ECE572 – Digital Image Processing

Project4 – Image restoration

Name: Sang-hyeb(Sam) Lee
NetID: slee91
Student ID: 000330428

98+20

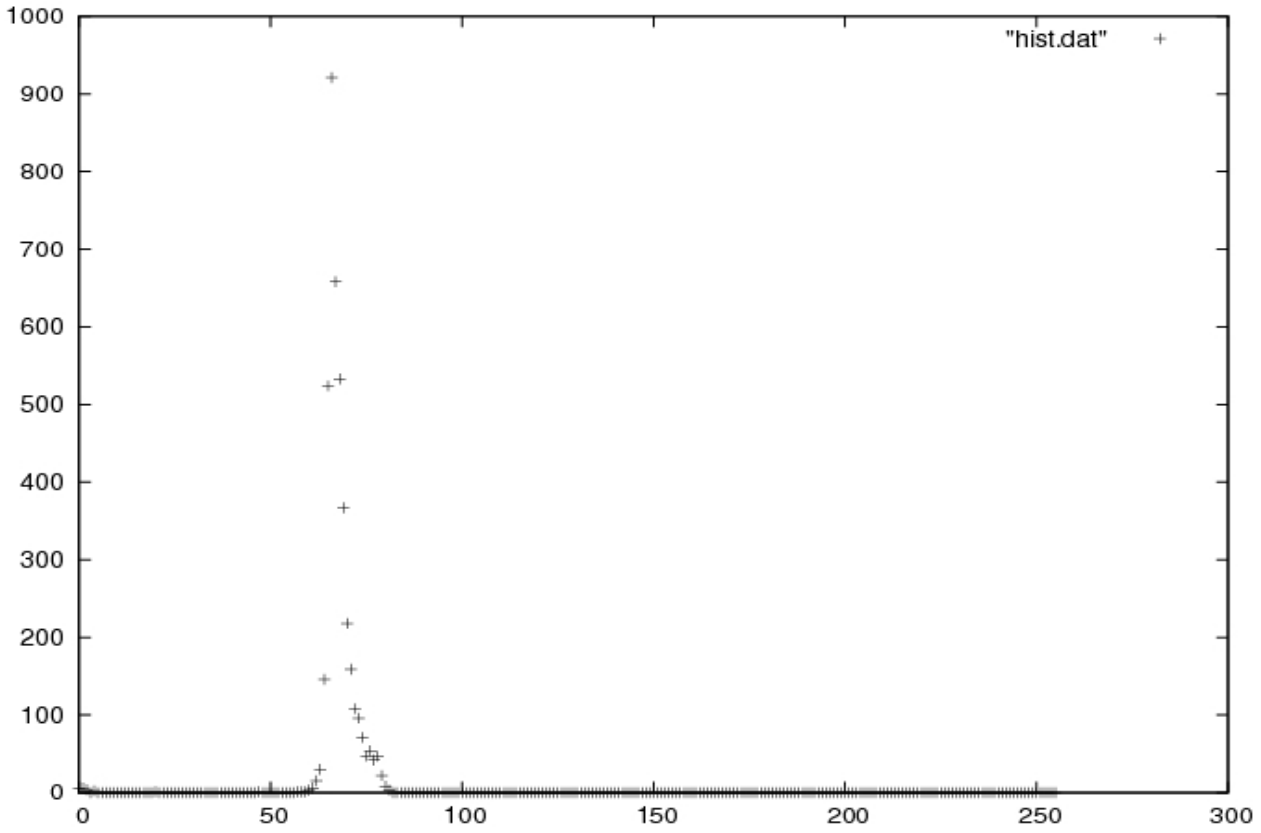
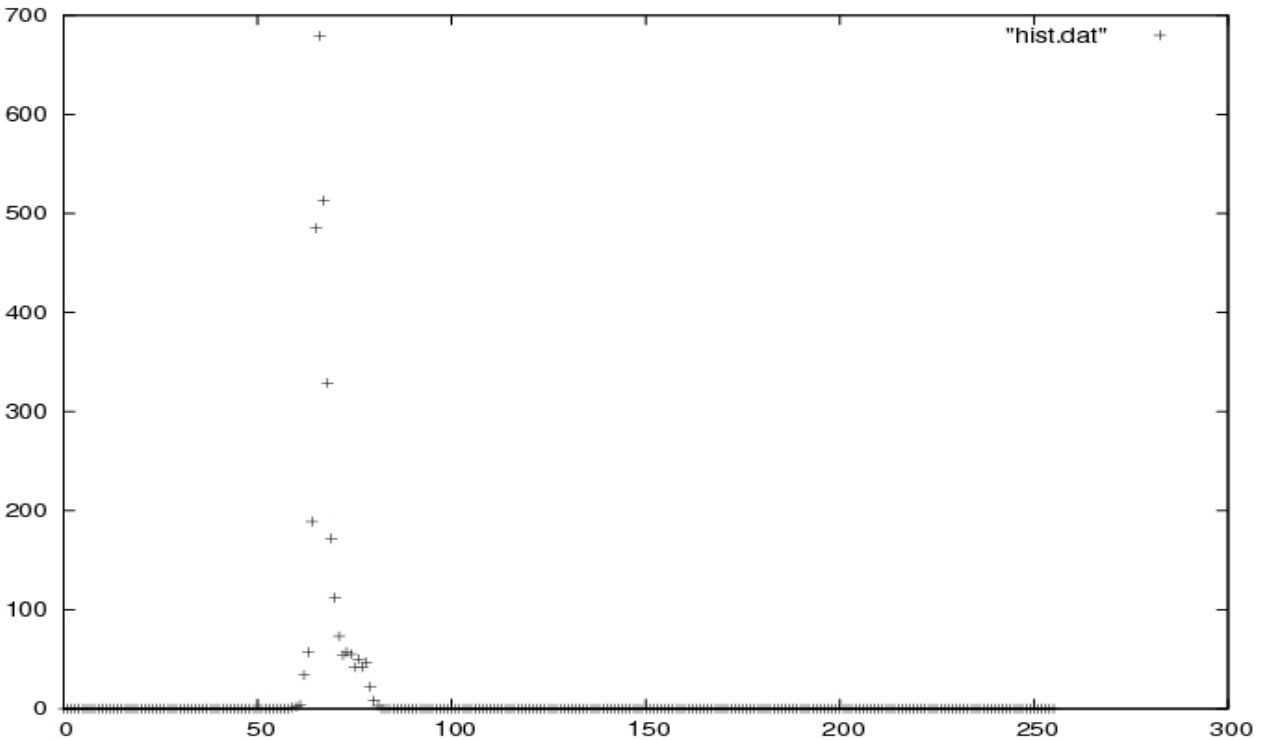
Abstract

The objectives of this project are to implement various geometric transformation, denosing and deblurring techniques, and also explore characteristics of each filter by generating sample images using each technique.

Task1.1a) Show the histogram you used to estimate the noise model in angiogram.

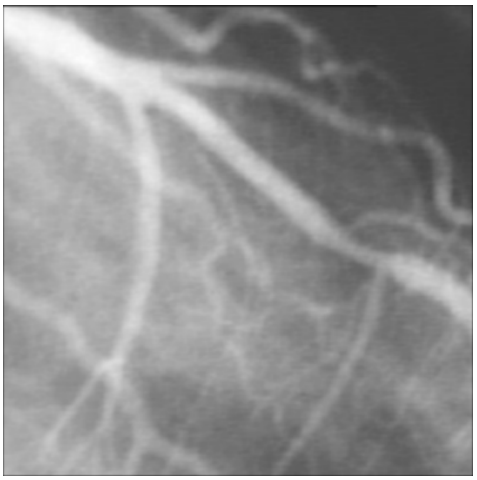
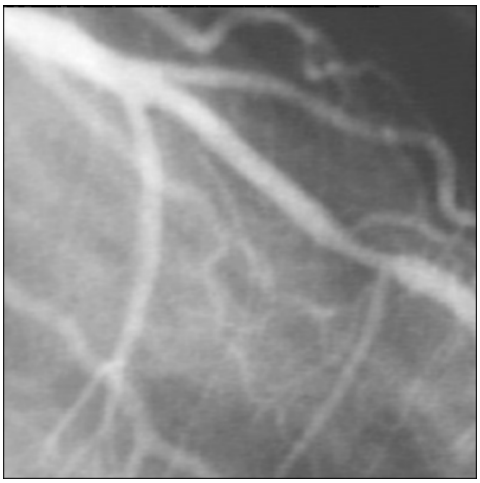
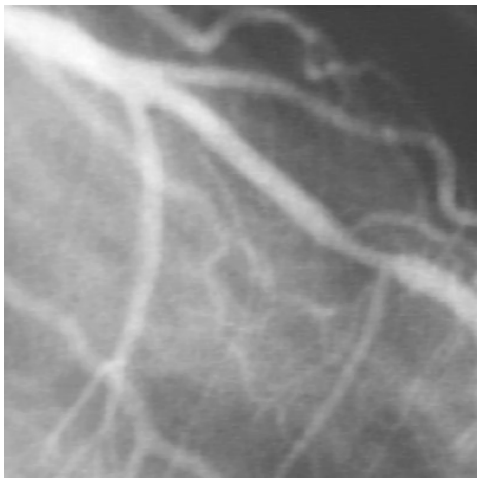
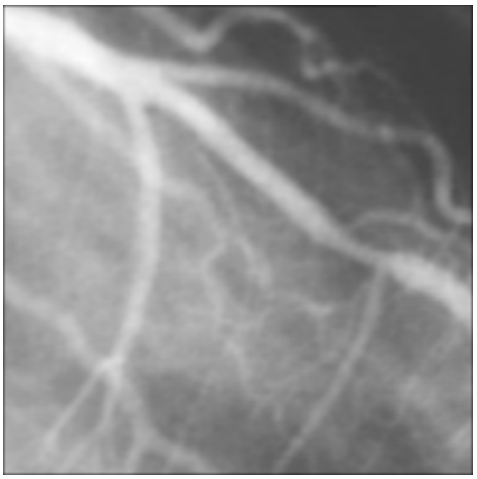
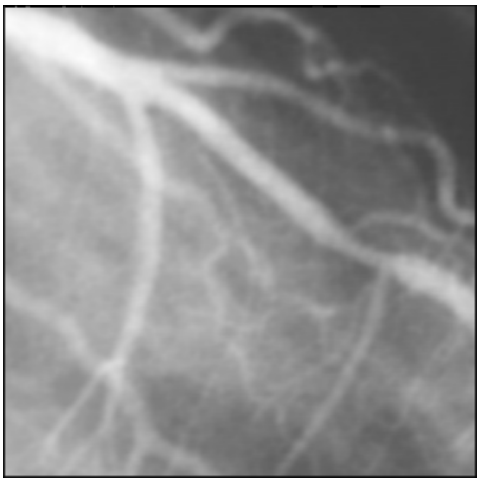
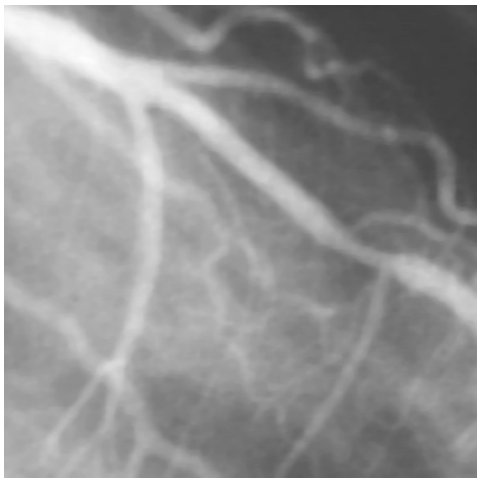
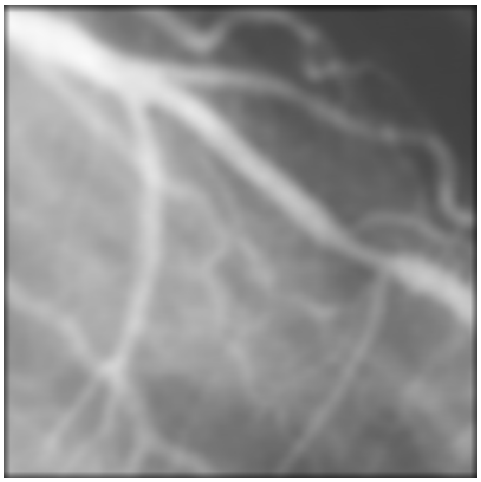
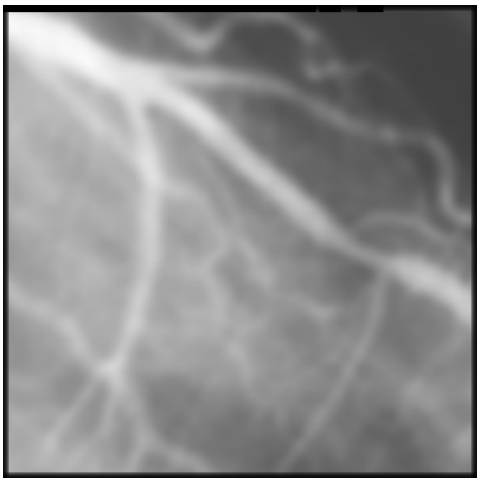
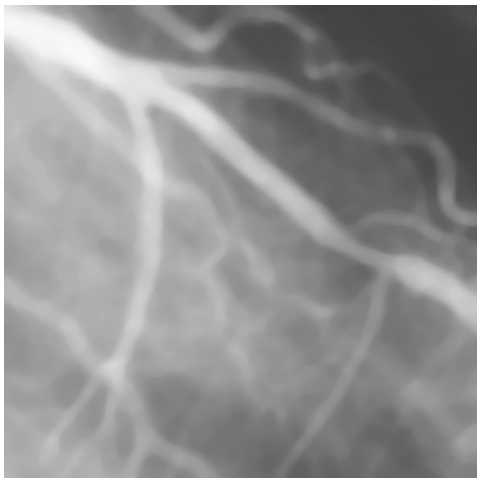
In order to estimate the noise model in angiogram. I took two small patches of reasonably constant background intensity as shown below. Based on the shape of the distribution, I suspect that the noise might be Gaussian.

