Digital Image Processing

Project report

Table of Contents

PROJECT 1 - HISTOGRAM EQUALIZATION 3

PROJECT 2 - COMBINING SPATIAL ENHANCEMENT METHODS 10

PROJECT 3 FILTERING IN FREQUENCY DOMAIN 17

PROJECT 4 - GENERATING DIFFERENT TYPES OF NOISE AND COMPARING DIFFERENT NOISE REDUCTION METHODS 28

PROJECT 5 - IMAGE RESTORATION 45

PROJECT 6 - GEOMETRIC TRANSFORM 54

PROJECT 8 - MORPHOLOGICAL PROCESSING 64

PROJECT 9 - IMAGE SEGMENTATION 74

Project 1 - Histogram Equalization

Problem

Histogram Equalization (test images: fig1.jpg, fig2.jpg)
(a) Write a computer program for computing the histogram of an image.
(b) Implement the histogram equalization technique.
(c) Your program must be general to allow any gray-level image as its input.
As a minimum, your report should include the original image, a plot of its histogram, a plot of the transformation function, the enhanced image, and a plot of its histogram.

Environment

Matlab R2016a
Atom 1.22 (with language-matlab package)

Principle

This method usually increases the global contrast of many images, especially when the usable data of the image is represented by close contrast values. Through this adjustment, the intensities can be better distributed on the histogram. This allows for areas of lower local contrast to gain a higher contrast. Histogram equalization accomplishes this by effectively spreading out the most frequent intensity values.
For discrete values, we deal with probabilities (histogram values) and summations instead of probability density functions and integrals. The probability of occurrence of intensity level r_k in a digital image is approximated by:

$$p_r(r_k) = \frac{n_k}{MN} \qquad k = 0, 1, 2, \ldots, L-1$$

And where MN is the total number of pixels in the image, nk is the number of pixels that have intensity rk, and L is the number of possible intensity levels in the image. The discrete form of the transformation is:

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p_r(r_j)$$

$$= \frac{(L-1)}{MN}\sum_{j=0}^{k} n_j \qquad k = 0, 1, 2, \ldots, L-1$$

Implementation

1. Clear the former environments at the beginning:

```
% Clear the environment
close all;
clc;
clear;
```

2. Input the image and get the information about the image (There the location of the input image is directly assigned for convenience and it also can be assigned by the function parameter after comment this line) :

```
% Set the default location of input image file
img_location = '../images/Fig1.jpg';
input_img = imread(img_location);

% Get the relative information of the i
info = imfinfo(img_location);
MN = info.Width * info.Height;

% For this program must be general to allow any
% gray-level image, so it needs to get the depth
% of the image
L = info.BitDepth;
```

3. Initial the gray distribution of input and output image arrays , p_r array and T(r) array with zero:

```
% Initialize the gray distribution and output image matrix
input_gray_distribution = zeros(2^L,1);
output_img = zeros(info.Height,info.Width);
output_gray_distribution = zeros(2^L,1);
transform = zeros(2^L,1);
```

4. Get the gray distribution of input image:

```
% Calculate the gray distribution of input image
for i = 1:2^L
    input_gray_distribution(i) = length(find(input_img == (i - 1)));
end
```

5. Get the pdf of input image by dividing MN:

```
% Get the pdf
input_img_pdf = input_gray_distribution / MN;
```

6. Get the tranformation:

```
% Calculate the transformation
for i = 1:2^L
  transform(i) = (2^L - 1) * sum(input_img_pdf(1:i));
end
```

7. Get the output image and cast the type from double to uint:

```
% Calculate the output image
for i = 1:info.Width
    for j = 1:info.Height
        output_img(j,i) = transform(double(input_img(j,i)) + 1);
    end
end

% Format the values of pixels
output_img = uint8(output_img);
```

8. Calculate the gery histogram of output image:

```
% Calculate the gray distribution of output image
for i = 1:2^L
    output_gray_distribution(i) = length(find(output_img == i));
end
```

9. Draw the bar graphs and show the input and output images:

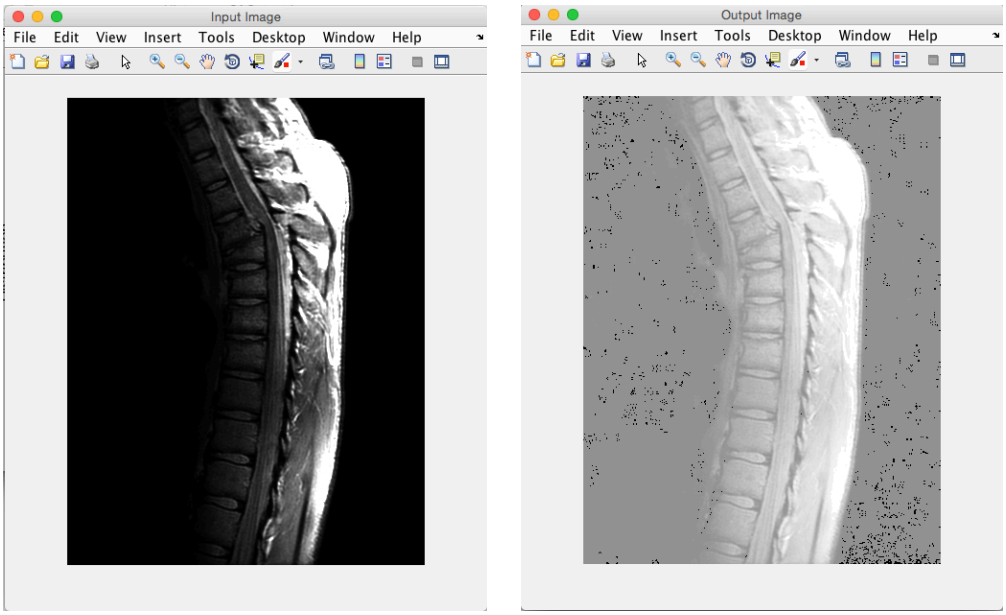
```
%%%%%%%%%%%%%%%% Show pictures & Draw graphs %%%%%%%%%%%%%%%%%%
% Show pictures and graphs seperately
figure('NumberTitle','off','Name','Histogram Of Input Image')
bar(input_gray_distribution,0.7);
xlabel('Gray Level','FontSize',16);
ylabel('Distribution','FontSize',16);
set(gca, 'XLim',[0 2^L]);
set(gca, 'YLim',[0 16000]);

figure('NumberTitle','off','Name','Histogram Of Output Image')
bar(output_gray_distribution,0.7);
xlabel('Gray Level','FontSize',16);
ylabel('Distribution','FontSize',16);
set(gca, 'XLim',[0 2^L]);
set(gca, 'YLim',[0 16000]);

figure('NumberTitle', 'off', 'Name', 'Transformation function')
plot(1:2^L, transform)
xlabel('Original Gray','FontSize',16);
ylabel('Transformed Gray','FontSize',16);
set(gca, 'XLim',[0 2^L]);
set(gca, 'YLim',[0 2^L]);

figure('NumberTitle','off','Name','Input Image')
imshow(input_img);

figure('NumberTitle','off','Name','Output Image')
imshow(output_img);
```

10. Compare them in one graph:

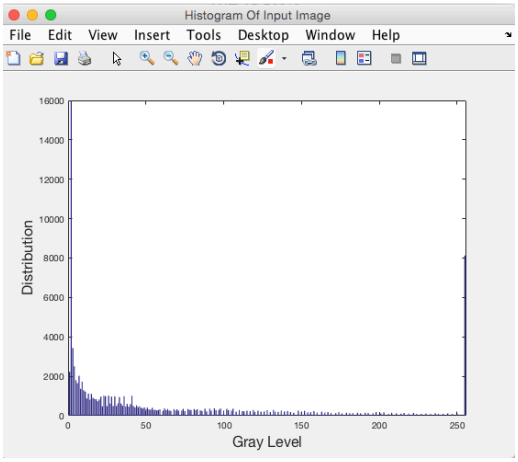
```
% Show them all together to compare
figure('NumberTitle','off','Name','Compare')
subplot(2,3,1);
imshow(input_img);
title('The Input Image');
```

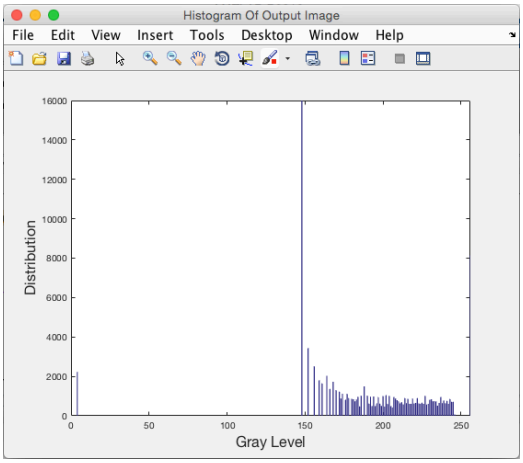
```
92  subplot(2,3,2);
93  bar(input_gray_distribution,0.7);
94  title('Histogram Of Input Image');
95  xlabel('Gray Level','FontSize',16);
96  ylabel('Distribution','FontSize',16);
97  set(gca, 'XLim',[0 2^L]);
98  set(gca, 'YLim',[0 16000]);
99
100 subplot(2,3,3);
101 plot(1:2^L, transform)
102 title('The Transformation Function')
103 xlabel('Original Gray','FontSize',16);
104 ylabel('Transformed Gray','FontSize',16);
105 set(gca, 'XLim',[0 2^L]);
106 set(gca, 'YLim',[0 2^L]);
107
108 subplot(2,3,4);
109 imshow(output_img);
110 title('The Output Image');
111
112 subplot(2,3,5);
113 bar(output_gray_distribution,0.7);
114 title('Histogram Of Output Image');
115 xlabel('Gray Level','FontSize',16);
116 ylabel('Distribution','FontSize',16);
117 set(gca, 'XLim',[0 2^L]);
118 set(gca, 'YLim',[0 16000]);
```

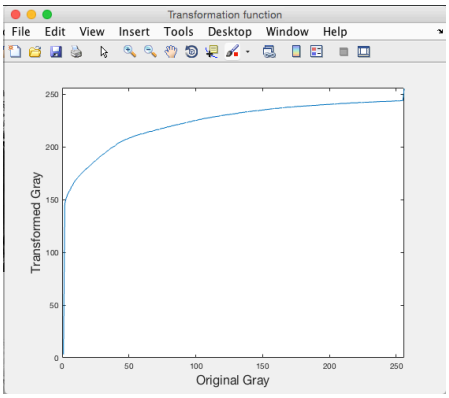
Result



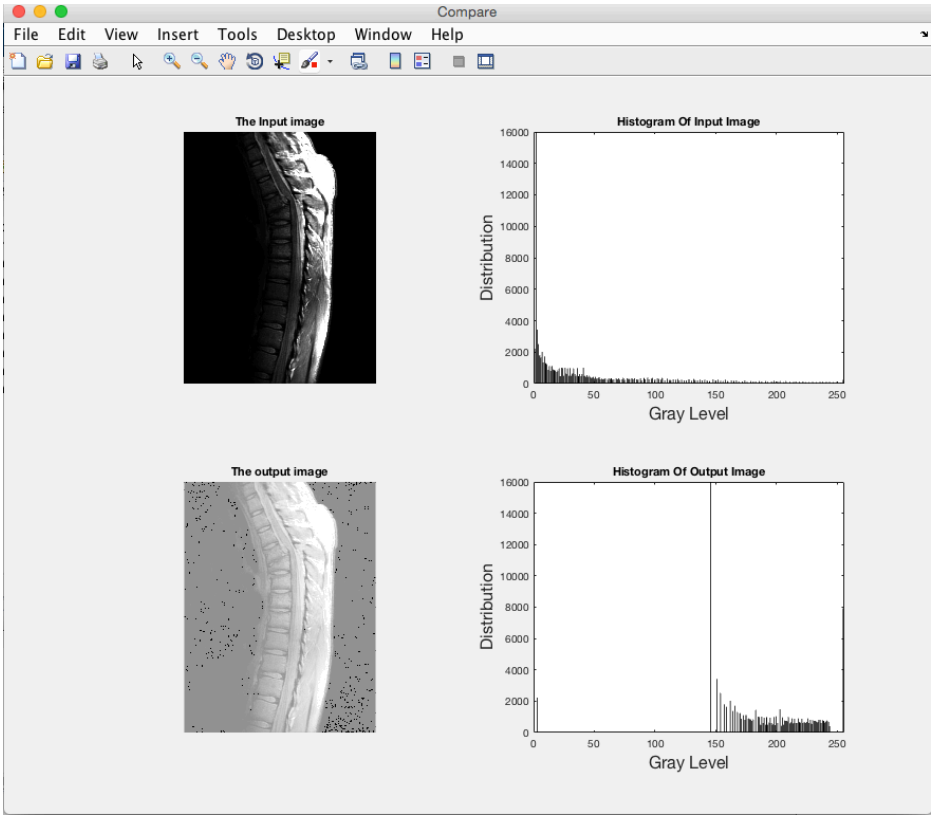
Input Image and output image



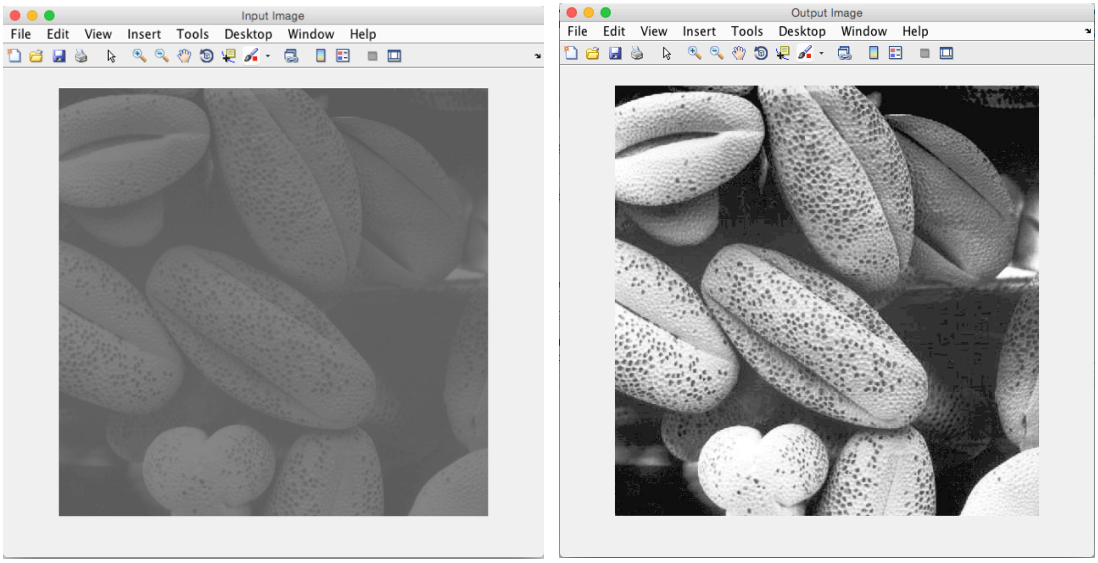




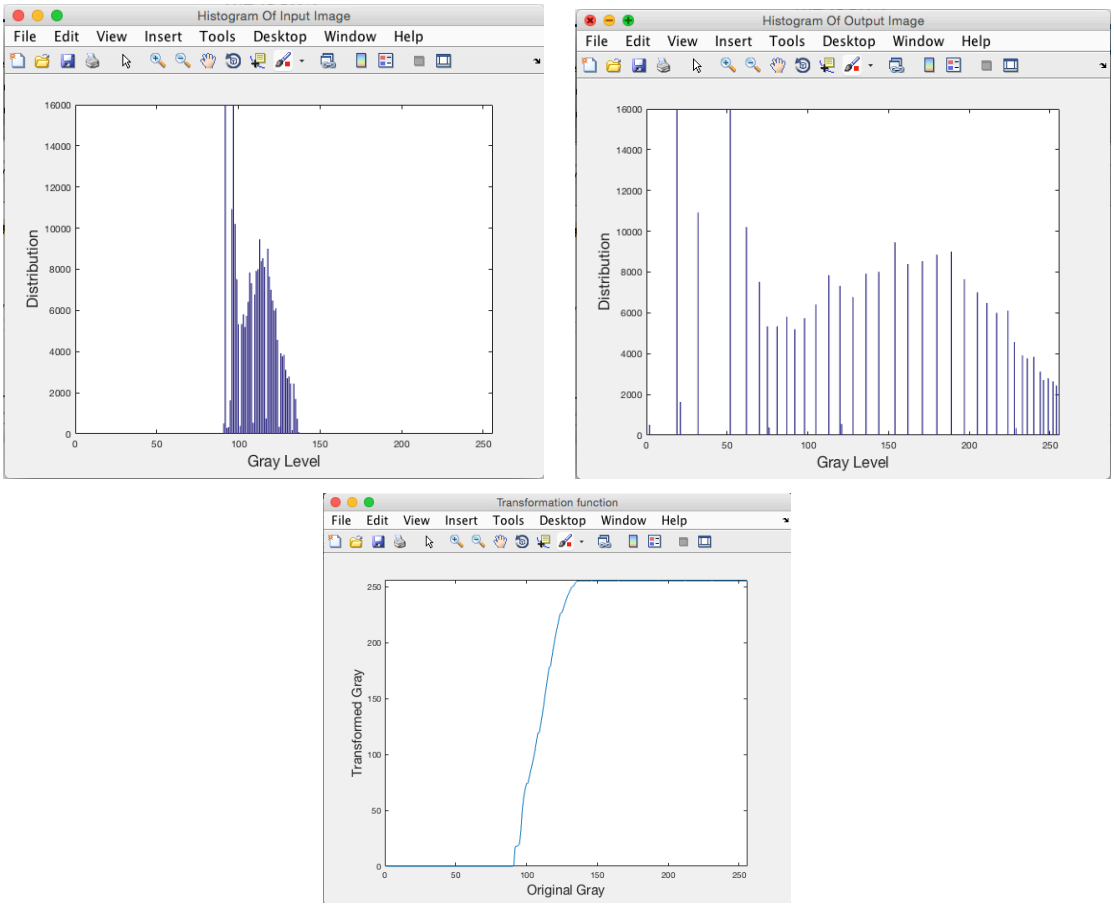
Corresponding histograms and transformation function



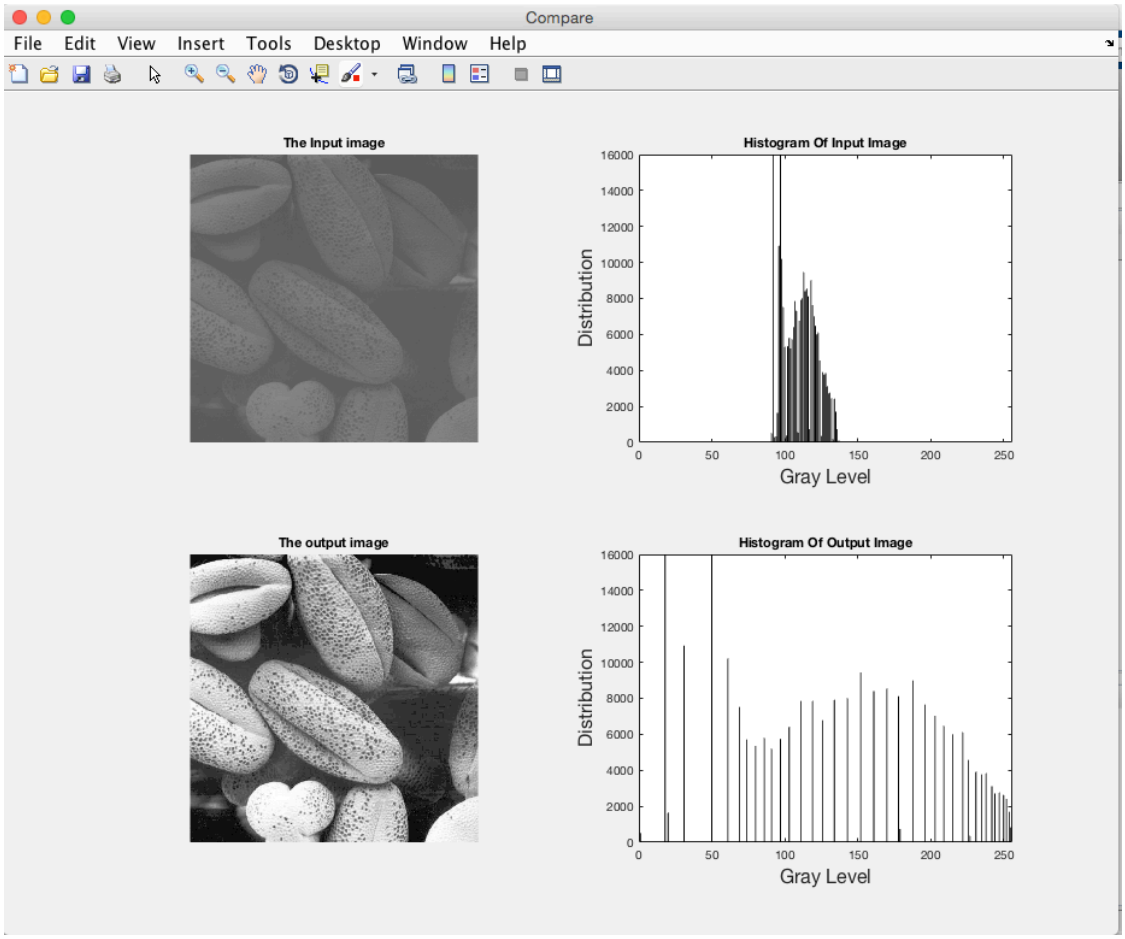
Comparison



Input Image and output image



Corresponding histograms and transformation function


Compare
File Edit View Insert Tools Desktop Window Help
The Input image
Histogram Of Input Image
Distribution
Gray Level
The output image
Histogram Of Output Image
Distribution
Gray Level

Comparison

Project 2 - Combining spatial enhancement methods

Problem

Combining spatial enhancement methods
Implement the image enhancement task of Section 3.7 (Fig 3.43, page 171). The image to be enhanced is skeleton_orig.tif. You should implement all steps in Figure 3.43. (You are encouraged to implement all functions by yourself, not to directly use Matlab functions such as imfilter or fspecial.)

Environment

Matlab R2016a
Atom 1.22 (with language-matlab package)

Principle

- **The Laplacian:**

  We can use the implementation of 2-D, second-order derivatives for image sharpening and the simplest isotropic derivative operator is the Laplacian, which, for a function (image) f(x, y) of two variables, is defined as

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

  In the x-direction and y-direction:

$$\frac{\partial^2 f}{\partial x^2} = f(x+1, y) + f(x-1, y) - 2f(x, y)$$

$$\frac{\partial^2 f}{\partial y^2} = f(x, y+1) + f(x, y-1) - 2f(x, y)$$

  And the discrete Laplacian of two variables:

$$\nabla^2 f(x, y) = f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y)$$

  It can be expressed as mask:

0	1	0	1	1	1
1	−4	1	1	−8	1
0	1	0	1	1	1

- **The Sobel gradient:**

  First derivatives in image processing are implemented using the magnitude of the gradient. For a function f(x, y), the gradient of f at coordinates (x, y) is defined as the two-dimensional column vector:

$$\nabla f \equiv \text{grad}(f) \equiv \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

  The magnitude (length) of vector §f, denoted as M(x, y), where:

$$M(x, y) = \text{mag}(\nabla f) = \sqrt{g_x^2 + g_y^2} \approx |g_x| + |g_y|$$

  So we can get Sobel gradient:

$$g_x = \frac{\partial f}{\partial x} = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)$$

$$g_y = \frac{\partial f}{\partial y} = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)$$

  The masks of Sobel operator:

−1	−2	−1	−1	0	1
0	0	0	−2	0	2
1	2	1	−1	0	1

- **Power-law(Gamma) transformation:**

  Power-law transformations have the basic form:

$$s = cr^{\gamma}$$

  where c and g are positive constants. Sometimes it is written as:

$$s = c(r + \varepsilon)^{\gamma}$$

to account for an offset.

