Thierry Cantenot

Report

Digital Image Processing

« Assignments »

2014-2015

Summary

A. Histogram Equalization

A.1 Problem statement

1. Write a computer program for computing the histogram of an image.
2. Implement the histogram equalization technique.
3. Your program must be general to allow any gray-level image as its input.

A.2 Python implementation

Usage : **python problem1.py [-h] image_path**

A.3 Figure 1

A.3.1 Histogram

Original image : A.1 | Original image's histogram : A.2

A.3.2 Histogram equalization

Enhanced image : A.3 | Enhanced image's histogram : A.4

A.4 Figure 2

A.4.1 Histogram

Original image : A.5 | Original image's histogram : A.6

A.4.2 Histogram equalization

Enhanced image : A.7 | Enhanced image's histogram : A.8

Figure A.1 – Original *Fig1.jpg*

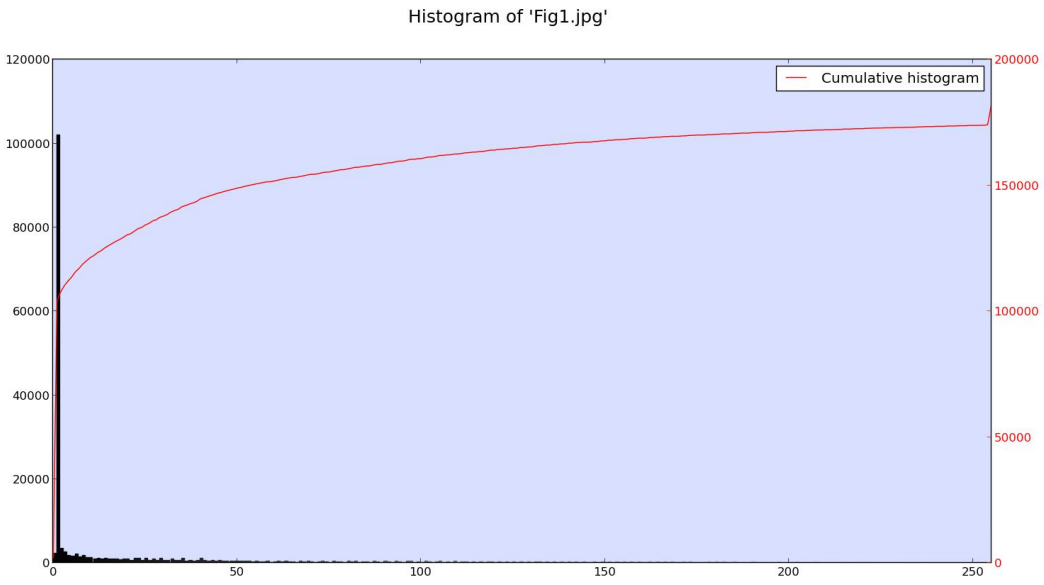

Figure A.2 – Histogram of *Fig1.jpg*

Figure A.3 – Enhanced *Fig1.jpg*

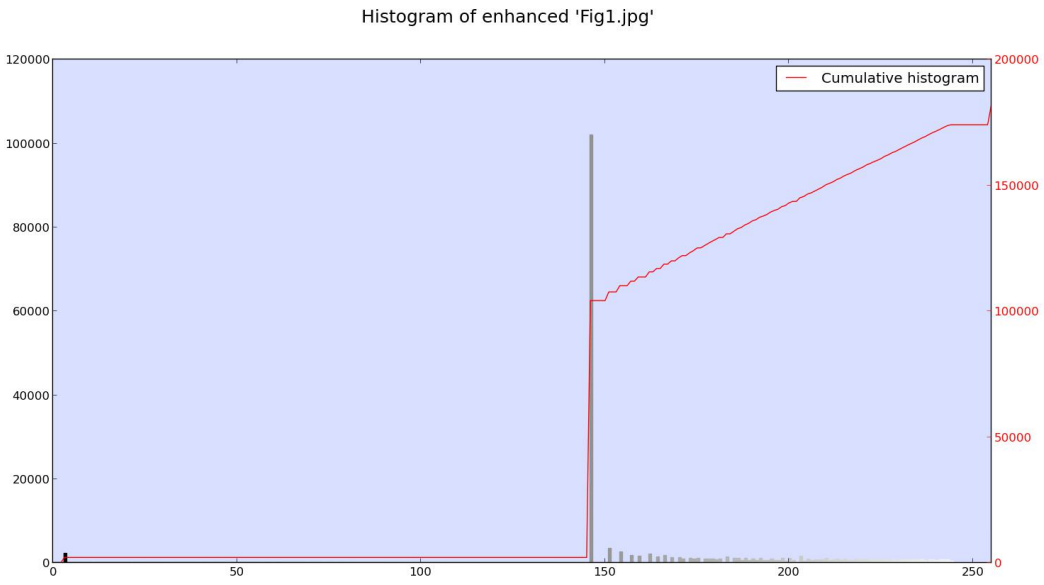

Figure A.4 – Equalized histogram of *Fig1.jpg*

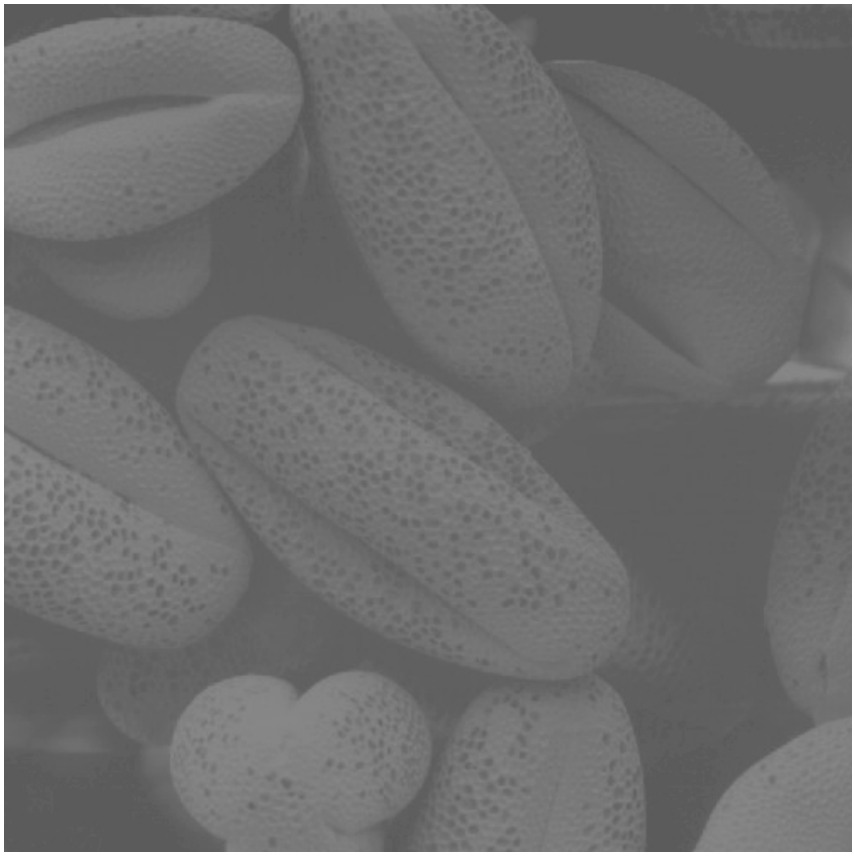

Figure A.5 – Original *Fig2.jpg*

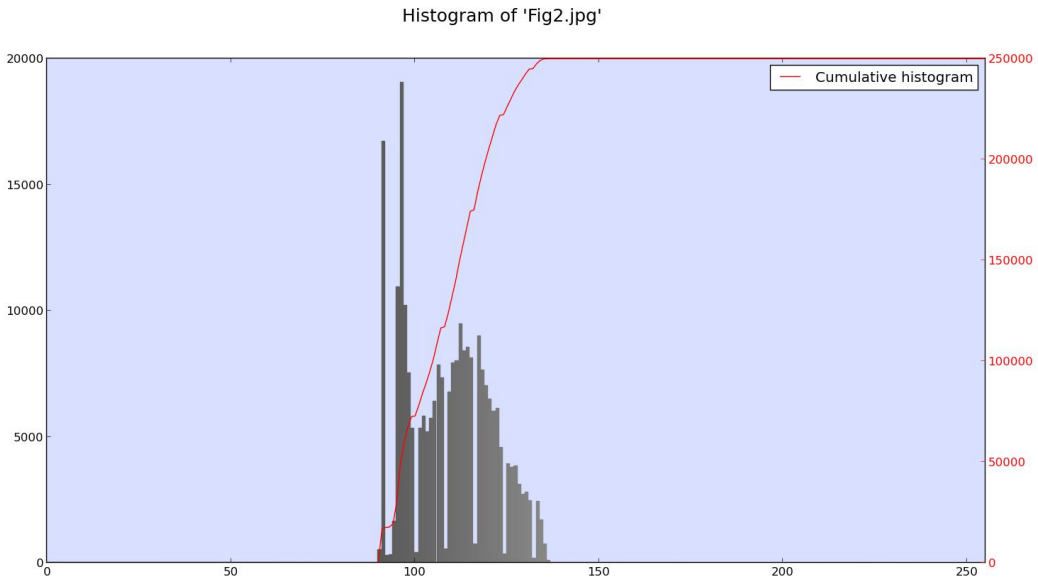

Figure A.6 – Histogram of *Fig2.jpg*

Figure A.7 – Enhanced *Fig2.jpg*

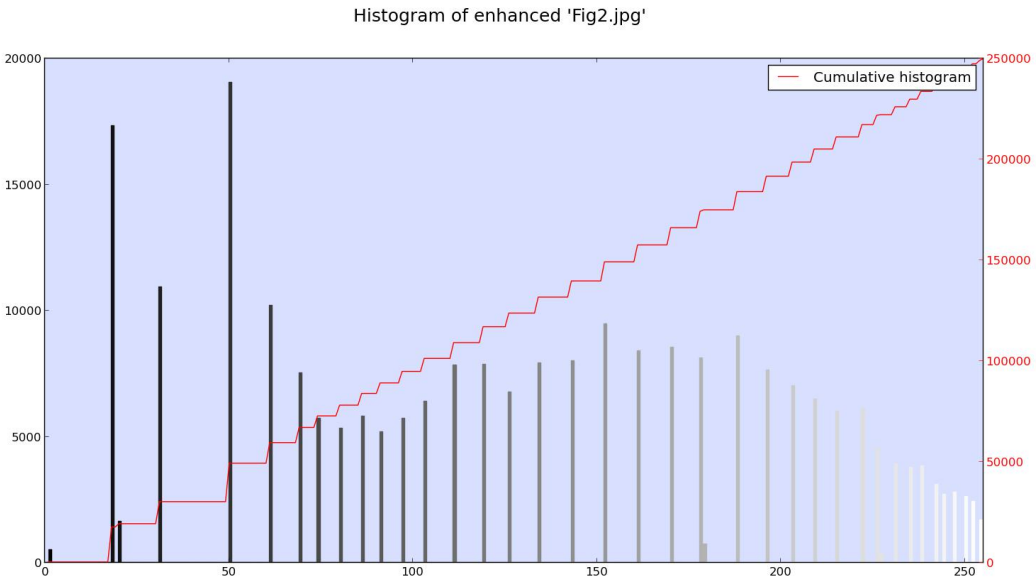

Figure A.8 – Equalized histogram of *Fig2.jpg*

B. Spatial enhancement methods

B.1 Problem statement

Implement the image enhancement task of Section 3.7 (Fig 3.43) (Section 3.8, Fig 3.46 in our slides).
The image to be enhanced is *skeleton_ orig.tif*.
You should implement all steps in Figure 3.43.
(You cannot directly use functions of Matlab such as imfilter or fspecial, implement all functions by yourself).

B.2 Python implementation

Usage : **python problem2.py [-h] [–laplacian] [–sobel] [-a A] [-g G] [-c C] image_path**

For example, to run the full image enhancement described in the assignment, using a 3×3 Laplacian filter with A = 1.7, then a Sobel, a smoothing filter and a Power-Law transformation with c = 1 and gamma = 0.5 type :

python problem2.py –laplacian -a 1.7 –sobel -g 0.5 -c 1 skeleton_orig.tif

B.3 Results

B.3.1 Original image

Figure B.1 – Original *skeleton_ orig.tif*

B.3.2 3x3 Laplacian (A = 0)

Figure B.2 – Laplacian (A=0)

Figure B.3 – Sharpened image

Figure B.4 – Original image

B.3.3 3x3 Laplacian (A = 1)

Figure B.5 – Laplacian (A=1)

Figure B.6 – Sharpened image

Figure B.7 – Original image

B.3.4 3x3 Laplacian (A = 1.7)

Figure B.8 – Laplacian (A=1.7)

Figure B.9 – Sharpened image

Figure B.10 – Original image

B.3.5 Sobel

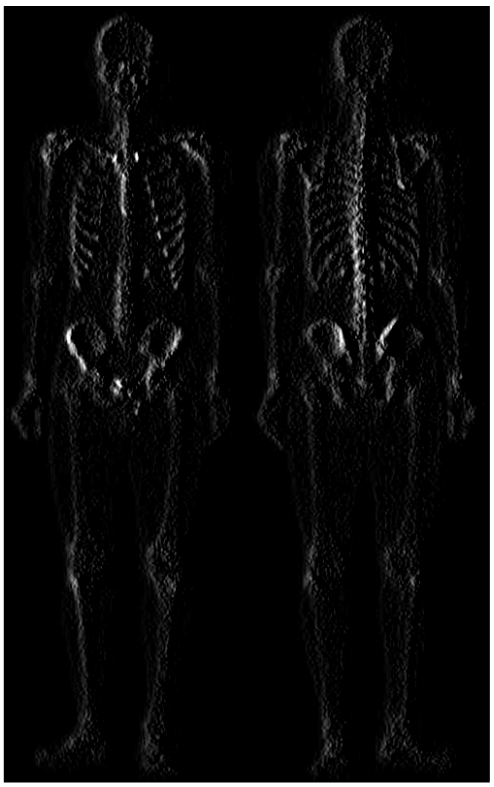

Figure B.11 – Sobel x-gradient

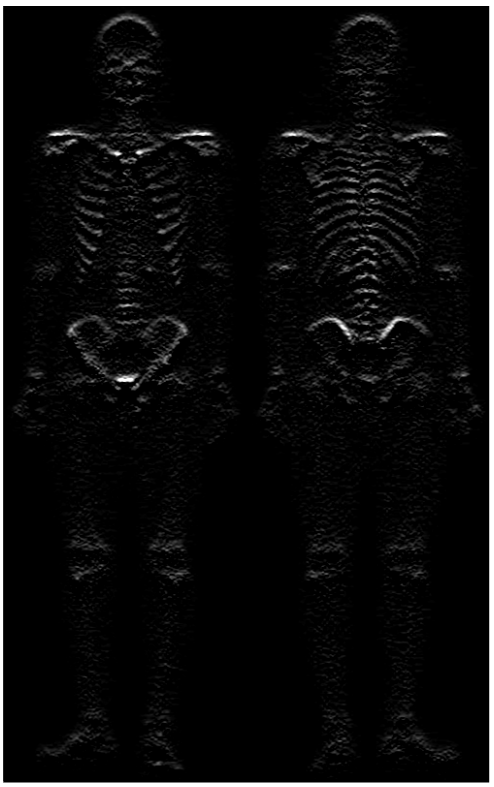

Figure B.12 – Sobel y-gradient

Figure B.13 – Full Sobel

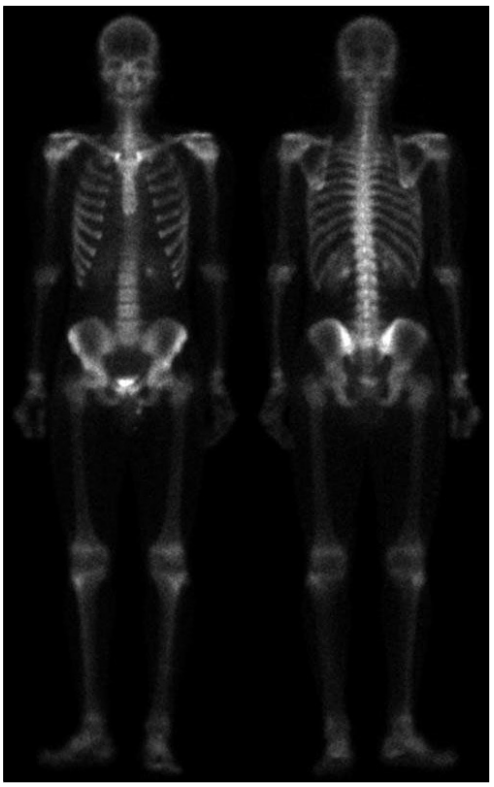

Figure B.14 – Original image

B.3.6 Smoothing, Sharpening and Power-Law transformation

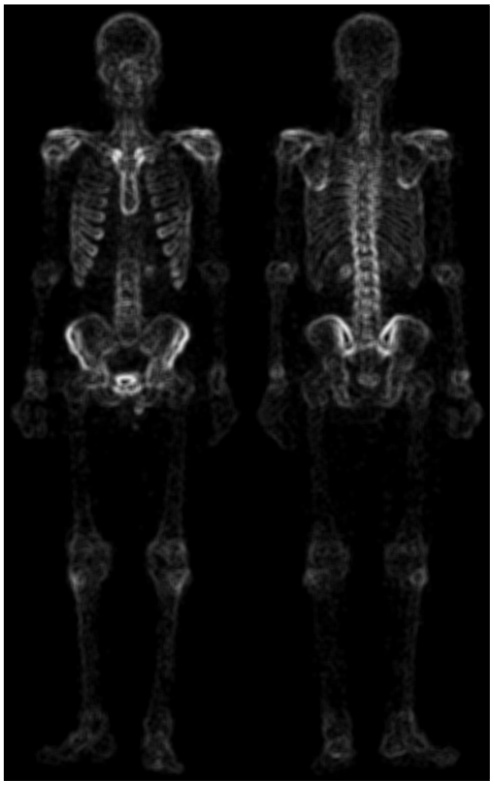

Figure B.15 – Smooth Sobel

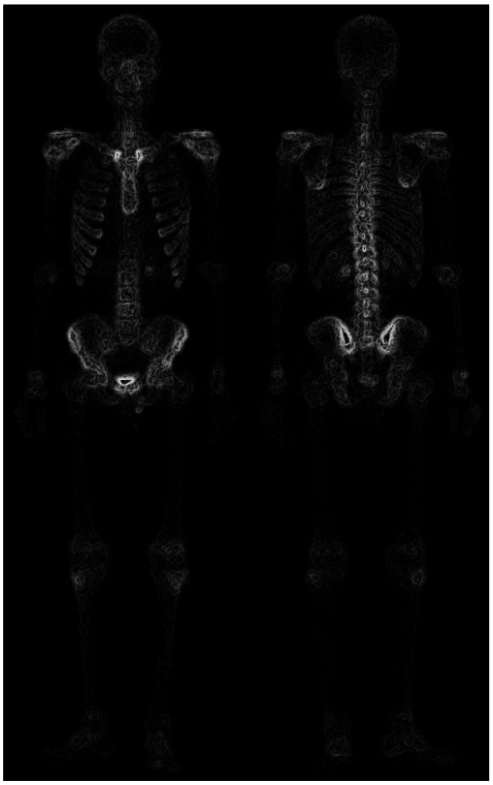

Figure B.16 – Laplacian x Sobel

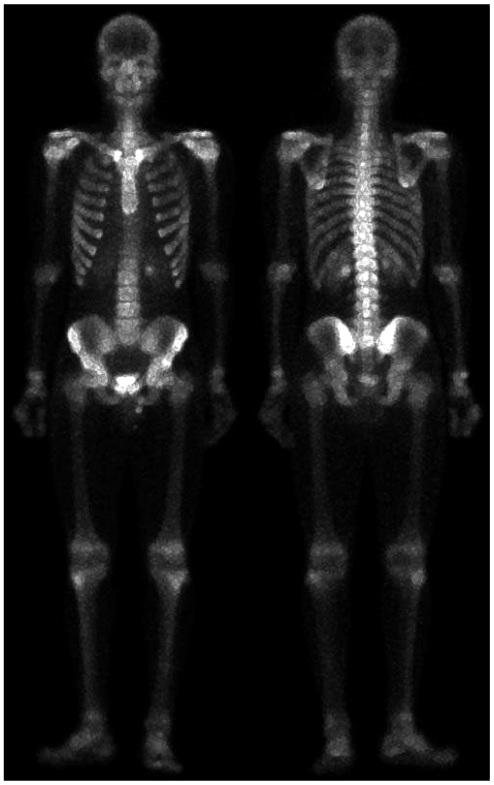

Figure B.17 – Original + Laplacian x Smooth Sobel

Figure B.18 – Final image (after Power-Law (c=1, g=0.5))

B.3.7 Comparison

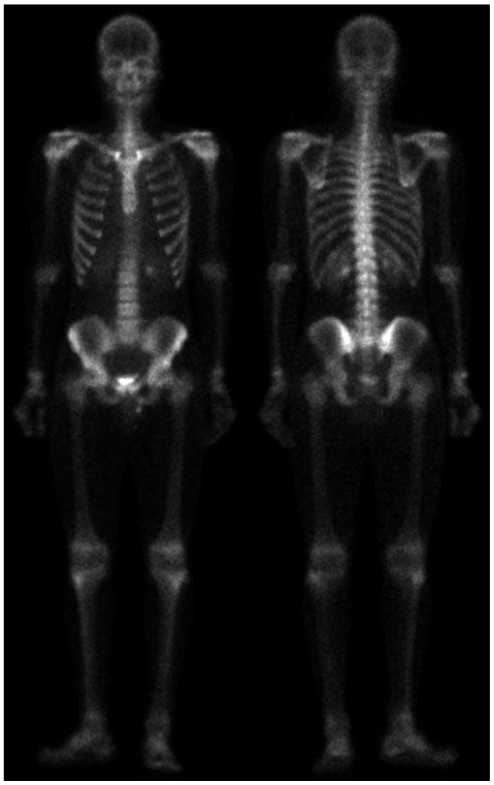

Figure B.19 – Original

Figure B.20 – Final image

C. Filtering in frequency domain

C.1 Problem statement

Implement the ideal, Butterworth and Gaussian lowpass and highpass filters and test them under different parameters using *characters_ test_ pattern.tif*.

C.2 Python implementation

Usage : **python problem3.py [-h] [–ideal] [–butterworth] [–gaussian] (–low | –high) [-d D] [-n N] image_path**

Use **python problem3.py -h** to see the help.

