谭树杰 11849060

Course Project 1

1. Use three different PGM images to perform the following options:

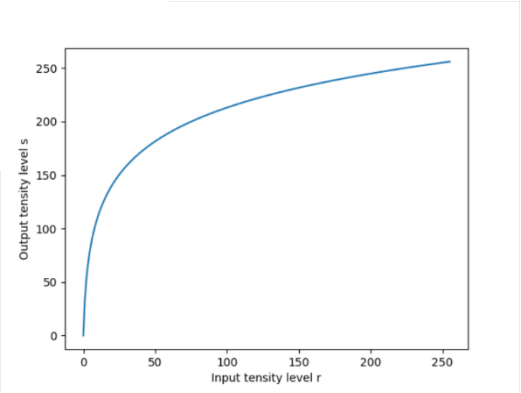
- log transformation and plot the transfer function
- γ correction with $\boldsymbol{\gamma = 0.25, 0.5, 1, 1.5, 2}$

Formula:

The general form of the log transformation in the right picture is:

$$s = c\log(1 + r)$$

where c is a constant, and it is assumed that The shape of the log curve shows that this transformation maps a narrow range of low intensity values in the input into a wider range of output levels. The opposite is true of higher values of input levels. We use a transformation of this type to expand the values of dark pixels in an image while compressing the higher-level values. The opposite is true of the inverse log transformation.

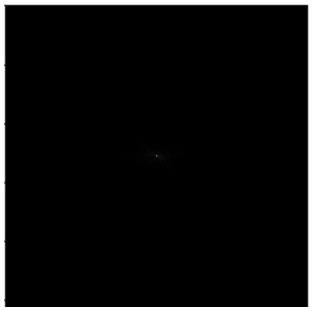
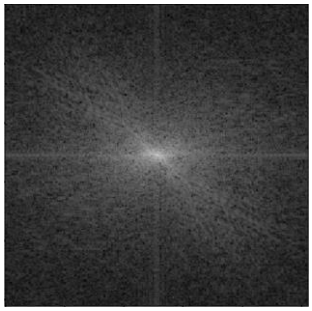


Algorithm:

The algorithm I used to process the image is applying the formula above with c = 20 to the image elementwise.

Results:

Images below in the left are goldhill.pgm, lena.pgm's magnitude image and bridge.pgm. The images in the right are their corresponding log transformed images:

Analysis:

We can observe that the goldhill.pgm and bridge.pgm are improved by expanding the values of dark pixels in an image while compressing the higher-level values. The details are more visible in these two log transformed images. The lena.pgm's magnitude image is improved more significant as we can see. In the original magnitude image, the details of low intensity are hardly can be seen. In contrast, we can analysis easily the magnitude features in the processed image.

Conclusion:

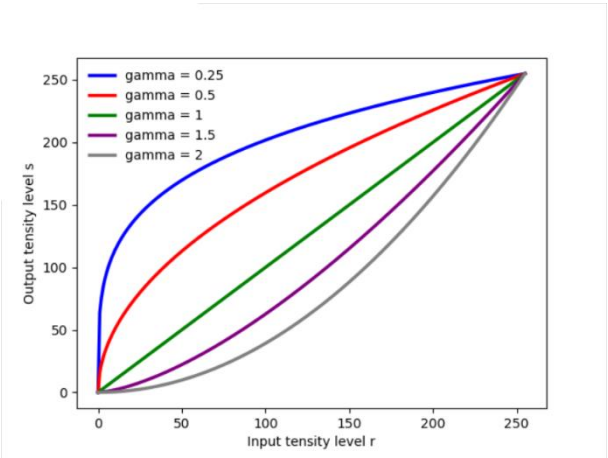
The log transform could be used to accomplish spreading/compressing of intensity levels in an image. Especially we want to analysis the details in low intensity level such as magnitude analysis in Fourier transform.

Formula:

Power-law transformations have the basic form:

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

where c and are positive constants. Sometimes the equation is written as to account for an offset (that is, a measurable output when the input is zero). However, offsets typically are an issue of display calibration and as a result they are normally ignored in the equation. Plots of s versus r for various values of are shown in right figure. As in the case of the log transformation, power-law curves with fractional values of map a narrow range of dark input values into a wider range of output values, with the opposite being true for higher values of input levels.

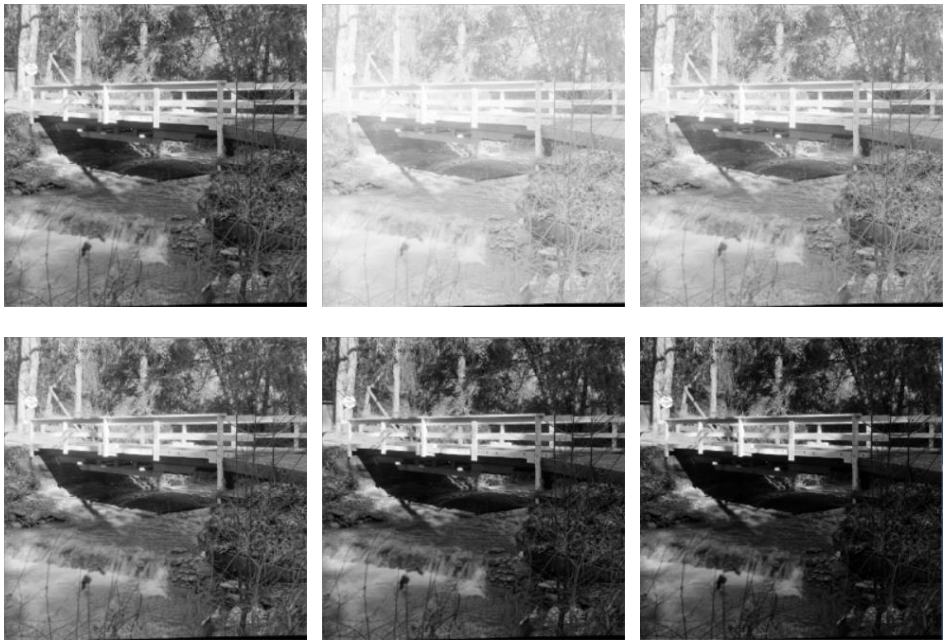


Algorithm:

I use the formula above with gamma = 0.25, 0.5, 1, 1.5, 2 to the image elementwise.