Implementation

1. Clear the former environments at the beginning:

```
% Clear the environment
close all;
clc
clear;
```

2. Input the image and get the information about the image (There the location of the input image is directly assigned for convenience and it also can be assigned by the function parameter after comment this line) :

```
% Set the default location of input image file
img_location = '../images/skeleton_orig.tif';
input_img = double(imread(img_location));

% For this program must be general to allow any
% gray-level image, so it needs to get the depth
% of the image
info = imfinfo(img_location);
L = info.BitDepth;
```

3. Set the laplacian mask , the sobel masks in x-direction and y-direction seperately and the smooth mask in 5 x 5 range:

```
% Set the laplacian and sobel gradient masks also smooth mask
laplacian_mask = [0,-1,0; -1,4,-1; 0,-1,0];
sobel_gradient_x_mask = [-1,-2,-1; 0,0,0; 1,2,1];
sobel_gradient_y_mask = [-1,0,1; -2,0,2; -1,0,1];
smooth_5x5_mask = ones(5,5) * 1/(5*5);
```

4. To filter the input image with the above masks, it needs a new function linear_filter(input_img, mask) to do the linear filter work:

```
% Linear filter
function [output_img] = linear_filter(input_img, mask)
% This function is used to mask to input image
```

   a. First we need padding to avoid the error when mask the boundary, it needs get the padding distance from the mask and use zero padding:

```
% Get the height and width of the mask
[m, n] = size(mask);

% Calculate the size of padding according to the size of mask
padding_m = floor((m - 1) / 2);
padding_n = floor((n - 1) / 2);
% Use zero padding to mask the boundary
padding_input_img = padarray(input_img, [padding_m, padding_n]);
```

   b. Then we can mask the input image with the mask “line by line”, and just return the part without padding:

```
% Get the height and width of the image
[M, N] = size(input_img);

% Initialize and calculate output image
output_img = zeros(M, N);
for i = padding_m + 1:padding_m + M
    for j = padding_n + 1:padding_n + N
        output_img(i - padding_m, j - padding_n) = sum(sum(padding_input_img
        (i-padding_m:i+padding_m,j-padding_n:j+padding_n) .* mask));
    end
end

end
```

5. So we can use the linear filter function to mask the image and “combine” them:

```
% Laplacian of the input image
laplacian_img = linear_filter(input_img, laplacian_mask);

% Use laplacian to sharpen the image
sharpen_laplacian_img = input_img + laplacian_img;

```

6. For sobel gradient mask, we use the sum of absolute values of x-direction and y-direction. We can also use the square sum of the value of x-direction and y-direction which maybe takes more calculation:

```
% Sobel gradient of the input image of direction x and y
sobel_gradient_x_img = linear_filter(input_img, sobel_gradient_x_mask);
sobel_gradient_y_img = linear_filter(input_img, sobel_gradient_y_mask);

% Sobel gradient of the input image it can also use square sum
sobel_gradient_img = abs(sobel_gradient_x_img) + abs(sobel_gradient_y_img);
```

7. Use 5 x 5 smooth mask to smooth:

```
% Use 5x5 mask to smooth
smooth_5x5_sobel_img = linear_filter(sobel_gradient_img, smooth_5x5_mask);

```

8. The product of sharpened laplacian and smoothed sobel:

```
% Calculate the product mask of laplacian and smooth sobel
mask_product_img = sharpen_laplacian_img .* smooth_5x5_sobel_img / (2^L - 1);
sharpen_smooth_sobel_img = input_img + mask_product_img;
```

9. Power-law transformation:

```

% Calculate the power-law transformation of smooth sobel image
power_law_tranform_img = (sharpen_smooth_sobel_img * (2^L - 1)).^ 0.5;

```

10. To show the pictures we need handle the negative grey values:

```
% Increase the gray level to handle the negative gray level
laplacian_img = laplacian_img - min(laplacian_img(:));
sharpen_laplacian_img =  sharpen_laplacian_img - min(sharpen_laplacian_img(:));
sobel_gradient_img = sobel_gradient_img - min(sobel_gradient_img(:));

```

11. Show the pictures and output them:

```
54
55  %%%%%%%%%%%%%%%%%%%%%%%% Show pictures %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
56  figure('NumberTitle','off','Name','Combining spatial enhancement methods')
57  subplot(2,4,1);imshow(input_img,[0,2^L-1]);title('(a) Input image');
58  subplot(2,4,2);imshow(laplacian_img,[0,2^L-1]);title('(b) Laplacian of (a)');
59  subplot(2,4,3);imshow(sharpen_laplacian_img,[0,2^L-1]);title('(c) Sum of (a) and (b)');
60  subplot(2,4,4);imshow(sobel_gradient_img,[0,2^L-1]);title('(d) Sobel of (a)');
61  subplot(2,4,5);imshow(smooth_5x5_sobel_img,[0,2^L-1]);title('(e) Smooth (d) with a 5x5 average filter');
62  subplot(2,4,6);imshow(mask_product_img,[0,2^L-1]);title('(f) Product of (c) and (e)');
63  subplot(2,4,7);imshow(sharpen_smooth_sobel_img,[0,2^L-1]);title('(g) Sum of (a) and (f)');
64  subplot(2,4,8);imshow(power_law_tranform_img,[0,2^L-1]);title('(h) Gamma transformation result of (g)');
65
66  %%%%%%%%%%%%%%%%%%%%%%%% Output pictures %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
67  imwrite(uint8(input_img), '1_origin_image.jpg');
68  imwrite(uint8(laplacian_img), '2_laplacian_of_image.jpg')
69  imwrite(uint8(sharpen_laplacian_img), '3_sharpen_laplacian_image.jpg')
70  imwrite(uint8(sobel_gradient_img), '4_sobel_gradient_image.jpg')
71  imwrite(uint8(smooth_5x5_sobel_img), '5_smooth_sobel_image.jpg')
72  imwrite(uint8(mask_product_img), '6_product_laplacian_sobel.jpg')
73  imwrite(uint8(sharpen_smooth_sobel_img), '7_sharpen_smooth_sobel.jpg')
74  imwrite(uint8(power_law_tranform_img), '8_power_law_transform.jpg')
75
```

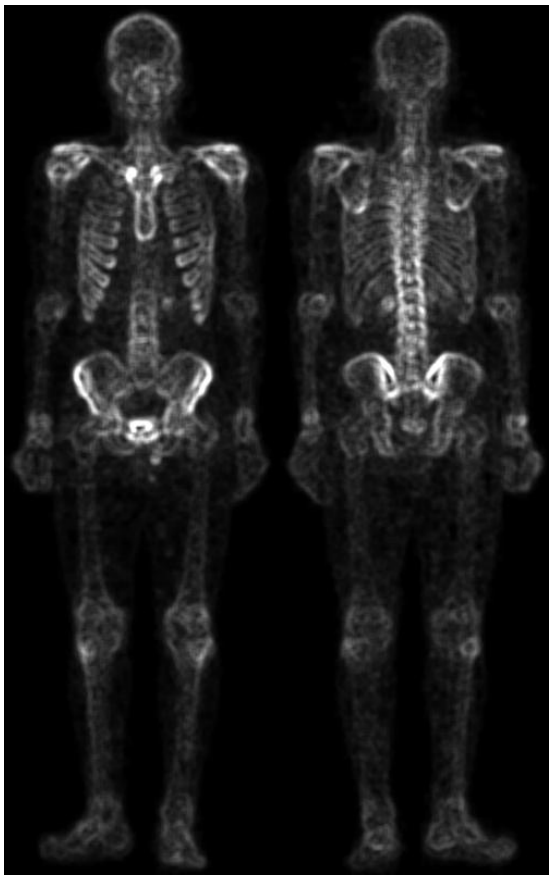
Result

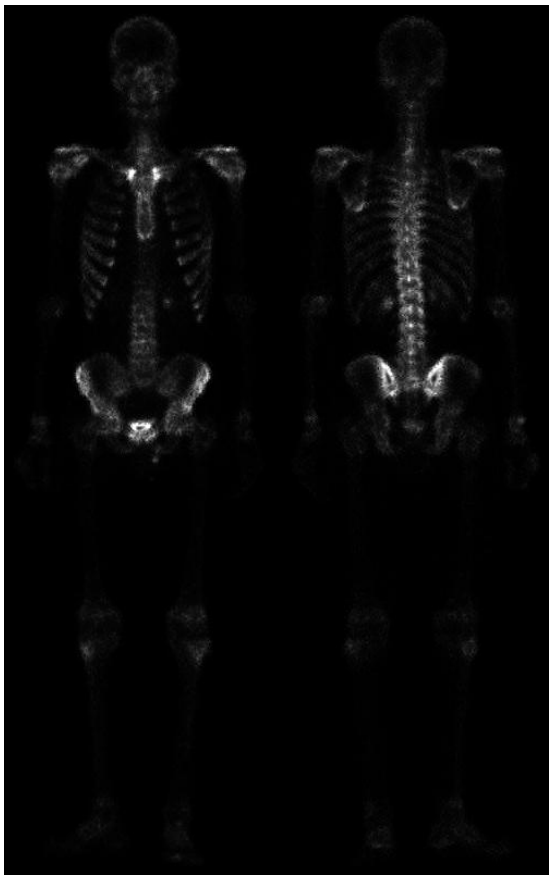
(a) Image of whole body bone scan.

(b) Laplacian of (a)

(c) Sharpened image obtained by adding (a) and (b).

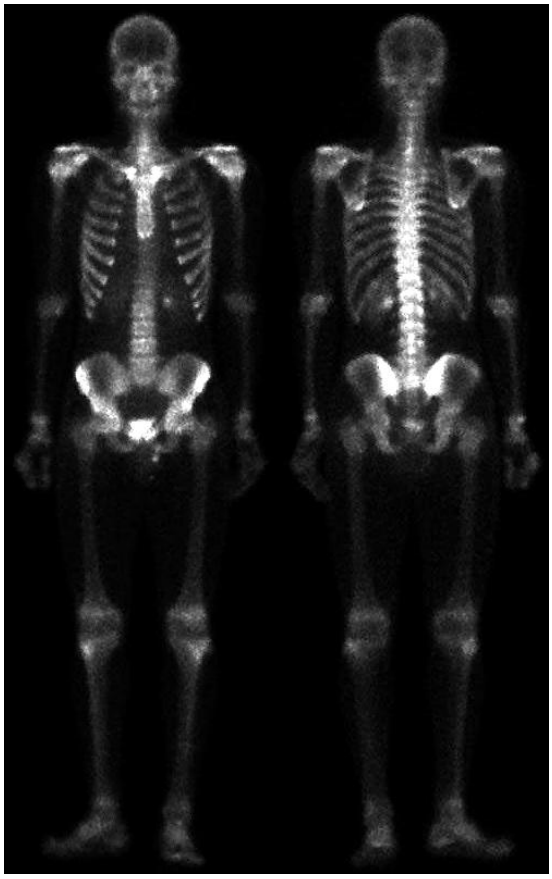
(d) Sobel gradient of (a)





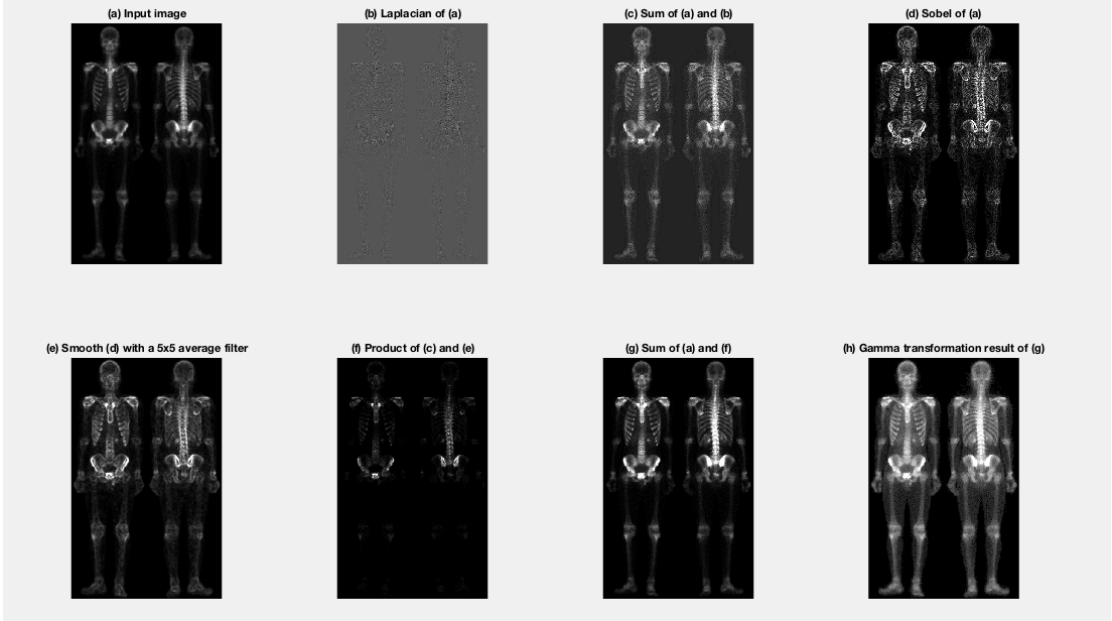
(e) Sobel image smoothed with a 5 * 5 averaging filter.

(f) Mask image formed by the product of (c) and (e).



(g) Sharpened image obtained by the sum of (a) and (f).

(h) Final result obtained by applying a power- law transformation to (g).



Project 3 Filtering in frequency domain

Problem

Filtering in frequency domain
Implement the ideal, Butterworth and Gaussian low-pass and high-pass filters and compare the results under different parameters using the image characters_test_pattern.tif (this image file can be found at the ftp server ftp://ftp.cs.sjtu.edu.cn:990/lu-ht/DIP/images) as the test pattern.

Environment

Matlab R2016a
Atom 1.22 (with language-matlab package)

Principle

- **The Fourier transform**

  We can use the Fourier transform to transform the signal of time domain to frequency domain. The Fourier transform of a continuous function f(t) of a continuous variable, t, denoted F(m), is defined by the equation

$$F(\mu) = \int_{-\infty}^{\infty} f(t)\, e^{-j2\pi\mu t}\, dt$$

  and for the discrete signal:

$$F_m = \sum_{n=0}^{M-1} f_n e^{-j2\pi mn/M} \qquad m = 0, 1, 2, \ldots, M-1$$

  Also the 2-D discrete Fourier transform is defined by

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)\, e^{-j2\pi(ux/M + vy/N)}$$

- **ideal low-pass filter**

$$H(u, v) = \begin{cases} 1 & \text{if } D(u, v) \le D_0 \\ 0 & \text{if } D(u, v) > D_0 \end{cases}$$

$$D(u, v) = \left[(u - P/2)^2 + (v - Q/2)^2\right]^{1/2}$$

- **ideal high-pass filter**

$$H(u,v) = \begin{cases} 0 & \text{if } D(u,v) \le D_0 \\ 1 & \text{if } D(u,v) > D_0 \end{cases}$$

- **Butterworth low-pass filter**

$$H(u,v) = \frac{1}{1 + [D(u,v)/D_0]^{2n}}$$

- **Butterworth high-pass filter**

$$H(u,v) = \frac{1}{1 + [D_0/D(u,v)]^{2n}}$$

- **Gaussian low-pass filter**

$$H(u,v) = e^{-D^2(u,v)/2\sigma^2}$$

- **Gaussian high-pass filter**

$$H(u,v) = 1 - e^{-D^2(u,v)/2D_0^2}$$

Implementation

1. Clear the former environments at the beginning:

```
4  % Clear the environment
5  close all;
6  clc
7  clear;
```

2. Input the image and get the information about the image (There the location of the input image is directly assigned for convenience and it also can be assigned by the function parameter after comment this line) :

```
% Set the default location of input image file
img_location = '../../images/characters_test_pattern.tif';
input_img = double(imread(img_location));

% Get the relative information of the image
info = imfinfo(img_location);
M = info.Height;
N = info.Width;
```

3. Set the tested cutoff values, noise patterns and pass type:

```
% Set the tested cutoff values, noise patterns and pass type
cutoffs = [10, 30, 60, 160, 460];
patterns = {'ideal', 'butterworth', 'gaussian'};
pass_types = {'lowpass', 'highpass'};
```

4. Get the discrete Fourier transform of the input image and shift the dft, in there we use double size of input image for a better precision:

```
% Get the Discrete Fourier Transform
dft = fft2(input_img, 2*M, 2*N);
dft = fftshift(dft);
```

5. For each pattern high/low pass with different cutoffs, use function frequency_filter (frequency_filter.m) to filter the dft, then use ifft to convert the filtered dft to the filtered image. Because we use larger-size dft to get better precision, so we need to crop the image converted back. For convenience and clearance, I define a function frequency_filter_figures to show and output the figures together:

```
% Test each pattern high/low pass with different cutoffs
for pass_type = pass_types
  for pattern = patterns
    % Collect the generated figures to show together later
    figures = {};
    figures{1} = input_img;

    for i = 1:numel(cutoffs)
      % Frequency filter
      filter_dft = frequency_filter(dft, pattern, pass_type, cutoffs(i), 2);
      filter_img = real(ifft2(ifftshift(filter_dft)));

      % For we use bigger dft image so we need to crop the image converted back
      filter_img = filter_img(1:M,1:N);
      figures{i+1} = filter_img;
    end

    % Show and output the generated figures together
    frequency_filter_figures(figures, {pattern, pass_type}, [0,cutoffs], true)
  end
end
```

6. Considering the clear structure and the shortness of main function, I take the filter and figure out from the main function. Compared the codes in Project2, it gets much better to read.
    a. The function frequency_filter() implements all the high/low pass filter patterns, the input needs to be the allowed formats.

```
% Frequency filters
function [ output_img ] = frequency_filter( input_img, pattern, pass_type, D0, n)
  % Input:
  %   input_img - the input image
  %   pattern - the pattern of filter, it need to be the value of next several values:
  %             {'ideal', 'butterWorth', 'gaussian'}
  %   pass_type - high pass or low pass
  %   D0 - the 'distance' D0
  %   n - the parameter used in the butterworth pattern
  % Output:
  %   the image after filtering


% The allowed patterns and high/low pass
patterns = {'ideal', 'butterworth', 'gaussian'};
pass_types = {'lowpass', 'highpass'};

% Convert the data type
input_img = double(input_img);

% Initialize
[M,N] = size(input_img);
H = zeros(M,N);

```

Then calculate the H matrix according to the patterns:

```
switch lower(char(pattern))
  case patterns{1}
    % Ideal
    for i = 1:M
      for j = 1:N
        if sqrt((i - M/2) ^ 2 + (j - N/2) ^ 2) <= D0
          H(i,j) = 1;
        end
      end
    end
  case patterns{2}
    % Butterworth
    for i = 1:M
      for j = 1:N
        H(i,j) = 1 / (1 + (sqrt((i - M/2) ^ 2 + (j - N/2) ^ 2) / D0) ^ (2 * n));
      end
    end
  case patterns{3}
    % Gaussian
    for i = 1:M
      for j = 1:N
          H(i,j) = exp(-(sqrt((i - M/2) ^ 2 + (j - N/2) ^ 2) ^ 2) / (2*(D0^2)));
      end
    end
  otherwise
    error(['The', ' ', pattern, ' ', 'filter pattern can not be found.'])
end
```

And we can easy to know that the “sum” of high pass and the low pass is equal to 1, so we can just calculate one matrix H and just get the other by 1 – H :

```
switch lower(char(pass_type))
  case pass_types{1}
    H = H;
  case pass_types{2}
    H = 1 - H;
end

output_img = H .* input_img;
end
```

b. To reduce the repeated codes, the pictures need to frequency_filter_figures() function to show and output, it needs to input the information concerned to get the titles and files name:

```
% Figures
function [] = frequency_filter_figures( input_imgs, image_info, cutoffs, output_file )
% Show the pictures and the figures

figure('NumberTitle','off','Name', ['The', ' ',char(image_info{1}), ' ',...
                                    char(image_info{2}), ' ', 'filter'])
column = 2;
[temp, num] = size(input_imgs);
row = ceil(num / column);

% Show the pictures in 2 pictures/lines format
% If the output_file flag is set, then output to files
for i = 1:row
  for j = 1:column
    if((i - 1)* column + j) <= num
      subplot(row, column, (i - 1)* column + j)
      imshow(uint8((input_imgs{(i - 1) * column + j})),[]);
      if (i - 1)* column + j == 1
        title('Origin input image')
      else
        title(['Cutoff = ', int2str(cutoffs((i - 1) * column + j))]);
      if output_file
        if (i - 1)* column + j > 1
          imwrite(uint8(input_imgs{(i - 1) * column + j}), [char(image_info{1}),...
           '_',char(image_info{2}), '_', int2str(cutoffs((i - 1) * column + j)), '.jpg']);
        end
      end
    end
  end
end

if output_file
  % Save the whole figure containing six pictures
  saveas(gcf, [char(image_info{1}), '_', char(image_info{2}), '_', 'filter', '.jpg'])
end
end

```

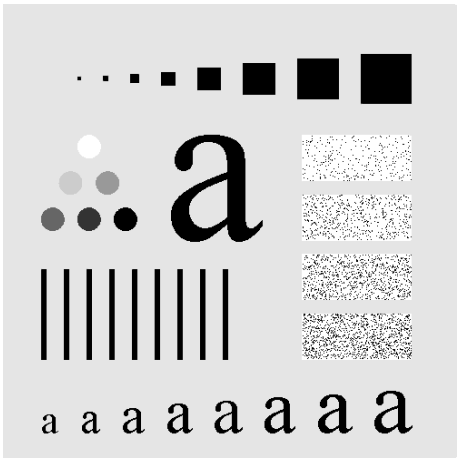
7. *Some small try with the basic DFT function:
At the beginning I want to implement from the basic dft function like this :

```
% Calculate the F(u,v) according to the 2D dft formula
x=0:n*M-1;
y=0:n*N-1;
[x,u]=meshgrid(x,x);
[v,y]=meshgrid(y,y);
output_img = exp(-1i * 2 * pi * (u .* x / M)) * extended_input_img * ...
             exp(-1i * 2 * pi * (v .* y / N));


% For better show the frequency distribution in one picture,
% shift the blocks to focus the high frequencies to the center
if shift_flag == true
  temp = output_img;
  output_img = zeros(M,N);
  output_img(1:floor(M/2), 1:floor(N/2)) = temp((ceil(M/2) + 1):M, (ceil(N/2) + 1):N);
  output_img(1:floor(M/2), ceil(N/2):N) = temp((ceil(M/2) + 1):M, 1:(floor(N/2) + 1));
  output_img(ceil(M/2):M, 1:floor(N/2)) = temp(1:(floor(M/2) + 1), (ceil(N/2) + 1):N);
  output_img(ceil(M/2):M, ceil(N/2):N) = temp(1:(floor(M/2) + 1), 1:(floor(N/2) + 1));
end
```

But it’s so quite slow compared with the fft() function of the matlab so I didn’t use it finally.

