_encode(image, codebook):
    flat_image = image.flatten()
    encoded_image = ''.join(codebook[pixel] for pixel in flat_image)
    return encoded_image
```

```
def huffman_decode(encoded_image, huffman_tree, shape):
    current_node = huffman_tree
    decoded_image = []

    for bit in encoded_image:
        current_node = current_node.left if bit == '0' else current_node.right

        if current_node.char is not None:
            decoded_image.append(current_node.char)
            current_node = huffman_tree

    return np.array(decoded_image).reshape(shape)
```

2.6.2 Golomb Coding

Golomb coding is a lossless data compression method that is particularly effective for geometric distributions.

Mathematical Representation: For a given parameter m, a value x is encoded as:

$$q = \left\lfloor \frac{x}{m} \right\rfloor \quad \text{and} \quad r = x \mod m$$

The encoded value is a unary representation of q followed by the binary representation of r.

Code:

```
def golomb_encode(image_array, m):
    flat_image = image_array.flatten()
    encoded_image = []

    for value in flat_image:
        q = value // m
        r = value % m
        unary = '1' * q + '0'
        binary = bin(r)[2:]
        if len(binary) < math.ceil(math.log2(m)):
            binary = '0' * (math.ceil(math.log2(m)) - len(binary)) + binary
        encoded_image.append(unary + binary)

    return ''.join(encoded_image)

def golomb_decode(encoded_image, m, shape):
    decoded_image = []
    i = 0
    while i < len(encoded_image):
        q = 0
        while encoded_image[i] == '1':
            q += 1
            i += 1
        i += 1
        r_bits = int(math.ceil(math.log2(m)))
        r = int(encoded_image[i:i + r_bits], 2)
        i += r_bits

        decoded_value = q * m + r
        decoded_image.append(decoded_value)

    return np.array(decoded_image).reshape(shape)
```

2.7 Enhancing Filters

Enhancing filters are used to improve the quality of the image by enhancing certain features.

2.7.1 Unsharp Masking

Unsharp masking enhances the edges of the image by subtracting a blurred version of the image from the original image.

Mathematical Representation:

$$I' = I + k \cdot (I - G_\sigma * I)$$

where $G_\sigma * I$ is the Gaussian-blurred image and k is a scaling factor.

Code:

```
def unsharp_masking(image, kernel_size=5, sigma=1, amount=1.0):
    kernel = self.gaussian_kernel(kernel_size, sigma)
    if image.ndim == 2:  # Grayscale image
        blurred = self.apply_filter(image, kernel)
        unsharp_mask = image - blurred
        enhanced_image = image + amount * unsharp_mask
    elif image.ndim == 3:  # RGB image
        enhanced_image = np.zeros_like(image)
        for i in range(3):  # Apply filter to each channel
            blurred = self.apply_filter(image[:, :, i], kernel)
            unsharp_mask = image[:, :, i] - blurred
            enhanced_image[:, :, i] = image[:, :, i] + amount * unsharp_mask
    return np.clip(enhanced_image, 0, 255).astype(np.uint8)
```

2.7.2 Highboost Filtering

Highboost filtering is a generalized version of unsharp masking that allows for more control over the amount of enhancement.

Mathematical Representation:

$$I' = I + k \cdot (I - G_\sigma * I)$$

where $G_\sigma * I$ is the Gaussian-blurred image and k is a scaling factor greater than 1.

Code:

```
def highboost_filtering(image, kernel_size=5, sigma=1, k=1.5):
    kernel = self.gaussian_kernel(kernel_size, sigma)
    if image.ndim == 2:  # Grayscale image
        blurred = self.apply_filter(image, kernel)
        unsharp_mask = image - blurred
        highboost_image = image + k * unsharp_mask
    elif image.ndim == 3:  # RGB image
        highboost_image = np.zeros_like(image)
        for i in range(3):  # Apply filter to each channel
            blurred = self.apply_filter(image[:, :, i], kernel)
            unsharp_mask = image[:, :, i] - blurred
            highboost_image[:, :, i] = image[:, :, i] + k * unsharp_mask
    return np.clip(highboost_image, 0, 255).astype(np.uint8)
```

2.7.3 DFT Filter

The Discrete Fourier Transform (DFT) filter is used to filter images in the frequency domain.

Mathematical Representation: The 2D DFT of an image I is given by:

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} I(x,y) \cdot e^{-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}$$

The inverse DFT is given by:

$$I(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) \cdot e^{j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}$$

Code:

