Homework 2: Image Enhancement & Image Restoration

Part I. Implementation

Gamma correction

▼ Screenshot

```python
"""
TODO Part 1: Gamma correction
"""
def gamma_correction(img, gamma):
    # Convert the image to float32
    img = img.astype(np.float32)

    # Normalize the image to 0-1 range
    img_normalized = img / 255.0

    # Apply gamma correction
    corrected = np.power(img_normalized, gamma)

    # Scale back to 0-255 range and convert to uint8
    corrected = (corrected * 255).astype(np.uint8)

    return corrected
```

- The gamma correction formula is: $g(x, y) = f(x, y)^{\gamma}$
- Where
  - $g(x, y)$ is the output pixel value
  - $f(x, y)$ is the input pixel value (normalized to [0,1])
  - γ is the correction factor
    - When $\gamma < 1$: The image becomes brighter
    - When $\gamma > 1$: The image becomes darker

Histogram Equalization

▼ Screenshot

```
"""
TODO Part 2: Histogram equalization
"""
def histogram_equalization(img, mode):
    # Convert to grayscale if the image is colored
    if mode == "BGR":
        B, G, R = cv2.split(img)

        B_hist = histogram(B.astype(np.float32))
        G_hist = histogram(G.astype(np.float32))
        R_hist = histogram(R.astype(np.float32))

        B_eq = transformation(B, B_hist, B.shape[0] * B.shape[1]).astype(np.uint8)
        G_eq = transformation(G, G_hist, G.shape[0] * G.shape[1]).astype(np.uint8)
        R_eq = transformation(R, R_hist, R.shape[0] * R.shape[1]).astype(np.uint8)

        equalized =  cv2.merge((B_eq, G_eq, R_eq))

    elif mode == "HSV":
        H, S, V = cv2.split(cv2.cvtColor(img, cv2.COLOR_BGR2HSV))
        V_hist = histogram(V.astype(np.float32))
        V_eq = transformation(V, V_hist, V.shape[0] * V.shape[1]).astype(np.uint8)
        equalized = cv2.cvtColor(cv2.merge((H, S, V_eq)), cv2.COLOR_HSV2BGR)

    elif mode == "LAB":
        L, A, B = cv2.split(cv2.cvtColor(img, cv2.COLOR_BGR2LAB))
        L_hist = histogram(L.astype(np.float32))
        L_eq = transformation(L, L_hist, L.shape[0] * L.shape[1]).astype(np.uint8)
        equalized = cv2.cvtColor(cv2.merge((L_eq, A, B)), cv2.COLOR_LAB2BGR)

    return equalized

def histogram(image):
    hist = np.zeros(256)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            hist[int(image[i,j])] += 1
    return hist

def transformation(image, histogram, n):
    cdf = np.zeros(256)
    for i in range(256):
        for j in range(i):
            cdf[i] += histogram[j]
        cdf[i] = 255 * cdf[i] / n
    return cdf[image]
```

- **Mode BGR**
    1. **Separate Channels**

       If the image is in BGR format, the function splits the image into its Blue, Green, and Red channels for later calculation.
    2. **Histogram Calculation**

       For each channel (B, G, R), it calculates the histogram using a helper function `histogram`.
    3. **Equalization**

       It applies a transformation to each channel of the image by using the calculated histogram to equalize the intensity distribution. This process involves first calculating the CDF (Cumulative Distribution Function) of the given histogram and then applying the CDF to the input image.
    4. **Merge Channels**

       The equalized channels are merged back into a single image.
- **Mode HSV and LAB**
    1. **Convert to HSV or LAB**

       The image is converted from BGR to HSV or LAB color space.

2. **Histogram Calculation**

   The histogram is calculated for the V (value) channel for HSV color space and L (lightness) channel for LAB color space.

3. **Equalization**

   Only the V channel and L channel is equalized using the transformation function.

4. **Merge and Convert Back**

   The equalized V channel and L channel is merged with the original H and S channels, and the image is converted back to BGR.