C.3 Results

C.3.1 Original image

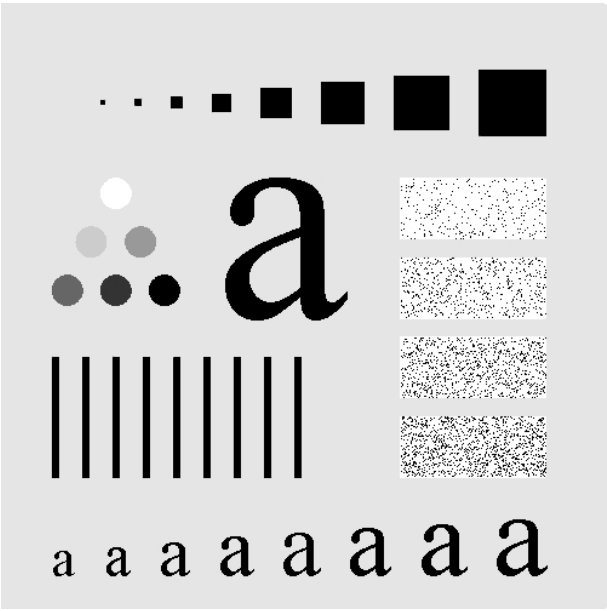

Figure C.1 – Original *characters_ test_ pattern.tif*

C.3.2 Ideal filter

Low pass

python problem3.py –ideal -d 5 –low characters_test_pattern.tif

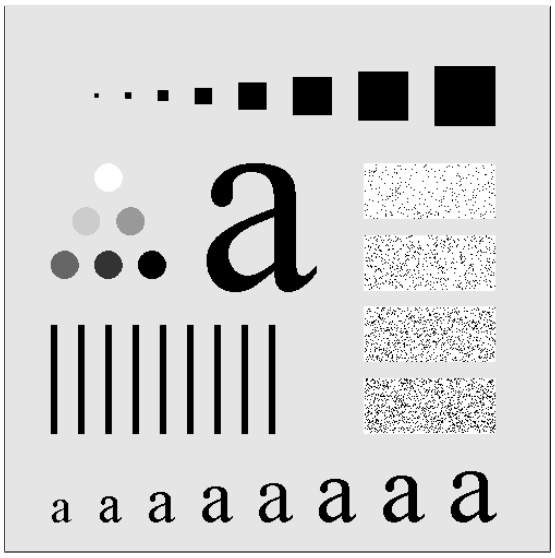

Figure C.2 – Original image

Figure C.3 – Ideal low 5

Figure C.4 – Ideal low 15

Figure C.5 – Ideal low 30

High pass

python problem3.py –ideal -d 5 –high characters_test_pattern.tif

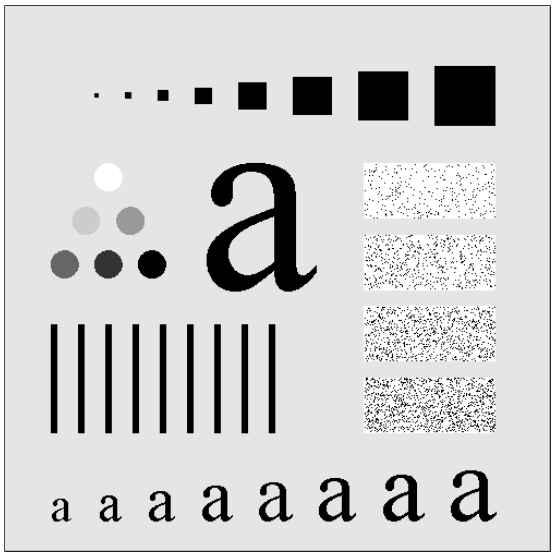

Figure C.6 – Original image

Figure C.7 – Ideal high 5

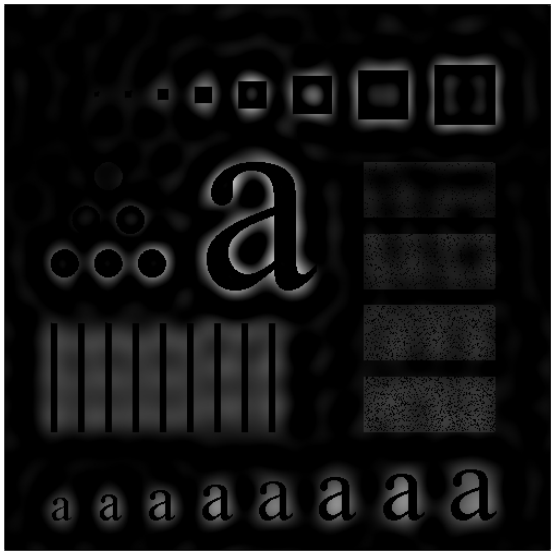

Figure C.8 – Ideal high 15

Figure C.9 – Ideal high 30

C.3.3 Butterworth order 2 filter

Low pass

python problem3.py –butterworth -d 5 -n 2 –low characters_test_pattern.tif

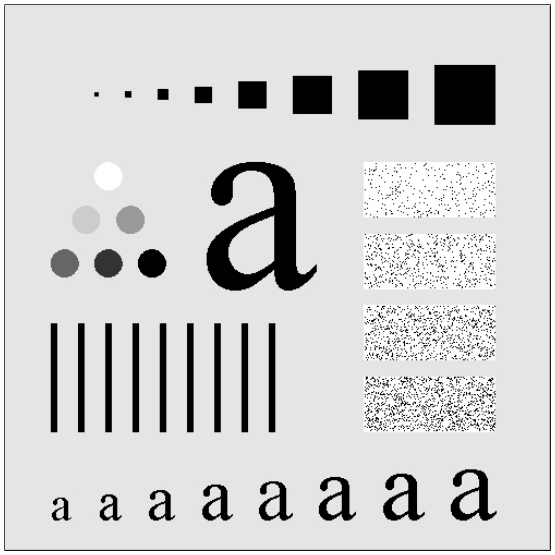

Figure C.10 – Original image

Figure C.11 – Butterworth low 5

Figure C.12 – Butterworth low 15

Figure C.13 – Butterworth low 30

High pass

python problem3.py –butterworth -d 5 -n 2 –high characters_test_pattern.tif

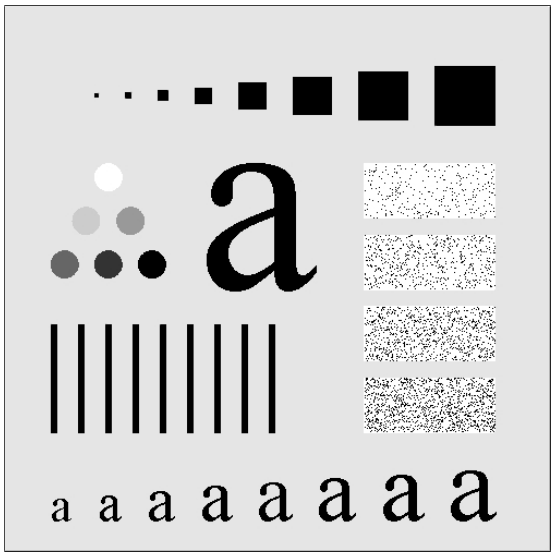

Figure C.14 – Original image

Figure C.15 – Butterworth high 5

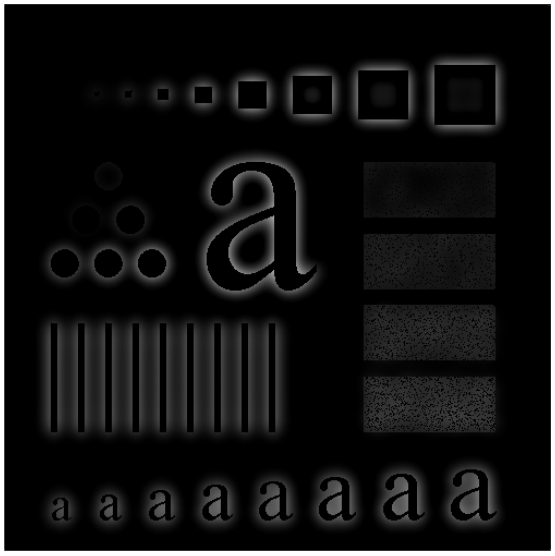

Figure C.16 – Butterworth high 15

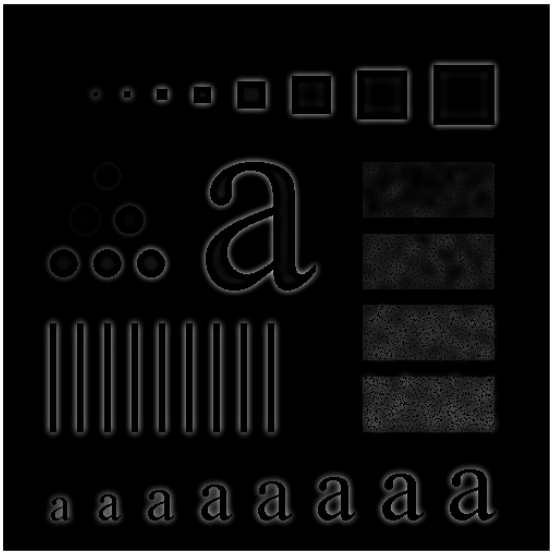

Figure C.17 – Butterworth high 30

C.3.4 Butterworth order 5 filter

Low pass

python problem3.py –butterworth -d 5 -n 5 –low characters_test_pattern.tif

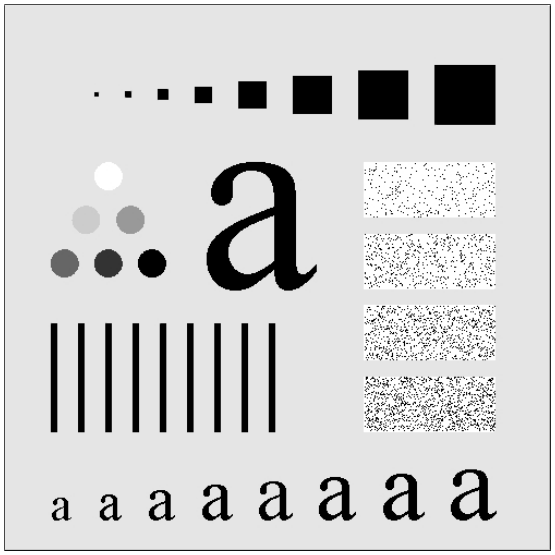

Figure C.18 – Original image

Figure C.19 – Butterworth low 5

Figure C.20 – Butterworth low 15

Figure C.21 – Butterworth low 30

High pass

python problem3.py –butterworth -d 5 -n 5 –high characters_test_pattern.tif

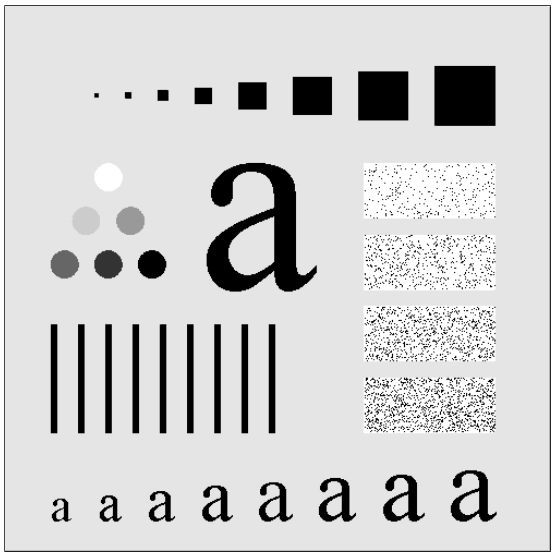

Figure C.22 – Original image

Figure C.23 – Butterworth high 5

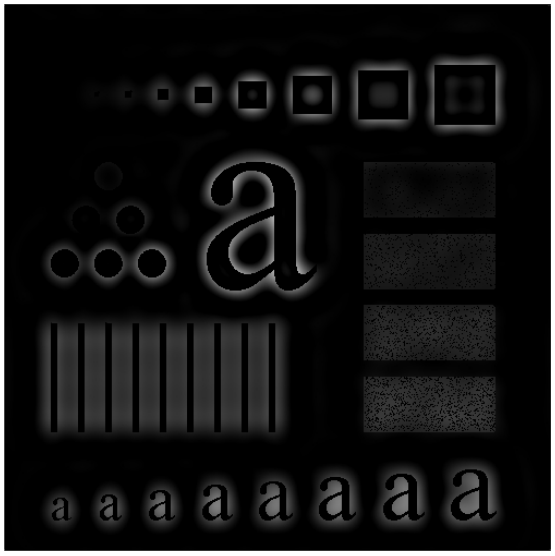

Figure C.24 – Butterworth high 15

Figure C.25 – Butterworth high 30

C.3.5 Gaussian filter

Low pass

python problem3.py –gaussian -d 5 –low characters_test_pattern.tif

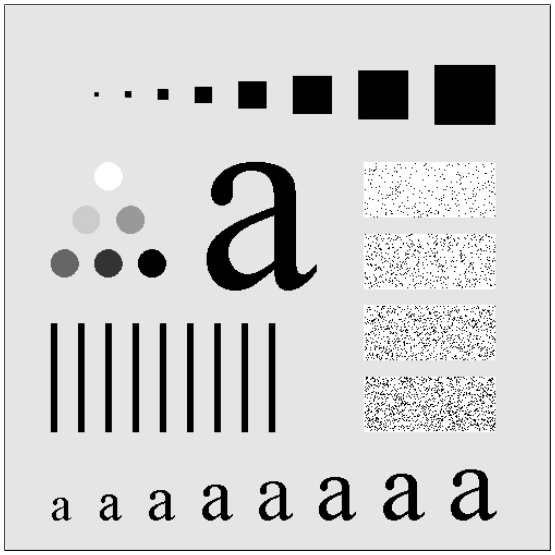

Figure C.26 – Original image

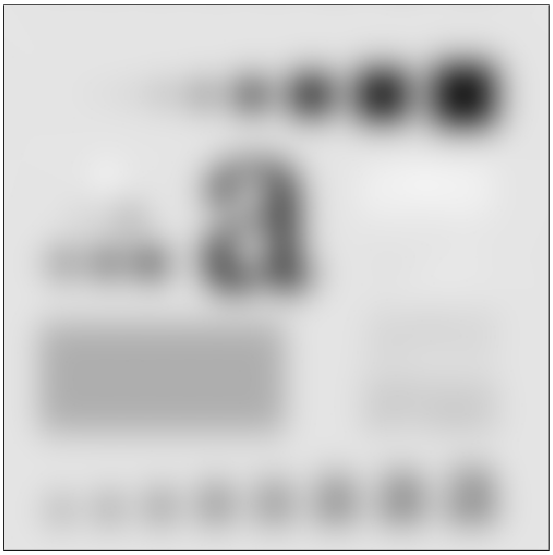

Figure C.27 – Gaussian low 5

Figure C.28 – Gaussian low 15

Figure C.29 – Gaussian low 30

High pass

python problem3.py –gaussian -d 5 –high characters_test_pattern.tif

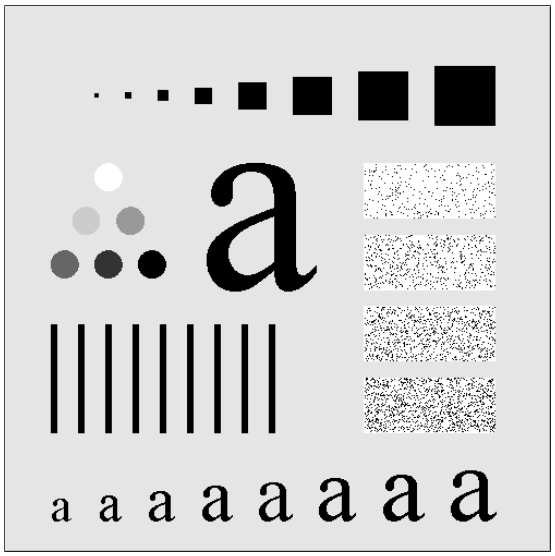

Figure C.30 – Original image

Figure C.31 – Gaussian high 5

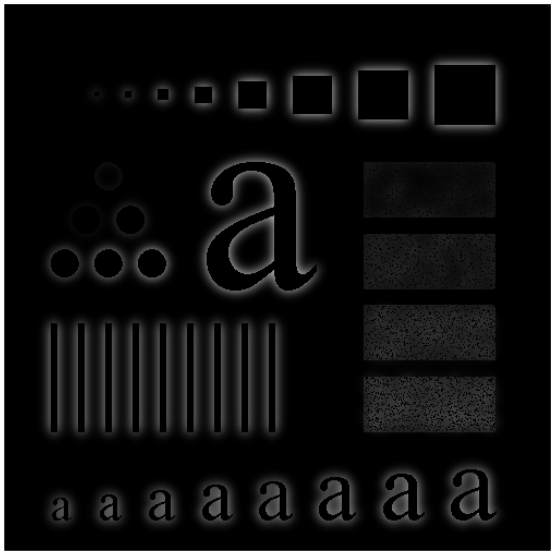

Figure C.32 – Gaussian high 15

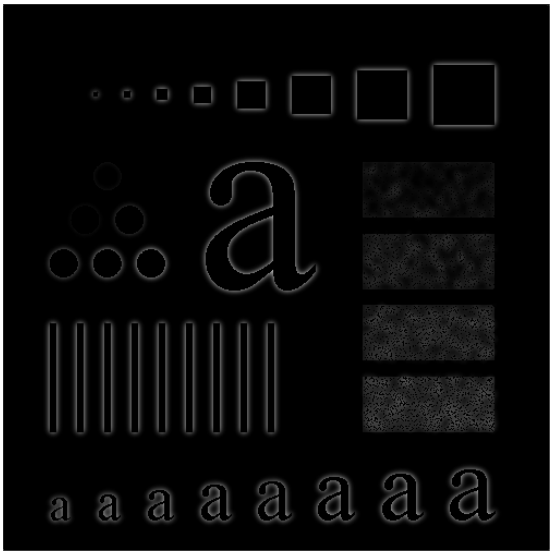

Figure C.33 – Gaussian high 30

D. Noise generation and noise reduction

D.1 Problem statement

In this problem, you are required to write a program to generate different types of random noises started from the Uniform noise and Gaussian noise.
And then add some of these noises to the circuit image and investigate the different mean filters and order statistics as the textbook did at pages 344-352.

D.2 Python implementation

Usage : **problem4.py [-h] (–uniform | –gaussian) [–histogram] [–arithmetic] [–geometric] [–harmonic] [–contraharmonic] [-q Q] [–median] [–max] [–min] [–midpoint] [–alpha] [-d D] image_path**

Use **python problem4.py -h** to see the help.

D.3 Gaussian noise

D.3.1 Noise generation and histogram

python problem4.py –gaussian circuit.tif

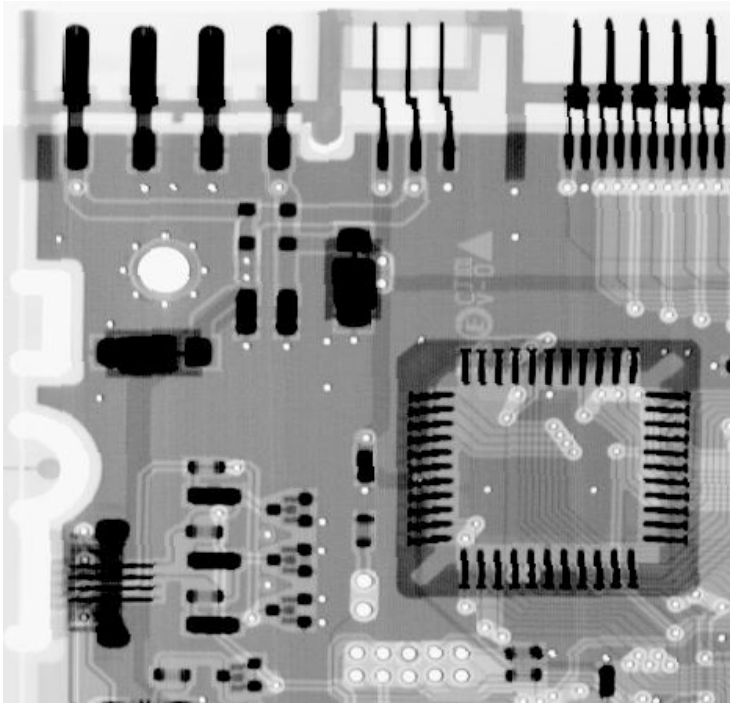

Figure D.1 – Original image

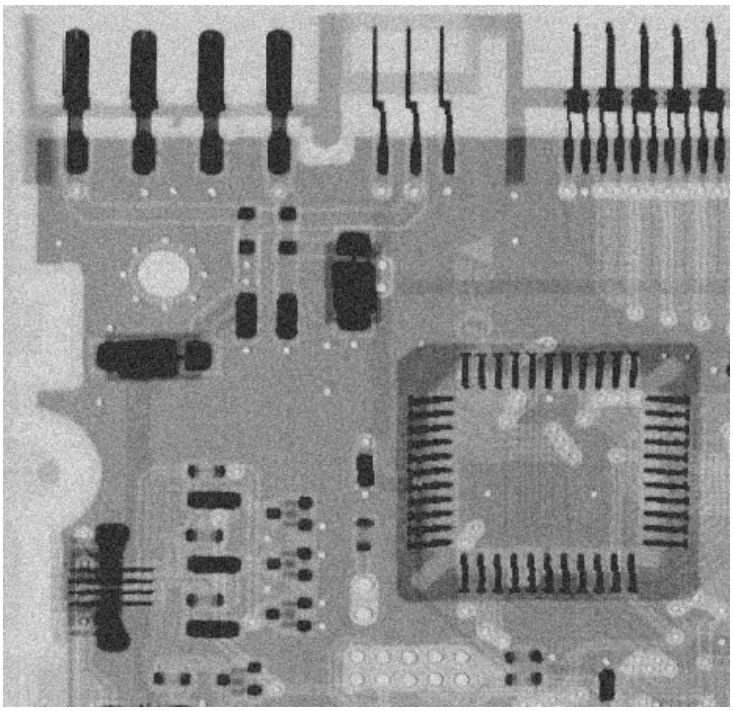

Figure D.2 – Image + Gaussian noise ($\mu = 0, \sigma = 20$)

python problem4.py –gaussian –histogram circuit.tif

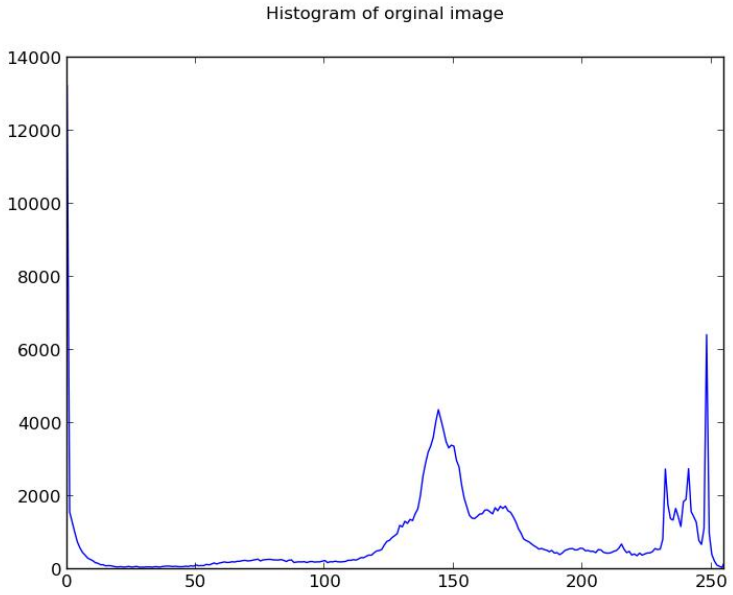

Figure D.3 – Histogram of original image

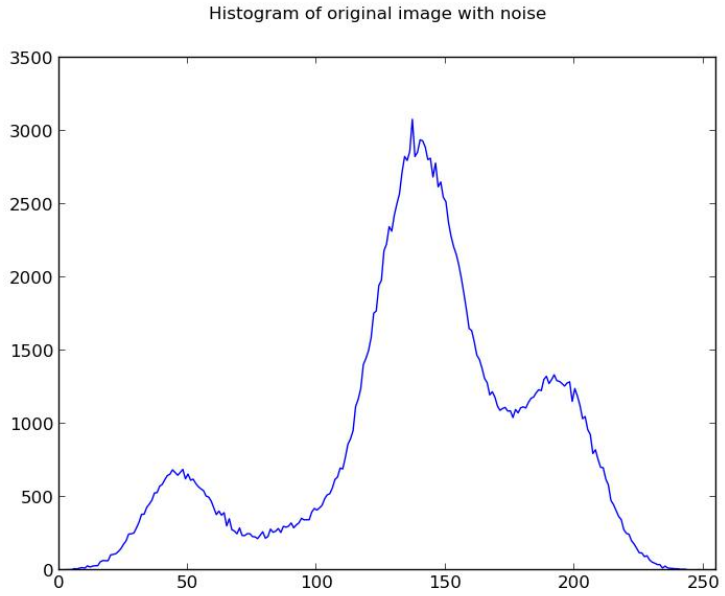

Figure D.4 – Histogram of image with Gaussian noise

D.3.2 Noise reduction

Mean filters

python problem4.py –gaussian –arithmetic circuit.tif
python problem4.py –gaussian –geometric circuit.tif

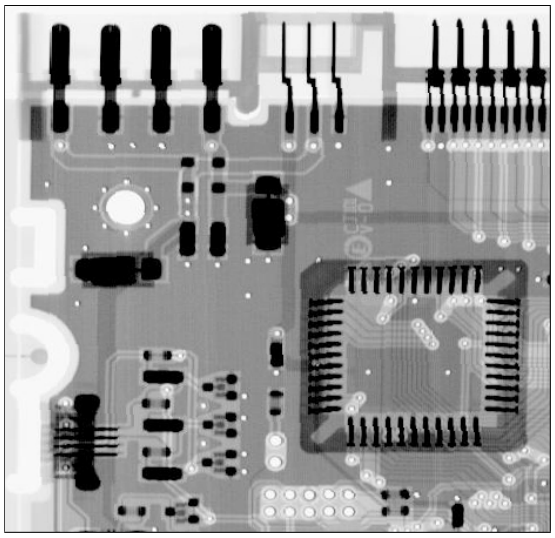

Figure D.5 – Original image

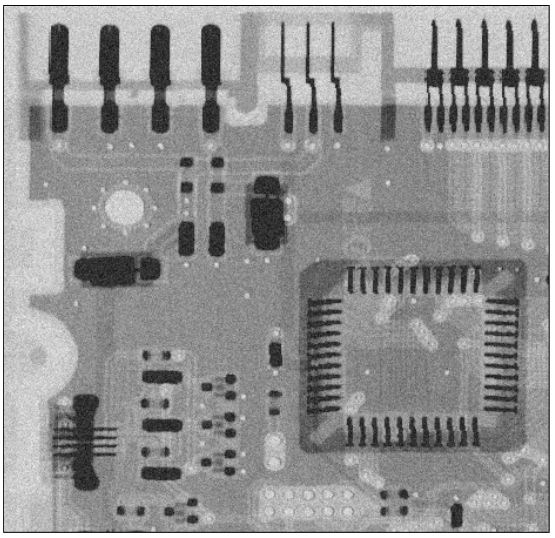

Figure D.6 – Image + Gaussian

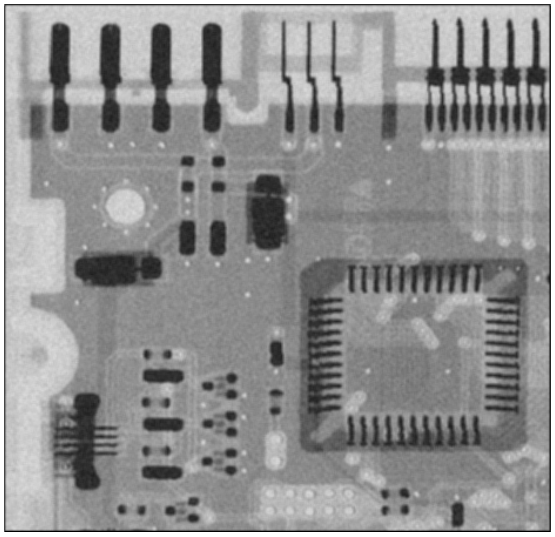

Figure D.7 – Arithmetic filter

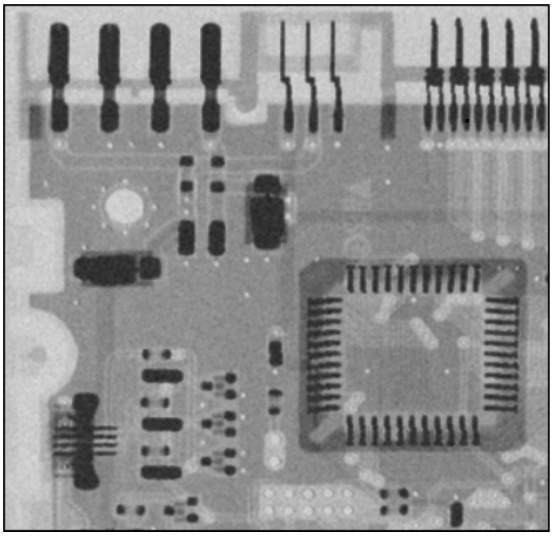

Figure D.8 – Geometric filter

python problem4.py –gaussian –harmonic circuit.tif

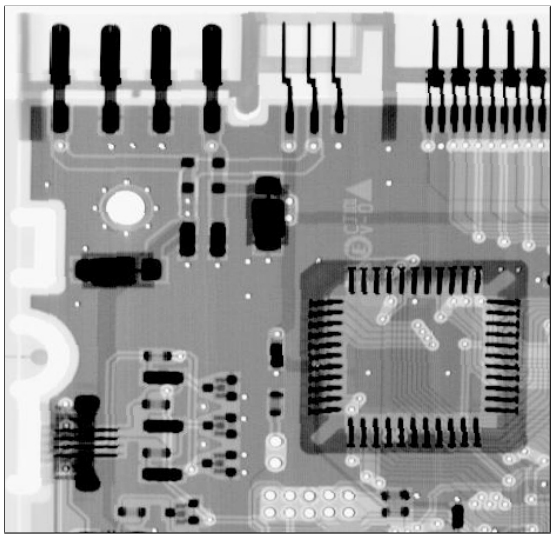

Figure D.9 – Original image

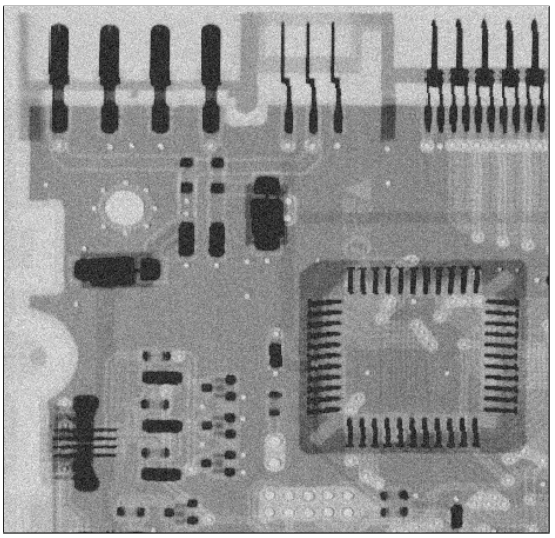

Figure D.10 – Image + Gaussian

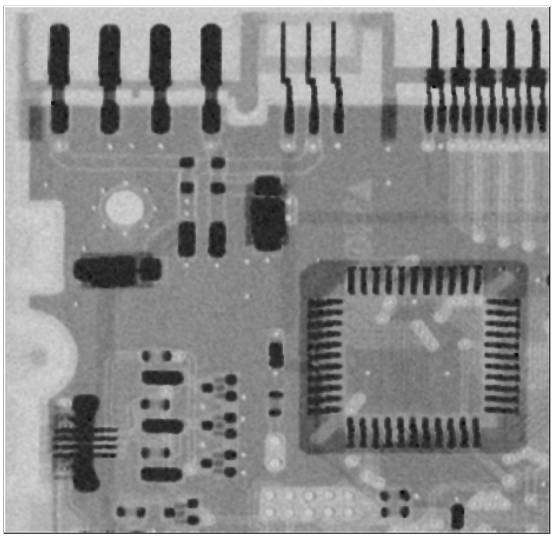

Figure D.11 – Harmonic filter

python problem4.py –gaussian –contraharmonic -q 1.5 circuit.tif
python problem4.py –gaussian –contraharmonic -q -1.5 circuit.tif