```
def dft2d(image):
    M, N = image.shape
    u = np.arange(M)
    v = np.arange(N)
    x = np.arange(M)
    y = np.arange(N)

    U, X = np.meshgrid(u, x)
    V, Y = np.meshgrid(v, y)

    exp_factor_u = np.exp(-2j * np.pi * U * X / M)
    exp_factor_v = np.exp(-2j * np.pi * V * Y / N)

    F = np.dot(exp_factor_u, np.dot(image, exp_factor_v))

    return F

def idft2d(F):
    M, N = F.shape
    u = np.arange(M)
    v = np.arange(N)
    x = np.arange(M)
    y = np.arange(N)

    U, X = np.meshgrid(u, x)
    V, Y = np.meshgrid(v, y)

    exp_factor_u = np.exp(2j * np.pi * U * X / M)
    exp_factor_v = np.exp(2j * np.pi * V * Y / N)

    image = np.dot(exp_factor_u, np.dot(F, exp_factor_v)) / (M * N)

    return image

def high_pass_filter(F, cutoff=30):
    M, N = F.shape
    U, V = np.meshgrid(np.arange(M), np.arange(N), indexing='ij')
    D = np.sqrt((U - M/2)**2 + (V - N/2)**2)
    H = D > cutoff
    return F * H

def apply_highpass_filter(image):
    if self.image_path:
        if self.mode_var.get() == "Grayscale":
            image_array = np.array(self.grayscale_image(self.original_image))
        else:
            image_array = np.array(self.grayscale_image(self.original_image))

        F = self.dft2d(image_array)
        F_filtered = self.high_pass_filter(F)
        reconstructed_image = self.idft2d(F_filtered)
        reconstructed_image = np.real(reconstructed_image)

        self.edited_image = Image.fromarray(reconstructed_image.astype('uint8'))
        self.display_image(self.edited_image)

        # Display the original and filtered images in the frequency domain
        self.display_frequency_domain(F, F_filtered)
    else:
        messagebox.showerror("Error", "No image loaded.")

def display_frequency_domain(F, F_filtered):
    magnitude_spectrum = np.log(np.abs(F) + 1)
    magnitude_spectrum_filtered = np.log(np.abs(F_filtered) + 1)

    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.title('Original Frequency Domain')
    plt.imshow(np.fft.fftshift(magnitude_spectrum), cmap='gray')
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.title('Filtered Frequency Domain')
    plt.imshow(np.fft.fftshift(magnitude_spectrum_filtered), cmap='gray')
    plt.axis('off')

    plt.show()
```

Chapter 3

Conclusion

This project demonstrates various image processing techniques implemented using Python, providing a practical tool for image manipulation through a user-friendly graphical user interface. The implemented algorithms cover essential areas of image processing, including blurring filters, noise addition, edge detection, histogram equalization, interpolation, enhancing filters, and coding techniques.

Each algorithm is supported by mathematical formulations that underpin their functionality, ensuring accuracy and efficiency. This foundation not only validates the implementations but also offers insights into the processes of each technique.

The project includes: - **Blurring Filters**: Techniques like blur, Gaussian, median, min, and max filters to smooth images and reduce noise. - **Noise Addition**: Methods to simulate Gaussian and salt-and-pepper noise, testing the robustness of image processing algorithms. - **Edge Detection**: Operators such as Sobel, Laplacian, and Roberts Cross-Gradient to identify edges in images. - **Histogram Equalization**: Enhancing image contrast for better detail visibility. - **Interpolation**: Nearest neighbor interpolation for resizing images while maintaining quality. - **Enhancing Filters**: Unsharp masking and highboost filtering to enhance image details. - **Coding Techniques**: Huffman and Golomb coding for compressing image data. - **DFT Filtering**: Using Discrete Fourier Transform for high-pass filtering in the frequency domain.

The collaborative efforts of team members Amr Khaled Abdeltawab, Ahmed Mohamed Mounir, John Medhat, and Mohamed Mostafa were crucial in this project's success. Each member's dedication to understanding the mathematical principles and their meticulous implementation ensured the project's accuracy and functionality.

In conclusion, this project showcases practical applications of image processing techniques and highlights the importance of a strong mathematical foundation. The resulting application serves as a valuable tool for both educational and practical purposes, offering a comprehensive suite of image processing functionalities.

Chapter 4

Team Members

- Amr Khaled Abdeltawab, ID: 224363
- Ahmed Mohamed Mounir, ID: 224643
- John Medhat, ID: 224343
- Mohamed Mostafa, ID: 223911