Image Processing Lab4

Name: 徐竣霆

Student ID: 110060012

Proj05-01: Noise Generators

Results

Fig.5.7 (a)

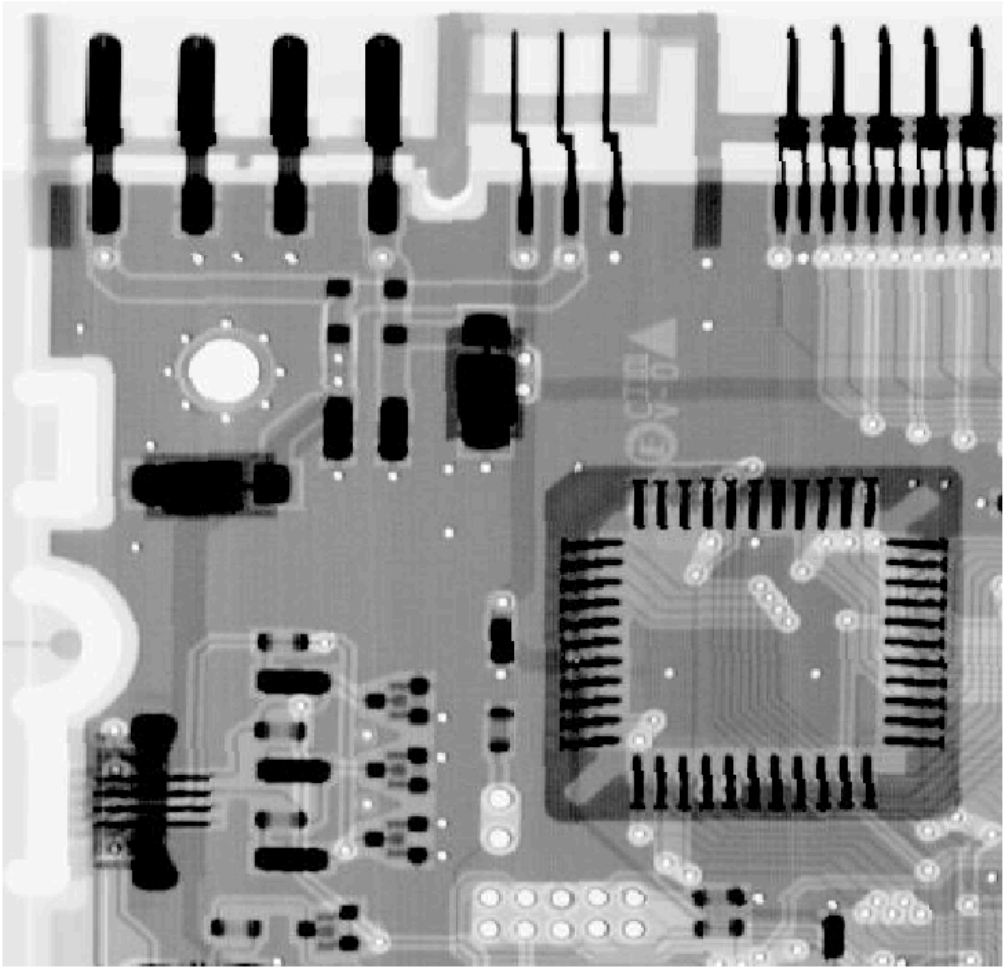
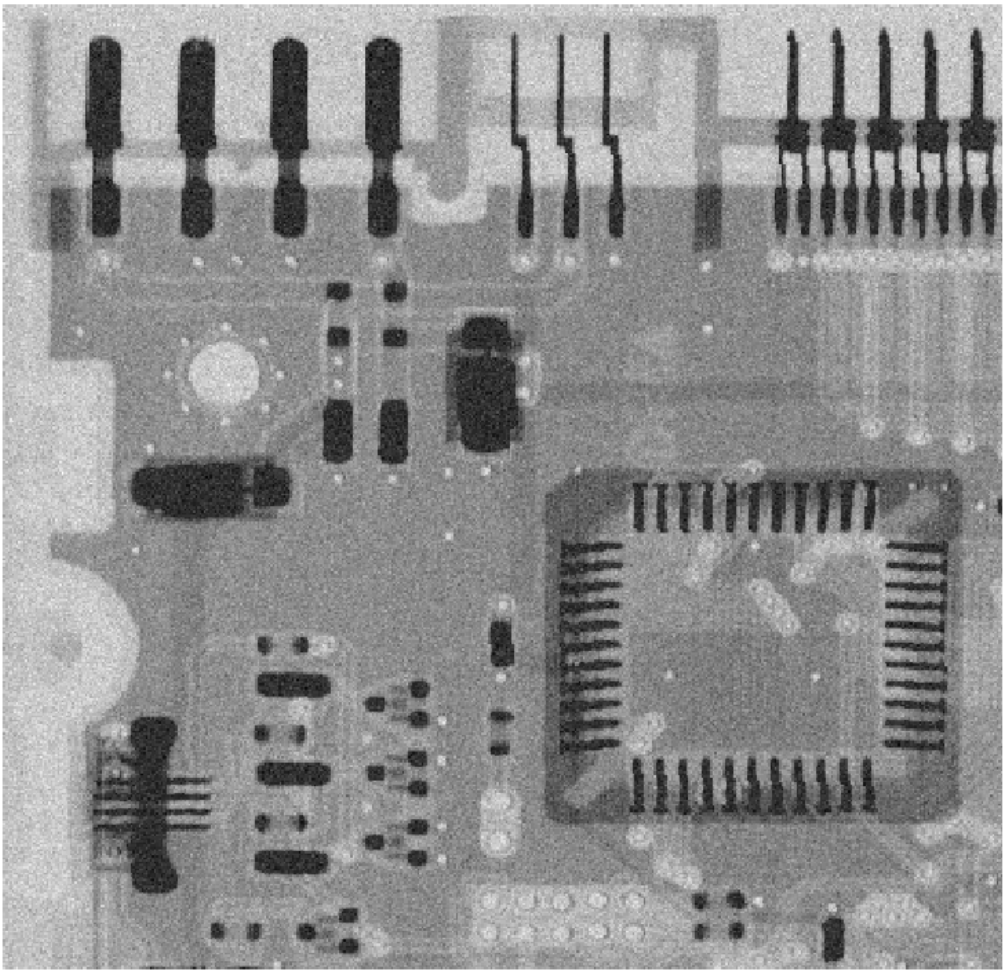
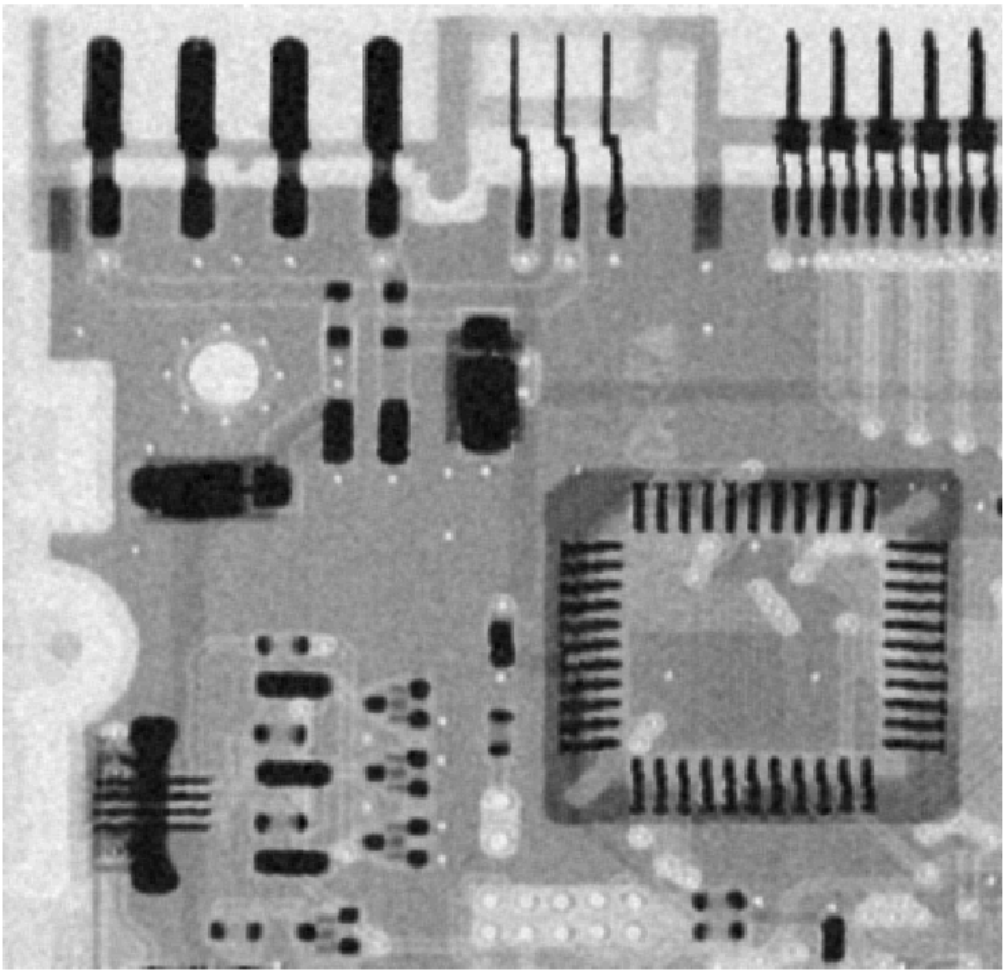


Fig.5.7 (b)



