# Digital Image Processing 2017fall
# Project Report

Name: Wang Yiqing
Student Number: 515 030 910 456
Email: WangYiqing_2015@sjtu.edu.cn

2017-12-10

# Contents

# 1 Project 1 - Histogram Equalization

## 1.1 Project Proposal

In this project we implement histogram equalization. First, plot the histogram of an image, then implement histogram equalization and display the equalized image and histogram. Test images are *Fig1.jpg, Fig2.jpg.*

## 1.2 Preliminaries

Let $r$ denote the intensity of a pixel in the original image and $r \in [0, L-1]$. We consider a transformation

$$s = T(r) \quad 0 \leq r \leq L-1 \tag{1.1}$$

that produce an output intensity level $s$ for every pixel in the input image having intensity $r$. We assume that:
**(a)** $T(r)$ is a monotonically increasing function in the interval $0 \leq r \leq L-1$; and
**(b)** $0 \leq T(r) \leq L-1$ for $r \in [0, L-1]$.
Condition **(a)** and **(b)** guarantee the existing of inverse function $r = T^{-1}(s)$ which is monotonically increasing. The central idea is that *intensity levels of an image can be viewed as random variables in the interval* $[0, L-1]$ *and can be described by probability density function (PDF).* Let $p_r(r)$ and $p_s(s)$ to be the PDF of $r$ and $s$ respectively. Thus we can apply the basic result of probability theory that *if* $T(r)$ *is differentiable over the range of interest, we have this below relationship between* $p_r(r)$ *and* $p_s(s)$*:*

$$p_s(s) = p_r(r) \left| \frac{dr}{ds} \right| \tag{1.2}$$

Then consider the cumulative distribution function(CDF) which is exactly used here as the transformation function $T(r)$

$$s = T(r) = (L-1) \int_0^r p_r(w) dw \tag{1.3}$$

Now we are ready to prove the transformation really does the histogram equalization. Firstly, compute the derivatives. The last = is from the assumption **(a)** stating the monotonically increasing.

$$\frac{ds}{dr} = \frac{dT(r)}{dr} = (L-1)p_r(r) = \left| \frac{ds}{dr} \right| \tag{1.4}$$

Thus, substituting the result for Eq.1.2, we get the desired result

$$p_s(s) = p_r(r) \frac{1}{(L-1)p_r(r)} = \frac{1}{L-1} \tag{1.5}$$

which show $p_s(s)$ follows uniform distribution.

## 1.3 Histogram equalization

Histogram equalization on an image of size $M \times N$ is like a discrete version of the process in the preliminaries section.

$$p_r(r_k) = \frac{n_k}{MN} \qquad k = 0, 1, 2, ..., L-1 \tag{1.6}$$

where $n_k$ is the number of pixels that have intensity $r_k$. The discrete form of Eq.1.3 is

$$s_k = T(r_k) = (L-1) \sum_{j=0}^{k} p_r(r_j) = \frac{L-1}{MN} \sum_{j=0}^{k} n_k \qquad k = 0, 1, 2, ..., L-1 \tag{1.7}$$

Based on these discrete form equation, I implement the matlab function and test them on *Fig1.jpg, Fig2.jpg* and get the results listed in 1.1 and 1.2 respectively.

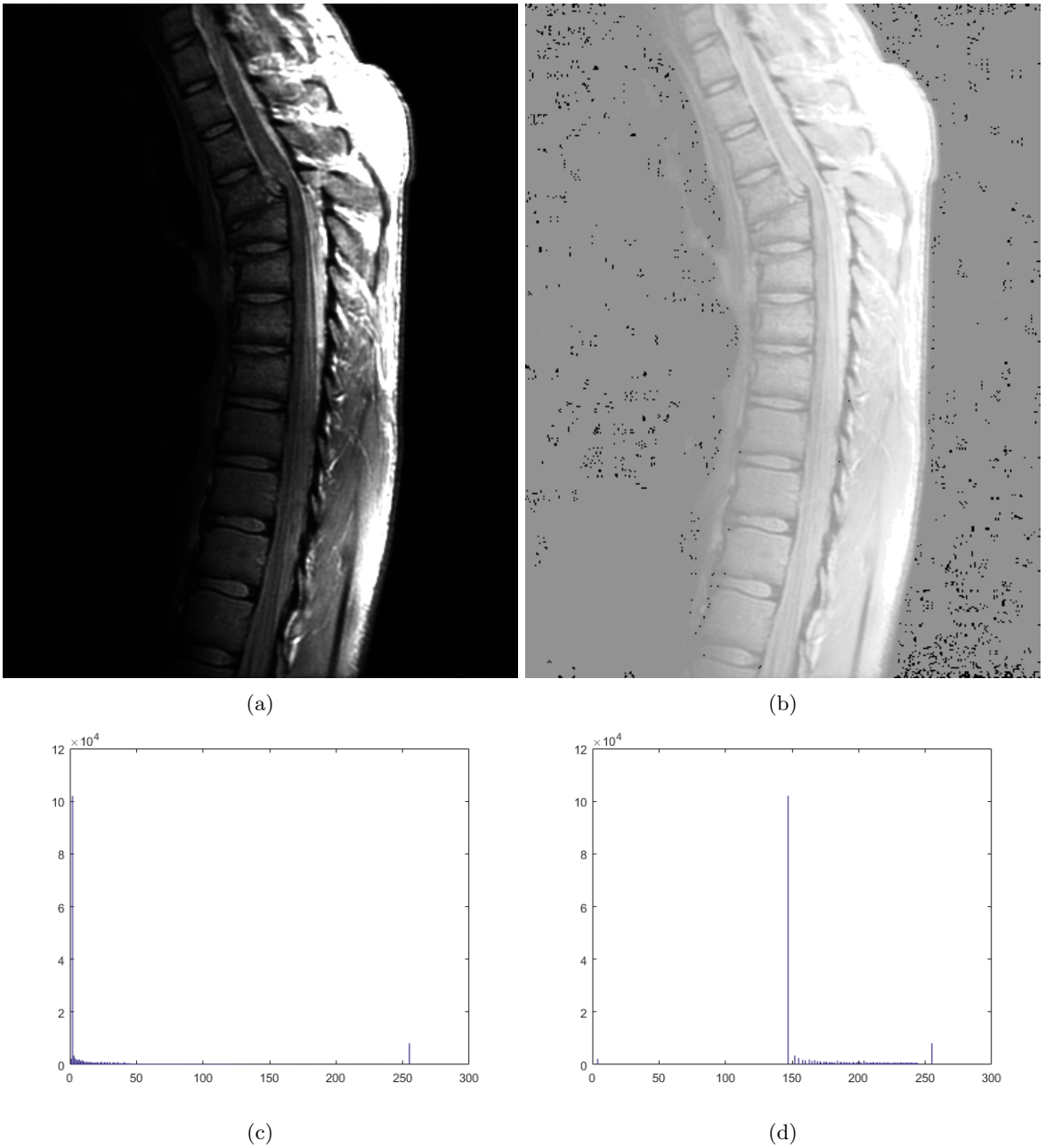



Figure 1.1: Results of Fig1.jpg. (a)Original image. (b)Processed image after applying histogram equalization. (c)Histogram of original image. (d)Histogram of processed image.

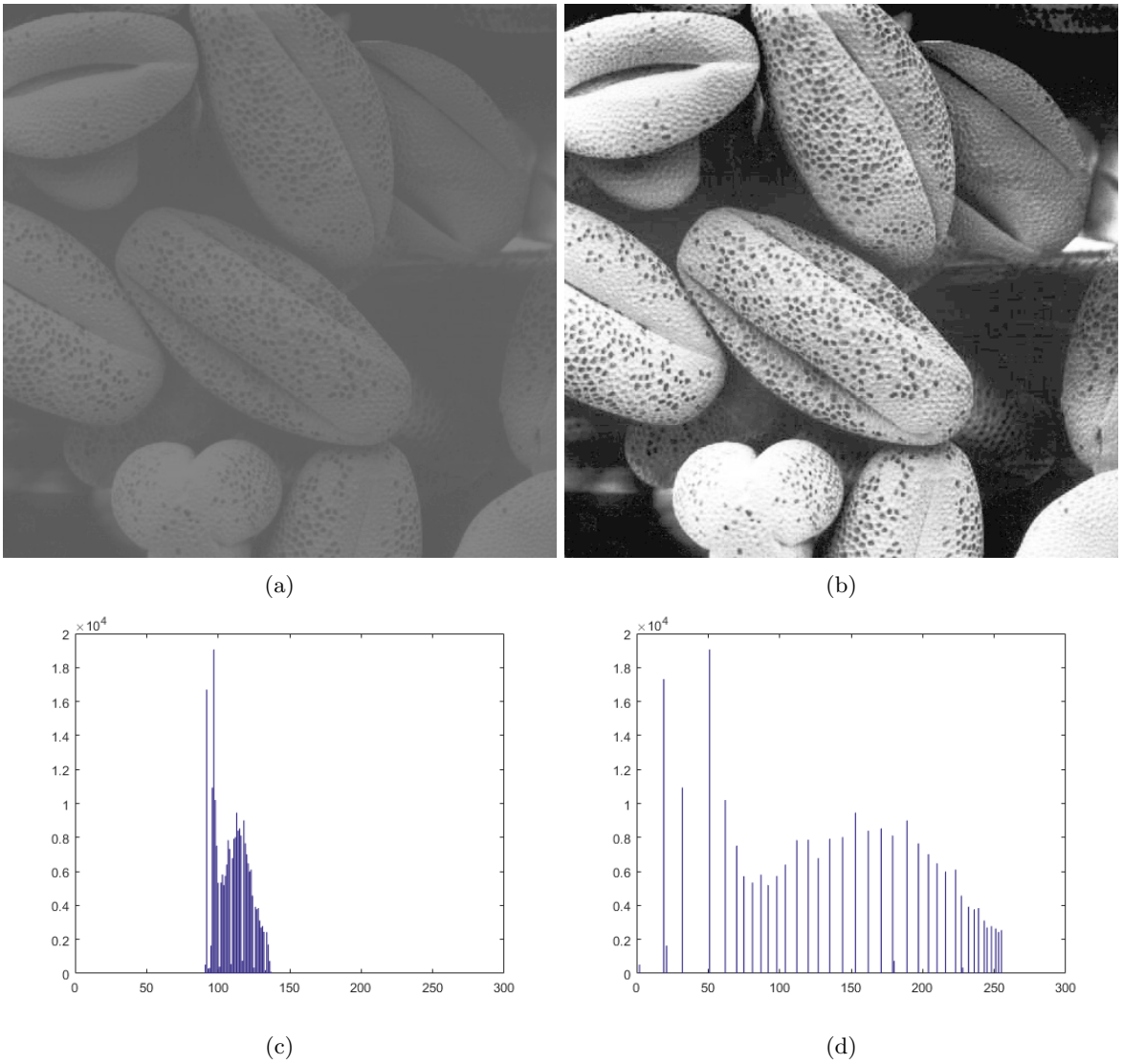



Figure 1.2: Results of Fig2.jpg. (a)Original image. (b)Processed image after applying histogram equalization. (c)Histogram of original image. (d)Histogram of processed image.

## 1.4 Discussion

We can see that histograms are quite different from continuous functions because there are gaps between each horizontal value. So the histogram of the equalized image is like a sparse version of the original image. The result of Fig2(low contrast ratio) shows an increasing on contrast ratio and it's more useful than the original result. However, Fig1(high contrast ratio) is not a good example of histogram equalization. The affect is just like a intensity transition. We can simply draw a not so serious conclusion that histogram equalization is useful for images with low contrast ratio but not for images with high contrast ratio.

## 1.5 Implementation

There is some key part of my implementation.

```
function [x, y] = histShow( imgf )
%HISTSHOW display the histogram graph of the imgf
% x - the horizontal axis of histogram,
% y - the vertical axis of histogram
g = imgf(:) + 1;
n = length(g);
x = (1 : 256);
y = zeros(1, 256);
for i = (1 : n)
    y(g(i)) = y(g(i)) + 1;
end

end

function [ imgg ] = histEqual( imgf )
%HISTEQUAL
%
[x, y] = histShow(imgf);
T = zeros(1, 256);
a = 0;
g = imgf(:);
n = length(g);
for i = (1 : 256)
    T(i) = a + y(i);
    a = T(i);
end
T = round(255 * T / n);
for i = (1 : n)
    g(i) = T(g(i)+1);
end
imgg = reshape(g, size(imgf));

end
```

# 2 Project 2 - Spatial Enhancement Methods

## 2.1 Project Proposal

Implement the image spatial enhancement task showed in text book Figure 3.43. The steps involve Laplacian filter, Sobel filter, average filter and power law transformation.

## 2.2 Preliminaries

### 2.2.1 Basic Spatial Filtering

In general, linear spatial filtering of an image of size $M \times N$ with a filter of size $m \times n$ is

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t) \tag{2.1}$$

where $a = (m-1)/2$ and $b = (n-1)/2$. There are two closely related concepts related to spatial filtering. One is *correlation* and the other is *convolution*. Correlation of filter $w(x,y)$ is defined as

$$\sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t) \tag{2.2}$$

while convolution is defined as

$$\sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x-s, y-t) \tag{2.3}$$

The confusion point is that we usually call spatial filter as convolution filter, convolution mask or convolution kernel but the terms are not necessarily refer to the true convolution operation defined in Eq.2.3.

**Smoothing spatial filters**

Smoothing filters are used for blurring and for noise reduction. The simplest smoothing filter is averaging filters, which is also called lowpass filter. $M \times N$ averaging filter can be represented as

$$g(x,y) = \frac{\sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t)}{\sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)} \tag{2.4}$$

The simple idea of filter size choosing is that choose the approximately the same size of the noisy you want to reduce. Average filter is an example of linear spatial filters. There are also nonlinear order-static spatial filters using the ranking information of each pixel. We will talk about order-static filters in *Project 8*.

**Sharpening spatial filters**

The principal objective of sharpening is to highlight transitions in intensity. Because averaging is analogous to integration, it is logical to conclude that sharpening can be accomplished by spatial differentiation. Thus we consider the sharpening filters based on first- and second-order derivatives.

**Second-order derivatives - the Laplacian**

Laplacian is defined as

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{2.5}$$

In x-direction and y-direction we have

$$\frac{\partial^2 f}{\partial x^2} = f(x+1,y) - f(x,y) - (f(x,y) - f(x-1,y)) = f(x+1,y) + f(x-1,y) - 2f(x,y) \tag{2.6}$$

$$\frac{\partial^2 f}{\partial y^2} = f(x,y+1) - f(x,y) - (f(x,y) - f(x,y-1)) = f(x,y+1) + f(x,y-1) - 2f(x,y) \tag{2.7}$$

Thus we get the discrete Laplacian of two variables:

$$nabla^2 f(x,y) = f(x+1,y) + f(x-1,y) f(x,y+1) + f(x,y-1) - 4(x,y) \tag{2.8}$$

Based on Eq.2.8 we have 4 masks displayed in Fig.2.1. The basic way we use Laplacian for image sharpening is

$$g(x,y) = f(x,y) + c\left[\nabla^2 f(x,y)\right] \tag{2.9}$$

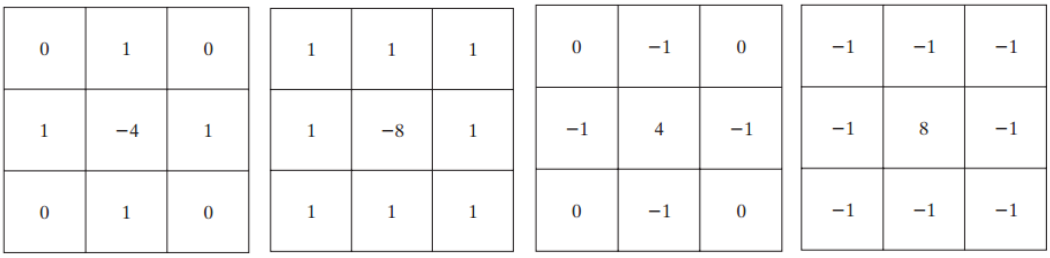



Figure 2.1: 4 Laplacian masks. **(a)**Direct implementation of Eq.2.8. **(b)**Extent (a) by adding two diagonal terms. **(c)(d)** sign inverse of (a)(b) relatively. More common in practice.

where $c = 1$ if we use Fig.2.1c or Fig.2.1d.