Results:

The bridge.pgm and gamma transformed images with gamma = 0.25, 0.5, 1, 1.5, 2 are shown below.



Analysis:

As we can see, power-law curves with 0.25 and 0.5 map a narrow range of dark input values into a wider range of output values, so the corresponding processed images show more details in dark areas. In contrast, power-law curves with 1.5 and 2 map a narrow range of white input values into a wider range of output values, after gamma transformation, we could observe details in white areas.

Conclusion:

The gamma transformations are much more versatile for spreading/compressing of intensity levels in an image.

Implementation codes:

```python
import numpy as np
import cv2
from myfilters import *


def log_transform(src, c=1):
    # dst = np.zeros_like(src, dtype=float)
    dst = np.log1p(src) * c
    cv2.normalize(dst, dst, 0, 1, cv2.NORM_MINMAX)
    return dst


def gamma_transform(src, gamma=1., c=1.):
    dst = np.power(src, gamma) * c
    cv2.normalize(dst, dst, 0, 1, cv2.NORM_MINMAX)
    return dst


if __name__ == "__main__":
    path_in = './in/'
    path_out = './out/'
    img_path = 'bridge.pgm'

    img_path_in = path_in + img_path
    img_path_out = path_out + 'goldhill_filtered.jpg'

    # Main code
    img = cv2.imread(img_path_in, cv2.IMREAD_GRAYSCALE)
    img = img.astype(float)
    cv2.normalize(img, img, 0, 1, cv2.NORM_MINMAX)
    # cv2.imshow("src Image", img)

    #####
    siz = 5
    # dst = log_transform(img, c=20)
    dst = gamma_transform(img, gamma=2)
    cv2.imshow("processed image", dst)

    cv2.imwrite(img_path_out, dst)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

```
r = np.linspace(0, 255, 256, endpoint=True)
s = 32 * np.log2(1 + r)
plt.plot(r, s)
plt.xlabel("Input tensity level r")
plt.ylabel("Output tensity level s")

r = np.linspace(0, 255, 256, endpoint=True)
s1 = np.power(r, 0.25) / np.power(255, 0.25) * 255
s2 = np.power(r, 0.5) / np.power(255, 0.5) * 255
s3 = np.power(r, 1) / np.power(255, 1) * 255
s4 = np.power(r, 1.5) / np.power(255, 1.5) * 255
s5 = np.power(r, 2) / np.power(255, 2) * 255
plt.plot(r, s1, color="blue", linewidth='2.5', linestyle='-', label="gamma = 0.25")
plt.plot(r, s2, color="red", linewidth='2.5', linestyle='-', label="gamma = 0.5")
plt.plot(r, s3, color="green", linewidth='2.5', linestyle='-', label="gamma = 1")
plt.plot(r, s4, color="purple", linewidth='2.5', linestyle='-', label="gamma = 1.5")
plt.plot(r, s5, color="gray", linewidth='2.5', linestyle='-', label="gamma = 2")
plt.legend(loc="upper left", frameon=False)
plt.xlabel("Input tensity level r")
plt.ylabel("Output tensity level s")
```

2. Generate a 256x256 8-bit image with gray levels varying (using a gaussian distribution) in the range [200, 220], insert a 100x100 square in the range [80, 100]. The image represents an object in a light gray background. Use its histogram information to design an algorithm to remove the background

Formula:

The histogram of an image of size $M \times N$ is given by:

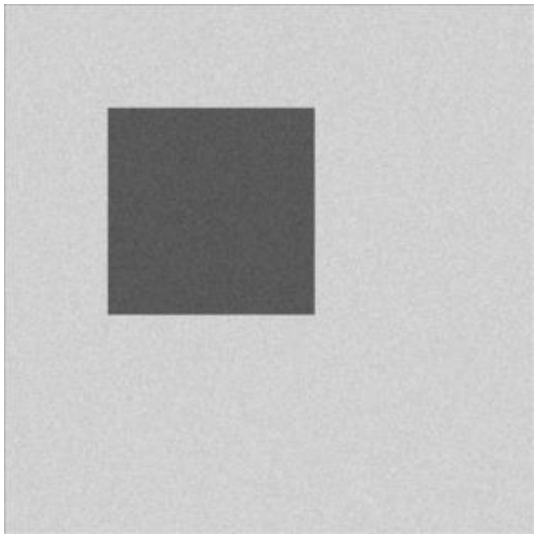
$$p(r_k) = r_k/MN, \text{ for } k = 0, 1, 2, \ldots, L - 1.$$

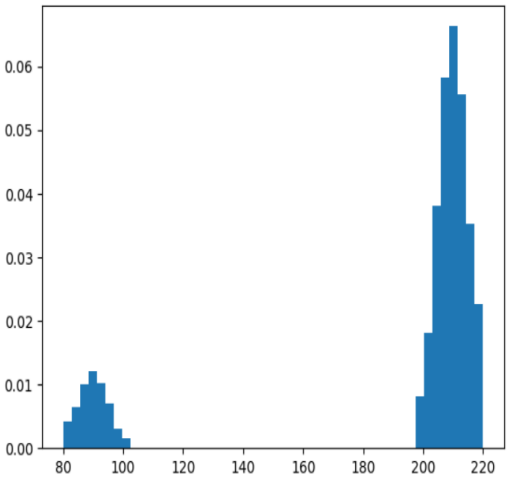
Algorithm:

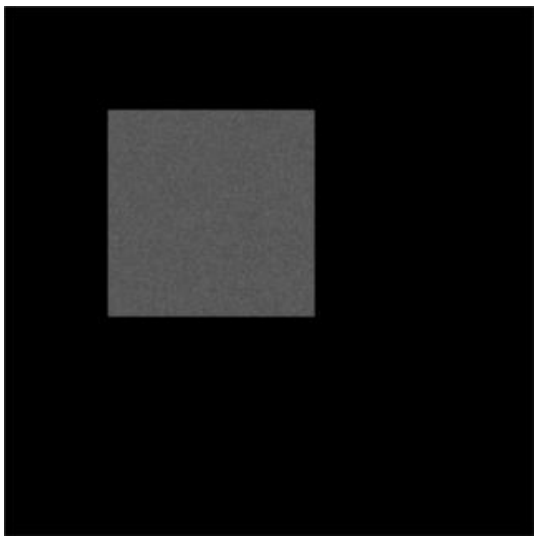
The Due to the discontinuities of the probability of pixel values, we can get the threshold by calculating the gradient of the histogram. When the probability descends to dramatically at one place, we set the corresponding pixel value as threshold. Then we set all pixel values larger than the threshold in the image to zero to remove the background.

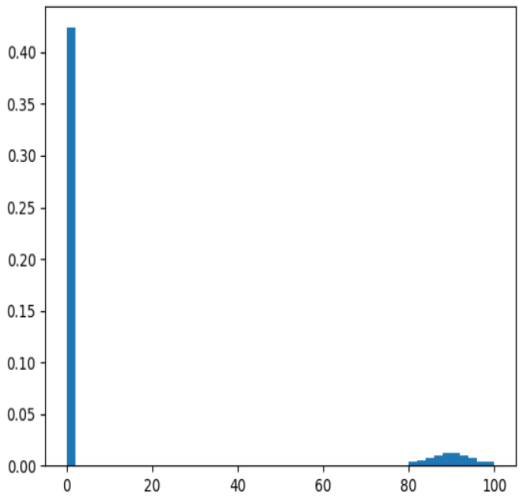
Result:

The original image and result image are shown below in the left side. Their histogram is plotted in the right side below.









Analysis:

We can observe that after remove the background from original image, we can concentrate on the object of interest.

Conclusion:

The histogram can be used to analysis the features of background and object. Then we can remove background from image by using algorithms based on histogram.

Implementation code:

```python
import numpy as np
import cv2
from matplotlib import pyplot as plt


def add_gaussian(src, mean=0, var=0.5):
    row, col = src.shape
    gauss = np.random.normal(mean, var, (row, col))
    gauss = gauss.reshape(row, col)
    noisy = src + gauss
    return noisy


def get_threshold(img):
    hist, holder = np.histogram(img, bins=range(257))
    d = np.diff(hist)
    for i in range(256):
        if d[i] != 0:
            for j in range(i, 256):
                if d[j] == 0:
                    print(j+1)
                    return j+1
    return -1


def thresholding(src, th):
    dst = src.copy()
    dst[src > th] = 0
    return dst


if __name__ == "__main__":
    img = np.zeros((256, 256), dtype=np.uint8)
    img = add_gaussian(img, 210, 5)
    img = np.clip(img, 200, 220)

    obj = np.zeros((100, 100), dtype=np.uint8)
    obj = add_gaussian(obj, 90, 5)
    obj = np.clip(obj, 80, 100)
    img[50:150, 50:150] = obj[:, :]
```

```
src = img
src_vector = np.reshape(src, (src.shape[0] * src.shape[1],))
plt.hist(src_vector, bins=50, normed=True)

img = img.astype(dtype=np.uint8)
cv2.imshow("original image", img)

threshold = get_threshold(img)

dst = thresholding(img, threshold)
cv2.imshow("result image", dst)
plt.title("result image")
plt.imshow(dst, cmap=plt.gray())
src = dst
src_vector = np.reshape(src, (src.shape[0] * src.shape[1],))
plt.hist(src_vector, bins=50, normed=True)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

3. Write a program to perform spatial filtering of an image. You can input the size of the spatial mask and the weight coefficients. Test your program using a blurring filter

Formula:

We use correlation to perform spatial filtering of an image.

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

Where the size of the spatial mask is $(2a + 1) * (2b + 1)$ and the weight coefficients are $w(s, t)$

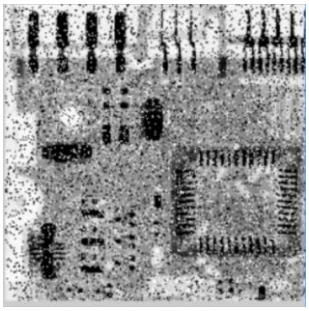
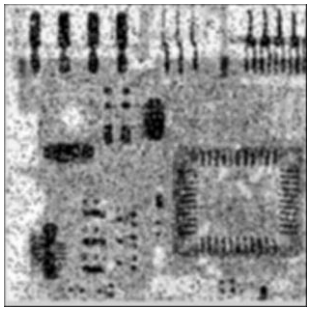
Algorithm:

We simply use mean filter of size 5×5 as blurring filter:

$$K = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Results:

The pcb.pgm(with salt and pepper noise) and its filtered image are shown below respectively.