..

Patch	Histogram
 Located at (0,300) – (40,399)	
 Located at (0,370) – (100,399)	

Task1.1b) Show results from amean, gmean, and median and comment on the results

Here are the results I obtained using amean, gmean, and median filter.

Kernel Size	Arithmetic mean filter	Geometric mean filter	Median filter
3 x 3	 amean with kernel 3x3	 gmean with kernel 3x3	 median with kernel 3x3
5 x 5	 amean with kernel 5x5		
11 x 11	 amean with kernel 11x11	 Gmean with kernel 11x11	 median with kernel 21x21

If you have a look at results obtained by using arithmetic mean filter. You will notice that the results are more blurred compared to the results obtained from other filters. Arithmetic mean filter is good at removing uniform and Gaussian

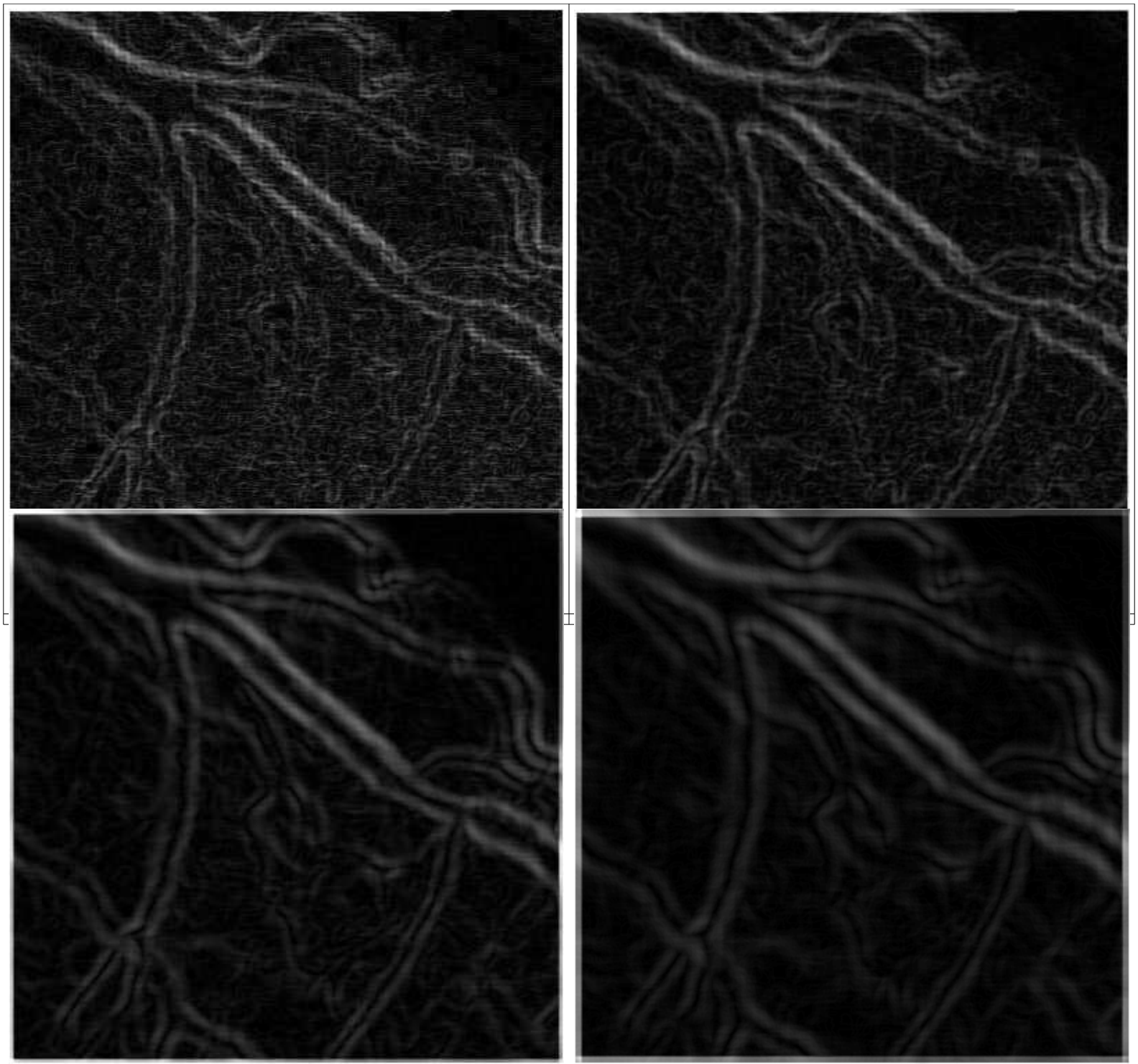
type filter at the cost of blurring the image.

Geometric median filter also introduces some blurring as the kernel size gets bigger. However, it preserves edges features better than arithmetic mean filter does. Moreover, it is shown to be effective against Gaussian type noise. However, it is very sensitive to pepper noises.

Median filter preserves edges in the image better than other two mean filters do. It works particularly well with salt-and-pepper noise.

Task1.1c) Show the edge images generated from both the noisy image and the restored image. Comment on the results.

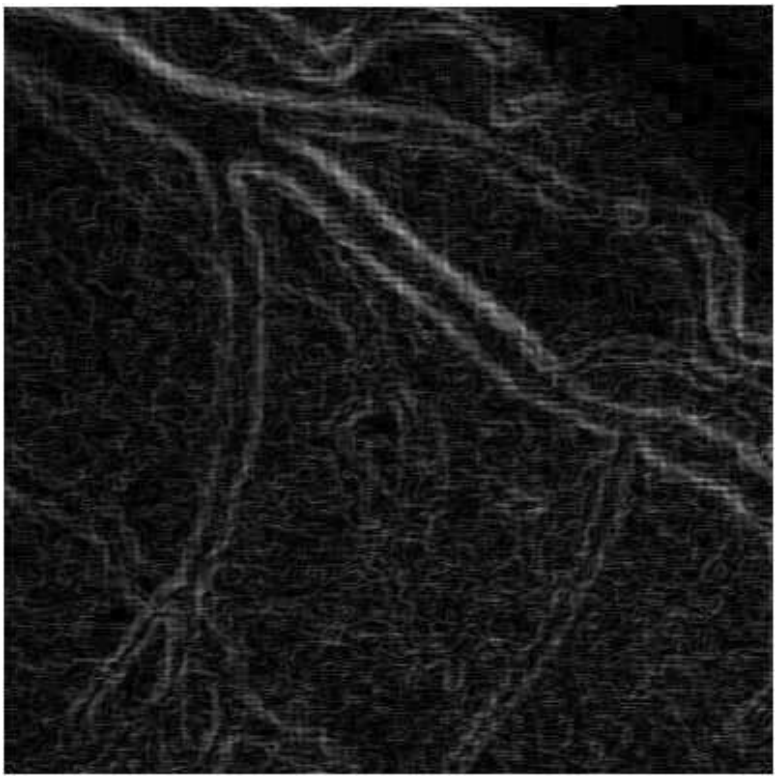
1. Arithmetic Mean Filter



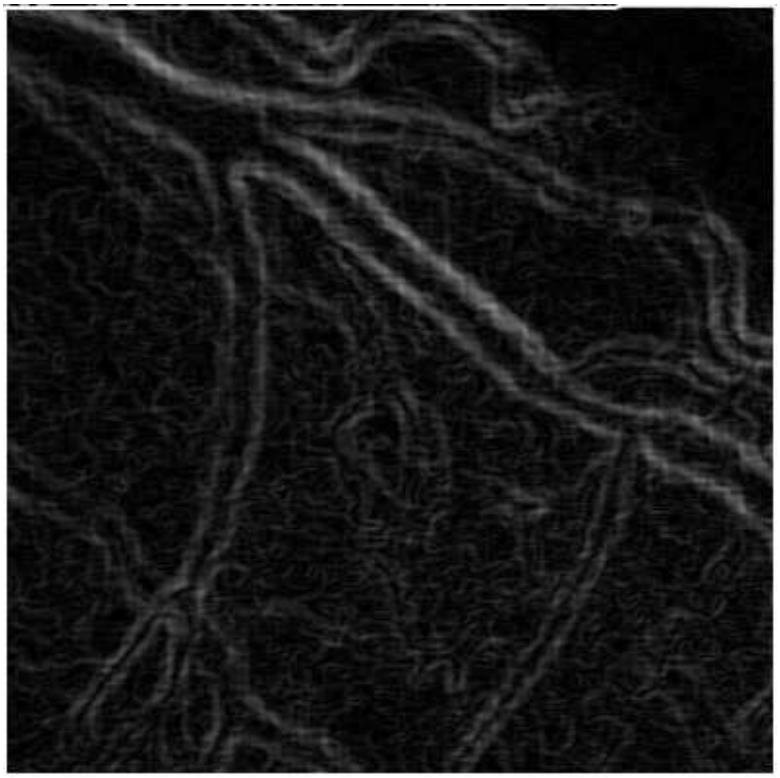
(Edges of the the image after arithmetic mean filtering with kernel 5x5)	(Edges of the the image after arithmetic mean filtering with kernel 5x5)

As you can see above, edges are getting more blurred as the kernel size increases. This is because arithmetic mean filter tends to blur the edges as well.

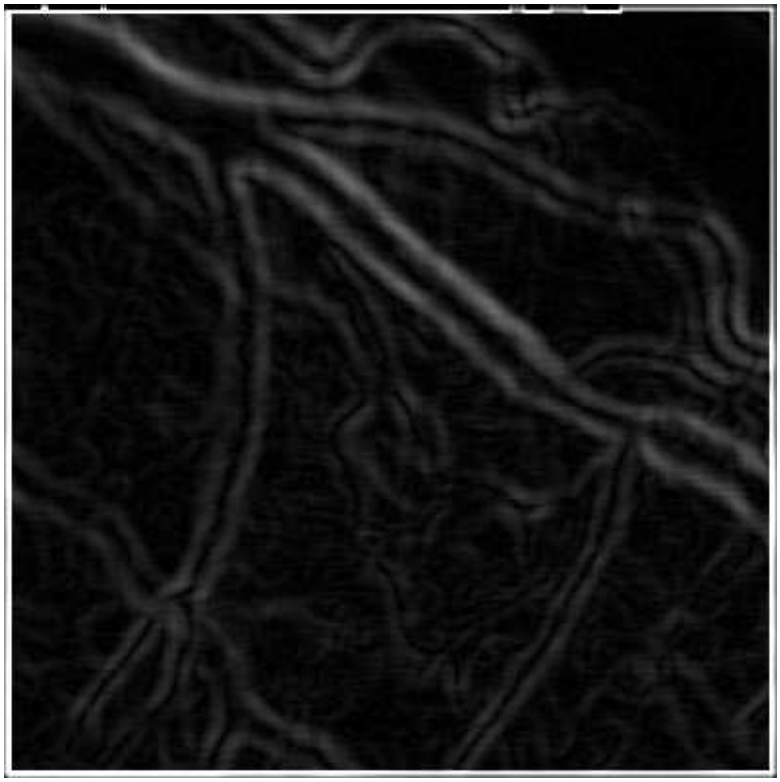
2. Geometric Mean Filter



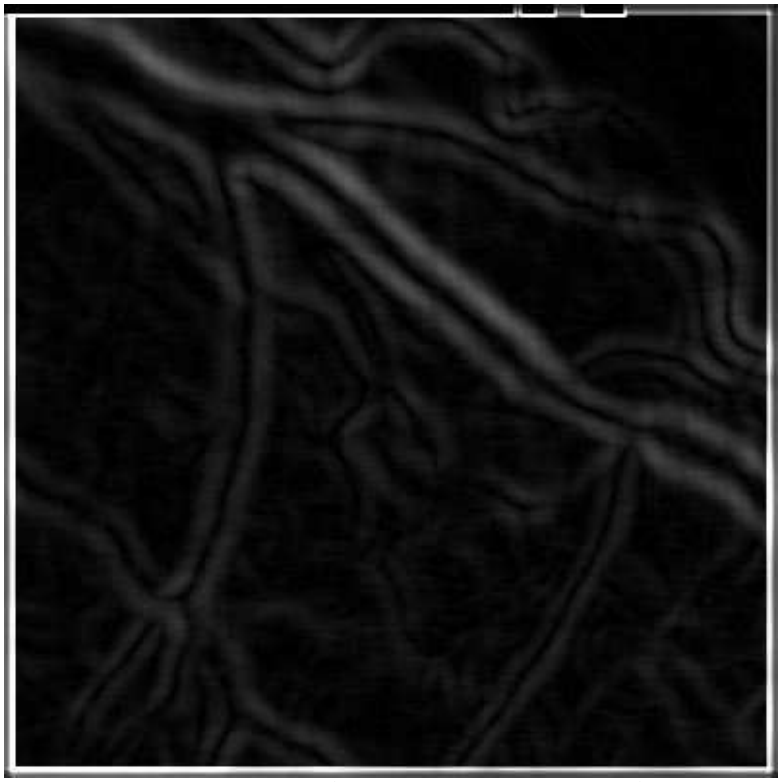
(Edges of the original image)



(Edges of the the image after geometric mean filtering with kernel 3x3)



