Yaman Salman 11926532
Sama Khraim 11820435

Image Processing - Assignment

Part 1: Piecewise linear transformation

Write a MATLAB function ***piecewise_linear_transform*** which computes an intensity transformed image for a given image and two additional parameters. The parameters are comprised of a set of input intensity values and a corresponding set of output intensity values, where the first intensity value is 0 and the last intensity value is 255 in both sets. The resulting intensity values in the output image are calculated from a piecewise linear transformation determined from the parameters. Then call the function ***piecewise_linear_transform*** for the ***airplane.png*** image with input intensities {0,100,130,255} and output intensities {0,70,200,255} to calculate the intensity transformed image. Display the resulting output. Briefly discuss the qualitative differences between the input and output images.

```
function piecewise_linear_transform(input_image, input_intensities,
    output_intensities)

    % Read and preprocess the image
    origin_Image = imread(input_image);
    origin_Image = im2double(origin_Image);
    [row, col] = size(origin_Image);
    num_segments = numel(input_intensities) - 1;
    image_stretch = zeros(row, col); % Initialize the stretched image

    for i = 1:num_segments
      lower_threshold = input_intensities(i);
      upper_threshold = input_intensities(i + 1);
      slope = (output_intensities(i + 1) - output_intensities(i)) / (upper_threshold
    - lower_threshold);

      % Apply the appropriate transformation based on the slope
      image_stretch(origin_Image >= lower_threshold & origin_Image <=
    upper_threshold) = slope * (origin_Image(origin_Image >= lower_threshold &
    origin_Image <= upper_threshold) - lower_threshold) + output_intensities(i);
    end
```

```matlab
    % Display the original and stretched images
    figure;
    subplot(1, 2, 1), imshow(origin_Image), title('Original Image');
    subplot(1, 2, 2), imshow(image_stretch), title('Transformed Image');

    % Plotting the piecewise linear function
    input_range = linspace(0, 1, 256);
    output_range = zeros(size(input_range));

    for i = 1:num_segments
      input_mask = (input_range >= input_intensities(i)/255) & (input_range <=
    input_intensities(i+1)/255);
      output_range(input_mask) = (input_range(input_mask) -
    input_intensities(i)/255) * (output_intensities(i+1)/255 -
    output_intensities(i)/255) / (input_intensities(i+1)/255 -
    input_intensities(i)/255) + output_intensities(i)/255;
    end

    % Plotting the piecewise linear transformation function
    figure;
    subplot(1, 2, 1), plot(input_range, output_range);
    title('Piecewise Linear Transformation Function');
    xlabel('Input Intensity');
    ylabel('Output Intensity');

    % Plotting the input-output intensities
    subplot(1, 2, 2), plot(input_intensities, output_intensities, 'o-');
    title('Input-Output Intensities');
    xlabel('Input Intensity');
    ylabel('Output Intensity');
end
```

Command Window:

```
piecewise_linear_transform('airplane.png', [0, 100, 130, 255], [0, 70, 200, 255]);
```

Part 2: Spatial Image Restoration
Write a MATLAB function function *adaptive_local_noise_reduction* that computes an estimate of the original image *f^(x,y)* given a noisy image g(x,y), the overall noise variance σ_η^2, and the filter window size using the adaptive expression

$$\hat{f}(x,y) = g(x,y) - \frac{\sigma_\eta^2}{\sigma_{S_{xy}}^2}\left(g(x,y) - \bar{z}_{S_{xy}}\right)$$

and enforce the assumption $\sigma_\eta^2 \leq \sigma_{S_{xy}}^2$, where the local mean $\bar{z}_{S_{xy}}$ and local variance $\sigma_{S_{xy}}^2$ are calculated over the filter window centered at (x,y). Apply Gaussian noise with variance using the function MATLAB function ***imnoise*** for the ***camera.pmg*** image. Then, call the *adaptive_local_noise_reduction* with the noisy image and a window size of 7x7. Display the input image, noisy image, and restored image. Briefly discuss how well the restored image approximates the original image and whether or not a local noise reduction filter is suitable for applying to this type of noise. What are the tradeoffs of using adaptive local noise reduction versus other noise reduction techniques such as an arithmetic mean filter? What are the advantages and disadvantages of using adaptive local noise reduction?