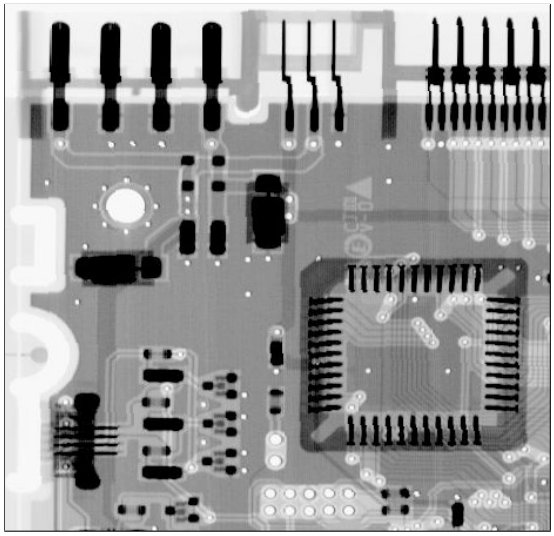

Figure D.12 – Original image

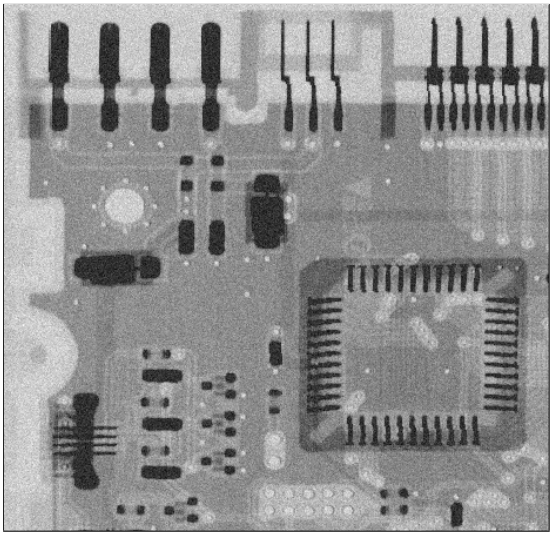

Figure D.13 – Image + Gaussian

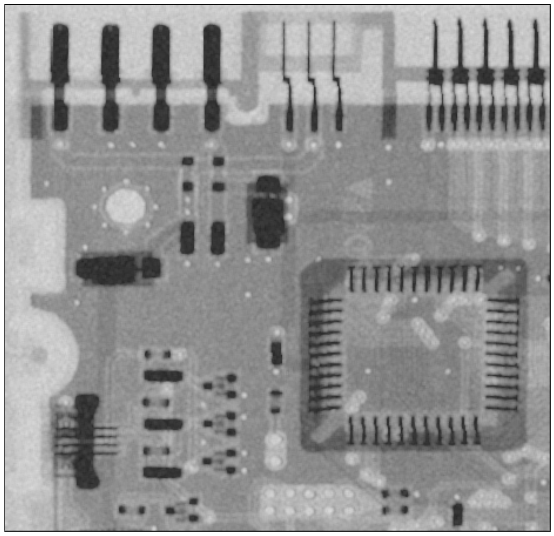

Figure D.14 – Contra-harmonic filter (Q=1.5)

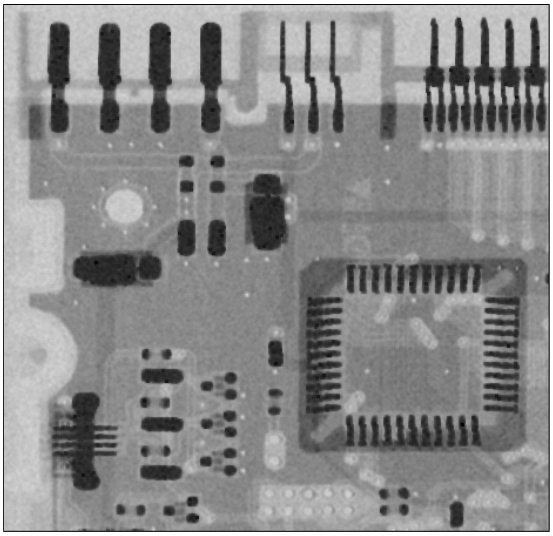

Figure D.15 – Contra-harmonic filter (Q=-1.5)

Order-Statistic filters

python problem4.py –gaussian –median circuit.tif

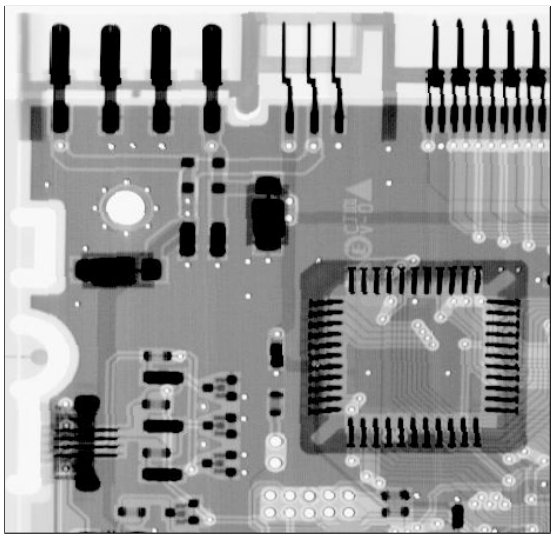

Figure D.16 – Original image

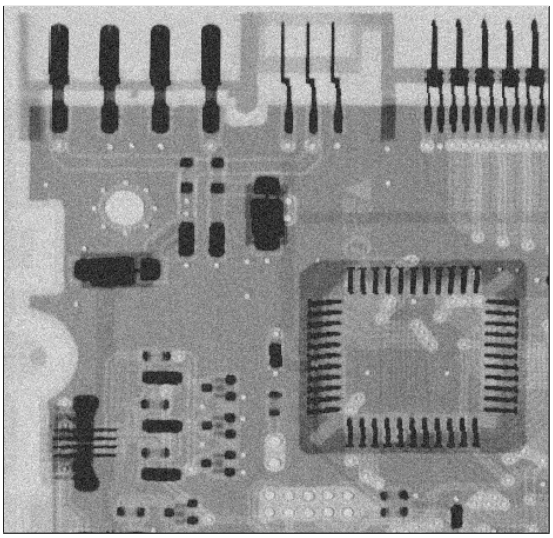

Figure D.17 – Image + Gaussian

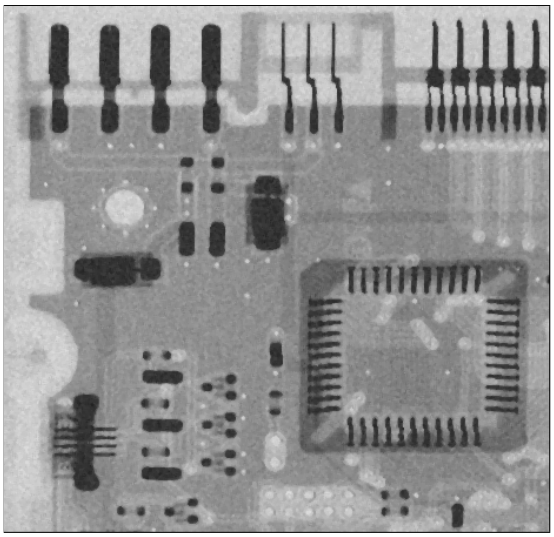

Figure D.18 – Median filter

python problem4.py –gaussian –max circuit.tif
python problem4.py –gaussian –min circuit.tif

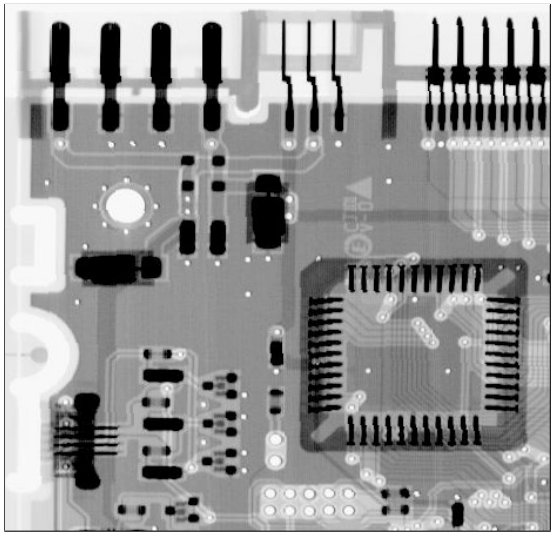

Figure D.19 – Original image

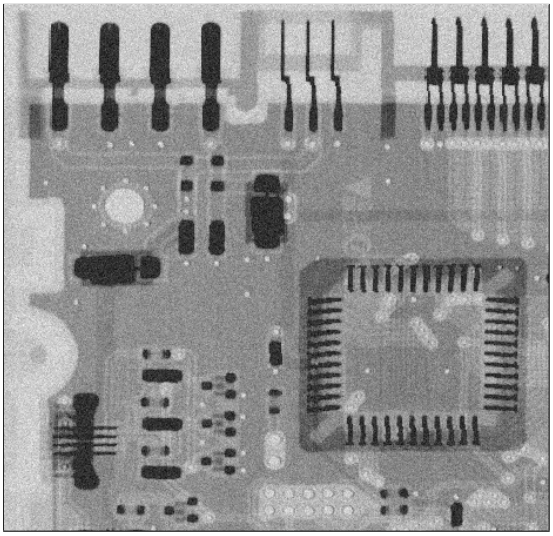

Figure D.20 – Image + Gaussian

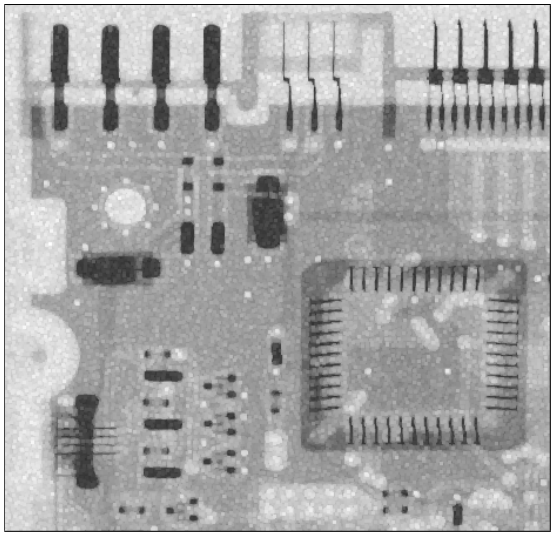

Figure D.21 – Max filter

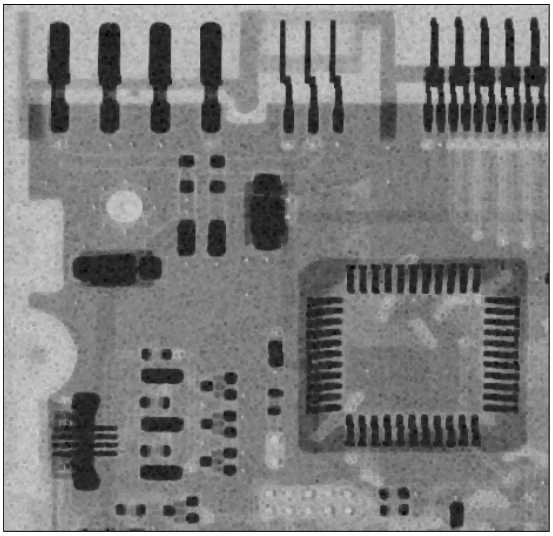

Figure D.22 – Min filter

python problem4.py –gaussian –midpoint circuit.tif

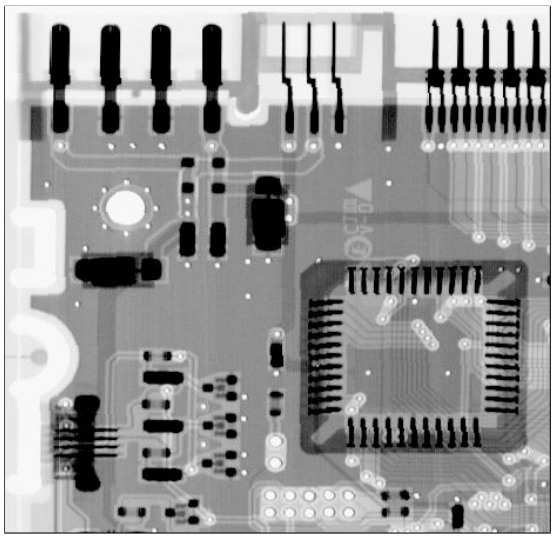

Figure D.23 – Original image

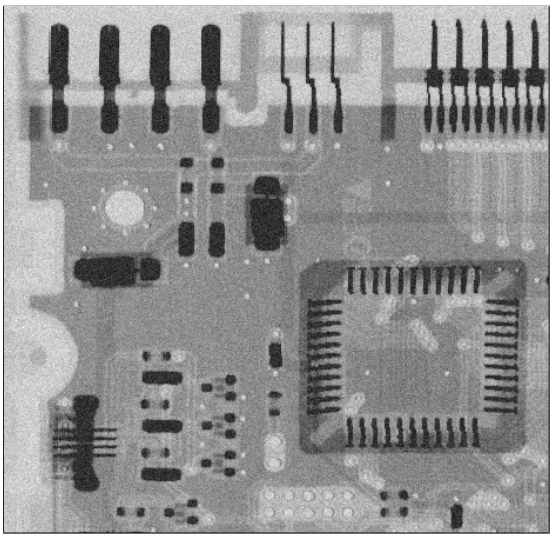

Figure D.24 – Image + Gaussian

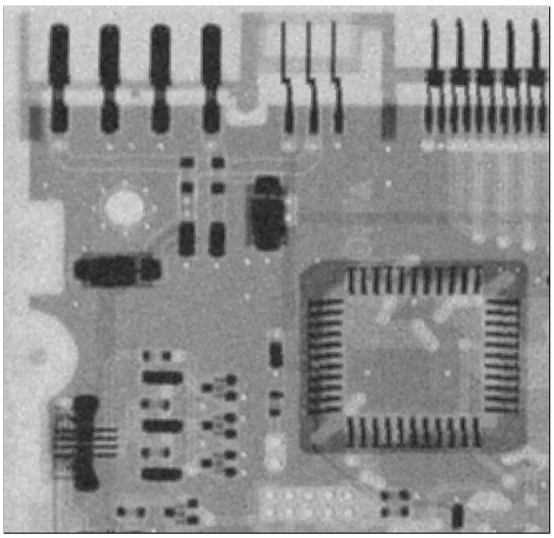

Figure D.25 – Midpoint filter

python problem4.py –gaussian –alpha -d 2 circuit.tif
python problem4.py –gaussian –alpha -d 4 circuit.tif

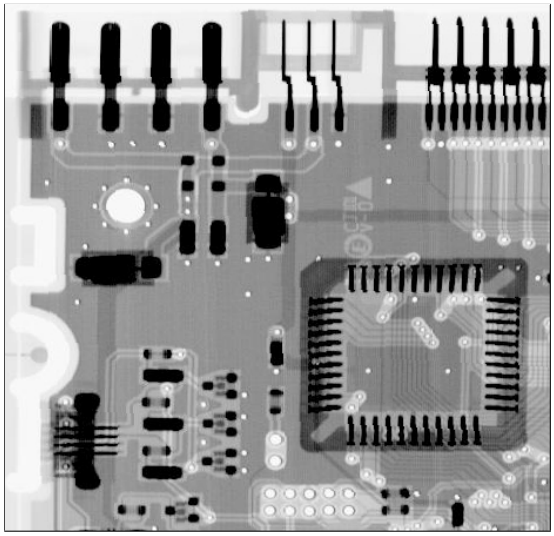

Figure D.26 – Original image

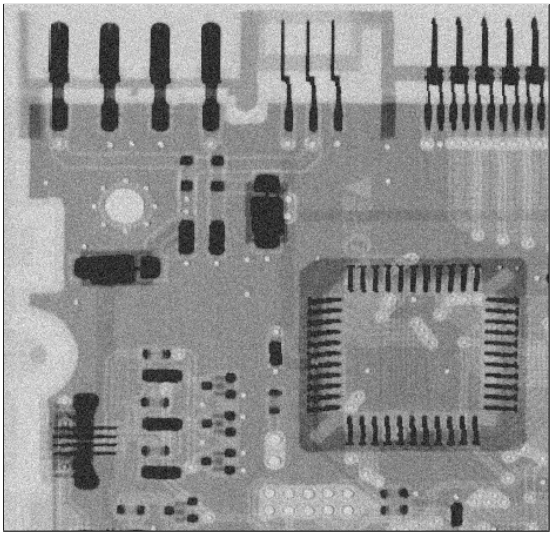

Figure D.27 – Image + Gaussian

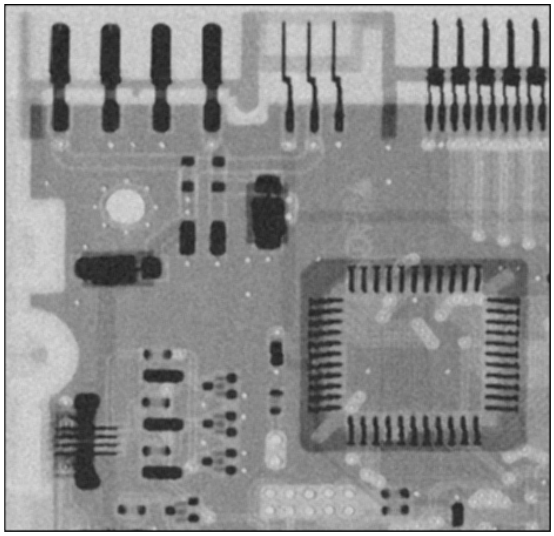

Figure D.28 – Alpha-trimmed filter (d = 2)

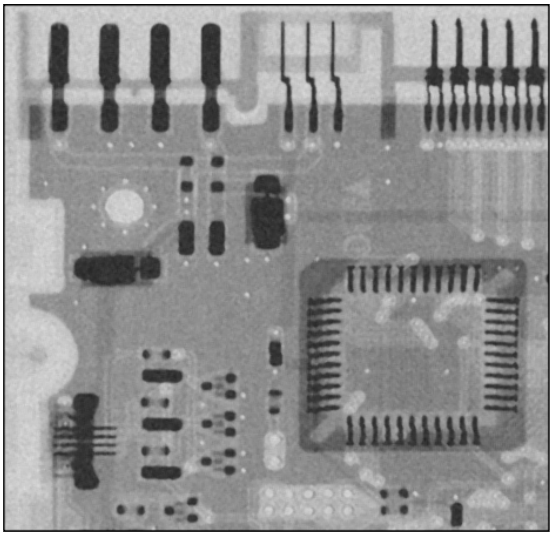

Figure D.29 – Alpha-trimmed filter (d = 4)

D.4 Uniform noise

D.4.1 Noise generation and histogram

python problem4.py –uniform circuit.tif

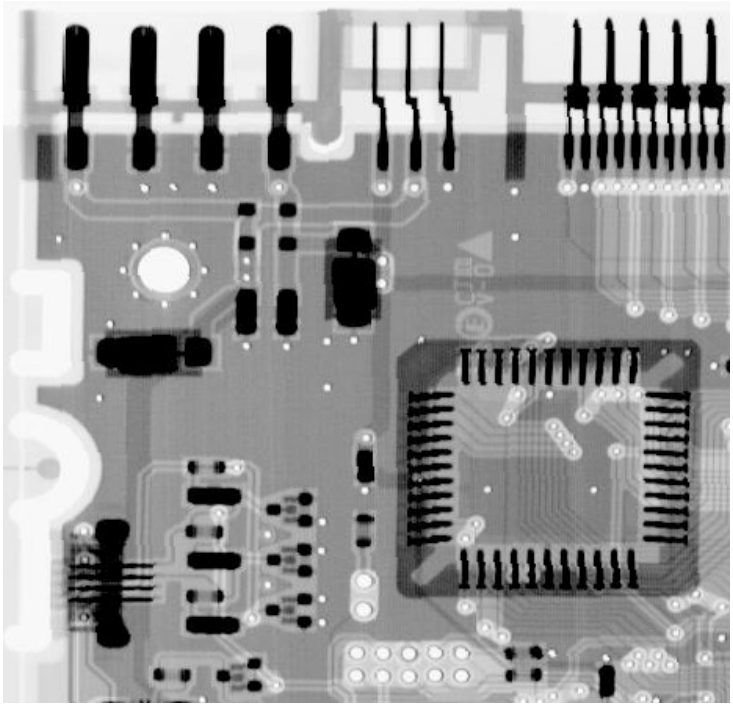

Figure D.30 – Original image

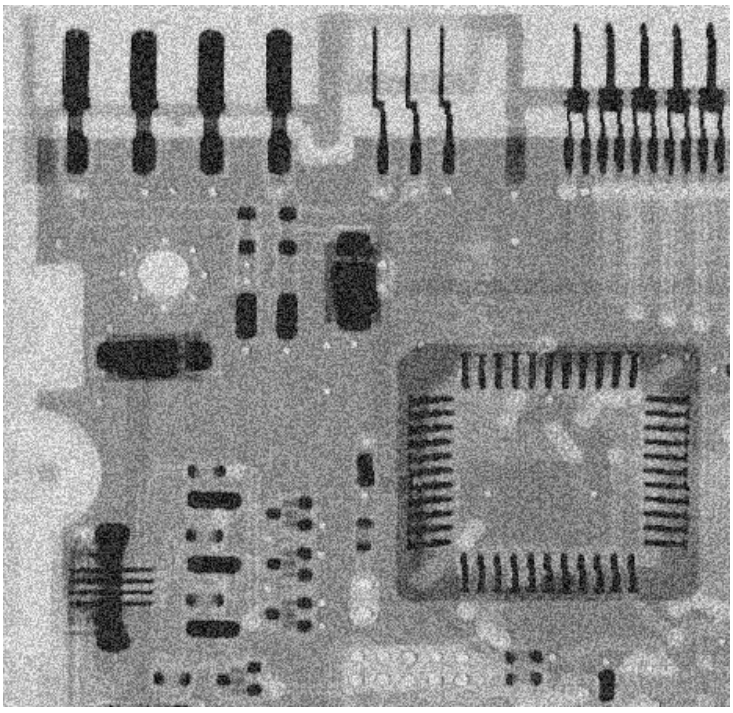

Figure D.31 – Image + Uniform noise (min = 0, max = 128)

python problem4.py –uniform –histogram circuit.tif

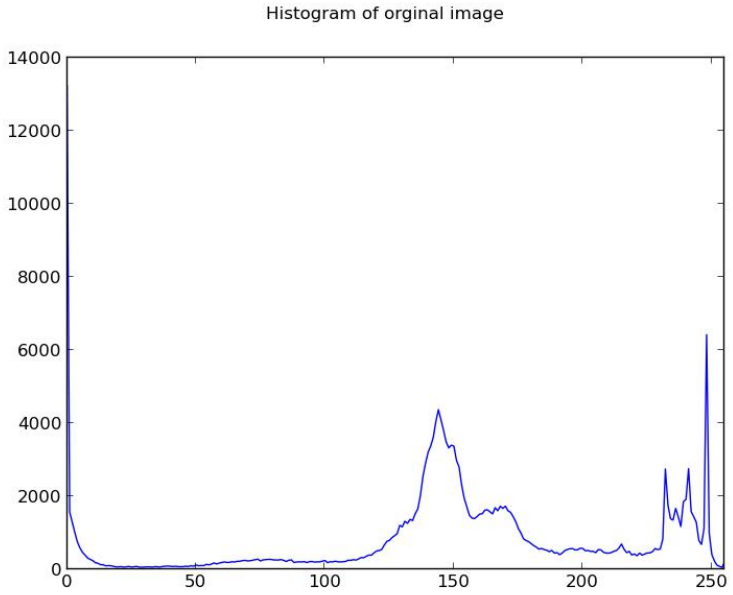

Figure D.32 – Histogram of original image

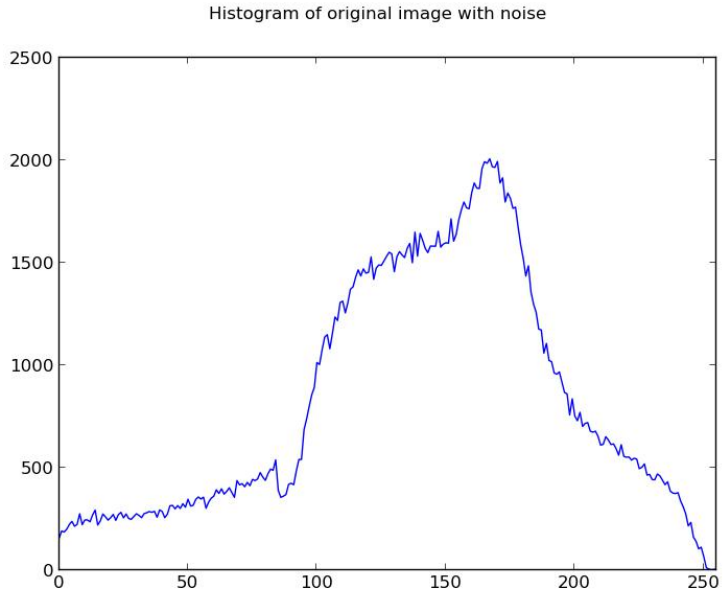

Figure D.33 – Histogram of image with Uniform noise

D.4.2 Noise reduction

Mean filters

python problem4.py –uniform –arithmetic circuit.tif
python problem4.py –uniform –geometric circuit.tif

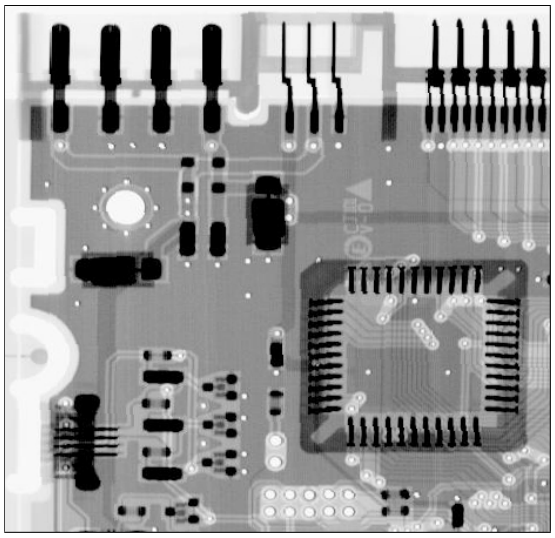

Figure D.34 – Original image

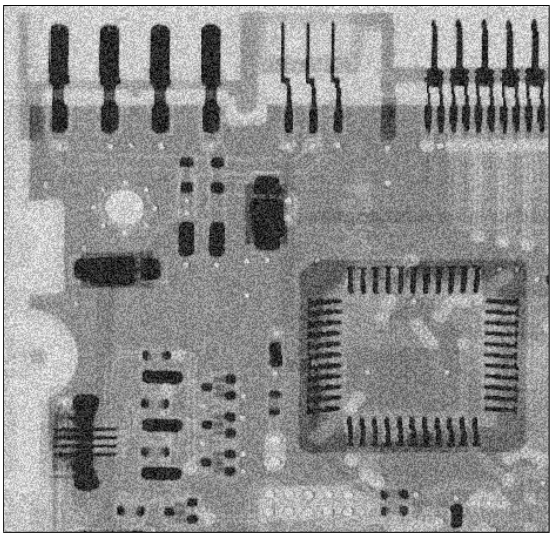

Figure D.35 – Image + Uniform

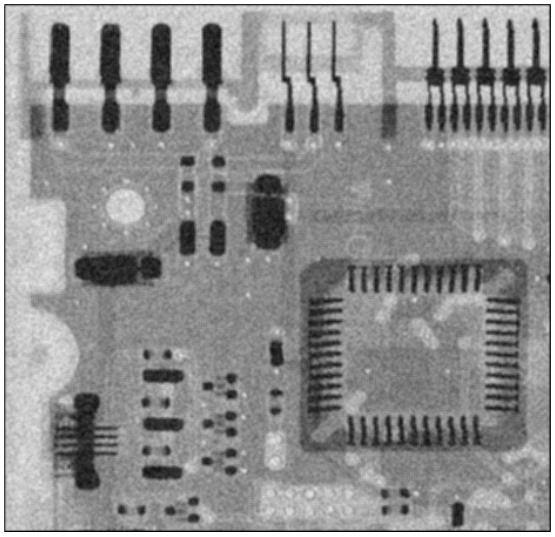

Figure D.36 – Arithmetic filter

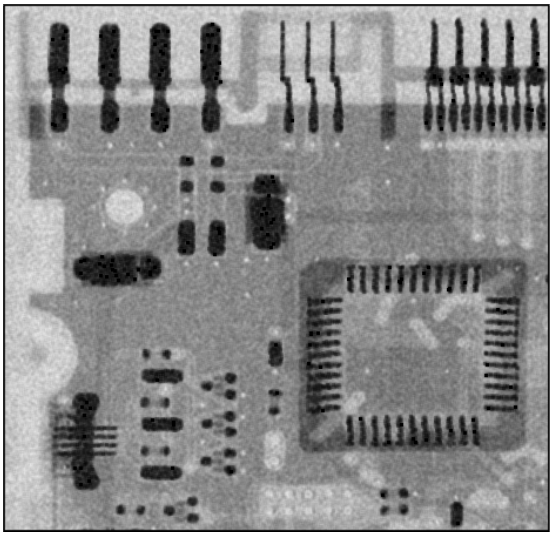

Figure D.37 – Geometric filter

python problem4.py –uniform –harmonic circuit.tif

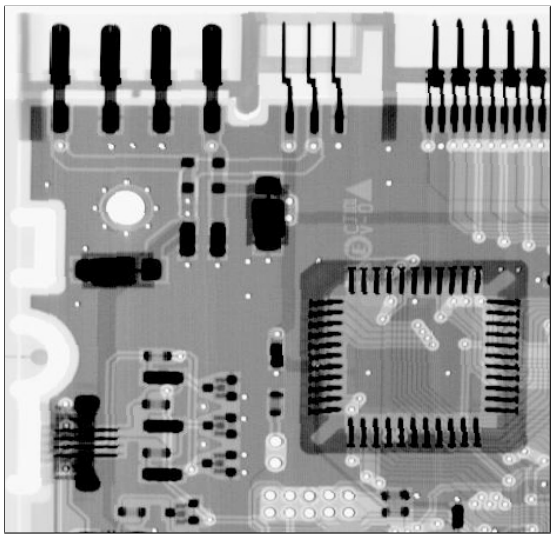

Figure D.38 – Original image

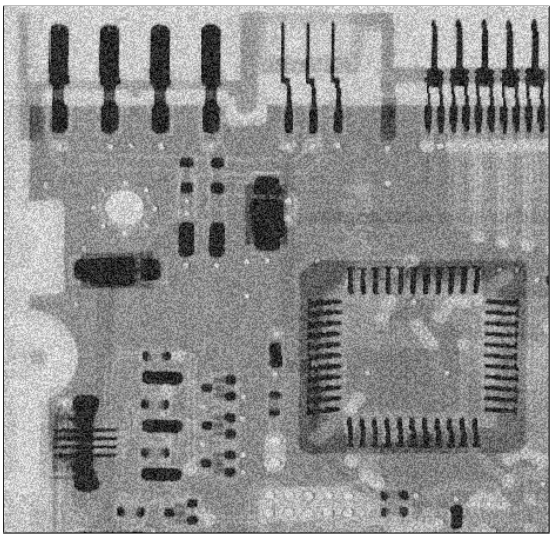

Figure D.39 – Image + Uniform

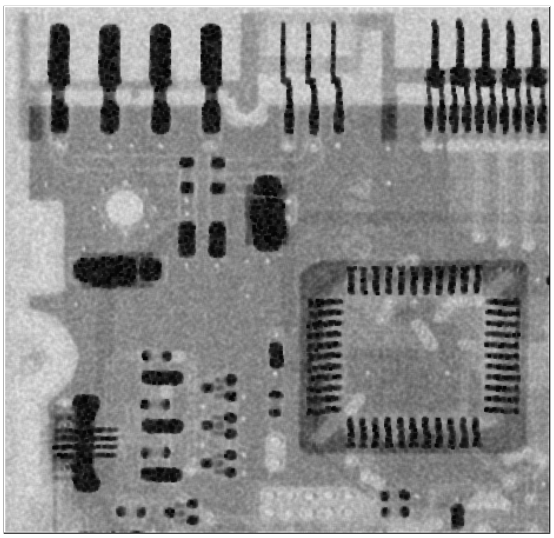

Figure D.40 – Harmonic filter

python problem4.py –uniform –contraharmonic -q 1.5 circuit.tif
python problem4.py –uniform –contraharmonic -q -1.5 circuit.tif