Result

1. Ideal lowpass

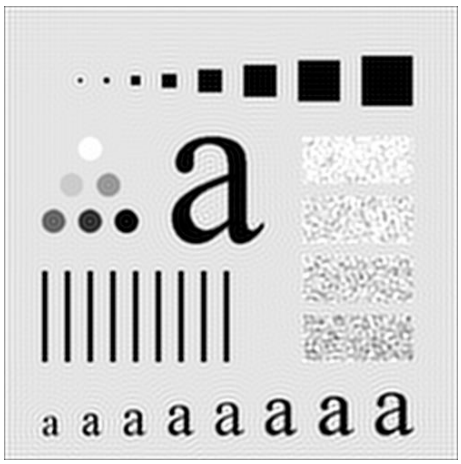


Original input image

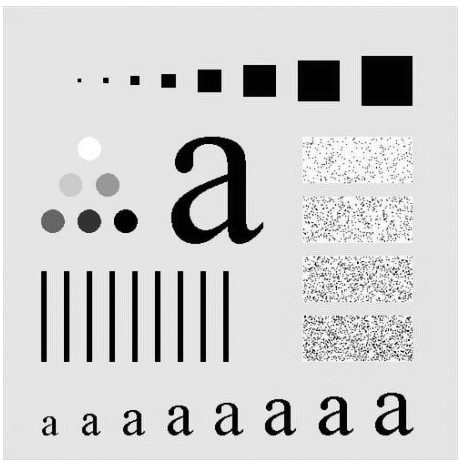
d0 = 10

d0 = 30

d0 = 60

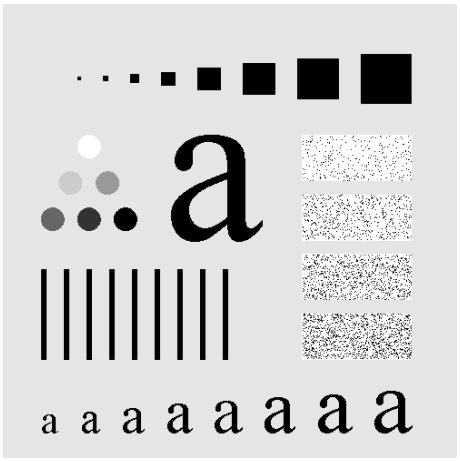


d0 = 160



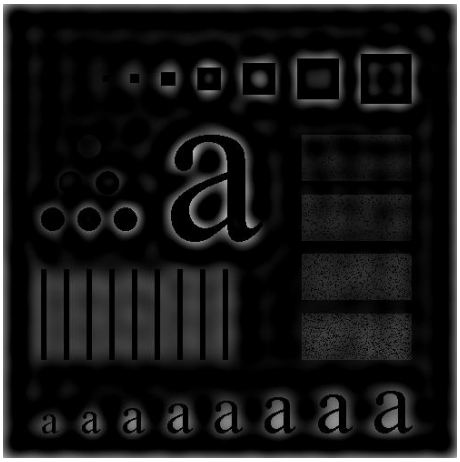
d0 = 460

2. Ideal highpass



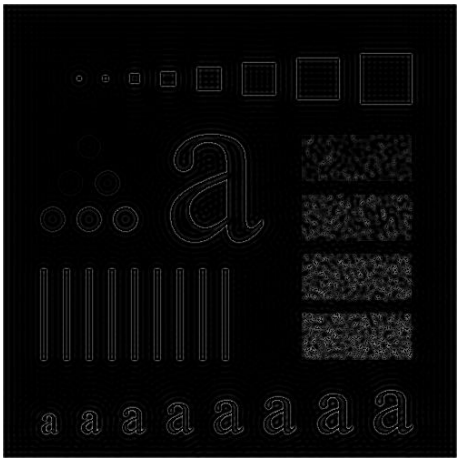
Original input image

d0 = 10

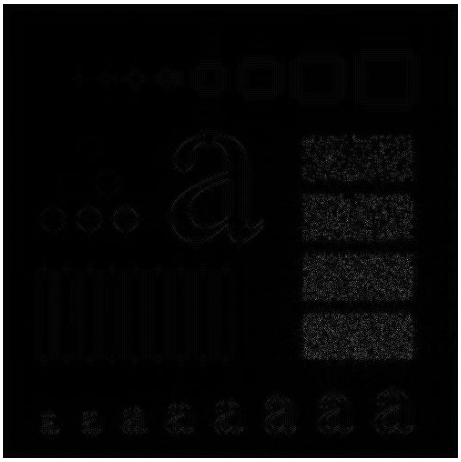


d0 = 30

d0 = 60

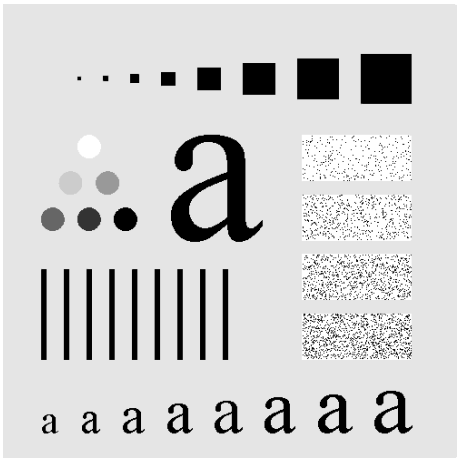


d0 = 160



d0 = 460

3. Butterworth lowpass

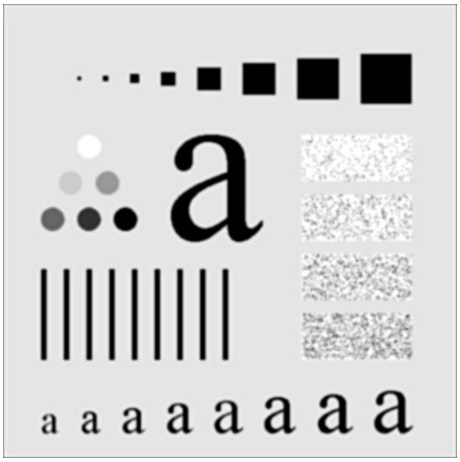


Original input image

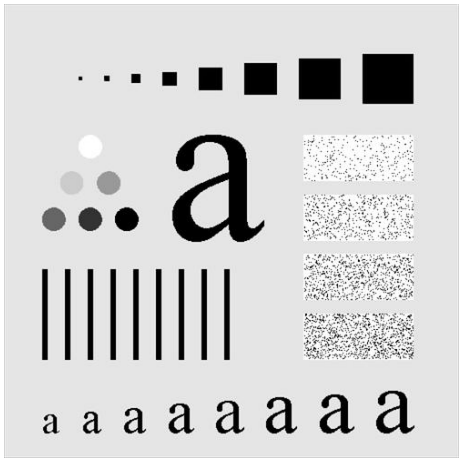
n = 2 d0 = 10

n = 2 d0 = 30

n = 2 d0 = 60

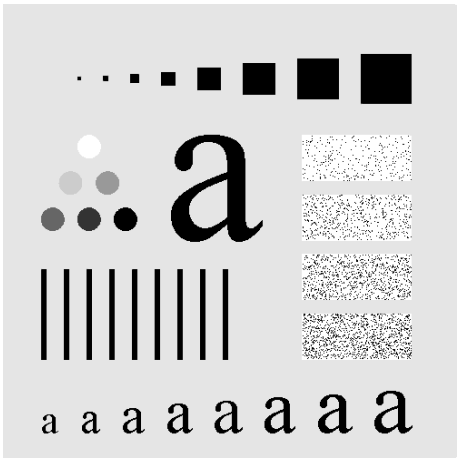


n = 2 d0 = 160



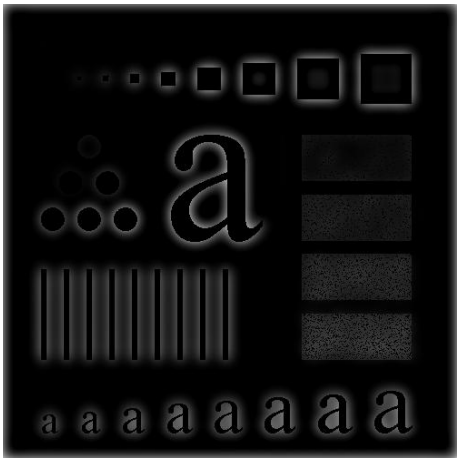
n = 2 d0 = 460

4. Butterworth highpass

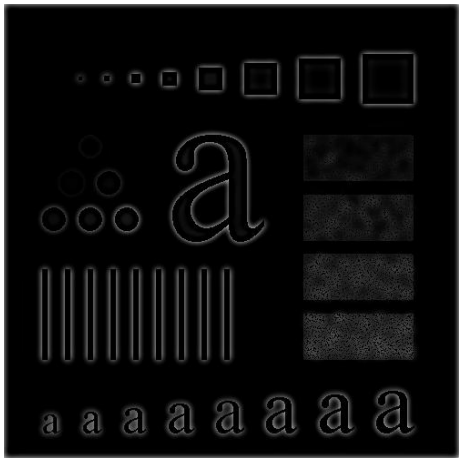


Original input image

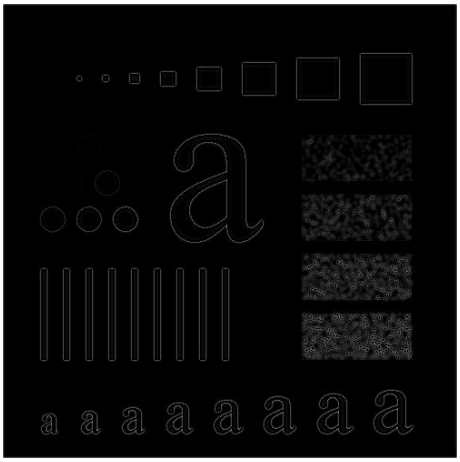
n = 2 d0 = 10



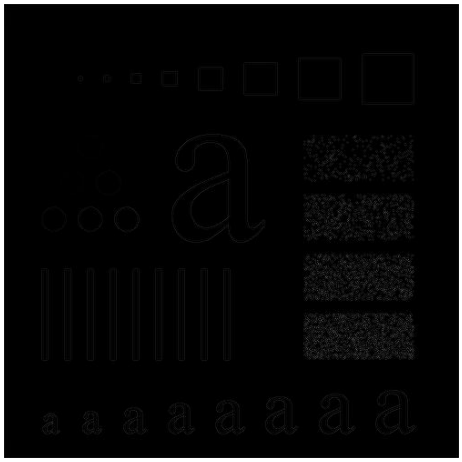
n = 2 d0 = 30



n = 2 d0 = 60

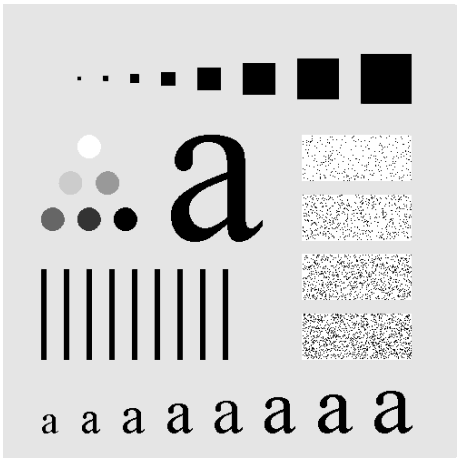


n = 2 d0 = 160



n = 2 d0 = 460

5. Gaussian lowpass

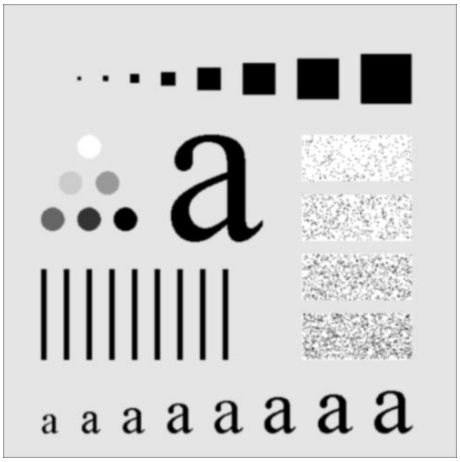


Original input image

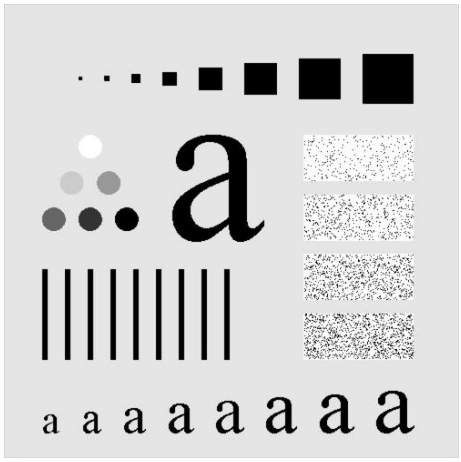
d0 = 10

d0 = 30

d0 = 60

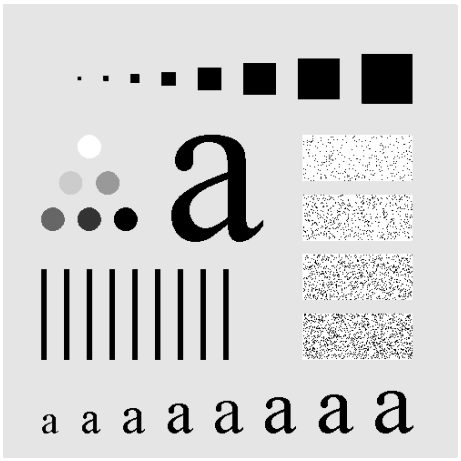


d0 = 160

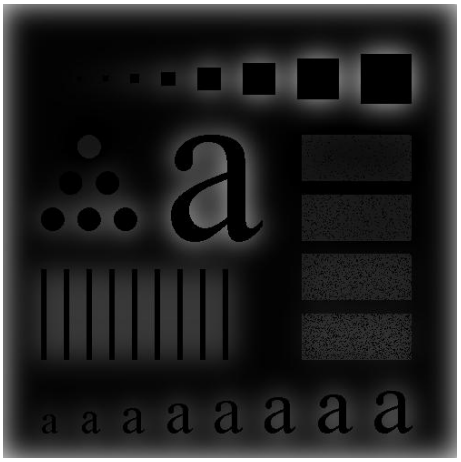


d0 = 460

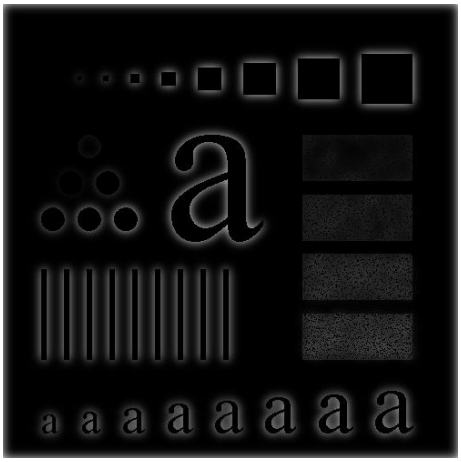
6. Gaussian highpass



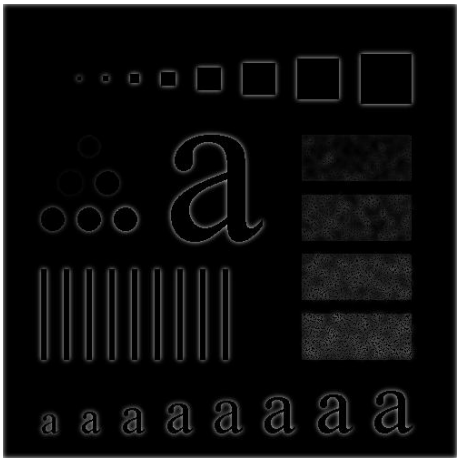
Original input image



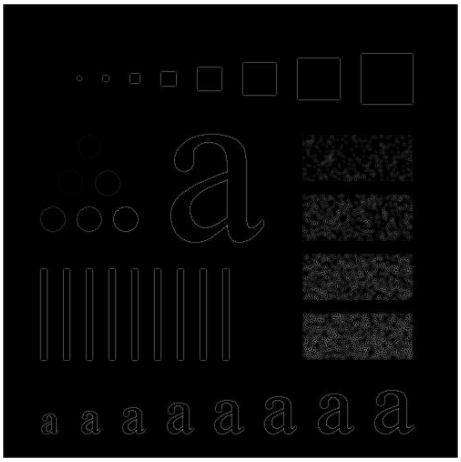
d0 = 10



d0 = 30



d0 = 60



d0 = 160

d0 = 460

Project 4 - Generating different types of noise and comparing different noise reduction methods

Problem

Generating different types of noise and comparing different noise reduction methods In this problem, you are required to write a program to generate different types of random noise (Uniform, Gaussian, Rayleigh, Gamma, Exponential and Impulse, first started from the uniform noise and then use some functions to convert the uniform noise to Gaussian, Rayleigh, Gamma and Exponential; Impulse noise is generated in a different way, consulting the textbook and some other references) and then add these noises to the test patter image Fig0503(original_pattern).tif to compare the visual results of the noisy images. Add some of these noises to the circuit image Circuit.tif (images can be found at ftp://ftp.cs.sjtu.edu.cn:990/lu-ht/DIP/images) and investigate the noise reduction results using different mean filters and order statistics filters as the textbook did at pages 344-352 (Pages 322-329 in the electronic version of the textbook).

Environment

Matlab R2016a
Atom 1.22 (with language-matlab package)

Principle

- **Noise Probability Density Functions**
  - **Gaussian noise**

    The PDF of a Gaussian random variable, z, is given by

$$p(z) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(z-\bar{z})^2/2\sigma^2}$$

    where z represents intensity, z is the mean† (average) value of z, and s is its standard deviation. In matlab, we can use the randn() function to get the Gaussian distribution random numbers so we can get Gaussian noise in this way:

$$z = \bar{z} + \sigma * randn(M, N)$$

- **Rayleigh noise**

  The PDF of Rayleigh noise is given by:

$$p(z) = \begin{cases} \frac{2}{b}(z-a)e^{-(z-a)^2/b} & \text{for } z \geq a \\ 0 & \text{for } z < a \end{cases}$$

  The mean and variance of this density are given by:

$$\bar{z} = a + \sqrt{\pi b/4}$$

  and

$$\sigma^2 = \frac{b(4-\pi)}{4}$$

- **Erlang (gamma) noise**

  The PDF of Erlang noise is given by :

$$p(z) = \begin{cases} \frac{a^b z^{b-1}}{(b-1)!}e^{-az} & \text{for } z \geq 0 \\ 0 & \text{for } z < 0 \end{cases}$$

  The mean and variance of this density are given by:

$$\bar{z} = \frac{b}{a}$$

  and

$$\sigma^2 = \frac{b}{a^2}$$

- **Exponential noise**

  The PDF of exponential noise is given by :

$$p(z) = \begin{cases} ae^{-az} & \text{for } z \geq 0 \\ 0 & \text{for } z < 0 \end{cases}$$

  The mean and variance of this density are given by:

$$\bar{z} = \frac{1}{a}$$

  and

$$\sigma^2 = \frac{1}{a^2}$$

- **Uniform noise**

  The PDF of uniform noise is given by :

$$p(z) = \begin{cases} \frac{1}{b-a} & \text{if } a \le z \le b \\ 0 & \text{otherwise} \end{cases}$$

  The mean and variance of this density are given by:

$$\bar{z} = \frac{a+b}{2}$$

  and

$$\sigma^2 = \frac{(b-a)^2}{12}$$

- **Impulse (salt-and-pepper) noise**

  The PDF of (bipolar) impulse noise is given by :

$$p(z) = \begin{cases} P_a & \text{for } z = a \\ P_b & \text{for } z = b \\ 0 & \text{otherwise} \end{cases}$$

- **Spatial Filtering**
  - **Mean Filters**
    - **Arithmetic mean filter**

$$\hat{f}(x,y) = \frac{1}{mn} \sum_{(s,t) \in S_{xy}} g(s,t)$$

    - **Geometric mean filter**

$$\hat{f}(x,y) = \left[\prod_{(s,t) \in S_{xy}} g(s,t) \right]^{\frac{1}{mn}}$$

    - **Harmonic mean filter**

$$\hat{f}(x,y) = \frac{mn}{\sum_{(s,t)\in S_{xy}} \frac{1}{g(s,t)}}$$

- **Contraharmonic mean filter**

$$\hat{f}(x,y) = \frac{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q+1}}{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q}}$$

- **Order-Statistic Filters**
  - **Median filter**

$$\hat{f}(x,y) = \underset{(s,t)\in S_{xy}}{\text{median}}\{g(s,t)\}$$

  - **Max and min filters**

$$\hat{f}(x,y) = \max_{(s,t)\in S_{xy}} \{g(s,t)\}$$

$$\hat{f}(x,y) = \min_{(s,t)\in S_{xy}} \{g(s,t)\}$$

  - **Midpoint filter**

$$\hat{f}(x,y) = \frac{1}{2}\left[\max_{(s,t)\in S_{xy}} \{g(s,t)\} + \min_{(s,t)\in S_{xy}} \{g(s,t)\}\right]$$

  - **Alpha-trimmed mean filter**

$$\hat{f}(x,y) = \frac{1}{mn-d}\sum_{(s,t)\in S_{xy}} g_r(s,t)$$

Implementation

1. The noise pattern part:
    a. First clean the environment:

```
%% %%%%%%%%%%%%%%%%%%%% The noise patterns part %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Clear the environment
clc
clear;
```

    b. Then input the image file and get the relative information:

```
% Set the default location of input image file that needs to add noise
noise_img_location = '../../images/Fig0503 (original_pattern).tif';
input_noise_img = double(imread(noise_img_location));

% Get the relative information of the i
noise_img_info = imfinfo(noise_img_location);
M = noise_img_info.Height;
N = noise_img_info.Width;
% For this program must be general to allow any
% gray-level image, so it needs to get the depth
% of the image
L = noise_img_info.BitDepth;
```

    c. For each pattern, assign the corresponding paramaters a and b, and use function noise_generator() to add the noise to the input image. To show the histogram of the noise image, use histogram drawing function in the Project1:

```
% Generating the noise and add them to the image that needs to add noise
a = 0; b = 20;
uniform_noise_img = noise_generator(input_noise_img, 'uniform', L,a,b);
uniform_noise_histogram = histogram(uniform_noise_img, L);

a = 0; b = 100;
gaussian_noise_img = noise_generator(input_noise_img, 'gaussian', L,a,b);
gaussian_noise_histogram = histogram(gaussian_noise_img, L);

a = 0; b = 400;
rayleigh_noise_img = noise_generator(input_noise_img, 'rayleigh', L,a,b);
rayleigh_noise_histogram = histogram(rayleigh_noise_img, L);

a = 0.1; b = 0.1;
impulse_noise_img = noise_generator(input_noise_img, 'impulse', L,a,b);
impulse_noise_histogram = histogram(impulse_noise_img, L);

a = 0.1; b = 1;
exponential_noise_img = noise_generator(input_noise_img, 'exponential', L,a,b);
exponential_noise_histogram = histogram(exponential_noise_img, L);

a = 0.1; b = 5;
gamma_noise_img = noise_generator(input_noise_img, 'gamma', L,a,b);
gamma_noise_histogram = histogram(gamma_noise_img, L);

input_img_histogram = histogram(input_noise_img, L);
```

    d. Use the function noise_figure() to show the pictures and figures:

```
%%%%%%%%%%%%%%%%%%%% Show pictures & Draw graphs %%%%%%%%%%%%%%%%%%%%%%%%%

noise_figure(input_noise_img, input_img_histogram, 'Input Image', L, true);
noise_figure(uniform_noise_img, uniform_noise_histogram, 'Uniform noise', L, true);
noise_figure(gaussian_noise_img, gaussian_noise_histogram, 'Gaussian noise', L, true);
noise_figure(rayleigh_noise_img, rayleigh_noise_histogram, 'Rayleigh noise', L, true);
noise_figure(impulse_noise_img, impulse_noise_histogram, 'Impulse noise', L, true);
noise_figure(exponential_noise_img, exponential_noise_histogram, 'Exponential noise', L, true);
noise_figure(gamma_noise_img, gamma_noise_histogram, 'Gamma noise', L, true);
```

2. The noise generator function:
    a. First define the permitted pattern values:

```
% Define the noise patterns can be generated
patterns = {'uniform', 'gaussian', 'rayleigh', 'gamma', 'exponential', 'impulse'};
```

    b. Then get the height and width of the input image, and initialize the noise and output image:

```
% Get the height and width of the input image
[M,N] = size(input_img);

% Initialize the noise and output image matrice
noise = zeros(M,N);
output_img = zeros(M,N);
```

    c. Accoding to the pattern, calculate the noise to be added to the input image. To avoid the negative values in the impulse image, there use some preprocessing on the output image matrix to make sure that the minimum value is zero not negative:

```
% Generate the noise accodring to the pattern and the random matrice rand(M,N)
switch lower(pattern)
  case patterns{1}
    % The uniform noise
    noise = a + (b - a) * rand(M, N);
  case patterns{2}
    % The Gaussian noise
    noise = a + sqrt(b) * randn(M, N);
  case patterns{3}
    % The Rayleigh noise
    noise = a + sqrt(-b * log(1 - rand(M, N)));
  case patterns{4}
    % The Gamma or we say Erlang noise
    for i = 1:b
      noise = noise + (-1 / a) * log(1 - rand(M, N));
    end
  case patterns{5}
    % The Exponential noise
    noise = (-1 / a) * log(1 - rand(M, N));
  case patterns{6}
    % The impulse or we say Salt and Pepper noise
    random = rand(M, N);
    noise(random < a) = 1;
    noise(random >= a & random < (a + b)) = 2^depth - 1;
    output_img(random < a) = -input_img(random < a);
    output_img(random >= a & random < (a + b)) = -input_img(random >= a & random < (a + b));
  otherwise
    error(['The', ' ', pattern, ' ', 'noise pattern can not be found.'])
end
output_img = input_img + noise + output_img;
```

3. The histogram function:

```
% Histogram of input images
function [ img_histogram ] = histogram(input_img,  depth)
% Used to generate the histogram of the input image
% Similiar with the project 1
input_img = uint8(input_img);
img_histogram = zeros(2^depth,1);
for i = 1:2^depth
    img_histogram(i) = length(find(input_img == i));
end

end  % function
```

4. The noise figure function:

```
% Show the noise image and draw its histogram graph
function [] = noise_figure( input_img, histogram, img_title, depth, output_file)
% If needs output the image file, set the output_file to be True

figure('NumberTitle', 'off', 'Name', ['The ', ' ', lower(img_title)])
subplot(1,2,1)
imshow(uint8(input_img));
title(img_title)
subplot(1,2,2)
bar(histogram, 0.7);
xlabel('Gray Level','FontSize',16);
ylabel('Distribution','FontSize',16);
axis square
set(gca, 'XLim',[0 2^depth]);
set(gca, 'YLim',[0 3000]);
if (output_file == true)
  saveas(gcf, [img_title, '.jpg'])
end
end
```

5. The reduciton filter part:
    a. First clean the formal environment and get the image and informations:

```
% %% %%%%%%%%%%%%%%%%%%%% The reduction filter part %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Clear the environment
clc
clear;

% the image that that to show the result of noise reduction
reduction_img_location = '../../images/Circuit.tif';
input_reduction_img = double(imread(reduction_img_location));

% Get the relative information of the i
reduction_img_info = imfinfo(reduction_img_location);
M = reduction_img_info.Height;
N = reduction_img_info.Width;
% For this program must be general to allow any
% gray-level image, so it needs to get the depth
% of the image
L = reduction_img_info.BitDepth;
```

    b. Then we need to add the some noise to the image, and use the function spatial_filter() to reduce the noise use different spatial filters. Also use figure function noise_reduction_figure() to show the pictures and figures.

For this part, it is asked to investigate the noise reduction results using different mean filters and order statistics filters as the textbook did at pages 322-329, so I implemented the process according to the book and show the results according to the figures in the book as following:

i. Figure 5.7

```
%%%%%%%%%% Mean filters %%%%%%%%%%%%%%%%

% FIGURE 5.7 –
% (a) Original input image.
% (b) Image corrupted by additive Gaussian noise.
% (c) Filtering with an arithmetic mean filter of size 3 * 3.
% (d) Filtering with a geometric mean filter of size 3 * 3.
a = 0; b = 400;
gaussian_noise_img_400 = noise_generator(input_reduction_img, 'gaussian', L,a,b);
arithmetic_mean_reduction = spatial_filter(gaussian_noise_img_400, 'arithmetic_mean', 3, 3, 0);
geometric_mean_reduction = spatial_filter(gaussian_noise_img_400, 'geometric_mean', 3, 3, 0);

noise_reduction_figure('Figure 5.7',...
{input_reduction_img, gaussian_noise_img_400, arithmetic_mean_reduction, geometric_mean_reduction},...
{'Input image', 'Gaussian noise', 'Arithmetic mean', 'Geometric mean'},...
true);
```

ii. Figure 5.8

```
% FIGURE 5.8 – P325
% (a) Image corrupted by pepper noise with a probability of 0.1.
% (b) Image corrupted by salt noise with a probability of 0.1.
% (c) Filtering (a) with a 3 * 3 contraharmonic filter of order 1.5.
% (d) Filtering (b) with a 3 * 3 contraharmonic filter of order -1.5.
a = 0.1; b = 0;
pepper_noise_img = noise_generator(input_reduction_img, 'impulse', L,a,b);
a = 0; b = 0.1;
salt_noise_img = noise_generator(input_reduction_img, 'impulse', L,a,b);
contraharmonic_mean_reduction_pepper = spatial_filter(pepper_noise_img, 'contraharmonic_mean', 3, 3, 1.5);
contraharmonic_mean_reduction_salt = spatial_filter(salt_noise_img, 'contraharmonic_mean', 3, 3, -1.5);
noise_reduction_figure('Figure 5.8',...
{pepper_noise_img, salt_noise_img, contraharmonic_mean_reduction_pepper, contraharmonic_mean_reduction_salt},...
{'Pepper noise', 'Salt noise', 'Contraharmonic mean Q=1.5', 'Contraharmonic mean Q=-1.5'},...
true);
```

iii. Figure 5.9

```
% FIGURE 5.9 – P326
% Results of selecting the wrong sign in contraharmonic filtering.
% (a) Filtering Fig. 5.8(a) with a contraharmonic filter of size 3 * 3 and Q = -1.5.
% (b) Filtering Fig. 5.8(b) with a contraharmonic filter of size 3 * 3 and Q = 1.5.
wrong_contraharmonic_mean_reduction_pepper = spatial_filter(pepper_noise_img, 'contraharmonic_mean', 3, 3, -1.5);
wrong_contraharmonic_mean_reduction_salt = spatial_filter(salt_noise_img, 'contraharmonic_mean', 3, 3, 1.5);
noise_reduction_figure('Figure 5.9',...
{wrong_contraharmonic_mean_reduction_pepper, wrong_contraharmonic_mean_reduction_salt},...
{'Wrong contraharmonic mean Q=-1.5', 'Wrong contraharmonic mean Q=1.5'},...
true);
```

iv. Figure 5.10

```
% FIGURE 5.10 – P328
% (a) Image corrupted by salt-and-pepper noise with probabilities Pa=Pb=0.1.
% (b) Result of one pass with a median filter of size 3 * 3.
% (c) Result of processing (b) with this filter.
% (d) Result of processing (c) with the same filter.
a = 0.1; b = 0.1;
impulse_noise_img = noise_generator(input_reduction_img, 'impulse', L,a,b);
median_reduction_1times = spatial_filter(impulse_noise_img, 'median', 3, 3, 0);
median_reduction_2times = spatial_filter(median_reduction_1times, 'median', 3, 3, 0);
median_reduction_3times = spatial_filter(median_reduction_2times, 'median', 3, 3, 0);
noise_reduction_figure('Figure 5.10',...
{impulse_noise_img, median_reduction_1times, median_reduction_2times, median_reduction_3times},...
{'Pepper and salt noise', 'Median filter 1 times', 'Median filter 2 times', 'Median filter 3 times'},...
true);
```

v. Figure 5.11

```
% FIGURE 5.11 – P328
% (a) Result of filtering Fig. 5.8(a) with a max filter of size 3 * 3.
% (b) Result of filtering 5.8(b) with a min filter of the same size.
max_reduction = spatial_filter(pepper_noise_img, 'max', 3, 3, 0);
min_reduction = spatial_filter(salt_noise_img, 'min', 3, 3, 0);
noise_reduction_figure('Figure 5.11',...
{max_reduction, min_reduction},...
{'Max filter', 'Min filter'},...
true);
```

vi. Figure 5.12

```
% FIGURE 5.12 – P329
% (a) Image corrupted by additive uniform noise.
% (b) Image additionally corrupted by additive salt-and-pepper noise. Image (b) filtered with a 5 * 5:
% (c) arithmetic mean filter;
% (d) geometric mean filter;
% (e) median filter;
% (f) alpha-trimmed mean filter with d = 5.
a = -20;  b = 20;
uniform_noise_img = noise_generator(input_reduction_img, 'uniform', L,a,b);
a = 0.1; b = 0.1;
impulse_uniform_noise_img = noise_generator(uniform_noise_img, 'impulse', L,a,b);
arithmetic_mean_reduction_5x5 = spatial_filter(impulse_uniform_noise_img, 'arithmetic_mean', 5, 5, 0);
geometric_mean_reduction_5x5 = spatial_filter(impulse_uniform_noise_img, 'geometric_mean', 5, 5, 0);
median_reduction_5x5 = spatial_filter(impulse_uniform_noise_img, 'median', 5, 5, 0);
alpha_trimmed_mean_reduction_5x5 = spatial_filter(impulse_uniform_noise_img, 'alpha_trimmed_mean', 5, 5, 5);
noise_reduction_figure('Figure 5.12',...
{uniform_noise_img, impulse_uniform_noise_img, arithmetic_mean_reduction_5x5, ...
geometric_mean_reduction_5x5, median_reduction_5x5, alpha_trimmed_mean_reduction_5x5},...
{'Uniform noise', 'Impulse noise added to uniform noise', 'Arithmetic mean 5x5', ...
'Geometric mean 5x5', 'Median filter 5x5', 'Alpha trimmed mean 5x5'},...
true);
```