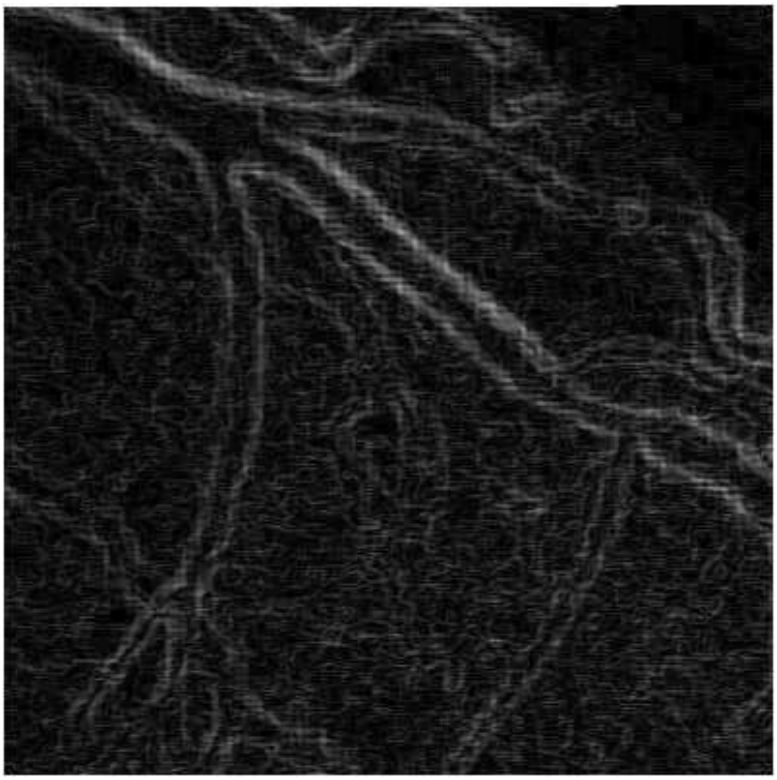
(Edges of the the image after geometric mean filtering with kernel 7x7)



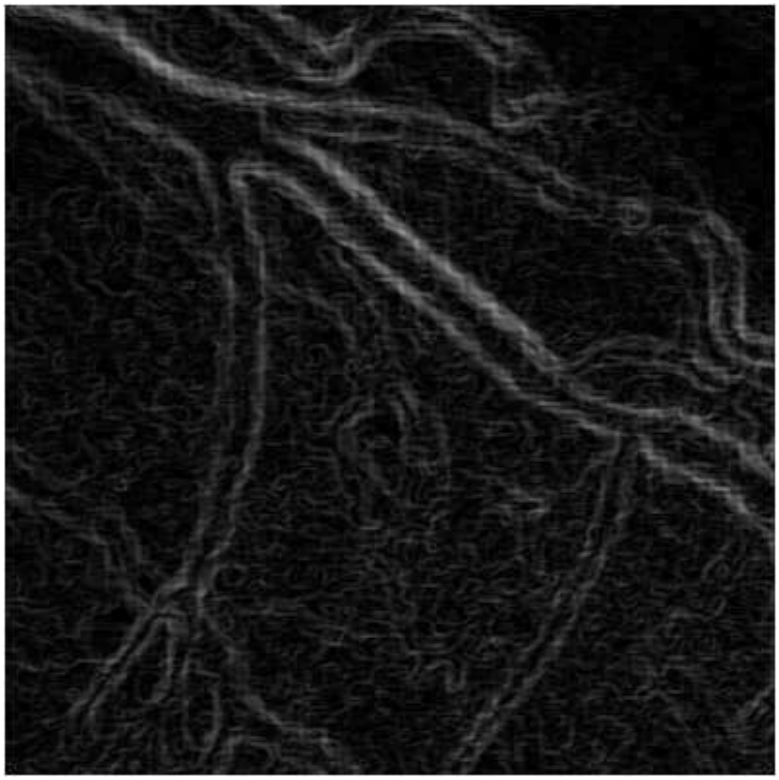
(Edges of the the image after geometric mean filtering with kernel 11x11)

Edges are more blurred with the bigger size of the kernel. However, if you have a look at both images generated by arithmetic mean filter and geometric mean filter with kernel size 11 respectively, it is obvious that geometric mean filter preserves edges better than arithmetic mean filter does.

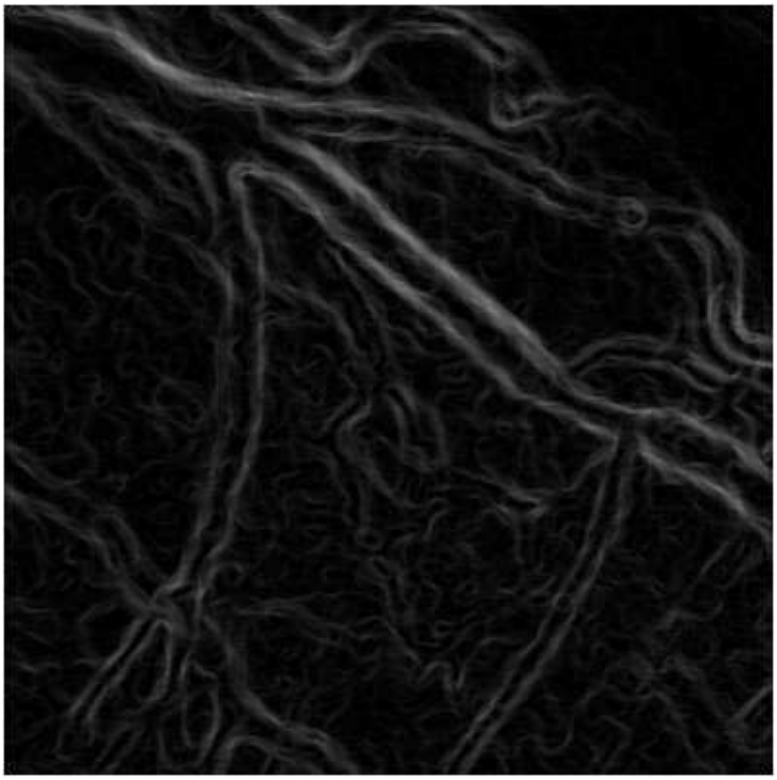
3. Median filter



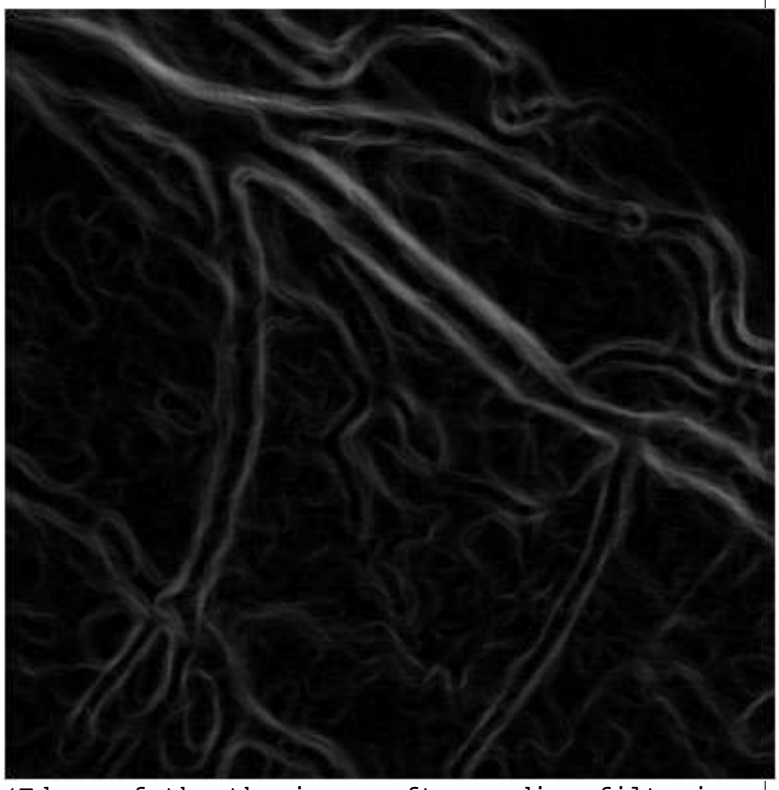
(Edges of the original image)



(Edges of the the image after median filtering with kernel 3x3)



(Edges of the the image after median filtering with kernel 7x7)



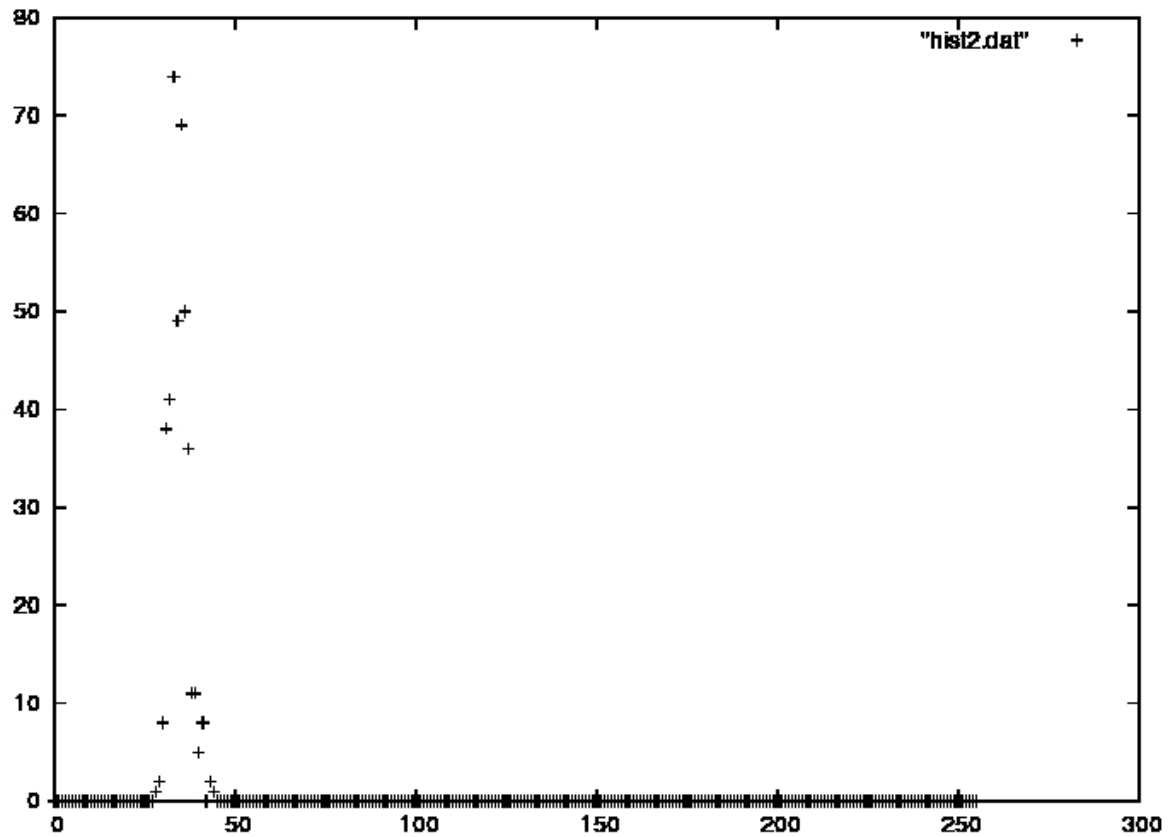
(Edges of the the image after median filtering with kernel 11x11)

Edge images generated from restored images with median filter present more clear edges compared to the ones generated by arithmetic and geometric mean filters. Median filter preserves edges features better than the other two filters.

Task1.2) Show results from contrah, median, and amedian on the wizardofoz image. Comment on the result images. Again, which filter is more suitable for what kind of noise. Show results of applying contrah with Q being both negative and positive value.

* Estimation of the noise model in wizarodfoz.pgm

By looking at the image carefully, we can notice that this image contains **salt noise.** Moreover, in order to examine the non-impulsive noise model in wizardofoz image, I took a small pactch of reasonably constant background intensity. Below is the histogram of the patch. Based on the shape of the distribution, I suspect that the noise might be **Gaussian.** Therefore, we can conclude that wizarodof.pgm has **both salt and Gaussian noise.**

PATCH	Histogram
 Located at (241,125) – (254,153)	

1. contrah

(Kernel size=3x3, Q= -1.5)

(Kernel size=5x5,Q=-1.5)

(Kernel size=3x3, Q= 1.5)

(Kernel size=5x5, Q=1.5)

(Kernel size=3x3, Q=0)

(Kernel size=5x5, Q=0)

Contra-harmonic mean filter is very interesting as we can control the behavior of the filter by adjusting the the value of Q. When the value of Q is positive, it

reduces the pepper noise. When the value of Q is negative, it reduces the salt noise. When the value of Q is zero, it behaves just like an arithmetic mean filter.

If you have a look at the images in the first row where a negative value of Q, -1.5, is used, the filter virtually removed the most of the salt noise. its effect can be increased by the increasing the size of the filter.

If you have a look at the images in the second row where a positive value of Q, 1.5 is used, the filter did not remove any salt noise. This is because the positive value of Q removes only pepper noise.

If you have a look at the images in the third row where Q is set to zero, the filter just removed little bit of the salt noise and blurred the image. This is because the contra-harmonic filter behaves likes an arithmetic mean filter when Q is set to zero.

2. amedian

(starting kernel size=3x3, maximum kernel size=21x21)

(starting kernel size=5x5, maximum kernel size=21x21)

Surprisingly, adaptive median filter did not successfully remove the noise in the image. It is because that there are both Gaussian and salt noises in the original image. When both types of noise are present, adaptive median filter can not remove impulsive noise. However, adaptive median filter successfully removes when the image only consists of one type of noise, impulsive or non-impulsive noise.