**First-order derivatives - the gradient**

First-order derivatives in image processing using the magnitude of the gradient. The gradient vector is

$$\nabla f \equiv \text{grad}(f) \equiv \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{2.10}$$

The magnitude of vector $\nabla f$ defined as

$$M(x, y) = mag(\nabla f) = \sqrt{g_x^2 + g_y^2} \approx |g_x| + |g_y| \tag{2.11}$$

is the value of the rate of change in the direction of the gradient vector. The second $=$ is a frequently used approximate to avoid square roots. A widely used approximate filter masks implementing gradient is Sobel defined as

$$g_x = \frac{\partial f}{\partial x} = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3) \tag{2.12}$$

and

$$g_y = \frac{\partial f}{\partial y} = (z_3 + 2z_6 + z_9) - (z_1 + 4z_2 + z_7) \tag{2.13}$$

$3 \times 3$ Sobel filter is displayed in Fig.2.2

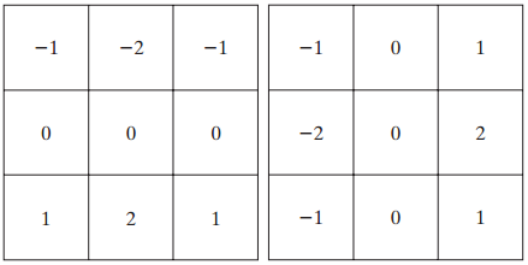



Figure 2.2: $3 \times 3$ SObel mask. Left is on x direction, right is on y direction.

## 2.3 Experiment - Combining Spatial Enhancement

We conduct the experiment on the Fig.2.3a image of whole body bone scan. Our goal is to enhance this image by sharpening it and by bringing out more of the skeletal detail. The narrow dynamic ranging of the intensity levels and high noise content make this problem difficult. The process and result images are displayed in Fig.2.3 and Fig.2.4. Here, I describe the detail process of the enhancement.

We first use Laplacian with the mask in Fig.2.1d to extract the sharp transitions in intensity. Note that Fig.2.3b is a scaled result because the numerical matrix result contains negative values which can not be visualized as image by matlab default function. After obtaining Laplacian, we add Laplacian(not the scaled one by the real laplacian) to the original image to get the sharpened Fig.2.3c. This step is just follow Eq.2.8. In Fig.2.3d we compute the Sobel gradient of original image using masks in Fig.2.2. Edges are more dominant than ones in Laplacian. In Fig.2.4e we use $5 \times 5$averaging filter to smooth the Sobel gradient for noisy reduction purpose. Its not proper to use median filter because medical image processing requires high level of convince. In Fig.2.4f, we multiply the Laplacian with Sobel. Multiply seems like a strange operation but we can consider it as use

Sobel to mask Laplacian in order to strengthen edges and reduce noisy. In Fig.2.4g we sum up the product and the original image. Finally in Fig.2.4h we increase the dynamic range of the sharpened image by powerlaw transformation which is defined as

$$s - cr^{\gamma} \tag{2.14}$$

where we use $\gamma = 0.5$ and $c = 1$. We use $\gamma < 1$ to spread the intensity level and note that histogram equalization would not give use satisfying result as discussed in Project 1.

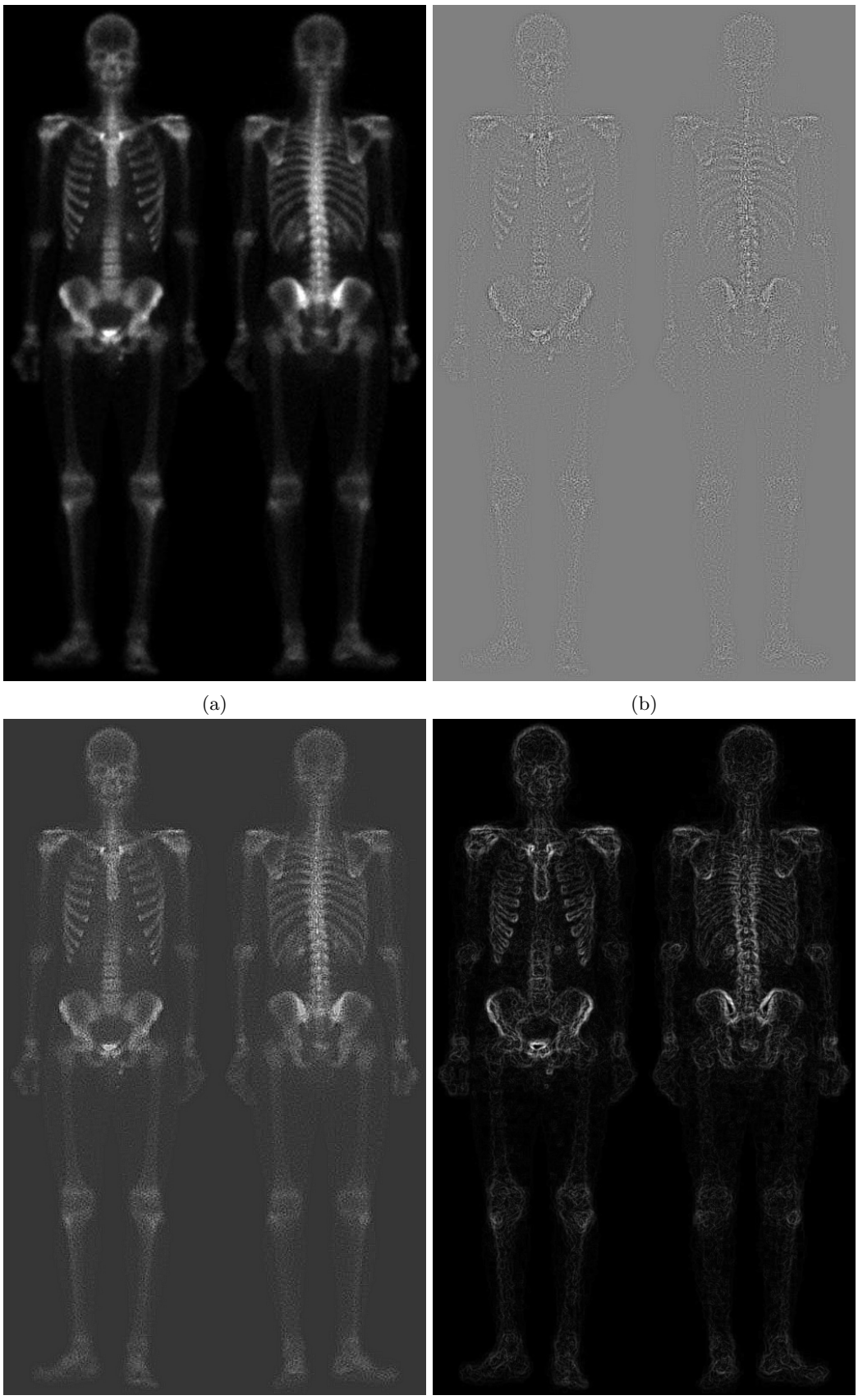
## 2.4 Discussion

We finally get a satisfy result in the Fig.2.4h which shows significant new visible features. There are two tricks we should pay attention to. First, the operations may produce intensity levels out of the range $[0, 255]$. In this kind of cases, a direct matlab command *imshow()* will cause problem. So we have to use scale function. While coding, I made a mistake that I didnt take the numerical type into consideration. I gain an experience that before computation, transform the matrix type to double at first.

## 2.5 Implementation

Here I paste the key parts of my matlab implementation.

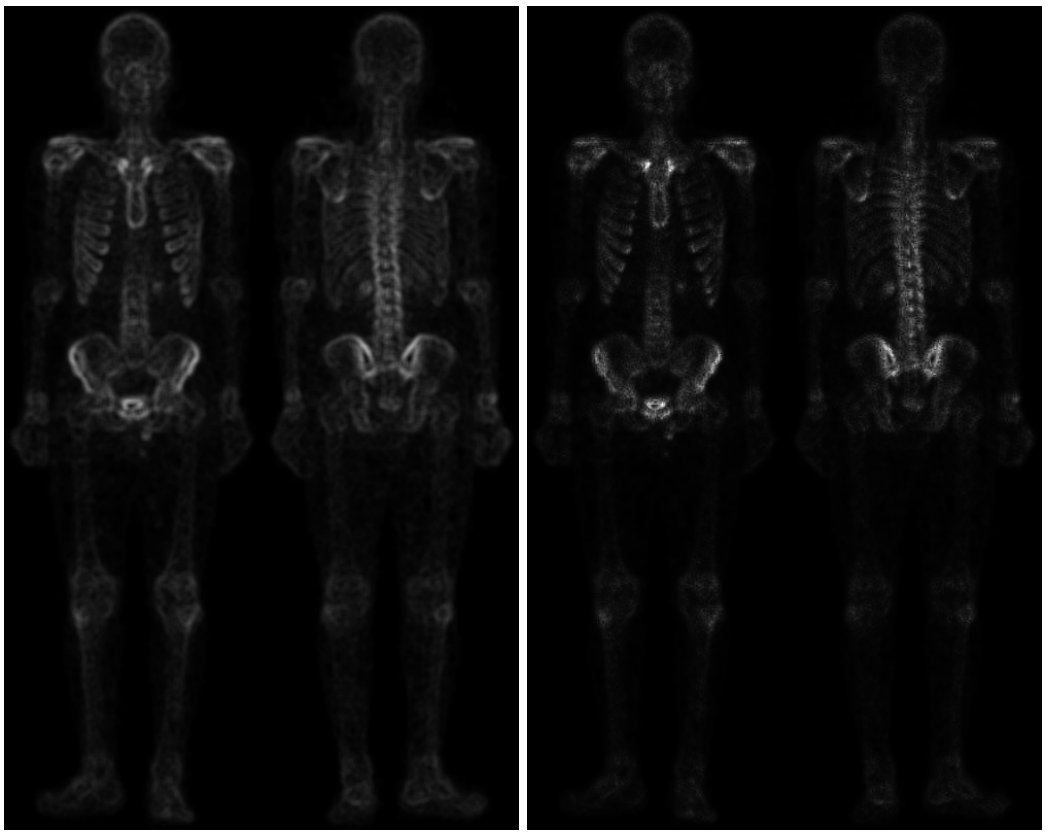
```
% key part of function erosion
for i = (1:M-m+1)
    for j = (1:N-n+1)
        x = imgf(i:i+m-1, j:j+n-1);
        if sum(sum(x.*B)) == m*n
            g(i+downshift, j+rightshift) = 1;
        end
    end
end

% key part of dilation_reconstruction
k_times = 0;
while( ~isequal(f0, f1) )
    k_times = k_times + 1;
    f0 = f1;
    f1 = geodesic_dilation(f1, G, B);
end

% key part of geodesic_dilation
imgg = dilation(imgf, B) & G;

% key part of opening_reconstruction
for i=(1:n_size)
    f_erosion = erosion(f_erosion, B);
end
[imgg, k_times] = dilation_reconstruction(f_erosion, imgf, ones(3,3)); % here
    can not use ones(51, 1)
```



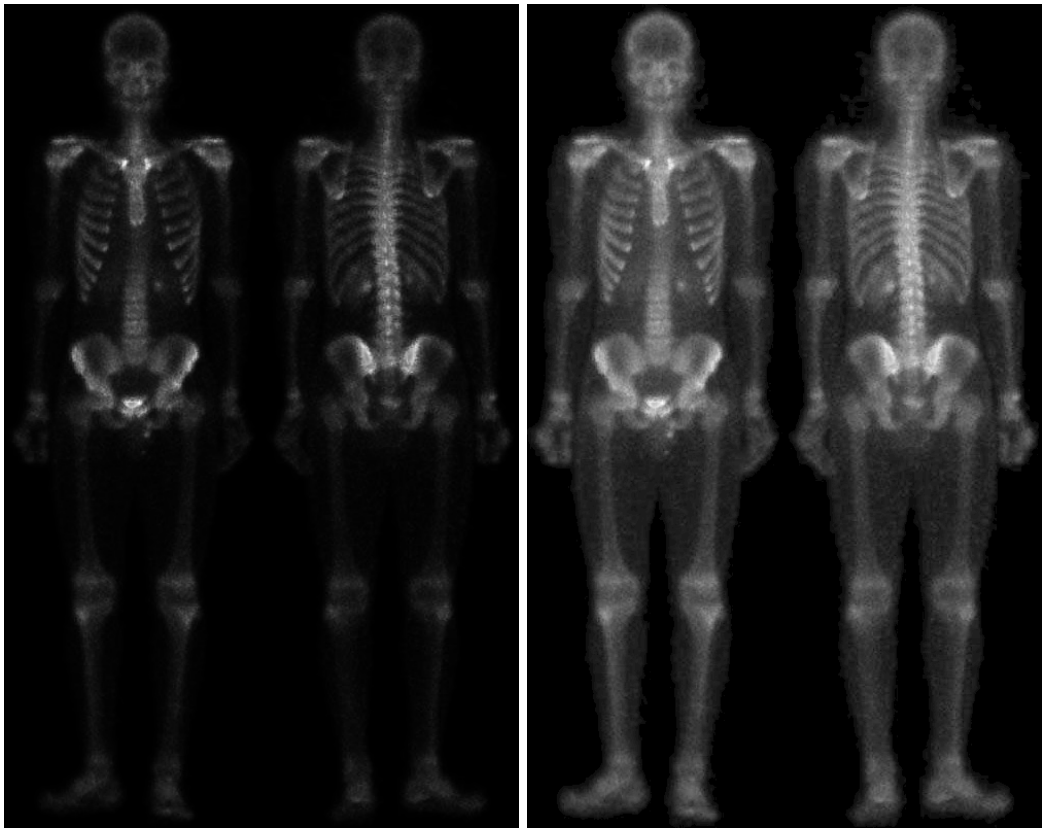
(a) (b)

(c) (d)

Figure 2.3: **a**Original image of whole body bone scan. **(b)**Laplacian of (a)(Rescaled to [0,255]). **(c)**Sharpened image obtained by add (a) and (c). **(d)**Sobel gradient of (a).



(e) (f)



(g) (h)

Figure 2.4: **(e)**Sobel image smoothed with $5 \times 5$ averaging filter. **(f)**Mask image formed by the product of (c) and (e). **(g)**Sharpened image obtained by applying a power-law transformation to (g). **(h)**Compare (g) and (h).

# 3 Project 3 - Filtering in Frequency Domain

## 3.1 Project Proposal

Implement the ideal, Butterworth and Gaussian lowpass and highpass filters and the results under different parameters using the image character_test_pattern.tif

## 3.2 Preliminaries

### 3.2.1 Summary of Fourier transform properties

Table 3.1: Summary of useful formulas.

| Name | Expression(s) |
|---|---|
| 1D FT | $F(u) = \int_{-\infty}^{\infty} f(t)e^{-j2\pi ut}dt$ (3.1) |
| 1D IFT | $f(t) = \int_{-\infty}^{\infty} F(u)e^{j2\pi ut}du$ (3.2) |
| 1D DFT | $F(u) = \sum_{t=0}^{M-1} f(t)e^{-j2\pi ut/M}$ (3.3) |
| 1D IDFT | $f(t) = \frac{1}{M}\sum_{u=0}^{M-1} F(u)e^{j2\pi ut/M}$ (3.4) |
| 2D FT | $F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-j2\pi(ux+vy)}dxdy$ (3.5) |
| 2D DFT | $F(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-j2\pi(ux/M+vy/N)}$ (3.6) |
| 2D IDFT | $f(x,y) = \frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1]} F(u,v)e^{j2\pi(ux/M+vy/N)}$ (3.7) |
| Power spectrum | $P(u,v) = \lvert F(u,v)\rvert^2$ (3.8) |

### 3.2.2 Steps for filtering in the frequency domain

1. Given an input image $f(x,y)$ of size $M \times N$, obtain the padding parameters $P = 2M$ and $Q = 2N$.Form a padded image, $f_p(x,y)$, of size $P \times Q$ by appending zeros.
2. Multiply $f_p(x,y)$ by $(-1)^{x+y}$ to center its transform.
3. Compute the DFT, $F(u,v)$ of the image from centered padded image.
4. Generate a real, symmetric filter function $H(u,v)$ of size $P \times Q$ with center at coordinates $(P/2, Q/2)$. Form the product $G(u,v) = H(u,v)F(u,v)$.
5. Obtain the precessed image: $g_p(x,y) = \left\{\text{real}\left[\ \mathcal{F}^{-1}[G(u,v)]\right]\right\}(-1)^{x+y}$ where the real part is selected in order to ignore parasitic complex components resulting from computational inaccuracies.
6. Obtain the final processed result, $g(x,y)$, by extracting the top left $M \times N$ quadrant of $g_p(x,y)$

## 3.3 Frequency Domain Filters

The filters used here all involving $D(u,v)$, which is the distance between $(u,v)$ in frequency domain and the center of the frequency rectangle

$$D(u,v) = \left[(u - P/2)^2 + (v - Q/2)^2\right]^{1/2} \tag{3.9}$$

. The 'pass' disappears in word 'lowpass' and 'highpass' means we set a range for the things reserved. The range is expressed as a value $D_0$ of $D(u,v)$. The percentage of power enclosed by the circle of radius $D_0$ with origin at the center of the frequency rectangle is

$$\alpha = 100\left[\sum_u \sum_v P(u,v)/P_T\right] \quad \text{(u,v) is inside the circle} \tag{3.10}$$

The three kind of frequency domain filters used in this project are *ideal filter*, *Butterworth filter* and *Gaussian filter*. Their lowpass version and highpass version are in Table.3.2.