Motion Blur PSF Generation

▼ Screenshot

```
"""
TODO Part 1: Motion blur PSF generation
"""
def generate_motion_blur_psf(size=51, angle=-45, len=40):
    # Create a zero array for PSF
    psf = np.zeros((size, size))

    # Convert angle to radians
    angle_rad = np.deg2rad(angle)

    # Calculate center point
    center = size // 2

    # Calculate cos and sin of angle
    cos_angle = np.cos(angle_rad)
    sin_angle = np.sin(angle_rad)

    # Generate the motion blur line
    for i in range(len):
        # Calculate offset from center
        offset = i - (len // 2)

        # Calculate x and y coordinates
        x = center + int(offset * cos_angle)
        y = center + int(offset * sin_angle)

        # Ensure we're within bounds
        if 0 <= x < size and 0 <= y < size:
            psf[y, x] = 1

    # Normalize PSF so sum equals 1
    psf = psf / np.sum(psf)

    return psf
```

1. **Function Parameters**:
   - `size`: The size of the PSF matrix.
   - `angle`: The angle of the motion blur in degrees.
   - `len`: The length of the motion blur.
2. **Initialize PSF**: Create a zero matrix of size `size x size`.
3. **Convert Angle:** Convert the motion blur angle from degrees to radians.
4. **Calculate Center**: Determine the center of the PSF matrix.

5. **Generate Line**: Use a loop to create a line in the PSF matrix based on the specified length and angle, ensuring coordinates are within bounds.
6. **Normalize PSF**: Normalize the PSF matrix so that its sum equals 1.
7. **Return PSF**: Return the normalized PSF matrix.

Wiener Filter

▼ Screenshot

```python
"""
TODO Part 2: Wiener filtering
"""
def wiener_filtering(img_blurred, psf, K=0.01):
    # Convert image to float for FFT
    img_blurred = img_blurred.astype(float)

    # Handle multi-channel (color) images
    restored_channels = []
    for channel in cv2.split(img_blurred):
        height, width = channel.shape
        psf_padded = np.zeros((height, width))
        psf_height, psf_width = psf.shape
        psf_padded[:psf_height, :psf_width] = psf

        # Center PSF
        psf_padded = np.roll(psf_padded, -(psf_height//2), axis=0)
        psf_padded = np.roll(psf_padded, -(psf_width//2), axis=1)

        # Convert to frequency domain
        img_freq = np.fft.fft2(channel)
        psf_freq = np.fft.fft2(psf_padded)

        # Wiener filter formula
        psf_freq_conj = np.conj(psf_freq)
        denominator = np.abs(psf_freq)**2 + K

        # Apply Wiener filter
        restored_freq = img_freq * psf_freq_conj / denominator

        # Convert back to spatial domain
        restored = np.real(np.fft.ifft2(restored_freq))
        restored_channels.append(restored)
    restored = cv2.merge(restored_channels)

    # Clip values to valid range and convert back to uint8
    restored = np.clip(restored, 0, 255).astype(np.uint8)

    return restored
```

1. **Convert Image**

   Convert the blurred image to a float type for frequency domain processing.
2. **Separate Channels**

   Split the image into its color channels for individual processing.

3. **Prepare PSF**
    - Pad the PSF to match the image dimensions.
    - Center the PSF in the padded array.
4. **Frequency Domain Conversion**

    Convert both the image channel and the padded PSF to the frequency domain using FFT (Fast Fourier Transform).
5. **Wiener Filter Application**:
    - Compute the conjugate of the PSF in the frequency domain.
    - Apply the Wiener filter formula to restore the image in the frequency domain.
    - Wiener filter formula

$$H(u,v) = \frac{H^*_{PSF}(u,v)}{|H_{PSF}(u,v)|^2 + K}$$

$$F(u,v) = H(u,v) \cdot G(u,v)$$

        - $H_{PSF}(u,v)$ is the Fourier transform of the PSF.
        - $H^*_{PSF}(u,v)$ is the complex conjugate of $H_{PSF}(u,v)$.
        - K is a constant that represents the noise-to-signal power ratio.
        - $H(u,v)$ is the CLS filter.
        - $G(u,v)$ is the Fourier transform of the blurred image.
        - $F(u,v)$ is the Fourier transform of original image.
6. **Inverse FFT**

    Convert the restored frequency domain image back to the spatial domain.
7. **Merge Channels**

    Combine the processed channels back into a single image.
8. **Clip and Convert**: Ensure pixel values are within the valid range (0-255) and convert the image back to uint8.