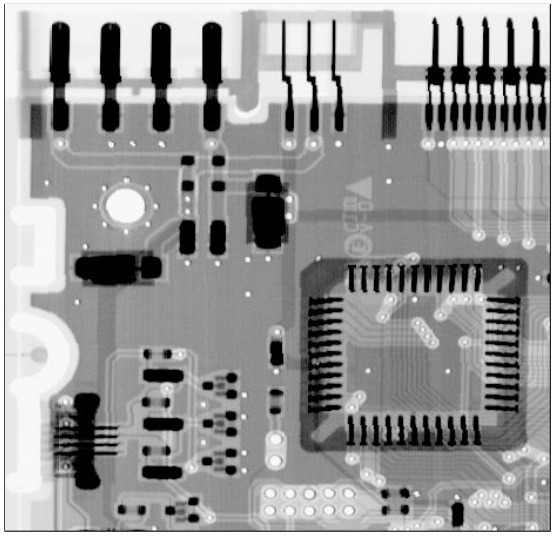

Figure D.41 – Original image

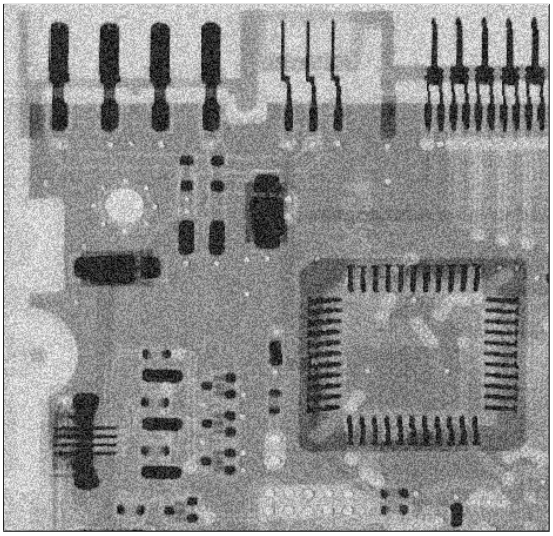

Figure D.42 – Image + Uniform

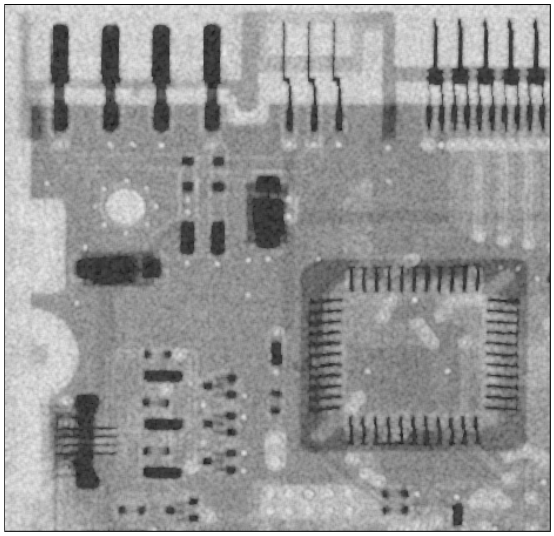

Figure D.43 – Contra-harmonic filter (Q=1.5)

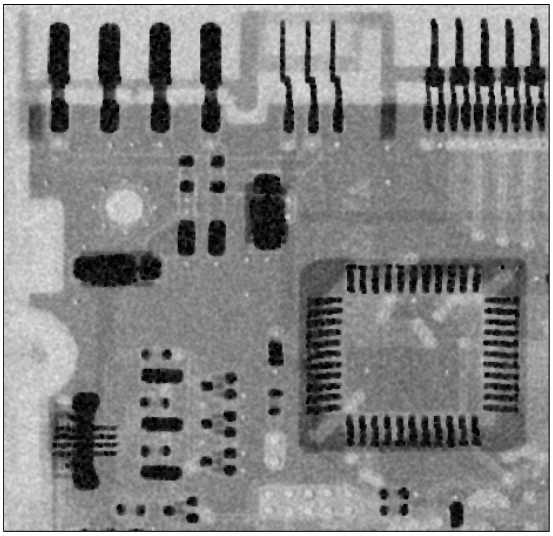

Figure D.44 – Contra-harmonic filter (Q=-1.5)

Order-Statistic filters

python problem4.py –uniform –median circuit.tif

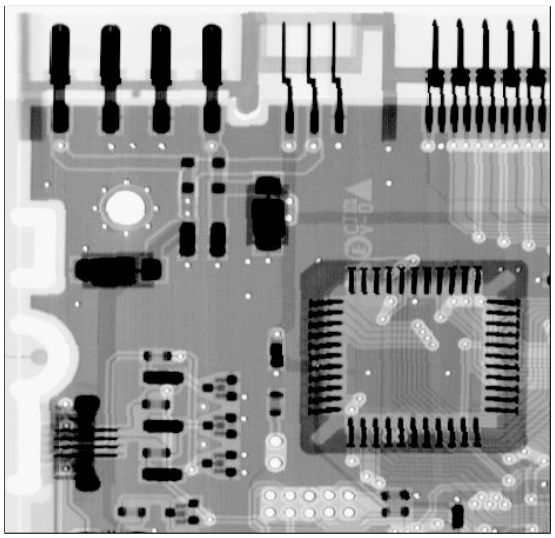

Figure D.45 – Original image

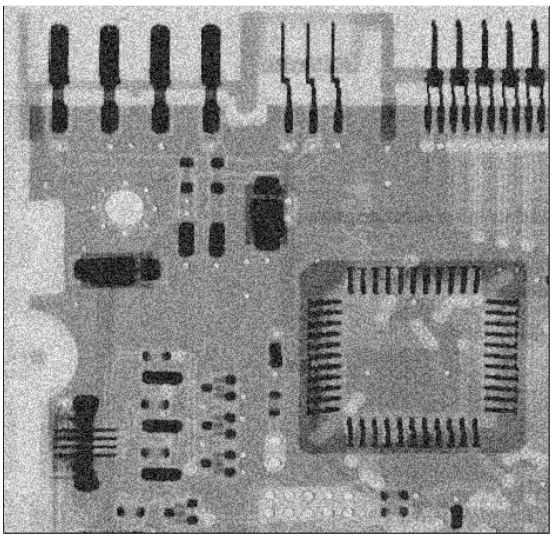

Figure D.46 – Image + Uniform

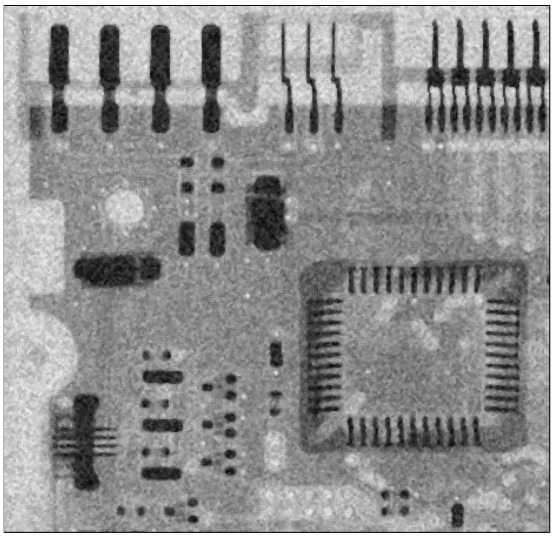

Figure D.47 – Median filter

python problem4.py –uniform –max circuit.tif
python problem4.py –uniform –min circuit.tif

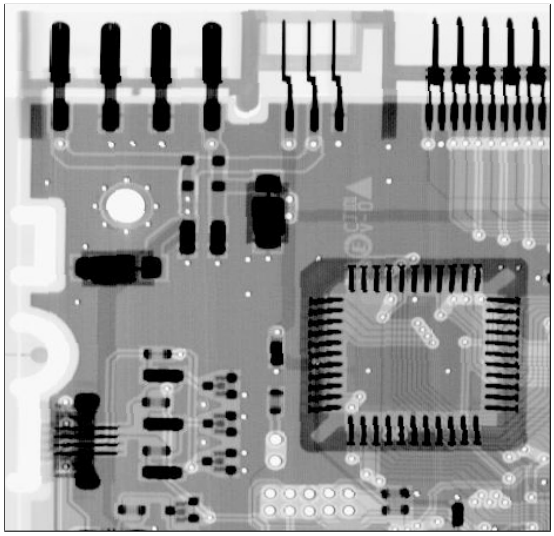

Figure D.48 – Original image

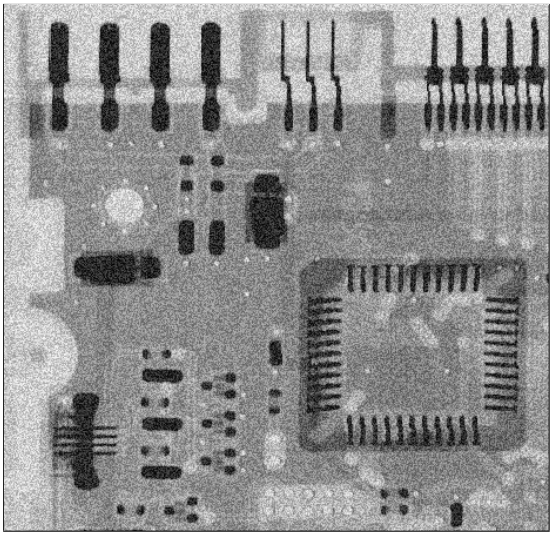

Figure D.49 – Image + Uniform

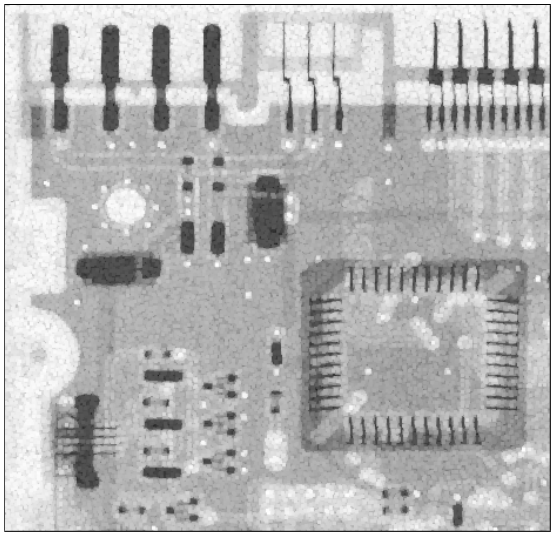

Figure D.50 – Max filter

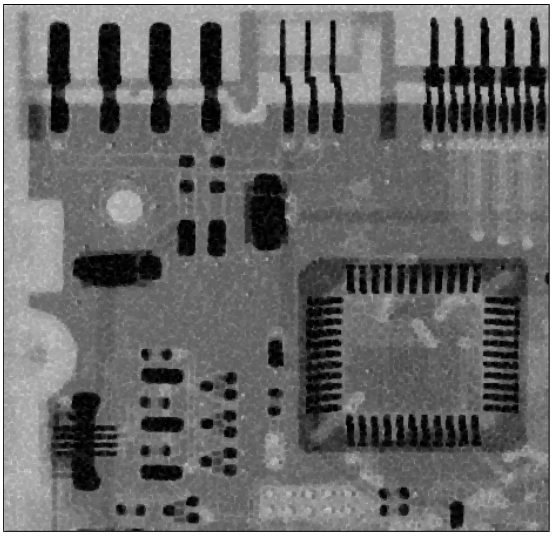

Figure D.51 – Min filter

python problem4.py –uniform –midpoint circuit.tif

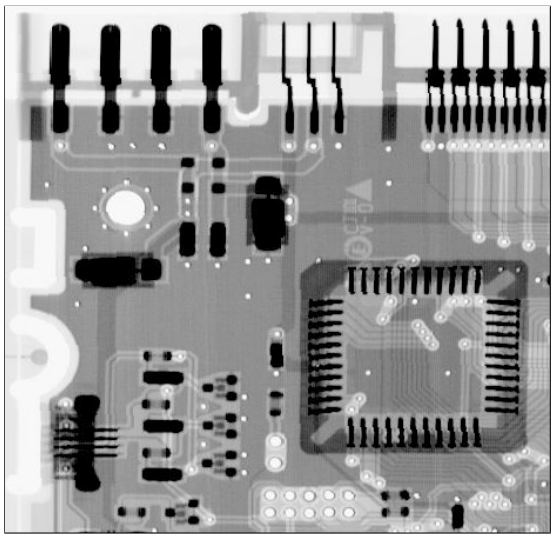

Figure D.52 – Original image

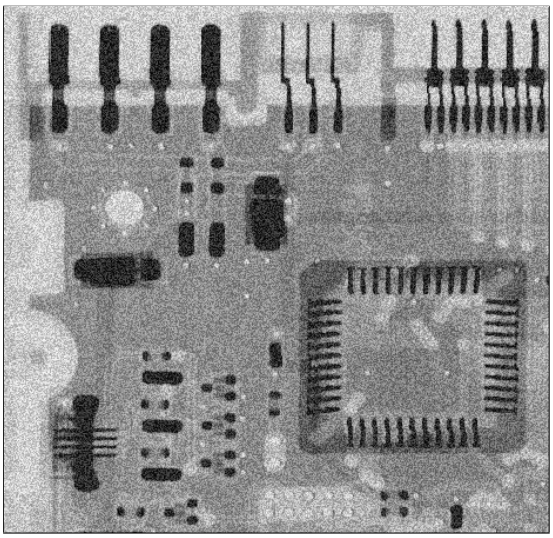

Figure D.53 – Image + Uniform

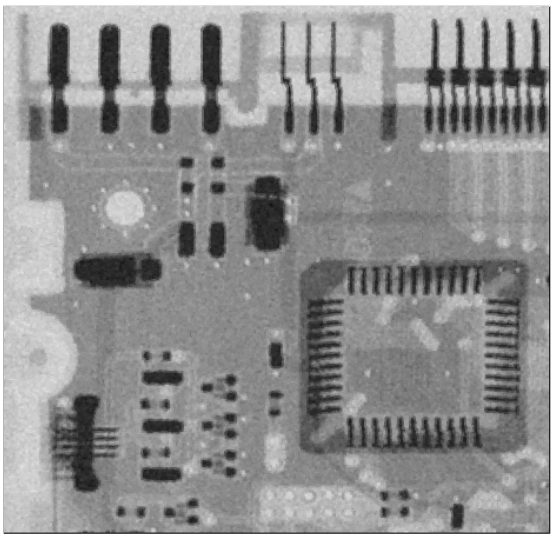

Figure D.54 – Midpoint filter

python problem4.py –uniform –alpha -d 2 circuit.tif
python problem4.py –uniform –alpha -d 4 circuit.tif

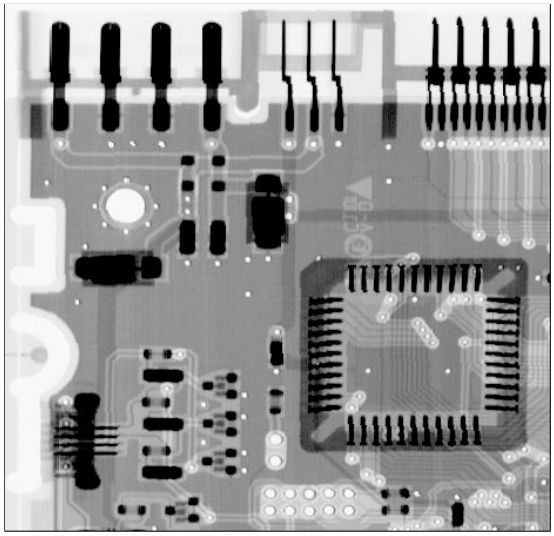

Figure D.55 – Original image

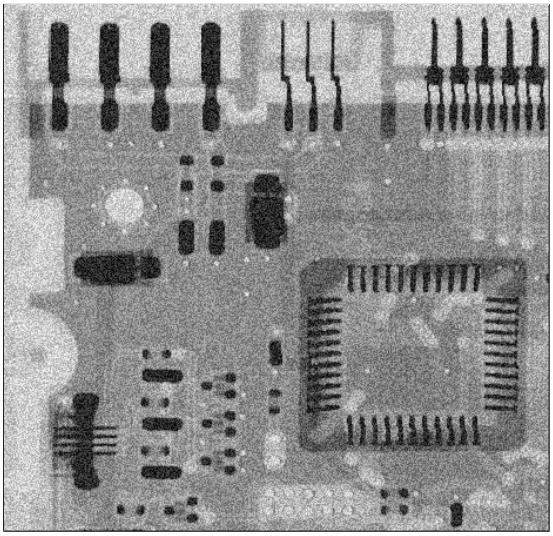

Figure D.56 – Image + Uniform

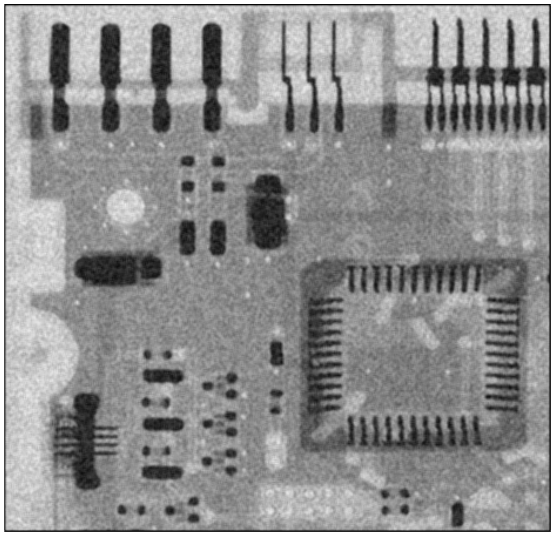

Figure D.57 – Alpha-trimmed filter (d = 2)

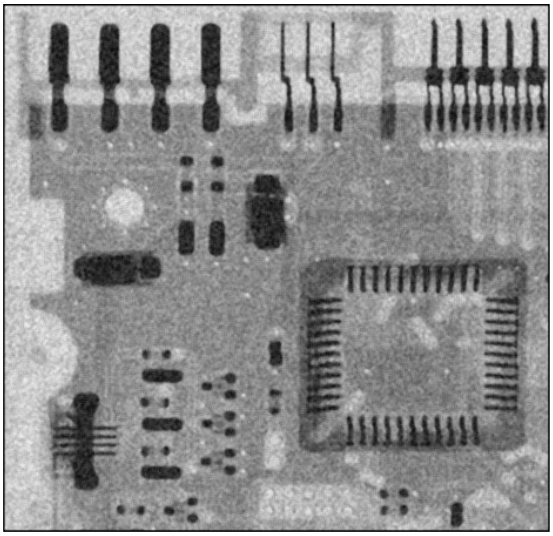

Figure D.58 – Alpha-trimmed filter (d = 4)

E. Image restoration

E.1 Problem statement

Suppose a blurring degradation function as

$$H(u,v) = T \times sinc(ua + vb) \times e^{-j\pi(ua+vb)} \tag{E.1}$$

(a) Implement a blurring filter using the equation above.

(b) Blur the test image book_cover.jpg using parameters $a = b = 0.1$ and $T = 1$.

(c) Add Gaussian noise of 0 mean and variance of 650 to the blurred image.

(d) Restore the blurred image and the blurred noisy image using the inverse filter, Wiener deconvolution filter and the parametric Wiener filter, respectively.

(e) Add Gaussian noise of 0 and different variances to the blurred image and repeat (d), investigate the performance of the Wiener deconvolution filter.

E.2 Python implementation

Usage : **problem5.py [-h] [–blurred BLURRED] [–noise] [-s S] [–inv] [–wiener] [-a A] [-b B] [-T T] [-K K] image_path**

Use **python problem5.py -h** to see the help.

E.3 Blurring and noising

python problem5.py book_cover.jpg

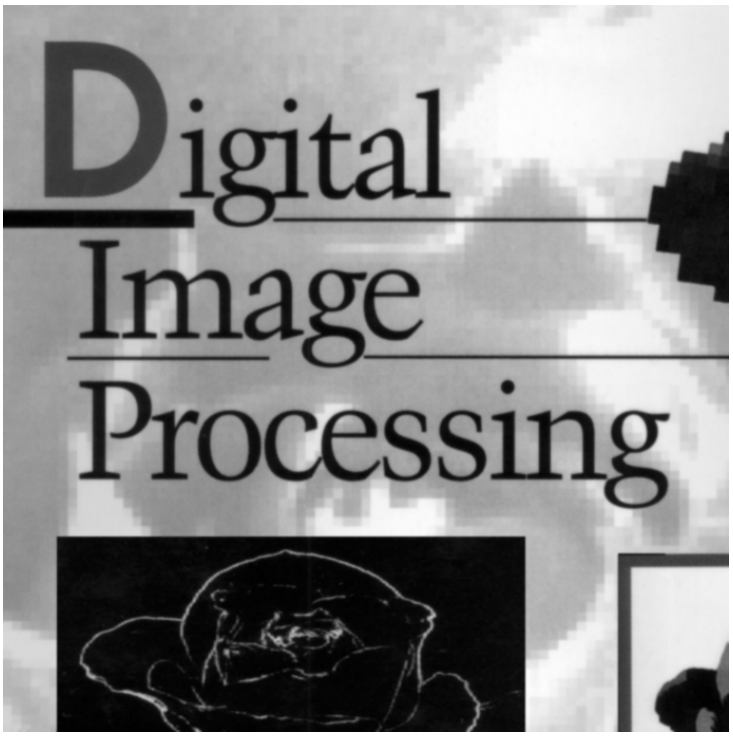

Figure E.1 – Original image

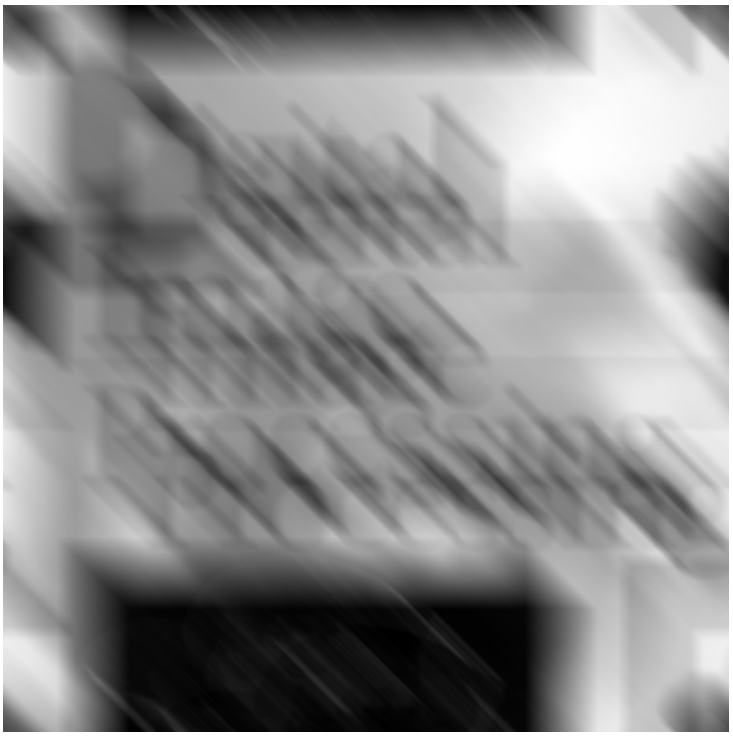

Figure E.2 – Blurred image ($a = b = 0.1; T = 1$)

python problem5.py book_cover.jpg –noise -s 650

Figure E.3 – Blur + Gaussian noise ($\mu = 0, \sigma^2 = 650$)

E.4 Restoration

E.4.1 Inverse filter

python problem5.py –inv book_cover.jpg
python problem5.py –noise -s 650 –inv book_cover.jpg

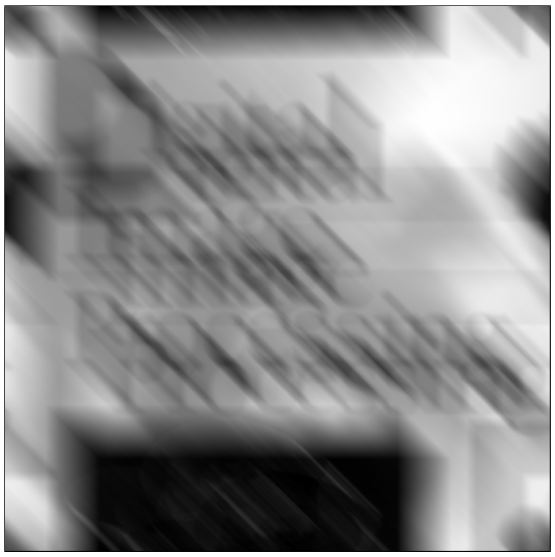

Figure E.4 – Blurred image

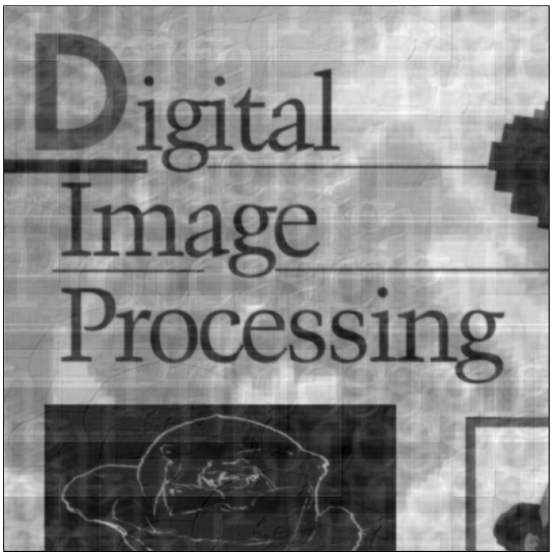

Figure E.5 – Inverse blurred image

Figure E.6 – Blur + noise

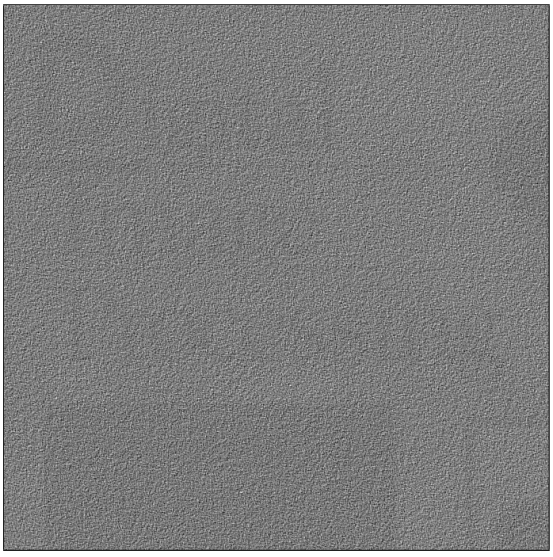

Figure E.7 – Inverse 'blur + noise'

The inverse filter is really affected by the noise because it ignores it (does not inverse it).