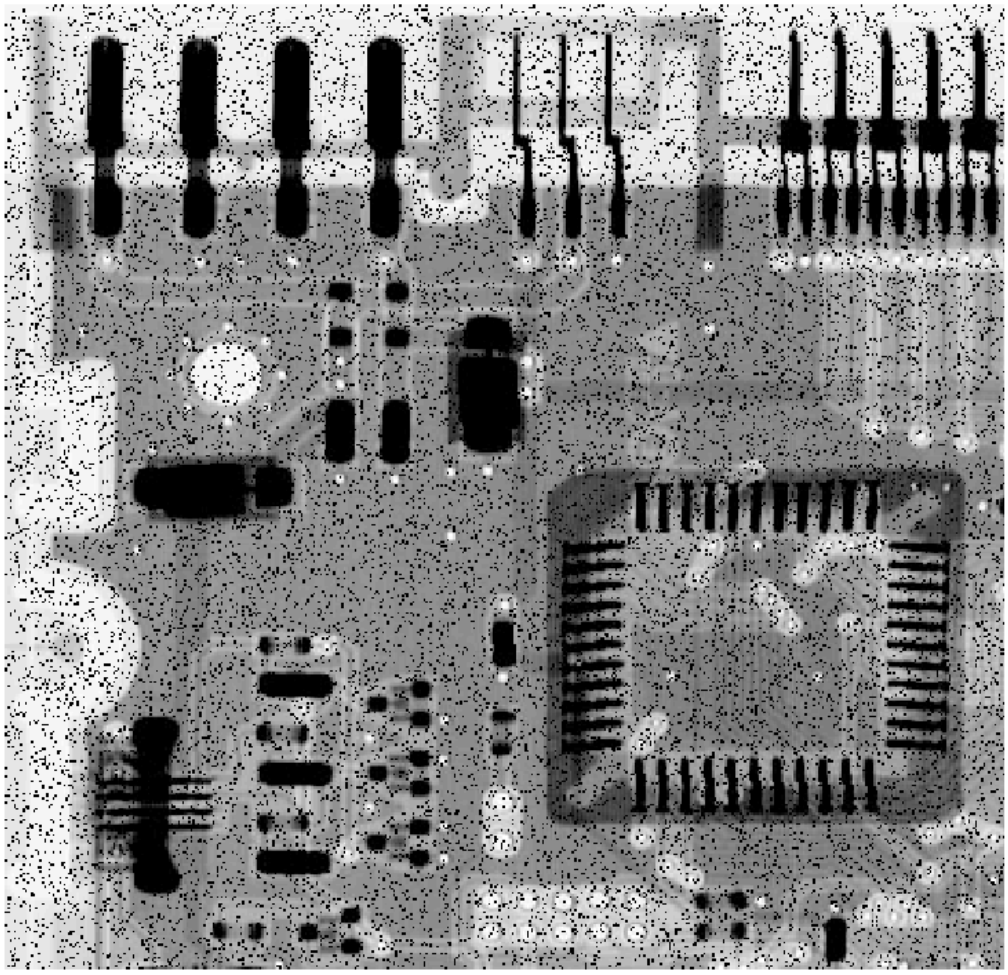
Use Geometric mean filter to denoise Gaussian Noise

Fig.5.7 (d)



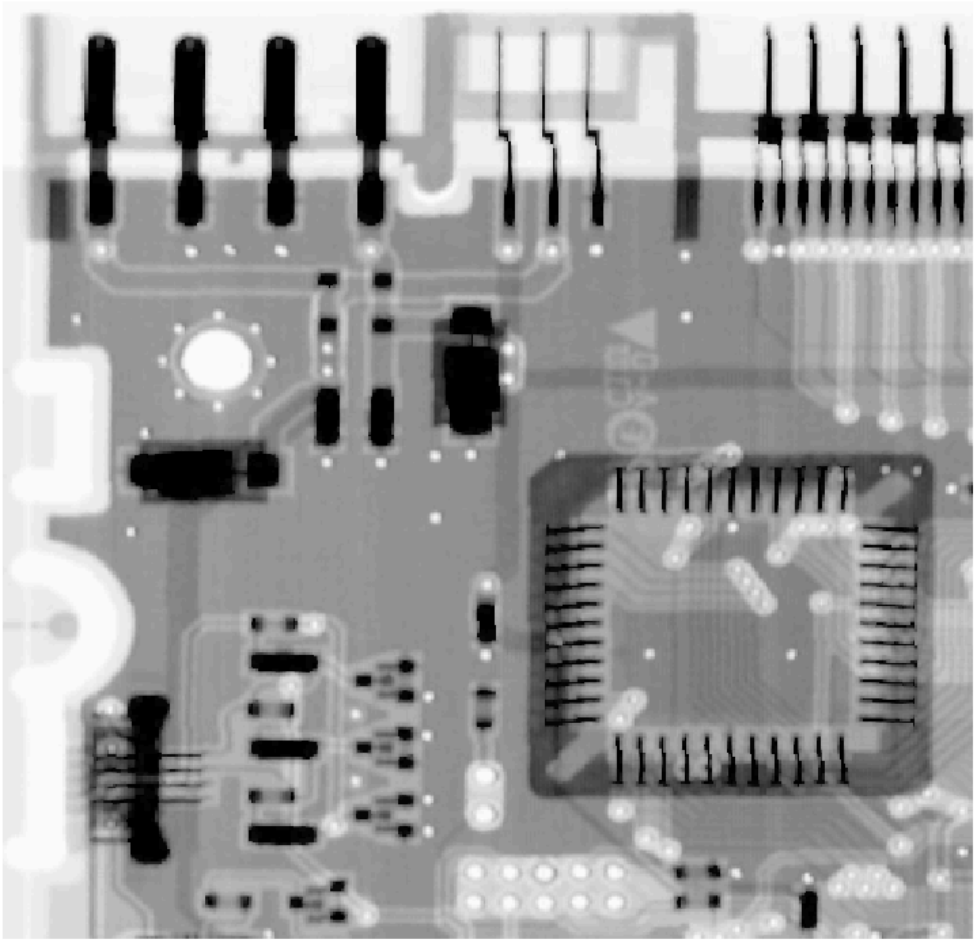
```
PSNR: 23.3823dB
```

Fig.5.8 (a)



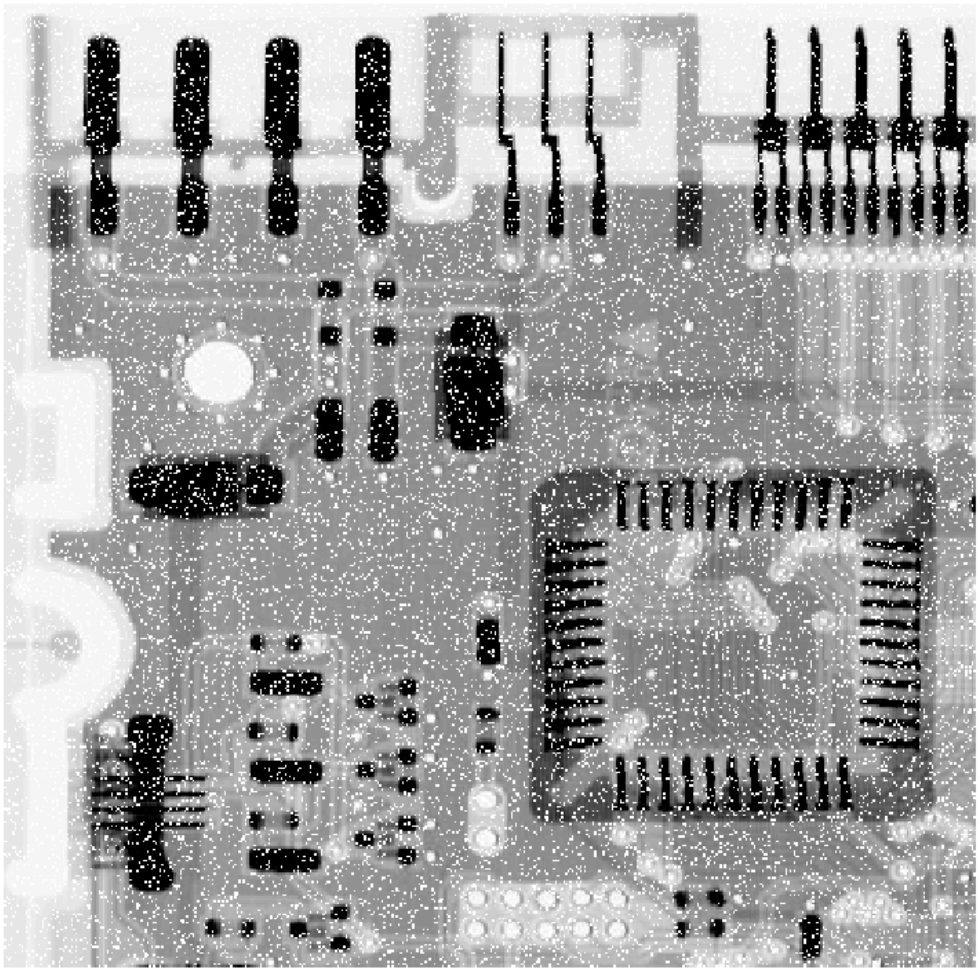
Use Contraharmonic mean filter to denoise Pepper Noise

Fig.5.8 (c)



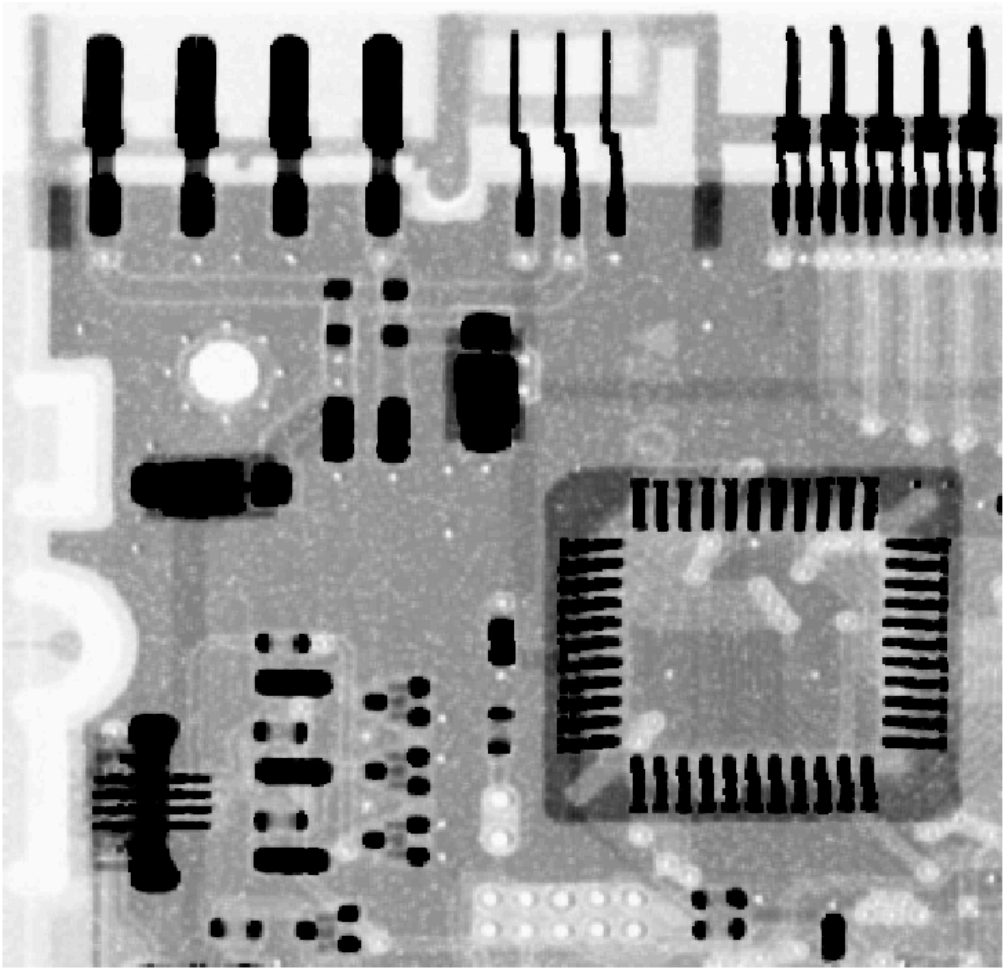
```
PSNR: 20.5022dB
```

Fig.5.8 (b)