CLS Filter

▼ Screenshot

```
"""
TODO Part 3: Constrained least squares filtering
"""
def constrained_least_square_filtering(img_blurred, psf, gamma=0.001):
    # Convert image to float
    img_blurred = img_blurred.astype(float)

    # Handle multi-channel (color) images
    restored_channels = []
    for channel in cv2.split(img_blurred):
        # Pad PSF to match image size
        height, width = channel.shape
        psf_padded = np.zeros((height, width))
        psf_height, psf_width = psf.shape
        psf_padded[:psf_height, :psf_width] = psf

        # Center PSF
        psf_padded = np.roll(psf_padded, -(psf_height//2), axis=0)
        psf_padded = np.roll(psf_padded, -(psf_width//2), axis=1)

        # Create Laplacian operator for regularization
        ksize = 5
        laplacian = laplacian_kernel(ksize)
        lap_padded = np.zeros((height, width))
        lap_padded[:ksize, :ksize] = laplacian

        # Convert to frequency domain
        img_freq = np.fft.fft2(channel)
        psf_freq = np.fft.fft2(psf_padded)
        lap_freq = np.fft.fft2(lap_padded)

        # Wiener filter formula
        psf_freq_conj = np.conj(psf_freq)
        denominator = np.abs(psf_freq)**2 + gamma * np.abs(lap_freq)**2
        denominator = np.where(denominator == 0, 1e-6, denominator)

        # Apply filter
        restored_freq = img_freq * psf_freq_conj / denominator

        # Convert back to spatial domain
        restored = np.real(np.fft.ifft2(restored_freq))
        restored_channels.append(restored)

    restored = cv2.merge(restored_channels)

    # Clip values and convert back to uint8
    restored = np.clip(restored, 0, 255).astype(np.uint8)

    return restored
```

```python
def laplacian_kernel(size):
    """
    Generate a Laplacian kernel of specified size
    Args:
        size: Kernel size (must be odd)
    Returns:
        Normalized Laplacian kernel
    """
    if size % 2 == 0:
        raise ValueError("Kernel size must be odd")

    # Create base kernel
    kernel = np.zeros((size, size))
    center = size // 2

    # Fill the kernel
    for i in range(size):
        for j in range(size):
            # Calculate Manhattan distance from center
            distance = abs(i - center) + abs(j - center)

            if distance == 0:  # Center point
                kernel[i, j] = 4
            elif distance == 1:  # Direct neighbors
                kernel[i, j] = -1
            # All other points remain 0

    return kernel
```

1. **Convert Image**

   Convert the blurred image to a float type for processing.

2. **Separate Channels**

   Split the image into its color channels for individual processing.

3. **Prepare PSF**

   - Pad the PSF to match the image dimensions.
   - Center the PSF in the padded array.

4. **Create Laplacian Operator**

   Generate a Laplacian kernel for regularization and pad it to match the image size.

5. **Frequency Domain Conversion**

   Convert the image channel, padded PSF, and padded Laplacian to the frequency domain using FFT.

6. **Apply Constrained Least Squares Filter**:

   - Compute the conjugate of the PSF in the frequency domain.
   - Use the constrained least squares formula to restore the image in the frequency domain, incorporating the Laplacian for regularization.
   - CLS formula

$$H(u,v) = \frac{H^*_{PSF}(u,v)}{|H_{PSF}(u,v)|^2 + \gamma|L(u,v)|^2}$$

$$F(u,v) = H(u,v) \cdot G(u,v)$$

     - $L(u,v)$ is the Fourier transform of the Laplacian operator, used for regularization.
     - $H_{PSF}(u,v)$ is the Fourier transform of the PSF.
     - $H^*_{PSF}(u,v)$ is the complex conjugate of $H_{PSF}(u,v)$.
     - $H(u,v)$ is the CLS filter.
     - $G(u,v)$ is the Fourier transform of the blurred image.
     - $F(u,v)$ is the Fourier transform of original image.

7. **Inverse FFT**

   Convert the restored frequency domain image back to the spatial domain.

8. **Merge Channels**

   Combine the processed channels back into a single image.

9. **Clip and Convert**

   Ensure pixel values are within the valid range (0-255) and convert the image back to uint8.