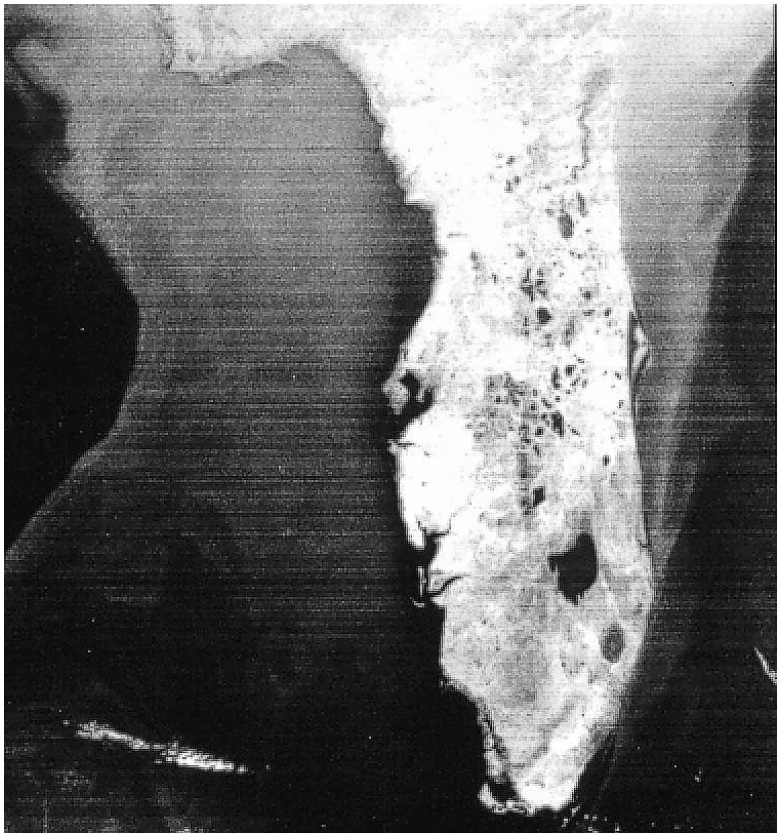
3. median

 (Kernel size=3x3)	 (Kernel size=5x5)

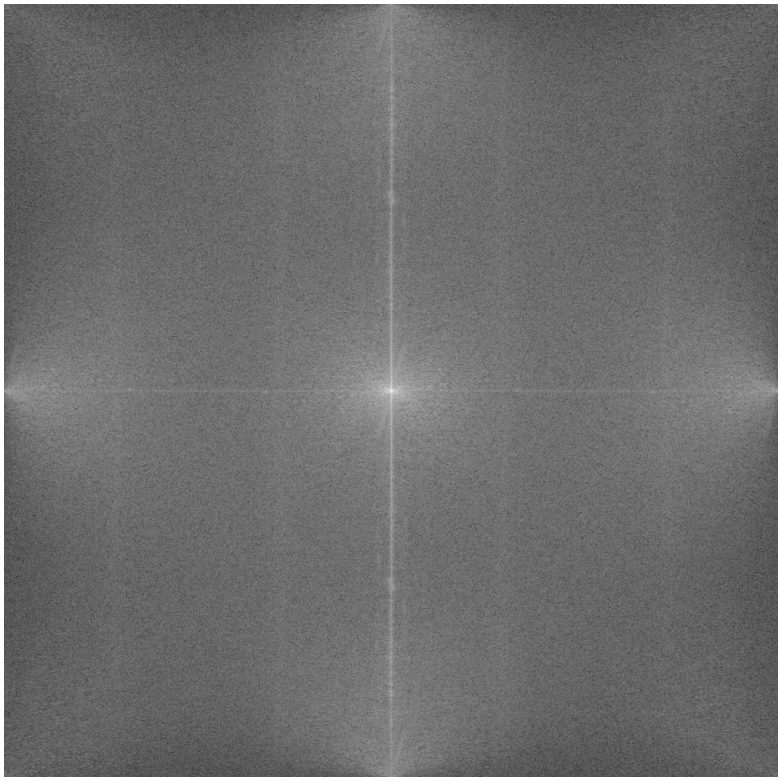
Median filter removes salt and Gaussian noises, and the effect is increased with the bigger kernel size. Edge features are better preserved in this image compared to contra-harmonic noise. Median filter normally does a good job of handling impulsive and non-impulsive noise. It particularly works well with salt and pepper noise.

Task1.5a) Show the frequency reponse to Florida satellite.

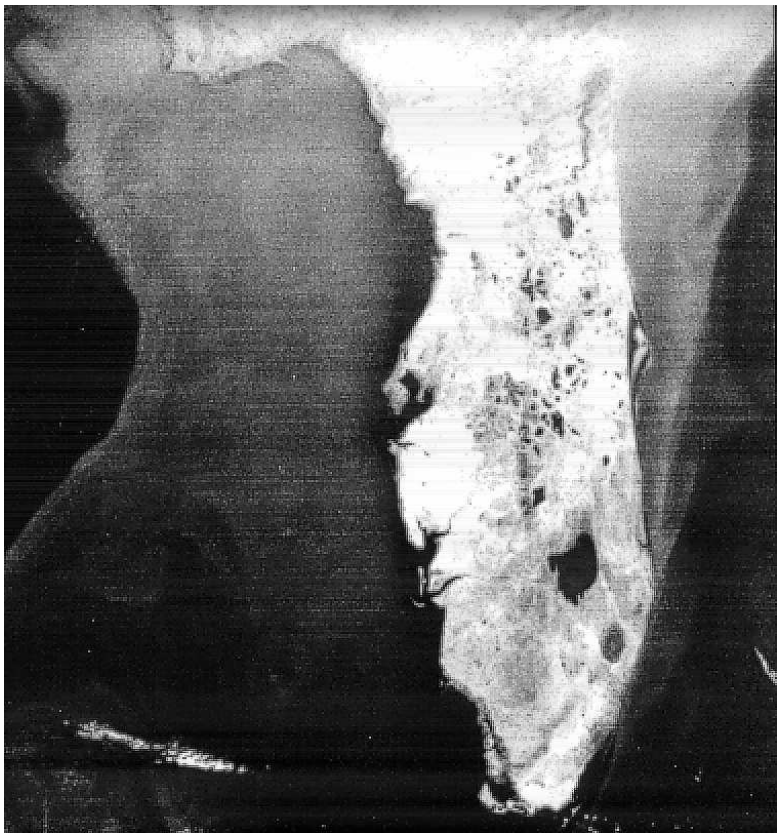
Below is a frequency response to Florida satellite.



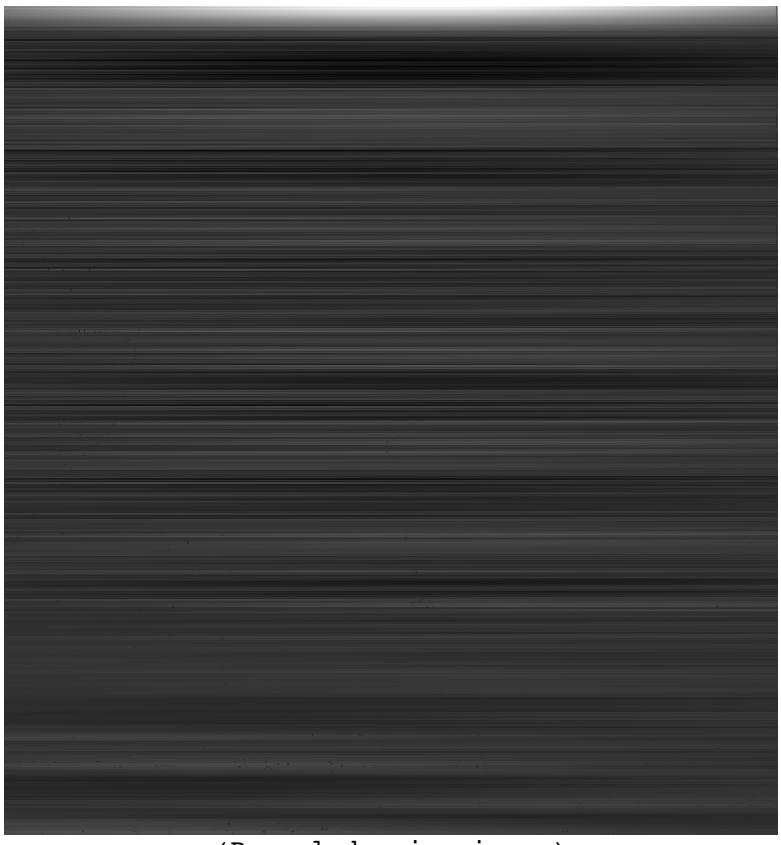
(Original Image)



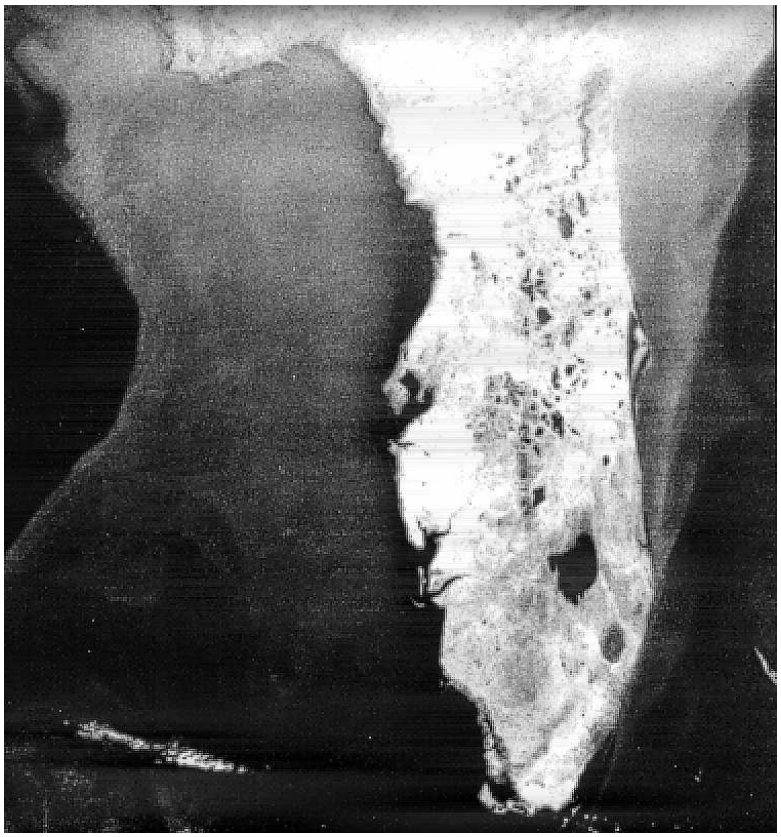
(Spectrum of the original image)



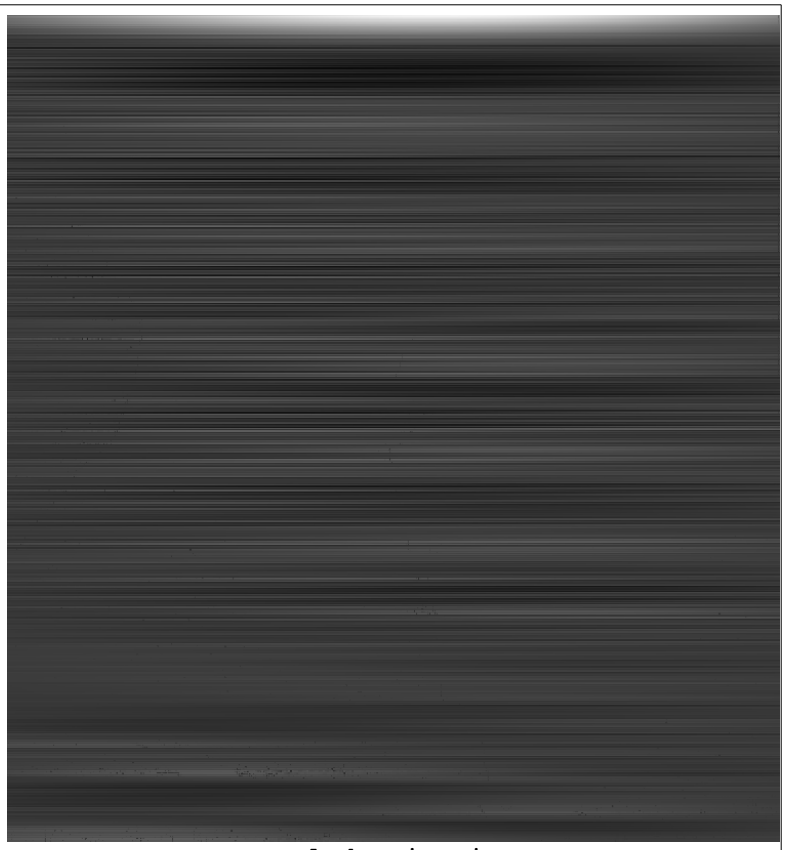
(Notch reject filter with D_0=20,width=3)



(Rescaled noise image)



(Notch reject filter with D_0=20,width=53)



(rescaled noise image)

Task 2.1a) Show the original lena, the blurred version, and the blurred+noise version(g).

(original image)

(blurred with 3x3 Gaussian kernel)

(blurred with 3x3 Gaussian kernel & SAP noise with probability 0.25)

Task2.1b) Show the results after applying amedian. You should probability try different Smax.

(amedian with Smax=5, starting kernel=3)

(amedian with Smax=7, starting kernel=3)

(amedian with Smax=15, starting kernel=3)

(amedian with Smax=21, starting kernel=3)

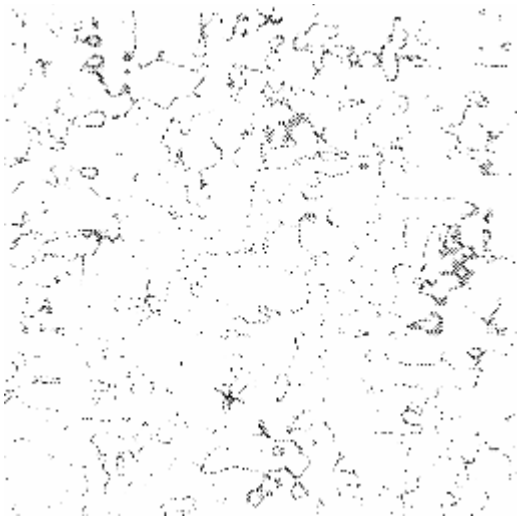
Task2.1c) Show results after applying invFilter and wiener. Provide parameters used in the functions. Comment on the differences between invfilter and wiener. Or why would invFilter fail the task?