Analysis:

We should pad the image with zeros first. We reduced the noise by using mean blurring filter, but the details of pcb.pgm was also be blurred severely.

Conclusion：

The spatial filter could be easily implemented. Therefore it's of great use in practice, although it has many limitations.

Implementation code:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

def myfilter2D(src, kernel):
    if kernel.shape[0] != kernel.shape[1]:
        raise Exception("kernel.shape[0] != kernel.shape[1]")
    margin = int(kernel.shape[0] / 2)
    dst = np.zeros_like(src, dtype=float)
    img_padded = np.pad(src, ((margin, margin), (margin, margin)), 'constant')
    for r in range(margin, img_padded.shape[0] - (margin+1)):
        for c in range(margin, img_padded.shape[1] - (margin + 1)):
            filter_window = np.copy(img_padded[r - margin:r + margin + 1, c - margin:c + margin + 1])
            dst[r - margin, c - margin] = np.vdot(filter_window, kernel)
    cv2.normalize(dst, dst, 0, 1, cv2.NORM_MINMAX)
    return dst

if __name__ == "__main__":
    path_in = './in/'
    path_out = './out/'
    img_path = 'pcb.pgm'

    img_path_in = path_in + img_path
    img_path_out = path_out + 'pcb_filtered.jpg'

    # Main code

    img = cv2.imread(img_path_in, cv2.IMREAD_GRAYSCALE)
    img = img.astype(float)
    cv2.normalize(img, img, 0, 1, cv2.NORM_MINMAX)
    cv2.imshow("lenaD1.pgm with noise", img)

    #####
    kernel = np.ones((5, 5), np.float32) / 25
    # src = dst
    dst = myfilter2D(img, kernel)
    cv2.imshow("pcb_filtered", dst)

    cv2.imwrite(img_path_out, dst)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

4. Add random noise to an image, implement 3x3, 5x5 median

filters to the noise added images. Also plot the image histograms before and after the filters

Formula:

The PDF of salt-and-pepper noise is given by:

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

If $b > a$ intensity b will appear as a light dot in the image. Conversely, level a will appear like a dark dot. If either P_a or P_b is zero, the impulse noise is called unipolar. If neither probability is zero, and especially if they are approximately equal, impulse noise values will resemble salt-and-pepper granules randomly distributed over the image.

The median filter replaces the value of a pixel by the median of the intensity levels in the neighborhood of that pixel:

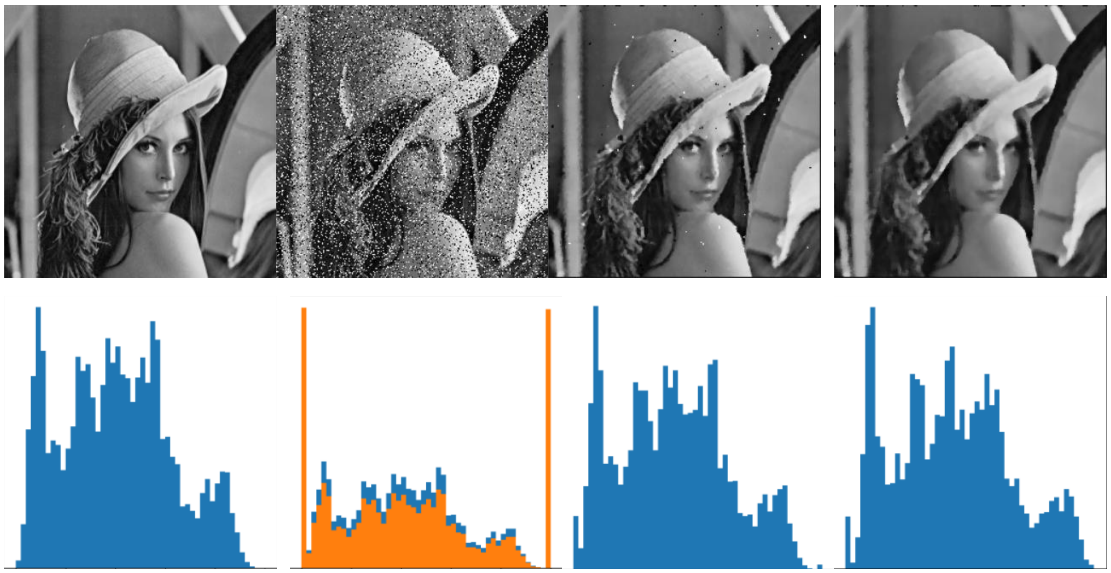
$$\hat{f}(x, y) = \underset{(s,t)\in S_{xy}}{\text{median}}\{g(s, t)\}$$

Algorithm:

We first add salt-and-pepper noise to the image, then simply use median filter to remove the noise. We use 64 histogram bins to plot the image histograms.

Results:

The lena.pgm, lena.pgm with salt-and-pepper noise, lena filtered by 3x3, 5x5 median filter and their corresponding histograms are shown below:



Analysis:

We can see that median filter of size 3x3 improved the noise image significantly although

there're some impulse noise still. The median filter of size 5x5 almost reduced all impulse noise but it blurred more details of the image. The corresponding histograms showed the features of these images. The histogram of lena with salt-and-pepper noise was hugely corrupted obviously. The histogram after median filtering almost restored the histogram of lena.pgm. The result has proved that median filter has excellent capabilities of reducing salt-and-pepper noise.

Conclusion:

The median filter has excellent capabilities of reducing salt-and-pepper noise. However, we should choose filter size carefully to get balance between noise reducing and details preserving in the result image.