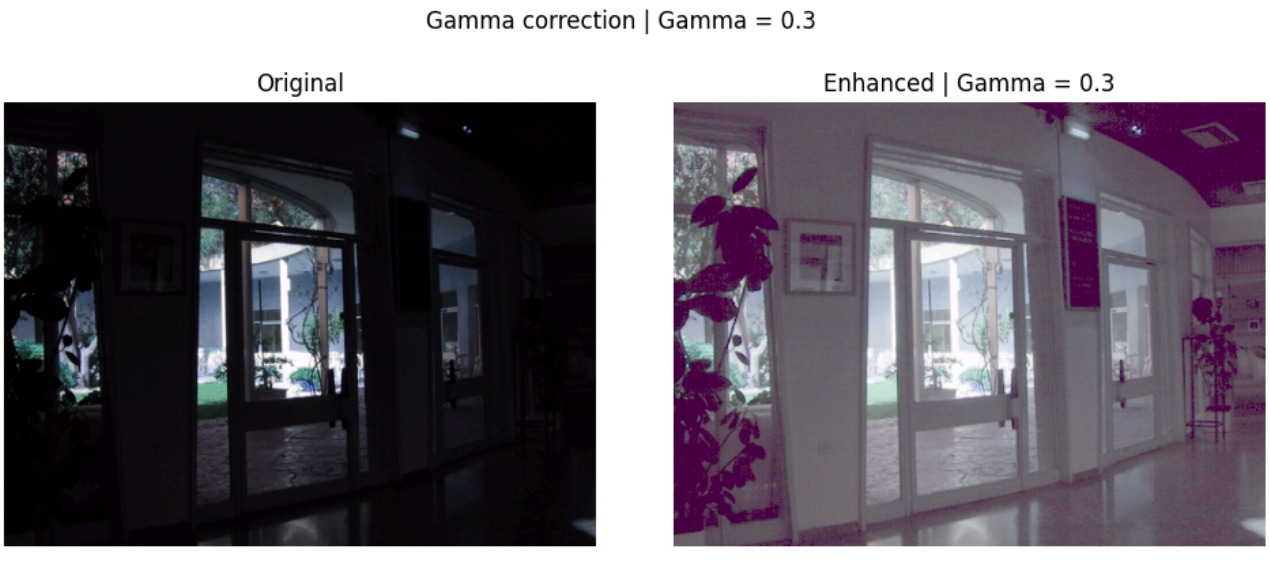
10. **Return Restored Image**

Part II. Results & Analysis

Task 1: Image Enhancement

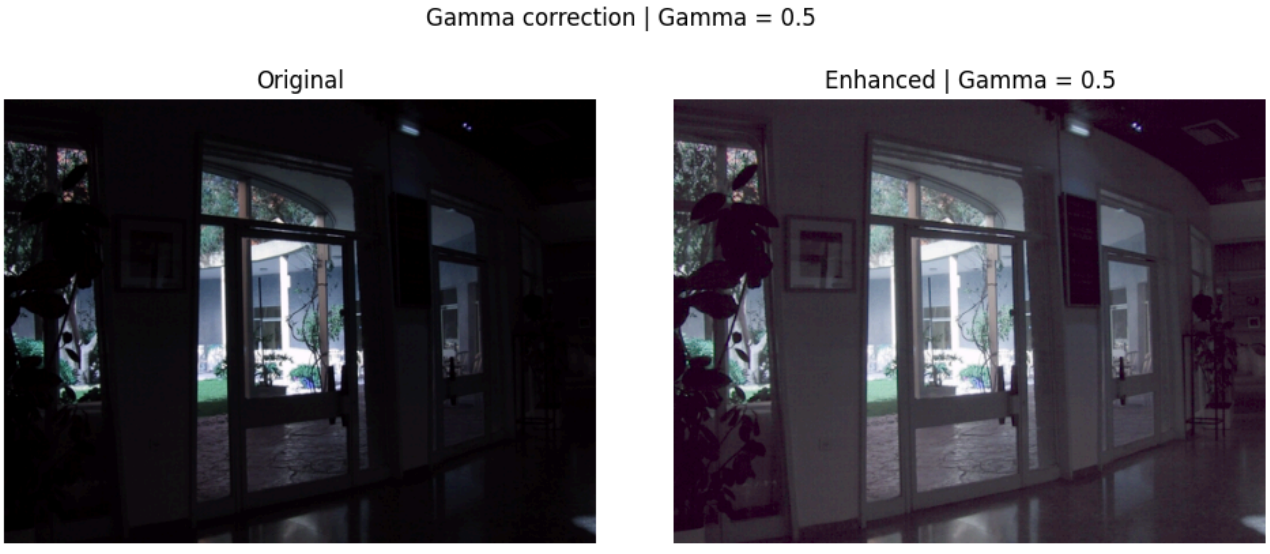
Gamma Correction

- Gamma = 0.3



  - **Brightness**: Significantly increased.
  - **Details**: Dark areas are much more visible, revealing hidden details.
  - **Color**: The enhancement process might disrupt the original white balance, resulting in an unnatural appearance.