E.4.2 Wiener filter

python problem5.py –wiener book_cover.jpg
python problem5.py –noise -s 650 –wiener book_cover.jpg

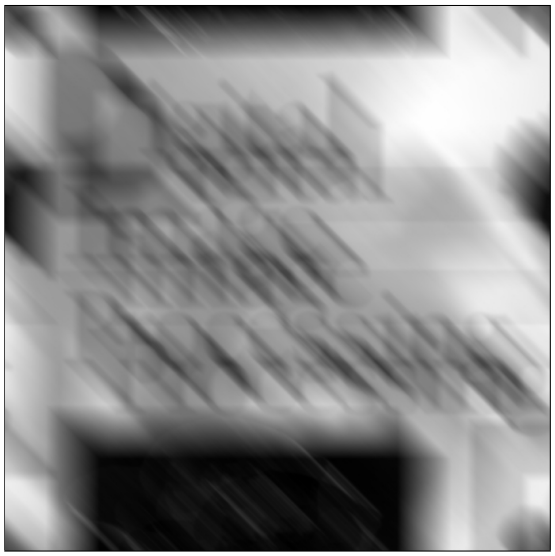

Figure E.8 – Blurred image

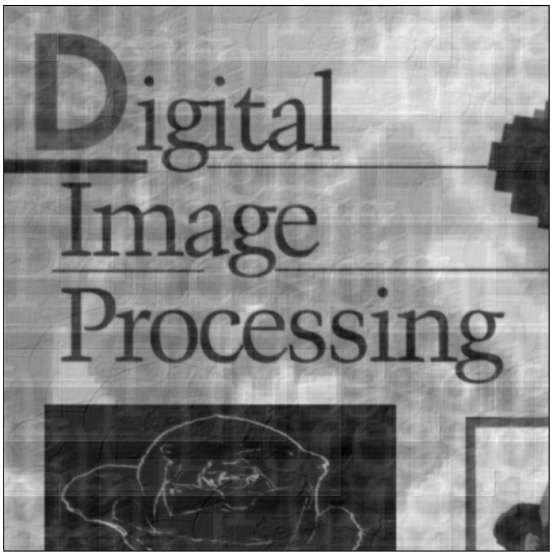

Figure E.9 – Wiener blurred image

Figure E.10 – Blur + noise

Figure E.11 – Wiener 'blur + noise'

The Wiener filter on the blurred image without noise is equivalent to the inverse filter (power spectrum of noise equals to 0).
The Wiener filter is more efficient to remove the noise as it takes it into consideration during the restoration process.

E.4.3 Experimentations

Gaussian noise ($\mu = 0, \sigma^2 = 300$)

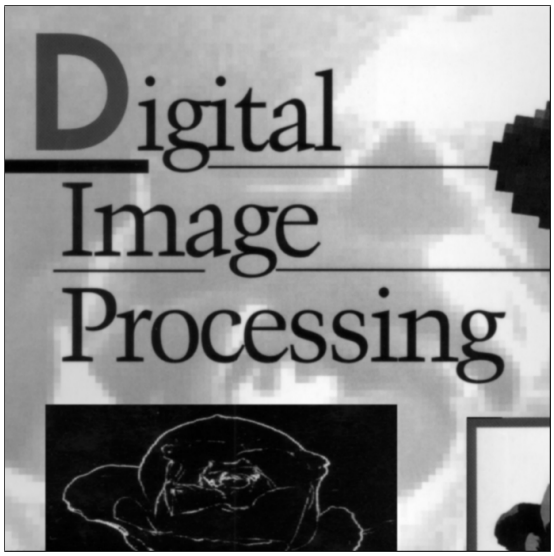

Figure E.12 – Original image

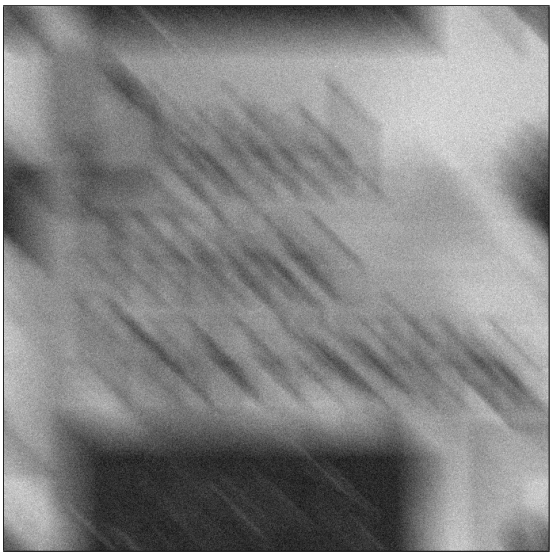

Figure E.13 – Blur + noise

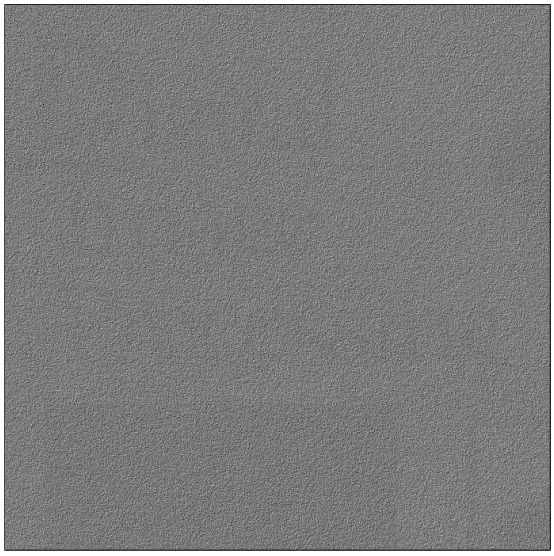

Figure E.14 – Inverse 'blur + noise'

Figure E.15 – Wiener 'blur + noise'

Gaussian noise ($\mu = 0, \sigma^2 = 100$)

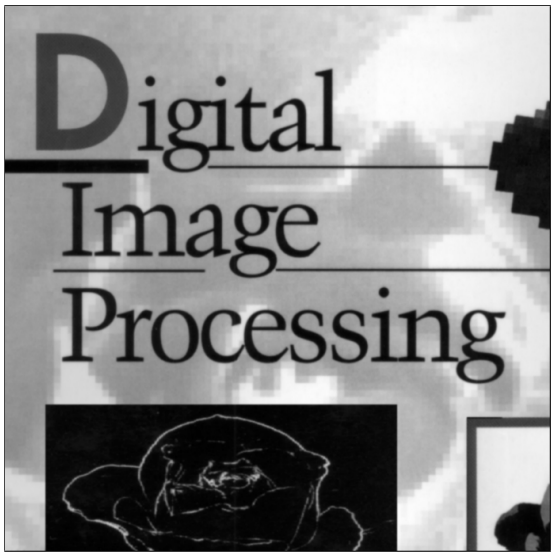

Figure E.16 – Original image

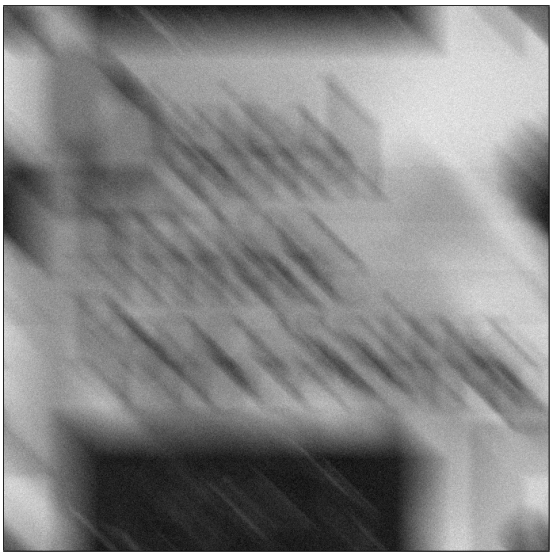

Figure E.17 – Blur + noise

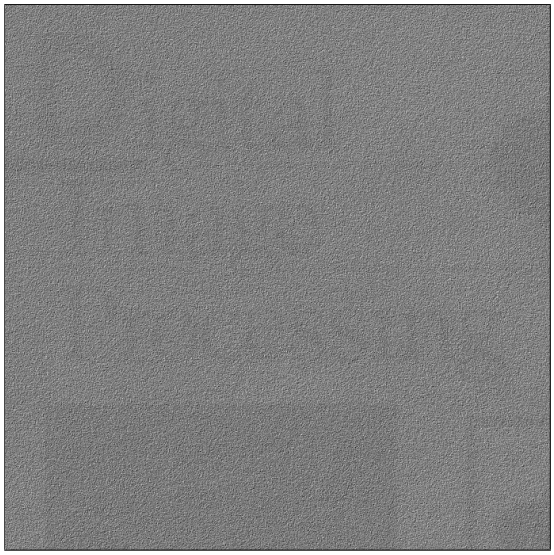

Figure E.18 – Inverse 'blur + noise'

Figure E.19 – Wiener 'blur + noise'

Gaussian noise ($\mu = 0, \sigma^2 = 30$)

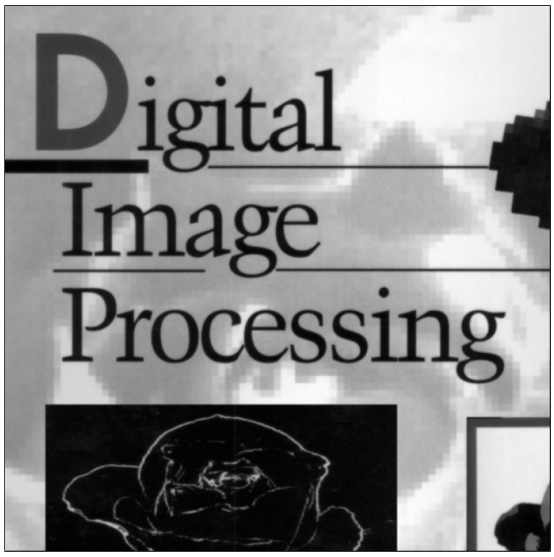

Figure E.20 – Original image

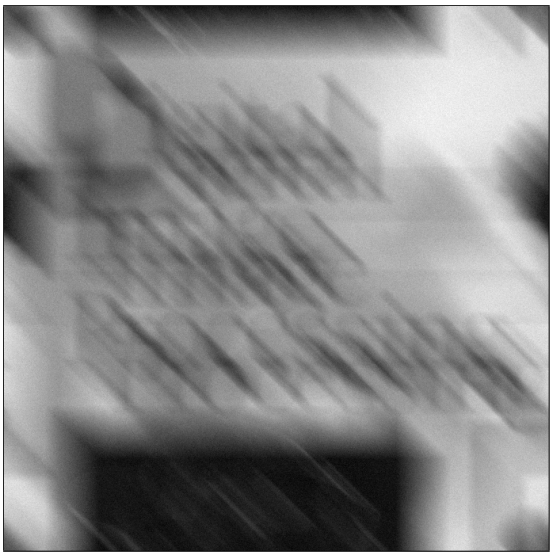

Figure E.21 – Blur + noise

Figure E.22 – Inverse 'blur + noise'

Figure E.23 – Wiener 'blur + noise'

Wiener performance

Figure E.24 – Wiener 'blur + noise ($\sigma^2 = 30$)'

Figure E.25 – Wiener 'blur + noise ($\sigma^2 = 100$)'

Figure E.26 – Inverse 'blur + noise ($\sigma^2 = 300$)'

Figure E.27 – Wiener 'blur + noise ($\sigma^2 = 650$)'

F. Geometric transforms

F.1 Problem statement

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

F.2 Python implementation

Usage : **problem6.py [-h] -i INPUT
(-t TRANSLATE [TRANSLATE ...] | -r ROTATE | -s SCALE) (–nearest | –bilinear)
[–ntruncate] [–debug]**

Use **python problem6.py -h** to see the help.

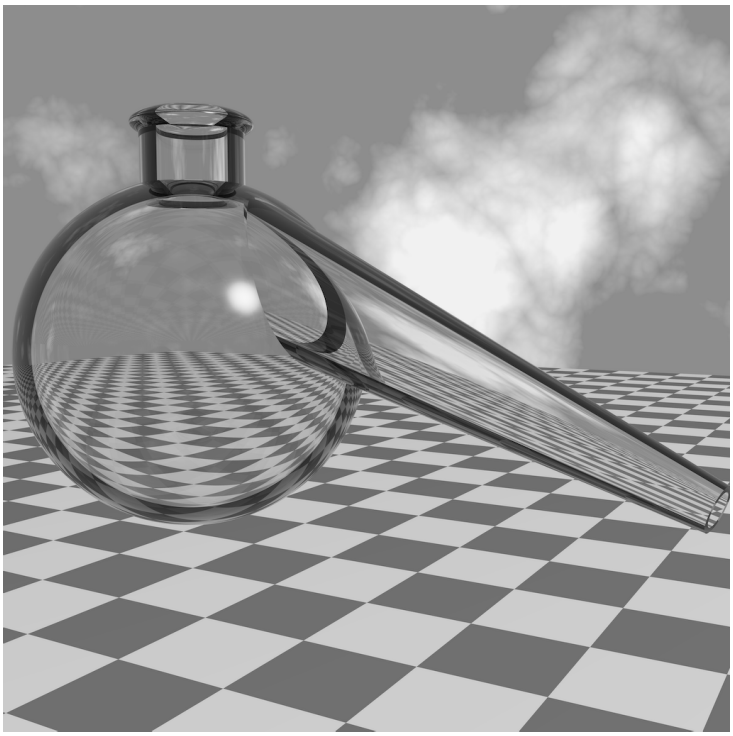

Figure F.1 – Original image

F.3 Translation

python problem6.py -i ray_trace_bottle.tif –bilinear -t tx ty, where (tx, ty) is the 2D translation vector.

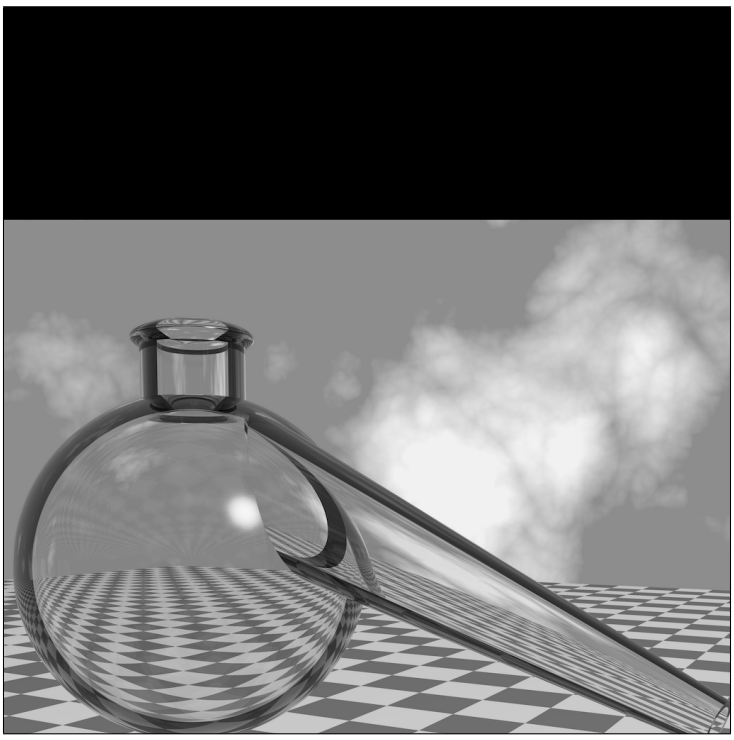

Figure F.2 – Translation (0, 300) nearest

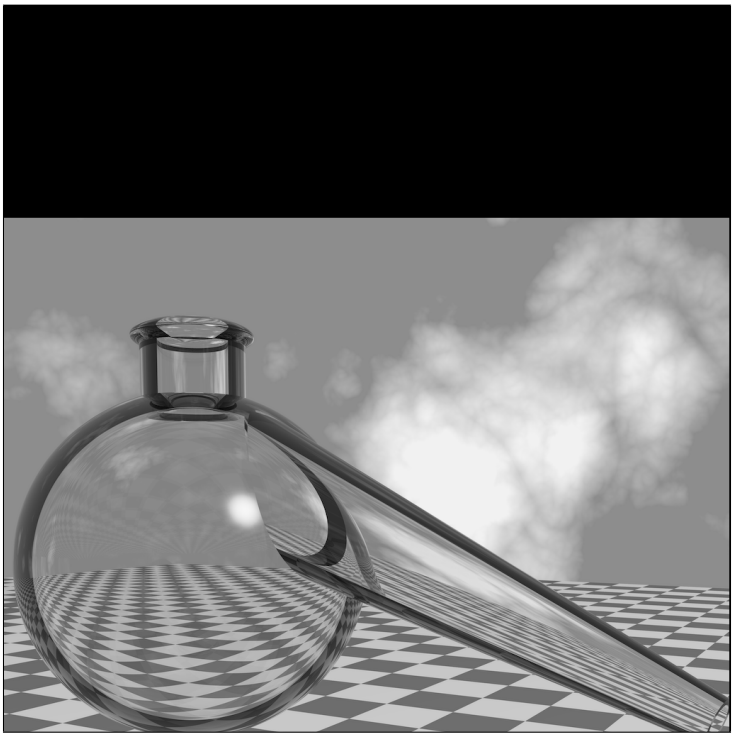

Figure F.3 – Translate (0, 300) bilinear

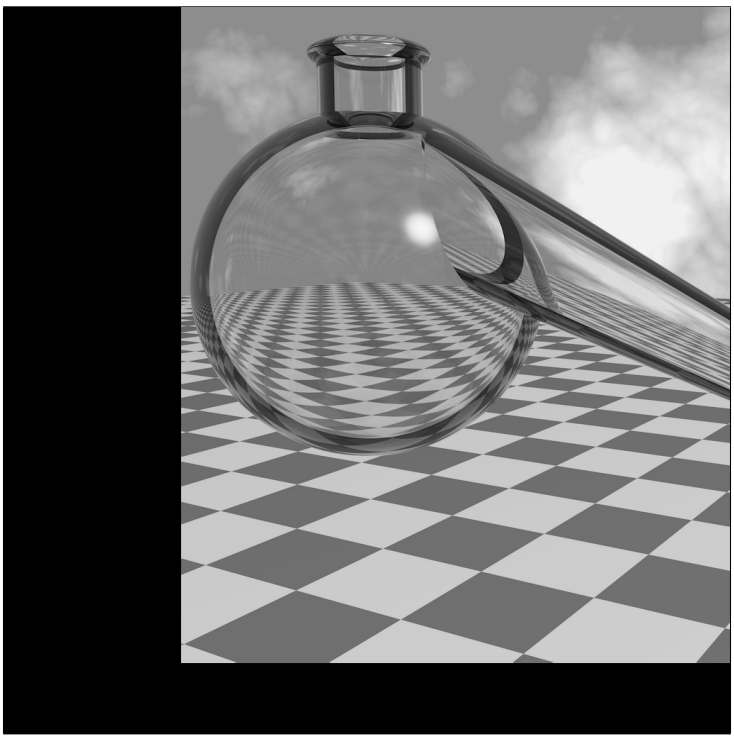

Figure F.4 – Translation (250, -100) nearest

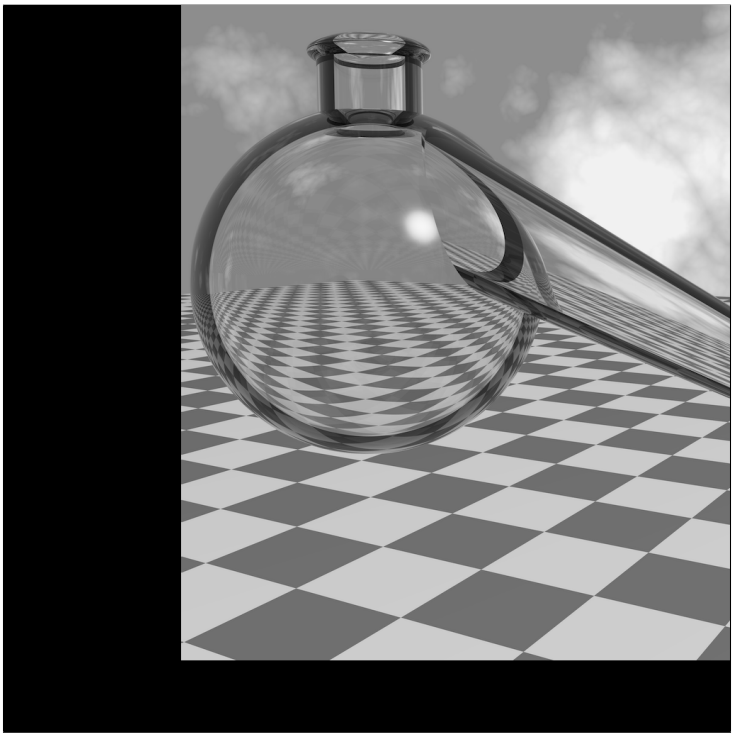

Figure F.5 – Translate (250, -100) bilinear

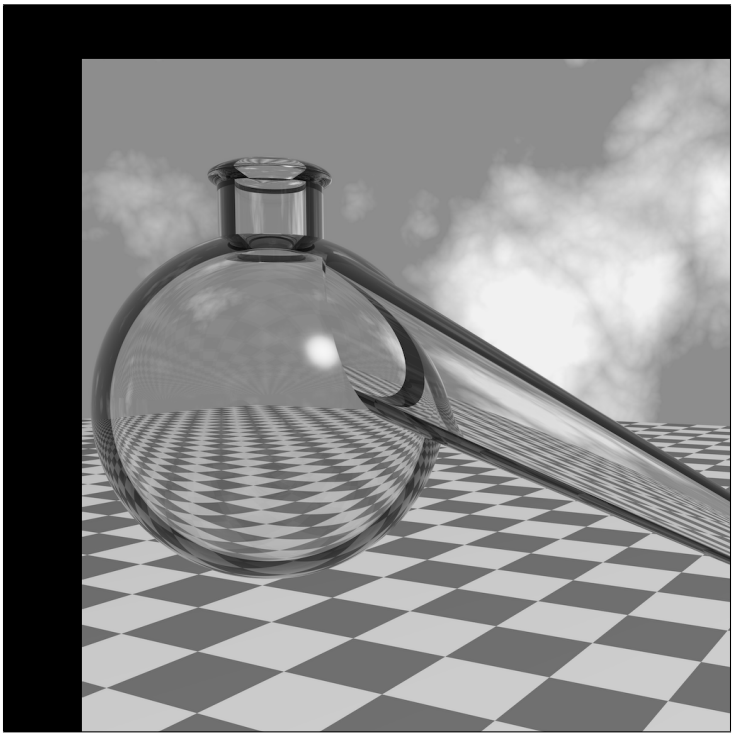

Figure F.6 – Translation (109.8, 75.5) nearest

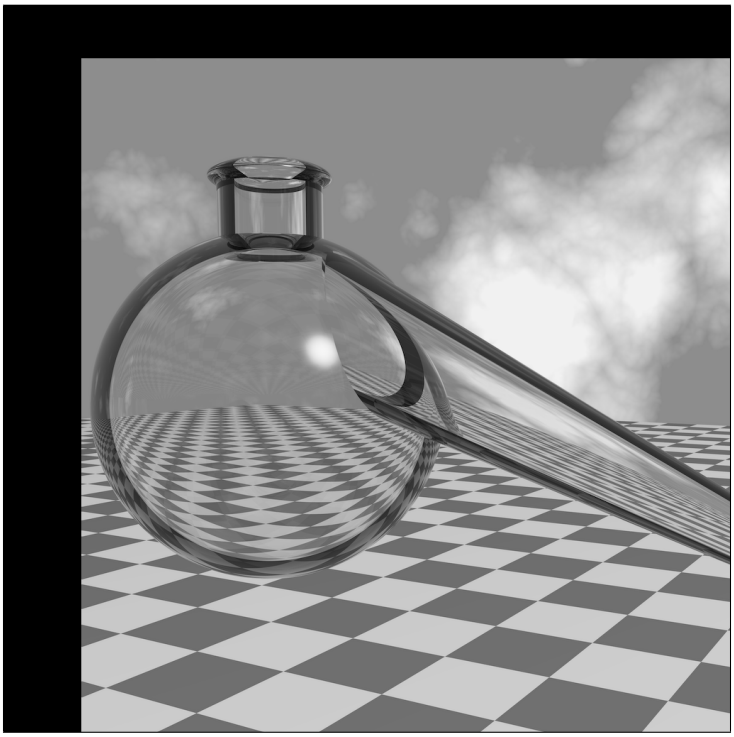

Figure F.7 – Translation (109.8, 75.5) bilinear

F.4 Rotation

python problem6.py -i ray_trace_bottle.tif –nearest -r theta, where theta is the angle of rotation.