6. The spatial filter function:
   a. First get the height and width of the input image also the mask, and padding the image with 0:

```
% Calculate the size of padding according to the size of mask
padding_m = floor((m - 1) / 2);
padding_n = floor((n - 1) / 2);

% Use zero padding to mask the boundary
padding_input_img = padarray(input_img, [padding_m, padding_n]);

% Get the height and width of the image
[M, N] = size(input_img);
```

   b. Then mask the padding input image with assigned pattern mask:

```
% Initialize and calculate output image
output_img = zeros(M, N);
for i = padding_m + 1:padding_m + M
    for j = padding_n + 1:padding_n + N
        img_in_mask = padding_input_img(i - padding_m:i + padding_m, j - padding_n:j+padding_n);
        output_img(i - padding_m, j - padding_n) = mask(img_in_mask, pattern, m, n, parameter);
    end
end
end
```

for the clear structure of the code, I take the mask process out from the main fucntion to be a saperate function mask(), it implements the detailed procession of the pixels in the mask according to the pattern:

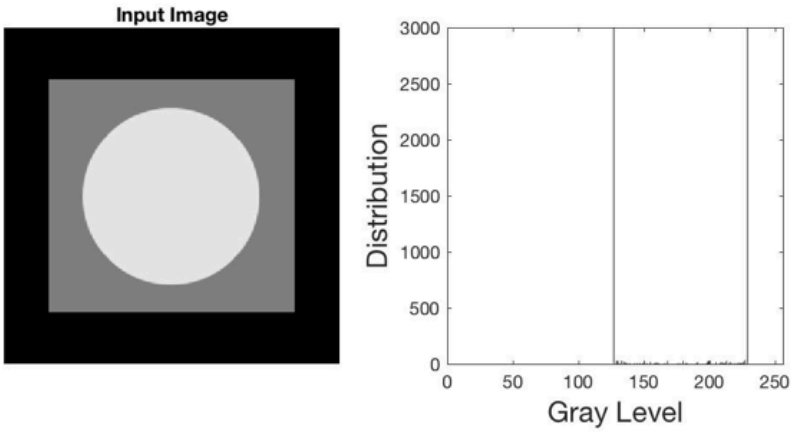
```
patterns = {'arithmetic_mean', 'geometric_mean', 'harmonic_mean', 'contraharmonic_mean',...
   'alpha_trimmed_mean', 'median', 'max', 'min', 'midpoint'};

switch lower(pattern)
    case patterns{1}
        mask_result = sum(sum(img_in_mask)) / (m * n);
    case patterns{2}
        mask_result = prod(prod(img_in_mask)) ^ (1 / (m * n));
    case patterns{3}
        mask_result = (m * n) / (sum(sum(1 ./ img_in_mask)));
    case patterns{4}
        % mask_result = sum(img_in_mask .^ (parameter + 1)) / sum(img_in_mask .^ parameter);
        mask_result = sum(sum(img_in_mask .^ (parameter + 1))) / sum(sum(img_in_mask .^ parameter));
    case patterns{5}
        sorted_pixels = sort(img_in_mask);
        low = round(parameter / 2);
        high = parameter - low;
        sorted_pixels = sorted_pixels(low : m * n - high - 1);
        mask_result = sum(sum(sorted_pixels)) / (m * n - parameter);
    case patterns{6}
        mask_result = median(median(img_in_mask));
    case patterns{7}
        mask_result = max(max(img_in_mask));
    case patterns{8}
        mask_result = min(min(img_in_mask));
    case patterns{9}
        mask_result = (max(max(img_in_mask)) + min(min(img_in_mask))) / 2;
    otherwise
        error(['The', ' ', pattern, ' ', 'filter can not be found.'])
end
end
```

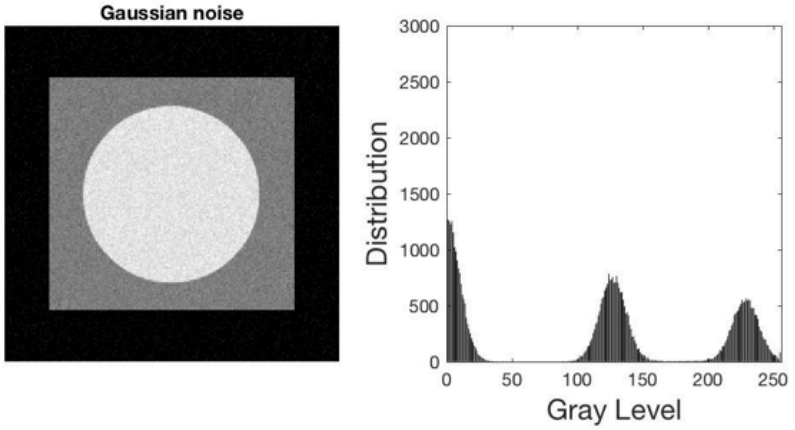
7. The noise reduction figure function:

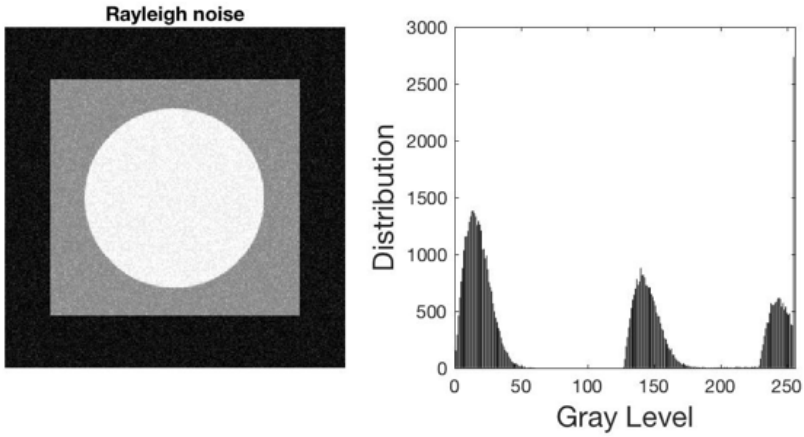
```
% Show the noise images before and after spatial filter
function [] = noise_reduction_figure(figure_title, images, titles, output_file)
% If needs output the image file, set the output_file to be True

[temp, num] = size(images);

column = 2;
row = floor((num + 1)/ column);

figure('NumberTitle','off','Name',figure_title)
for i = 0:row-1
    for j = 1:column
        if (i * column + j <= num)
            subplot(row, column, i * column + j );
            imshow(uint8(images{i * column + j}));
            title(titles{i * column + j});
            if (output_file == true)
              imwrite(uint8(images{i * column + j}), [figure_title, '-', titles{i * column + j}, '.jpg']);
            end
        end
    end
end
if (output_file == true)
  saveas(gcf, [figure_title, '.jpg'])
end
end

```

Result

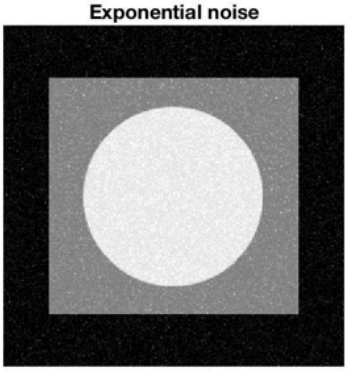
The different-pattern noises:

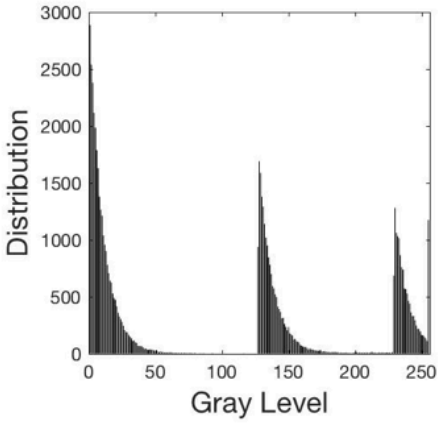

Input Image
Distribution
Gray Level
0
500
1000
1500
2000
2500
3000
0
50
100
150
200
250


Gaussian noise
Distribution
Gray Level
0
500
1000
1500
2000
2500
3000
0
50
100
150
200
250

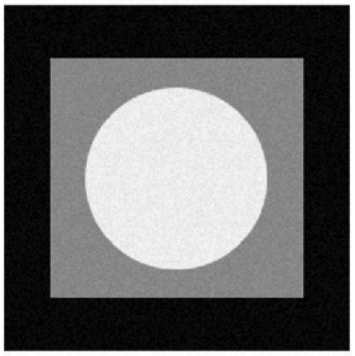

Rayleigh noise
Distribution
Gray Level
0
500
1000
1500
2000
2500
3000
0
50
100
150
200
250

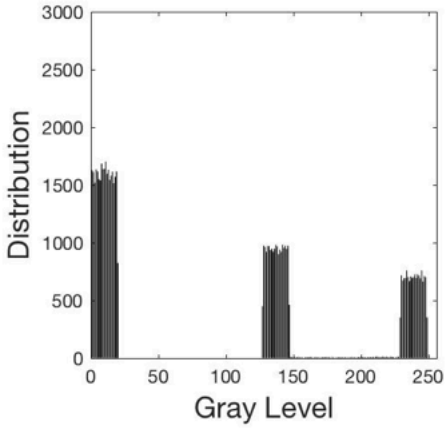
Exponential noise



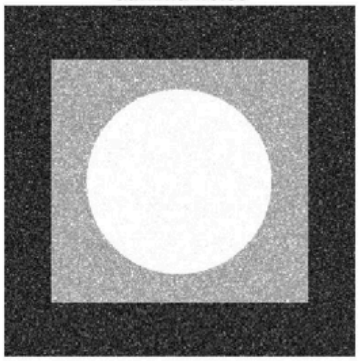


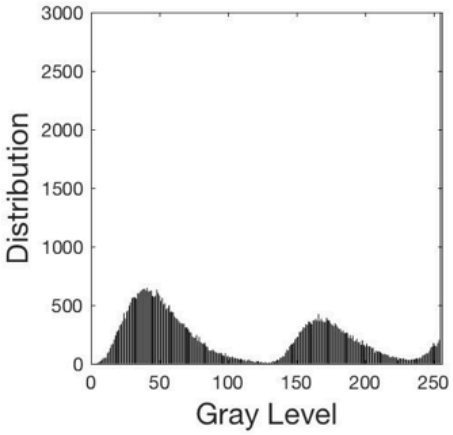
Uniform noise

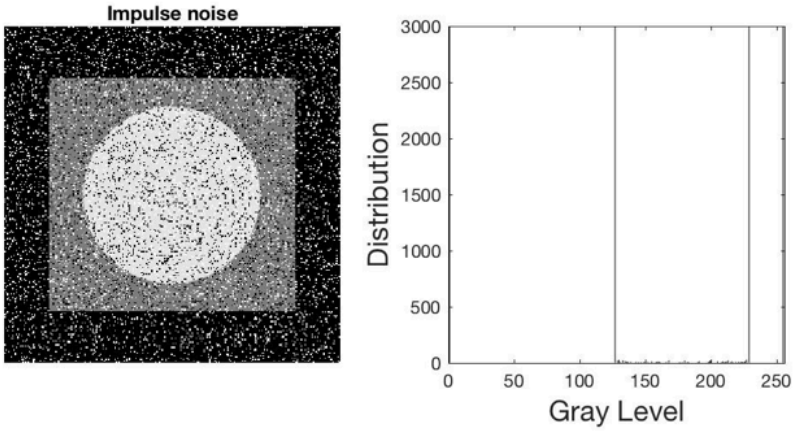




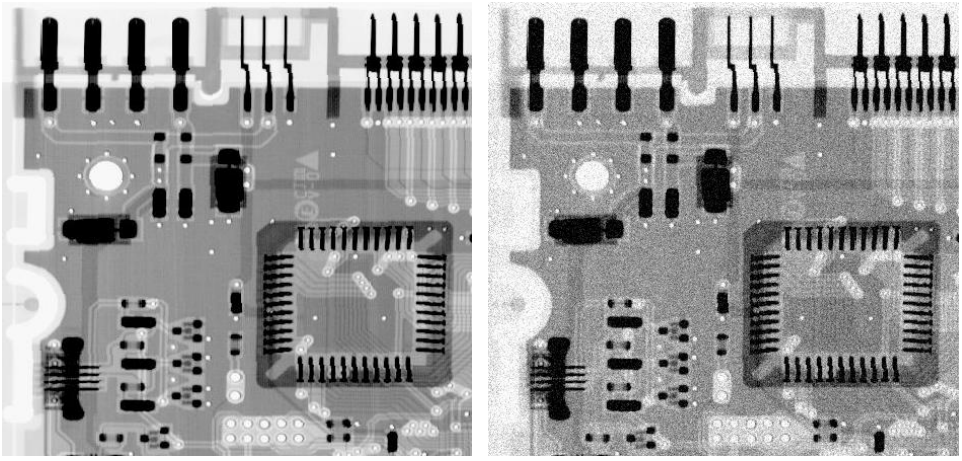
Gamma noise







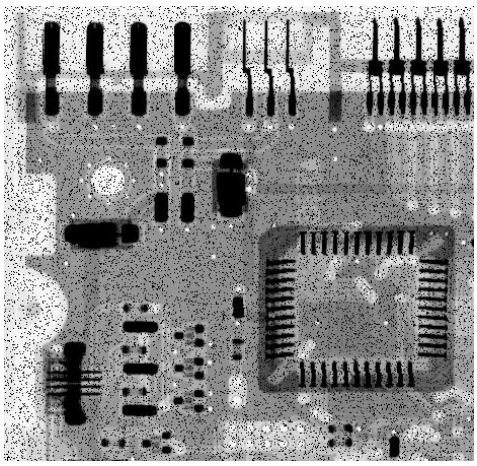
The spatial filter reduction:



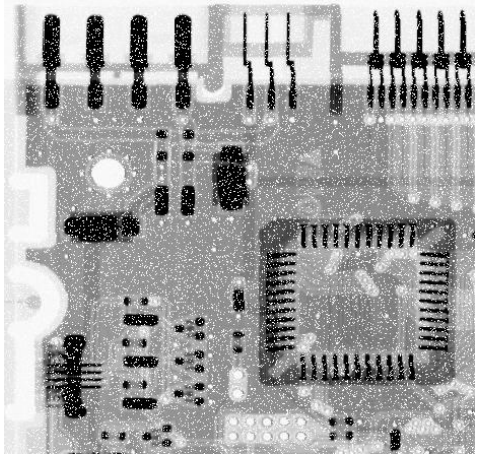
(a) Original input image (b) Add Gaussian noise

(c) Arithmetic filter with 3x3 mask (d) Geometric filter with 3x3 mask

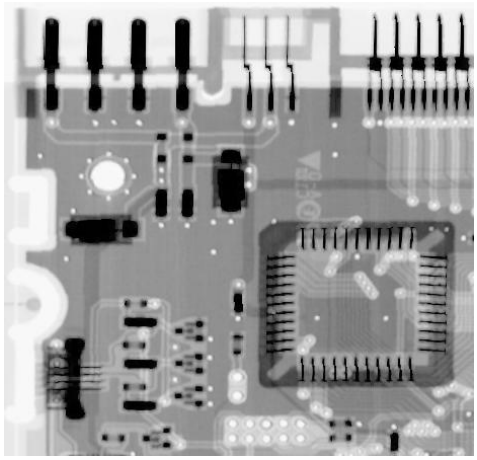
Figure 5.7 (P324)



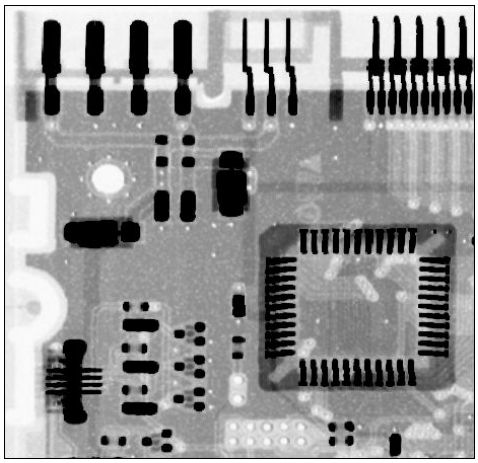
(a) Pepper noise with probability 0.1



(b) Salt noise with probability 0.1

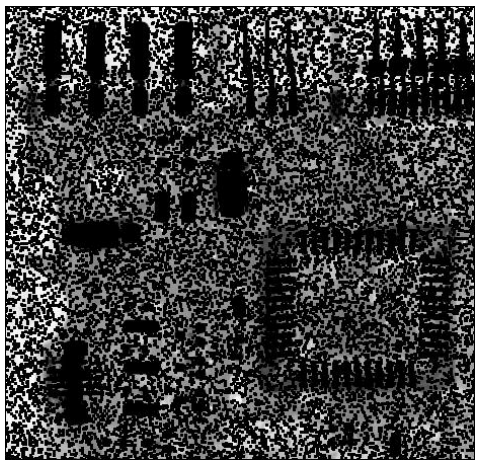


(c) contraharmonic filter order 1.5

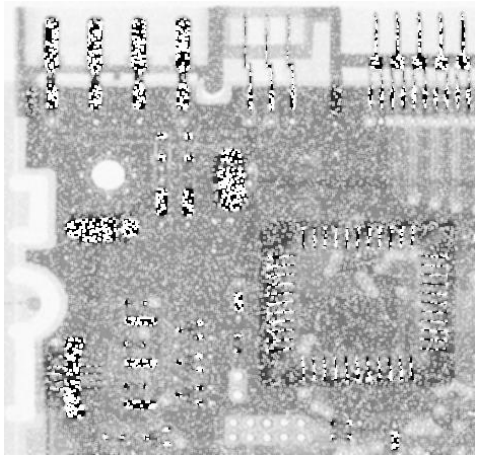


(d) contraharmonic filter order -1.5

Figure 5.8 (P325)

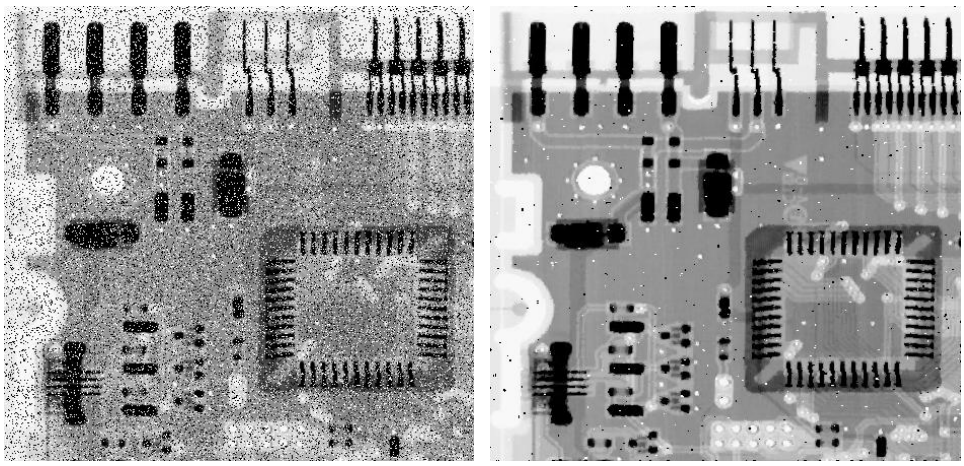


(a) contraharmonic filter order -1.5

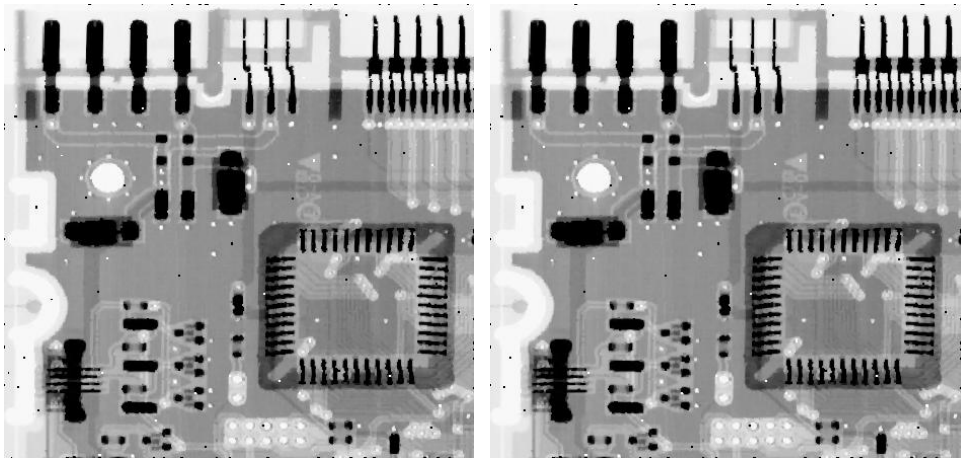


(b) contraharmonic filter order 1.5

Figure 5.9 (P326)

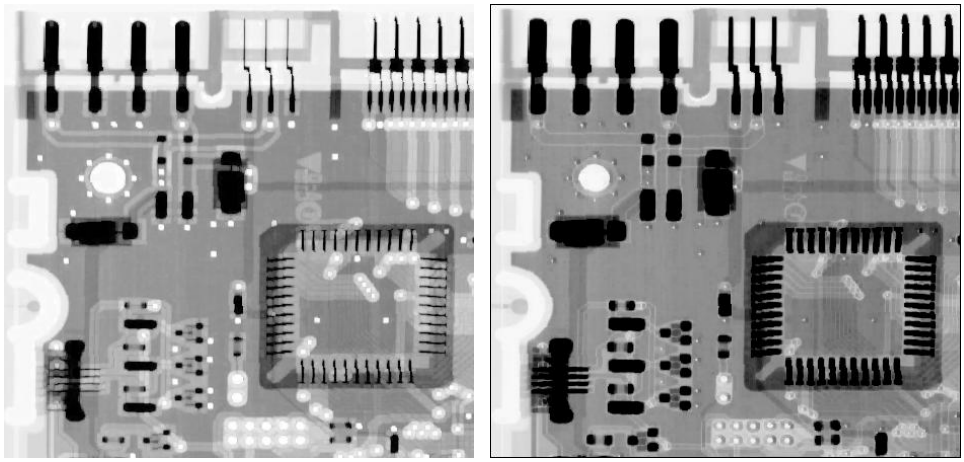


(a) Pepper and salt noise (b) pass (a) with a 3x3 median filter



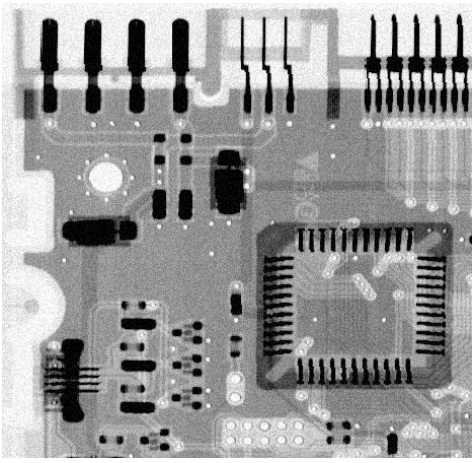
(c) pass (b) with same median filter (d) pass (c) with same median filter

Figure 5.10 (P328)

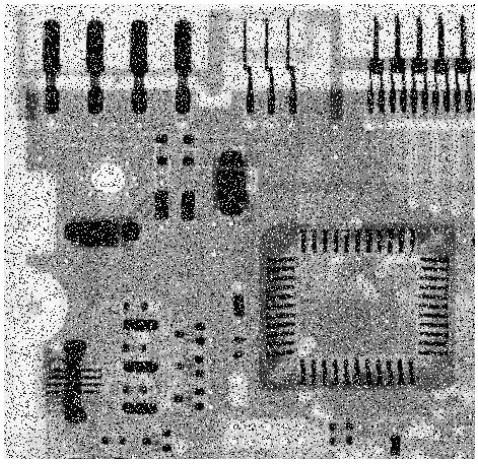


(a) 3x3 max filter to Figure5.8(a) (b) 3x3 min filter to Figure5.8(b)

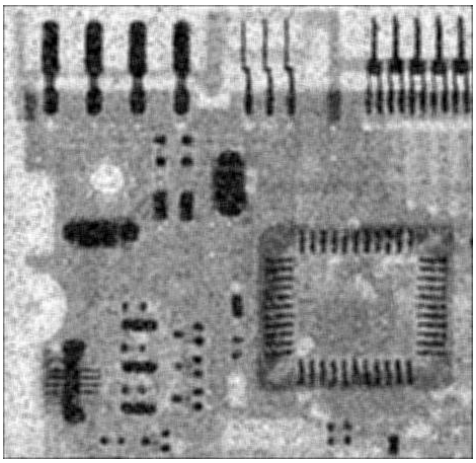
Figure 5.11 (P328)



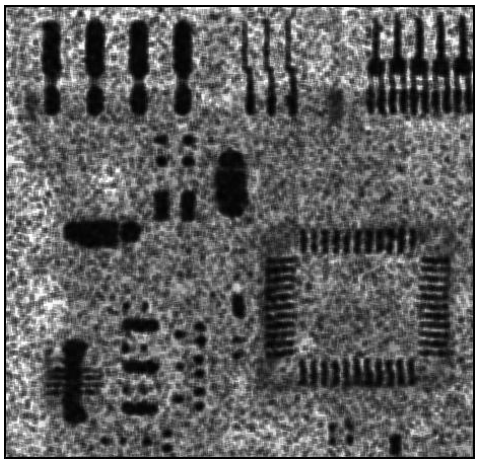
(a) Uniform noise



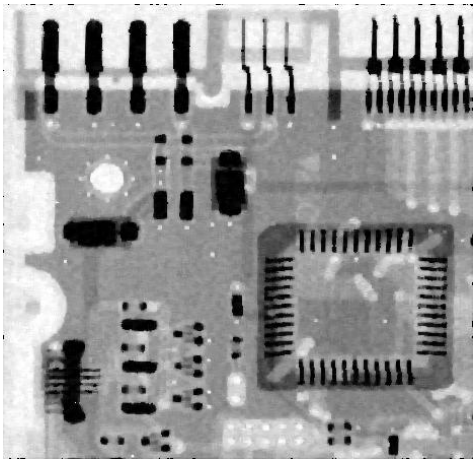
(b) add pepper and salt noise to (a)



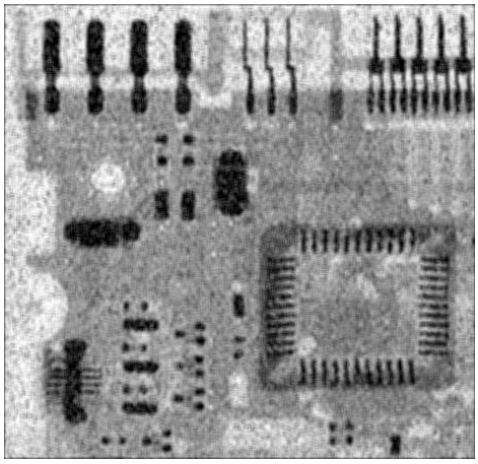
(c) Arithmetic filter with 5x5 mask



(d) Geometric filter with 5x5 mask



(e) Median filter with 5x5 mask



(f) Alpha-trimmed filter with 5x5 mask

Figure 5.12 (P329)

Conclusion

We can see from the results that:

1) The mean filters are nice ways to reduce the noises of which the pattern has random distribution, such as uniform noise, Gaussian noise and Rayleigh noise, while, for pepper and salt noise, it is much better to use the min/max filters or the contraharmonic filter.
2) For exponential noise and Gamma noise, the min filter and contraharmonic are nice ways for these two noise will make the image turns to be a little bit of white.
3) We can use median filter many times on pepper and salt noise to get a better result compared with use it only once.