Table 3.2: Summary of 3 filters.

| Name | Expression(s) |
|---|---|
| Ideal lowpass filter (ILPF) | $H(u,v) = \begin{cases} 1 & \text{if } D(u,v) \le D_0 \\ 0 & \text{otherwise} \end{cases}$ (3.11) |
| Butterworth lowpass filter (BLPF) | $H(u,v) = \frac{1}{1 + [D(u,v)/D_0]^{2n}}$ (3.12) |
| Gaussian lowpass filter (GLPF) | $H(u,v) = e^{-D^2(u,v)/2D_0^2}$ (3.13) |
| Ideal highpass filter (IHPF) | $H(u,v) = \begin{cases} 0 & \text{if } D(u,v) \le D_0 \\ 1 & \text{otherwise} \end{cases}$ (3.14) |
| Butterworth highpass filter (BHPF) | $H(u,v) = \frac{1}{1 + [D_0/D(u,v)]^{2n}}$ (3.15) |
| Gaussian highpass filter (GHPF) | $H(u,v) = 1 - e^{-D^2(u,v)/2D_0^2}$ (3.16) |

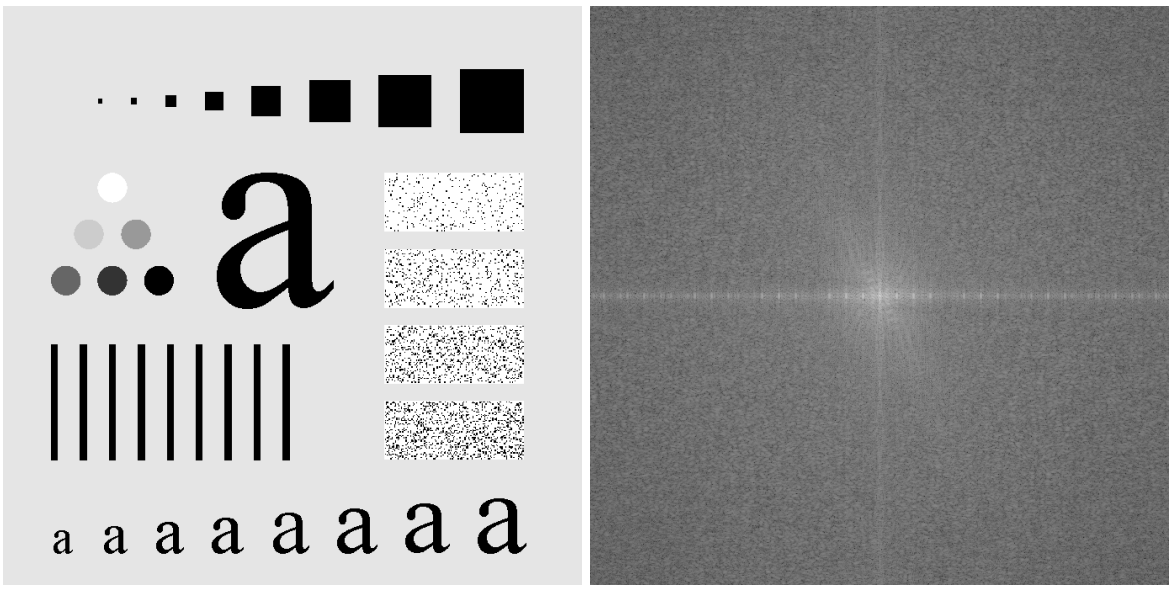
## 3.4 Experiment

The original test image of size $688 \times 688$ pixels and its non-padded centered Fourier spectrum are in Figure.3.1.

### 3.4.1 Image smoothing by lowpass filtering

Edges and other sharp intensity transitions (such as noise) in an image contribute significantly to the high frequency content of its Fourier transform. Hence, smoothing (blurring) is achieved in the frequency domain by high-frequency attenuation; that is, by lowpass filtering.

I use 4 values of $D_0$ (10, 100, 1000, 4000) as the parameter of lowpass filters, generate the smoothed images and calculate power ratio of the filtering. The result is displayed in Figure.3.2, Figure.3.3 and Figure.3.4. In Butterworth filtering, I use $n = 2$ as the order.



(a) (b)

Figure 3.1: (3.1a)Original test image of size 688 × 688 pixels. (3.1b)Non-padded centered Fourier spectrum of the original test image.

### 3.4.2 Image sharpening by highpass filtering

Because edges and other abrupt changes in intensities are associated with high-frequency components, image sharpening can be achieved in the frequency domain by highpass filtering, which attenuates the low-frequency components without disturbing high-frequency information in the Fourier transform.

I use 3 values of $D_0$ (10, 100, 1000) as the parameter of highpass filters, generate the sharpened images and calculate power ratio of the filtering. The result is displayed in Figure.3.2, Figure.3.3 and Figure.3.4.

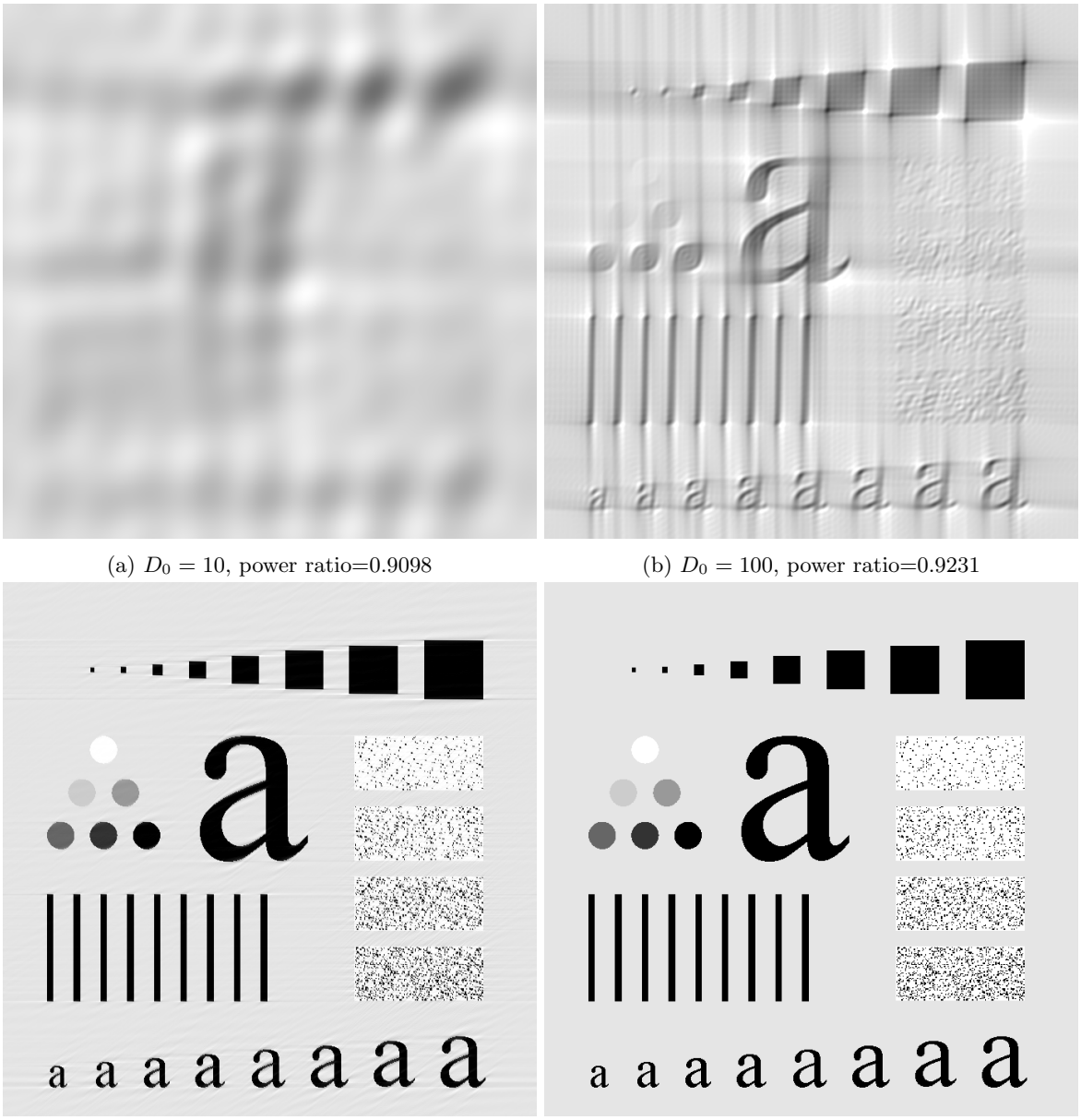
## 3.5 Discussion

In lowpass filter smoothing, the larger the $D_0$ is the less blurring there is in the images. In ideal lowpass filtering, we can clearly see ringing in Figure.3.2a and Figure.3.2b. This ringing is a characteristic of ideal lowpass. The Fourier transform of a vertical falling is the combination of infinite number of triangle functions. In Butterworth filtering with order $n = 2$, there are still mild ringing but much less than those in the ideal filtering. I also notice a little vertical texture in the left part of the result images. In Gaussian lowpass filtering, we hardly see ringing and this is an important benefit in medical image processing because any artifact is unacceptable. However, Gaussian lowpass filtering provide less power ratio than the Butterworth with the same $D_0$.

In highpass filter sharpening, the larger the $D_0$ is the darker the result image is. This is because that the remain high frequency component is less when $D_0$ gets large. Thus the highpass result can help us to understand what the so-called high frequency component is in an image. It's not simple to find out differences between Butterworth highpass and Gaussian highpass. However, we can easily see that these two are much sharpen than the result of ideal highpass filtering which is influenced by ringing again.

## 3.6 Implementation

The key part of the implementation is to correctly follow the steps of frequency domain filtering. The steps are encapsulated in function *dft_filter_func* in Code.**??**. Another thing to talk is how to generate the filter which is implemented in *gen_filter* in Code.**??**.



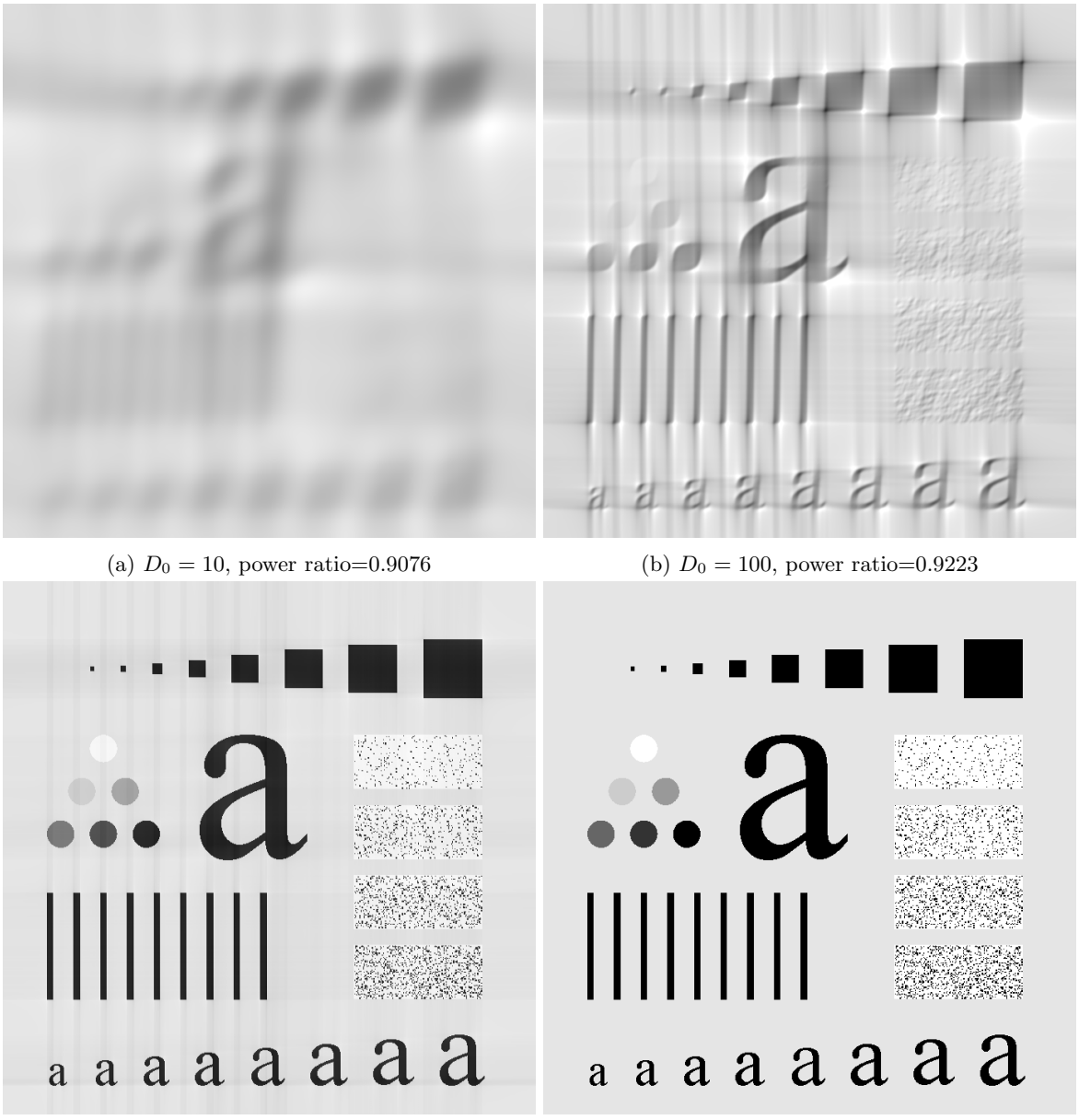
(a) $D_0 = 10$, power ratio=0.9098

(b) $D_0 = 100$, power ratio=0.9231

(c) $D_0 = 1000$, power ratio=0.9971

(d) $D_0 = 4000$, power ratio=1.0000

Figure 3.2: Results of ideal lowpass filtering.



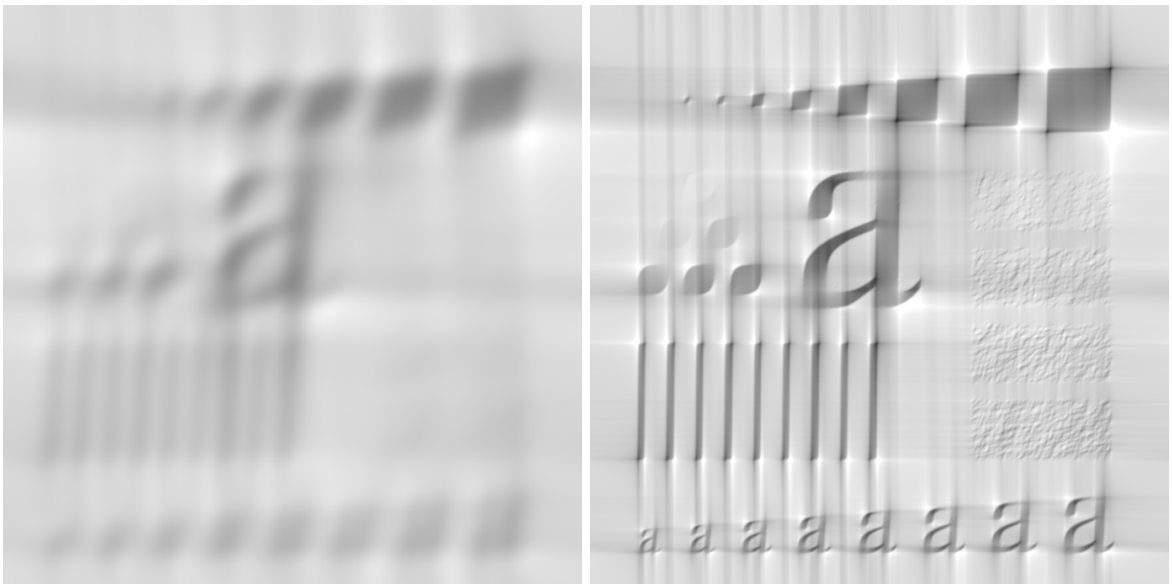
(a) $D_0 = 10$, power ratio=0.9076

(b) $D_0 = 100$, power ratio=0.9223

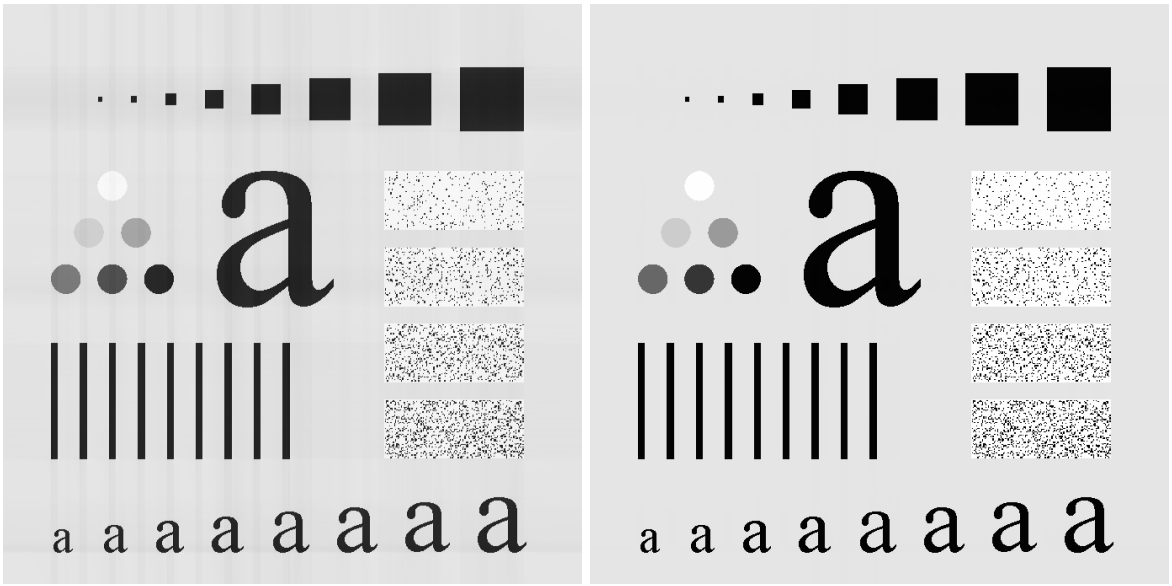
(c) $D_0 = 1000$, power ratio=0.9673

(d) $D_0 = 4000$, power ratio=0.9997

Figure 3.3: Results of Butterworth lowpass filtering.