Implementation code:

```python
import cv2
import numpy as np
from matplotlib import pyplot as plt
# pa, pb is the probability of salt and pepper noise respectively
def add_salt_and_pepper(src, pa, pb):
    rnd = np.random.rand(src.shape[0], src.shape[1])
    noisy = src.copy()
    noisy[rnd < pa] = 1
    noisy[rnd > 1-pb] = 0
    return noisy


def median_filter(src, size=3):
    margin = int(size / 2)
    dst = np.zeros_like(src, dtype=float)
    img_padded = np.pad(src, ((margin, margin), (margin, margin)), 'constant')
    for r in range(margin, img_padded.shape[0] - (margin + 1)):
        for c in range(margin, img_padded.shape[1] - (margin + 1)):
            filter_window = np.copy(img_padded[r - margin:r + margin + 1, c - margin:c + margin + 1])
            dst[r - margin, c - margin] = np.median(filter_window)
    return dst

if __name__ == "__main__":
    path_in = './in/'
    path_out = './out/'
    img_path = 'lena.pgm'

    img_path_in = path_in + img_path
    img_path_out = path_out + 'lena_filtered.jpg'
```

```
# Main code

img = cv2.imread(img_path_in, cv2.IMREAD_GRAYSCALE)
img = img.astype(float)
cv2.normalize(img, img, 0, 1, cv2.NORM_MINMAX)
cv2.imshow("lena.pgm", img)
src = img
src_vector = np.reshape(src, (src.shape[0] * src.shape[1],))
plt.hist(src_vector, bins=50, normed=True)

# add random noise
img_noise = add_salt_and_pepper(img, 0.1, 0.1)
cv2.imshow("lena_noise", img_noise)
src = img_noise
src_vector = np.reshape(src, (src.shape[0] * src.shape[1],))
plt.hist(src_vector, bins=50, normed=True)

dst = median_filter(img_noise, 5)
cv2.imshow("lena_filtered", dst)
src = dst
src_vector = np.reshape(src, (src.shape[0] * src.shape[1],))
plt.hist(src_vector, bins=50, normed=True)

cv2.imwrite(img_path_out, dst)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

5. Use the unsharp masking technique to enhance three images

Formula:

The median filter has excellent capabilities of reducing salt-and-pepper noise. However, we should choose filter size carefully to get balance between noise reducing and details preserving in the result image.

Unsharp masking consists of the following steps:

1. Blur the original image.
2. Subtract the blurred image from the original (the resulting difference is called the mask.)
3. Add the mask to the original.

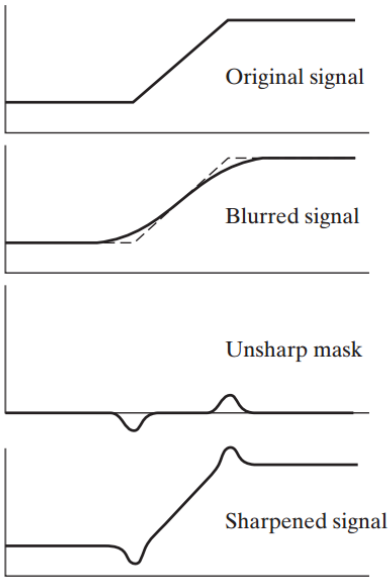
Letting $\overline{f}(x,y)$ denote the blurred image, unsharp masking is expressed in equation form as follows. First we obtain the mask:

$$g_{\text{mask}}(x, y) = f(x, y) - \bar{f}(x, y)$$

Then we add a weighted portion of the mask back to the original image:

$$g(x, y) = f(x, y) + k * g_{\text{mask}}(x, y)$$

where we included a weight, for generality. When we have a weight $k, (k \geq 0)$, for generality, When $k = 1$ we have unsharp masking, as defined above. When $k > 1$,the process is referred to as highboost filtering. Choosing $k < 1$ de-emphasizes the contribution of the unsharp mask.



The above figure shows how it works.

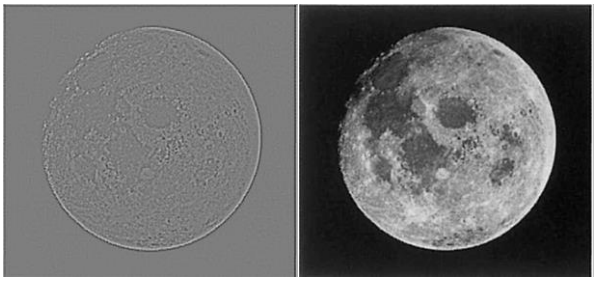
Algorithm:

I just implement it as the above formula and scale the image for visual consideration.

Results:

The image below is the moon.pgm, its blurred image using gaussian_filter(sigma=1), its unsharp mask image and result of using high boost filtering(k=1.5).

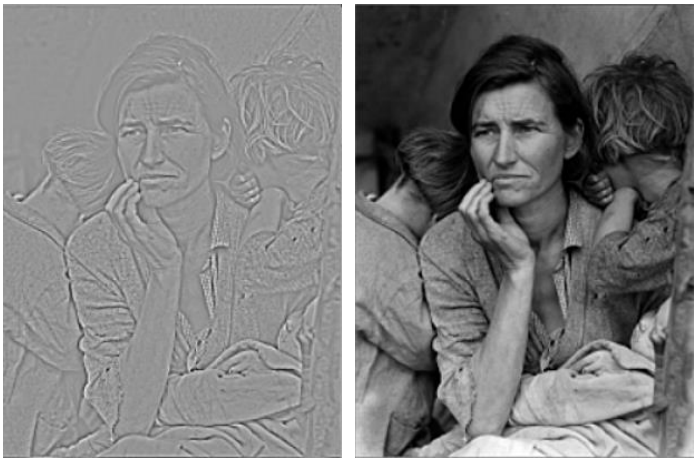
Origin image blurred image



Unsharp mask result image

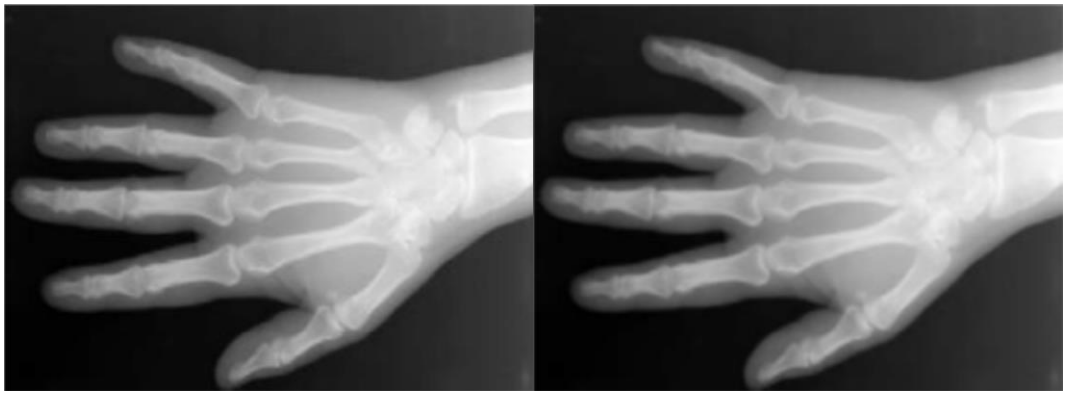
The original image is man.pgm, other settings are same as previous image.

Origin image blurred image



Unsharp mask result image

The original image is hand.pgm, other settings are same as previous image except k=2.5



Origin image

blurred image

Unsharp mask

result image

Analysis:

The We can see that unsharp mask has improved moon.pgm and man.pgm significantly, but only improved the hand.pgm slightly. Maybe we should try larger k to the hand.pgm. We can observe more details of the result image using high boost filtering.

Conclusion:

The unsharp masking technique is very useful to sharpen images. If we are interested in details of an image, we may consider using this method. We should choose a appropriate weight to get best performance of the algorithm.

Implementation code:

```
from __future__ import division
import numpy as np
from scipy.ndimage.filters import gaussian_filter
# from skimage import img_as_float
import cv2
from matplotlib import pyplot as plt



def IHPF(image, D0):
    h = np.ones_like(image)
    cx = int(image.shape[0] / 2)
    cy = int(image.shape[1] / 2)
    for row in range(image.shape[0]):
        for col in range(image.shape[1]):
            if (row-cx)**2 + (col-cy)**2 < D0**2:
                h[row][col] = 0
    return h
```

```
def BHPF(image, D0, N):
    h = np.ones_like(image)
    cx = int(image.shape[0] / 2)
    cy = int(image.shape[1] / 2)
    for row in range(image.shape[0]):
        for col in range(image.shape[1]):
            dist_squre = (row-cx)**2 + (col-cy)**2
            if dist_squre < D0**2:
                tmp = np.power(dist_squre, N)
                h[row][col] = 1. - (1 + tmp / pow(D0**2, 2*N))
    return h


def GHPF(image, D0):
    h = np.ones_like(image)
    cx = int(image.shape[0] / 2)
    cy = int(image.shape[1] / 2)
    for row in range(image.shape[0]):
        for col in range(image.shape[1]):
            dist_squre = (row - cx) ** 2 + (col - cy) ** 2
            if dist_squre < D0**2:
                h[row][col] = 1 - np.exp(-dist_squre / (2. * D0 * D0))
    return h


def apply_filter(img, filter="IHPF", para=()):
    if len(img.shape) is not 2:
        raise Exception('Improper image')
    img_fft = np.fft.fft2(img)
    img_fft_fftshift = np.fft.fftshift(img_fft)

    # calculate magnitude
    # mag = np.abs(img_fft_fftshift)
    # mag = 20 * np.log(mag)
    # img_show('magnitude before filtering', mag)

    if filter == "IHPF":
        h = IHPF(img, para[0])
    elif filter == "BHPF":
        h = BHPF(img, para[0], para[1])
    else:
        h = GHPF(img, para[0])
```

```
    img_fft_filtered = img_fft_fftshift * h
    img_fft_filtered_unshift = np.fft.fftshift(img_fft_filtered)
    img_filt = np.fft.ifft2(img_fft_filtered_unshift)
    dst = np.abs(img_filt)
    return dst


def img_show(strg, image):
    dst = np.copy(image)
    np.clip(image, 0, 1, dst)
    cv2.imshow(strg, dst)
    return None


if __name__ == "__main__":
    path_in = './in/'
    path_out = './out/'
    img_path = 'cameraman.pgm'

    img_path_in = path_in + img_path
    img_path_out = path_out + 'cameraman_filtered.jpg'

    # Main code

    img = cv2.imread(img_path_in, cv2.IMREAD_GRAYSCALE)
    img = img.astype(float)
    cv2.normalize(img, img, 0, 1, cv2.NORM_MINMAX)
    img_show("cameraman.pgm", img)

    # dst = apply_filter(img, "IHPF", (30,))
    # img_show("IHPF 30", dst)
    # dst = apply_filter(img, "IHPF", (10,))
    # img_show("IHPF 10", dst)
    # dst = apply_filter(img, "IHPF", (1,))
    # img_show("IHPF 1", dst)

    # dst = apply_filter(img, "BHPF", (30, 2))
    # img_show("GHPF 30", dst)
    # dst = apply_filter(img, "IHPF", (10, 2))
    # img_show("GHPF 10", dst)
    # dst = apply_filter(img, "IHPF", (1, 2))
    # img_show("GHPF 1", dst)
```