1) invFilter

(target image)

(inverse filter with D_0=60)

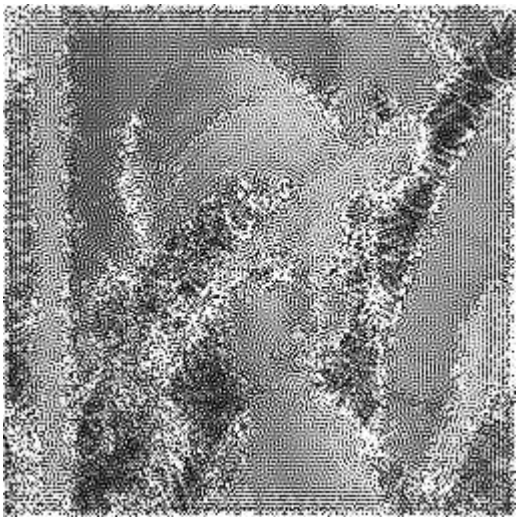
(inverse filter with D_0=70)



(inverse filter with D_0=80)

(inverse filter with D_0=80)

As you can see above. Inverse filter does not successfully deblur the image. This is because that some elements of the matrix H are very small, so division produces dramatically large values which dominates the output. I used a thresholding value(0.01) to overcome this problem. If the value of the element in H is less than 0.01, I do not perform the division for that pixel. These are the image I generated with improved inverse filter.



(modified inverse filter with D_0=80)

(modified inverse filter with D_0=100)

2) wiener

(target image)

(wiener filter with D_0=100, K=0.5)

(wiener filter with D_0=100, K=0.2)

(wiener filter with D_0=100, K=0.3)

Unlike inverse filter, Wiener filter does not suffer from the same problem as the inverse filter does. Inverse filter produces incorrect outputs when the degradation function has zero or very small values. Wiener filter avoids this problem by introducing |H(u,v)^2| / (|H(u,v)^2|+K) into the equation. The wiener filter result is by no means perfect.

Task2.1d) Provide MSE and PSNR comparison with different choices of Smax and invFilter, and different choices of wiener filter parameters.

1) Different choises of Smax

(Adaptive Mean Filtering with Smax=5)
MSE: 238.85
PSNR: 12.1748

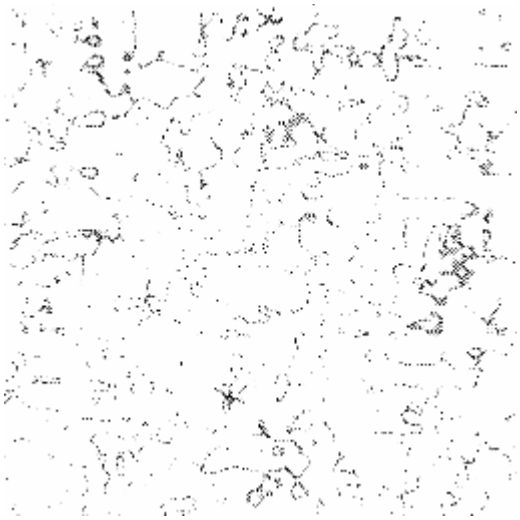
(Adaptive Mean Filtering with Smax=7)
MSE: 149.899
PSNR: 13.1864

(Adaptive Mean Filtering with Smax=15)
MSE: 147.239
PSNR: 13.2253

(Adaptive Mean Filtering with Smax=21)
MSE: 147.239
PSNR: 13.2253

The value of MSE decreases as we increase the size of Smax, and the value of PSNR increases as we increase the size of PSNR. However, the value of MSE and PSNR do change once the size of Smax is over 15.

2) Inverse Filter



(inverse filter with D_0 = 80)
MSE = 19163.1
PSNR = 2.65308

3) Wiener Filter

(Wiener filter with D_0=100,K=0.3)
MSE: 1134.03
PSNR: 8.79228

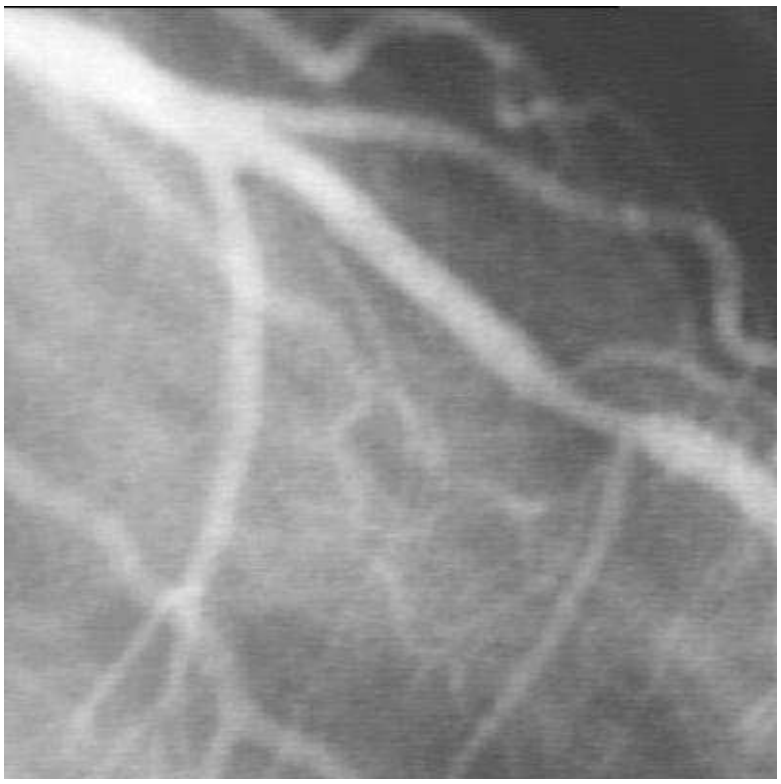
(Wiener filter with D_0=100,K=0.2)
MSE: 667.052
PSNR: 9.9446

(Wiener filter with D_0=100,K=0.1)
MSE: 298.794
PSNR: 11.6885

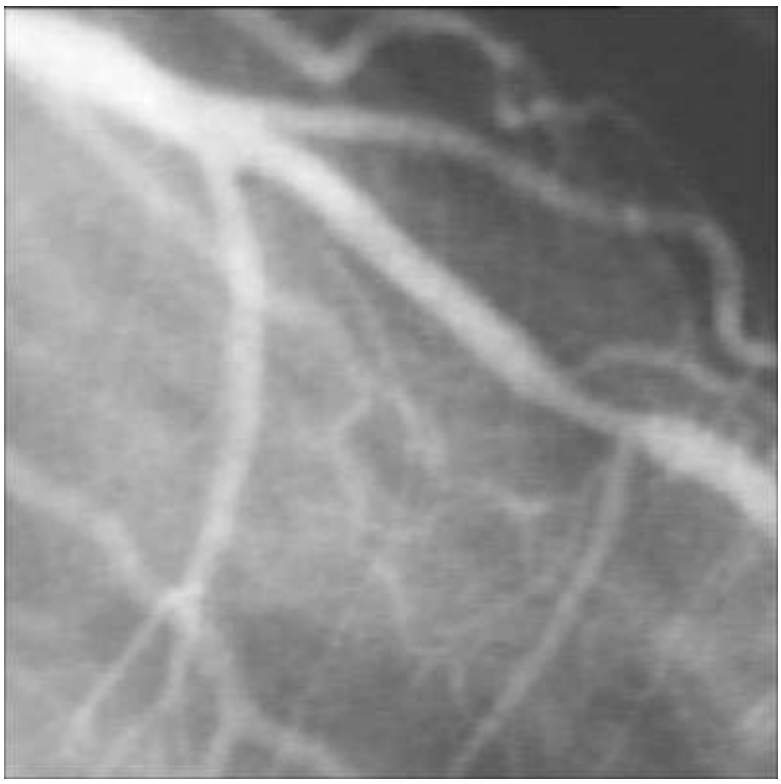
(Wiener filter with D_0=100,K=0.05)
MSE: 183.056
PSNR: 12.7525

Task2.2) Repeat the process with angiogram image

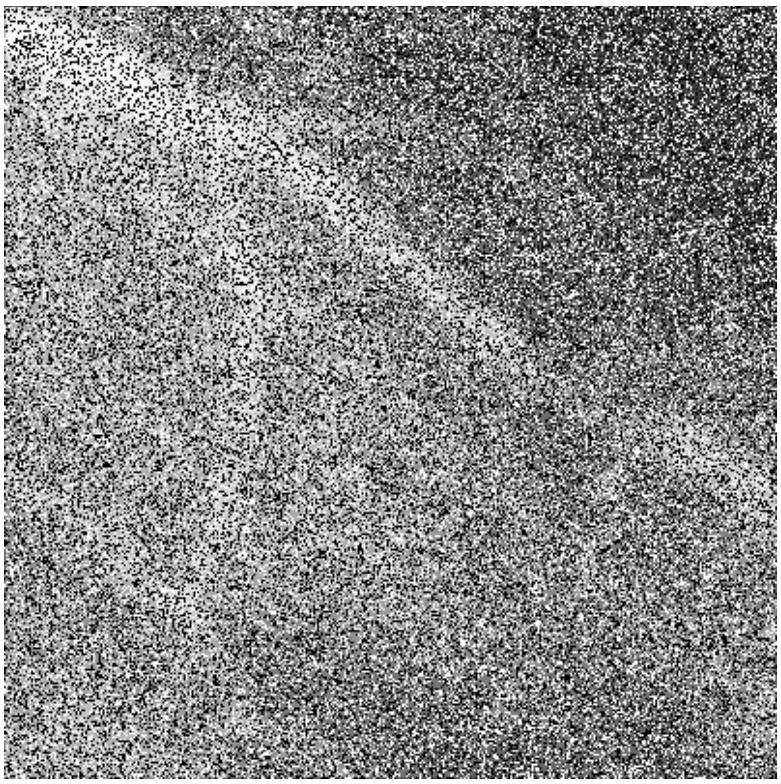
1) Show the original image, the blurred version, and the blurred+noise version