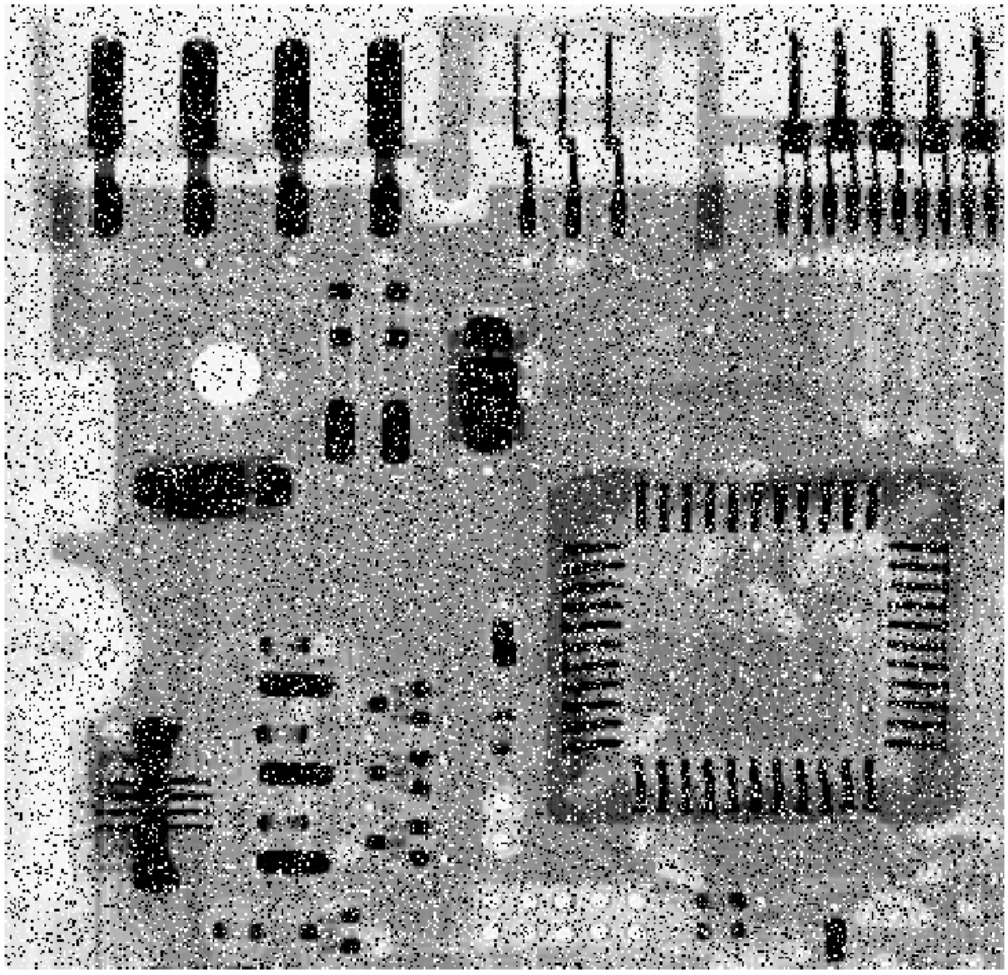
Use Contraharmonic mean filter to denoise Salt Noise

Fig.5.8 (d)



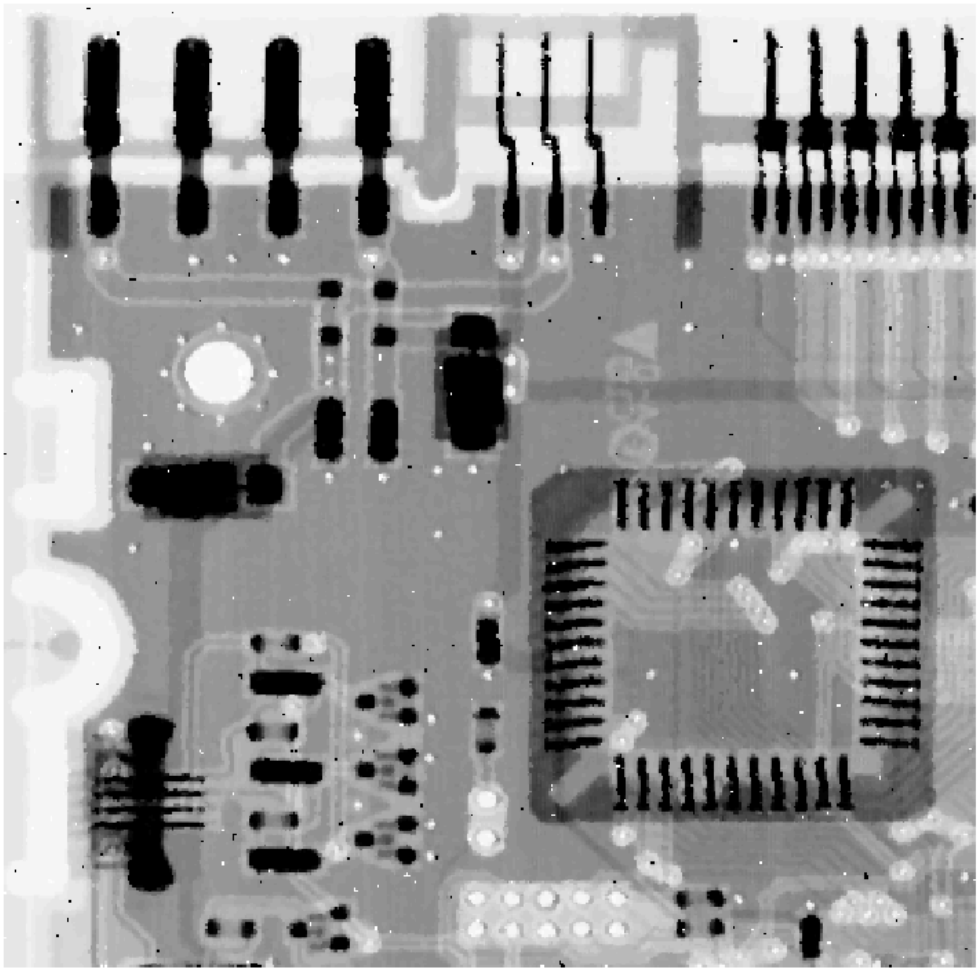
PSNR: 20.5918dB

Fig.5.10 (a)



Use Median filter to denoise Salt and Pepper Noise

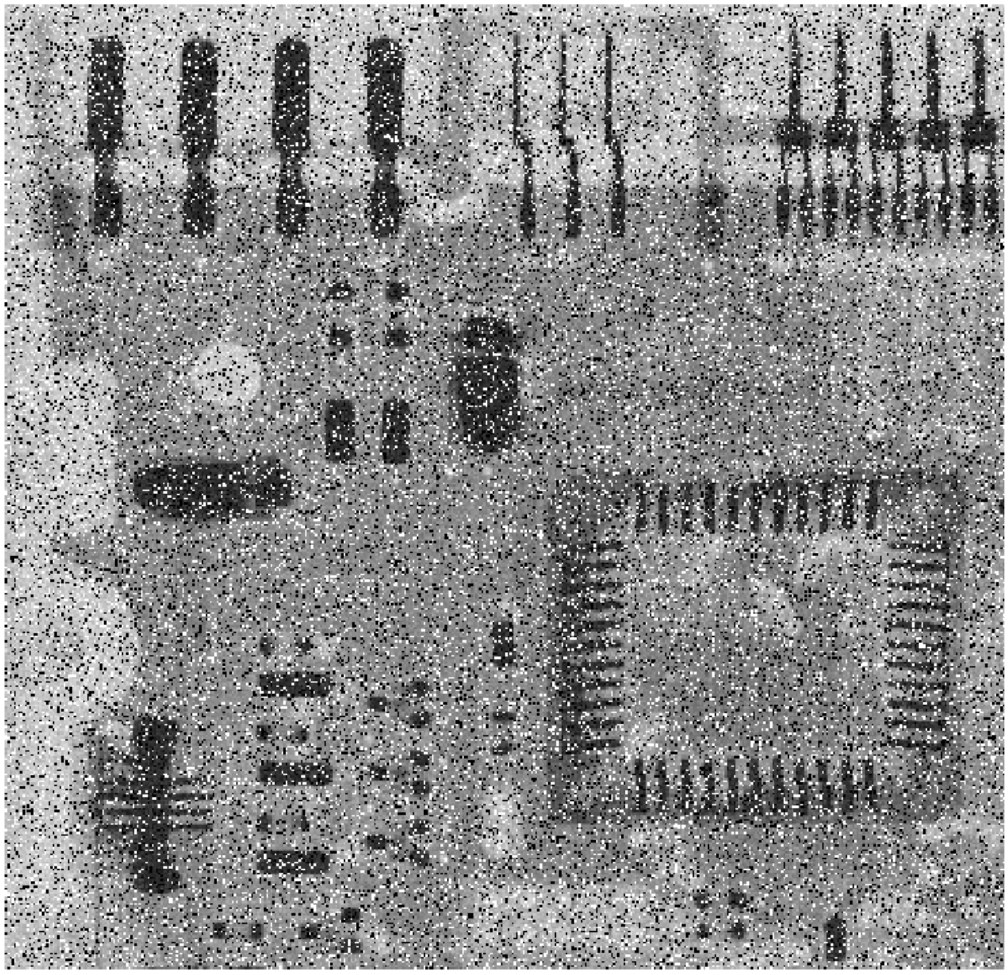
Fig.5.10 (b)