```
dst = apply_filter(img, "GHPF", (30,))
img_show("GHPF 30", dst)
dst = apply_filter(img, "GHPF", (10,))
img_show("GHPF 10", dst)
dst = apply_filter(img, "GHPF", (1,))
img_show("GHPF 1", dst)

cv2.imwrite(img_path_out, dst)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

6. Implement IHPF, BHPF, GHPF on three images

Formula:

A 2-D ideal highpass filter (IHPF) is defined as

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

where $D0$ is the cutoff frequency and:

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

A 2-D Butterworth highpass filter (BHPF) of order n and cutoff frequency is defined as:

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

The transfer function of the Gaussian highpass filter (GHPF) with cutoff frequency locus at a distance from the center of the frequency rectangle is given by

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

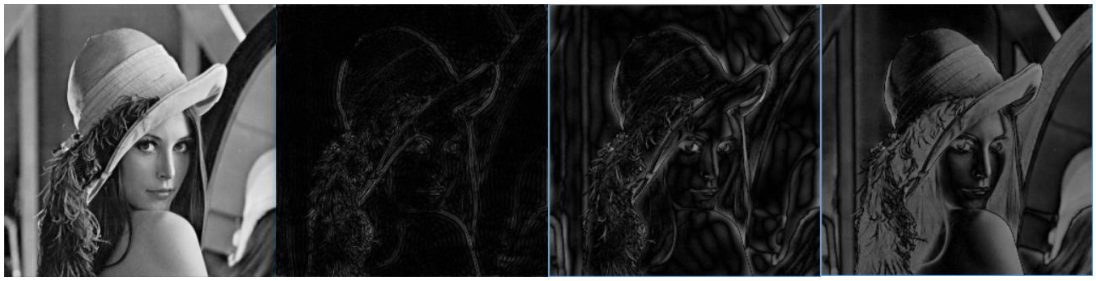
Algorithm:

The images are fisrt transformed to frequency domain by DFT than using above filters to pass the high frequencies. Then restored the image to spatial domain by IDFT:

Result:

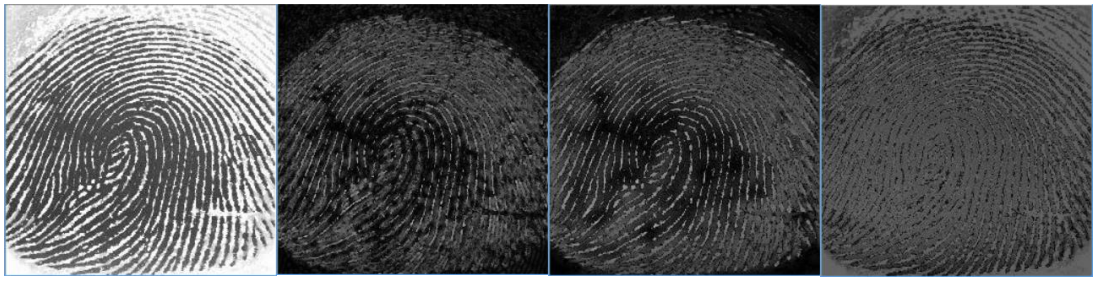
(1) Result of Using IHPF:

Lena.pgm and IHPF filtered images with D0 = 30, 10, 1 are shown below:



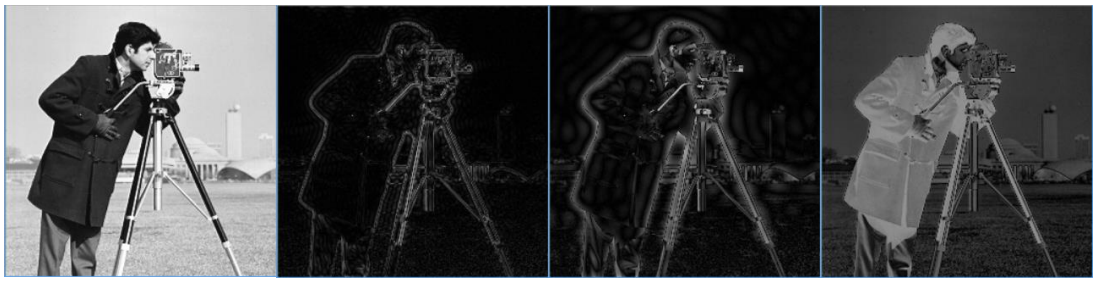
Lena.pgm D0 = 30 D0=10 D0 = 1

fingerprint.pgm and IHPF filtered images with D0 = 30, 10, 1 are shown below:



fingerprint.pgm D0 = 30 D0=10 D0 = 1

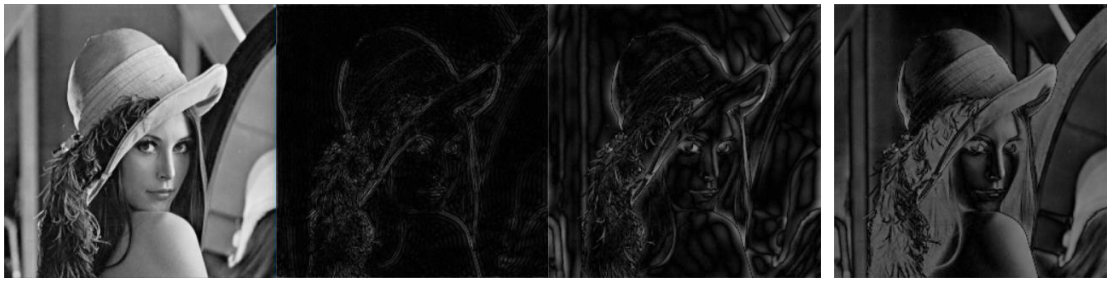
cameraman.pgm and IHPF filtered images with D0 = 30, 10, 1 are shown below:



cameraman.pgm D0 = 30 D0=10 D0 = 1

(2) Result of Using BHPF:

Lena.pgm and BHPF filtered images with D0 = 30, 10, 1, n=2 are shown below:

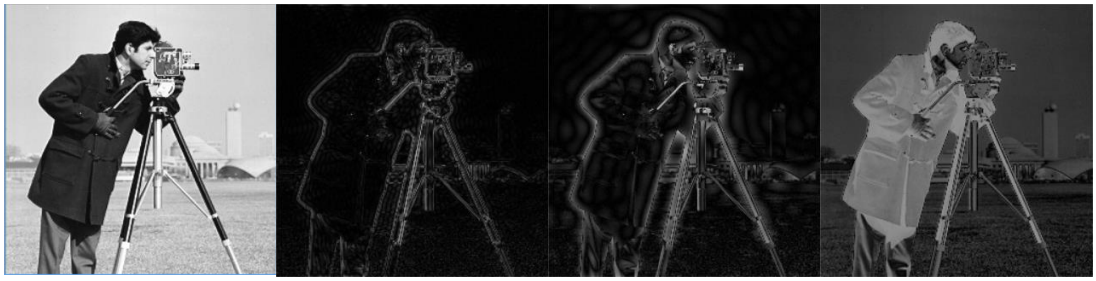


Lena.pgm D0 = 30 D0=10 D0 = 1

fingerprint.pgm and BHPF filtered images with D0 = 30, 10, 1, n=2 are shown below:

fingerprint.pgm D0 = 30 D0=10 D0 = 1

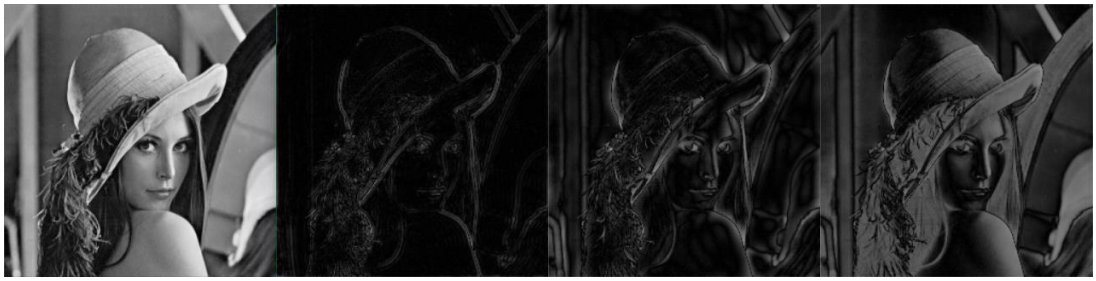
cameraman.pgm and BHPF filtered images with D0 = 30, 10, 1, n=2 are shown below:



cameraman.pgm D0 = 30 D0=10 D0 = 1

(3) Result of Using GHPF:

Lena.pgm and GHPF filtered images with D0 = 30, 10, 1 are shown below:



Lena.pgm D0 = 30 D0=10 D0 = 1

fingerprint.pgm and GHPF filtered images with D0 = 30, 10, 1 are shown below:

fingerprint.pgm D0 = 30 D0=10 D0 = 1

cameraman.pgm and GHPF filtered images with D0 = 30, 10, 1 are shown below:

cameraman.pgm	D0 = 30	D0=10	D0 = 1

Analysis:

We can see the enhancement of print ridges in fingerprint.pgm. The highpass filter won't change the high frequencies contained in the ridges. In contrast, the filter reduces low frequency components corresponding to the smudges and background.

The improvement of lena.pgm and camera.pgm is not obvious, for that we are not interested in ridges of these images. But the HPF can be used to capture the features of an image, for that the edges provide abundant information of the image as features.

As expected, the highpass-filtered image lost its gray tones because the dc term was reduced to 0

Conclusion:

The HPF can be used to detect ridges of an image. It would be especially useful when we are interested in details rather than background information.

Implementation code:

```
from __future__ import division
import numpy as np
from scipy.ndimage.filters import gaussian_filter
# from skimage import img_as_float
import cv2
from matplotlib import pyplot as plt
def unsharp_mask(image, radius, k, vrange=None):
    blurred = gaussian_filter(image, sigma=radius, mode='reflect')
    img_show("blurred image", blurred)
    g_mask = image - blurred
    g_mask_show = g_mask.copy()
    cv2.normalize(g_mask, g_mask_show, 0, 1, cv2.NORM_MINMAX)
    cv2.imshow("unsharp mask", g_mask_show)
    result = image + k * g_mask
    img_show("result", result)
    if vrange is not None:
        return np.clip(result, vrange[0], vrange[1], out=result)
    return result
```

```
def img_show(strg, image):
    dst = np.copy(image)
    np.clip(image, 0, 1, dst)
    cv2.imshow(strg, dst)
    return None
if __name__ == "__main__":
    path_in = './in/'
    path_out = './out/'
    img_path = 'hand.pgm'

    img_path_in = path_in + img_path
    img_path_out = path_out + 'hand_filtered.jpg'

    # Main code

    img = cv2.imread(img_path_in, cv2.IMREAD_GRAYSCALE)
    img = img.astype(float)
    cv2.normalize(img, img, 0, 1, cv2.NORM_MINMAX)
    img_show("hand.pgm", img)

    dst = unsharp_mask(img, 1, 2.5)

    cv2.imwrite(img_path_out, dst)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```