Original

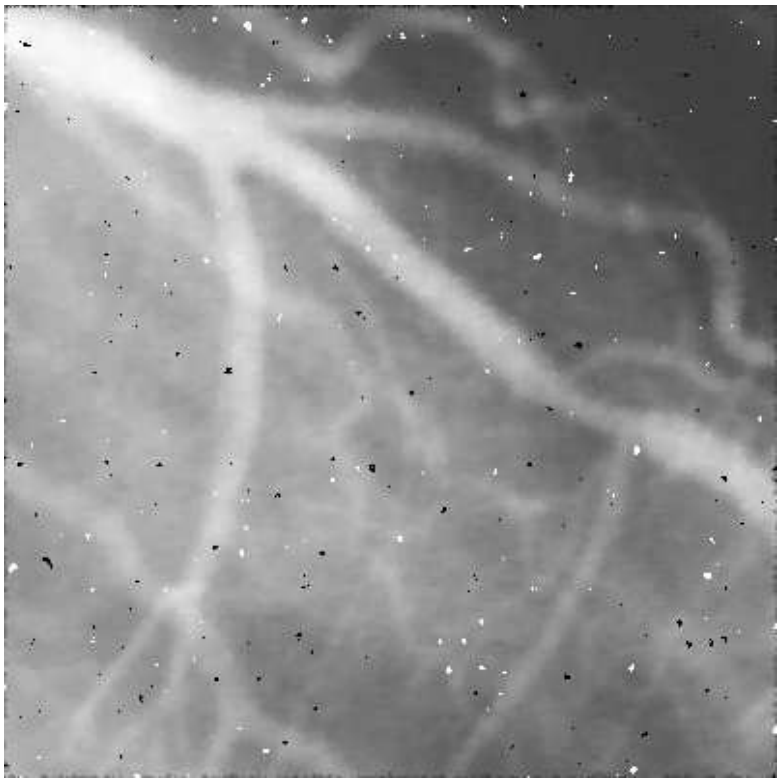


Blurred with 3x3 Gaussian kernel

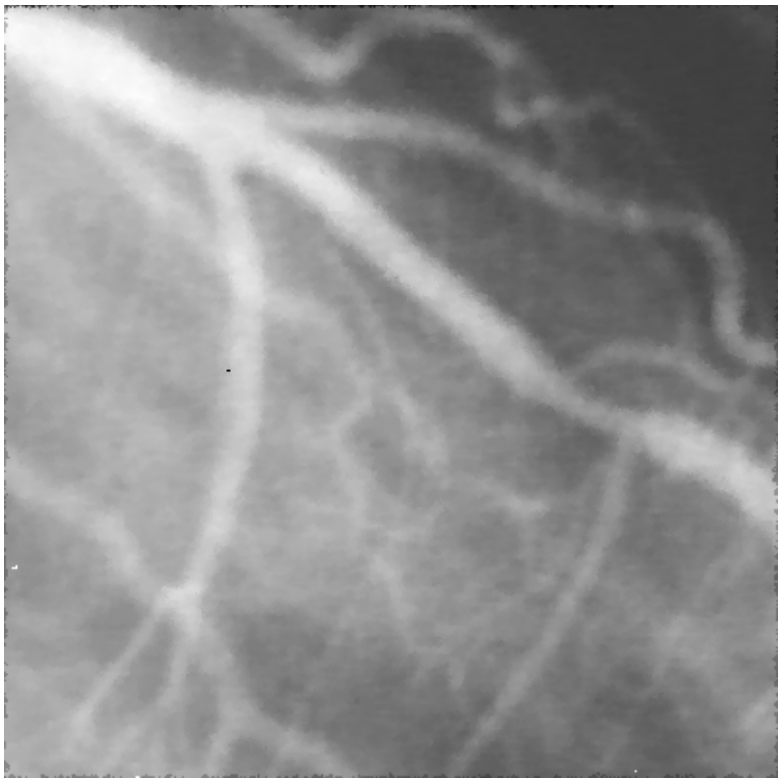


Blurred with 3x3 Gaussian kernel+ Salt and Pepper Noise with probability 0.25

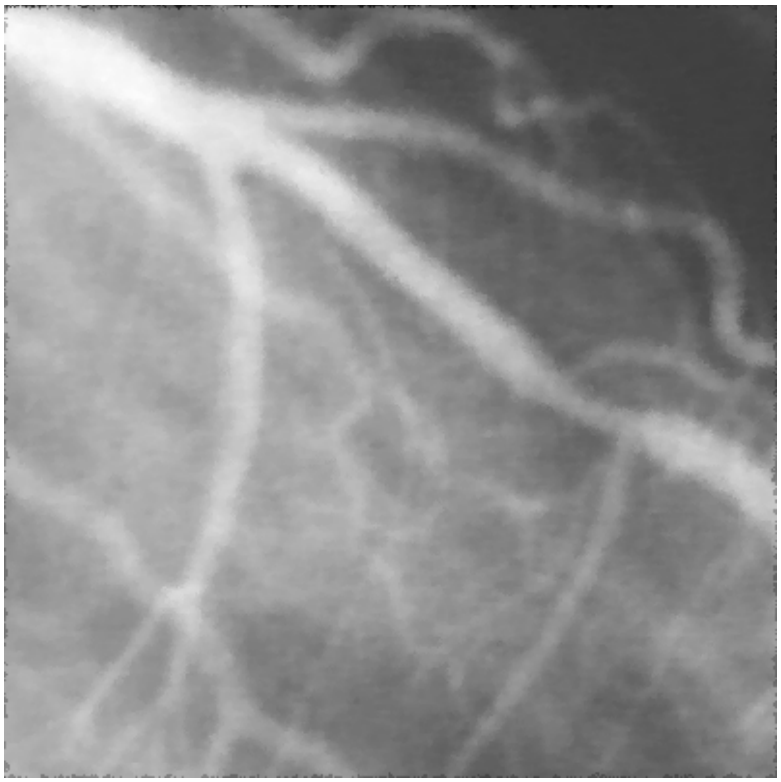
2) Apply adaptive median filter to remove the noise.



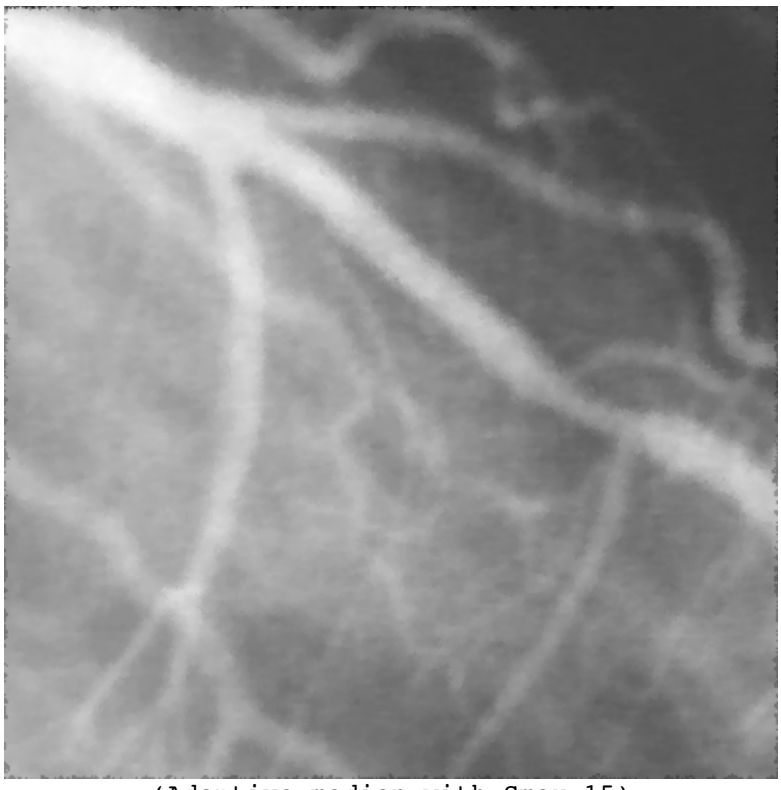
(Adaptive median with Smax=5)
MSE: 108.357
PSNR: 13.8911



(Adaptive median with Smax=7)
MSE: 27.5398
PSNR: 16.8656

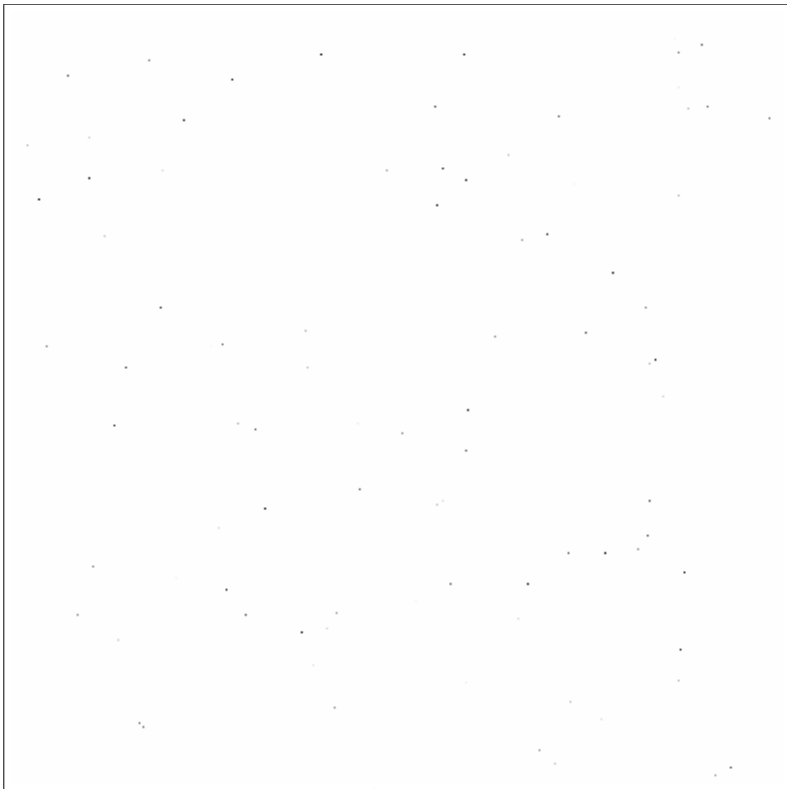


(Adaptive median with Smax=11)
MSE: 26.5949
PSNR: 16.9414

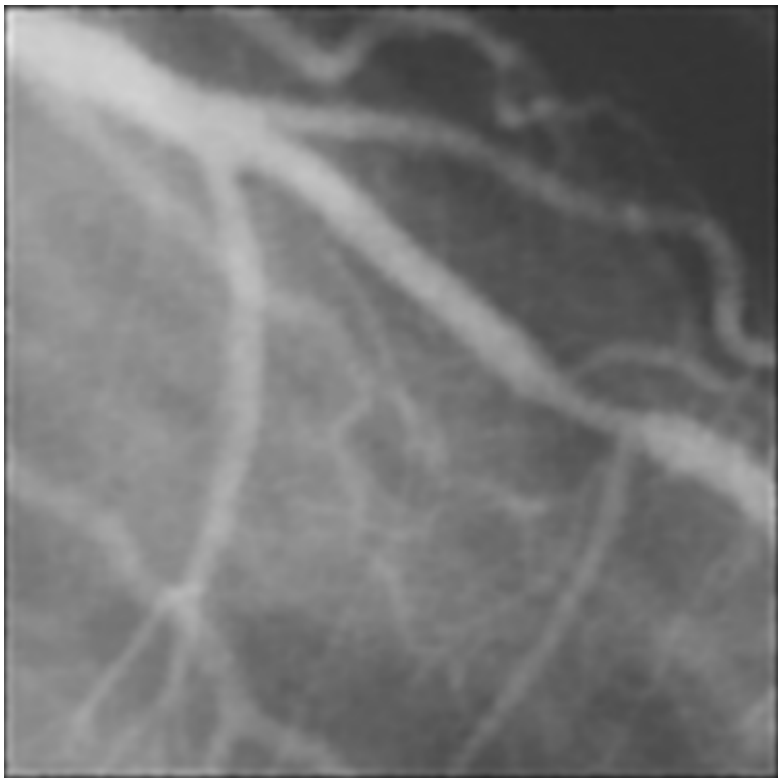


(Adaptive median with Smax=15)
MSE: 26.5949
PSNR: 16.9414

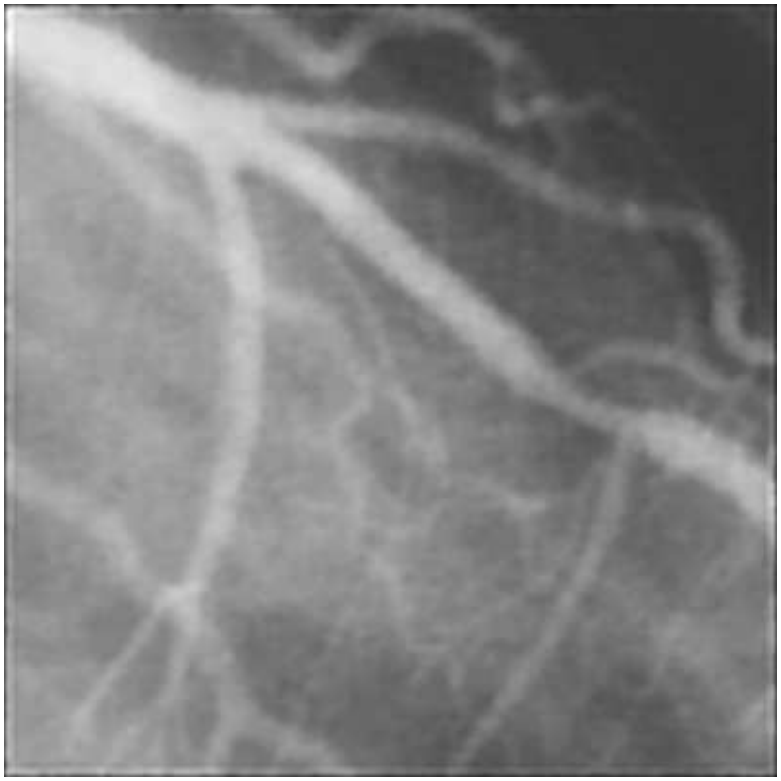
3) Apply inverse and wiener filters to deblur.



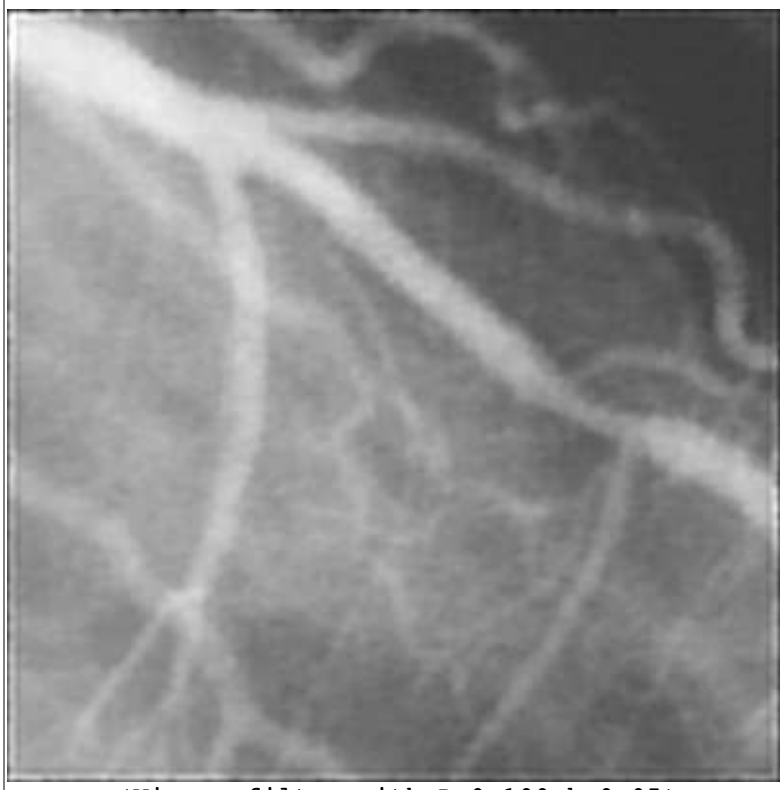
(Inverse filter D_0=100)
MSE: 13951.8
PSNR: 3.34225



(Wiener filter with D_0=100,k=0.2)
MSE: 723.306
PSNR: 9.76879



(Wiener filter with D_0=100,k=0.1)
MSE: 253.439
PSNR: 12.046



(Wiener filter with D_0=100,k=0.05)
MSE: 104.353
PSNR: 13.9729

Task3.1a) an explanation of difference between affine transform and perspective transform

(Example of perspective transformation)

Affine transformations preserve lines and parallel lines whereas the perspective transformation preserve parallel lines only when they are parallel to the project plane. If not, lines converge to a vanishing point. That is, in affine transformation, parallel projections are used to project points onto the image plane along parallel lines;however, in perspective transformation, the perspective projection projects points onto the image plane along lines that starts from a single point, called the center of projection. It implies that an object has a smaller project when it is far away from the center of projection and a large projection when it is closer.