PSNR: 25.6699dB

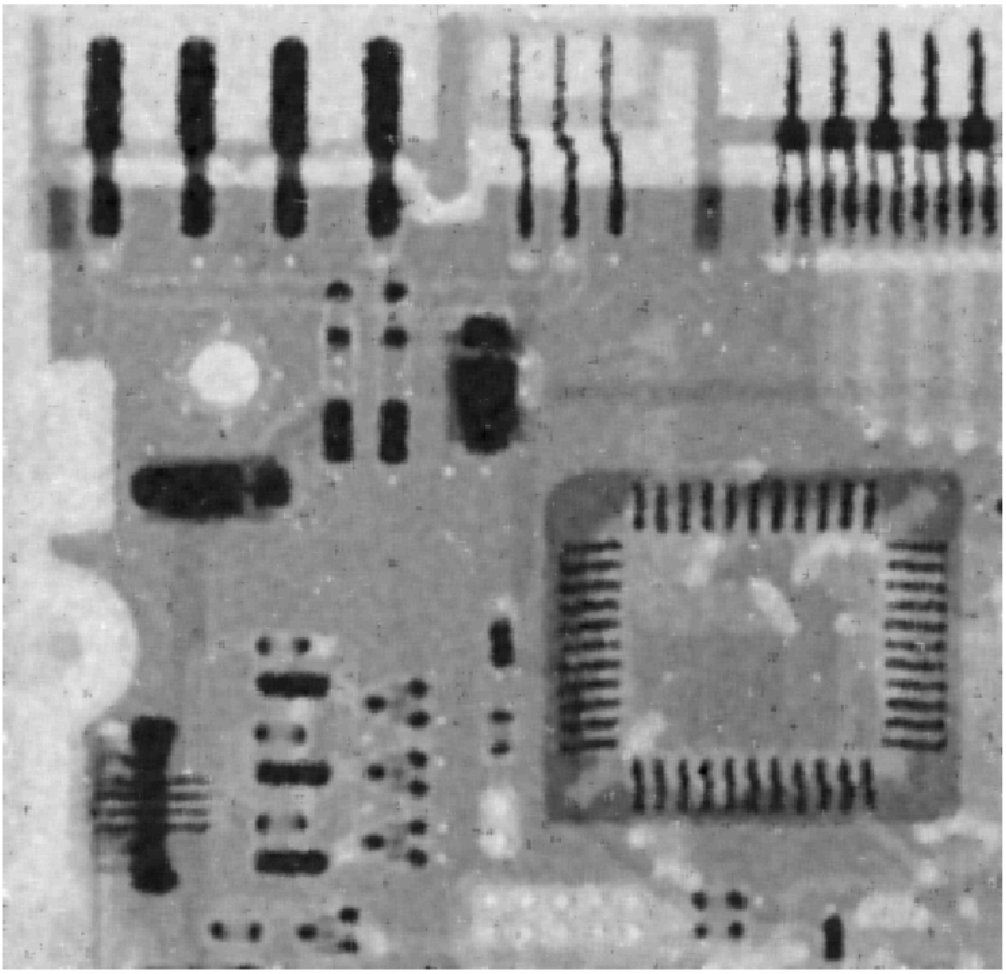
Gaussian Noise + Salt and Pepper Noise

Fig.5.12 (b)



Use Alpha-trimmed mean filter to denoise combined Noise

Fig.5.12 (f)



```
PSNR: 20.5883dB
```

Implementation

addGaussianNoise

```
function output_s = addGaussianNoise(input_s, mu, sigma)
    noise = sigma * randn(size(input_s)) + mu;
    output_s = double(input_s) + noise;
    output_s = uint8(255 * mat2gray(output_s));
end
```

addImpulseNoise

```
function output_s = addImpulseNoise(input_s, Ps, Pp)
    prob = rand(size(input_s));
    output_s = input_s;

    output_s(prob <= Pp) = 0; % pepper
    output_s(prob >= 1 - Ps) = 255; % salt
end
```

geometricMeanFilter

```
function output = geometricMeanFilter(input, m, n)
    input = double(input);
    [M, N] = size(input);
    output = zeros(M, N);

    % Pad the image to handle borders
    pad_size_m = floor(m / 2);
    pad_size_n = floor(n / 2);
    padded_img = padarray(input, [pad_size_m, pad_size_n], 'replicate');

    % Apply the geometric mean filter
    for i = 1 : M
        for j = 1 : N
            % Extract the neighborhood
            neighborhood = padded_img(i : i + m - 1, j : j + n - 1);

            % Calculate the geometric mean
            neighborhood(neighborhood == 0) = 1; % Avoid log(0)
            geo_mean = exp(sum(log(neighborhood(:))) / numel(neighborhood));
            output(i, j) = geo_mean;
        end
    end

    output = uint8(255 * mat2gray(output));
end
```

contraharmonicMeanFilter

```
function output = contraharmonicMeanFilter(input, m, n, Q)
    input = double(input);
    [M, N] = size(input);
    output = zeros(M, N);

    % Define the neighborhood size based on m and n
    m = floor(m / 2);
    n = floor(n / 2);

    for i = 1 : M
        for j = 1 : N
            % Define the neighborhood indices
            row_start = max(1, i - m);
            row_end = min(M, i + m);
            col_start = max(1, j - n);
            col_end = min(N, j + n);

            % Extract the neighborhood
            neighborhood = input(row_start:row_end, col_start:col_end);

            % Calculate the contraharmonic mean
            numerator = sum(sum(neighborhood .^ (Q + 1)));
            denominator = sum(sum(neighborhood .^ Q));

            % Handle division by zero if the denominator is zero
            if denominator == 0
                output(i, j) = 0; % or keep the original pixel value
            else
                output(i, j) = numerator / denominator;
            end
        end
    end

    output = uint8(255 * mat2gray(output));
end
```

medianFilter

```
function output = medianFilter(input, m, n)
    input = double(input);
    [M, N] = size(input);
    output = zeros(M, N);

    % Pad the image to handle borders
    pad_size_m = floor(m / 2);
    pad_size_n = floor(n / 2);
    padded_img = padarray(input, [pad_size_m, pad_size_n], 'replicate');

    % Apply the median filter
    for i = 1 : M
        for j = 1 : N
            % Extract the neighborhood
            neighborhood = padded_img(i : i + m - 1, j : j + n - 1);

            % Compute the median of the neighborhood
            output(i, j) = median(neighborhood(:));
        end
    end

    output = uint8(255 * mat2gray(output));
end
```

alphaTrimmedMeanFilter

```
function output = alphaTrimmedMeanFilter(input, m, n, d)
    input = double(input);
    [M, N] = size(input);
    output = zeros(M, N);

    % Pad the image to handle borders
    pad_size_m = floor(m / 2);
    pad_size_n = floor(n / 2);
    padded_img = padarray(input, [pad_size_m, pad_size_n], 'replicate');

    % Apply the alpha-trimmed mean filter
    for i = 1 : M
        for j = 1 : N
            % Extract the neighborhood
            neighborhood = padded_img(i : i + m - 1, j : j + n - 1);

            % Sort the neighborhood values
            sorted_values = sort(neighborhood(:));

            % Trim the lowest and highest d values
            trimmed_values = sorted_values((d + 1):(end - d));  % Exclude d smal

            % Calculate the mean of the trimmed values
            output(i, j) = mean(trimmed_values);
        end
    end

    output = uint8(255 * mat2gray(output));
end
```

Discussion

Because of the type and floating point precision, I choose to translate the type to double, and then translate back to uint8 by `output = uint8(255 * mat2gray(output));`.

Also, it seems that setting `Q = -1.5` in my contraharmonicMeanFilter implementation will cause some error(the denominator extremely small while != 0), so I set `Q = -1` (no smaller than -1.5).

Proj05-03: Periodic Noise Reduction Using a Notch Filter

Results

Fig.5.26 (a)