Function:

```
function adaptive_local_noise_reduction(input_image_path, noise_variance, window_size)

    % Load the original image
    original_image = imread(input_image_path);
    original_image = im2double(original_image);

    % Add Gaussian noise to the original image
    noisy_image = imnoise(original_image, 'gaussian', 0, noise_variance);

    % Get image dimensions
    [m, n] = size(noisy_image);


    % Initialize restored image
    restored_image = zeros(m, n);

    % Calculate window half size
    window_half = floor(window_size / 2);

    % Create a figure for plotting
```

```
Figure;

% Display the original image
subplot(1, 3, 1);
imshow(original_image);
title('Original Image');

% Display the noisy image with added Gaussian noise
subplot(1, 3, 2);
imshow(noisy_image);
title('Noisy Image with Gaussian Noise');

% Process each pixel
for i = 1:m
  for j = 1:n

        % Define the limits of the filter window
        x_min = max(i - window_half, 1);
        x_max = min(i + window_half, m);
        y_min = max(j - window_half, 1);
        y_max = min(j + window_half, n);

        % Extract the local window from the noisy image
        local_window = noisy_image(x_min:x_max, y_min:y_max);

        % Calculate the local mean and local variance
        local_mean = mean(local_window(:));
        local_var = var(local_window(:));

        % Calculate the adaptive expression
        alpha = max(0, 1 - noise_variance / local_var);


        % Calculate the restored pixel value
        restored_pixel = local_mean + alpha * (noisy_image(i, j) - local_mean);
        restored_image(i, j) = restored_pixel;
  end
end

% Display the restored image
```

```
    subplot(1, 3, 3);
    imshow(restored_image);
    title('Restored Image');
    sgtitle('Adaptive Local Noise Reduction');
end
```

Command Window:

input_image_path=('camera.png');
noise_variance=0.02;
window_size=7;
adaptive_local_noise_reduction('camera.png', 0.02, 7);

Part 3: Inverse Filtering

The degraded image *d(x,y)* is the result of convolving an input image with the degradation function

$$h(x,y) = \frac{1}{239}\begin{bmatrix} 3 & 7 & 7 & 7 & 8 & 9 & 5 \\ 4 & 6 & 2 & 5 & 8 & 8 & 3 \\ 8 & 5 & 4 & 4 & 6 & 5 & 8 \\ 1 & 5 & 6 & 5 & 4 & 6 & 2 \\ 1 & 3 & 8 & 3 & 8 & 6 & 3 \\ 2 & 7 & 1 & 5 & 5 & 2 & 2 \\ 6 & 2 & 9 & 5 & 4 & 3 & 3 \end{bmatrix}$$

Write a MATLAB function ***inverse_filter*** that performs inverse filtering in the frequency domain. The function takes as input the degraded image *d(x,y)*, degradation function *h(x,y)* and a threshold value *t*, and returns the estimated image after inverse filtering. Once you obtain the estimated function $\hat{F}$ in the frequency domain, set values in $\hat{F}$ at coordinates *(u,v)* to 0, if the magnitude of *H* at the same coordinates *(u,v)* is less than the threshold value t. Then, map $\hat{F}$ to the estimated image $\hat{f}$ in the spatial domain. You can use the MATLAB function ***conv2*** for performing convolution and ***fft2, fftshift, ifft2*** functions for performing 2D Fourier transform related operations. Call the function ***inverse_filter*** with the ***yard.png*** image, the degradation function *h(x,y)* described above and a threshold $t = 0.01$. Display the degraded image and the estimated image after inverse filtering. Discuss how well the inverse filtering restored the original image. How well would inverse filtering work if noise was also applied to the image?

```
function output_image = inverse_filter(degraded_image, degradation_function, threshold)

    % Fourier transformation of the degraded image using fftshift and fft2
    degraded_image_freq = fftshift(fft2(degraded_image));
```

```
    % Fourier transformation of the degradation function using fftshift and fft2
      And cropping it so it fits the degraded image size
    degradation_function_freq = fftshift(fft2(degradation_function,
    size(degraded_image, 1), size(degraded_image, 2)));

    % Perform inverse filtering in the frequency domain by getting the
    estimated_image_freq = degraded_image_freq ./ degradation_function_freq;
    estimated_image_freq(abs(degradation_function_freq) < threshold) = 0;

    % Returning the image to the spatial domain and removing any imaginary values
    estimated_image = ifft2(ifftshift(estimated_image_freq));
    output_image = real(estimated_image);


    % Ensure the output image is within valid intensity range
    output_image = max(0, min(255, output_image));
end
```