Project 5 - Image restoration

Problem

Image restoration (the test image book_cover.jpg can be found at ftp://ftp.cs.sjtu.edu.cn:990/lu-ht/DIP/images)
Suppose a blurring degradation function as

$$H(u,v)=\frac{T}{\pi(ua+vb)}\sin[\pi(ua+vb)]e^{-j\pi(ua+vb)} \qquad (1)$$

(a) Implement a blurring filter using Eq. (1).
(b) Blur the test image book_cover.jpg using parameters a=b=0.1 and T = 1.
(c) Add Gaussian noise of 0 mean and variance of 650 to the blurred image.
(d) Restore the blurred image and the blurred noisy image using the inverse filter, Wiener filter, respectively.
(e) Add Gaussian noise of 0 and different variances to the blurred image and repeat
(d) Investigate the performance of the Wiener filter .

Environment

Matlab R2016a
Atom 1.22 (with language-matlab package)

Principle

- **Blurring degradation function**

$$H(u,v)=\frac{T}{\pi(ua+vb)}\sin[\pi(ua+vb)]e^{-j\pi(ua+vb)}$$

- **Inverse filter**

  The simplest approach to restoration is direct inverse filtering, where we compute an estimate, $\hat{F}(u,v)$, of the transform of the original image simply by dividing the transform of the degraded image, G(u, v), by the degradation function:

$$\hat{F}(u,v)=\frac{G(u,v)}{H(u,v)}$$

  But when the H(u, v) turns to be very small, the result will be too large to get the output picture show normally, so we need to set a minimum for H(u, v) to avoid this condition.

- **Wiener deconvolution filter (Minimum Mean Square Error Filter)**

  To solve the problem caused by small H(u, v) and to deal with blurred image added noise, Wiener deconvolution filter is used. The basic idea is to assume that the noise and the image are uncorrelated; that one or the other has zero mean; and that the intensity levels in the estimate are a linear function of the levels in the degraded image. Based on these conditions, the minimum of the error function is given in the frequency domain by the expression

$$\hat{F}(u,v) = \left[\frac{H^*(u,v)S_f(u,v)}{S_f(u,v)|H(u,v)|^2 + S_\eta(u,v)}\right]G(u,v)$$

$$= \left[\frac{H^*(u,v)}{|H(u,v)|^2 + S_\eta(u,v)/S_f(u,v)}\right]G(u,v)$$

$$= \left[\frac{1}{H(u,v)}\frac{|H(u,v)|^2}{|H(u,v)|^2 + S_\eta(u,v)/S_f(u,v)}\right]G(u,v)$$

  When we are dealing with spectrally white noise, the spectrum | N(u, v) | 2 is a constant, which simplifies things considerably. However, the power spectrum of the undegraded image seldom is known. An approach used frequently when these quantities are not known or cannot be estimated is to approximate the formula by the expression

$$\hat{F}(u,v) = \left[\frac{1}{H(u,v)}\frac{|H(u,v)|^2}{|H(u,v)|^2 + K}\right]G(u,v)$$

Implementation

1. Clear the former environments at the beginning:

```
% Clear the environment
close all;
clc
clear;
```

2. Input the image and get the information about the image (There the location of the input image is directly assigned for convenience and it also can be assigned by the function parameter after comment this line) :

```
% Set the default location of input image file
img_location = '../../images/book_cover.jpg';
input_img = double(imread(img_location));

% Get the relative information of the image
info = imfinfo(img_location);
M = info.Height;
N = info.Width;
depth = info.BitDepth;
```

3. First we get the dft of the input image and blur the frequency image using blurring() function. Then convert the blurred image back to the normal image:

```
% Blurring the input image
dft_input_img = fftshift(fft2(input_img));
blur_img = zeros(M, N);
for i = 1:M
    for j = 1:N
      % Use the H(u,v) to get the blurring with motion effects
        H = blurring(i - M/2, j - N/2, 0.1, 0.1, 1);
        blur_img(i,j) = dft_input_img(i,j) * H;
    end
end
blur_img = ifft2(ifftshift(blur_img));
% Show the pictures and output them
figures(...
{input_img, blur_img},...
'Bluring degradation with a=b=0.1 & T=1',...
{'Origin input image', 'Blurred image'},...
true)
```

4. For the blurring function, it just need calculate the output according to the formula and avoid infinity and NaN appears in the output:

```
% Blurring degradation function
function [ output ] = blurring(u, v, a, b, T)
% blurring degradation function.

t = u * a + v * b;
output = (T / (pi * t)) * sin(pi * t) * exp(-1i * pi * t);

% when the output gets infinity or NaN, we just do nothing
output(isinf(output)) = 1;
output(isnan(output)) = 1;

end
```

5. Then we need add the Gaussian noise to the blurred image, there I choose to add the Gaussian noise which has the variance 650, 65, 6.5 respectively to get the compare. The generation of Gaussian noise is same with the Project4 so I just use the noise_generator function in the Project4 to get the noise:

```
% For different variance gaussian noises
for variance = [650, 65, 6.5]
  % Add gaussian noises to the blurred image
  gaussian_img = noise_generator(blur_img, 'gaussian', depth, 0, variance);

```

6. To restore the image by inverse filter, first get the dft of the image and calculate the H according to the formula, if we just directly divide H it may get very large in some condition when H is quite small. So we set a minimum threshold to avoid the H getting too small. To get the pictures similar with the textbooks we show the direct inverse result and the threshold inverse result both:

```
% Restore the image using the inverse filter.
dft_blur_img = fftshift(fft2(blur_img));
direct_inverse_img = zeros(M, N);
inverse_img = zeros(M, N);
for i = 1:M
    for j = 1:N
        H = blurring(i - M/2, j - N/2, 0.1, 0.1, 1);
        direct_inverse_img(i,j) = (dft_blur_img(i,j) / H);
        % For there with be conditions H = 0
        % we need to set a threshold such as 0.001
        if abs(H) < 0.001
            H = 0.001;
        end
        inverse_img(i,j) = (dft_blur_img(i,j) / H);
    end
end
direct_inverse_img = uint8(real(ifft2(ifftshift(direct_inverse_img))));
inverse_img = uint8(real(ifft2(ifftshift(inverse_img))));

```

7. It's similar for the process of restoring by wiener filter:

```
% Restore the image using wiener deconvolution filters.
dft_gaussian_img = fftshift(fft2(gaussian_img));
wiener_img = zeros(M, N);
for i = 1:M
    for j = 1:N
        H = blurring(i - M/2, j - N/2, 0.1, 0.1, 1);
        wiener_img(i,j) = (dft_gaussian_img (i,j) / H) * (abs(H)^2 / (abs(H)^2 + 0.01));
    end
end
wiener_img = uint8(real(ifft2(ifftshift(wiener_img))));

```

8. At last we show them all:

```
% Show the pictures and output them
figures(...
{gaussian_img, direct_inverse_img, inverse_img, wiener_img},...
['Restore the image with gaussian noise(var=', int2str(variance), ')'],...
{['Gaussian noise image(var=', int2str(variance), ')'], ['Direct inverse restore (var=',...
 int2str(variance), ')'], ['Inverse restore (var=', int2str(variance), ')'], ['Wiener restore...
  K = 0.01(var=', int2str(variance), ')']},...
true)
end

```

9. To investigate the performance of the wiener filter, I choose K = 0.1, 0.01, 0.001 and 0.0001 to restore a blurred image adding Gaussian noise with variance 65. By comparing the result of these result we can investigate the performance of the wiener filter:

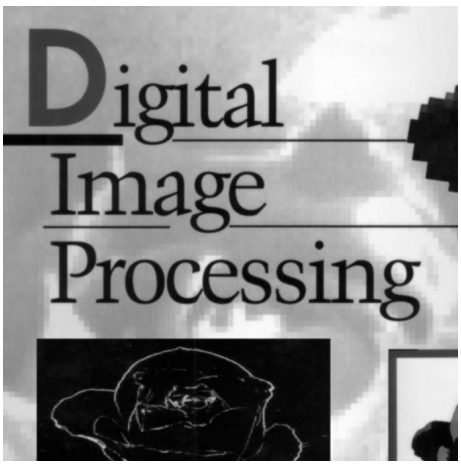
```
81
82  % Investigate the performance of wiener filter
83  % Add gaussian noises to the blurred image
84  gaussian_img = noise_generator(blur_img, 'gaussian', depth, 0, 65);
85
86  for K = [0.1, 0.01, 0.001, 0.0001]
87    % Restore the image using wiener deconvolution filters.
88    dft_gaussian_img = fftshift(fft2(gaussian_img));
89    wiener_img = zeros(M, N);
90    for i = 1:M
91        for j = 1:N
92            H = blurring(i - M/2, j - N/2, 0.1, 0.1, 1);
93            wiener_img(i,j) = (dft_gaussian_img (i,j) / H) * (abs(H)^2 / (abs(H)^2 + K));
94        end
95    end
96    wiener_img = uint8(real(ifft2(ifftshift(wiener_img))));
97
98    % Show the pictures and output them
99    figures({wiener_img}, ['Restore the image using wiener filter(K=', num2str(K), ')'],...
100   {['Wiener filter(K=', num2str(K), ')']}, true)
101 end
```

10. For all the pictures and figures, I use the figure function similar with the former project to show:

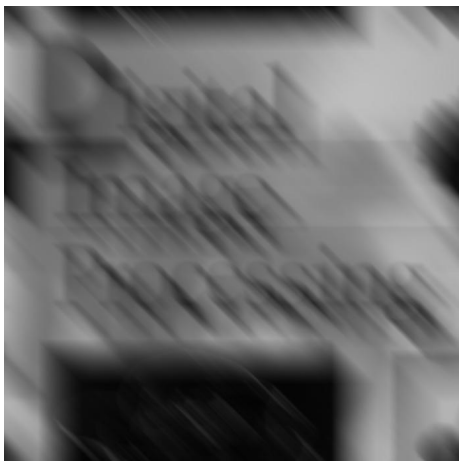
```
5   figure('NumberTitle','off','Name', figure_title)
6   column = 2;
7   [temp, num] = size(input_imgs);
8   row = ceil(num / column);
9
10  for i = 1:row
11    for j = 1:column
12      if((i - 1)* column + j) <= num
13        subplot(row, column, (i - 1)* column + j)
14        imshow(uint8(input_imgs{(i - 1) * column + j}),[]);
15        title(char(image_titles{(i - 1) * column + j}))
16        if output_file
17          imwrite(uint8(input_imgs{(i - 1) * column + j}), [char(image_titles{(i - 1) * column + j}), '.jpg']);
18        end
19      end
20    end
21  end
```

Result

- **Blur the test image:**



(a) Original input image



(b) Motion blurring (a)

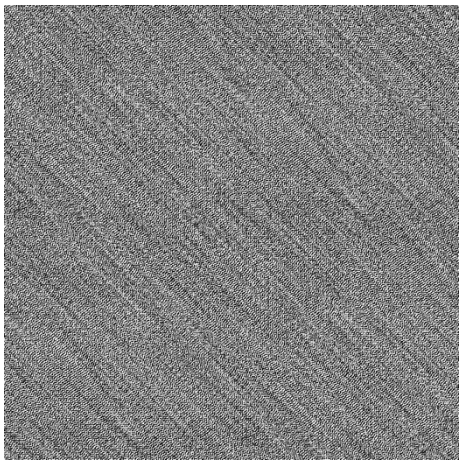
- **Add Gaussian noise with variance 650**

(a) Blurred input image

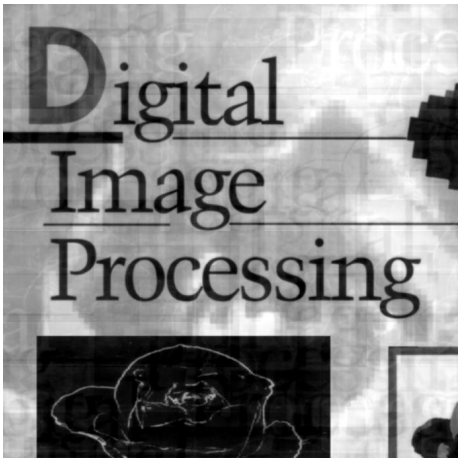
(b) Add Gaussian noise

- **Restore the variance 650 Gaussian noise image using different filters**

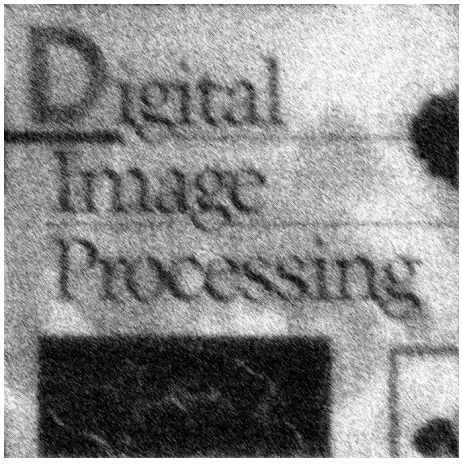
(a) Blurred image with noise



(b) Restore with direct inverse filter

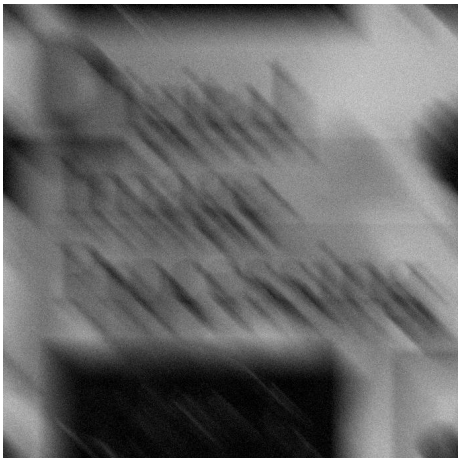


(c) Restore with threshold inverse filter

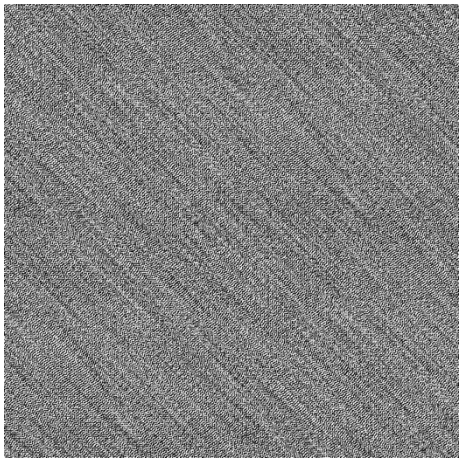


(d) Restore with wiener filter

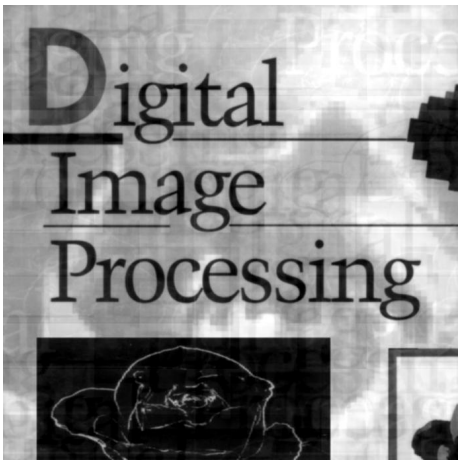
- **Restore the variance 65 Gaussian noise image using different filters**



(a) Blurred image with noise



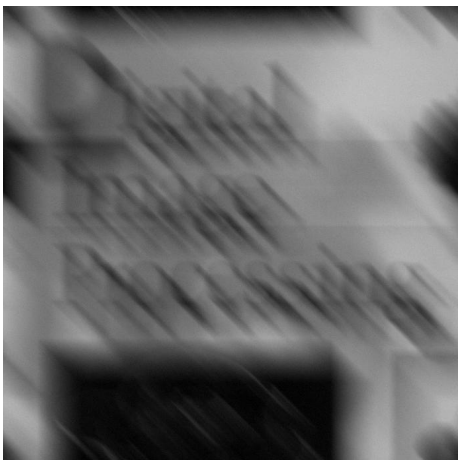
(b) Restore with direct inverse filter



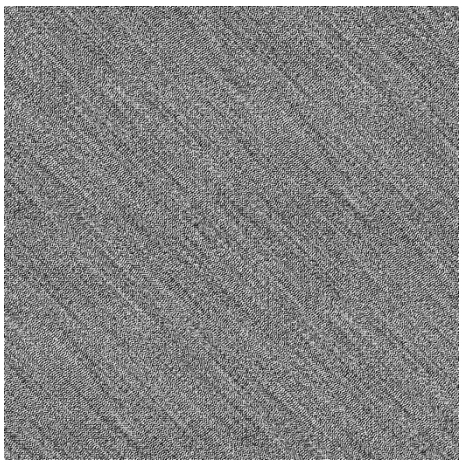
(c) Restore with threshold inverse filter

(d) Restore with wiener filter

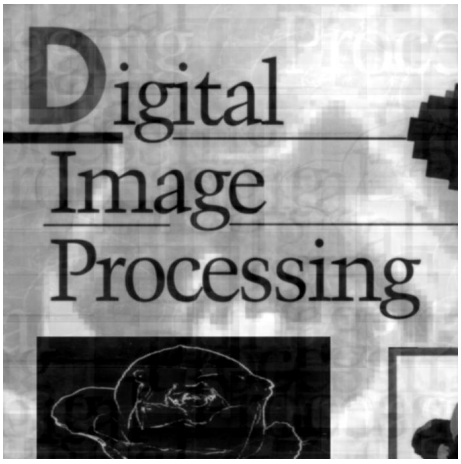
- **Restore the variance 6.5 Gaussian noise image using different filters**



(a) Blurred image with noise



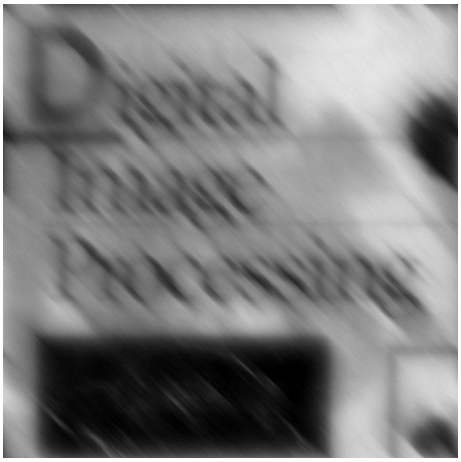
(b) Restore with direct inverse filter



(c) Restore with threshold inverse filter

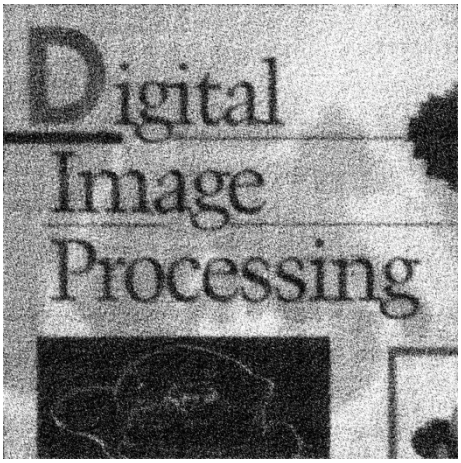
(d) Restore with wiener filter

- **Restore the variance 65 Gaussian noise image using wiener filter with different K value**

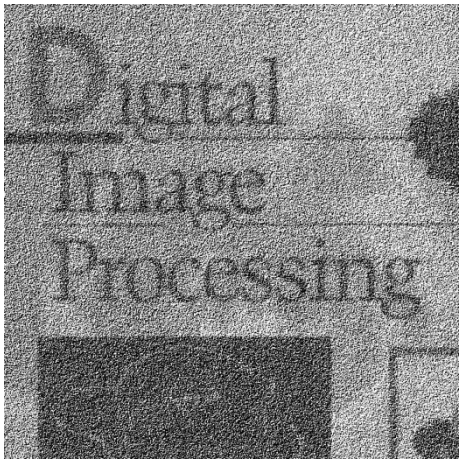


(a) K = 0.1

(b) K = 0.01



(c) K = 0.001



(d) K = 0.0001

Conclusion

We can see from the results that:

1) The direct inverse filter will totally damage the process of restoring because of the small H, after setting a threshold, this problem can be avoided but the restore effect is not very nice.
2) The wiener filter can get a better result relatively. But if the value of K gets too small there still will be a lot of noise, while if the value of K gets too big the image will still be blurred. Therefore the K needs to be adjusted according to actual conditions.

Project 6 - Geometric transform

Problem

Develop a geometric transform program that will rotate, translate, and scale an image by specified amounts, using the nearest neighbor and bilinear interpolation methods, respectively.

Environment

Matlab R2016a
Atom 1.22 (with language-matlab package)

Principle

- **Spatial transformation**

  The spatial transformation defines the spatial relationship between points in the input image and points in the output image.

  $$g(x,y) = f(x',y') = f[a(x,y),b(x,y)]$$

  - **Translate**

    $$a(x,y) = x + x_0 \qquad b(x,y) = y + y_0$$

    Can be expressed in homogeneous coordinates as

    $$\begin{bmatrix} a(x,y) \\ b(x,y) \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

  - **Scale**

    $$a(x,y) = x/c \qquad b(x,y) = y/d$$

    Homogeneous form

    $$\begin{bmatrix} a(x,y) \\ b(x,y) \\ 1 \end{bmatrix} = \begin{bmatrix} 1/c & 0 & 0 \\ 0 & 1/d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

    When c = -1, d = 1 : reflection about the y-axis

    $$a(x,y) = -x \qquad b(x,y) = y$$

  - **Rotate**

$$a(x,y) = x\cos(\theta) - y\sin(\theta)$$
$$b(x,y) = x\sin(\theta) + y\cos(\theta)$$
$$\begin{bmatrix} a(x,y) \\ b(x,y) \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- **Gray-level interpolation methods**

  Gray-level interpolation is necessary because the gray-level values are defined only at integral values of x and y, however, spatial transformation may make the output image g(x, y) taking values at fractional positions of a(x, y) and b(x, y)

  - **Nearest neighbor interpolation**
    The gray level of the output pixel is taken to be that of the input pixel nearest the location to which the output pixel maps.