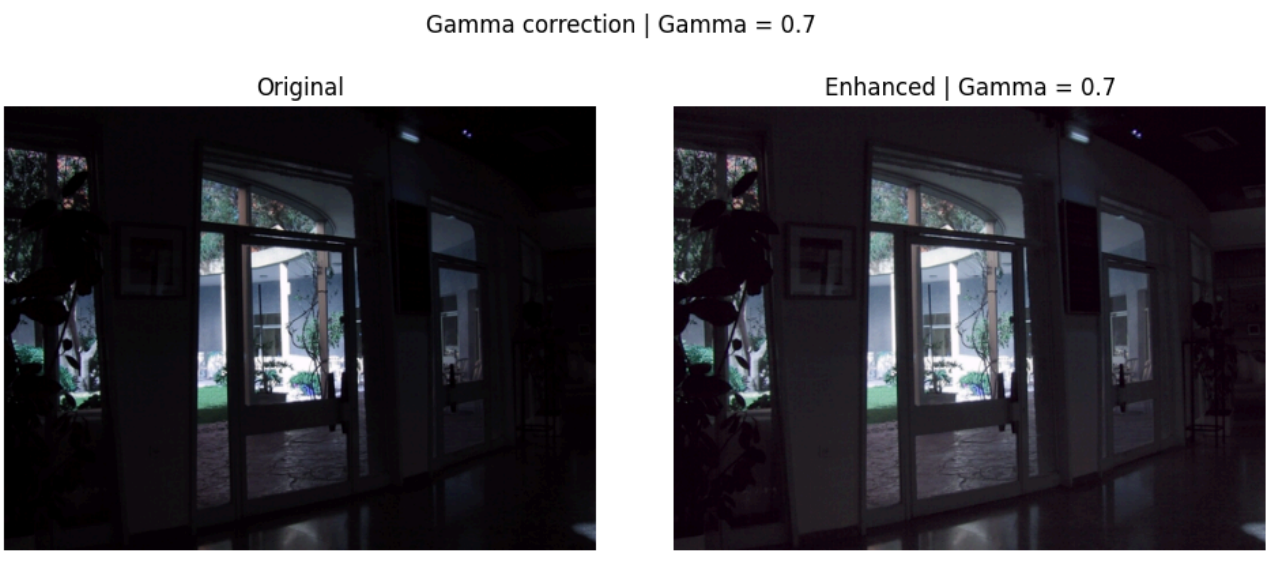
- Gamma = 0.5



  - **Brightness**: Moderately increased.
  - **Details**: Shadows are lifted, providing better visibility without overwhelming brightness.

    - **Color**: The enhancement maintains a more natural appearance, with minimal disruption to the original white balance.

- Gamma = 0.7



    - **Brightness**: Slightly increased.
    - **Details**: Subtle enhancement in darker regions, preserving the original contrast.
    - **Color**: The enhancement process preserves the original white balance, resulting in a natural appearance with minimal color shift.

Overall Comparison

Gamma correction affects image brightness, detail visibility, and color balance. A lower gamma value, like 0.3, significantly brightens the image, revealing hidden details but potentially disrupting the original white balance. A moderate gamma of 0.5 balances brightness and detail enhancement while maintaining natural colors. A higher gamma, such as 0.7, subtly increases brightness, preserving contrast and color balance, ideal for slight enhancements without altering the original appearance. The choice of gamma depends on the desired level of enhancement and color preservation.

Histogram Equalization

- Equalization over each channel of RGB Color Space

    - **Brightness**: Increased across all channels, leading to a more uniform enhancement.
    - **Details**: Improved visibility in shadows, but with potential color distortion.
    - **Color**: Colors may appear washed out or overly bright, affecting the natural look.

- Equalization over only V channel of HSV Color Space

- **Brightness**: Significantly increased, especially in the value (V) channel.
- **Details**: Dark areas are much more visible, revealing hidden details.
- **Color**: The enhancement may lead to a shift in hue and saturation, resulting in a slightly unnatural appearance.

Comparison

- **RGB**: Provides a more uniform enhancement but may lead to color distortion and less natural results.
- **HSV**: Offers more targeted enhancement by focusing on brightness, but can alter color balance.

Overall, the choice between HSV and BGR depends on whether preserving color balance or achieving maximum brightness is the priority.

Bonus: Clipped Histogram Equalization

▼ Screenshot

```python
"""
TODO Bonus: Clipped histogram equalization
"""
def contrast_enhancement(img, mode):
    if mode == "HSV":
        lab = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
        H, S, V = cv2.split(lab)
        V_eq = clip_histogram_equalization(V, 6.0)
        limg = cv2.merge((H, S, V_eq))
        enhanced_image = cv2.cvtColor(limg, cv2.COLOR_HSV2BGR)

    elif mode == "LAB":
        lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
        L, A, B = cv2.split(lab)
        L_eq = clip_histogram_equalization(L, 6.0)
        limg = cv2.merge((L_eq, A, B))
        enhanced_image = cv2.cvtColor(limg, cv2.COLOR_LAB2BGR)

    return enhanced_image

def clip_histogram_equalization(image, clipLimit):
    # Get image dimensions and calculate histogram
    h, w = image.shape
    hist = histogram(image.astype(np.float32))

    # Clip the histogram
    clipLimit = int(clipLimit * image.size / 256)
    if clipLimit > 0:
        excess = hist - clipLimit
        excess[excess < 0] = 0
        clipped_hist = hist - excess
        redistribute = excess.sum() // 256
        clipped_hist += redistribute

    # Apply histogram equalization using clipped histogram
    clahe_image = transformation(image, clipped_hist, h * w).astype(np.uint8)

    return clahe_image
```