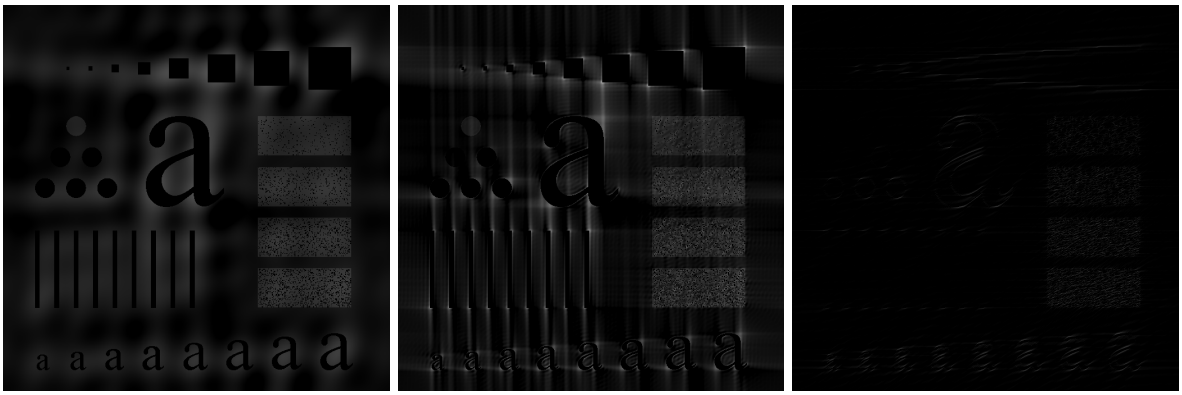


(a) $D_0 = 10$, power ratio=0.0.9075 (b) $D_0 = 100$, power ratio=0.9219



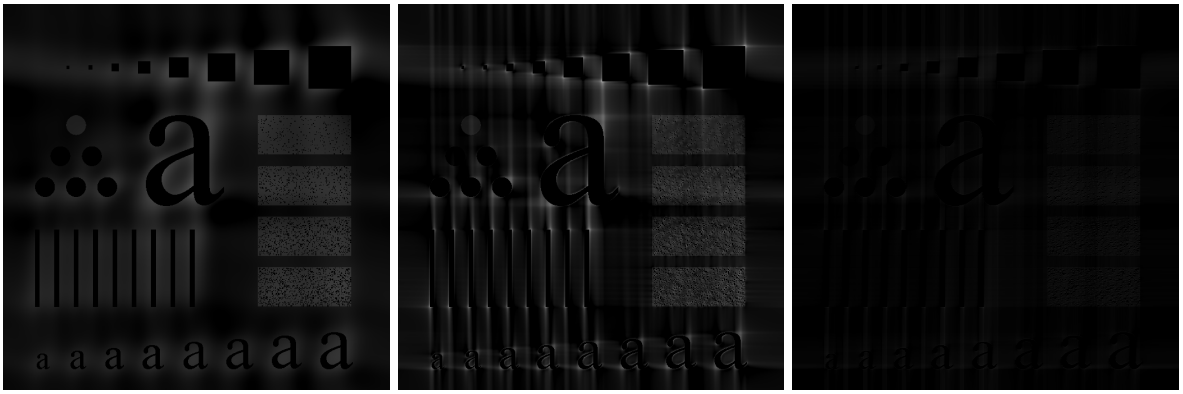
(c) $D_0 = 1000$, power ratio=0.9673 (d) $D_0 = 4000$, power ratio=0.9972

Figure 3.4: Results of Gaussian lowpass filtering.



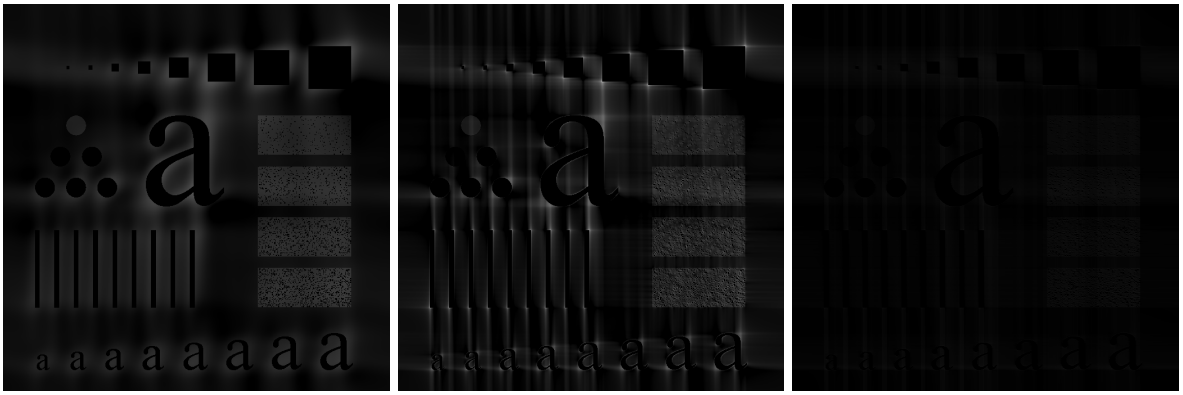
(a) $D_0 = 10$ power ratio=0.0901 (b) $D_0 = 100$ power ratio=0.0768 (c) $D_0 = 1000$ power ratio=0.0028

Figure 3.5: Results of ideal highpass filtering.



(a) $D_0 = 10$ power ratio=0.0885 (b) $D_0 = 100$ power ratio=0.0759 (c) $D_0 = 1000$ power ratio=0.0069

Figure 3.6: Results of Butterworth highpass filtering.



(a) $D_0 = 10$ power ratio=0.0.0873 (b) $D_0 = 100$ power ratio=0.0755 (c) $D_0 = 1000$ power ratio=0.0054

Figure 3.7: Results of Gaussian highpass filtering.

```
function [ imgg, energy_ratio ] = dft_filter_func( imgf, H )
%DFT_FILTER_FUNC
%   DFT_FILTER_FUNC accept an image imgf, and a filter H
%   output the filtered image imgg and enegy_ratio

% automatically pad imgf: pad_f
imgf = double(imgf);
% fast fourier transform
F = fft2(imgf);
% filtered by H
G = H .* F;
% inverse fft and obtain the real part
imgg = real(ifft2(G));
% cut out the original size
imgg = uint8(imgg);
% calculate energy ratio
energy_f = sum(sum( abs(F).^2 ));
energy_g = sum(sum( abs(G).^2 ));
energy_ratio = energy_g / energy_f;

end
```

```
function [ U, V ] = gen_filter( P, Q )
%GEN_FILTER generate filter

u = (0:P-1);
v = (0:Q-1);
idx = find(u > P/2);
u(idx) = u(idx) - P;
idy = find(v > Q/2);
u(idy) = u(idy) - Q;
[V, U] = meshgrid(v, u);

end
```

# 4 Project 4 - Image Restoration

## 4.1 Project Proposal

First implement the random noise generation functions. The random noise functions include uniform, Gaussian, Rayleigh, Gamma, exponential and impulse. Requirement is that only uniform noise is allowed to use built-in function while any other noise should be generated based on uniform noise with self-defined functions. The test image is *Fig0503.tif*. Second follow the process shown in textbook *FIGURE 5.7, FIGURE 5.8, FIGURE5.9, FIGURE 5.10, FIGURE 5.11, FIGURE 5.12*. Each figure displays a process of add some specified type of noise and then use restoration method to reduce noise.

## 4.2 Preliminaries

### 4.2.1 Noise Models

The principal source of noise in digital images arise during image acquisition and/or transmission. We assume in this project that noise is independent of spatial coordinates and uncorrelated with respect to the image itself. Thus, what we concerned is the statistical behavior of the intensity value in the noise component which can be described by PDF.

**Uniform distribution**

$$p(z) = \begin{cases} \frac{1}{b-a} & a \le z \le b \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

$$E(z) = \frac{b-a}{2}, \quad Var(z) = \frac{(b-a)^2}{12} \tag{4.2}$$

In the process of generating other distribution, we usually use $a = 0$ and $b = 1$.

**Gaussian distribution**

$$p(z) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(z-\mu)^2/2\sigma^2} \tag{4.3}$$

$$E(z) = \mu, \quad Var(z) = \sigma \tag{4.4}$$

**Rayleigh distribution**

$$p(z) = \frac{z-a}{\sigma^2} e^{-(z-a)^2/(2\sigma^2)}, z \ge a \tag{4.5}$$

$$E(z) = \sigma\sqrt{\pi/2}, \quad Var(z) = \frac{4-\pi}{2}\sigma^2 \tag{4.6}$$

**Erlang(Gamma) distribution**

$$p(z) = \frac{\lambda x^{k-1} e^{-\lambda x}}{(k-1)!} \text{ for } x, \lambda \ge 0 \tag{4.7}$$

$$E(z) = k/\lambda, \quad Var(z) = k/\lambda^2 \tag{4.8}$$

**Exponential distribution**

$$p(z) = \lambda e^{-\lambda z}, z \ge 0 \tag{4.9}$$

$$E(z) = 1/\lambda, \quad Var(z) = 1/\lambda^2 \tag{4.10}$$

**Salt-and-pepper noise**

$$p(z) = \begin{cases} P_a & z = a \\ P_b & z = b \\ 0 & textotherwise \end{cases} \tag{4.11}$$

### 4.2.2 Generation of Noise Distribution

We can view the random variable from another view that does not involved PDF directly by generating random variable from uniform distribution. The idea used here is similar to the one appear in histogram equalization. We generate random variable $s$ that follow uniform distribution and use a transformation $r = T(s)$ to generate variable $r$ that follow desired distribution. Sometimes, we need more than one variables from uniform distribution to interactive in order to generate desired distribution. Another thing to note is that we do not generate the desired distribution with the exact the same parameters but first to obtain the basic form with the zero mean and unit variance, and then transform the variable by multiplying the square root of variance and plussing the mean.

**Generation of Gaussian distribution** $N(\mu, \sigma)$

First obtain $x_1 \sim U[0,1]$ and $x_1 \sim U[0,1]$. Second, $z = \sqrt{-2\ln(x_1)}\cos(2\pi x_2)$ and we can prove that $z \sim N(0,1)$. Third, $z \leftarrow \sigma z + \mu$.

**Rayleigh distribution** $R(a, \sigma)$
First obtain $x \sim U(0,1)$. Second $z = \sqrt{-2\ln(1-x)}$, that $z \sim R(0,1)$. Third, $z \leftarrow \sigma z + a$.
**Erlang(Gamma) distribution** $G(k, \lambda)$
First, obtain $k$ variables $x_1, x_2, ..., x_k$ that $x_i \sim U(0,1)$. Second, $z \approx -\frac{1}{\lambda} \ln \prod_{i=1}^{k} x_i$ that $z \sim G(k, \lambda)$.
**Exponential distribution** $Exp(\lambda)$
First, obtain $x \sim U(0,1)$. Second, $z = -\ln(x)/\lambda$ that $z \sim Exp(\lambda)$.
**Salt-and-pepper noise**
First, obtain $x \sim U(0,1)$. Second, if $0 \leq x \leq P_a$, $z = a$, if $P_a < x \leq P_a + P_b$, $z = b$.

### 4.2.3 Restoratoin in the Presence of Noise Only-Spatial Filtering

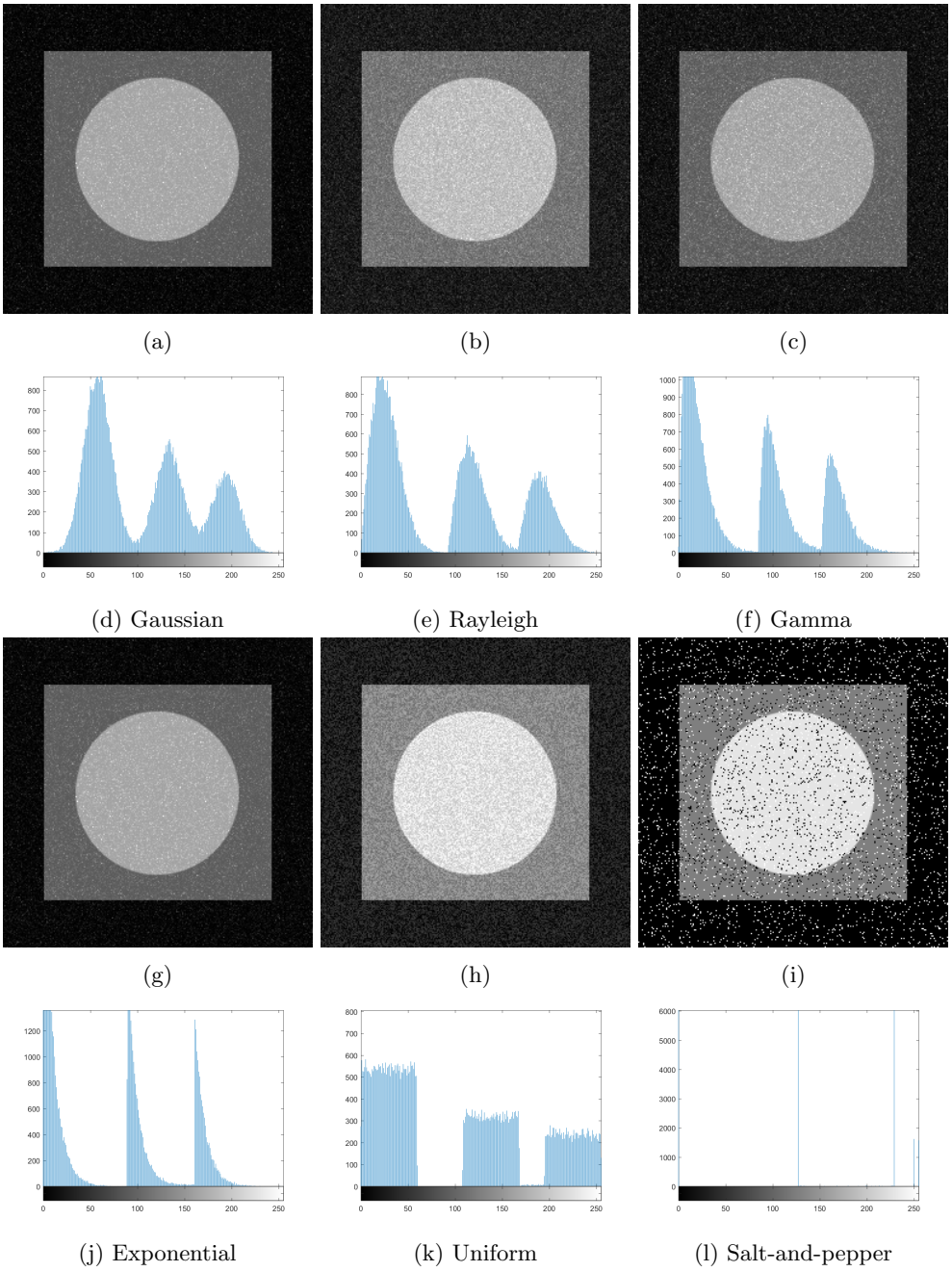
Spatial filtering is the method of choice in situations when only additive random noise is present. The filters used in this project is listed in Tabel.4.1

Table 4.1: Summary of noise-reduction spatial filters.

| Filter Name | Expression(s) | Characteristic |
|---|---|---|
| Arithmetic mean filters | $\hat{f}(x,y) = \frac{1}{mn} \sum_{(s,t)\in S_{xy}} g(s,t)$ (4.12) | Cause more blurring than geometric mean filters when reduce Gaussian noise. |
| Geometric mean filters | $\hat{f}(x,y) = [\prod_{(s,t)\in S_{xy}} g(s,t)]^{1/mn}$ (4.13) | Cause less blurring than arithmetic mean filters when reduce Gaussian noise. |
| Harmonic mean filter | $\hat{f}(x,y) = mn / \sum_{(s,t)\in S_{xy}} \frac{1}{g(s,t)}$ (4.14) | Work well with salt noise and Gaussian noise, but fails for pepper noise. |
| Contraharmonic mean filter | $\hat{f}(x,y) = \frac{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q+1}}{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q}}$ (4.15) | $Q$ is the order of filter. Negative $Q$ for salt noise and positive $Q$ for pepper noise. |
| Median filter | $\hat{f}(x,y) = \text{median}_{(s,t)\in S_{xy}} g(s,t)$ (4.16) | Good for salt-pepper-noise and causes less blurring than linear filters |
| Max filter | $\hat{f}(x,y) = \max_{(s,t)\in S_{xy}} g(s,t)$ (4.17) | Good for pepper noise reduction. |
| Max filter | $\hat{f}(x,y) = \min_{(s,t)\in S_{xy}} g(s,t)$ (4.18) | Good for salt noise reduction. |
| Midpoint filter | $\hat{f}(x,y) = \frac{1}{2}\left(\max_{(s,t)\in S_{xy}} g(s,t) + \min_{(s,t)\in S_{xy}} g(s,t)\right)$ (4.19) | Works best for random noise, like Gaussian and uniform noise. |
| Alpha-trimmed mean filter | $\hat{f}(x,y) = \frac{1}{mn-d} \sum_{(s,t)\in S_{xy}} g_r(s,t)$ (4.20) | Delete the $d/2$ lowest and $d/2$ highest intensity values of $g(s,t)$. $g_r(s,t)$ represents the remains pixels. Choose different $d$ can fit different situation. |

## 4.3 Experiment-1 Generate different noise

This is the first part of experiment. The target of parameter choosing is to make the obtained histogram be visually similar to the histogram displayed on the book Fig5.4. In each histogram, we can clearly see the special shape of the specified distribution. This phenomenon shows that the distribution generation algorithm is correct.