  - **Bilinear interpolation**
    For a function f(x, y) of two variables, given its values at four vertices of the unit square f(0,0), f(0,1), f(1,0) and f(1,1), we desire to determine the value of f(x, y) at an arbitrary point inside the square. This can be done by a hyperbolic paraboloid defined by the bilinear equation

$$f(x,y) = ax + by + cxy + d$$

$$f(x,y) = (f(1,0) - f(0,0))x + [f(0,1) - f(0,0)]y$$
$$+ [f(1,1) + f(0,0) - f(0,1) - f(1,0)]xy + f(0,0)$$

Implementation

1. Clear the former environments at the beginning:

```
4  % Clear the environment
5  close all;
6  clc
7  clear;
```

2. Input the image and get the information about the image (There the location of the input image is directly assigned for convenience and it also can be assigned by the function parameter after comment this line) :

```
% Set the default location of input image file
img_location = '../../images/ray_trace_bottle.tif';
input_img = double(imread(img_location));

% Get the relative information of the i
info = imfinfo(img_location);
M = info.Height;
N = info.Width;
```

3. Define the allowed patterns of spatial transformations and interpolation methods, also assign the parameters will be tested:

```
% Define the spatial transformation and interpolation patterns
transform_patterns = { 'translate', 'rotate', 'scale' };
interpolate_patterns = {'nearest', 'bilinear'};

% Get all the parameters accodring to the order of the patterns
all_parameters = {...
{[200,300],[-200,300]},...
{[90,0],[30,0],[45,0]},...
{[1,1.2],[2,1],[3,3]}...
};
```

4. For each spatial transformation and interpolation methods, we test with the a set of parameters using function spatial_transform:

```
% All three spatial transformation patterns and two interpolation patterns
% Get the output image and resize the output image to avoid cropping
for i = 1:3
  transform_pattern = transform_patterns{i};
  parameters = all_parameters{i};
  for parameter = parameters
      nearest_img = spatial_transform(input_img, transform_pattern, 'nearest', false, parameter{1});
      resize_nearest_img = spatial_transform(input_img, transform_pattern, 'nearest', true, parameter{1});
      bilinear_img = spatial_transform(input_img, transform_pattern, 'bilinear', false, parameter{1});
      resize_bilinear_img = spatial_transform(input_img, transform_pattern, 'bilinear', true, parameter{1});
```

5. The spatial transform function
    a. we first get the transformation matrix T according to the pattern. Because for some transformtions, the image size will change so it needs to resize the output image size according to the resize flag.

```
switch lower(char(transform_pattern))
  case transform_patterns{1}
    % Translate
    T = [1, 0, 0;
        0, 1, 0;
        parameters(1), parameters(2), 1];

    % Calculate inverse matrix of T.
    T = inv(T);

    % Resize the image to avoid the crop
    if resize
        H = M + abs(parameters(1));
        W = N + abs(parameters(2));
    end

  case transform_patterns{2}
    % Rotate
```

```
    parameter = (parameters(1) / 180) * pi;
    % For positive x-axis points down in matlab coordinates, and rotating
    % from x to y means counter clockwise, so there needs to inverse T.
    T = [cos(parameter), sin(parameter), 0;
        -sin(parameter), cos(parameter), 0;
        0, 0, 1];

    % Resize the image to avoid the crop
    if resize
        H = ceil(sqrt(M^2 + N^2) + 2);
        W = H;
    end
    parameter(1) = 0;
    parameter(2) = 0;

  case transform_patterns{3}
    % Scale

    T = [parameters(1), 0, 0;
        0, parameters(2), 0;
        0, 0, 1];

    T = inv(T);

    % Resize the image to avoid the crop
    if resize
        H = ceil(M * parameters(1));
        W = ceil(N * parameters(2));
    end

    parameters(1) = 0;
    parameters(2) = 0;
```

b. Then we get the output image by multiply the transform matrix T and use function interpolation() to interpolate:

```
% Get the output image by the matrix T
output_img = zeros(H, W);
for i = -H/2 + 1:H/2
  for j = -W/2 + 1:W/2
    p = [i j 1] * T;
    % Add the offset
    u = p(1) + (M + parameters(1))/2;
    v = p(2) + (N + parameters(2))/2;
    if u >= 1 && u <= M && v >= 1 && v <= N
        output_img(i + H/2, j + W/2) = interpolation(input_img, interpolate_pattern, u, v);
    end
  end
end

% Transfrom the data type to be uint8
output_img = uint8(output_img);
```

c. For rotate transformation, it will generate invalid black border, so we remove it at last:

```
% Cut off invaild black border.
if (strcmp(char(transform_pattern), 'rotate') && resize)
    fy = find(max(output_img));
    fx = find(max(output_img, [], 2));
    output_img = output_img(min(fx):max(fx), min(fy):max(fy));
end
```

6. The interpolation function

We can just calculate according to the formula, there we directly use (round(x), round(y)) to find the nearest point:

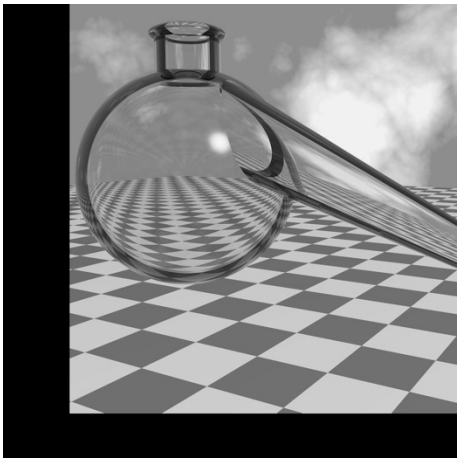
```
% Get the interpolation result accodring to the pattern
switch lower(char(pattern))
  case patterns{1}
    % nearest
    % There use the round(x), round(y) to find the nearest point
    output = input_img(round(x), round(y));
  case patterns{2}
    % bilinear
    xm = floor(x);
    ym = floor(y);
    if x == xm && y == ym
        output = input_img(x, y);
    elseif x == xm && y ~= ym
        output = (y - ym) * input_img(x, ym + 1) + (ym + 1 - y) * input_img(x, ym);
    elseif x ~= xm && y == ym
        output = (x - xm) * input_img(xm + 1, y) + (xm + 1 - x) * input_img(xm, y);
    else
        t1 = (x - xm) * input_img(xm + 1, ym) + (xm + 1 - x) * input_img(xm, ym);
        t2 = (x - xm) * input_img(xm + 1, ym + 1) + (xm + 1 - x) * input_img(xm, ym + 1);
        output = (y - ym) * t2 + (ym + 1 - y) * t1;
    end
  otherwise
    error(['The', ' ', pattern, ' ', 'interpolation pattern can not be found.'])
end
```

7. For the output is relatively regular so just direct show the pictures and output the file:

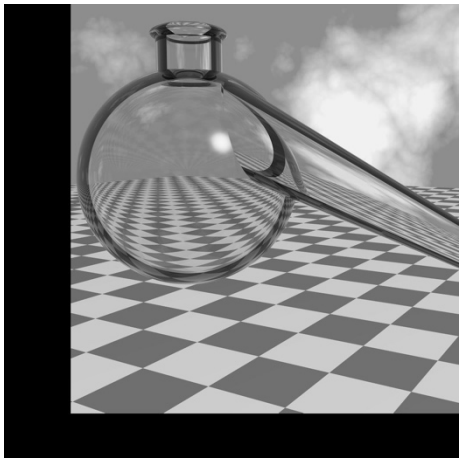
```
    %%%%%%%%%%%%%%%%%%%%%%%%%%% Show pictures %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    figure('NumberTitle', 'off', 'Name', ['The ', ' ', char(transform_pattern), ' transform'])
    if strcmp(char(transform_pattern), 'rotate')
      img_title = [int2str(parameter{1}(1)), ' degree'];
    else
      img_title = [int2str(parameter{1}(1)), ',', int2str(parameter{1}(2))];
    end
    subplot(2,2,1);
    imshow(nearest_img);
    title(['nearest - ', img_title]);
    subplot(2,2,2);
    imshow(resize_nearest_img);
    title('After resize');
    subplot(2,2,3);
    imshow(bilinear_img);
    title(['bilinear - ', img_title]);
    subplot(2,2,4);
    imshow(resize_bilinear_img);
    title('After resize');

    %%%%%%%%%%%%%%%%%%%%%%%%%%% Output the file %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    imwrite(nearest_img, ['nearest - ', img_title, '.jpg']);
    imwrite(resize_nearest_img, ['resize nearest - ', img_title, '.jpg']);
    imwrite(bilinear_img, ['bilinear - ', img_title, '.jpg']);
    imwrite(resize_bilinear_img, ['resize bilinear - ', img_title, '.jpg']);
end
```

Result

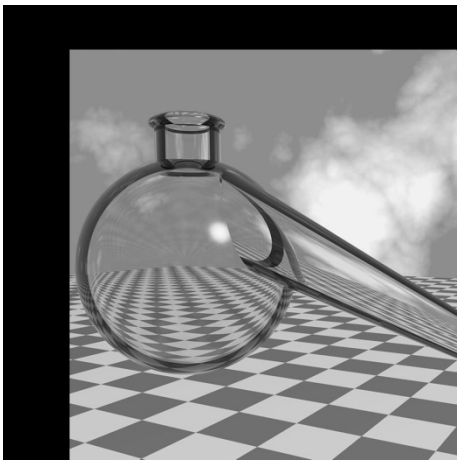
- **Translate**



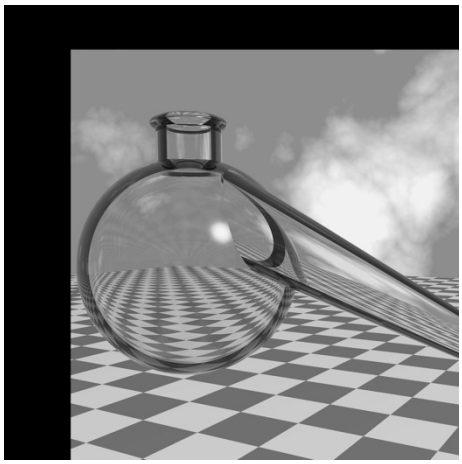
(a) Translate (300,200) - nearest



(b) Translate (300,200) - bilinear

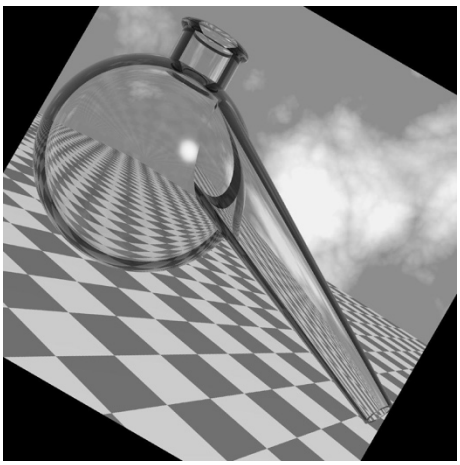


(c) Translate (300,-200) - nearest

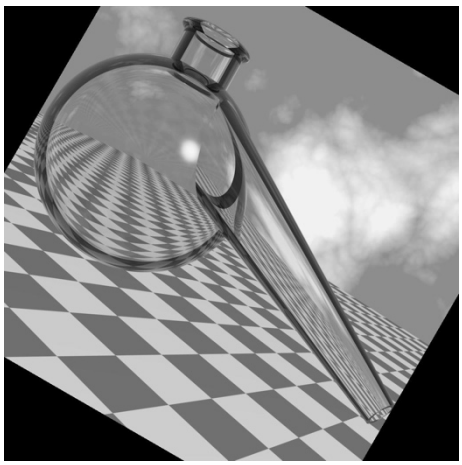


(d) Translate (300,-200) - bilinear

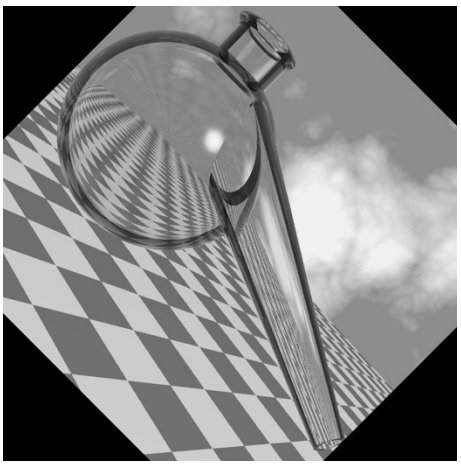
- **Rotate**
  - **Cropped**



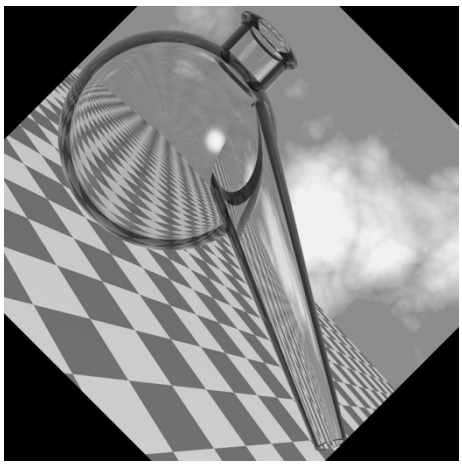
(a) Rotate 30 degree - nearest



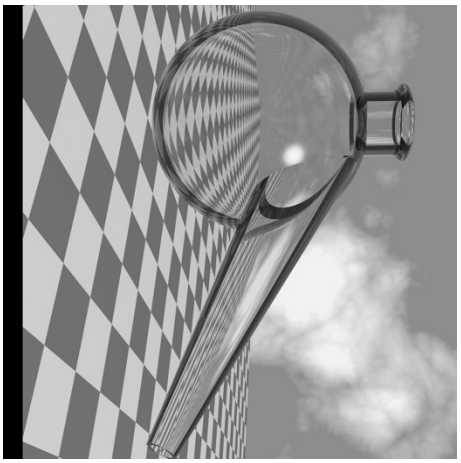
(b) Rotate 30 degree - bilinear



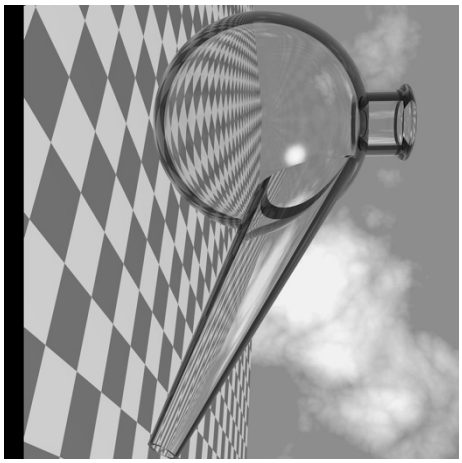
(c) Rotate 60 degree - nearest



(d) Rotate 60 degree - bilinear

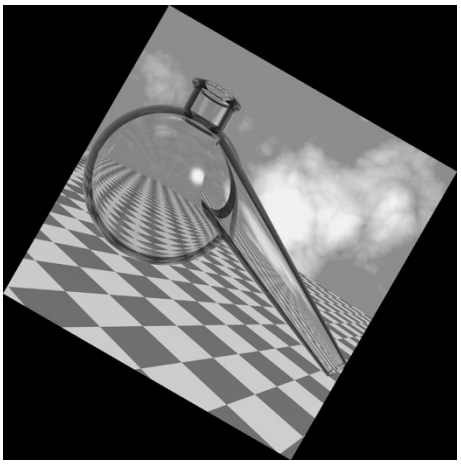


(e) Rotate 90 degree - nearest

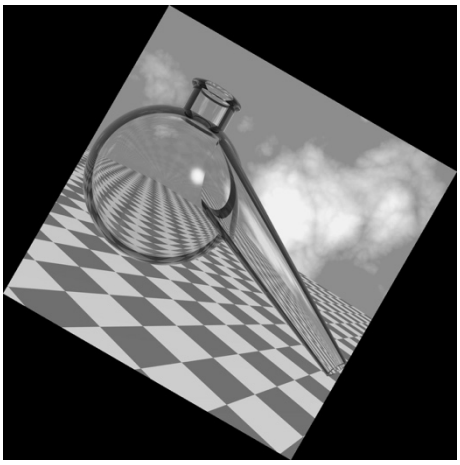


(f) Rotate 90 degree - bilinear

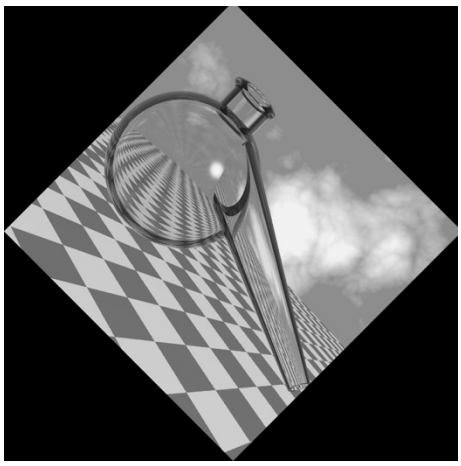
- **Resized**



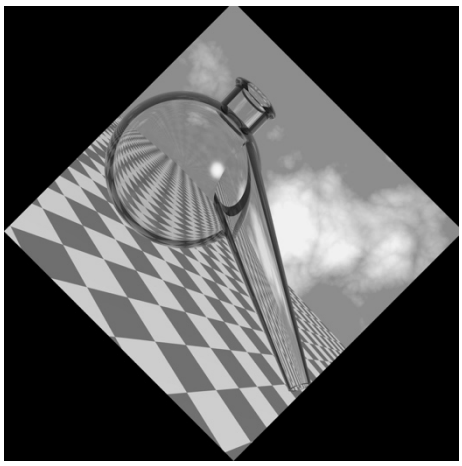
(a) Rotate 30 degree - nearest



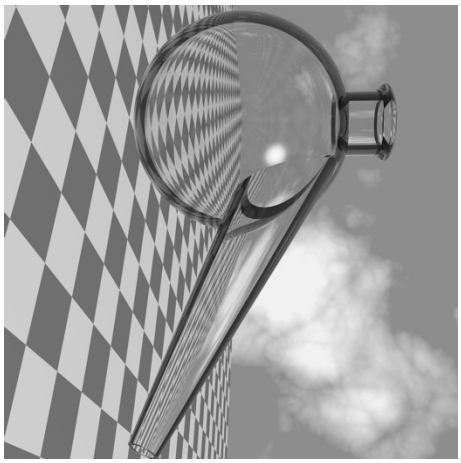
(b) Rotate 30 degree - bilinear



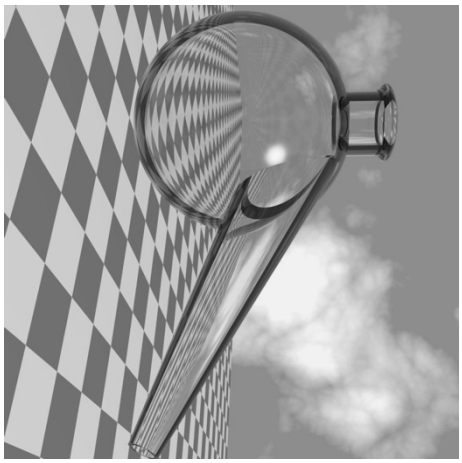
(c) Rotate 60 degree - nearest



(d) Rotate 60 degree - bilinear

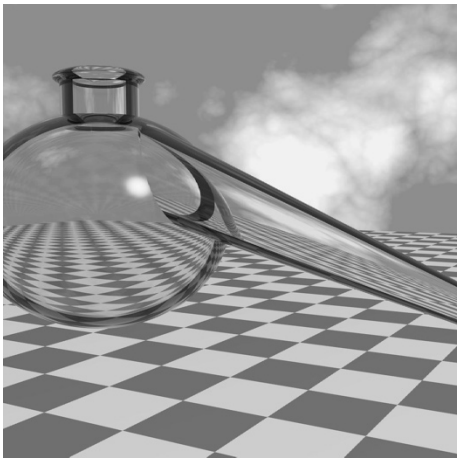


(e) Rotate 90 degree - nearest

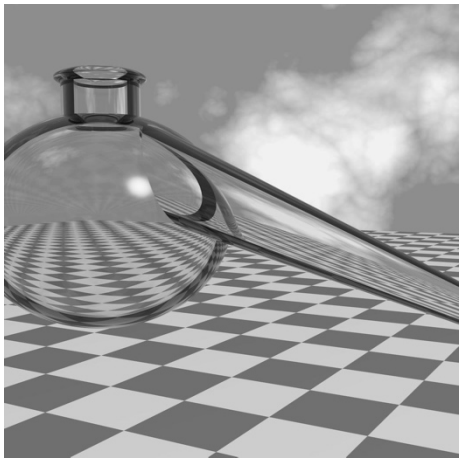


(f) Rotate 90 degree - bilinear

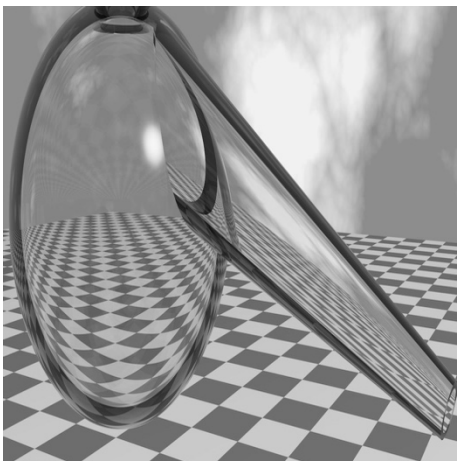
- **Scale**
  - **Cropped**



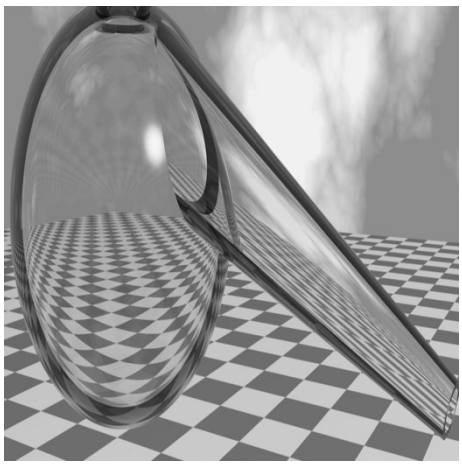
(a) Scale (1,1.2) - nearest



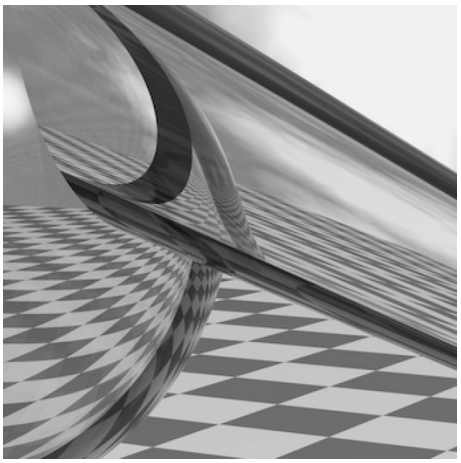
(b) Scale (1,1.2) - bilinear



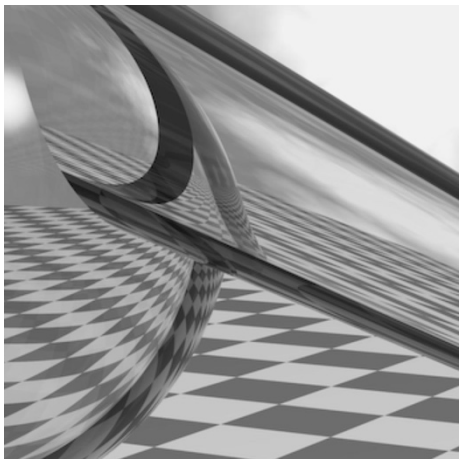
(c) Scale (2,1) - nearest



(d) Scale (2,1) - bilinear

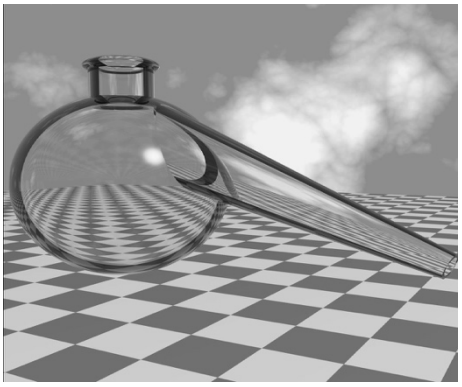


(e) Scale (3,3) - nearest

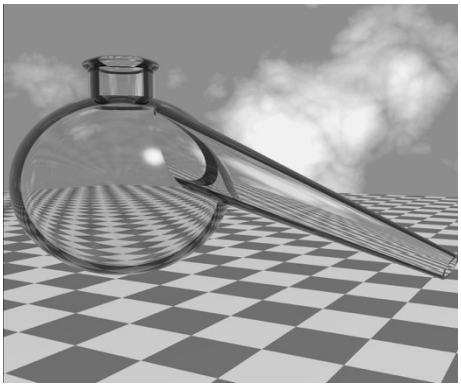


(f) Scale (3,3) - bilinear

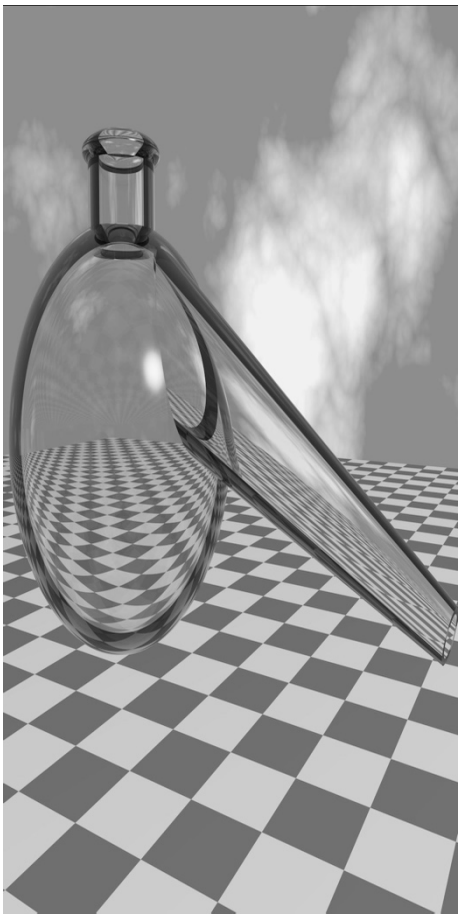
- **Resized**



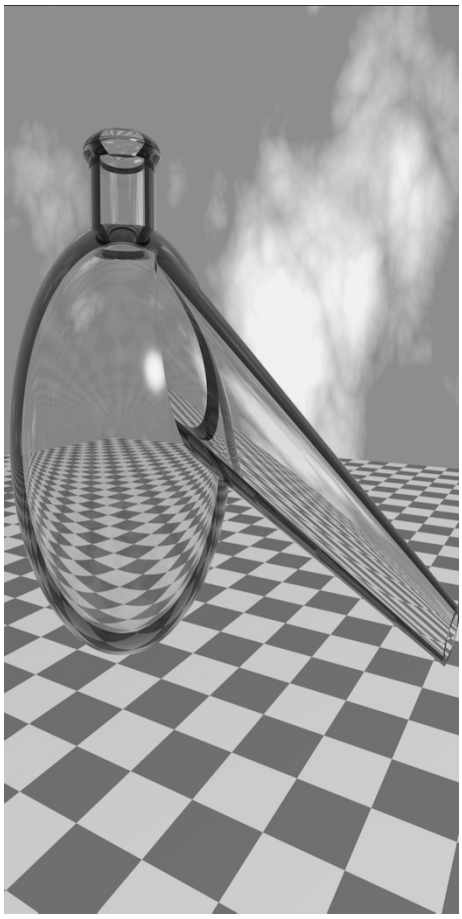
(a) Scale (1,1.2) - nearest



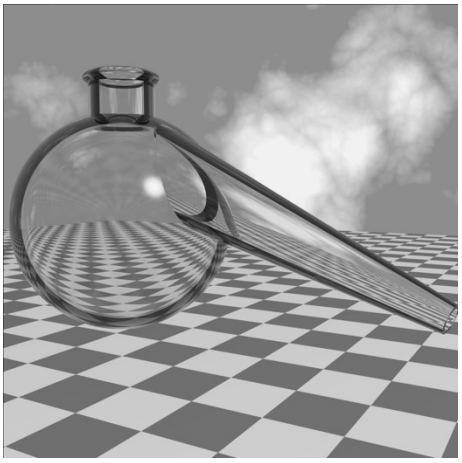
(b) Scale (1,1.2) - bilinear



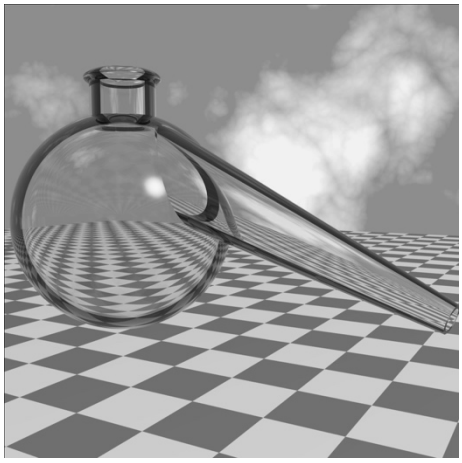
(c) Scale (2,1) - nearest



(d) Scale (2,1) - bilinear



(e) Scale (3,3) - nearest



(f) Scale (3,3) - bilinear

Project 8 - Morphological Processing

Problem

Implement the "Opening by reconstruction", "Filling holes" and "Border clearing" operations on pages 681-683 (pp.659-661 for the electronic textbook), and reproduce the results in Figure 9.29, Figures 9.31, and Figure 9.32. (The images fig9.29(a) and fig9.31(a) can be found at ftp://ftp.cs.sjtu.edu.cn:990/lu- ht/DIP/images)

Environment

Matlab R2016a
Atom 1.22 (with language-matlab package)

Principle

- **Erosion and Dilation**
  - **Erosion**
    With A and B as sets in Z^2,, the erosion of A by B, denoted A $\ominus$ B, is defined as:

    $$A \ominus B = \{z|(B)_z \subseteq A\}$$

    Because the statement that B has to be contained in A is equivalent to B not sharing any common elements with the background, we can express erosion in the following equivalent form:

    $$A \ominus B = \{z|(B)_z \cap A^c = \varnothing\}$$

  - **Dilation**
    With A and B as sets in Z^2, the dilation of A by B, denoted A $\oplus$ B, is defined as:

    $$A \oplus B = \left\{z|(\hat{B})_z \cap A \neq \varnothing\right\}$$

    This equation is based on reflecting B about its origin, and shifting this reflection by z. The dilation of A by B then is the set of all displacements, z, such that BN and A overlap by at least one element. Based on this interpretation, Equation above can be written equivalently as:

    $$A \oplus B = \left\{z|[(\hat{B})_z \cap A] \subseteq A\right\}$$

- **Duality**
  Erosion and dilation are duals of each other with respect to set complementation and reflection. That is:

$$(A \ominus B)^c = A^c \oplus \hat{B}$$

and

$$(A \oplus B)^c = A^c \ominus \hat{B}$$

- **Opening by reconstruction**

The opening of set A by structuring element B, denoted A ∘ B, is defined as

$$A \circ B = (A \ominus B) \oplus B$$

The opening by reconstruction of size n of an image F is defined as the reconstruction by dilation of F from the erosion of size n of F; that is,

$$O_R^{(n)}(F) = R_F^D\left[(F \ominus nB)\right]$$

- **Filling holes**

Let I(x, y) denote a binary image and suppose that we form a marker image F that is 0 everywhere, except at the image border, where it is set to 1 - I; that is,

$$F(x, y) = \begin{cases} 1 - I(x, y) & \text{if } (x, y) \text{ is on the border of } I \\ 0 & \text{otherwise} \end{cases}$$

Then

$$H = \left[R_{I^c}^D(F)\right]^c$$

is a binary image equal to I with all holes filled.