Explanation of Methods

- **Purpose**

  Enhances contrast using clipped histogram equalization in either HSV or LAB color spaces.

- **Process**:

  1. Converts the image to the specified color space (HSV or LAB).
  2. Splits the image into its respective channels.
  3. Applies clipped histogram equalization to the luminance or value channel.
     - Calculates the histogram of the image.
     - Clips the histogram at a specified limit to prevent over-enhancement.
     - Redistributes the clipped values across the histogram.
     - Applies the transformation to enhance contrast.
  4. Merges the channels back and converts the image to BGR.

Result

Clipped histogram equalization | HSV Color Space

Original

Enhanced

Clipped histogram equalization | LAB Color Space

Original

Enhanced

Compared to Standard Histogram Equalization

- **Noise Reduction**: Clipping limits the amplification of noise, resulting in a cleaner image.
- **Overexposure Prevention**: Reduces the risk of overexposure in bright areas by controlling the histogram spread.
- **Natural Appearance**: Maintains a more balanced look by preventing excessive contrast enhancement.

Overall, clipped histogram equalization offers a balanced approach, enhancing contrast while minimizing artifacts and preserving image quality.

Compare the above three methods

- **Gamma Correction**
  - **Brightness Control**: Adjusts brightness non-linearly using a gamma value.
  - **Color Preservation**: Maintains color balance well, with minimal shifts.
  - **Use Case**: Ideal for uniformly dark or bright images needing overall brightness adjustment.
- **Standard Histogram Equalization**
  - **Contrast Enhancement**: Redistributes pixel intensities to enhance contrast.
  - **Color Shifts**: Can cause color distortion, especially in color images.
  - **Artifacts**: May lead to overexposure and noise amplification.
  - **Use Case**: Best for images with poor dynamic range needing significant contrast improvement.
- **Clipped Histogram Equalization**
  - **Controlled Contrast**: Enhances contrast while clipping limits noise and overexposure.
  - **Color Balance**: Better preserves natural appearance compared to standard histogram equalization.

    - **Artifacts**: Reduces noise amplification and prevents excessive contrast.
    - **Use Case**: Suitable for enhancing contrast with minimal artifacts, maintaining a balanced look.
- **Overall Comparison**
    - Gamma Correction is best for brightness adjustments with minimal color impact.
    - Standard Histogram Equalization offers strong contrast enhancement but can introduce artifacts.
    - Clipped Histogram Equalization provides a balanced approach, enhancing contrast while minimizing noise and preserving color balance.