(a) (b) (c)

(d) Gaussian (e) Rayleigh (f) Gamma

(g) (h) (i)

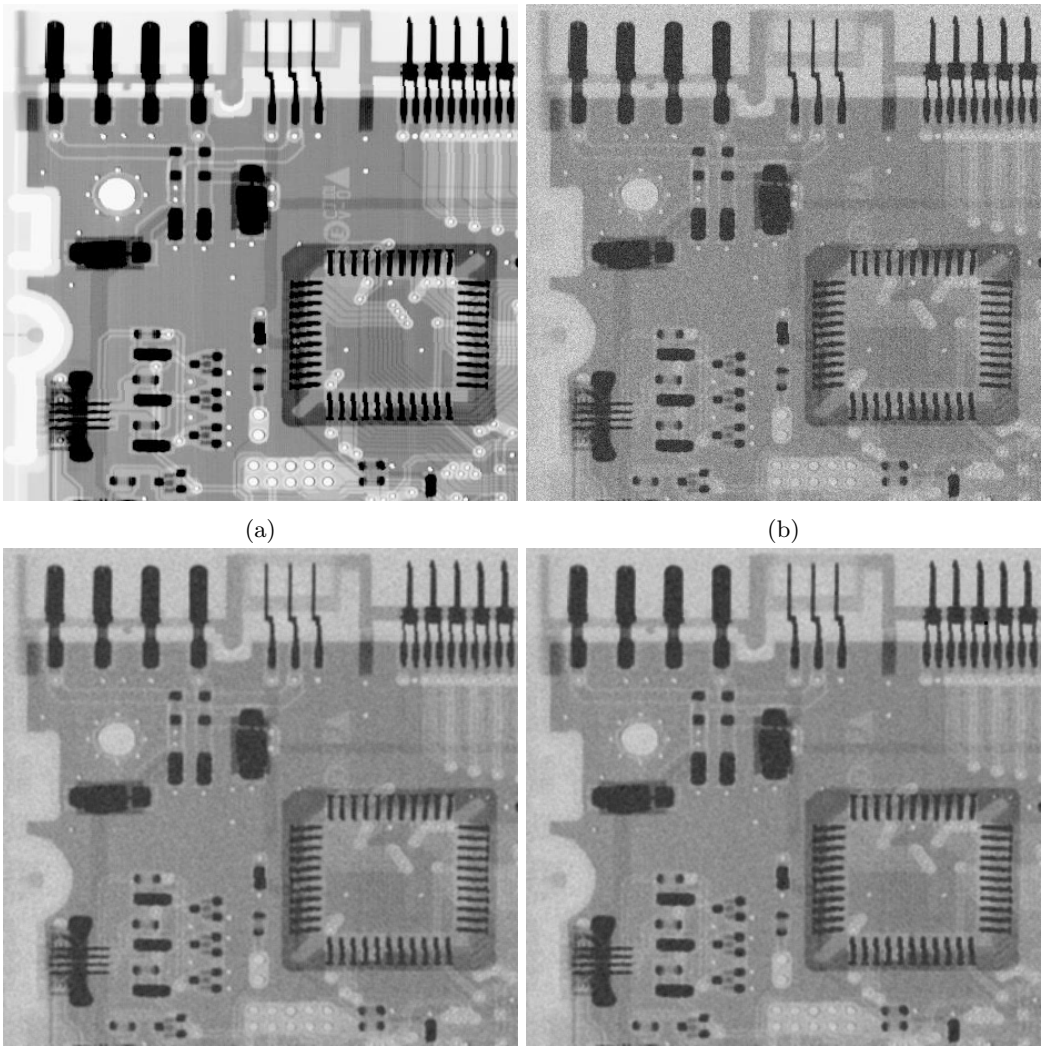
(j) Exponential (k) Uniform (l) Salt-and-pepper

Figure 4.1: (3.1a)Original test image of size $688 \times 688$ pixels. (3.1b)Non-padded centered Fourier spectrum of the original test image.

## 4.4 Experiment-2 Noise reduction

### 4.4.1 Follow fig5.7 in textbook

The work in this part is shown in Figure.4.2. On the original X-ray image in Figure.4.2a of circuit board, we add Gaussian noise of $N(0, 400)$. The corrupted image is shown in Figure.4.2b. In Figure.4.2c, we use $3 \times 3$ arithmetic mean filter to reduce the noise. In Figure.4.2d is the result of geometric mean filtering. We can see that both filters did a reasonable job of attenuating noise. Moreover, geometric mean filter performs better on Gaussian noise as it causes less blurring than arithmetic mean filter do.



(a) (b)

(c) (d)

Figure 4.2: (4.2a)Original image. (4.2b)Add Gaussian noise $N(0, 400)$. (4.2c)Result of filtering with an arithmetic mean filter of size $3 \times 3$. (4.2d)Result of filtering with an geometric mean filter of size $3 \times 3$

### 4.4.2 Follow fig5.8 in textbook

The work in this part is shown in Figure.4.3. We add pepper noise and salt noise separately in Figure.4.3a and Figure.4.3b relatively. Then we use two $3 \times 3$ contraharmonic mean filters with $Q = 1.5$ and $Q = -1.5$ separately to pepper and salt noise relatively. The result is fine but we have to know the type of the noise, which is an ideal situation.

### 4.4.3 Follow fig.5.9 in textbook

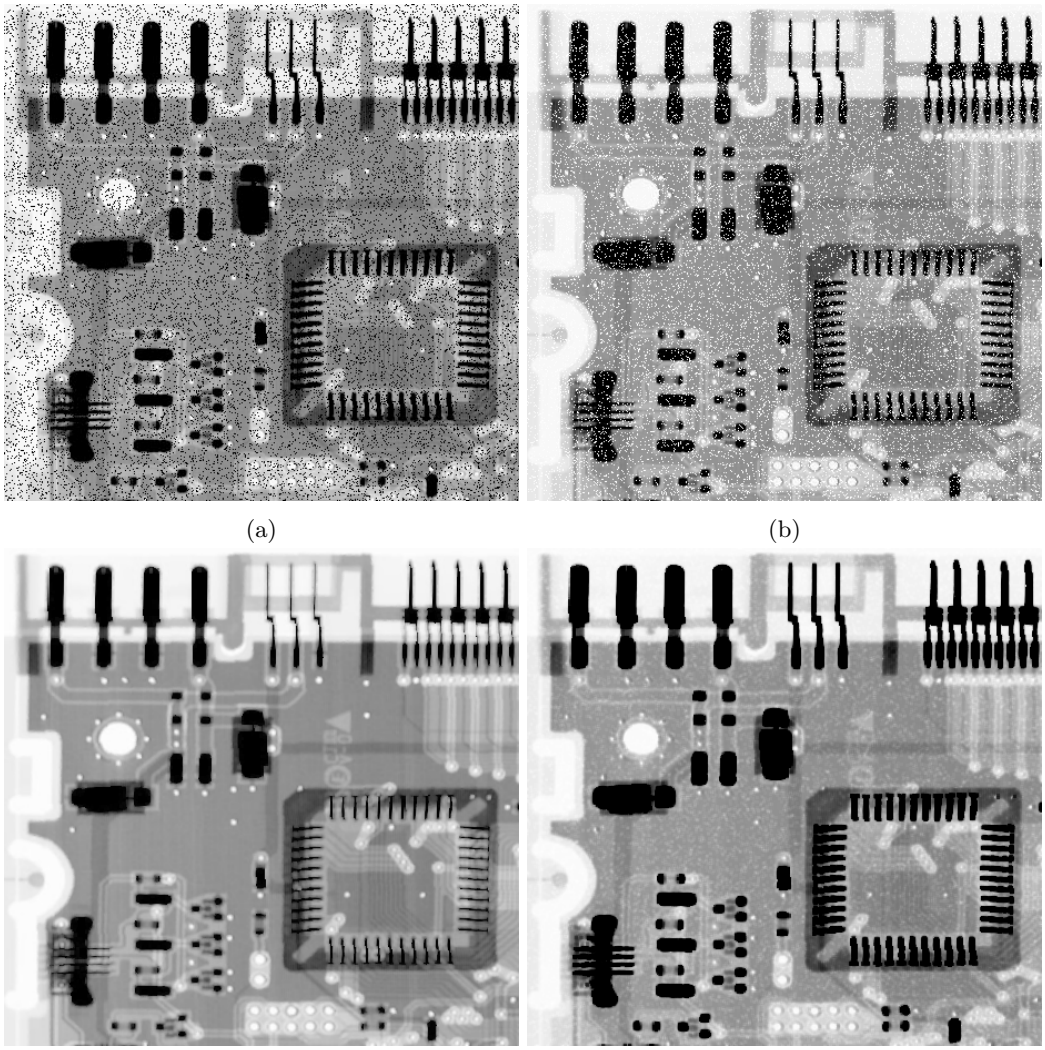
The work in this part shows the wrong using of contraharmonic mean filter. If we use $Q = -1.5$ for pepper noise in Figure.4.3a and $Q = 1.5$ for salt noise in Figure.4.3b, then we will get the bad results in Figure.4.4a and Figure.4.4b respectively.

### 4.4.4 Follow fig.5.10 in textbook

Here we add salt noise and pepper noise with $P_a = P_b = 0.1$ at the same time in Figure.4.5a. The we use $3 \times 3$ median filter to filtering the image for 3 times and the result of each step is showed in Figure.4.5b, Figure.4.5c and Figure.4.5d respectively. We can see that the first time of filtering removes most of the noise and the next two times filtering get even better. The result is excellent as no blurring appears.

### 4.4.5 Follow fig.5.11 in textbook

In last part, we have see the power of order-statistic filter in pepper-and-salt noise reduction. Here, we use max filter for Figure.4.3a and min filter for Figure.4.3b. The result is also excellent because there is no import of



(a) (b)

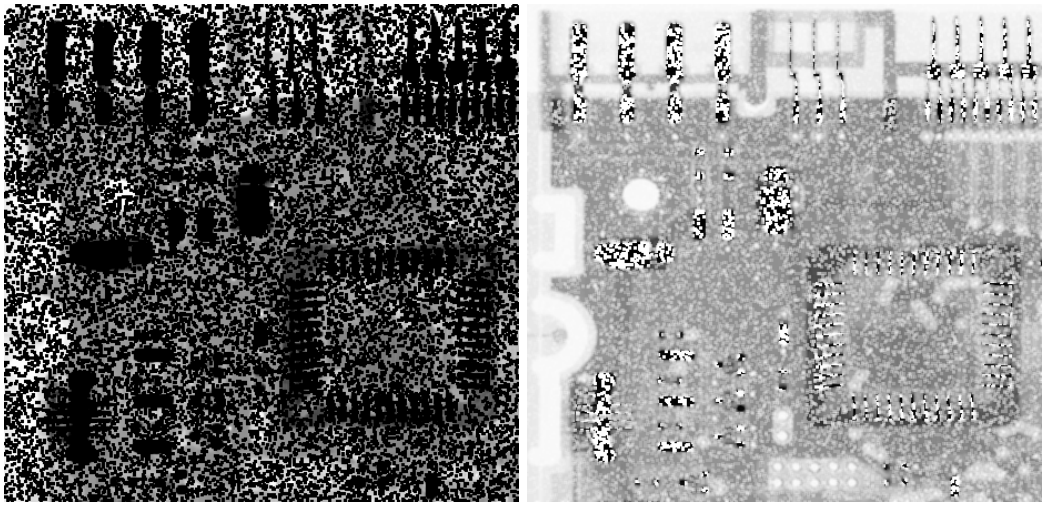
(c) (d)

Figure 4.3: (4.3a)Image corrupted by pepper noise with a probability of 0.1. (4.3b)Image corrupted by salt noise with probability of 0.1. (4.3c)Result of filtering 4.3a with a $3 \times 3$ contraharmonic filter of order $Q = 1.5$ . (4.3d)Result of filtering 4.3b with $Q = -1.5$

blurring as well as thorough reduction of noise.

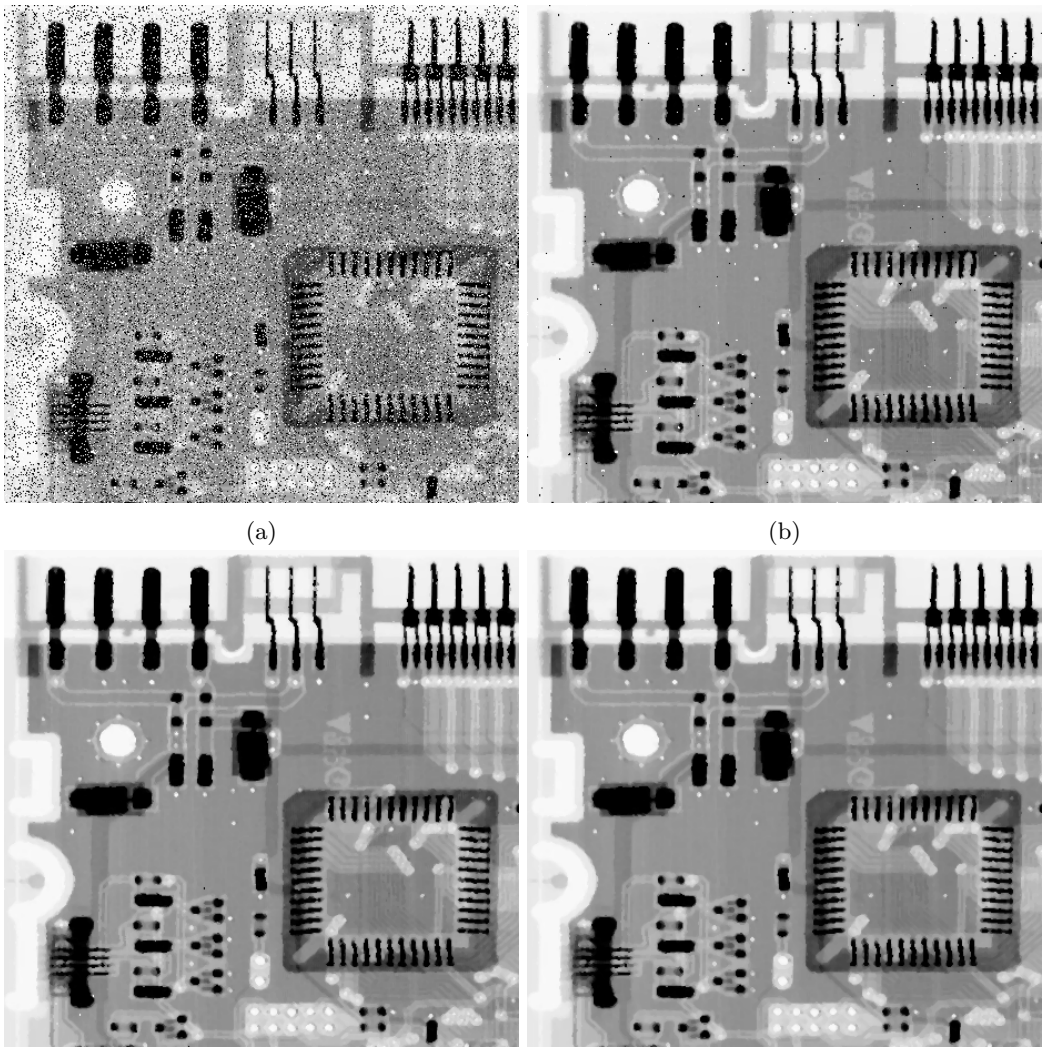
#### 4.4.6 Follow fig.5.12 in textbook

In this part, we add uniform of zero mean and variance 800 and peper-and-salt with noise $P_a = P_b = 0.1$ at the same time in Figure.4.7b. Then we apply 4 kinds of filter of size $5 \times 5$ to reduce noise. In Figure.4.7c,4.7d,4.7e,4.7f, arithmetic mean filter, geometric mean filter, median filter, and alpha-trimmed filter with $d = 5$ are used respectively. The two mean filter do not perform well as there are pepper-and-salt noise and the two order statistic filter do much better. However there are still some blurring due to the presence of uniform noise. The alpha-trimmed filter gets smoother results than median filter because it can remove the polar value which may contains much pepper-and-salt noise.