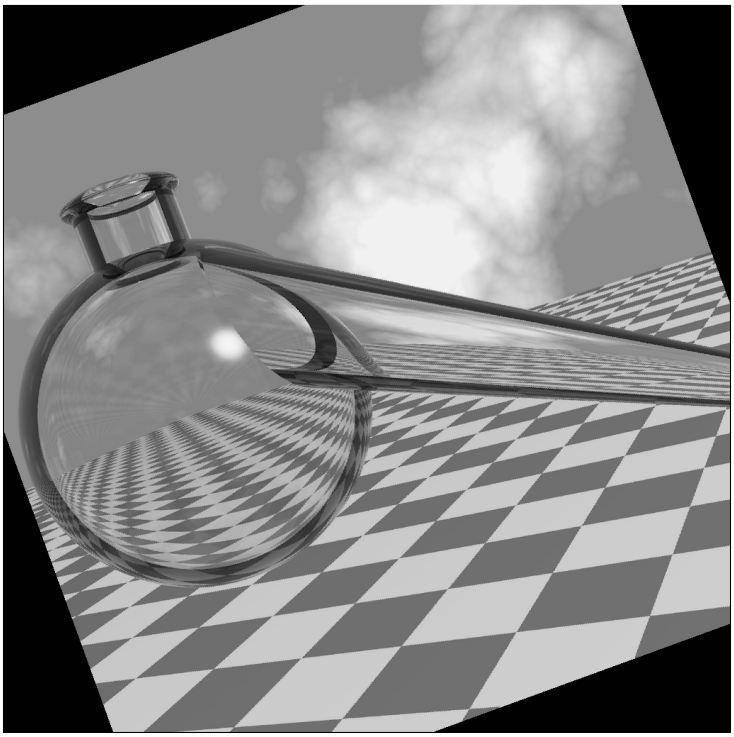

Figure F.8 – Rotate 20° nearest

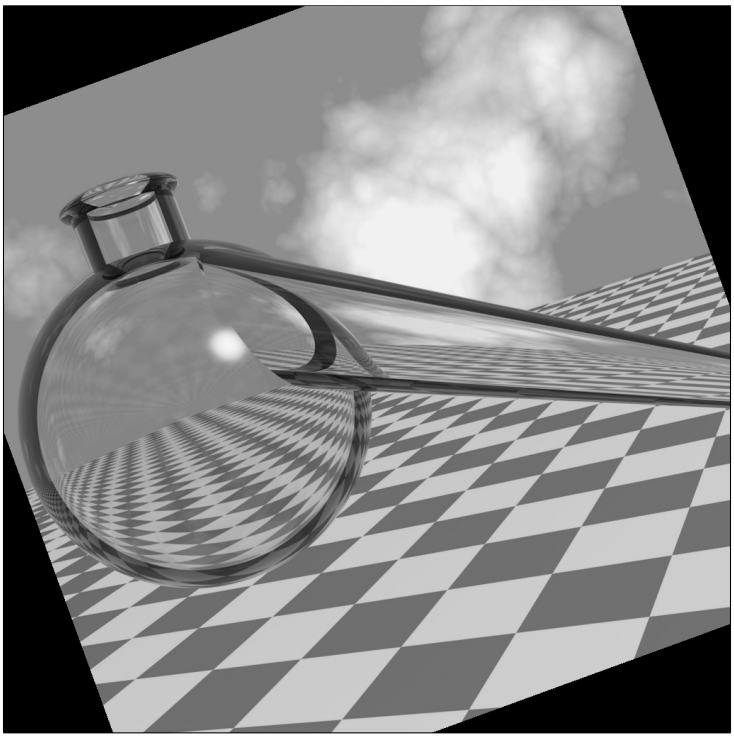

Figure F.9 – Rotate 20° bilinear

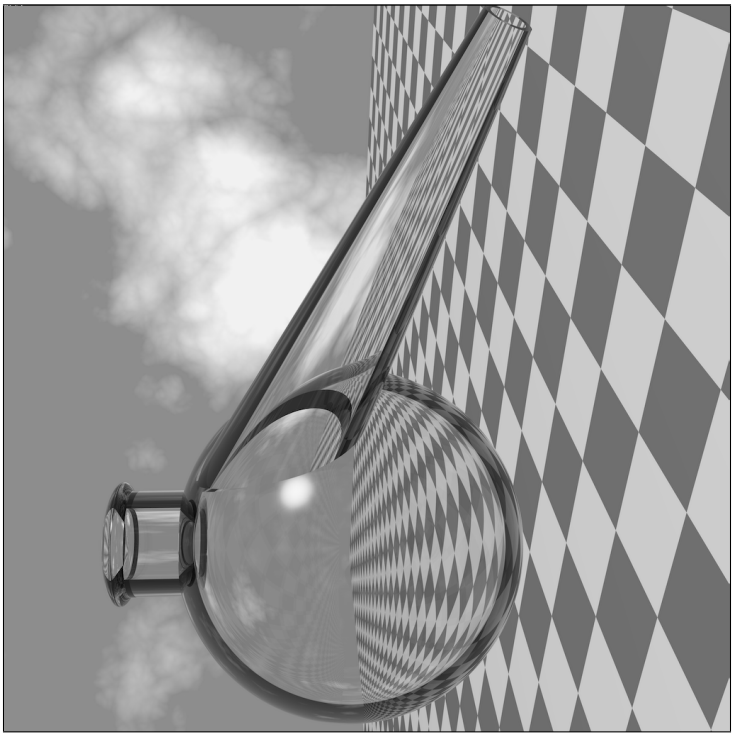

Figure F.10 – Rotate 90° nearest

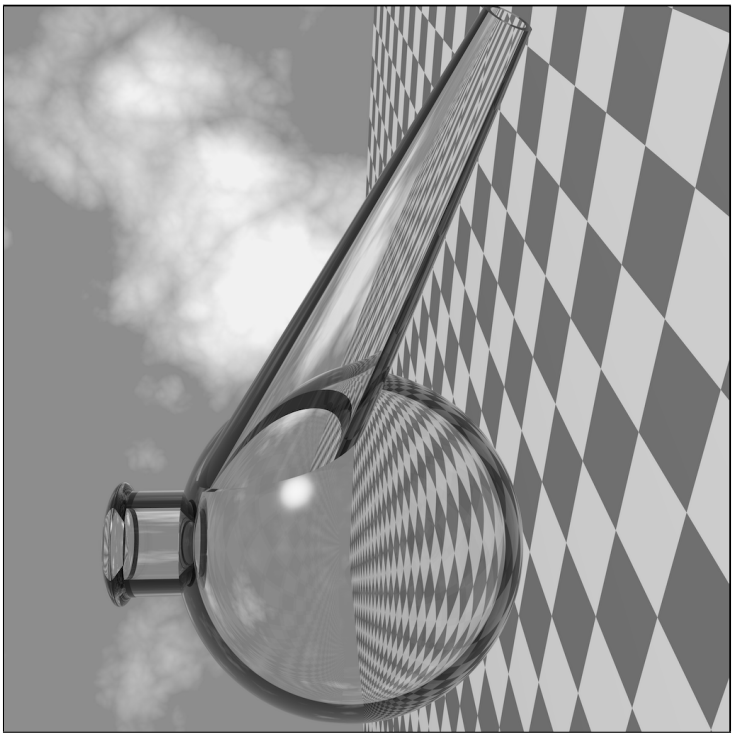

Figure F.11 – Rotate 90° bilinear

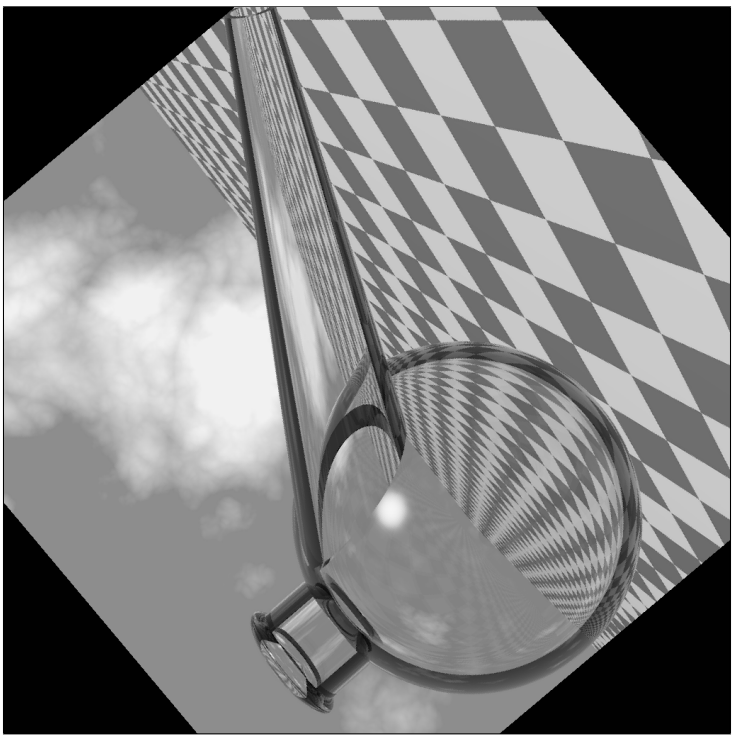

Figure F.12 – Rotate 130° nearest

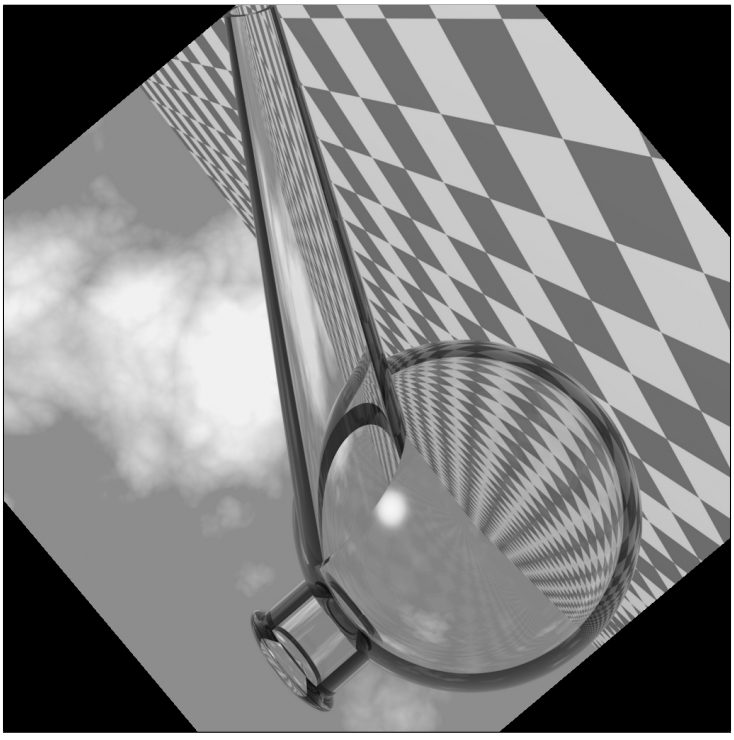

Figure F.13 – Rotate 130° bilinear

F.5 Scaling

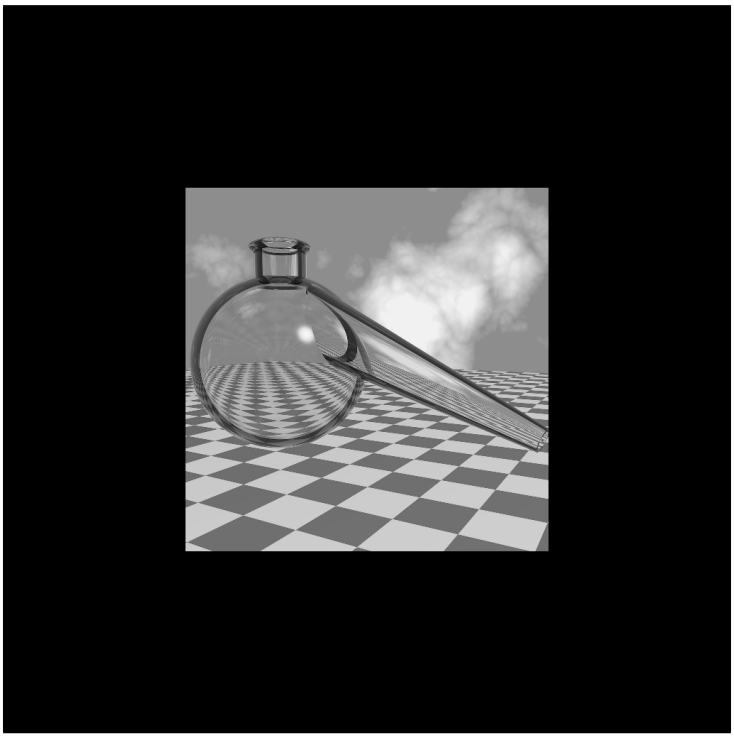

Figure F.14 – Scaling 0.5 nearest

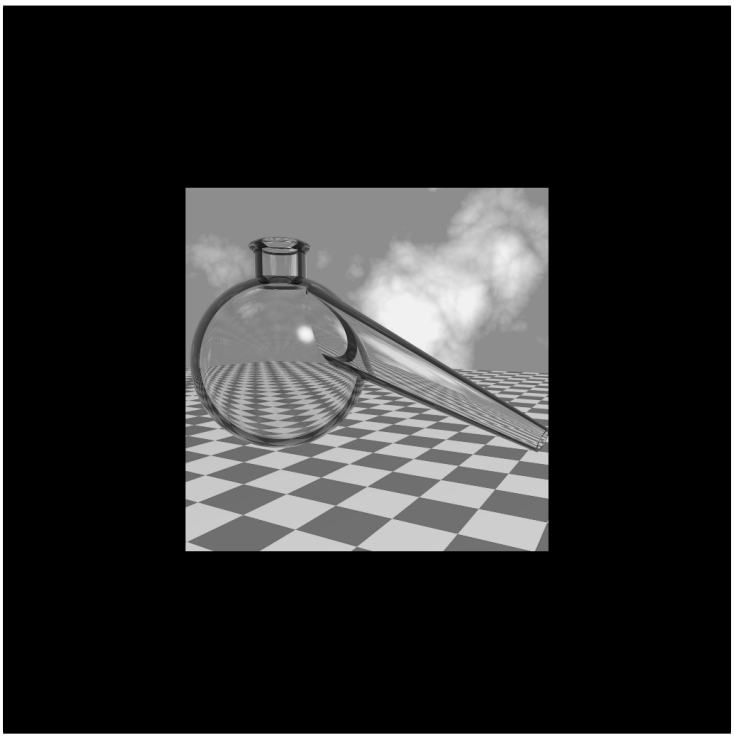

Figure F.15 – Scaling 0.5 bilinear

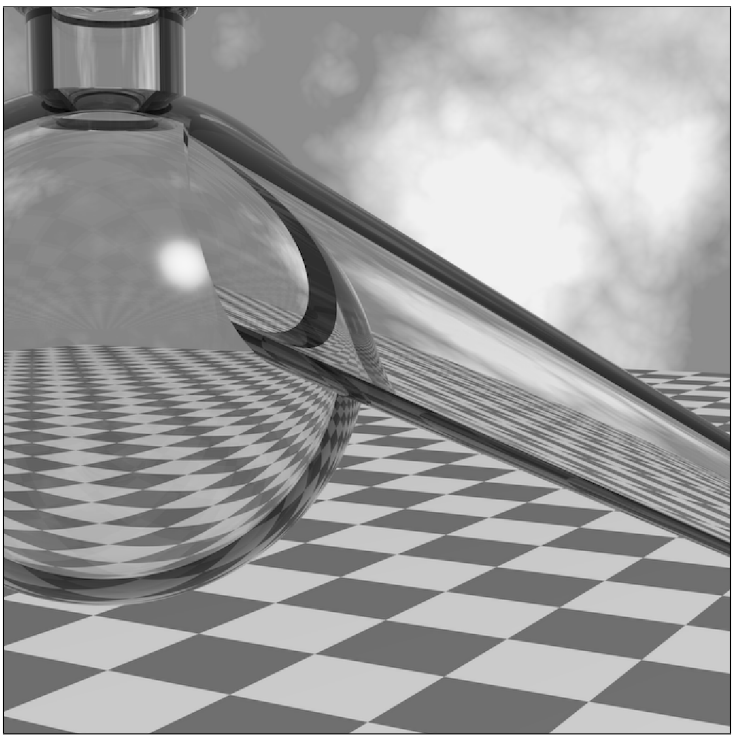

Figure F.16 – Scaling 1.5 nearest

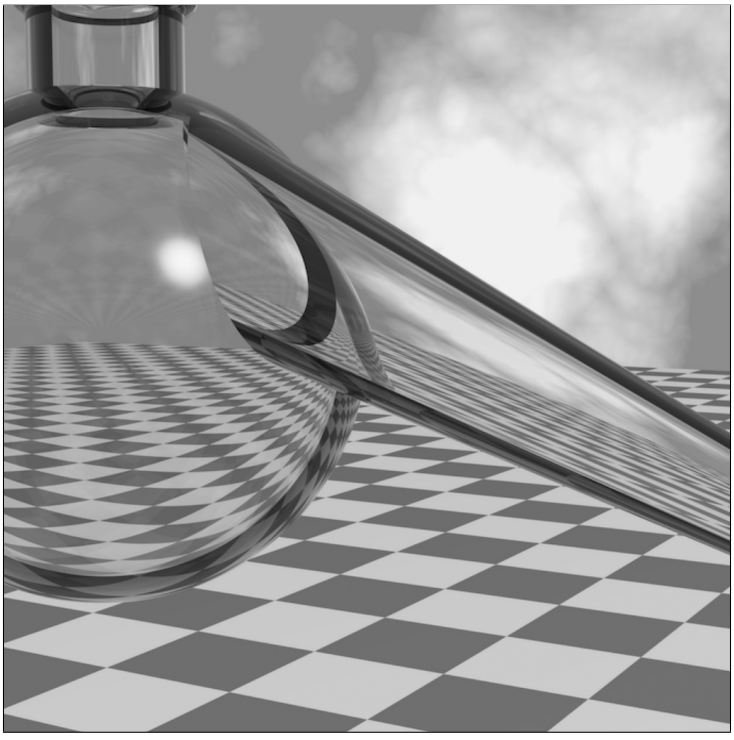

Figure F.17 – Scaling 1.5 bilinear

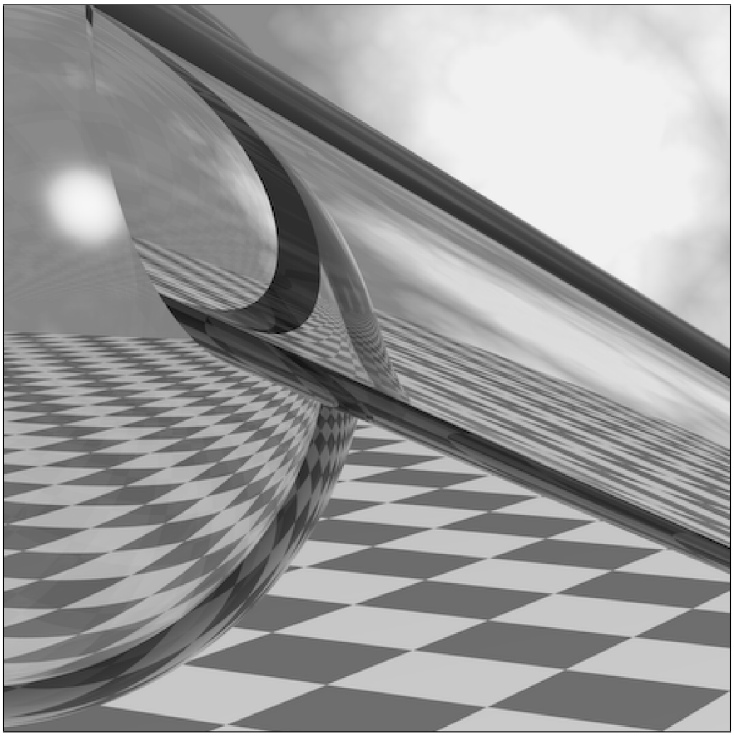

Figure F.18 – Scaling 2.3 nearest

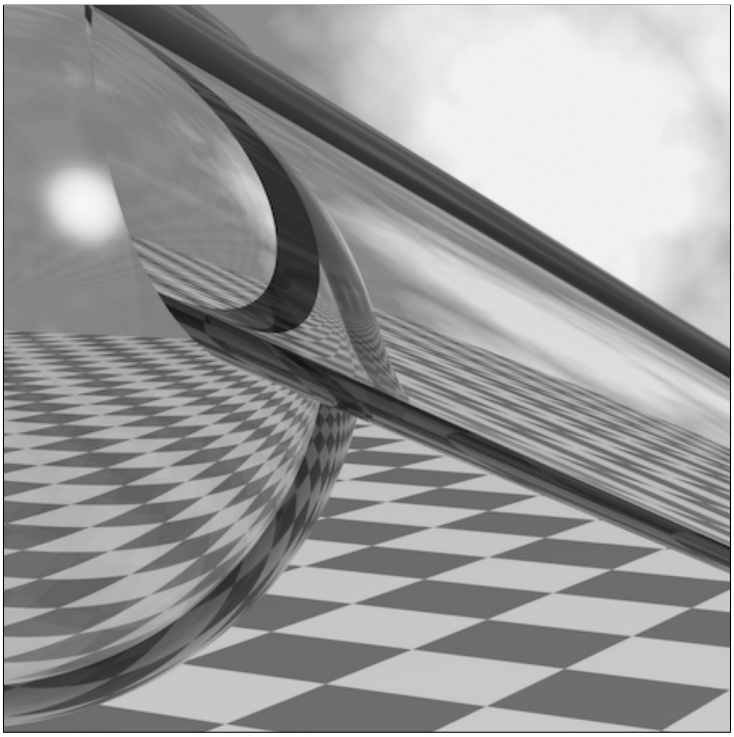

Figure F.19 – Scaling 2.3 bilinear

G. Transform image compression

G.1 Problem statement

(a) Investigate image compression based on DCT. Divide the image into 8-by-8 subimages, compute the two-dimensional discrete cosine transform of each subimage, compress the test image to different qualities by discarding some DCT coefficients based on zonal mask and threshold mask and using the inverse discrete cosine transform with fewer transform coefficients. Display the original image, the reconstructed images and the difference images.

(b) Investigate image compression based on wavelets.
Consider four types of wavelets :
   (a) Haar (2x2)
   (b) Daubechies (8-tap)
   (c) Symlet (8-tap)
   (d) Cohen-Daubechies-Feauveau (17-tap)

G.2 Python implementation

Two programs :

– DCT compression : **dct.py** Usage : **dct.py [-h] [–zonal | –threshold] [-z Z] image_path**
Use **python dct.py -h** to see the help.

– Wavelet compression : **wavelet.py** Usage : **wavelets.py [-h] [–haar | –daub | –symlet | –cohen] [-l LEVEL]**
[-t THRESHOLD] image_path Use **python wavelets.py -h** to see the help.

Figure G.1 – Original image

G.3 Image compression based on DCT

G.3.1 Zonal mask

python dct.py –zonal -z Z lenna.tif where $Z = 1, 4$ or 7.

Example : Zonal 5

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{G.1}$$

Example : Zonal 2

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{G.2}$$

Figure G.2 – Zonal 7

Figure G.3 – Zonal 7 - Diff

Figure G.4 – Zonal 4

Figure G.5 – Zonal 4 - Diff

Figure G.6 – Zonal 2

Figure G.7 – Zonal 2 - Diff

Figure G.8 – Zonal 1

Figure G.9 – Zonal 1 - Diff

G.4 Image compression based on Wavelets

Decompose the image by wavelets, truncate some coefficients to 0 below a threshold (0 in this case) and reconstructed the image.
python wavelets.py –haar lenna.tif -l 1 -t 0.

G.4.1 Haar

Figure G.10 – Haar

Figure G.11 – Haar reconstructed

Figure G.12 – Haar difference

G.4.2 Daubechies

Figure G.13 – Daubechies

Figure G.14 – Daubechies reconstructed

Figure G.15 – Daubechies difference

G.4.3 Symlet

Figure G.16 – Symlet

Figure G.17 – Symlet reconstructed

Figure G.18 – Symlet difference

G.4.4 Cohen-Daubechies-Feauveau

Figure G.19 – Cohen

Figure G.20 – Cohen reconstructed

Figure G.21 – Cohen difference

H. Morpholigical Processing

H.1 Problem statement

(a) Implement the morphological operations : erosion, dilation, opening and closing, and use the noisy_fingerprint.tif to check your implementation.

(b) Implement boundary extraction, hole filling, connected component extraction. Using lincoln_from_penny.tif, region_filling_refletion.tif and chickenfilet_with_bones.tif to the results, respectively.

H.2 Python implementation

Usage : **problem8.py [-h] [–debug] [–test] [–erosion] [–dilation] [–opening] [–closing] [–boundary] [–filling] [–connected] image_path**

Use **python problem8.py -h** to see the help.

H.3 Erosion

python problem8.py –erosion noisy_fingerprint.tif

Figure H.1 – Original image

Figure H.2 – Erosion

H.4 Dilation

python problem8.py –dilation noisy_fingerprint.tif

Figure H.3 – Original image

Figure H.4 – Dilation

H.5 Opening

python problem8.py –opening noisy_fingerprint.tif

Figure H.5 – Original image

Figure H.6 – Opening

H.6 Closing

python problem8.py –closing noisy_fingerprint.tif

Figure H.7 – Original image

Figure H.8 – Closing

H.7 Boundary extraction

python problem8.py –boundary licoln_from_penny.tif

Figure H.9 – Original image

Figure H.10 – Boundary extraction

H.8 Hole filling

python problem8.py –filling region_filling_reflections.tif

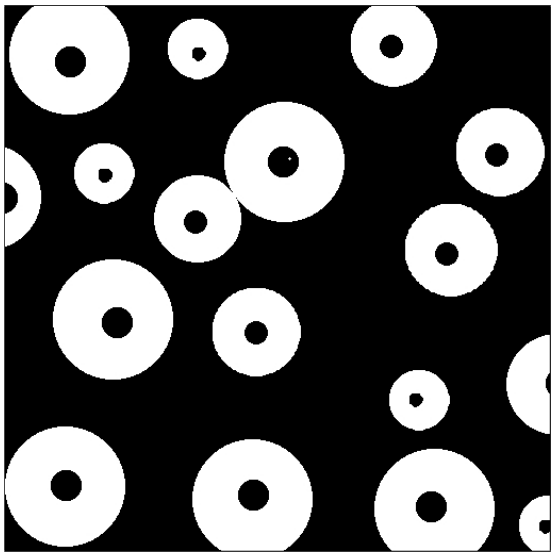

Figure H.11 – Original image

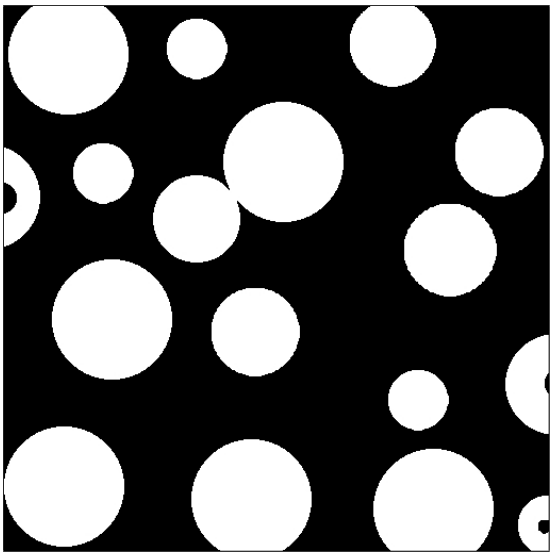

Figure H.12 – Hole filling

H.9 Connected component extraction

python problem8.py –connected chickenfilet_with_bones.tif

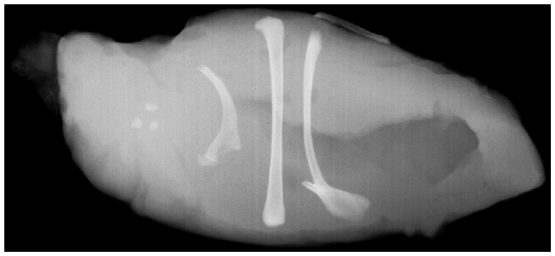

Figure H.13 – Original image

Figure H.14 – Thresholding

Figure H.15 – Erosion 5×5

Connected components :

Connected component	No of pixels
01	836
02	690
03	44
04	148
05	1
06	18
07	19
08	26
09	2
10	14
11	11
12	13
13	28
14	87
15	18

I. Image segmentation

I.1 Problem statement

(a) Develop a program to implement the Roberts, Prewitt, Sobel, the Marr-Hildreth and the Canny edge detectors. Use the image 'building.tif' to test your detectors. (For technique details of Marr-Hildreth and Canny, please refer to pp.736-747 (3rd edition, Gonzalez DIP) or MH-Canny.pdf at the same address of the slides.)

(b) Develop a program to implement the Otsu's method of thresholding segementation, and compare the results with the global thresholding method using test image 'polymersomes.tif. (For technique details, please refer to pp.763-770 (3rd edition, Gonzalez DIP), or Otsu.pdf at the same ftp address of slides.)

I.2 Python implementation

Three programs :

- Marr-Hildreth edge detector : **marr.py**
  Usage : **marr.py [-h] [-s SIGMA] image_path**
  Use **python marr.py -h** to see the help.

- Canny edge detector : **canny.py**
  Usage : **canny.py [-h] (–roberts | –sobel | –prewitt) [-s SIGMA] [–th TH] [–tl TL] image_path**
  Use **python canny.py -h** to see the help.

- Otsu's method of thresholding segmentation : **otsu.py**
  Usage : **otsu.py [-h] [-o] [-g] image_path**
  Use **python otsu.py -h** to see the help.