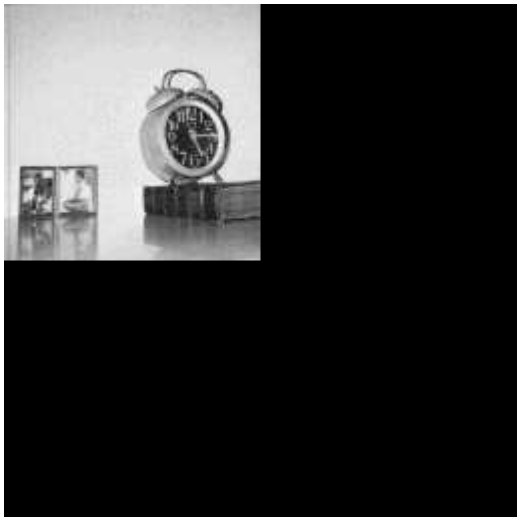
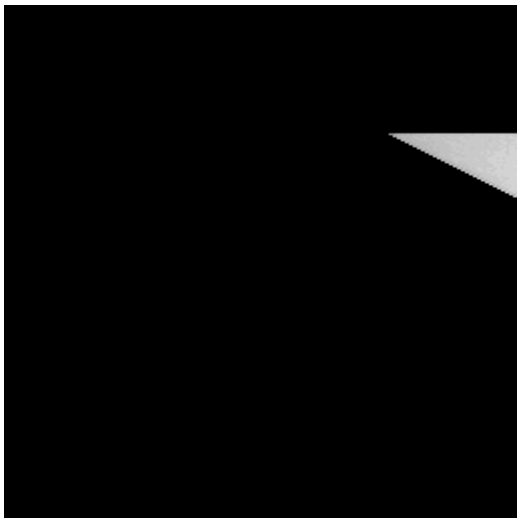
Task3.1b) an explanation of difference between inverse and forward transforms

In forward mapping, we determine the pixel values of output image by mapping all pixels in the input image to the output image. Forward mapping suffers from two problems: gaps and overlaps. Since we are working in a discrete domain, it is possible that some output pixels do not receive any input image pixels. These are called gaps. Moreover, some output pixels might receive more than one input image pixel. There are called overlaps.

Inverse mapping avoids problems with gaps and overlaps by starting from the output image. In inverse mapping, for each pixel in the output image, we apply inverse spatial transformation to determine the corresponding location in input image, and we apply approximation or interpolation technique to get an approximate value for the pixel.

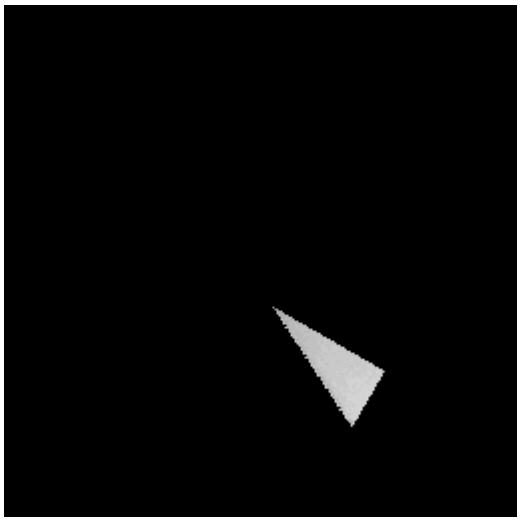
Task3.2a) Show the composite matrix. Show results generated from Task 3.2 using composite and intermediate results from sequentials. Compare the results and comment on performance.

1) Sequentially applying matrix

Matrix	Result
1) Shrink by 2 0.5 0 0 0 0.5 0 0 0 1	
2) Translation to the center of the image 1 0 64 0 1 64 0 0 1	
3) Shear, in the horizontal direction by 2 1 0 0 2 1 0 0 0 1	

4) rotation, clockwise by 30 degrees

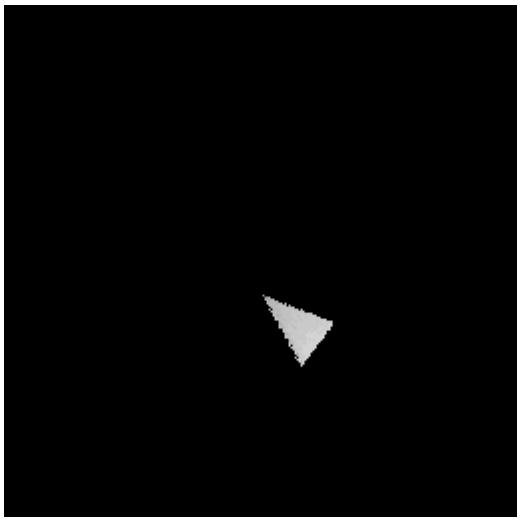
```
0.866075 0.499914    0
-0.499914 0.866075    0
   0    0    1
```



5) perspective transformation

```
0.666232          0        0
   0           0.666232    0
-0.00130889 -0.00130889   1
```

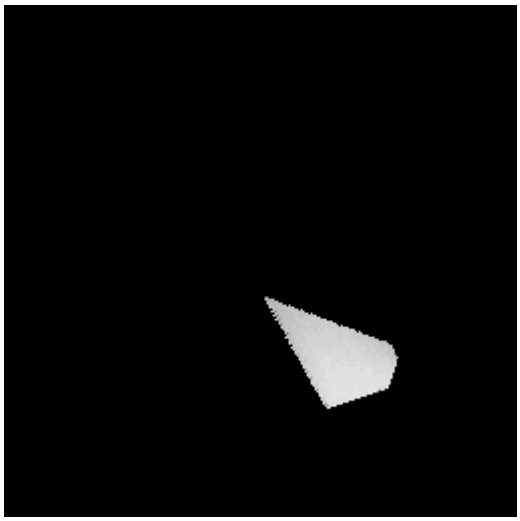
not correct -2



1) results with composite matrix

Composite matrix for 1)shrink, 2)translation, 3)shear, 4)rotation

```
0.932644 0.249741 151.345
0.616584 0.433162 134.368
   0    0    1
```

After perspective transformation	

Bonus1) Implement the adaptive local noise reduction filter and compare it with the adaptive median filter. Use Lena as the testing image. Add SAP noise with probability 0.1 and 0.25. Compare the result using PSNR.

Adaptive local noise reduction is implemented in adaptive.cpp. It can be tested using testadaptive program. Here are the results from both adaptive median filter and adaptive local noise reduction filter

(original image with salt and pepper noise with probability with 0.1)

(Adaptive median filter with SMax=7)
MSE: 48.2165
PSNR: 15.6494

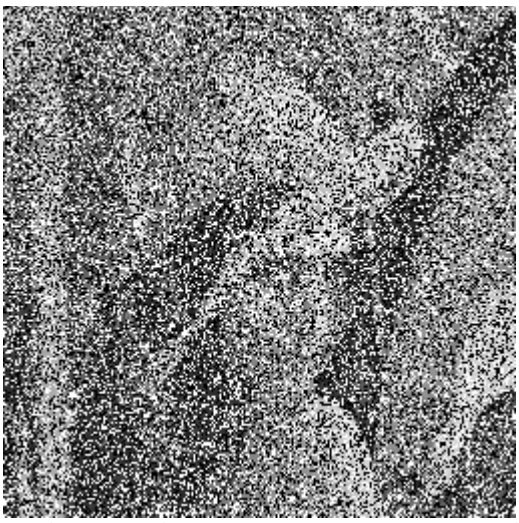
(Adaptive median filter with Smax=11)
MSE: 48.2064
PSNR: 15.6499

(Adaptive local noise reduction filter with variance=2550, kernel size=5)
MSE: 1222.62
PSNR: 8.62894

(Adaptive local noise reduction filter with variance=3300, kernel size=5)
MSE: 1141.92
PSNR: 8.77722

As you can see above, Adaptive median filter performs much better than adaptive local noise reduction filter. This is because that the noise is the impulse noise. Adaptive median filter is known to perform particularly well with the impulse noise whereas adaptive local noise reduction filter is known to work well with non-impulse noise such as Gaussian noise.

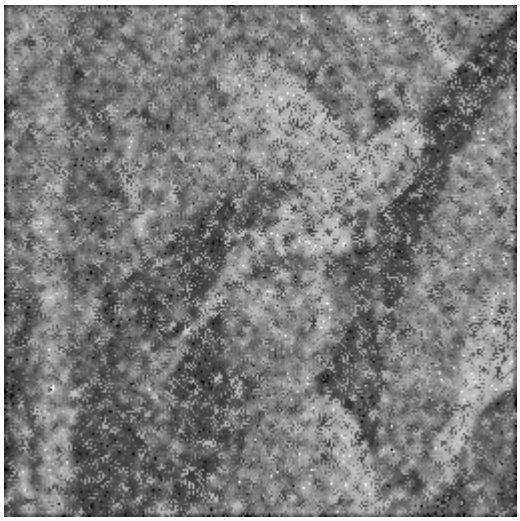
Adaptive local noise reduction filter and Adaptive median filter were also applied to the testing image with salt and pepper with probability with 0.25.



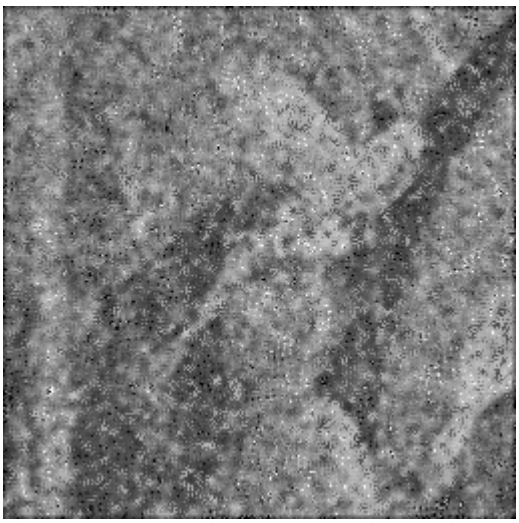
(original image with salt and pepper noise with probability with 0.25)

(Adaptive median filter with SMax=7)
MSE: 140.65
PSNR: 13.3247

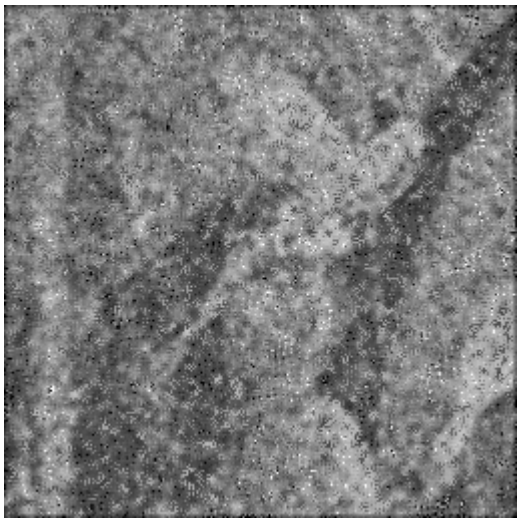
(Adaptive median filter with Smax=11)
MSE: 134.461
PSNR: 13.4224



(Adaptive local noise reduction filter with variance=7000, kernel size=5)
MSE: 1649.6
PSNR: 7.9785



(Adaptive local noise reduction filter with variance=8000, kernel size=5)
MSE: 1546.15
PSNR: 8.11914



(Adaptive local noise reduction filter with variance=8500, kernel size=5)
MSE: 1576.3
PSNR: 8.0772

+10

Bonus2) Implement the geometric transformation using polynomial approximation.

Geometric transformation is implemented in geotran.cpp. It can be tested using testgeotran program. It takes a list of points as a text file. A sample file is defined as below.

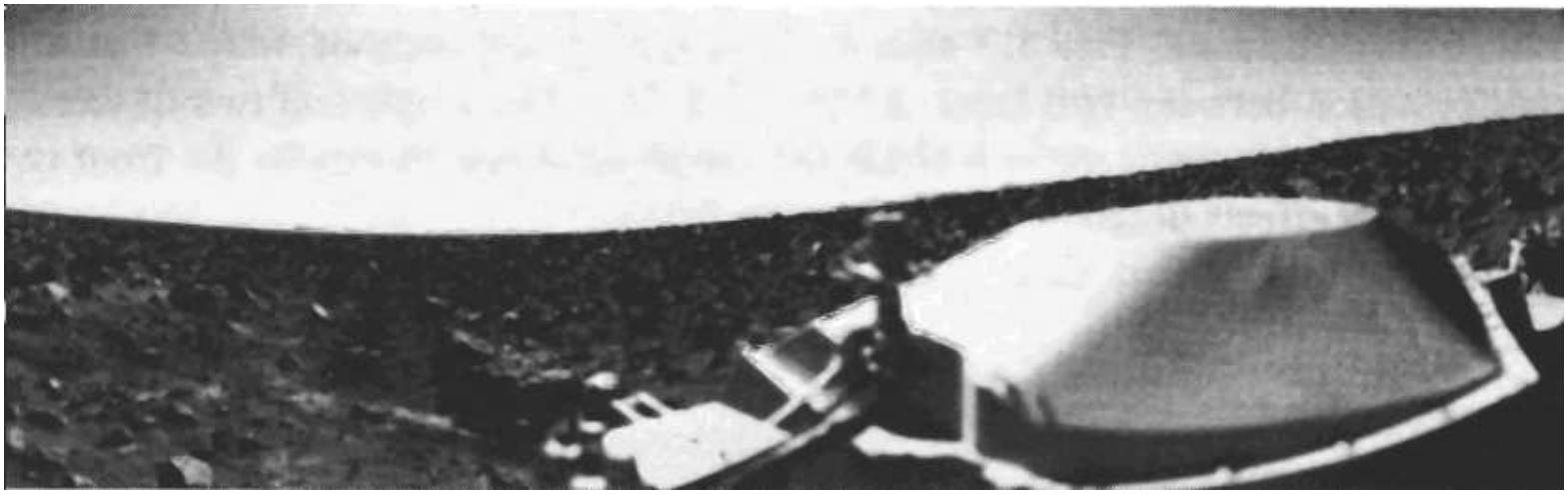
```
4
0 0
255 0
0 255
255 255
```

<sample input file for geotran.cpp>

The first line represents the number of points. Next lines represent (x,y) co-ordinates of the points. The following 2nd degree polynomial approximation is used to model inverse of the distortion.