Task 2: Image Restoration

Minimum Mean Square Error (Wiener) Filtering

Testcase Result

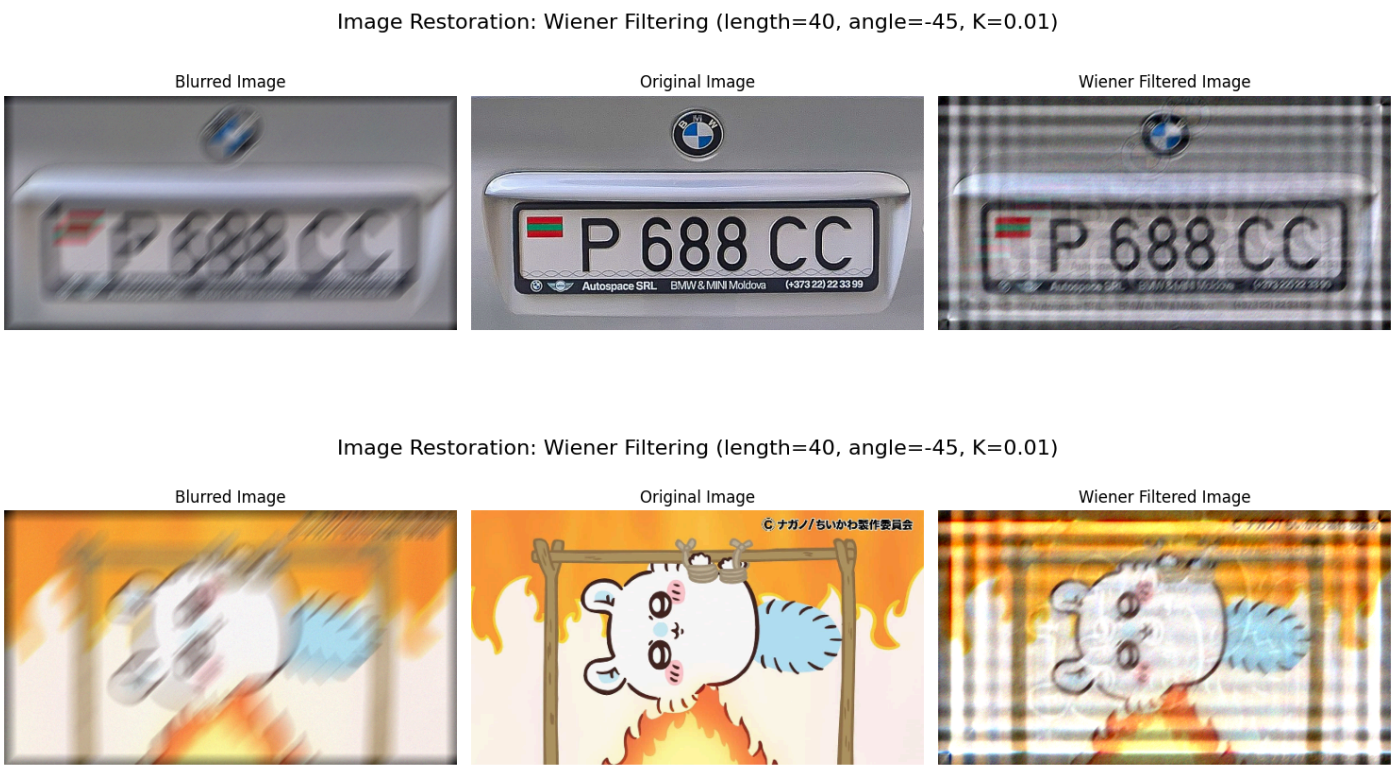


Image 1: License Plate

- **Improvement**: The text becomes more legible, and some details are restored.
- **Artifacts**: Noticeable ringing artifacts and grid-like patterns appear, which can be a side effect of the filtering process.

Image 2: Cartoon Character

- **Improvement**: Some details are recovered, and the character is more defined.
- **Artifacts**: Similar to the first image, grid-like patterns and ringing artifacts are present, affecting the overall quality.

Overall Analysis

- **Effectiveness**: Wiener filtering helps in recovering some details from blurred images, making them more recognizable.
- **Artifacts**: The presence of ringing and grid patterns indicates limitations in the filter's ability to perfectly restore images, especially when the blur is severe or the PSF is not perfectly matched.
- **Use Case**: Best for moderate blur where some artifacts are acceptable in exchange for improved detail visibility. Adjusting parameters like the noise-to-signal ratio (K) might help reduce artifacts.

Constrained Least Squares Restoration

Testcase Result

Image 1: License Plate

- **Improvement**: Some details are restored, and the text is more legible.
- **Artifacts**: Ringing and grid-like patterns are present, similar to Wiener filtering, but potentially more pronounced due to the regularization

Image 2: Cartoon Character

- **Improvement**: Some details are recovered, and the character is more defined.
- **Artifacts**: Noticeable grid patterns and ringing, affecting the overall quality.

Overall Analysis

- **Effectiveness**: CLS filtering helps recover details from blurred images, similar to Wiener filtering, but with a focus on regularization to control noise.
- **Artifacts**: The presence of ringing and grid patterns indicates challenges in perfectly restoring images, especially with severe blur or mismatched PSF.
- **Use Case**: Suitable for moderate blur where regularization helps balance detail recovery and noise suppression. Adjusting the gamma parameter can help manage artifacts.