- **Border clearing**

In this application, we use the original image as the mask and the following marker image:

$$F(x, y) = \begin{cases} I(x, y) & \text{if } (x, y) \text{ is on the border of } I \\ 0 & \text{otherwise} \end{cases}$$

The border-clearing algorithm first computes the morphological reconstruction R_I^D(F) (which simply extracts the objects touching the border) and then computes the difference：

$$X = I - R_I^D(F)$$

to obtain an image, X, with no objects touching the border.

Implementation

1. The opening by reconstruction part
    a. First clear the former environments and get the input image :

```
%%%%%%%%%%%%%%%%%%%%%%%%% Opening by reconstruction %%%%%%%%%%%%%%%%%%%%%%%%%
% Clear the environment
clc
clear;

% Set the default location of input image file
img_location = '../../images/Fig0929(a)(text_image).tif';
input_img = double(imread(img_location));

% Get the relative information of the image
info = imfinfo(img_location);
M = info.Height;
N = info.Width;
```

    b. Use function operations() to get the erosion input image and dilation image:

```
% Get the erosion image and dilation image
erosion_img = operations(input_img, 'erosion', 51,1);
dilation_img = operations(erosion_img, 'dilation', 51,1);

```

    c. To get the reconstruction image, just do dilation operations on dilation input image until it doesn't change any more, then we can get the final result :

```
% First assign the dilation image to reconstruction image
% Then use dilation operation on reconstruction image
% Until the reconstruction image doesn't change any more
reconstruction_img = dilation_img;
while(true)
    new_img = operations(reconstruction_img, 'dilation', 3, 3);
    new_img = new_img & input_img;
    if all(all(new_img == reconstruction_img))
        break;
    end
    reconstruction_img = new_img;
end

% Show the pictures and output them
figures(...
{input_img, erosion_img, dilation_img, reconstruction_img},...
'FIG9.29 Opening by reconstruction',...
{'FIG9.29(a) Origin input image', 'FIG9.29(b) Erosion', 'FIG9.29(c) Opening', ...
'FIG9.29(d) Reconstruction'},...
true)
```

2. The operations function
    a. First we get the height and width of the input image and zero padding it:

```
[M,N] = size(input_img);
H = (m - 1)/2;
W = (n - 1)/2;

% Zero padding
im_padded = zeros(M + m - 1, N + n - 1);
im_padded(H + 1:H + M,W + 1:W + N) = input_img;

```

    b. Then do the erosion or the dilation operation according to the pattern to

get the output image:

```
output_img = zeros(M, N) == 1;
switch lower(operation)
    case 'erosion'
      for i = H + 1:H + M
        for j = W + 1:W + N
          % Considering the efficiency, use the square mask with 1s
          if all(all(im_padded(i - H:i + H,j - W:j + W)))
              output_img(i - H, j - W) = 1;
          end
        end
      end
    case 'dilation'
      for i = H + 1:H + M
        for j = W + 1:W + N
          if any(any(im_padded(i - H:i + H, j - W:j + W)))
              output_img(i - H, j - W) = 1;
          end
        end
      end
    otherwise
      error(['The', ' ', operation, ' ', 'operation pattern can not be found.'])
end
end
```

3. The filling holes part
    a. First clear the former environments and get the input image :

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Filling holes %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clear the environment
clc
clear;

% Set the default location of input image file
img_location = '../../images/Fig0929(a)(text_image).tif';
input_img = double(imread(img_location));

% Get the relative information of the image
info = imfinfo(img_location);
M = info.Height;
N = info.Width;
```

    b. Then we get the complement of the input image also the erosion and dilation image：

```
% Get the conplement of the input image
complement_img = ~input_img;

% Get the erosion image and dilation image
erosion_img = operations(complement_img, 'erosion', 51, 3);
dilation_img = operations(erosion_img, 'dilation', 51, 3);
```

    c. To get the marker image we need add edges to the dilation image:

```
% First assign the dilation image to filling image
% And add edges to it
filling_img = dilation_img;
filling_img([1, end], :) = 1;
filling_img(:, [1, end]) = 1;
filling_img = filling_img & complement_img;

marker_img = zeros(M,N);
marker_img([1, end], :) = input_img([1, end], :) == 1;
marker_img(:, [1, end]) = input_img(:, [1, end]) == 1;
```

d. Similar to the former part, also repeat the dilation operations until it doesn't change any more：

```
% Then use dilation operation on filling image
% Until the filling image doesn't change any more
while(true)
    new_img = operations(filling_img, 'dilation', 3, 3);
    new_img = new_img & complement_img;
    if all(all(new_img == filling_img))
        break;
    end
    filling_img = new_img;
end

filling_img = ~filling_img;

% Show the pictures and output them
figures(...
{input_img, complement_img, marker_img, filling_img},...
'FIG9.31',...
{'FIG9.31(a) Origin input image', 'FIG9.31(b) Complement', 'FIG9.31(c) Marker',...
 'FIG9.31(d) Hole-filling'},...
true)
```

4. The border clearing part
    a. First also clear the former environments and get the input image:

```
% Clear the environment
clc
clear;

% Set the default location of input image file
img_location = '../../images/Fig0929(a)(text_image).tif';
input_img = double(imread(img_location));

% Get the relative information of the image
info = imfinfo(img_location);
M = info.Height;
N = info.Width;

% First get the border of the input image
border_img = zeros(M,N);
border_img([1, end], :) = input_img([1, end], :);
border_img(:, [1, end]) = input_img(:, [1, end]);
```

    b. Also to get the marker image we need add edges to the dilation image:

```
% First get the border of the input image
border_img = zeros(M,N);
border_img([1, end], :) = input_img([1, end], :);
border_img(:, [1, end]) = input_img(:, [1, end]);
```

    c. Also repeat the dilation operations until no changes:

```
% Then use dilation operation on filling image
% Until the filling image doesn't change any more
while(true)
    new_img = operations(border_img, 'dilation', 3, 3);
    new_img = new_img & input_img;
    imshow(new_img);
    if all(all(new_img == border_img))
        break;
    end
    border_img = new_img;
end
border_clearing_img = input_img - border_img;
```

d. Then we can get the border image after reduced by the original input image:

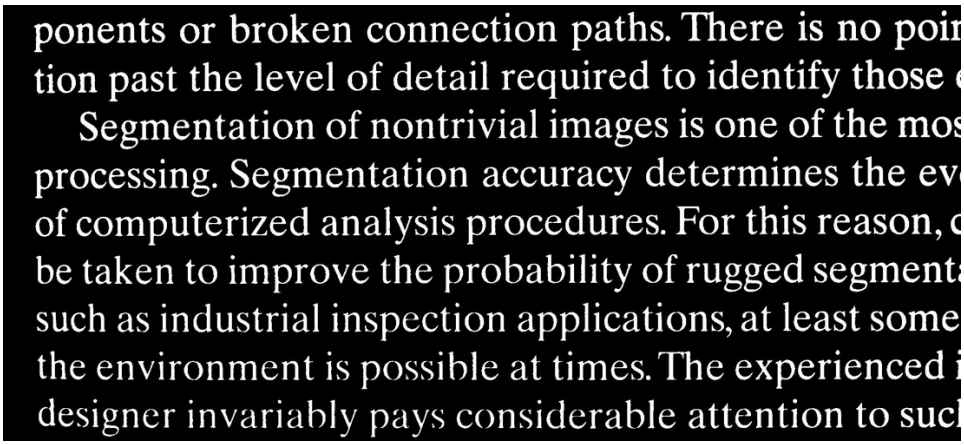
```
130  border_clearing_img = input_img - border_img;
131
132  % Show the pictures and output them
133  figures(...
134  {border_img, border_clearing_img, input_img},...
135  'FIG9.32',...
136  {'FIG9.32(a) Marker', 'FIG9.32(b) Border clearing', 'FIG9.32 Origin input image'},...
137  true)
138
```

5. The figure function
   Also use the similar function to show the pictures and figures:

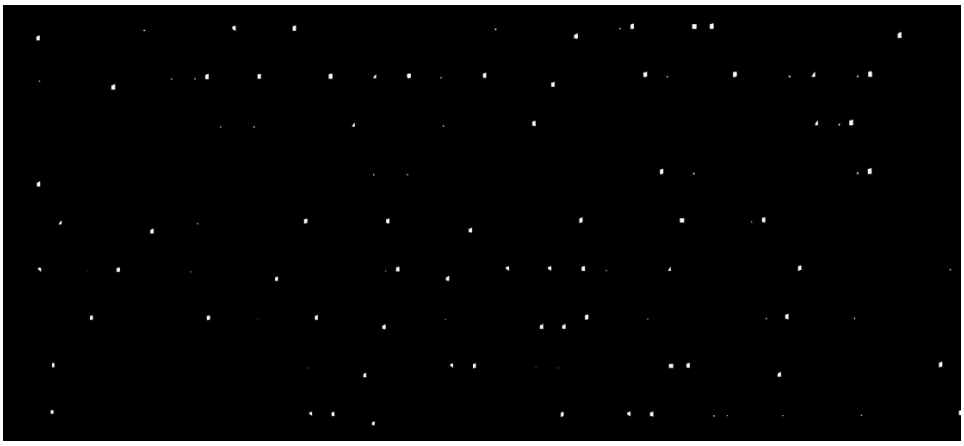
```
function [] = figures( input_imgs, figure_title, image_titles, output_file)
% Show the pictures and the figures

figure('NumberTitle','off','Name', figure_title)
column = 2;
[temp, num] = size(input_imgs);
row = ceil(num / column);

for i = 1:row
  for j = 1:column
    if((i - 1)* column + j) <= num
      subplot(row, column, (i - 1)* column + j)
      imshow(uint8(input_imgs{(i - 1) * column + j}),[]);
      title(char(image_titles{(i - 1) * column + j}))
      if output_file
        imwrite(uint8(input_imgs{(i - 1) * column + j} * 255), [char(image_titles{(i - 1) * column + j}), '.jpg']);
      end
    end
  end
end

end
```

Result

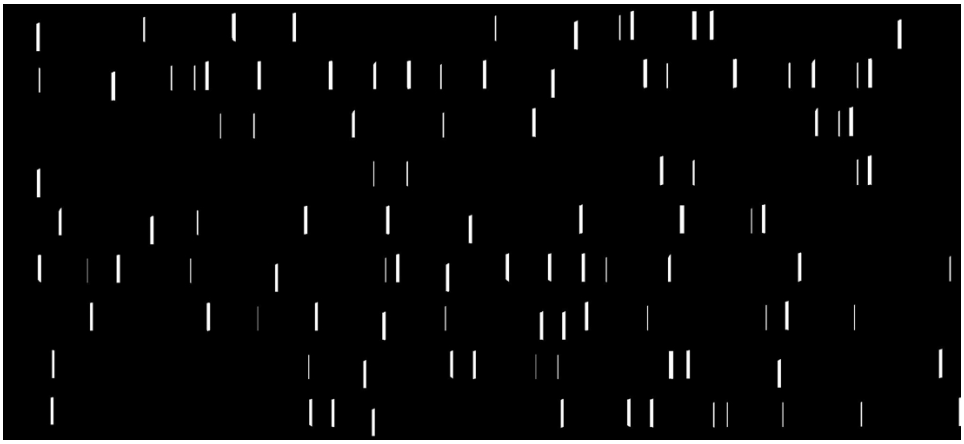
- **Opening by reconstruction**


ponents or broken connection paths. There is no poi
tion past the level of detail required to identify those
Segmentation of nontrivial images is one of the mos
processing. Segmentation accuracy determines the ev
of computerized analysis procedures. For this reason,
be taken to improve the probability of rugged segment
such as industrial inspection applications, at least some
the environment is possible at times. The experienced
designer invariably pays considerable attention to suc

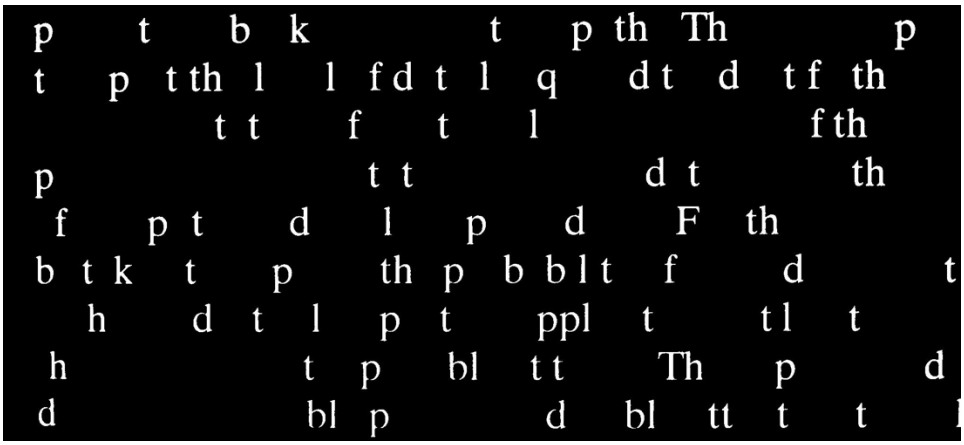
Original input image



Erosion using 51*1 mask

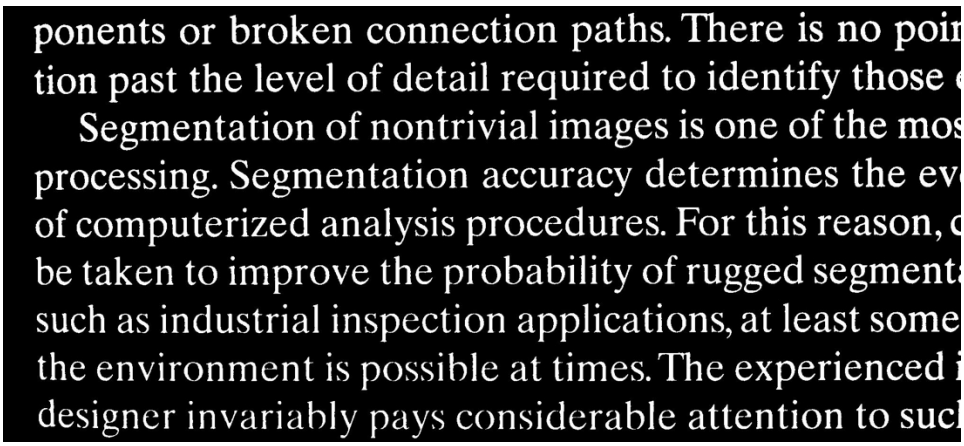


Dilation using 51*1 mask

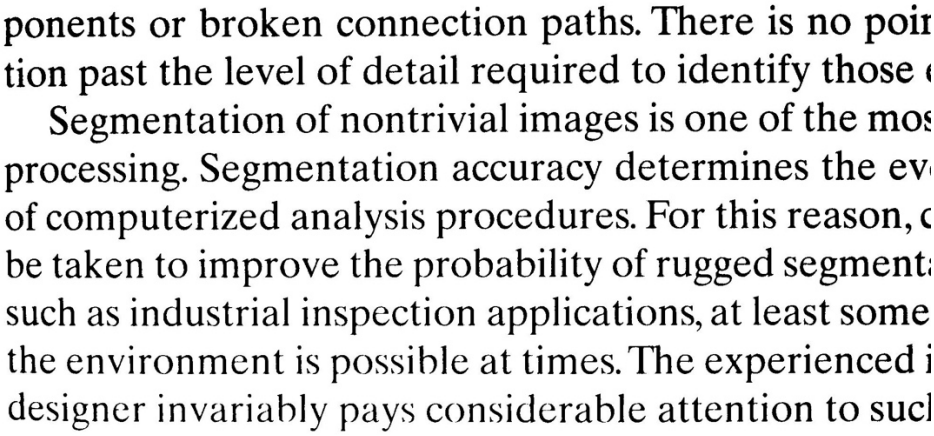


Final Result

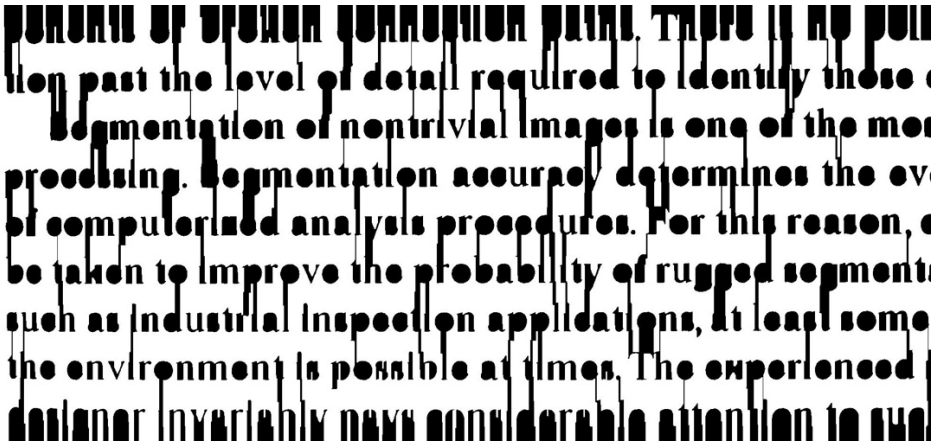
- **Filling holes**



Original input image

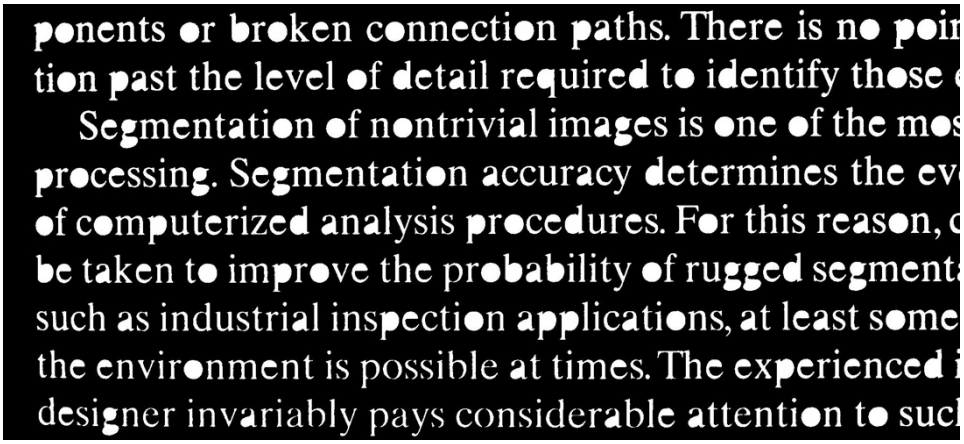


Complement of the input image



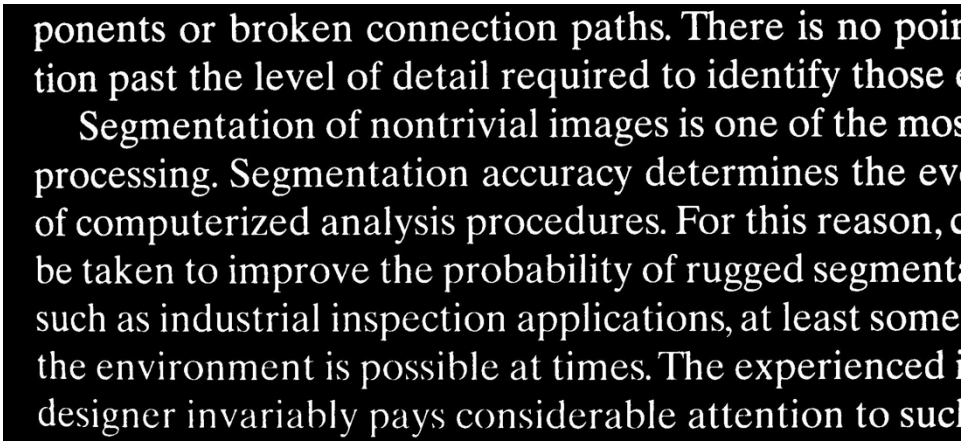
Dilation using 51*1 mask

Marker image



Final Result

- **Border clearing**



Original input image

Border image

ponents or broken connection paths. There is no poi
tion past the level of detail required to identify those
Segmentation of nontrivial images is one of the mo
processing. Segmentation accuracy determines the ev
of computerized analysis procedures. For this reason,
be taken to improve the probability of rugged segment
such as industrial inspection applications, at least some
the environment is possible at times. The experienced
designer invariably pays considerable attention to suc

Final result

Project 9 - Image segmentation

Problem

(a). Develop a program to implement the Roberts, Prewitt, Sobel, the Marr-Hildreth and the Canny edge detectors. Use the image 'building.tif' to test your detectors. (For technique details of Marr-Hildreth and Canny, please refer to pp.736-747 (3rd edition, Gonzalez DIP, or pp.705-725 for the electronic textbook) or MH-Canny.pdf at the same address of the slides.)
(b). Develop a program to implement the Otsu's method of thresholding segmentation, and compare the results with the global thresholding method using test image 'polymersomes.tif. (For technique details, please refer to pp.763-770 (3rd edition, Gonzalez DIP, or pp.742-747 for the electronic textbook, or Otsu.pdf at the same ftp address of slides.)

Environment

Matlab R2016a
Atom 1.22 (with language-matlab package)

Principle

- **Edge detection**
  - **Basic**
    As illustrated in the previous section, detecting changes in intensity for the purpose of finding edges can be accomplished using first- or second-order derivatives.

$$\nabla f \equiv \text{grad}(f) \equiv \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

    - **Roberts cross-gradient operators**
      The Roberts operators are based on implementing the diagonal differences:

$$g_x = \frac{\partial f}{\partial x} = (z_9 - z_5)$$

      and:

$$g_y = \frac{\partial f}{\partial y} = (z_8 - z_6)$$

−1	0	0	−1
0	1	1	0

Roberts

- **Prewitt operators**
  The simplest digital approximations to the partial derivatives using masks of size 3 * 3 are given by:

$$g_x = \frac{\partial f}{\partial x} = (z_7 + z_8 + z_9) - (z_1 + z_2 + z_3)$$

and

$$g_y = \frac{\partial f}{\partial y} = (z_3 + z_6 + z_9) - (z_1 + z_4 + z_7)$$

−1	−1	−1	−1	0	1
0	0	0	−1	0	1
1	1	1	−1	0	1

Prewitt

0	1	1	−1	−1	0
−1	0	1	−1	0	1
−1	−1	0	0	1	1

Prewitt

- **Sobel cross-gradient operators**
  A slight variation of the preceding two equations uses a weight of 2 in the center coefficient:

$$g_x = \frac{\partial f}{\partial x} = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)$$

and

$$g_y = \frac{\partial f}{\partial y} = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)$$

−1	−2	−1	−1	0	1
0	0	0	−2	0	2
1	2	1	−1	0	1

Sobel

0	1	2	−2	−1	0
−1	0	1	−1	0	1
−2	−1	0	0	1	2

Sobel

- **Mill-Hildreth edge detection**

The Laplacian of a Gaussian (LoG):

$$\nabla^2 G(x, y) = \left[\frac{x^2 + y^2 - 2\sigma^2}{\sigma^4}\right] e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The Marr-Hildreth algorithm consists of convolving the LoG filter with an input image, f(x, y):

$$g(x, y) = [\nabla^2 G(x, y)] \star f(x, y)$$

and then finding the zero crossings of g(x, y) to determine the locations of edges in f(x, y). Because these are linear processes, g(x, y) can be written also as

$$g(x, y) = \nabla^2 [G(x, y) \star f(x, y)]$$

The Marr-Hildreth edge-detection algorithm may be summarized as follows:

- Filter the input image with an n * n Gaussian low-pass filter obtained by sampling:

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

- Compute the Laplacian of the image resulting from Step1 using mask.
- Find the zero crossings of the image from Step 2.

- **Canny edge detection**

  - **Smooth the input image with a Gaussian filter**
    Let f(x, y) denote the input image and G(x, y) denote the Gaussian function:

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

    We form a smoothed image, $f_s(x, y)$, by convolving G and f:

$$f_s(x, y) = G(x, y) \star f(x, y)$$

  - **Compute the gradient magnitude and angle images.**
    This operation is followed by computing the gradient magnitude and direction (angle):

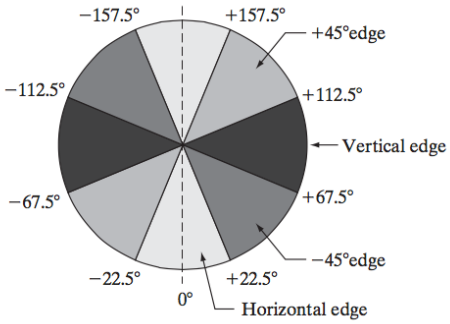
$$M(x, y) = \sqrt{g_x^2 + g_y^2}$$

    and

$$\alpha(x, y) = \tan^{-1}\left[\frac{g_y}{g_x}\right]$$

  - **Apply nonmaxima suppression to the gradient magnitude image.**
    Let d_1, d_2, d_3, $d_4$denote the four basic edge directions just discussed for a 3 * 3 region: horizontal, - 45°, vertical, and + 45°.

- Find the direction d_k that is closest to a(x, y).
- If the value of M(x, y) is less than at least one of its two neighbors along d_k, let $g_n(x, y) = 0$ (suppression); otherwise, let $g_n(x, y) = M(x, y)$



- **Use double thresholding and connectivity analysis to detect and link edges.**

$$g_{NH}(x, y) = g_N(x, y) \geq T_H$$

$$g_{NL}(x, y) = g_N(x, y) \geq T_L$$

$$g_{NL}(x, y) = g_{NL}(x, y) - g_{NH}(x, y)$$

- **Thresholding segmentation**
  - **Global thresholding segmentation**
    - Select an initial estimate for the global threshold, T.
    - Segment the image using T:

$$g(x, y) = \begin{cases} 1 & \text{if } f(x, y) > T \\ 0 & \text{if } f(x, y) \leq T \end{cases}$$

      This will produce two groups of pixels: G1 consisting of all pixels with intensity values > T, and G2 consisting of pixels with values <= T.
    - Compute the average (mean) intensity values m_1 and m_2 for the pixels in G1 and G2, respectively.
    - Compute a new threshold value:

$$T = \frac{1}{2}(m_1 + m_2)$$

    - Repeat Steps 2 through 4 until the difference between values of T in successive iterations is smaller than a predefined parameter ΔT.

- **Otsu's method of thresholding segmentation**

The basic idea is that well-thresholded classes should be distinct with respect to the intensity values of their pixels and, conversely, that a threshold giving the best separation between classes in terms of their intensity values would be the best (optimum) threshold.
Otsu's algorithm may be summarized as follows:

- Compute the normalized histogram of the input image. Denote the components of the histogram by p_i, i = 0, 1, 2, …, L - 1.
- Compute the cumulative sums, $P_1(k)$, for k = 0, 1, 2, …, L - 1, using

$$P_1(k) = \sum_{i=0}^{k} p_i$$

- Compute the cumulative means, m(k), for k = 0, 1, 2, …, L - 1, using

$$m(k) = \sum_{i=0}^{k} i p_i$$

- Compute the global intensity mean, m_G, using

$$m_G = \sum_{i=0}^{L-1} i p_i$$

- Compute the between-class variance, $\sigma_B^2(k)$, for k = 0, 1, 2, …, L - 1, using

$$\sigma_B^2(k) = \frac{\left[m_G P_1(k) - m(k)\right]^2}{P_1(k)\left[1 - P_1(k)\right]}$$

- Obtain the Otsu threshold, k*, as the value of k for which $\sigma_B^2(k)$ is maximum. If the maximum is not unique, obtain k* by averaging the values of k corresponding to the various maxima detected.
- Obtain the separability measure, η*, by evaluating

$$\eta(k) = \frac{\sigma_B^2(k)}{\sigma_G^2}$$

at k = k*.

Implementation

1. The edge detection part
    a. First clear the former environments and get the input image :