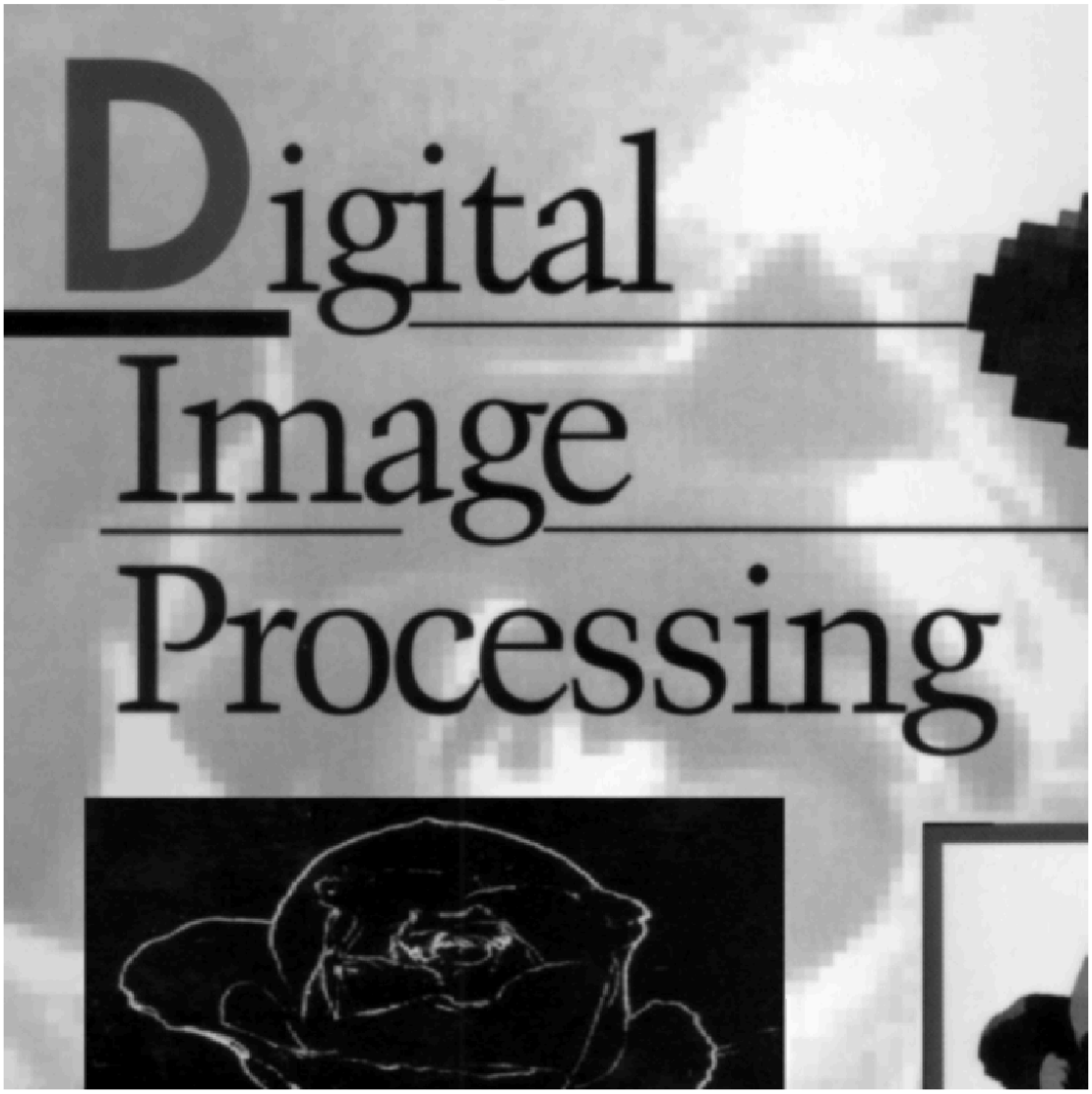
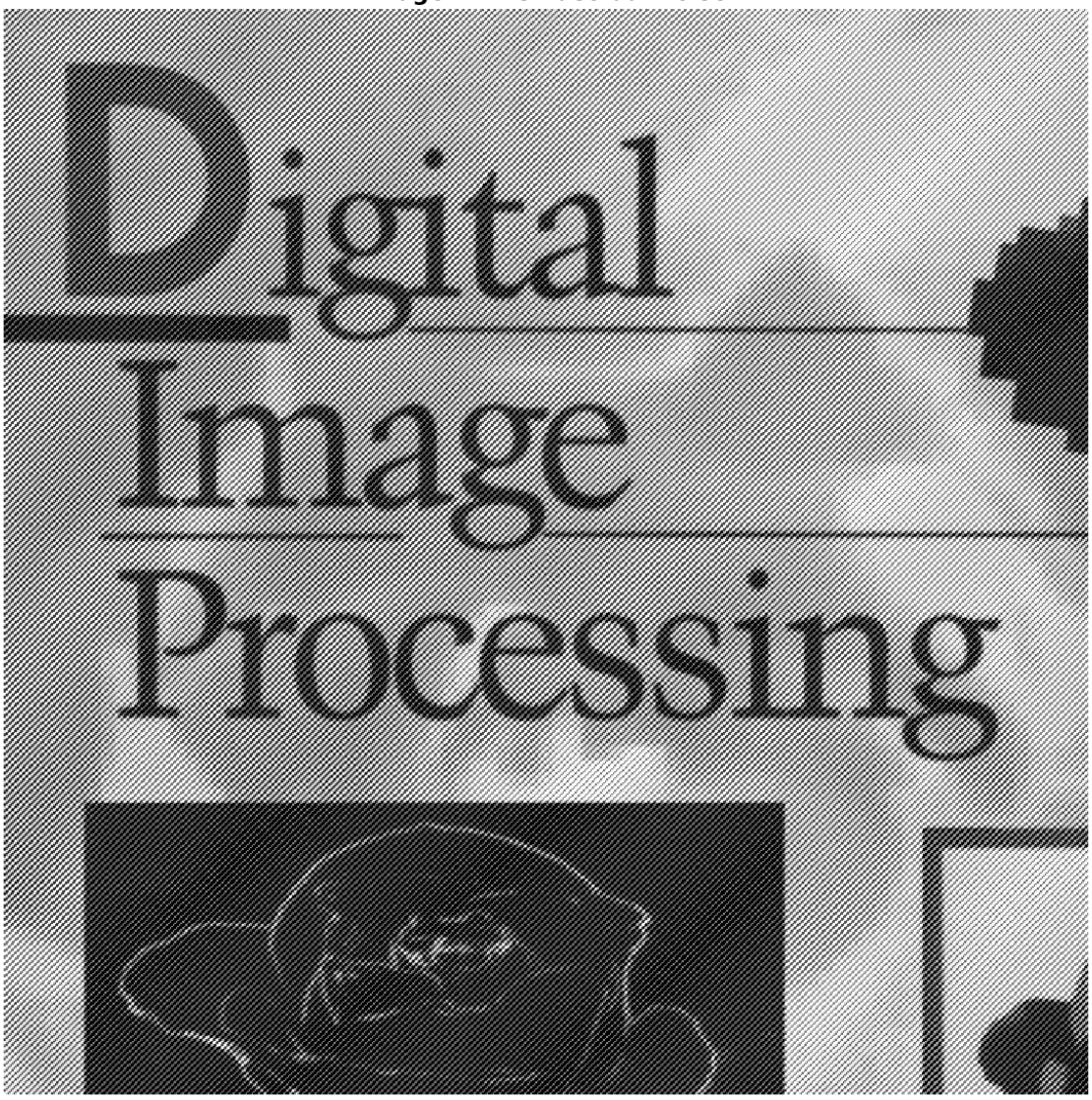
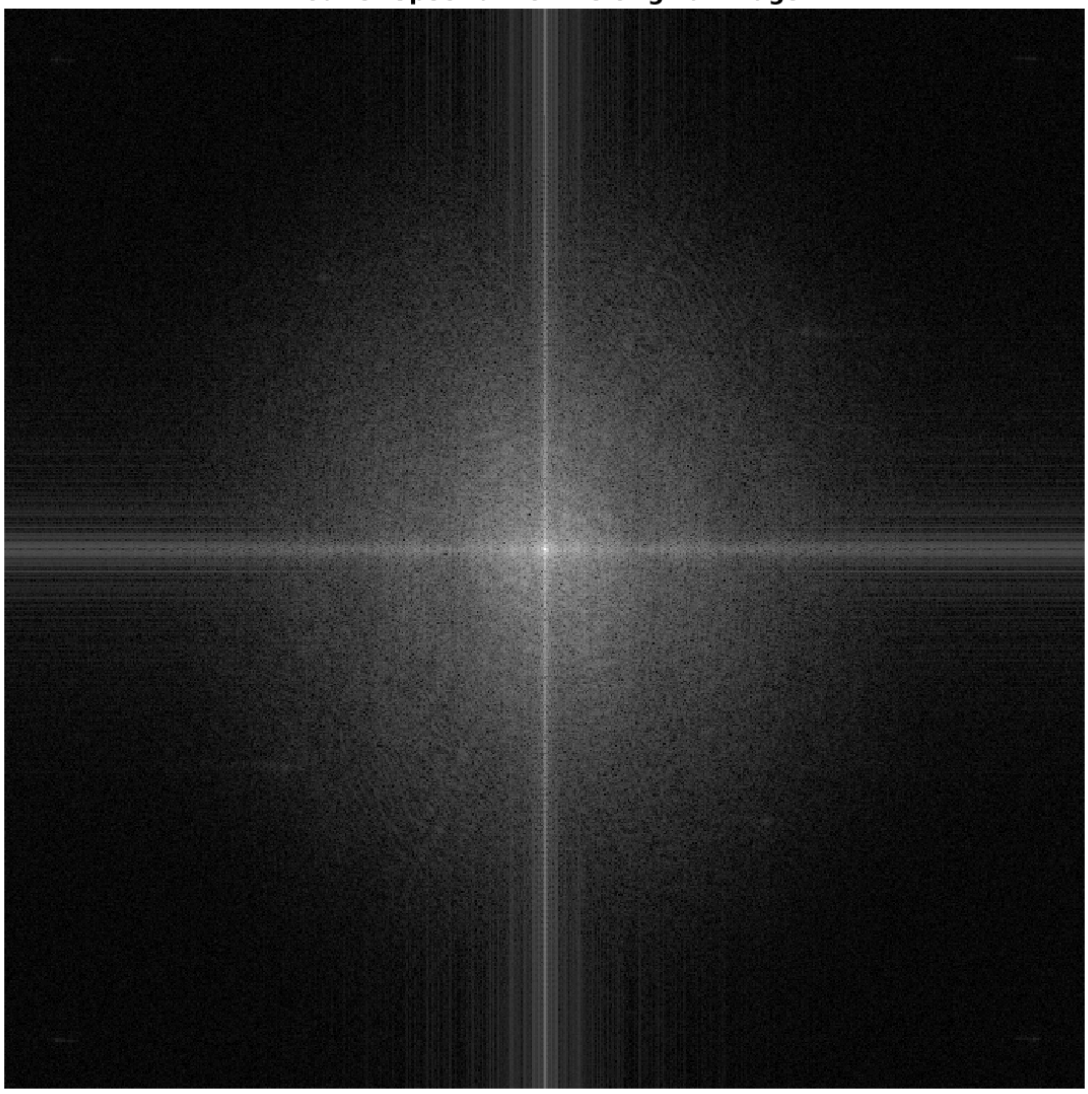