Bonus: Lucy-Richardson Deconvolution

▼ Screenshot

```
"""
TODO Bonus: Other restoration algorithm
"""
def lucy_richardson_deconvolution(blurred_image, psf, num_iterations=100):
    image = blurred_image.astype(np.float32) / 255.0

    restored_channels = []
    for channel in cv2.split(image):
        estimate = channel.copy()
        psf_hat = np.flip(np.flip(psf, 0), 1)

        # Lucy-Richardson Iteration
        for _ in range(num_iterations):
            conv = cv2.filter2D(estimate, -1, psf)
            relative_blur = channel / (conv + 1e-10)
            correction = cv2.filter2D(relative_blur, -1, psf_hat)
            estimate *= correction

        restored_channels.append(estimate)

    restored = cv2.merge(restored_channels)
    restored = np.clip(restored * 255, 0, 255).astype(np.uint8)
    return restored
```

Explanation of Methods

- **Purpose**: An iterative method for deblurring images, particularly effective for motion blur and out-of-focus blur.
- **Process**:
  1. **Initialization**: Convert the blurred image to a float and normalize it.
  2. **PSF Preparation**: Flip the PSF to create psf_hat for the correction step.
  3. **Iteration**: For a specified number of iterations:
     a. Convolve the current estimate with the PSF.
     b. Calculate the relative blur by dividing the original channel by the convolved result.
     c. Convolve the relative blur with psf_hat to get the correction factor.
     d. Update the estimate by multiplying it with the correction factor.
  4. **Finalization**: Merge the processed channels, scale back to 0-255, and convert to uint8.

Improvements in Image Quality

- **Detail Recovery**: Lucy-Richardson can recover finer details due to its iterative nature, which refines the estimate progressively.
- **Artifacts**: Generally produces fewer ringing artifacts compared to Wiener filtering, as it adapts the estimate in each iteration.
- **Noise Handling**: Better at handling noise without introducing excessive artifacts, due to its iterative correction process.
- **Control**: Offers more control over the restoration process through the number of iterations, allowing for fine-tuning based on the blur severity.

Overall, Lucy-Richardson deconvolution provides a robust approach to image restoration, offering improved detail recovery and reduced artifacts compared to Wiener and CLS filtering, especially when the number of iterations is appropriately chosen.

Testcase Result

Compare the above three methods

Test Case 1: License Plate

Method	Detail Recovery	Artifacts
Wiener Filtering	Provides the clearest text and detail recovery.	Some ringing and grid patterns, but overall clarity is high.
CLS Filtering	Least clear, with moderate text legibility.	Pronounced ringing and grid patterns
Lucy-Richardson Deconvolution	Good recovery of text details, slightly less clear than Wiener.	Fewer artifacts compared to the other methods

Test Case 2: Cartoon Character

Method	Detail Recovery	Artifacts
Wiener Filtering	Best clarity in character definition.	Present but less impactful on overall clarity.
CLS Filtering	Least clear, with moderate character definition.	Noticeable grid patterns and ringing
Lucy-Richardson Deconvolution	Good detail and edge restoration, slightly less clear than Wiener.	Minimal artifacts, providing a cleaner result

Overall Comparison

- **Wiener Filtering**: Offers the clearest detail recovery, making it the best choice for high clarity.
- **CLS Filtering**: Least effective in detail clarity, with more pronounced artifacts.
- **Lucy-Richardson Deconvolution**: Provides good detail with minimal artifacts, slightly less clear than Wiener.

Wiener filtering excels in detail clarity, followed by Lucy-Richardson, with CLS filtering being the least clear.

Part III. Answer the questions

1. Please describe a problem you encountered and how you solved it.

Problem

While implementing image restoration using Wiener filtering, I encountered significant ringing artifacts, especially around sharp edges in the image. This was due to the mismatch between the estimated point spread function (PSF) and the actual blur characteristics.

Solution

To address this, I refined the PSF estimation by experimenting with different lengths and angles that better matched the observed blur. Additionally, I adjusted the noise-to-signal ratio parameter (K) to find a balance that minimized artifacts while maintaining detail recovery.

2. What potential limitations might arise when using Minimum Mean Square Error (Wiener) Filtering for image restorations?

Limitations:

- Wiener filtering can produce ringing and grid patterns, particularly when the PSF estimation is inaccurate.
- The filter may amplify noise when the noise-to-signal ratio is incorrectly calibrated.

Solutions:

- Use advanced techniques such as blind deconvolution to achieve more accurate blur estimation.
- Fine-tune the K value to achieve optimal balance between noise reduction and detail preservation.
- Apply denoising techniques (e.g., median filtering or bilateral filtering) after Wiener filtering to reduce noise.

3. What potential limitations might arise when using Constrained Least Squares Restoration for image restorations?

Limitations:

- Ringing and grid patterns can be pronounced, especially with high regularization parameters.
- The choice of regularization parameter (gamma) can significantly affect the outcome, requiring careful tuning.

Solutions:

- Adjust the gamma parameter to balance between smoothness and detail preservation, reducing artifacts.
- Combine CLS with other methods, such as pre-filtering or post-processing, to enhance results and reduce artifacts.