```
%% %%%%%%%%%%%%%%%%%%%%%%% Edge detection %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clc;
clear;

% Set the default location of input image file
img_location = '../../images/building.tif';
input_img = double(imread(img_location));

% Get the relative information of the image
info = imfinfo(img_location);
M = info.Height;
N = info.Width;

```

    b. Implement the Robert edge detection using function fiters():

```
%% Roberts edge detection
roberts_x_img = abs(filters(input_img, 'roberts', 'x'));
roberts_y_img = abs(filters(input_img, 'roberts', 'y'));
roberts_img = (roberts_x_img + roberts_y_img);

% Show the pictures and output them
figures(...
{input_img, roberts_x_img, roberts_y_img, roberts_img},...
'Roberts ',...
{'Origin input image', 'Roberts |gx|', 'Roberts |gy|', 'Roberts |gx|+|gy|'},...
true)
```

    c. Also the Prewitt edge detection:

```
%% Prewitt edge detection
prewitt_x_img = abs(filters(input_img, 'prewitt', 'x'));
prewitt_y_img = abs(filters(input_img, 'prewitt', 'y'));
prewitt_img = (prewitt_x_img + prewitt_y_img);

% Show the pictures and output them
figures(...
{input_img, prewitt_x_img, prewitt_y_img, prewitt_img},...
'Prewitt ',...
{'Origin input image', 'Prewitt |gx|', 'Prewitt |gy|', 'Prewitt |gx|+|gy|'},...
true)

```

    d. The Sobel edge detection:

```
%% Sobel edge detection
sobel_x_img = abs(filters(input_img, 'sobel', 'x'));
sobel_y_img = abs(filters(input_img, 'sobel', 'y'));
sobel_img = (sobel_x_img + sobel_y_img);

% Show the pictures and output them
figures(...
{input_img, sobel_x_img, sobel_y_img, sobel_img},...
'Sobel ',...
{'Origin input image', 'Sobel |gx|', 'Sobel |gy|', 'Sobel |gx|+|gy|'},...
true)

```

    e. Smooth the result of Sobel edge detectioni using 5x5 mask:

```
%% Smooth the result of sobel
smooth_input_img = filters(input_img, 'smooth');
sobel_smooth5x5_x_img = abs(filters(smooth_input_img, 'sobel', 'x'));
sobel_smooth5x5_y_img = abs(filters(smooth_input_img, 'sobel', 'y'));
sobel_smooth5x5_img = (sobel_smooth5x5_x_img + sobel_smooth5x5_y_img);

% Show the pictures and output them
figures(...
{smooth_input_img, sobel_smooth5x5_x_img, sobel_smooth5x5_y_img, sobel_smooth5x5_img
'Sobel smoothed input image with 5x5 mask',...
{'Smoothed input image', 'Sobel Smoothed |gx|', 'Sobel Smoothed |gy|', 'Sobel Smooth
true)
```

f. To implement the Marr-Hildreth edge detection, we need first get the Laplacian of Gaussian (LoG) matrix and a matrix used to record the negative or posotive of LoG matrix:

```
%% Marr-Hildreth edge detection.
% The Laplacian of Gaussian
LoG_img = filters(input_img, 'log', [25,4]);
sign = zeros(M, N);
sign(LoG_img > 0) = 1;
sign(LoG_img < 0) = -1;
```

Then find zero crossings with a threshold of 0:

```
% Find zero crossings with a threshold of 0.
zero_crossing_0 = zeros(M, N);
for i = 2:M-1
    for j = 2:N-1
        if sign(i-1, j-1) * sign(i+1, j+1) < 0 || ...
                sign(i-1, j) * sign(i+1, j) < 0 || ...
                sign(i, j-1) * sign(i, j+1) < 0 || ...
                sign(i+1, j-1) * sign(i-1, j+1) < 0
            zero_crossing_0(i, j) = 1;
        end
    end
end
```

Also with a threshold of 0.04:

```
% Find zeros crossings with threshold of 4%
zero_crossing_04 = zeros(M, N);
threshold = 0.04 * (max(LoG_img(:)) - min(LoG_img(:)));
for i = 2:M-1
    for j = 2:N-1
        if sign(i-1, j-1) * sign(i+1, j+1) < 0 && ...
                abs(LoG_img(i-1, j-1) - LoG_img(i+1, j+1)) > threshold
            zero_crossing_04(i, j) = 1;
        elseif sign(i, j-1) * sign(i, j+1) < 0 && ...
                abs(LoG_img(i, j-1) - LoG_img(i, j+1)) > threshold
            zero_crossing_04(i, j) = 1;
        elseif sign(i-1, j) * sign(i+1, j) < 0 && ...
                abs(LoG_img(i-1, j) - LoG_img(i+1, j)) > threshold
            zero_crossing_04(i,j) = 1;
        elseif sign(i+1, j-1) * sign(i-1, j+1) < 0 && ...
                abs(LoG_img(i+1, j-1) - LoG_img(i-1, j+1)) > threshold
            zero_crossing_04(i,j) = 1;
        end
    end
end
```

To show the pictures there needs to do some scaling :

```
% To show the image seems like in the book, do some scaling
LoG_img = LoG_img - min(LoG_img(:));
output_LoG_img = LoG_img / max(LoG_img(:));
output_LoG_img = output_LoG_img * 255;
zero_crossing_0 = zero_crossing_0 * 255;
zero_crossing_04 = zero_crossing_04 * 255;

% Show the pictures and output them
figures(...
{input_img, output_LoG_img, zero_crossing_0, zero_crossing_04},...
'Marr-Hildreth edge detection',...
{'Origin input image', 'LoG ', 'Zero crossings with threshold 0', ...
'Zero crossings with threshold 0.04'},...
true)
```

g. To implement the Canny edge detection, we need to use Gaussian blur to handle the image, there using filter() function:

```
%% Canny edge detection.
% Gaussian blur.
gaussian_img = filters(input_img, 'gaussian', [25, 4]);
```

Then calculate the gradient magnitude and angle images:

```
% Calculate the gradient magnitude and angle images.
magnitude_x_img = filters(gaussian_img, 'sobel', 'x');
magnitude_y_img = filters(gaussian_img, 'sobel', 'y');
magnitude_img = sqrt(magnitude_x_img.^2 + magnitude_y_img.^2);

angle = zeros(M, N);
% For different direction set different number:
%   Horizonal edge - 1
%   Vertical edge - 2
%   -45 degree edge - 3
%   +45 degree edge - 4
for i = 1:M
    for j = 1:N
        if magnitude_y_img(i, j) == 0
            angle(i, j) = 1;
        else
            t = magnitude_x_img(i, j) / magnitude_y_img(i, j);
            if abs(t) >= tan(0.375 * pi)
                angle(i, j) = 1;
            elseif abs(t) <= tan(0.125 * pi)
                angle(i, j) = 2;
            elseif t > 0
                angle(i, j) = 3;
            else
                angle(i, j) = 4;
            end
        end
    end
```

Then apply nonmaxima suppression to the gradient magnitude image:

```
suppression_img = zeros(M, N);
for i = 2:M-1
    for j = 2:N-1
        t = angle(i, j);
        m = magnitude_img(i - 1:i + 1, j - 1:j + 1);
        if (t == 1 && m(2, 2) > m(3, 2) && m(2, 2) > m(1, 2)) || ...
                (t == 2 && m(2, 2) > m(2, 3) && m(2, 2) > m(2, 1)) || ...
                (t == 3 && m(2, 2) > m(1, 1) && m(2, 2) > m(3, 3)) || ...
                (t == 4 && m(2, 2) > m(1, 3) && m(2, 2) > m(3, 1))
            suppression_img(i,j) = m(2, 2);
        end
    end
end

```

For the convenience of thresholding process, scale the supression matrix:

```
% Scale the suppression image to [0,1]
suppression_img = suppression_img - min(suppression_img(:));
suppression_img = suppression_img / max(suppression_img(:));
```

Then use double thresholding and connectivity analysis to detect and link edges and use queue to store high pixel positions:

```
% Use double thresholding and connectivity analysis to detect and link edges
low_threshold_img = zeros(M, N);
high_threshold_img = zeros(M, N);
low_threshold_img(suppression_img > 0.04) = 1;
high_threshold_img(suppression_img > 0.10) = 1;

% Use queue to store high pixel positions.
[x, y] = find(high_threshold_img);
head = 1;
tail = length(x);
canny_edge_img = high_threshold_img;
while(head <= tail)
    x0 = x(head);
    y0 = y(head);
    for i = x0 - 1:x0 + 1
        for j = y0 - 1:y0 + 1
            if low_threshold_img(i, j) && ~canny_edge_img(i, j)
                tail = tail + 1;
                x(tail) = i;
                y(tail) = j;
                canny_edge_img(i, j) = 1;
            end
        end
    end
    head = head + 1;
end
```

For show the pictures better we scale the images back to [0,255], and show them:

```
% To show the image seems like in the book, do some scaling
suppression_img = suppression_img * 255;
low_threshold_img = low_threshold_img * 255;
high_threshold_img = high_threshold_img * 255;
canny_edge_img = canny_edge_img * 255;


% Show the pictures and output them
figures(...
{input_img, gaussian_img, magnitude_img, suppression_img},...
'Canny edge detection',...
{'Origin input image', 'Gaussian blur', 'Gradient magnitude', 'Nonmaxima suppression'},...
true)
figures(...
{input_img, low_threshold_img, high_threshold_img, canny_edge_img},...
'Canny edge detection use double thresholding',...
{'Origin input image', 'Low threshold 0.04', 'High threshold 0.10', 'Canny edge detection'},..
true)
```

** Actually for the final result of Robert, Prewitt, Sobel and smoothed Sobel, it should be the (|gx| + |gy|) / 2 , but when represent on this word/pdf report the effect is not that good for the pictures are scaled small, so I just let the result to be |gx| + |gy|.*

2. The filters function
    a. First we need to calculate the mask according to the filter pattern:

```
% Get the mask
switch lower(pattern)
case patterns{1}
    mask = [-1, 0; 0, 1];
    if strcmp(parameters, 'y')
        mask = [0, -1; 1, 0];
    end
  case patterns{2}
    mask = [-1, -1, -1; 0, 0, 0; 1, 1, 1];
    if strcmp(parameters, 'y')
        mask = mask';
    end
  case patterns{3}
    mask = [-1, -2, -1; 0, 0, 0; 1, 2, 1];
    if strcmp(parameters, 'y')
        mask = mask';
    end
  case patterns{4}
    mask = ones(5,5) / 25;
  case patterns{5}
    r = (parameters(1) - 1)/2;
    [x, y] = meshgrid(-r:r, -r:r);
    mask = exp(-(x .* x + y .* y)/(2 * parameters(2) * parameters(2)));
    if sum(mask(:)) ~= 0
        mask  = mask/sum(mask(:));
    end
  case patterns{6}
    r = (parameters(1) - 1)/2;
    [x, y] = meshgrid(-r:r, -r:r);
    mask = exp(-(x .* x + y .* y)/(2 * parameters(2)^2));
    mask = mask .* (x .* x + y .* y - 2 * parameters(2)^2)/(parameters(2)^4);
    mask = mask - sum(mask(:))/numel(mask);
  otherwise
    error(['The', ' ', pattern, ' ', 'filter pattern can not be found.'])
end
```

b. Then we use zero padding to pad the input image, for the size of mask maybe even so we need some handling to left and right shift:

```
% Zero padding
padding_m = floor((m - 1)/2);
padding_m_ = m - padding_m - 1;
padding_n = floor((n - 1)/2);
padding_n_ = n - padding_n - 1;

padding_img = padarray(input_img, [padding_m, padding_n]);
padding_img = zeros(M + m - 1, N + m - 1);
padding_img(padding_m + 1:padding_m + M, padding_n + 1:padding_n + N) = input_img;
```

c. At last we can just mask the output image:

```
output_img = zeros(M, N);
for i = padding_m + 1:padding_m + M
    for j = padding_n + 1:padding_n + N
        temp = padding_img(i - padding_m:i + padding_m_, j - padding_n:j + padding_n_) .* mask;
        output_img(i - padding_m, j - padding_n) = sum(temp(:));
    end
end
```

3. The thresholding segmentation part
   a. Also first clear the former environments and get the input image:

```
%% %%%%%%%%%%%%%%%%% Thresholding segmentation  %%%%%%%%%%%%%%%%%%%%%%%%%
clc;
clear;

% Set the default location of input image file
img_location = '../../images/polymersomes.tif';
input_img = double(imread(img_location));

% Get the relative information of the image
info = imfinfo(img_location);
M = info.Height;
N = info.Width;
```

   b. Implement the global thresholding method according to the formula:

```
% global thresholding method.
t = mean(mean(input_img));
delta = 0.1;
delta_t = 255;

while(delta_t > delta)
    delta_t = t;
    G1 = input_img(input_img > t);
    G2 = input_img(input_img <= t);
    mG1 = mean(G1);
    mG2 = mean(G2);
    t = (mG1 + mG2) / 2;
    delta_t = abs(delta_t - t);
end

global_threshold_img = zeros(M, N);
global_threshold_img(input_img > t) = 1;
global_threshold_img = global_threshold_img * 255;
figures(...
{input_img, global_threshold_img},...
'Global thresholding',...
{'Original input image', 'Global thresholding'},...
true)
```

c. Implement the Otsu's method in similar way:

```
% Otsu's method
L = 256;
Pi = zeros(L, 1);
P1k = zeros(L, 1);
mk = zeros(L, 1);

for i = 1:L
    Pi(i) = sum(sum(input_img == (i - 1))) / (M * N);
    if i == 1
        P1k(i) = Pi(i);
    else
        P1k(i) = P1k(i - 1) + Pi(i);
        mk(i) = mk(i - 1) + i * Pi(i);
    end
end
mG = mk(L);

sigma = ((mG * P1k - mk) .^ 2) ./ (P1k .* (1 - P1k));
sigma(isnan(sigma)) = 0;
k = mean(find(sigma == max(sigma))) - 1;

otsu_img = zeros(M, N);
otsu_img(input_img > k) = 1;
otsu_img = otsu_img * 255;
figures(...
{input_img, otsu_img},...
'Otsu`s method',...
{'Original input image', 'Otsu`s method'},...
true)
```

4. The figures function
   Use the same funciton in the former projects to show the pictures and figures also output to files:

```
% Figures
function [] = figures( input_imgs, figure_title, image_titles, output_file)
% Show the pictures and the figures

figure('NumberTitle','off','Name', figure_title)
column = 2;
[temp, num] = size(input_imgs);
row = ceil(num / column);

for i = 1:row
  for j = 1:column
    if((i - 1)* column + j) <= num
      subplot(row, column, (i - 1)* column + j)
      imshow(uint8(input_imgs{(i - 1) * column + j}),[]);
      title(char(image_titles{(i - 1) * column + j}))
      if output_file
        imwrite(uint8(input_imgs{(i - 1) * column + j}), [char(image_titles{(i - 1) * column + j]
      end
    end
  end
end

end
```

Result

- **Edge detection**
  - **Robert edge detection**

Original input image

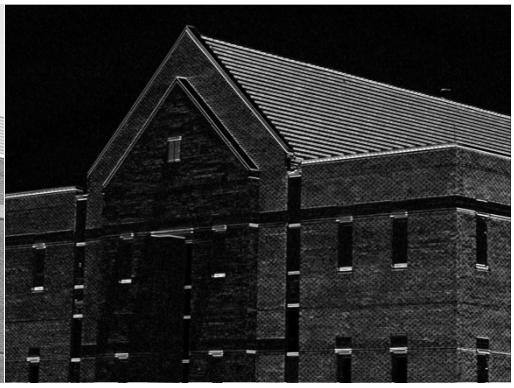
|gx|

|gy|

|gx| + |gy|

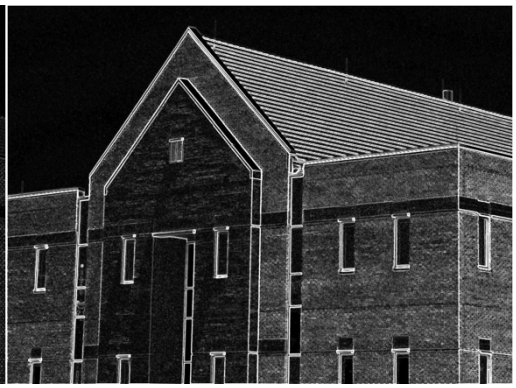
  - **Prewitt edge detection**

Original input image



|gx|

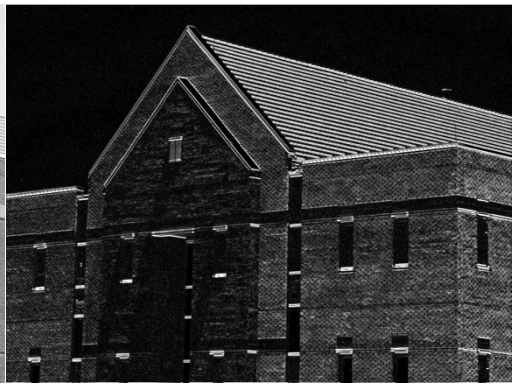
|gy|



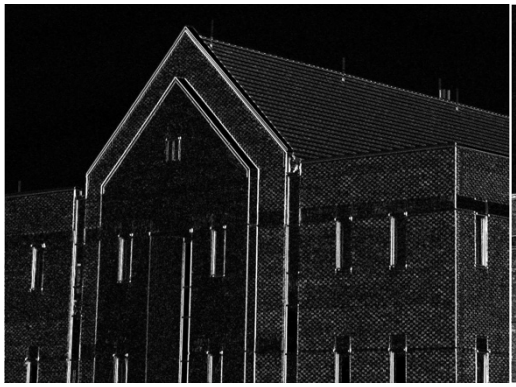
|gx| + |gy|

- **Sobel edge detection**

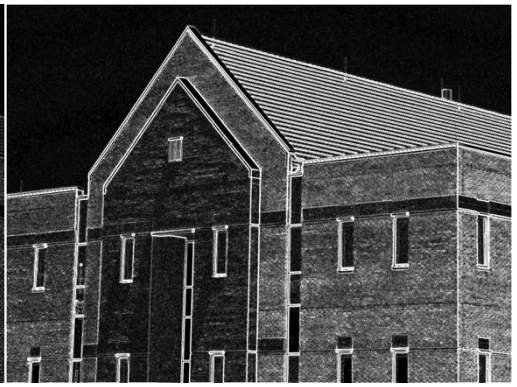
Original input image



|gx|



|gy|



|gx| + |gy|

- **Sobel smoothed edge detection**

Smoothed input image

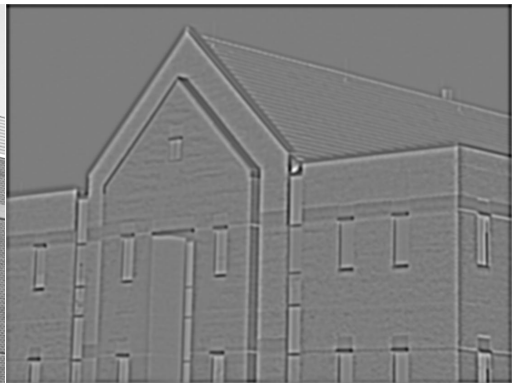
|gx|

|gy|

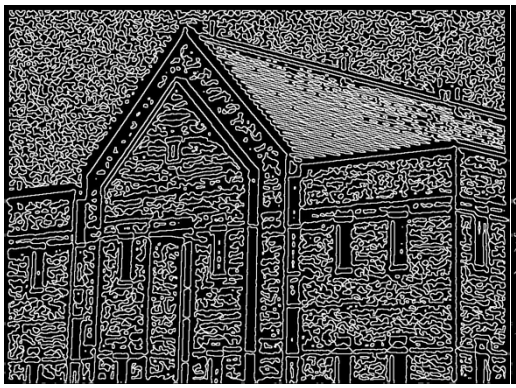
|gx| + |gy|

- **Marr-Hidereth edge detection**

Original input image



Laplacian of Gaussian image



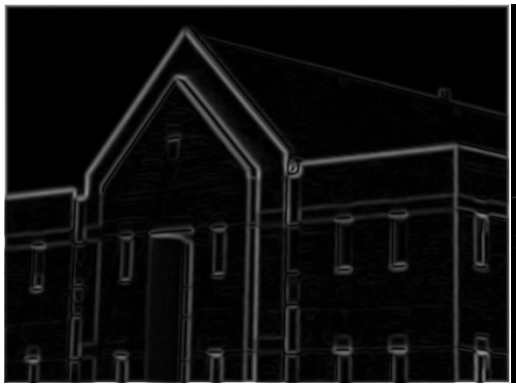
Zero crossing with threshold 0

Zero crossing with threshold 0.04

- **Canny edge detection**

Original input image

Gaussian blurred image



Gradient magnitude image

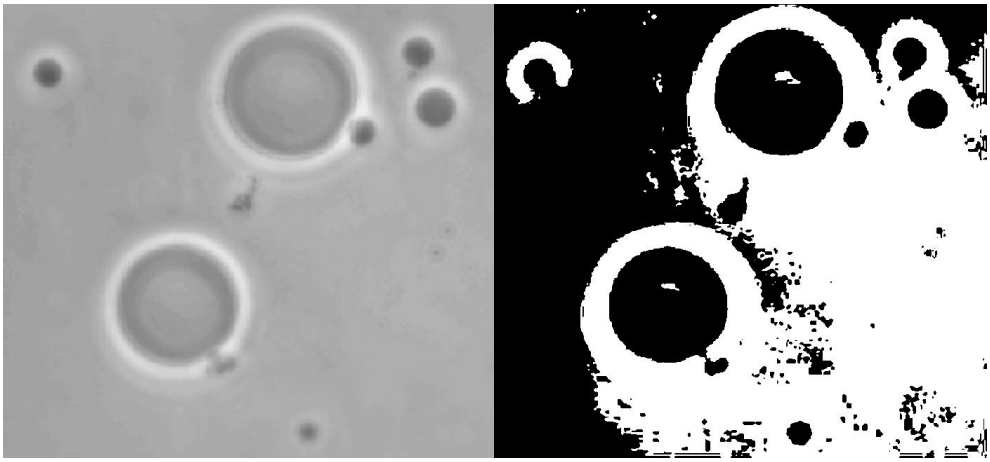
Nonmaxima suppression image

Low threshold 0.04

High threshold 0.10

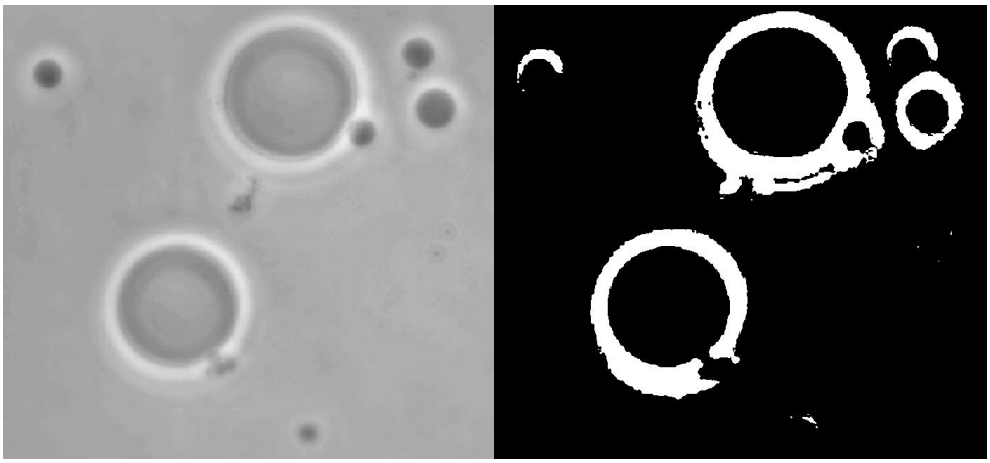
Final result

- **Thresholding segmentation**
  - **Global thresholding method**



Original input image Final result

  - **Otsu's method**



Original input image Final result