Image with sinusoidal noise

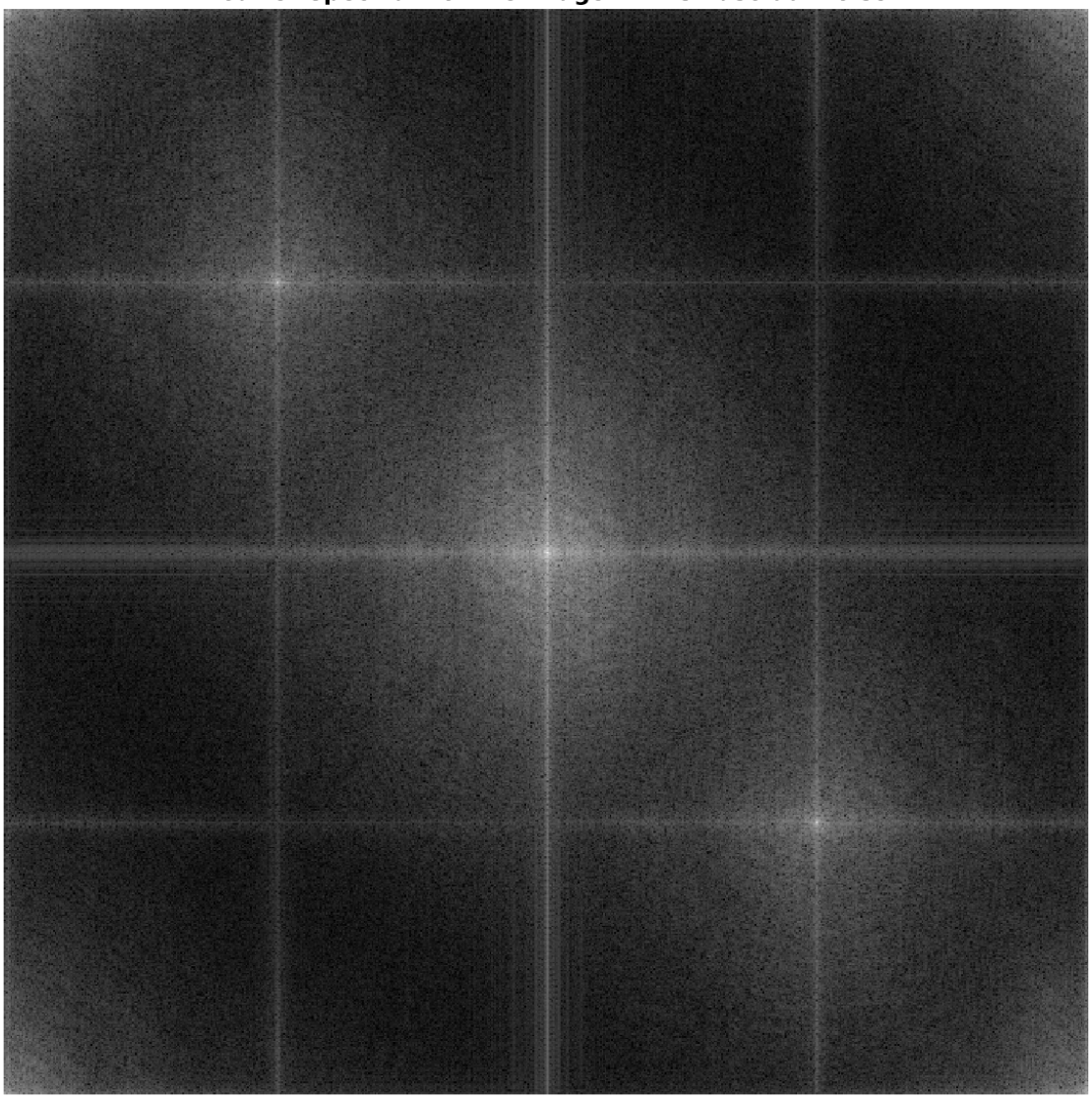


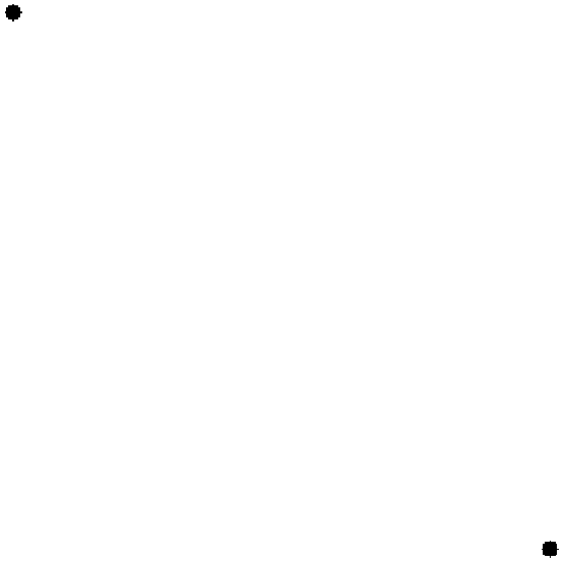
$A = 0.5, u0 = M/4 - 1, v0 = N/4 - 1$

Fourier Spectrum of the original image



Fourier Spectrum of the image with sinusoidal noise





$D0 = 5, u0 = M/4 - 1, v0 = N/4 - 1$

Fourier Spectrum of the image with sinusoidal noise after filtering

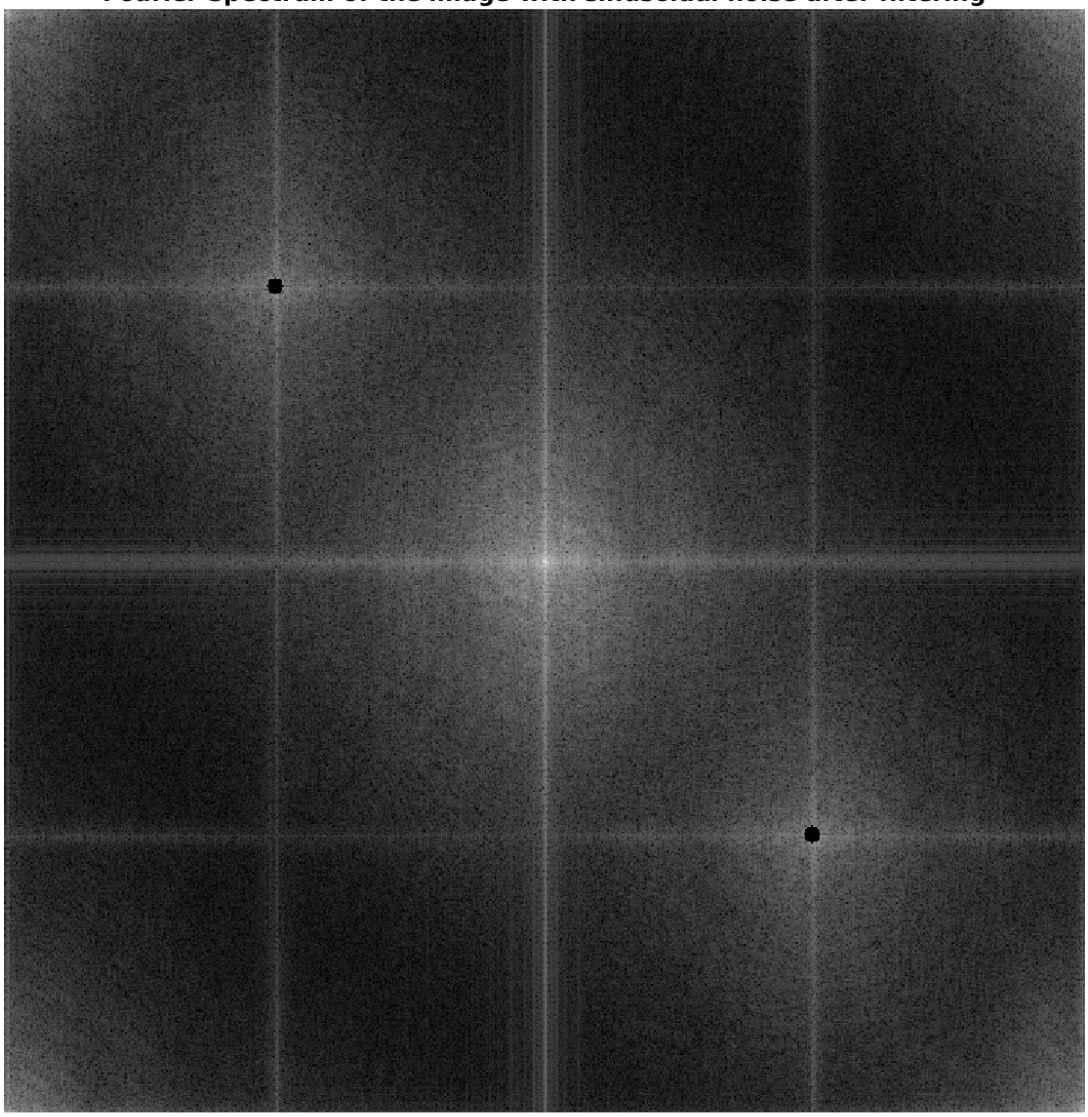
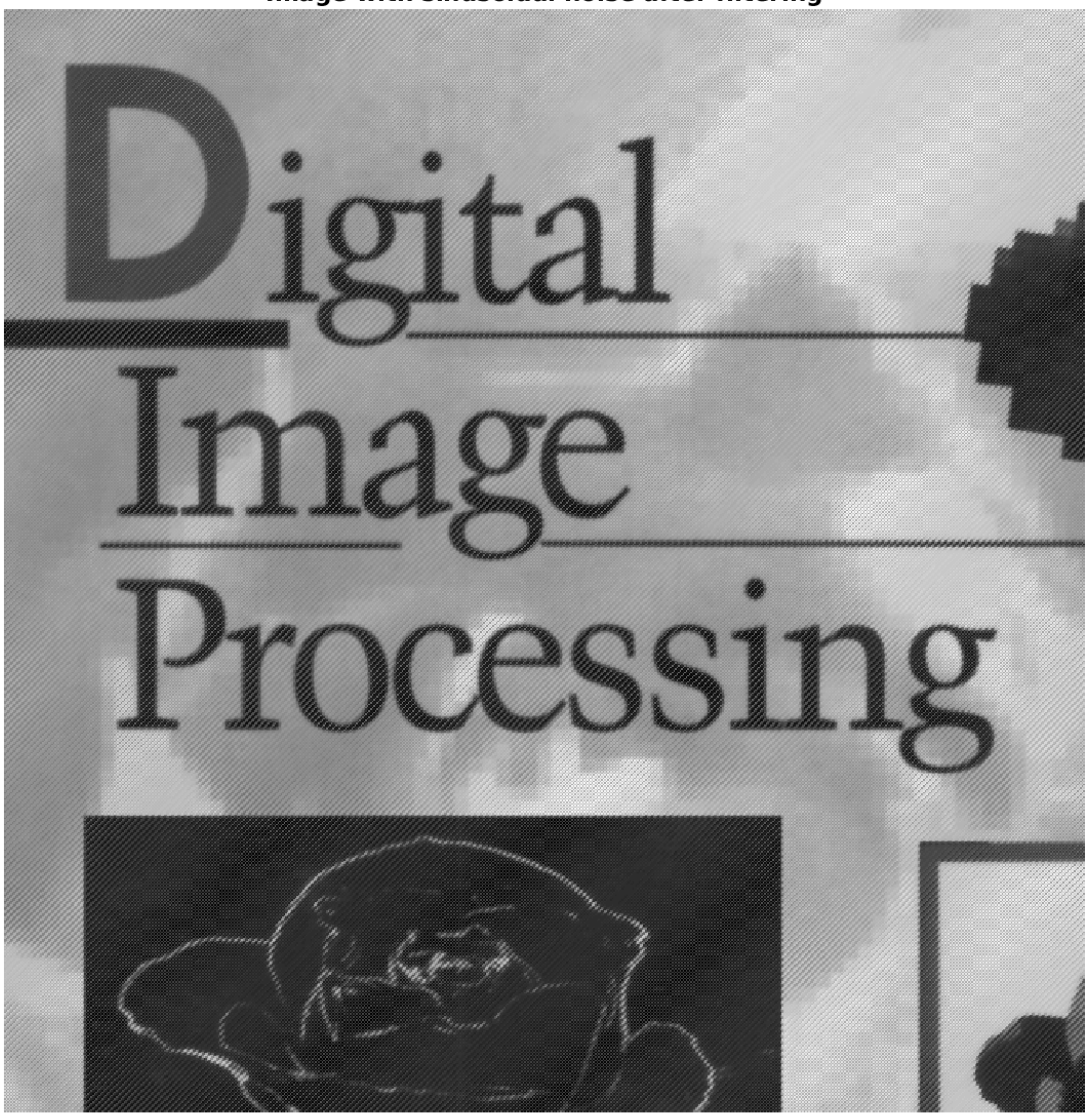


Image with sinusoidal noise after filtering




```
PSNR: 18.9053dB
```

Implementation

addSinNoise

```
function output_s = addSinNoise(input_s, A, u0, v0)
    [M, N] = size(input_s);
    [X, Y] = meshgrid(1:N, 1:M);
    noise = A * sin(2 * pi * (u0 * X / M + v0 * Y / N));
    output_s = input_s + noise;
    output_s = max(min(output_s, 1), 0);
end
```

notchFiltering

```
function [output_f, Notch] = notchFiltering(input_f, D0, u0, v0)
    [M, N] = size(input_f);
    [U, V] = meshgrid(1:N, 1:M);

    % Center the frequencies
    U = U - N / 2;
    V = V - M / 2;

    % Calculate distance for notch locations
    D1 = sqrt((U - u0).^2 + (V - v0).^2);
    D2 = sqrt((U + u0).^2 + (V + v0).^2);

    % Ideal notch reject filter
    Notch = ones(M, N);
    Notch(D1 <= D0) = 0;
    Notch(D2 <= D0) = 0;

    % Apply the notch filter
    output_f = input_f .* Notch;
end
```

computePSNR

```
function psnr = computePSNR(input1_s, input2_s)
    input1_s = double(input1_s);
    input2_s = double(input2_s);
    mse = mean((input1_s(:) - input2_s(:)).^2);
    max_pixel_value = 1;
    if mse == 0
        psnr = Inf;
    else
        psnr = 10 * log10((max_pixel_value^2) / mse);
    end
end
```

Discussion

The PSNR is lower than all of the filter I have implemented above. The result is quite reasonable, since I can only use two-circle notch filter to filter out the noise. If we take a look at the frequency domain, we can see that there are some remaining noise of sine wave haven't been filtered out.