(a) (b)

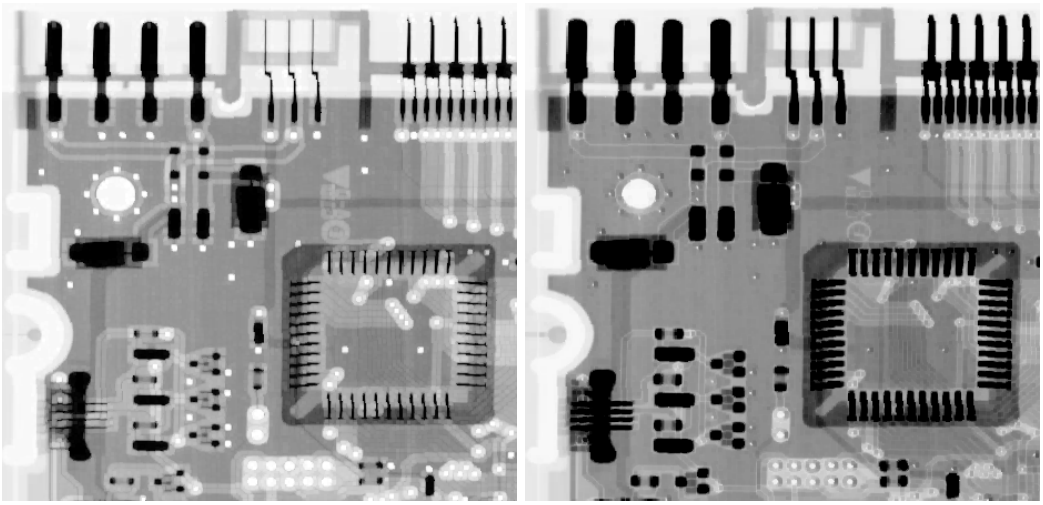
Figure 4.4: (4.4a)Use contraharmonic filter with $Q = -1.5$ for pepper noise in 4.3a. (4.4b) Use contraharmonic filter with $Q = 1.5$ for salt noise in 4.3b.



(a) (b)

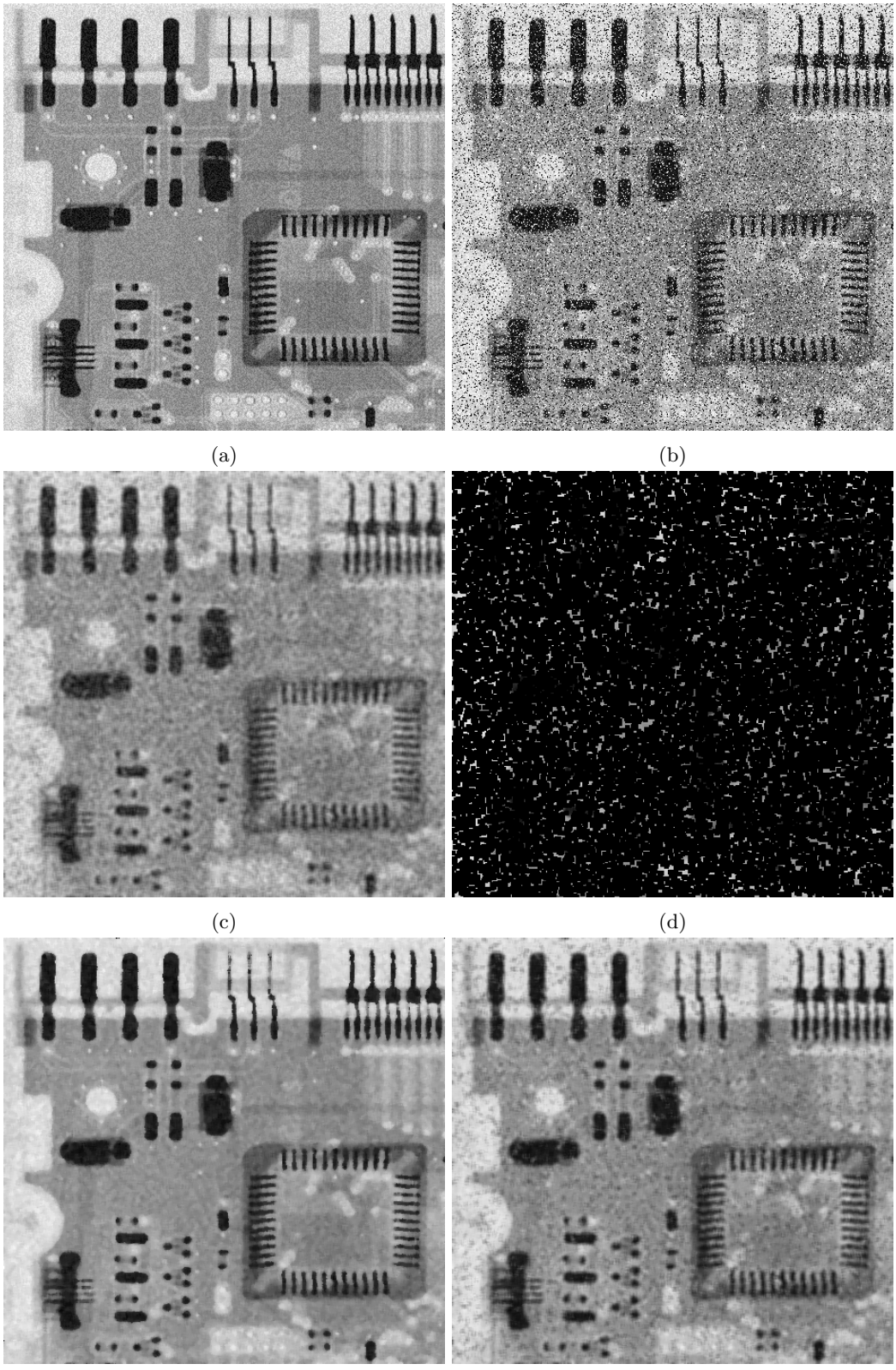
(c) (d)

Figure 4.5: (4.5a)Add pepper noise and salt noise with $P_a = P_b = 0.1$ at the same time to original image. (4.5b)Apply $3 \times 3$ median filter to 4.5a. (4.5c)Apply the same filter to 4.5b. (4.5d)Apply the same filter to 4.5c.



(a) (b)

Figure 4.6: (4.6a)Use max filter on 4.3a (4.6b) Use min filter on 4.3b.



(a) (b) (c) (d) (e) (f)

Figure 4.7: (4.7a)Image corrupted by uniform noise of $U(0, 800)$. (4.7b)Image corrupted by pepper-and-salt noise with $P_a = P_b = 0.1$. (4.7c)Result of $5 \times 5$ arithmetic mean filtering. (4.7d) Result of $5 \times 5$ geometric mean filtering (4.7e)Result of $5 \times 5$ median filtering. (4.7f)Result of $5 \times 5$ alpha-trimmed filtering with $d = 5$.

## 4.5 Implementation

The key points of implementation here are (1)the generation of different distribution from uniform distribution and (2)the implementation of different filters. All of these are listed in the following code piece.

```
function [ n_gaussian ] = gen_gaussian( mu, variance, M, N )
%GEN_GAUSSIAN
%   generate gaussian distribution from uniform distribution
n_gaussian = zeros(M, N);
for i = (1:M)
    for j = (1:N)
        x1 = rand();
        x2 = rand();
        n_gaussian(i,j) = sqrt(-2 * log(x1))*cos(2*pi*x2);
    end
end
n_gaussian = n_gaussian * variance + mu;
end
```

```
function [ n_rayleigh ] = gen_rayleigh( lowbound, variance, M, N )
%GEN_RAYLEIGH
%   generate rayleigh distribution from uniform distribution
n_rayleigh = zeros(M, N);
for i = (1:M)
    for j = (1:N)
        x = rand();
        n_rayleigh(i, j) = sqrt(-2 * variance^2 * log(1-x));
    end
end
n_rayleigh = n_rayleigh + lowbound;
end
```

```
function [ n_gamma ] = gen_gamma( gamma_a, gamma_b, M, N )
%GEN_GAMMA
%   generate gamma distribution from uniform distribution
n_gamma = zeros(M, N);
for i = (1:M)
    for j = (1:N)
        x = rand(1, gamma_b);
        x = log(1-x);
        n_gamma(i,j) = -1/gamma_a * sum(x);
    end
end
end
```

```
function [ n_exp ] = gen_exp( exp_lambda, M, N )
%GEN_EXP
%   generate exponential distribution from uniform distribution
n_exp = zeros(M, N);
for i = (1:M)
    for j = (1:N)
        x = rand();
        n_exp(i, j) = -1.0 / exp_lambda * log(1-x);
    end
end
end
```

```
function [ n_saltpepper ] = gen_saltpepper( salt, pepper, p_salt, p_pepper, M, N
    )
%GEN_SALTPEPPER
%
```

```
n_saltpepper  = zeros(M, N);
for i = (1:M)
    for j = (1:N)
        x = rand();
        if x < p_salt
            n_saltpepper(i, j) = salt;
        else
            if x < p_salt + p_pepper
                n_saltpepper(i, j) = pepper;
            end
        end
    end
end

end
```

```
function [ imgg ] = arithmetic_mean( imgf, p )
%ARITHMIC_MEAN
%
[M, N] = size(imgf);
f = replicate_padding(imgf, p-1);
f = double(f);
g = zeros(M+p-1, N+p-1);
for i = ((p+1)/2:M+p-1+(p-1)/2)
    for j = ((p+1)/2:N+p-1+(p-1)/2)
        g(i, j) = sum(sum(f(i-(p-1)/2:i+(p-1)/2, j-(p-1)/2:j+(p-1)/2))) / p / p;
    end
end
imgg = uint8(g(p:M+p-1, p:N+p-1));
end
```

```
function [ imgg ] = geometric_mean( imgf, p )
%GEOMETRY_MEAN
%
[M, N] = size(imgf);
f = replicate_padding(imgf, p-1);
f = double(f);
g = zeros(M+p-1, N+p-1);
for i = ((p+1)/2:M+p-1+(p-1)/2)
    for j = ((p+1)/2:N+p-1+(p-1)/2)
        g(i, j) = (prod(prod(f(i-(p-1)/2:i+(p-1)/2, j-(p-1)/2:j+(p-1)/2)))) ^
            (1/p/p) ;
    end
end
imgg = uint8(g(p:M+p-1, p:N+p-1));
end
```

```
function [ imgg ] = contra_harmonic( imgf, p, Q )
%CONTRA_HARMONIC
%
[M, N] = size(imgf);
f = replicate_padding(imgf, p-1);
f = double(f);
g = zeros(M+p-1, N+p-1);
for i = ((p+1)/2:M+p-1+(p-1)/2)
    for j = ((p+1)/2:N+p-1+(p-1)/2)
        g(i, j) = sum(sum(f(i-(p-1)/2:i+(p-1)/2, j-(p-1)/2:j+(p-1)/2) .^ (Q+1)))
            / ...
                sum(sum(f(i-(p-1)/2:i+(p-1)/2, j-(p-1)/2:j+(p-1)/2) .^ Q));
    end
```

```
end
imgg = scale255(g(p:M+p-1, p:N+p-1));
imgg = uint8(imgg);
end
```

```
function [ imgg ] = median_spatial( imgf, p )
%MEDIAN
%
[M, N] = size(imgf);
f = replicate_padding(imgf, p-1);
f = double(f);
g = zeros(M+p-1, N+p-1);
for i = ((p+1)/2:M+p-1+(p-1)/2)
    for j = ((p+1)/2:N+p-1+(p-1)/2)
        spatial = f(i-(p-1)/2:i+(p-1)/2, j-(p-1)/2:j+(p-1)/2);
        g(i, j) = median(spatial(:));
    end
end
imgg = uint8(g(p:M+p-1, p:N+p-1));

end
```

```
function [ imgg ] = max_spatial( imgf, p )
%MAX_SPATIAL
%
[M, N] = size(imgf);
f = replicate_padding(imgf, p-1);
f = double(f);
g = zeros(M+p-1, N+p-1);
for i = ((p+1)/2:M+p-1+(p-1)/2)
    for j = ((p+1)/2:N+p-1+(p-1)/2)
        spatial = f(i-(p-1)/2:i+(p-1)/2, j-(p-1)/2:j+(p-1)/2);
        g(i, j) = max(spatial(:));
    end
end
imgg = uint8(g(p:M+p-1, p:N+p-1));
end
```

```
function [ imgg ] = alpha_trimmed( imgf, p, d )
%ALPHA_TRIMMED
%
[M, N] = size(imgf);
f = replicate_padding(imgf, p-1);
f = double(f);
g = zeros(M+p-1, N+p-1);
for i = ((p+1)/2:M+p-1+(p-1)/2)
    for j = ((p+1)/2:N+p-1+(p-1)/2)
        t = f(i-(p-1)/2:i+(p-1)/2, j-(p-1)/2:j+(p-1)/2);
        tt = sort(t(:));
        g(i, j) = sum(tt( fix(d/2)+1 : p*p-(d-fix(d/2)) )) / (p*p - d);
    end
end
imgg = uint8(g(p:M+p-1, p:N+p-1));

end
```

# 5 Project 6 - Geometric Transformation

## 5.1 Project Proposal

Implement 4 geometric transform functions - rotate by a specified angle, translate by a given vector, scale by a given rate using nearest neighbor and scale by a given rate using bilinear interpolation.