I.3 Marr-Hildreth edge detector

python marr.py -s 4 building.tif

Figure I.1 – Original image

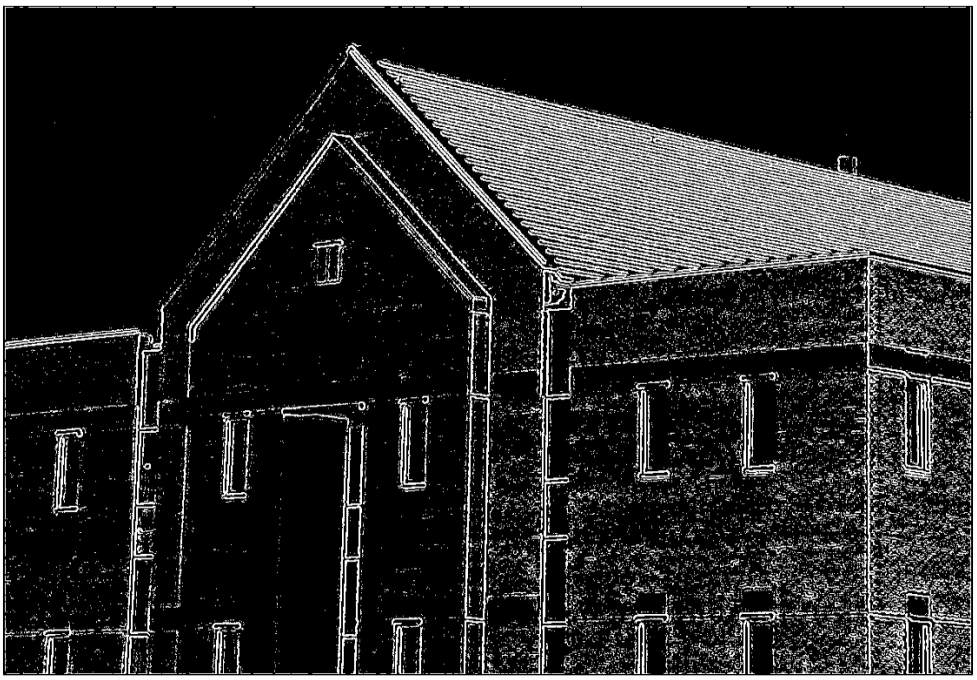

Figure I.2 – Marr-Hildreth

I.4 Canny edge detector

I.4.1 Roberts

python canny.py –roberts -s 4 –tl 0.04 –th 0.10 building.tif

Figure I.3 – Original image

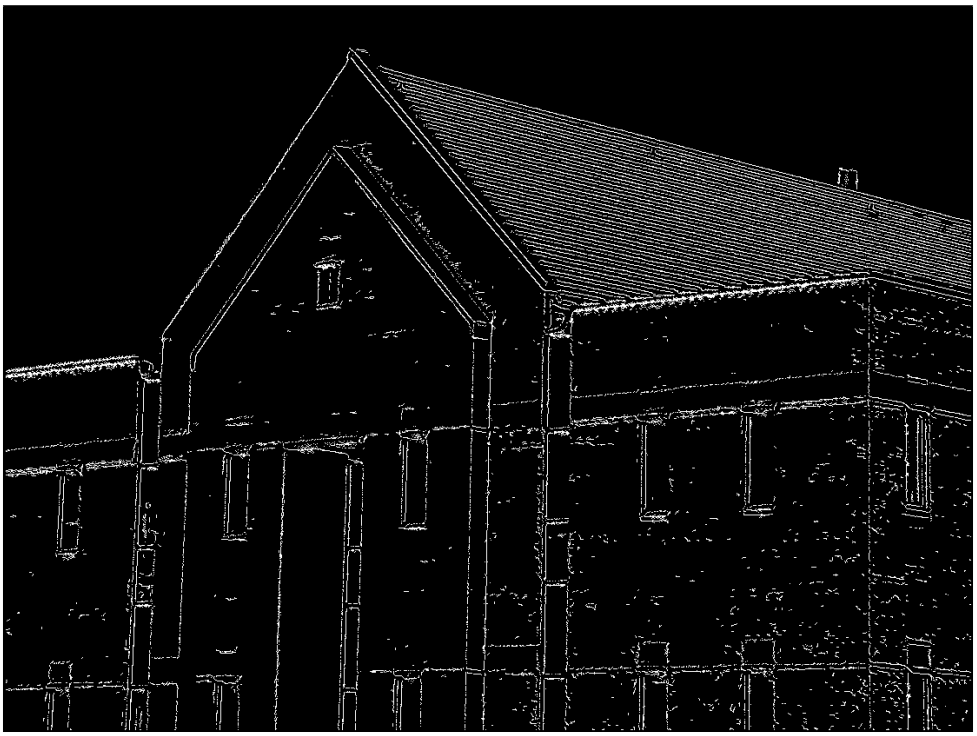

Figure I.4 – Roberts

I.4.2 Prewitt

python canny.py –prewitt -s 4 –tl 0.04 –th 0.10 building.tif

Figure I.5 – Original image

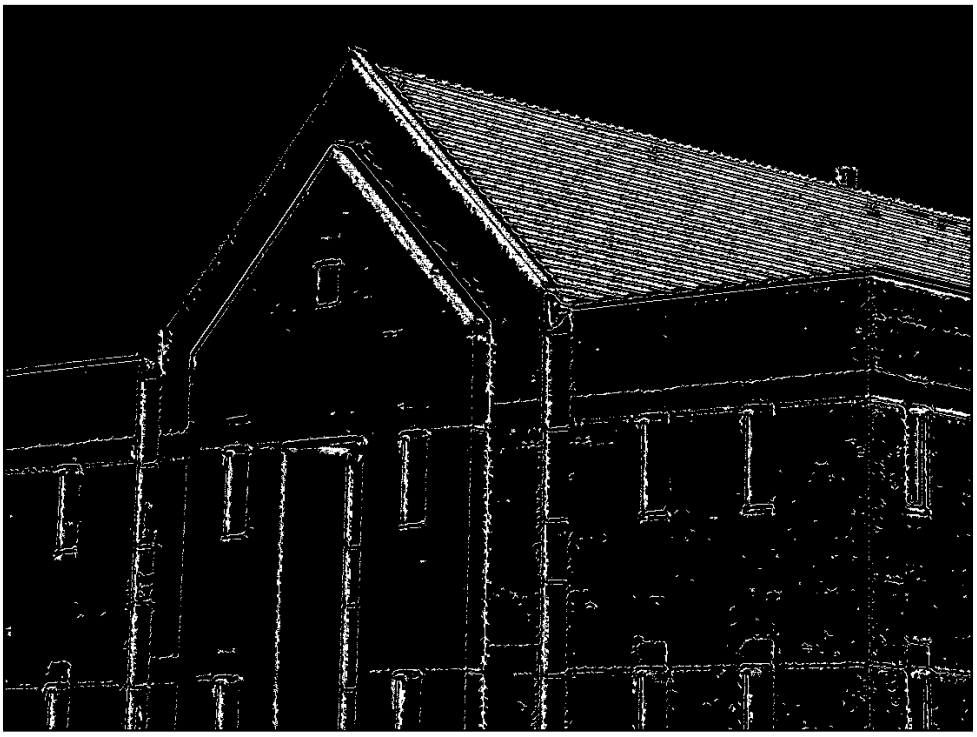

Figure I.6 – Prewitt

I.4.3 Sobel

python canny.py –sobel -s 4 –tl 0.04 –th 0.10 building.tif

Figure I.7 – Original image

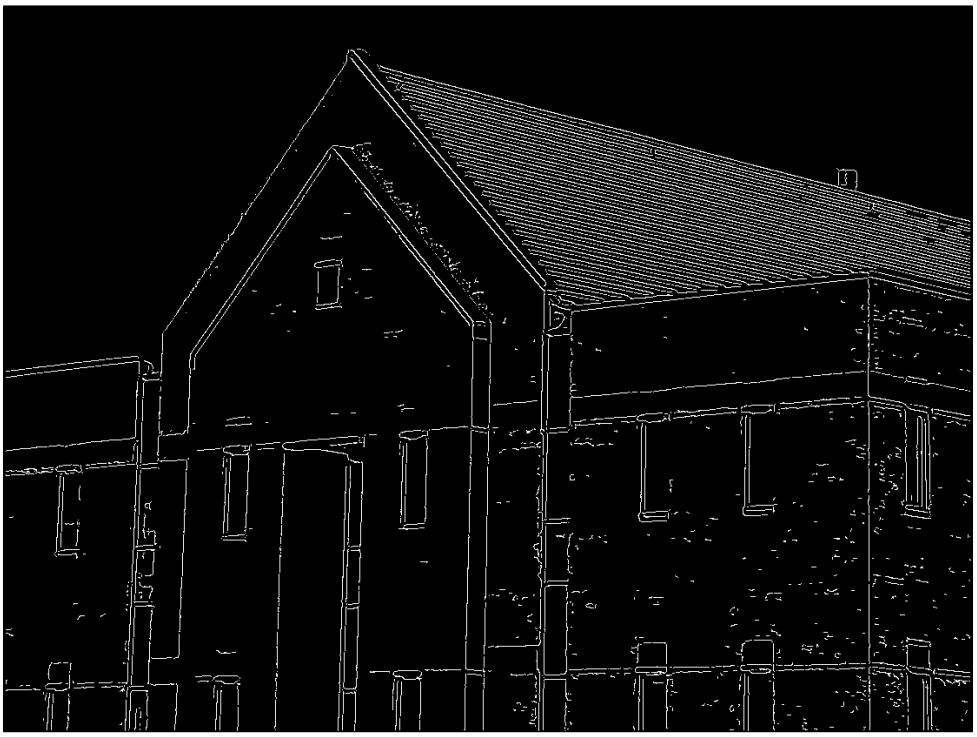

Figure I.8 – Sobel

I.5 Thresholding segmentation

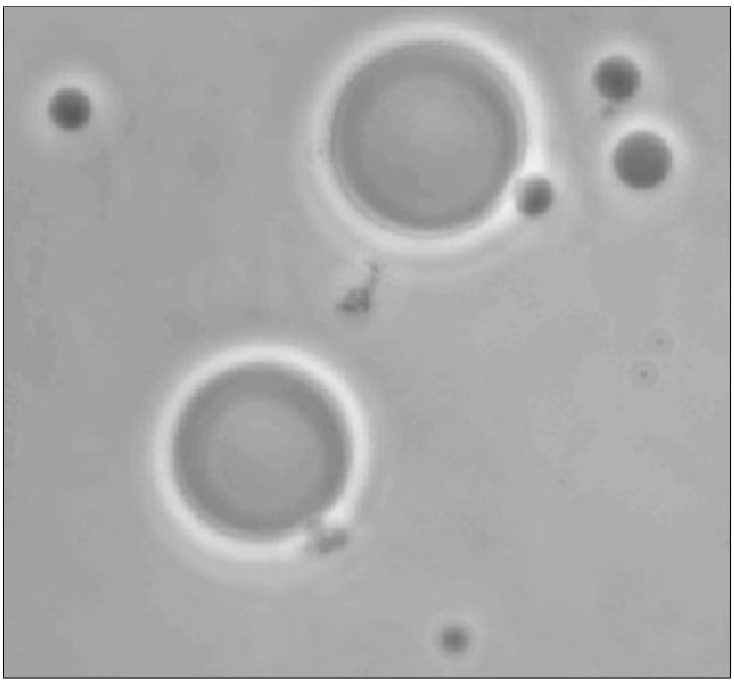

Figure I.9 – Original image

I.5.1 Otsu

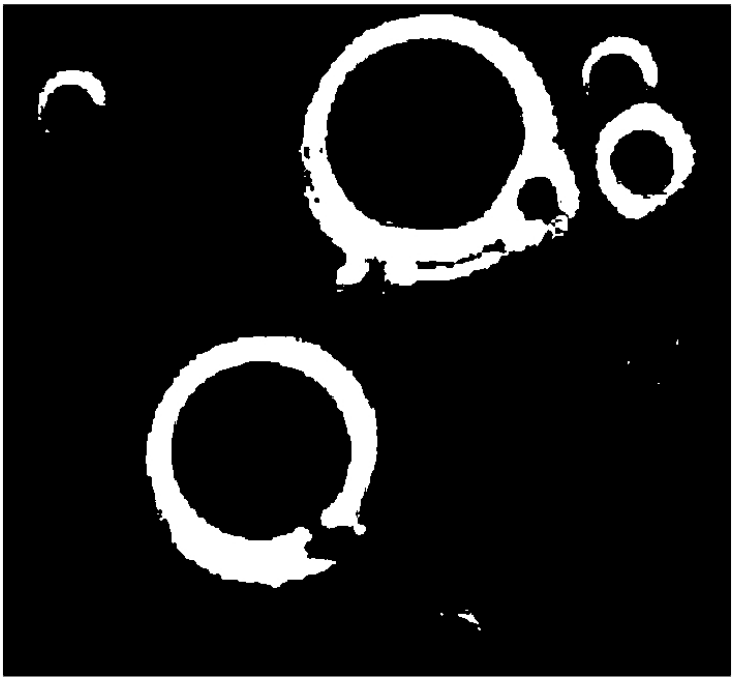

Figure I.10 – Otsu

I.5.2 Global thresholding

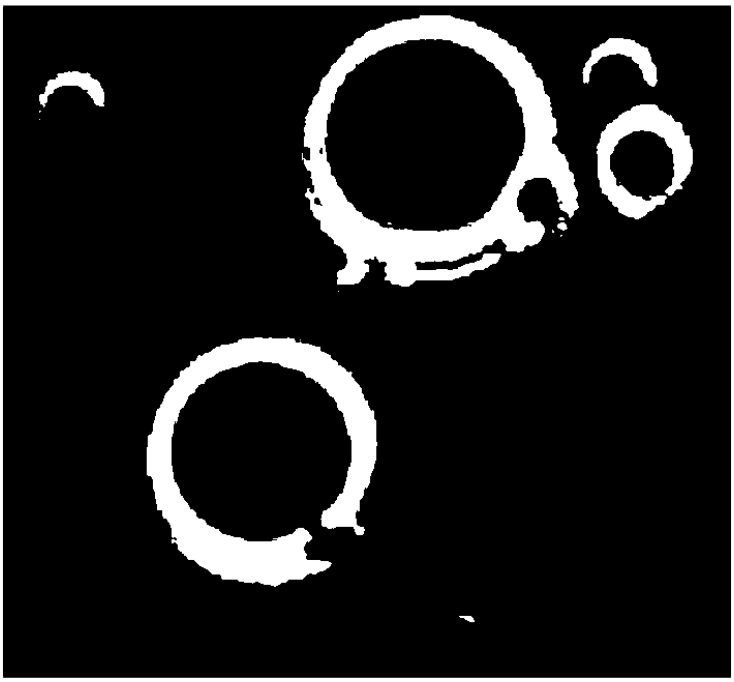

Figure I.11 – Global thresholding

J. Image representation and description

J.1 Problem statement

(a) Develop a program to implement the boundary following algorithm, the resampling grid and calculate the chain code and the first difference chain code. Use the image 'noisy_stroke.tif' for test. (For technique details, please refer to pp.818-822 (3rd edition, Gonzalez DIP) or boundaryfollowing.pdf at the same address of the slides.)

(b) Develop a program to implement the image description by the principal components (PC). Calculate and display the PC images and the reconstructed images from 2 PCs. Use the six images in 'washingtonDC.rar' as the test images.

J.2 Python implementation

Four programs :

- Boundary following : **boundary.py**
  Usage : **boundary.py [-h] [–smooth] image_path**
  Use **python boundary.py -h** to see the help.

- Resampling grid : **resampling.py**
  Usage : **resampling.py [-h] [-s SAMPLING [SAMPLING ...]] boundary_image**
  Use **python resampling.py -h** to see the help.

- Chain code : **chaincode.py**
  Usage : **chaincode.py [-h] [-s SAMPLING [SAMPLING ...]] boundary_image**
  Use **python chaincode.py -h** to see the help.

- Image description by the principal components (PC) : **pc.py**
  Usage : **pc.py [-h] [-n N] [–debug] [–diff] [–nshow]**
  Use **python pc.py -h** to see the help.

J.3 Boundary following

python boundary.py noisy_stroke.tif.

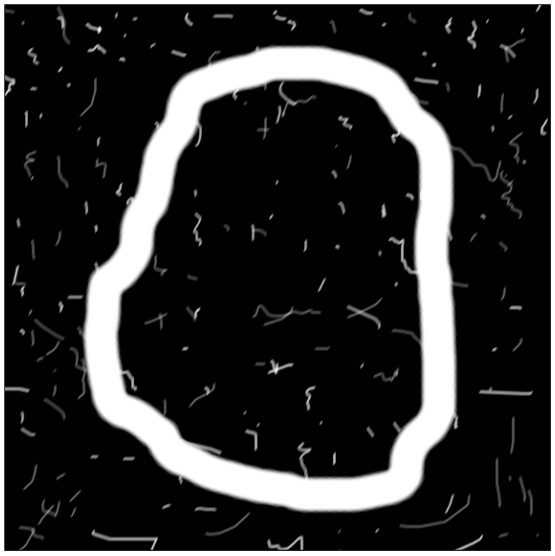

Figure J.1 – Original image

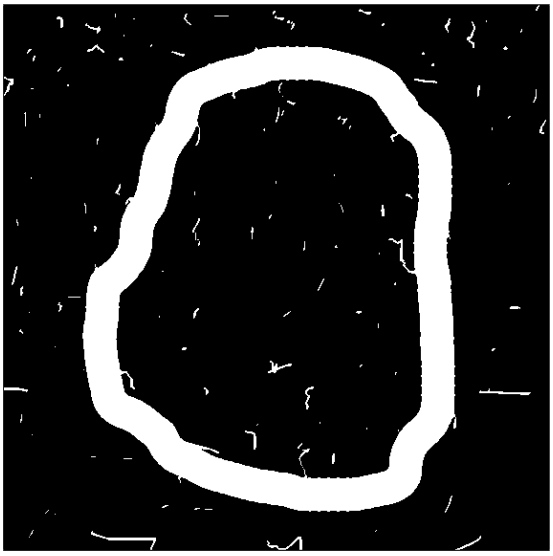

Figure J.2 – Black & white

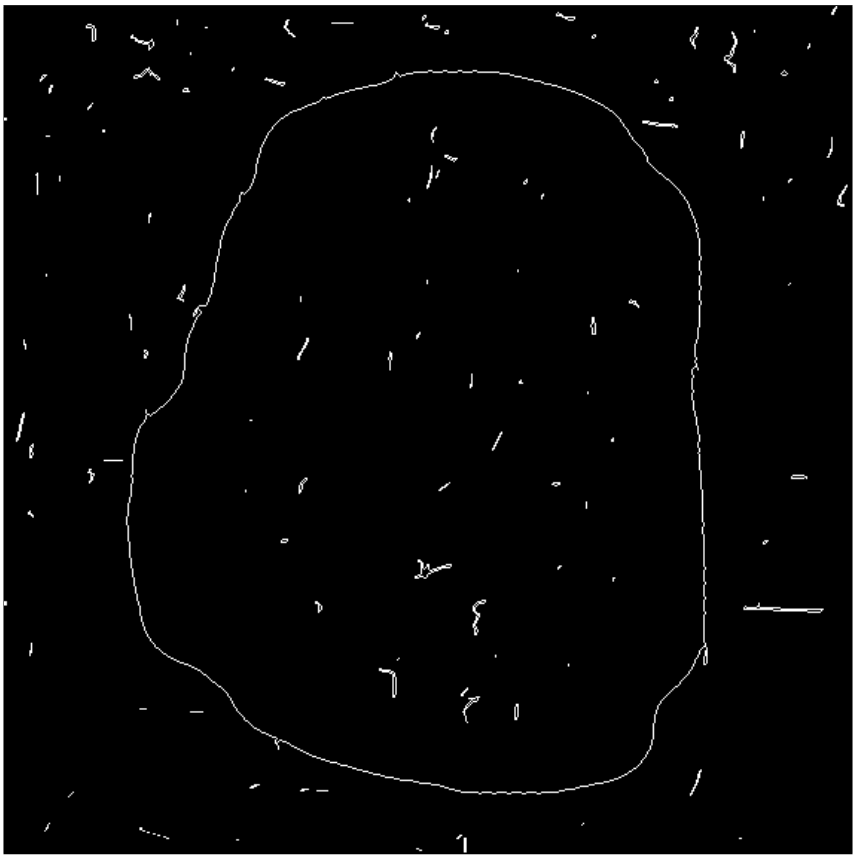

Figure J.3 – Boundaries

Even the boundaries of the noise are found... We need to remove the noise beforehand. For that, let use a Gaussian blur of mean 0 and variance 10.

python boundary.py noisy_stroke.tif –smooth.

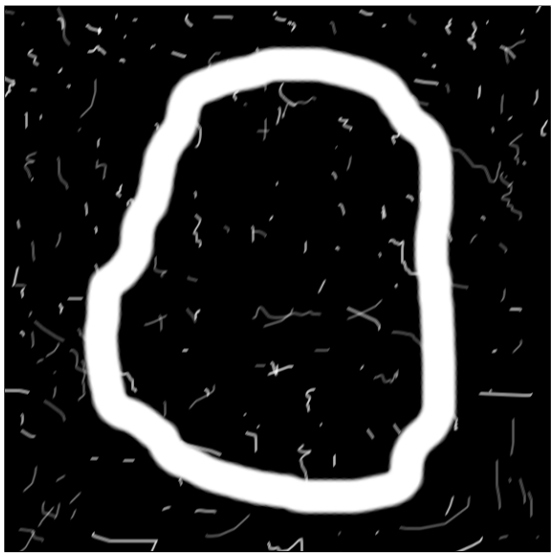

Figure J.4 – Original image

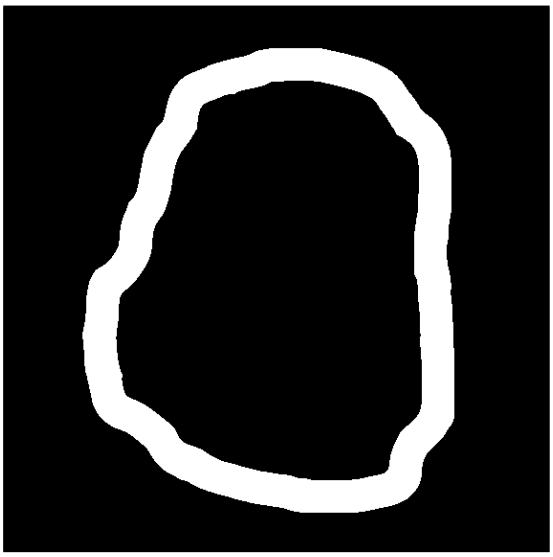

Figure J.5 – Smoothing + binarisation

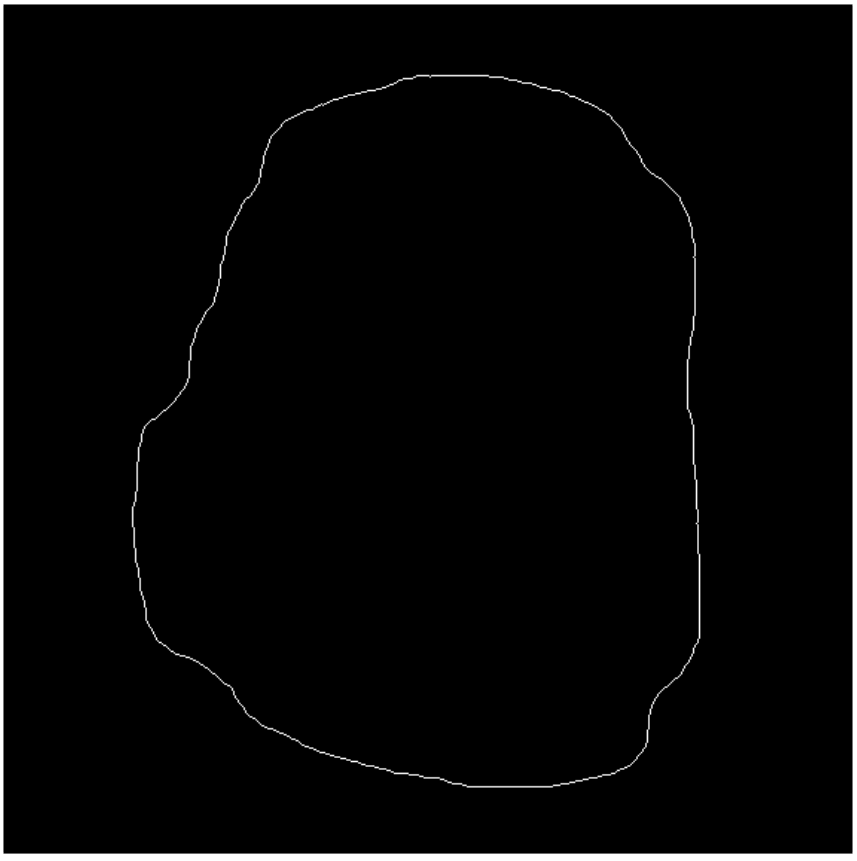

Figure J.6 – Boundaries

J.4 Resampling grid

python resampling.py noisy_stroke_boundary.png -s Sx Sy, where Sx and Sy are the sampling intervalles along the X and Y axis.