Proj05-04: Parametric Wiener Filter

Results

Fig.5.26 (a)

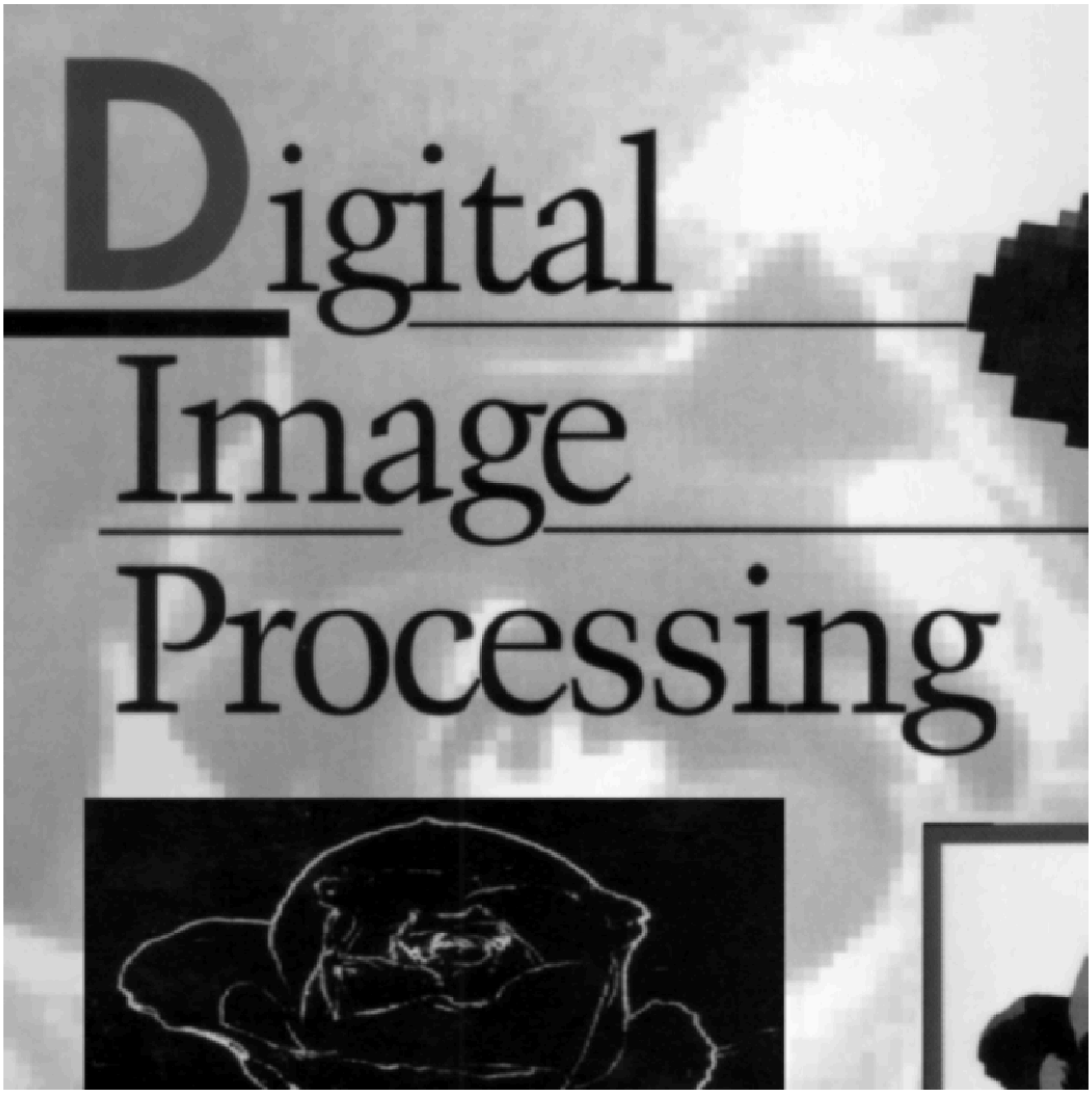
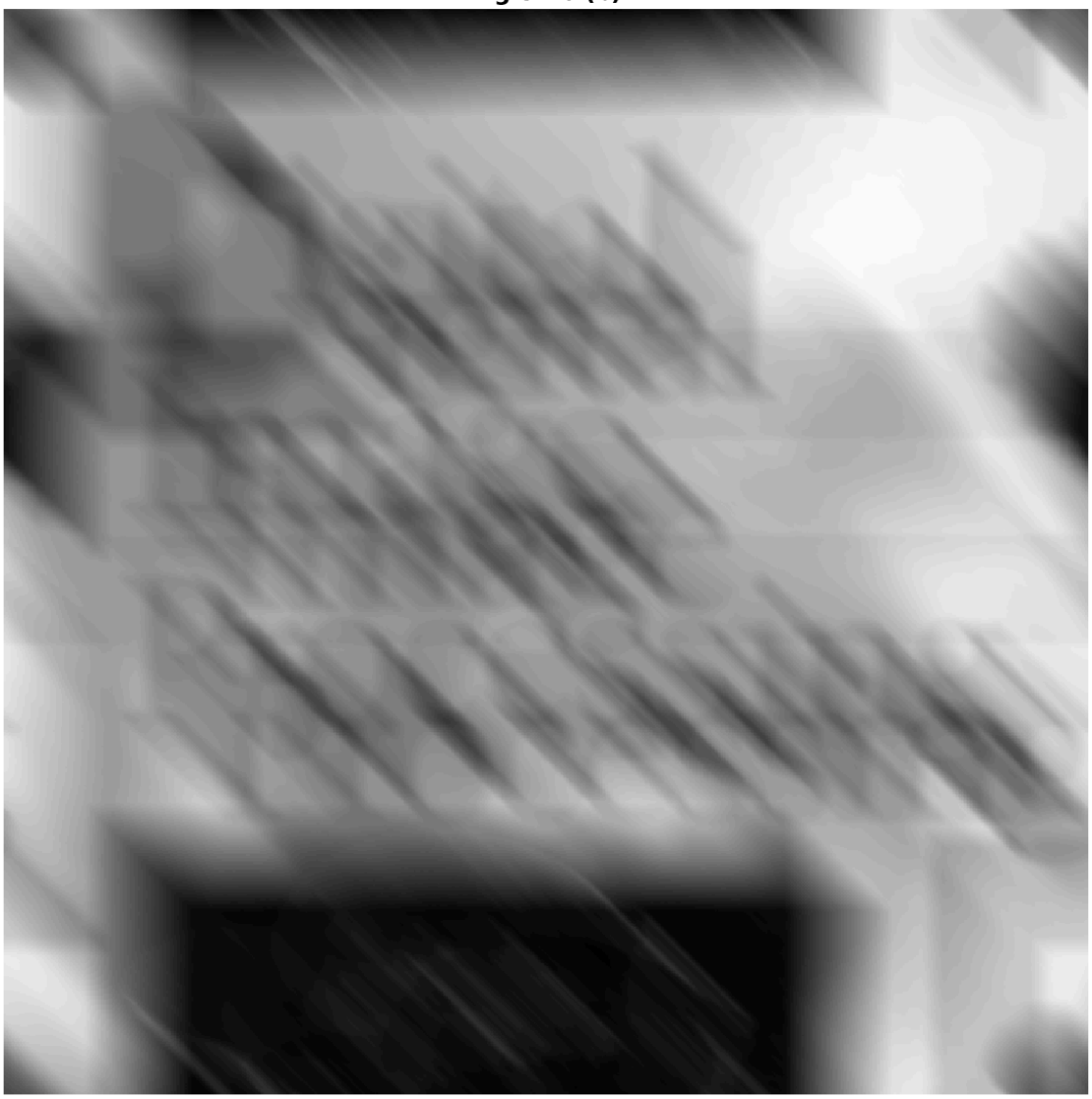
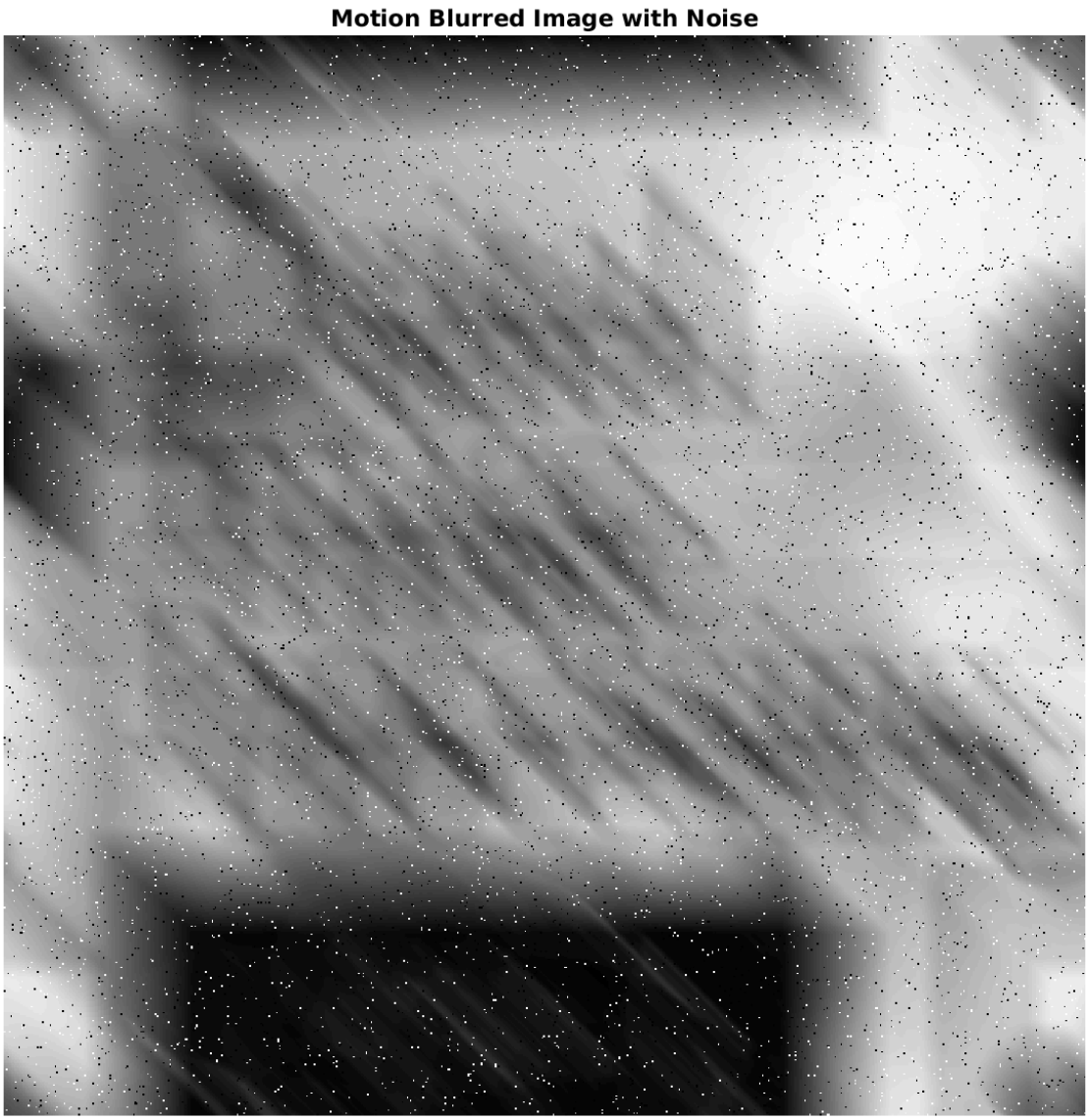