## 5.2 Preliminary

### 5.2.1 Spatial Transform

A spatial transformation of an image is a geometric transformation of the image coordinate system. First we define the coordinate system in an image: *axis-X* is from left to right while *axis-Y* is from top to bottom. In a spatial transformation each point $(x, y)$ of origin image $A$ is mapped to point $(u, v) = (f_1(x, y), f_2(x, y))$ in a new coordinate system.
Affine transformation is any transformation that preserve collinearity and ratios of distances. Rotations, translations, magnifications, shears and their combinations are all affine transformations. Any single affine operation can be expressed as

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5.1}$$

We extend the 2D point to 3D in order to use a homogeneous matrix multiplication notation. Then we can represent translation, rotation, scaling under this notation. We denote the $3 \times 3$ transformation matrix as $T$

$$T = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \text{ Translation by } (x_0, y_0) \tag{5.2}$$

$$T = \begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ Scale by } s_1 \text{ and } s_2 \tag{5.3}$$

$$T = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ Rotate by } \theta \text{ (clockwise)} \tag{5.4}$$

$$T = \begin{bmatrix} 1 & \alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ Horizontal shearing by } \alpha \tag{5.5}$$

$$T = \begin{bmatrix} 1 & 0 & 0 \\ \beta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ Vertical shearing by } \beta \tag{5.6}$$

Another advantage of matrix multiplication notation for affine transformation is that we can easily get composite affine transformation

$$T = \prod_i T_i \tag{5.7}$$

as well as the inverse transformation $T^{-1}$

### 5.2.2 Interpolation

After conduct geometric spatial transformation, the new grid points may not locate exactly on grid points in the new coordinate system. Interpolation is needed in this kind situation to find out th value of the grid points in the new coordinate system. Here, we talk about two methods of interpolation - nearest neighbor interpolation and bilinear interpolation.
Nearest neighbor interpolation is the simplest and fastest interpolation method. In practice, we just use the $(x, y, 1)^T = round(T^{-1}(u, v, 1)^T)$ as nearest neighbor interpolation.
In bilinear interpolation, we first calculate $(x', y')$ in $O_{xy}$ which is the original phase of a grid point $(u, v)$ in $O_{uv}$. Then find out the smallest rectangle that surround $(x', y')$ which is formed by 4 grid points $\{(x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)\}$ in $O_xy$. Thus we calculate the value at grid point $(x', y')$ as well as $(u, v)$ in $O_{uv}$ in the following way including 2 level of linear interpolation:

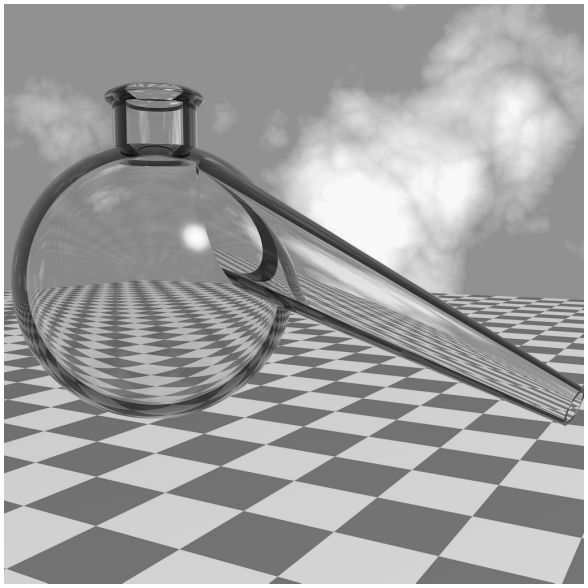
$$g_{0,1}(x', y') = \frac{x_1 - x'}{x_1 - x_0} f(x_0, y_0) + \frac{x' - x_0}{x_1 - x_0} f(x_1, y_1) \tag{5.8}$$

$$g_{3,2}(x', y') = \frac{x_2 - x'}{x_2 - x_3} f(x_3, y_3) + \frac{x' - x_3}{x_2 - x_3} f(x_2, y_2) \tag{5.9}$$

$$g(x', y') = \frac{y_2 - y'}{y_2 - y_1} g_{0,1}(x', y') + \frac{y' - y_1}{y_2 - y_1} g_{3,2}(x', y') \tag{5.10}$$

## 5.3 Experiment

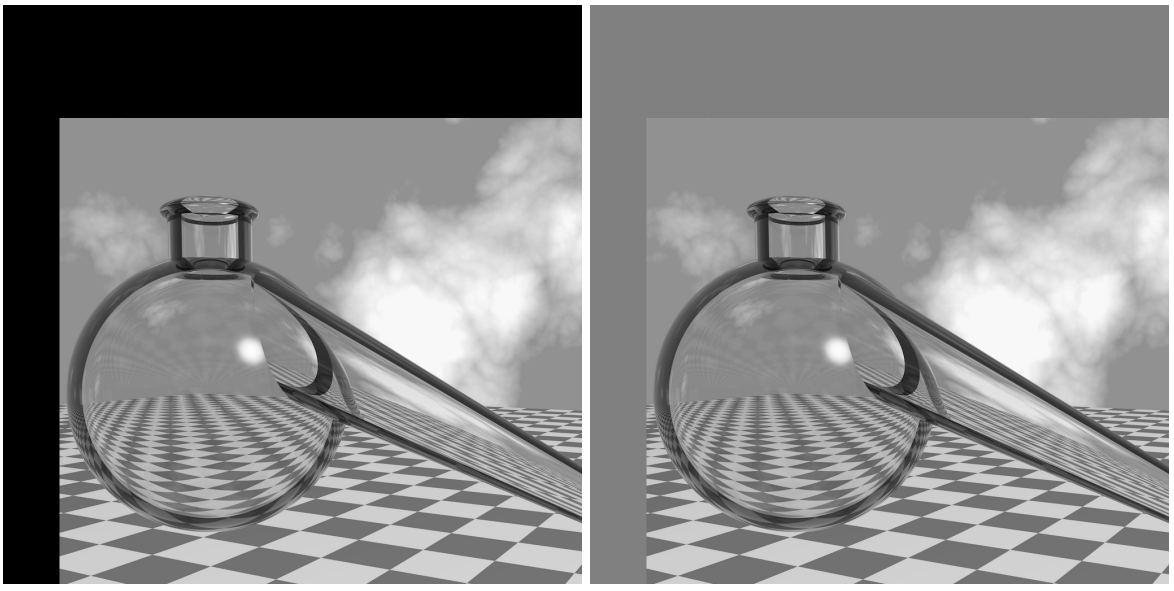
In this experiment we use Fig.5.1 for testing.



(a)

Figure 5.1: Origin test image

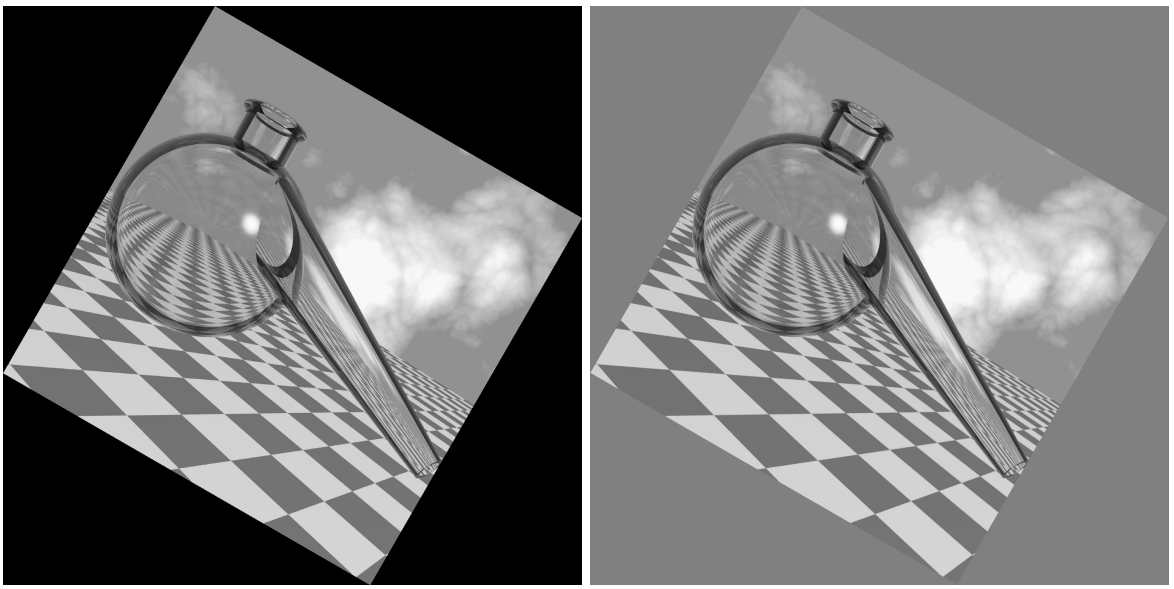
**Translate** The image is translated by 100 pixel at right and by 200 pixel down. The result is display in Fig.5.2.



(a) Use nearest neighbor interpolation. (b) Use bilinear interpolation.

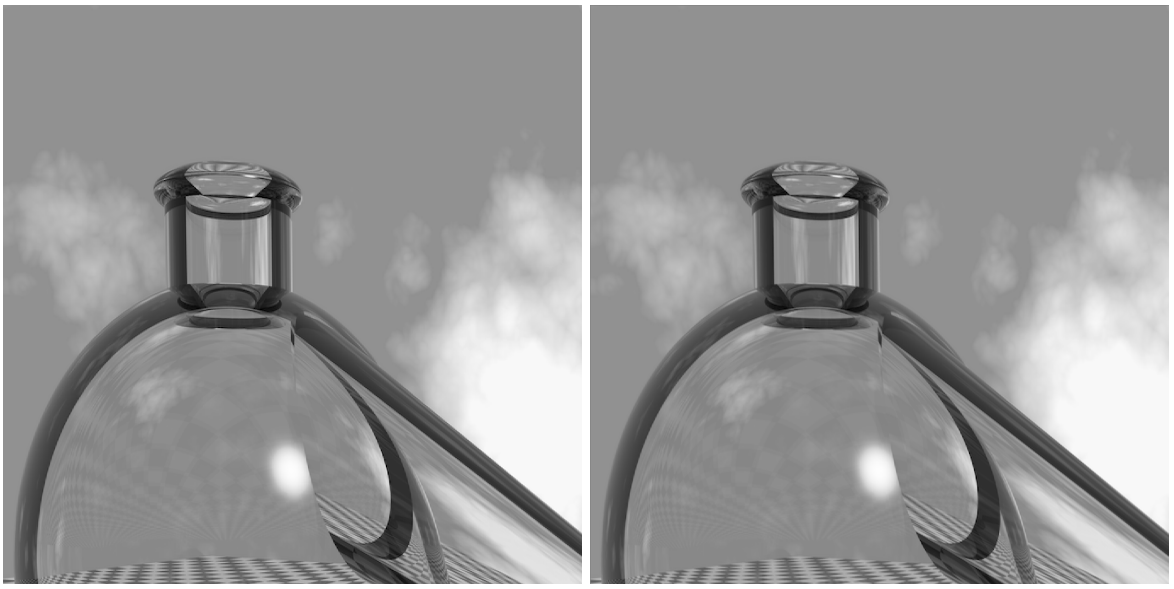
Figure 5.2: Results of translate 100 from left to right and 200 down.

**Rotation** The image is rotated by $\pi/6$ clockwise. The result is display in Fig.5.3.
**Scaling** The image is scaled by 1.5 at axis-x and by 2 at axis-y. The result is display in Fig.5.3.



(a) Use nearest neighbor interpolation. (b) Use bilinear interpolation.

Figure 5.3: Results of rotation. $\pi/6$ clockwise.



(a) Use nearest neighbor interpolation. (b) Use bilinear interpolation.

Figure 5.4: Results of scaling. Times 1.5 at axis-X and times 2 at axis-Y.

## 5.4 Implementation

The main part of this implementation is on the function *mytransform*. The input parameters are three necessary parameters - input image, $3 \times 3$ affine transform matrix, interpolation method and 3 optional parameters - horizontal and vertical display boundaries and filling colors for empty pixels.

The idea is simple: First, find out the output image's size based only on the transform matrix and input image. The approach is to calculate the coordinates for four rectangle vertices. After consider the user defined boundary, we get the size of final output rectangle range. Then use the each grid coordinate $(u, v)$ to calculate the corresponding $(x, y)$ in input. After get $(x, y)$, get the pixel value $g(u, v)$ according to the interpolation methods. To get the little square where $(x, y)$ is in, another function *bilinear_rect* is written.

The code of *mytransform* and *bilinear_rect*is listed below:

```
1 function [ imgg ] = mytransform( imgf, matT, interpolation, varargin )
2 %MYTRANSFORM
```

```
%
ip = inputParser;
ip.addParameter('fillValue', 0);
ip.addParameter('boundX', [0, 0]);
ip.addParameter('boundY', [0, 0]);
ip.parse(varargin{:});
para=ip.Results;

boundX = para.boundX;
boundY = para.boundY;
fillValue = para.fillValue;
setBound = 1;
if (boundX(2)==boundX(1) || boundY(1)==boundY(2))
    setBound=0;
end

[M, N] = size(imgf);
X = [1 N N 1];
Y = [1 1 M M];
tmp = matT * [X; Y; ones(1,4)];

minX = floor(min(tmp(1,:)));
minY = floor(min(tmp(2,:)));
maxX = ceil(max(tmp(1,:)));
maxY = ceil(max(tmp(2,:)));

if(setBound==1)
    minX = boundX(1);
    maxX = boundX(2);
    minY = boundY(1);
    maxY = boundY(2);
end
imgg = zeros(maxY-minY, maxX-minX);

for i=(minY:maxY)
    for j=(minX:maxX)
        tmp = matT \ [j;i;1];
        u = tmp(1);
        v = tmp(2);
        if(isequal('bilinear', interpolation))
            [X, Y, out] = bilinear_rect(tmp, M, N);
            if(out==1)
                imgg(i-minY+1, j-minX+1) = fillValue;
                continue;
            end
            g_12 = (X(2)-u)/(X(2)-X(1))*imgf(Y(1),X(1)) + (u-X(1))/(X(2)-X(1))*
                imgf(Y(1),X(1));
            g_34 = (X(3)-u)/(X(3)-X(4))*imgf(Y(4),X(4)) + (u-X(4))/(X(3)-X(4))*
                imgf(Y(3),X(3));
            imgg(i-minY+1, j-minX+1) = (Y(3)-v)/(Y(3)-Y(2))*g_12 + (v-Y(2))/(Y
                (3)-Y(2))*g_34;
        end
        if(isequal('nearest', interpolation))
                if(round(u)>=1 && round(u)<=N && round(v)>=1 && round(v)<=M)
                imgg(i-minY+1, j-minX+1) = imgf(round(v), round(u));
            end
        end
    end
end
end

```

```
function [ X, Y, out ] = bilinear_rect(orig_p , M, N)
u = orig_p(1);
v = orig_p(2);
X = zeros(1,4);
Y = zeros(1,4);
X(1) = floor(u);
X(3) = ceil(u);
if(X(1)==X(3))
        X(3) = X(1)+1;
end
Y(1) = floor(v);
Y(3) = ceil(v);
if(Y(1)==Y(3))
        Y(3) = Y(1)+1;
end
out = 0;
if(X(1)<1 || X(3)>N || Y(1)<1 || Y(3)>M)
         out = 1;
end
X(4) = X(1);
X(2) = X(3);
Y(4) = Y(3);
Y(2) = Y(1);
end
```

## 5.5 Discussion

When we are implementing the coordinate transform, we should pay attention to that the coordinate of an image is not the same as the one in mathematics. For example, for efficiency reason, we usually loop 'y' outer and loop 'x' inner. Hence, be careful to use the loop variables correctly while indexing the image matrix. Another difference is that in mathematics, we use counter-clock direction as positive direction for rotation while here we use clockwise direction as positive direction. In this experiment, I didn't see large difference between bilinear interpolation and nearest neighbor interpolation. However, in practice, we should consider the efficiency and effects when we are choosing the proper interpolation algorithm.

# 6 Project 8 - Morphological Processing

## 6.1 Project Proposal

Implement the *"Opening by reconstruction", "Filling holes" and "Border clearing"* operations on textbook *chapter 9.5*. The task is to reproduce the results in *Figure 9.29, 9.31 and 9.32*.

## 6.2 Preliminaries

### 6.2.1 Basic morphological operations

Morphology offers a unified and powerful approach to numerous image processing problem. These operations defined based on set theory. In this project we consider that morphological operations are conducted on binary images (preprocessing is required for gray-level images). The below table describes the basic widely used morphological processing.

Table 6.1: Summary of basic morphological operations.

| Operation | Equation | Comments |
|---|---|---|
| Translation | $(B)_z = \{w \mid w = b + z,\ b \in B\}$ | Translation the origin of $B$ to point $z$. |
| Reflection | $\hat{B} = \{w \mid w = -b,\ b \in B\}$ | Reflects all elements of B about the origin of this set. |
| Complement | $A^c = \{w \mid w \notin A\}$ | Set of points not in $A$. |
| Difference | $A - B = \{w \mid w \in A, w \notin B\}$ | Set of points that belong to $A$ but not to $B$. |
| Dilation | $A \oplus B = \left\{z \mid (\hat{B}_z) \cap A \neq \emptyset\right\}$ | Expands the boundary of $A$. |
| Erosion | $A \ominus B = \{z \mid (B)_z \subseteq A\}$ | Contracts the boundary of $A$. |
| Opening | $A \circ B = (A \ominus B) \oplus B$ | Smoothes contours, breaks narrow isthmuses, and eliminates small islands and sharp peaks. |
| Closing | $A \bullet B = (A \oplus B) \ominus B$ | Smoothes contours, fuses narrow breaks and long thin gulfs and eliminates small holes. |
| Hit-or-miss transform | $A \otimes B = (A \ominus B_1) \cap (A^c \ominus B_2)$ | The set of pints at which, simultaneously $B_1$ found a hit in $A$ and $B_2$ found a match in $A^c$. |

### 6.2.2 Morphological restoration

With the basic operations, we can discuss a powerful morphological transformation *morphological restoration* that involves two images and a structuring element. One image, the *marker*, contains the starting points for the transformation. The other image, the *mask*, constrains the transformation.

#### Geodesic dilation and erosion

Geodesic dilation and erosion are the central concepts to morphologic reconstruction. Let $F$ denote the maker and $G$ denote the mask and $F \subseteq G$. The *geodesic dilation* of size 1 of the marker with respect to the mask, denoted by $D_G^(1)(F)$, is defined as

$$D_G^{(1)}(F) = (F \oplus B) \cap G \tag{6.1}$$

The geodesic dilation of size $n$ of $F$ with resplect to $G$ is defined as

$$D_G^{(n)}(F) = D_G^{(1)}\left[D_G^{(n-1)}(F)\right] \tag{6.2}$$

with $D_G^{(0)}(F) = F$. Similarly, the *geodesic erosion* of size 1 of marker $F$ with respect to mask $G$ is defined as

$$E_G^{(1)}(F) = (F \ominus B) \cup G \tag{6.3}$$

The geodesic erosion of size $n$ of $F$ with respect to $G$ is defined as

$$E_G^{(n)}(F) = E_G^{(1)}\left[E_G^{(n-1)}(F)\right] \tag{6.4}$$

with $E_G^{(0)}(F) = F$. Geodesic dilation and erosion are duals with respect to set complementation.