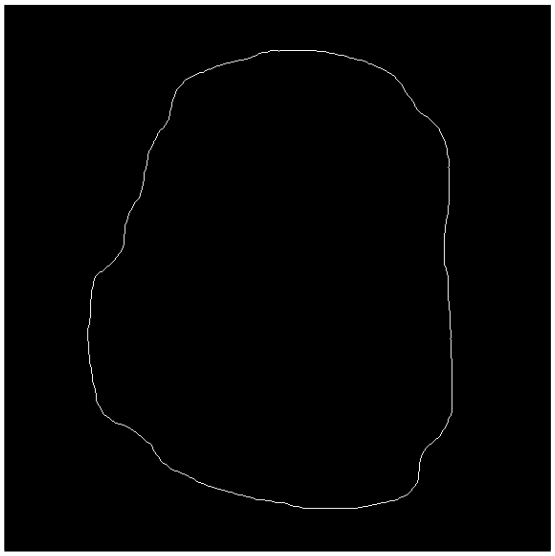

Figure J.7 – Original image

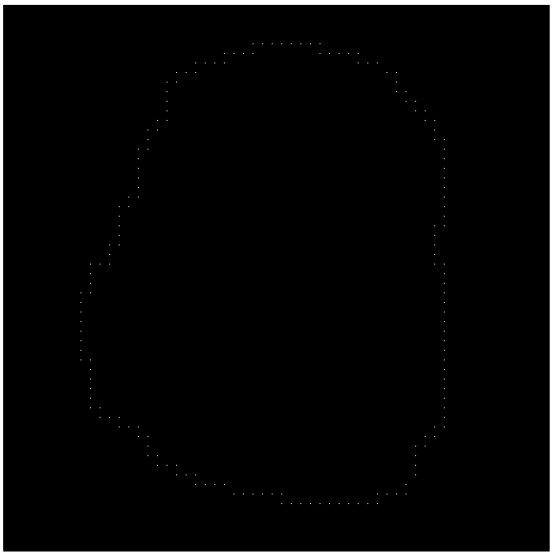

Figure J.8 – R-grid (S = (10, 10))

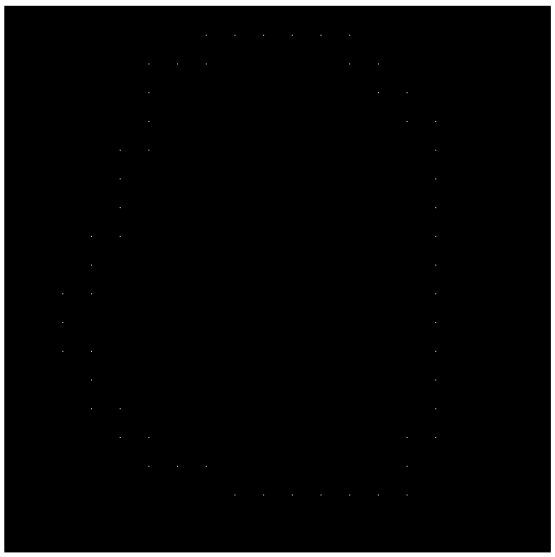

Figure J.9 – R-grid (S = (30, 30))

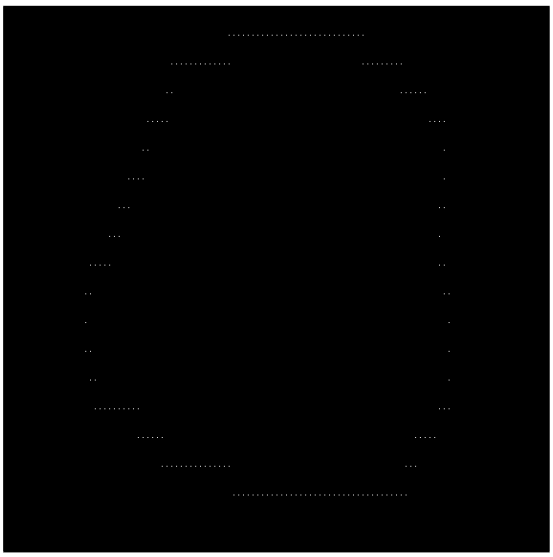

Figure J.10 – R-grid (S = (5, 30))

J.5 Chain code and first difference chain code

J.5.1 Chain code - resampling grid (10, 10)

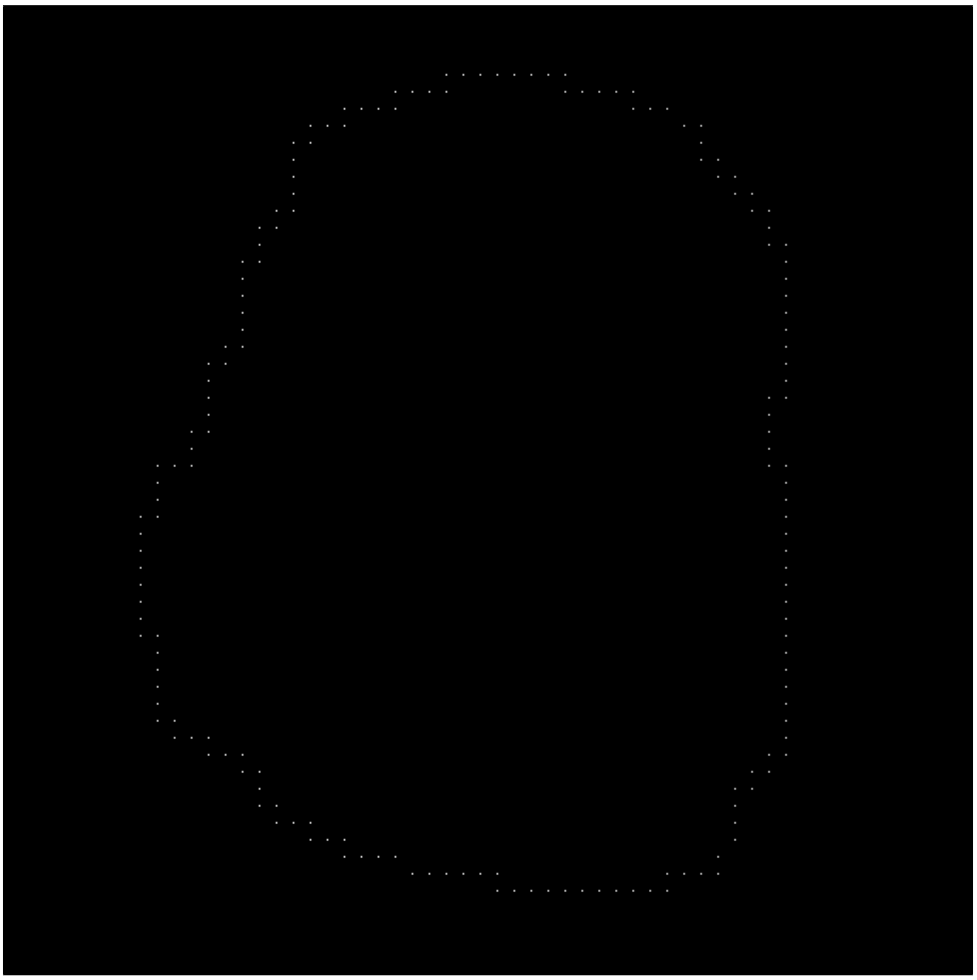

Figure J.11 – Resampling grid (S = (10, 10))

Chaincode (length = 170) :

00000060000600706606060606606666666
66466660666666666666666664646466656
44464444444444244444344424424424224
24424424222224222222202220022022220
20222220220202222020020002 0002

First difference (length = 169) :

00000620006207160262626260260000000
06200026000000000000000062626200716
00260000000006200007100620620626026
20620626000026000000620060206200062
62000062062620006260260026002

J.5.2 Chain code - resampling grid (30, 30)

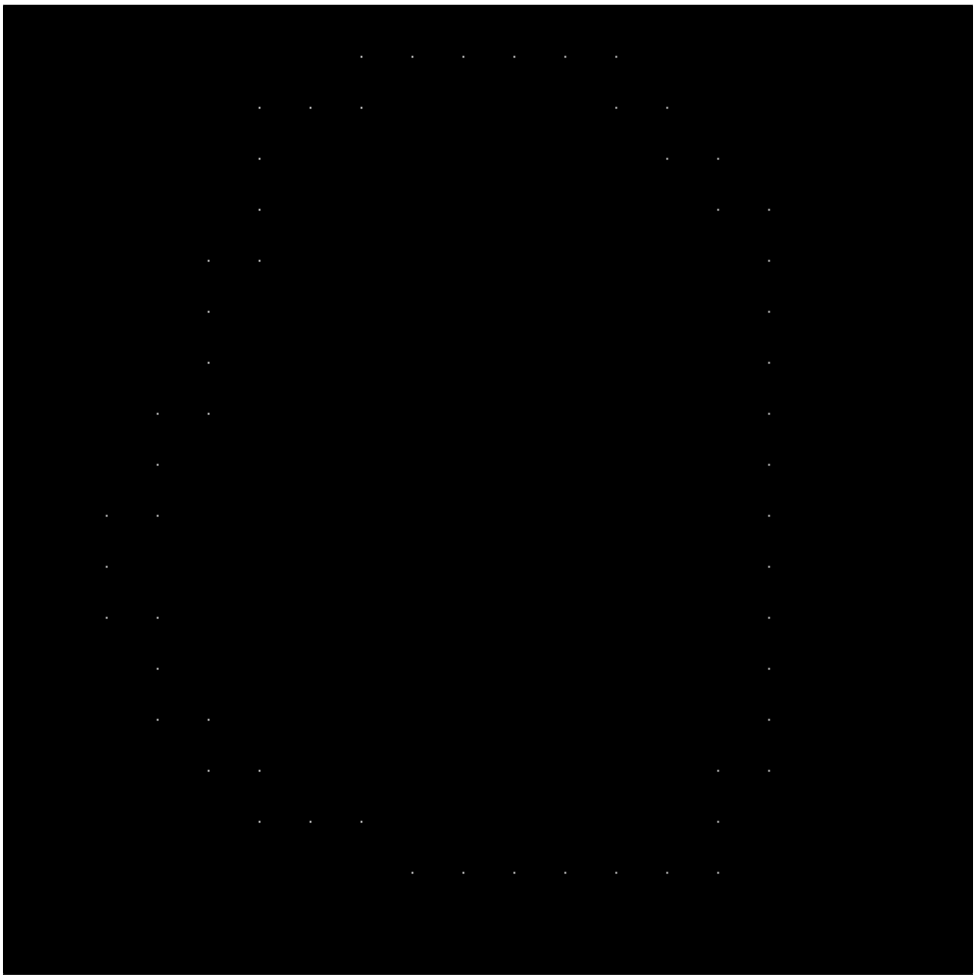

Figure J.12 – Resampling grid (S = (30, 30))

Chaincode (length = 56) :

00060606066666666666466444444344242422422022022202220020

First difference (length = 55) :

00626262600000000006206000007106262602606206200620060 26

J.6 Principal components

python pc.py –diff -n 2

We keep only 2 principal components for the images reconstruction.

Figure J.13 – Band 1

Figure J.14 – Band 1 reconstructed

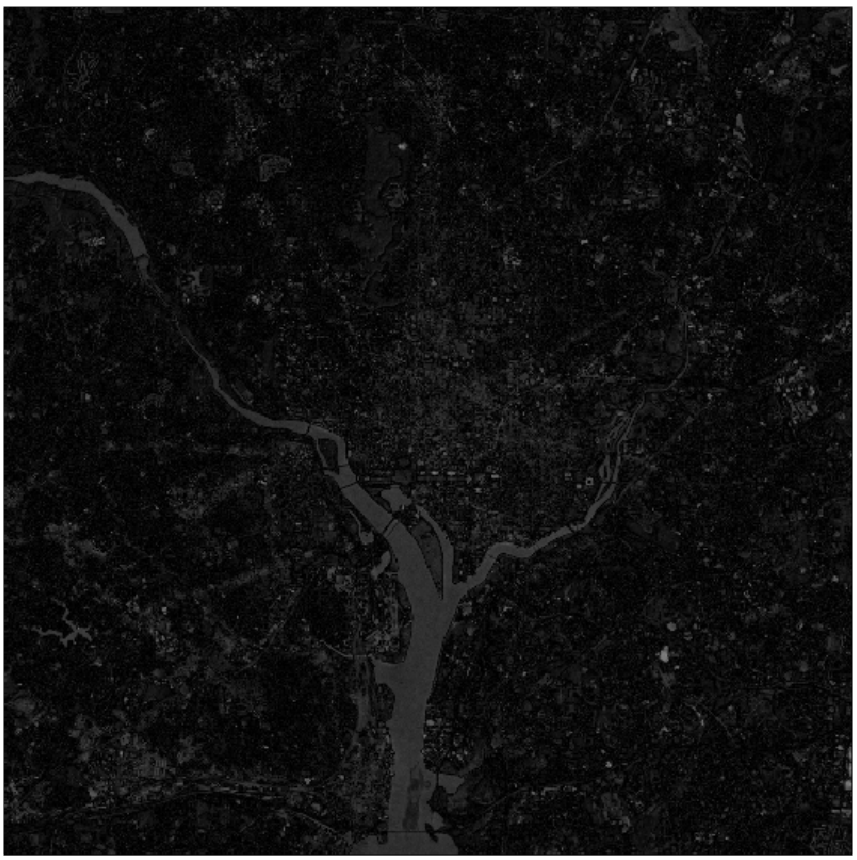

Figure J.15 – Band 1 difference

Figure J.16 – Band 2

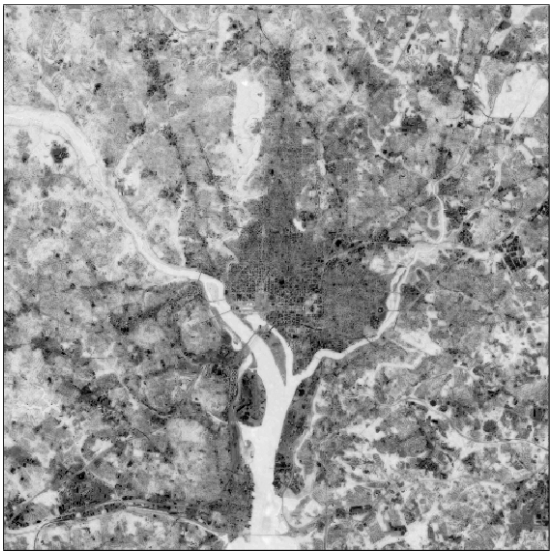

Figure J.17 – Band 2 reconstructed

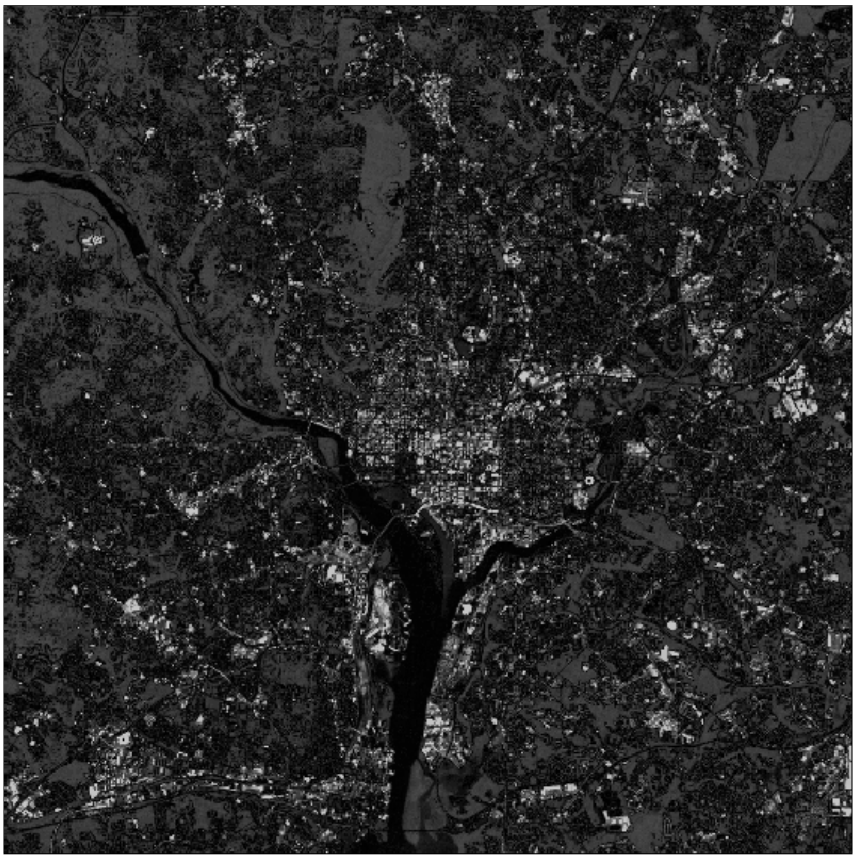

Figure J.18 – Band 2 difference

Figure J.19 – Band 3

Figure J.20 – Band 3 reconstructed

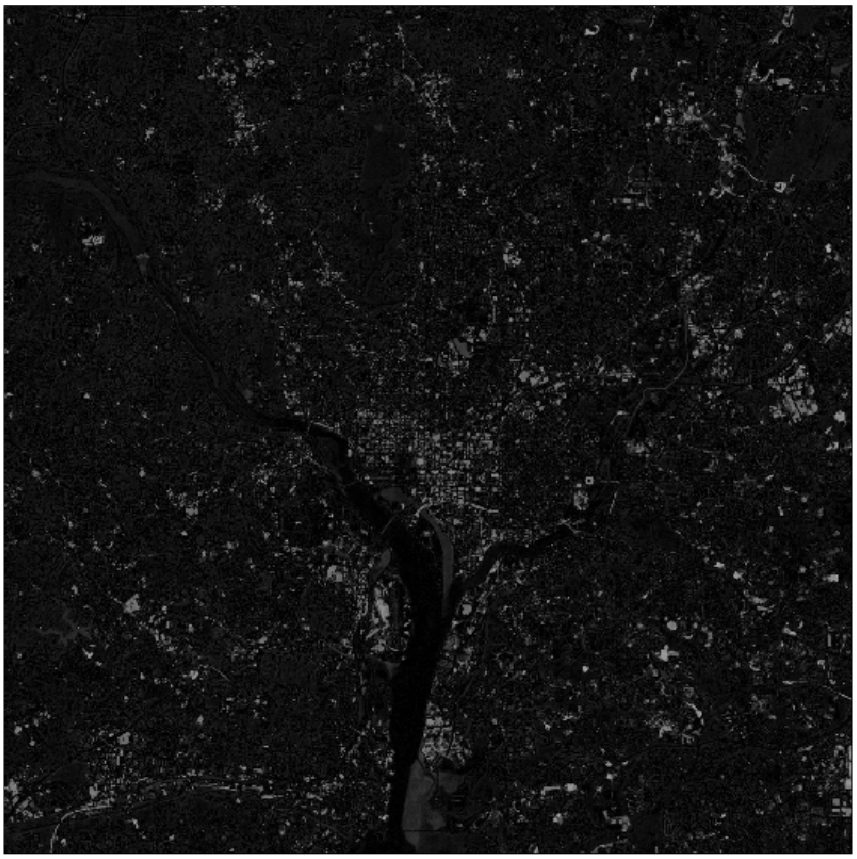

Figure J.21 – Band 3 difference

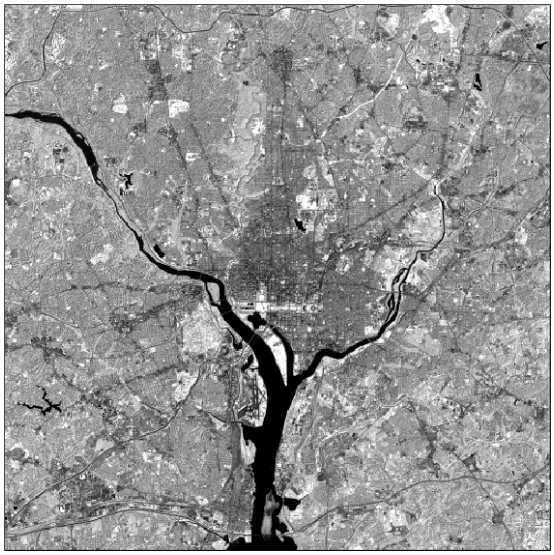

Figure J.22 – Band 4

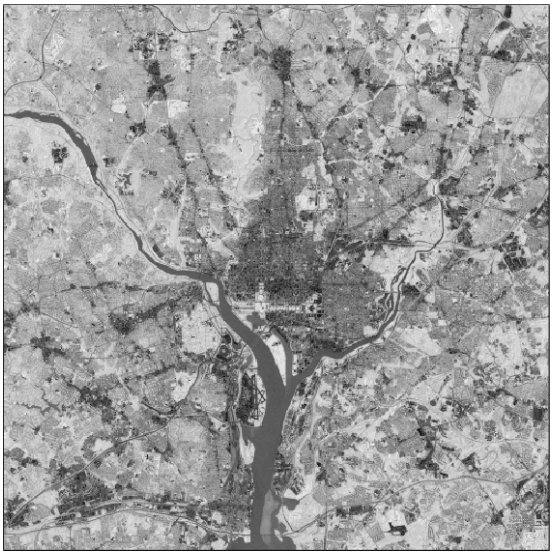

Figure J.23 – Band 4 reconstructed

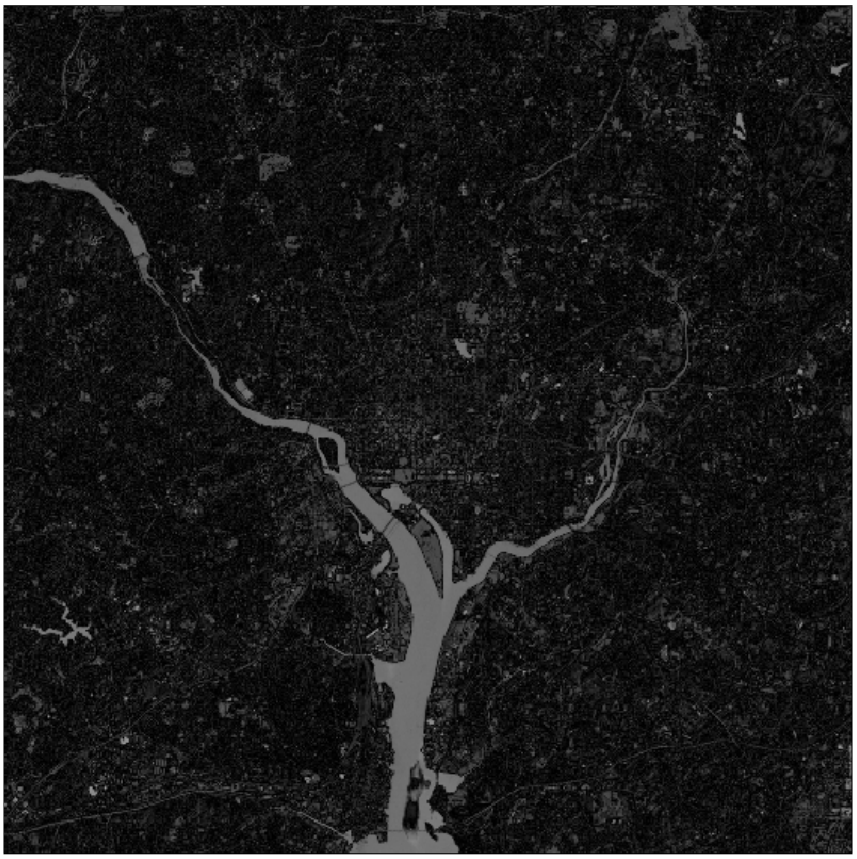

Figure J.24 – Band 4 difference

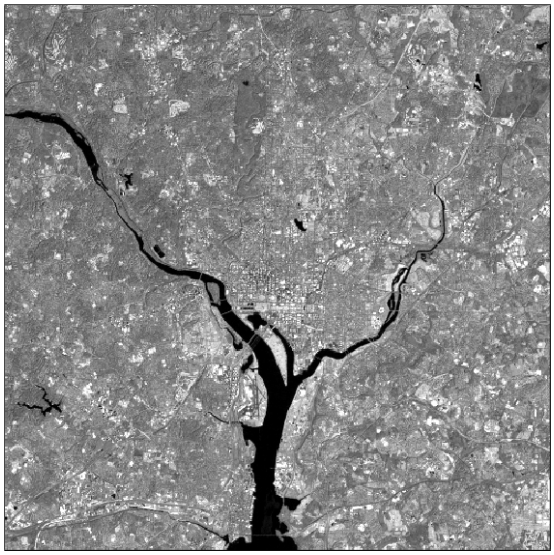

Figure J.25 – Band 5

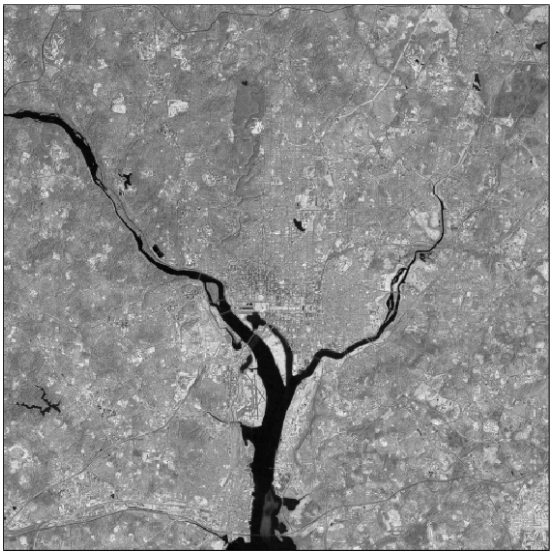

Figure J.26 – Band 5 reconstructed

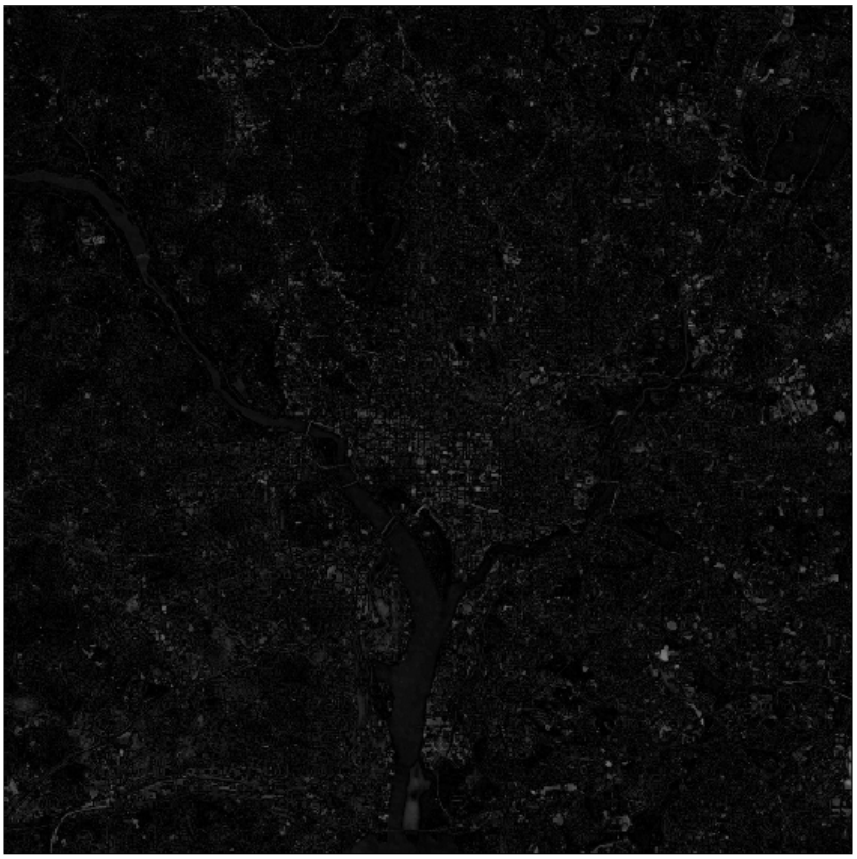

Figure J.27 – Band 5 difference

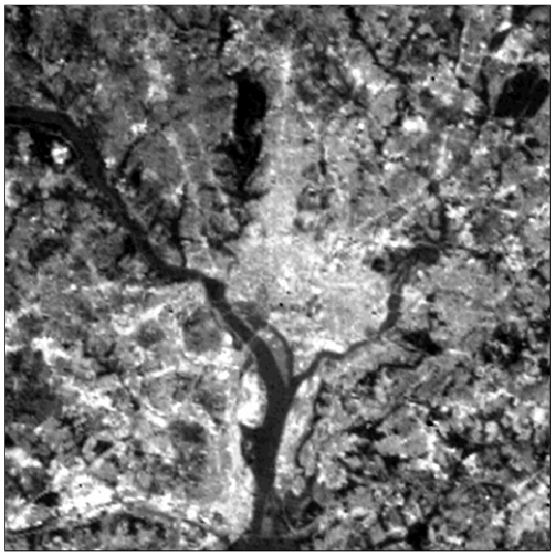

Figure J.28 – Band 6

Figure J.29 – Band 6 reconstructed

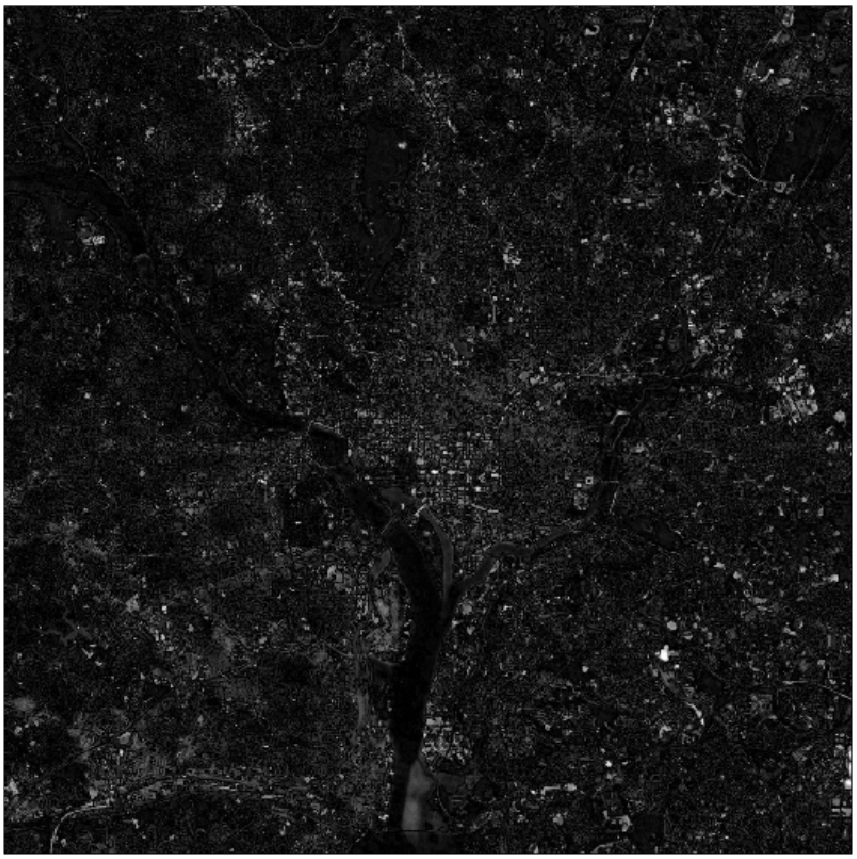

Figure J.30 – Band 6 difference