Fig.5.26 (b)



```
PSNR: 11.8087 dB
```

Add `output = imnoise(output, 'salt & pepper', 0.02);`

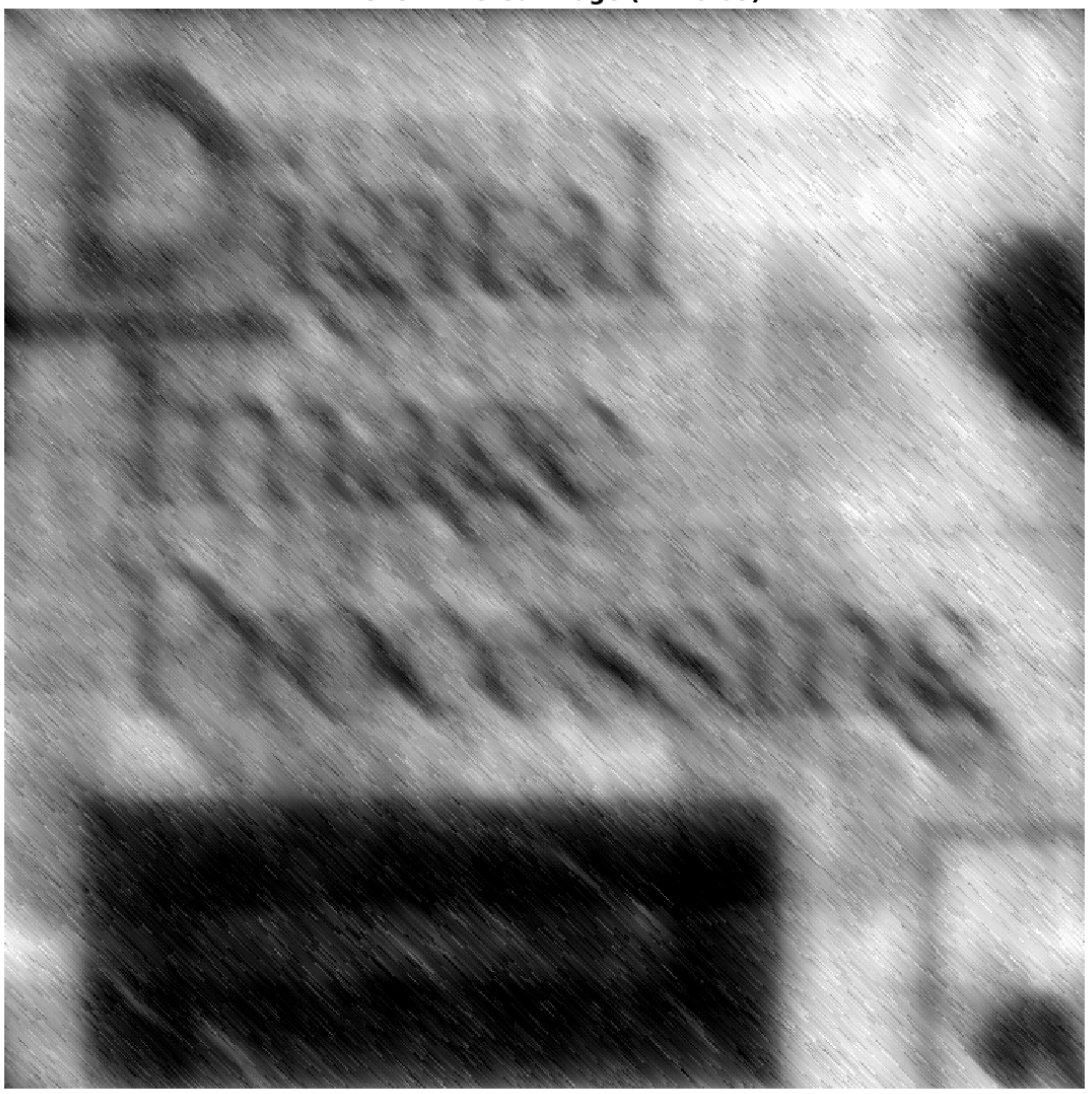



```
PSNR: 11.4419 dB
```

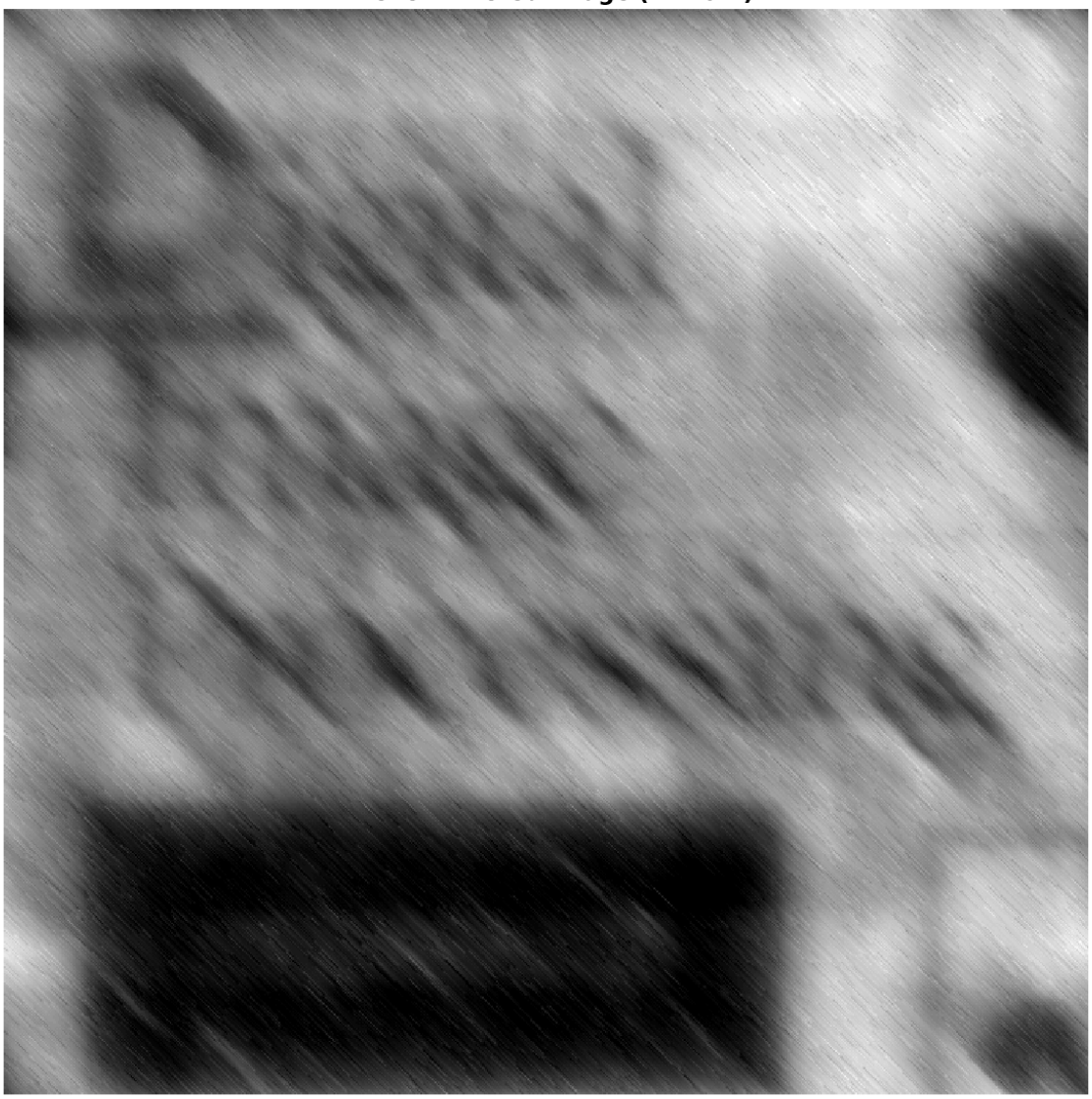
Wiener Filtered Image (K = 0.01)

```
PSNR (K = 0.01): 14.4564 dB
```

Wiener Filtered Image (K = 0.05)



```
PSNR (K = 0.05): 16.261 dB
```




```
PSNR (K = 0.1): 15.5079 dB
```

Implementation

addMotionBlur

```
function [output_f, H] = addMotionBlur(input_f, T, a, b)
    [M, N] = size(input_f);
    [U, V] = meshgrid((0:N-1) - floor(N/2), (0:M-1) - floor(M/2));

    % Compute the motion blur filter H
    W = pi * (U * a + V * b);
    H = (T ./ W) .* sin(W) .* exp(-1j * W);

    % Handle division by zero
    H(W == 0) = T;

    % Apply the motion blur filter
    output_f = input_f .* H;
end
```

wienerFiltering

```
function output_f = wienerFiltering(input_f, H, K)
    % Compute the Wiener filter
    H_conj = conj(H);
    H_abs_sq = abs(H).^2;
    Wiener_filter = (H_conj ./ (H_abs_sq + K));

    % Apply the Wiener filter
    output_f = Wiener_filter .* input_f;
end
```

Discussion

For `K = 0.01`, the image after Wiener Filtering is the most identifiable(among these three), while the PSNR(dB) of which is the lowest. We can see that after Wiener Filtering, the PSNR ratio has risen from 11 to 14~16(by carefully tunning), which means that the image become more identifiable(to some degree). So we actually implemented some kind of restoration?