**Morphological reconstruction by dilation and by erosion**

*Morphological reconstruction by dilation of a mask G from a marker F*, denoted $R_G^D F$ is defined

$$R_G^D(F) = D_G^{(k)}(F) \tag{6.5}$$

with $k$ such that

$$D_G^{(k)}(F) = D_G^{(k+1)}(F) \tag{6.6}$$

*Morphological reconstruction by erosion of mask G from a marker F*, denoted $R_G^E(F)$ is defined

$$R_G^E(F) = E_G^{(k)}(F) \tag{6.7}$$

with $k$ such that

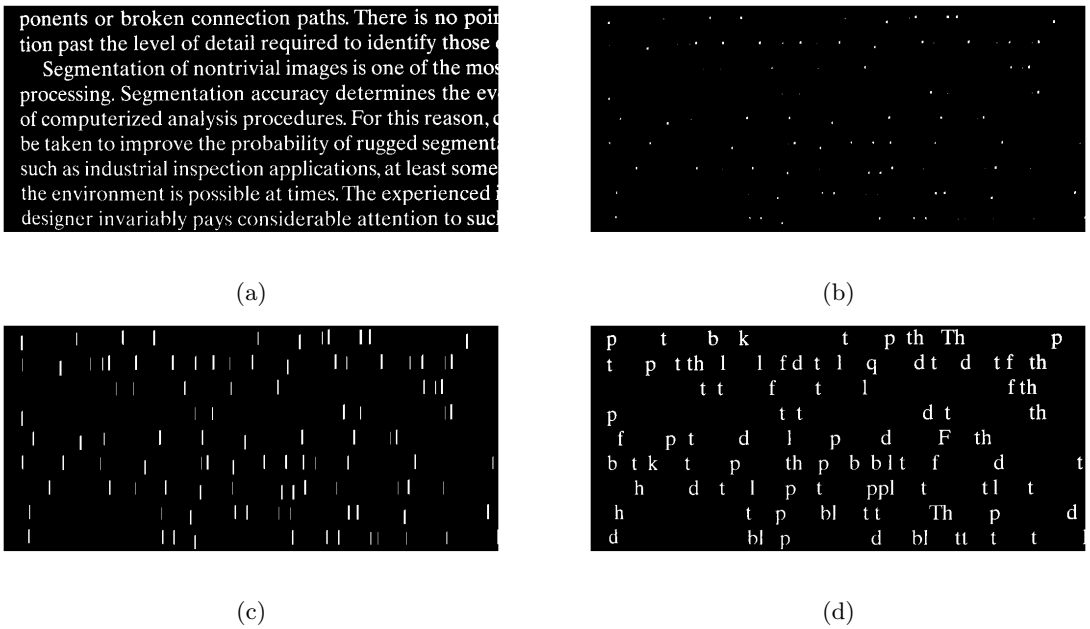
$$E_G^{(k)}(F) = E_G^{(k+1)}(F) \tag{6.8}$$

## 6.3 Task-1 Opening by reconstruction

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

$$O_R^{(n)}(F) = R_F^D\left[(F \ominus nB)\right] \tag{6.9}$$

where $(F \ominus nB)$ indicates $n$ erosions of $F$ by $B$.
Figure 6.1a is the original image for this task. We are interested in extracting the characters that contain long, vertical strokes. The origin image is of size 918 × 2018 pixels. The approximate average height of the tall characters is 50 pixels. Thus we use a structuring element of size 51 × 1 pixels to erode the original image. The erosion image is shown in Figure 6.1b. For comparison, Figure 6.1c shows the opening of Figure 6.1a with the same structuring element. Using erosion image (b) as the marker and the original image (a) as mask, we restored the characters containing long vertical strokes accurately via opening by reconstruction. This result is displayed in Figure 6.1d.

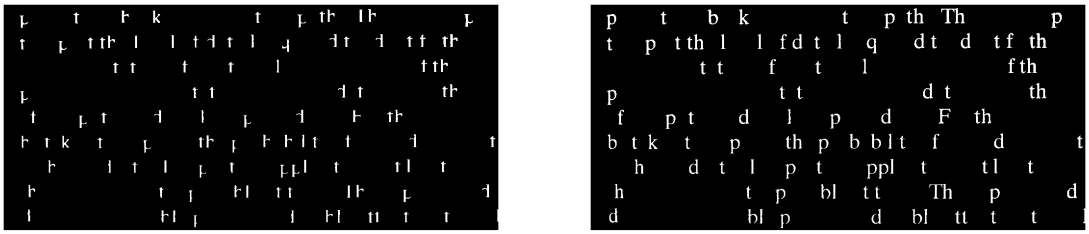



(a) (b)

(c) (d)

Figure 6.1: Task1-opening by reconstruction. (a)Original binary image 918 × 2018. (b)Erosion of (a) with a structuring element of size 51 × 1 pixels. (c)Opening of(a) with the structuring element 51 × 1, shown for comparison. (d)Result of opening by reconstruction.

The size of geodesic dilation reconstruction is 76. This process cost about 7 minutes. I output some of the intermediate results in Figure 6.2 that help us understand the reconstruction better.



(a) (b)

Figure 6.2: See task-1's process in details. (a)Dilation of size 20. (b)Dilation of size 76, the final results.

## 6.4 Task-2 Hole filling

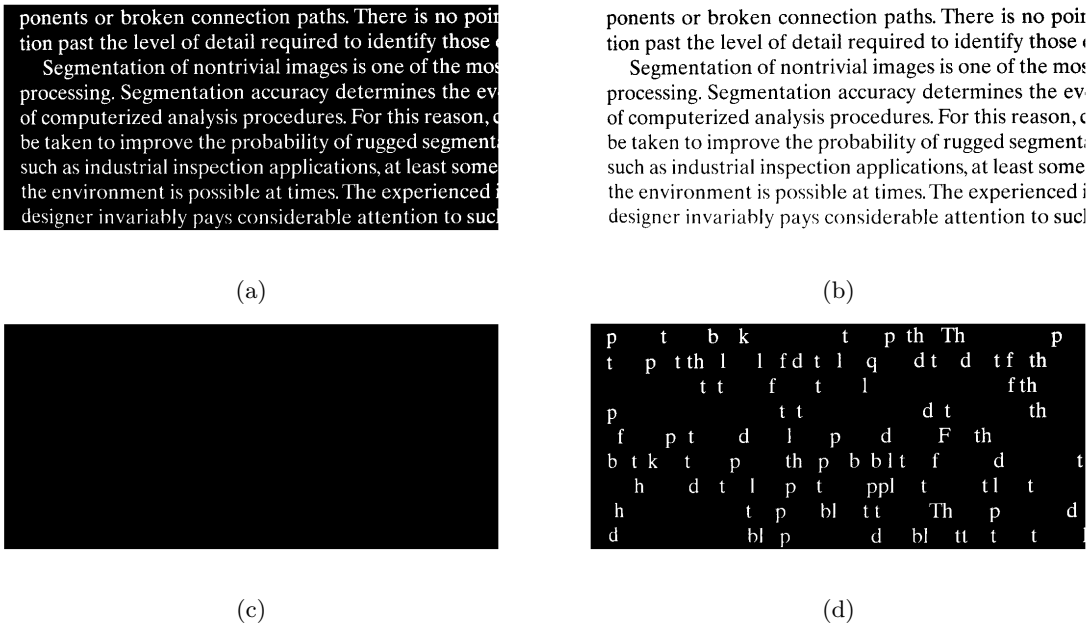
Here we develop a fully automated procedure based on morphological reconstruction. Let $I(x, y)$ denote a binary image and we form a marker $F$ that is 0 everywhere, except at the image border; that is,

$$F(x,y) = \begin{cases} 1 - I(x,y) & \text{if } (x,y) \text{ is on the border of } I \\ 0 & \text{otherwise} \end{cases} \tag{6.10}$$

Then

$$H = \left[R_{I^c}^D(F)\right]^c \tag{6.11}$$

is a binary image equal to $I$ with all holes filled.
I use 3 structuring element. The hole filling images are shown in Figure 6.3. A detailed process of dilation reconstruction with intermediate results are shown in Figure 6.4. The reconstruction takes 479 steps of dilation, which costs a very long period of about 40 minutes.



(a) (b)

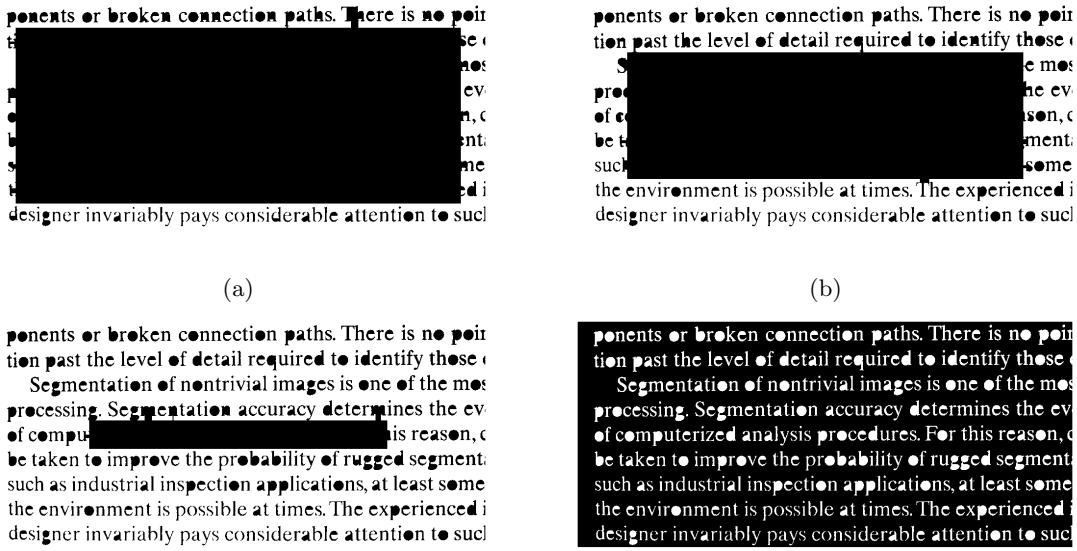
(c) (d)

Figure 6.3: Task2-hole filling. (a)Original binary image 918 × 2018. (b)Complement of (a). Used as a mask. (c)Marker image. Seems like a whole black one, but in fact with some white on border. (d)Result of opening by reconstruction.

## 6.5 Task-3 Border clearing

Removing objects that touch the border is a useful work because it can screen images so that only complete objects remain for further processing. The marker $F(x, y)$ is defined as:

$$F(x,y) = \begin{cases} I(x,y) & \text{if } (x,y) \text{ is on the border of } I \\ 0 & \text{otherwise} \end{cases} \tag{6.12}$$



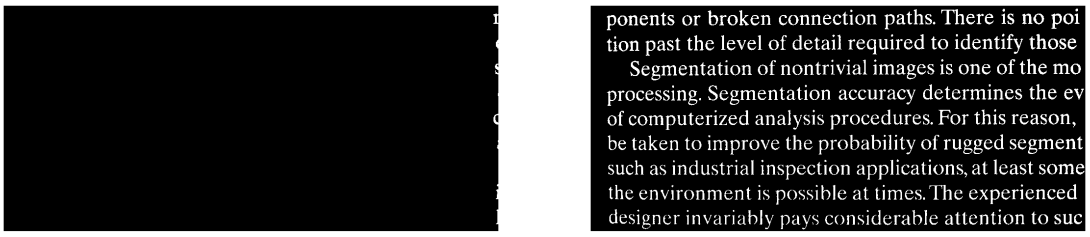
(a) (b) (c) (d)

Figure 6.4: The process of hole filling in details. (a)After 100 steps of dilation. (b)After 200 steps of dilation. (c)After 400 steps of dilation. Much closer to success! (d)The final result with completion.

First we computes morphological reconstruction $R_I^D(F)$ and then computes the desired image $X$

$$X = I - R_I^D(F) \tag{6.13}$$

I use structuring element of size $3 \times 3$. This task is much easier than task 2 and takes just 21 steps of dilation in about 2 minutes. Results are shown in Figure 6.5.



(a) (b)

Figure 6.5: Border clearing. (a)After use border marker for 21-dilation reconstruction. The border letter. (b)The final result after subtraction.

## 6.6 Implementation

Matlab functions *erosion, dilation, geodesic_dilation, dilation_reconstruction and opening_reconstruction* are implemented for this project. I made a **mistake** on opening_reconstruction at first that I still use $51 \times 1$ as the maker while calling dilation_reconstruction. The wrong output image is the same as (c) as all the shorter horizontal adjacent relationship in our wanted characters was damaged. So for correction, we must use $3 \times 3$ as structuring element for dilation_reconstruction called in opening_reconstruction.
In the below code frame, I list the key part of these functions. Other process of calling these functions in main script are trivial so omitted here.

```
% key part of function erosion
for i = (1:M-m+1)
    for j = (1:N-n+1)
        x = imgf(i:i+m-1, j:j+n-1);
        if sum(sum(x.*B)) == m*n
```

```
            g(i+downshift, j+rightshift) = 1;
        end
    end
end

% key part of dilation_reconstruction
k_times = 0;
while( ~isequal(f0, f1) )
    k_times = k_times + 1;
    f0 = f1;
    f1 = geodesic_dilation(f1, G, B);
end

% key part of geodesic_dilation
imgg = dilation(imgf, B) & G;

% key part of opening_reconstruction
for i=(1:n_size)
    f_erosion = erosion(f_erosion, B);
end
[imgg, k_times] = dilation_reconstruction(f_erosion, imgf, ones(3,3)); % here
    can not use ones(51, 1)
```

## 6.7 Discussion

These three tasks show the basic idea of morphological reconstruction used in feature extraction. We start from a marker which can be obtained easily. Then we use the mask as the constrain to conduct set operations. More advanced topics are morphological operations on gray intensity images and image segmentation. This is a very interesting subproject. The main obstacle here is **time**! I tried to do much vectorization but still have a big problem that 20-step-dilation cost about 2 minutes on the $918 \times 2018$ image but the complexity is only about $918 \times 2018 \times 20 \times 9 \approx 4 \times 10^8$. I think the computation of this complexity need no more than 2 sec on language like C++ or python. However, I tried the matlab toolbox *IPT* and it's just as fast as the expectation (within 2 sec). This is a strange but interesting problem.

# 7 Project 9 - Image Segmentation

## 7.1 Project Proposal

There tow parts of project 9. One task is for edge detection: implement the Roberts, Prewitt, Sobel, the Marr-Hildreth and the Canny edge detectors. The test image is *building.tif*. The other task is to implement the Otsus method of thresholding segmentation, and compare the results with the global thresholding method using test image *polymersomes.tif*.

## 7.2 Preliminaries

### 7.2.1 Edge detection

The central idea of edge detection is that local changes in intensity can be detected using derivatives. We have the following conclusions which show that first- and second-order derivatives are particularly well suited for this purpose: (1) First-order derivatives generally produce thicker edges in an image. (2) Second-order derivatives have a stronger response to fine details such as thin lines, isolated points, and noise. (3) Second-order derivatives produce a double-edge response at ramp and step transitions in intensity. (4) The sign of the second derivative can be used to determine whether a transition into an edge is from light to dark or dark to light.

### 7.2.2 Basic edge detection

The tool of choice for finding edge strength and direction at location $(x, y)$ of an image $f$ is the gradient.

$$\nabla f \equiv \text{grad}(f) \equiv \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{7.1}$$

The magnitude of vector $\nabla f$ defined as

$$M(x, y) = mag(\nabla f) = \sqrt{g_x^2 + g_y^2} \approx |g_x| + |g_y| \tag{7.2}$$

is the value of the rate of change in the direction of the gradient vector. The second $=$ is a frequently used approximate to avoid square roots. The direction of gradient vector is given by the angle

$$\alpha(x, y) = \tan^{-1}\left[\frac{g_y}{g_x}\right] \tag{7.3}$$

The direction of an edge at an arbitrary point $(x, y)$ is orthogonal to the direction $\alpha(x, y)$. Here we mention three masks(Figure 7.1) Roberts, Prewitt and Sobel which can be used to compute the gradient of the center point with convolution operation.

| | | | |
|---|---|---|---|
| −1 | 0 | 0 | −1 |
| 0 | 1 | 1 | 0 |

Roberts

| | | | | | |
|---|---|---|---|---|---|
| −1 | −1 | −1 | −1 | 0 | 1 |
| 0 | 0 | 0 | −1 | 0 | 1 |
| 1 | 1 | 1 | −1 | 0 | 1 |

Prewitt

| | | | | | |
|---|---|---|---|---|---|
| −1 | −2 | −1 | −1 | 0 | 1 |
| 0 | 0 | 0 | −2 | 0 | 2 |
| 1 | 2 | 1 | −1 | 0 | 1 |

Sobel

Figure 7.1: 3 masks for gradient computation: Roberts, Prewitt, Sobel

### 7.2.3 More advanced techniques for edge detection