Command Window:

```
% Initializing the degradation function
```

```
degradation_function = [
   3, 7, 7, 7, 8, 9, 5;
   4, 6, 2, 5, 8, 8, 3;
   8, 5, 4, 4, 6, 5, 8;
   1, 5, 6, 5, 4, 6, 2;
   1, 3, 8, 3, 8, 6, 3;
   2, 7, 1, 5, 5, 2, 2;
   6, 2, 9, 5, 4, 3, 3
] / 239;
```

```
% Reading the image and making sure its in grayscale
```

```
original_image = imread('yard.png');
original_image = rgb2gray(original_image);
```

```
% Degrading the image using conv2, reading the original image as double
, taking the degradation function, and using the 'same' keyword to
    specify the output of the image having the same size as the input

degraded_image = conv2(double(original_image), degradation_function, 'same');

% Initializing the threshold variable to 0.01 and calling the
    inverse_filter method

threshold = 0.01;
restored_image = inverse_filter(degraded_image, degradation_function, threshold);



% Creating a figure and plotting the original image and the results.

figure;

subplot(3, 3, 1); imshow(original_image, []); title('Original Image');
subplot(3, 3, 2); imshow(log(1 + abs(fftshift(fft2(original_image)))), []); title('Original Image
    Frequency Domain');

subplot(3, 3, 4); imshow(degraded_image, []); title('Degraded Image');
subplot(3, 3, 5); imshow(log(1 + abs(fftshift(fft2(degraded_image)))), []); title('Degraded Image
    Frequency Domain');

subplot(3, 3, 7); imshow(restored_image, []); title('Restored Estimated Image');
subplot(3, 3, 8); imshow(log(1 + abs(fftshift(fft2(restored_image)))), []); title('Restored Estimated
    Image Frequency Domain');
```

Part 4: Edge Detection

Write a MATLAB function ***edge_detection*** that takes an image as input and performs the following steps:

- Normalizes the image to [0,1] range
- Applies Gaussian smoothing with standard deviation 0.5
- Computes the gradient magnitude and angle images
- Applies nonmaximal suppression to the gradient magnitude image. For non-maximum suppression, discretize the angle into 8 directions (or bins), where each bin accounts for 45 degrees. For example, first bin would range from [-22.5,22.5] degrees, second bin would range from [22.5,67.5] degrees, and so on.
- Returns the gradient magnitude image, angle image and gradient image after non-maximal suppression

Call the function ***edge_detection*** function with the ***house.png*** image. Display the input image and all three output images from your ***edge_detection*** function. Briefly discuss the resulting images and how you might implement edge linking techniques using these outputs. Explain your reasoning.

```
function [gradient_magnitude, gradient_angle, gradient_suppressed] =
      edge_detection(image)

      % Normalize the image to [0, 1] range
      normalized_image = double(image) / 255.0;

      % Create a Gaussian kernel for smoothing
      sigma = 0.5;
      kernel_size = 5;
      half_size = (kernel_size - 1) / 2;
      [x, y] = meshgrid(-half_size:half_size, -half_size:half_size);
      gaussian_kernel = exp(-(x.^2 + y.^2) / (2 * sigma^2));
      gaussian_kernel = gaussian_kernel / sum(gaussian_kernel(:));

      % Apply Gaussian smoothing using convolution
      smoothed_image = conv2(normalized_image, gaussian_kernel, 'same');

      % Compute the gradient magnitude and angle images
      [gradient_x, gradient_y] = gradient(smoothed_image);
      gradient_magnitude = sqrt(gradient_x.^2 + gradient_y.^2);
      gradient_angle = atan2(gradient_y, gradient_x);

      % Discretize angles into 8 directions (8 bins)
      angle_bins = [-22.5, 22.5, 67.5, 112.5, 157.5, -157.5, -112.5, -67.5];

      % Apply non maximal suppression to the gradient magnitude image
      gradient_suppressed = zeros(size(gradient_magnitude));
      for i = 1:numel(angle_bins)
```

```
        angle_bin = angle_bins(i);
        mask = (gradient_angle >= (angle_bin - 22.5) * pi / 180) & (gradient_angle <
      angle_bin + 22.5) * pi / 180);
        (gradient_suppressed = max(gradient_suppressed, gradient_magnitude .* mask);
      end
end
```

Command Window:

input_image = imread('house.png');

[gradient_magnitude, gradient_angle, gradient_suppressed] = edge_detection(input_image);

subplot(2, 2, 1), imshow(input_image), title('Input Image');
subplot(2, 2, 2), imshow(gradient_magnitude), title('Gradient Magnitude');
subplot(2, 2, 3), imshow(gradient_angle, []), title('Gradient Angle');
subplot(2, 2, 4), imshow(gradient_suppressed), title('Gradient after Non-Maximal Suppression');