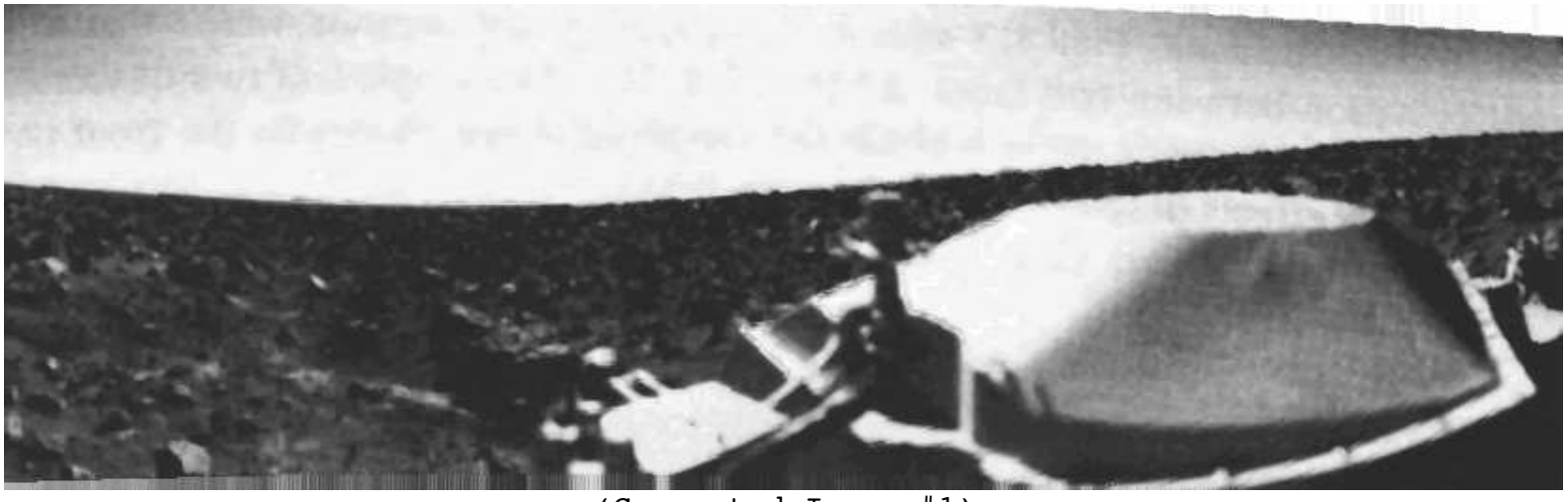
$$x_{est} = a000 + a101v + a110u + a202v^2 + a220u^2 + a211uv$$
$$y_{est} = b000 + b101v + b110u + a202v^2 + a220u^2 + a211uv$$

Here is the original image.



(Original Image)

Here is our first attempt of geometric correction.

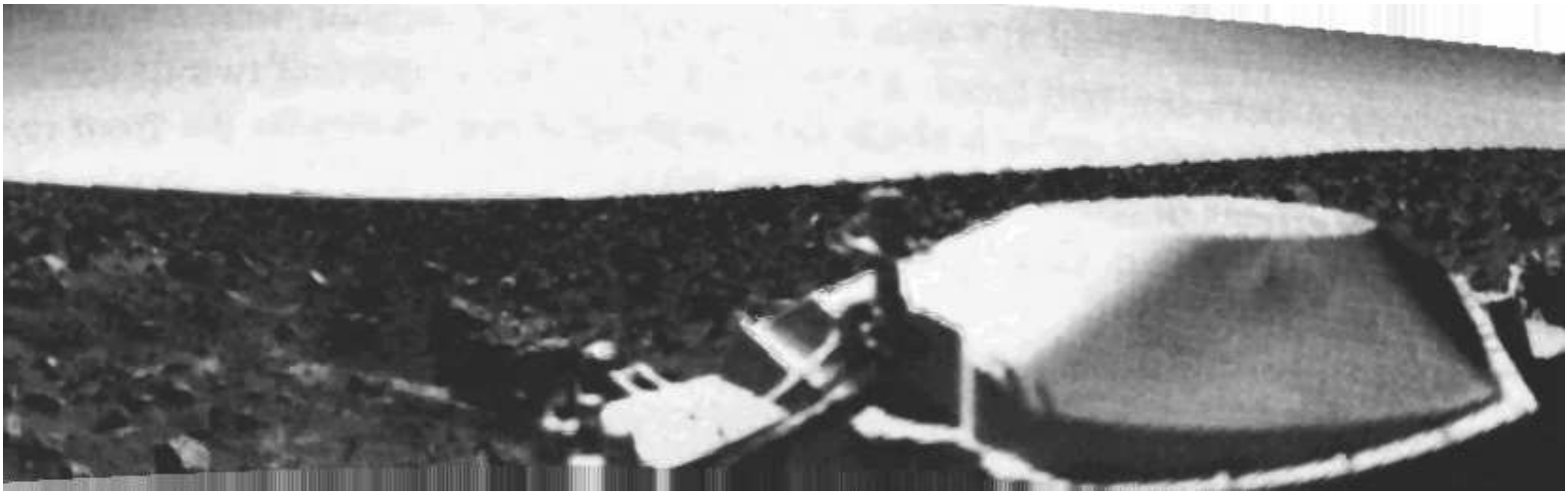


(Corrected Image #1)

The following control points were used.

Original points	Correct Points
0 0	0 0
0 100	0 100
0 300	0 300
0 500	0 500
0 700	0 700
0 827	0 827
108 0	91 0
117 100	91 100
122 200	91 200
120 300	91 300
107 400	91 400
95 500	91 500
91 600	91 600
74 700	91 700
59 826	91 826
257 0	257 0
257 100	257 100
257 300	257 300
257 500	257 500
257 700	257 700
257 826	257 826

Here is our second attempt. In this case, I introduced more control points



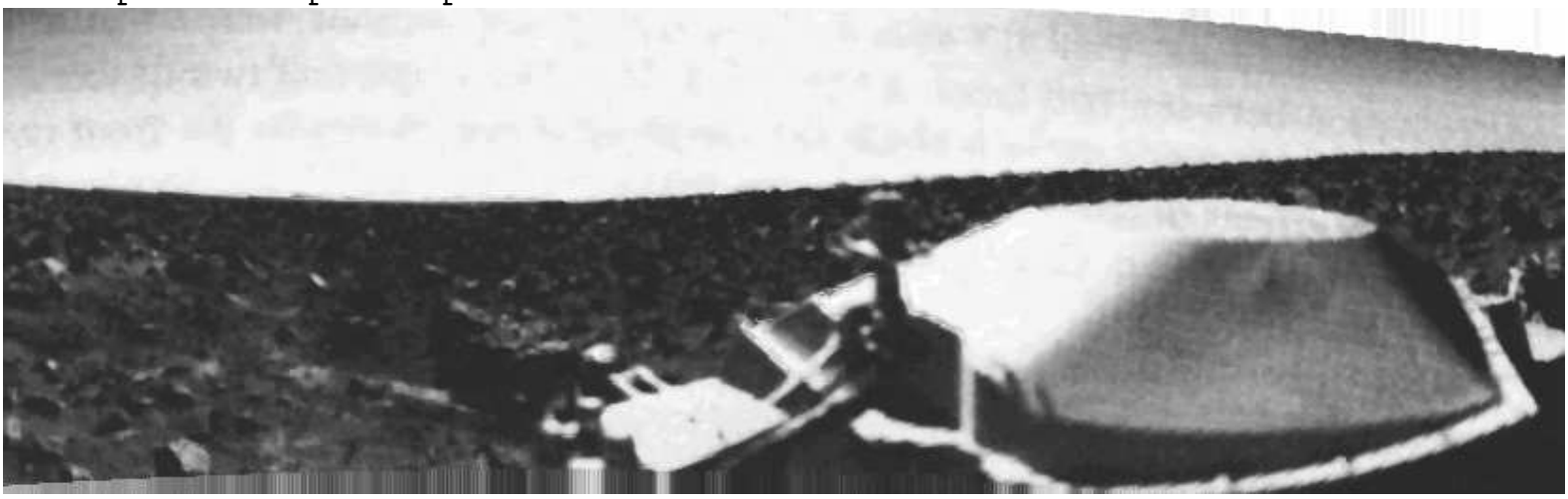
(Corrected Image #2)

The following control points were used.

Original points	Correct Points
0 0	0 0
0 100	0 100
0 300	0 300
0 500	0 500
0 700	0 700
0 827	0 827
108 0	91 0
113 50	91 50
117 100	91 100
120 150	91 150
122 200	91 200

122 250	91 250
120 300	91 300
117 350	91 350
113 400	91 400
107 450	91 450
100 500	91 500
95 550	91 550
89 600	91 600
82 650	91 650
74 700	91 700
67 750	91 750
60 800	91 800
58 826	91 826
257 0	257 0
257 100	257 100
257 300	257 300
257 500	257 500
257 700	257 700
257 826	257 826

Here is our third attempt. In this case, I introduced more control points around the top of the spaceship.



(Corrected Image #3)

The following control points were used.

Original points	Correct Points
0 0	0 0
0 100	0 100
0 300	0 300
0 500	0 500
0 700	0 700
0 827	0 827
108 0	91 0
113 50	91 50
117 100	91 100
120 150	91 150
122 200	91 200
122 250	91 250
120 300	91 300
117 350	91 350
113 400	91 400
107 450	91 450

100 500 95 550 89 600 82 650 74 700 67 750 60 800 58 826 257 0 257 100 257 300 257 500 257 700 257 826 115 544 103 640 126 624 117 720	91 500 91 550 91 600 91 650 91 700 91 750 91 800 91 826 257 0 257 100 257 300 257 500 257 700 257 826 115 544 115 640 126 624 126 720

+10

Conclusion)

In this project, I have learned a great deal about various geometric transformation, denoising and debluring techniques by actually implementing them. I really enjoyed doing this project; however, it would have been great if the function prototypes are given so we know what arguments our function should take in. Overall, I am confident that I have much deeper understanding of image restoration now.

```
/*************************************************************
 * adaptive.cpp: adaptive local noise filter
 *
 *  - adaptive: adaptvie local noise filter
 *
 * Author: Sanghyeb Lee (C) slee91@utk.edu
 *
 * Created: 10/10/11
 *
 *************************************************************/

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cmath>
#include <cstdlib>

using namespace std;

/**
 * Adaptive local noise reduction filter
 * @param inimg input image
 * @param
 */

//1) value of the noisy image
//2)variance of the noise corrupting f(x,y) to form g(x,y)
//3) local mean of the pixel
//4)local variance of the pixels in Sxy.

Image adaptive(Image& inimg, int kernelSize,float noise) {
  int nr,nc,ntype,nchan;
  Image outimg;
  int i,j,k;
  int radius=kernelSize/2;
  double local_mean;
  double local_variance;
  double local_sum;

  //allocate memory
   nr = inimg.getRow();
  nc = inimg.getCol();
  ntype = inimg.getType();
  nchan = inimg.getChannel();


  //create an output image
  outimg.createImage(nr,nc,nchan);

  //for each position (i,j,k) in the image
  for (i=0; i<nr; i++) {
    for (j=0; j<nc; j++) {
      for (k=0; k<nchan; k++) {

        //work out the local sum
        local_sum=0;
        for(int xi=-radius;xi<(-radius+kernelSize);xi++) {
          for(int xj=-radius;xj<(-radius+kernelSize);xj++) {
            if( (xi+i)>=0 && (xi+i)<nr && (xj+j)>=0 && (xj+j)<nc) {
              //work out the locl sum
              local_sum = local_sum + inimg(xi+i,xj+j,k);
            }
          }
        }
        //work out the local mean
        local_mean = local_sum / (kernelSize*kernelSize);

        //work out the local variance
        local_variance=0;
        for(int xi=-radius;xi<(-radius+kernelSize);xi++) {
          for(int xj=-radius;xj<(-radius+kernelSize);xj++) {
            if( (xi+i)>=0 && (xi+i)<nr && (xj+j)>=0 && (xj+j)<nc) {
              //work out the locl sum
              local_variance = local_variance + (pow((inimg(xi+i,xj+j,k) -local_mea
n),2));
            }
          }
        }
        local_variance = local_variance / (kernelSize*kernelSize);
        //cout<<"Local variance: "<<local_variance<<endl;
        //cout<<"Local_mean: "<<local_mean<<endl;

        //Assign pixel value!
        outimg(i,j,k) = inimg(i,j,k) - ((noise/local_variance) * (inimg(i,j,k) - lo
cal_mean));
        //outimg(i,j,k) = local_mean;
        //      cout<<"Assigning "<<outimg(i,j,k)<<" at "<<i<<"  , "<<j<<" ,"<<k<<e
ndl;
      }
    }
  }

  return outimg;

}
```

next time please add dip.h

```
/*************************************************************
 * deblur.cpp: deblurring filters.
 *
 *  - invFilter: inverse filter
 *  - wiener: wiener filter
 * Author: Sanghyeb Lee (C) slee91@utk.edu
 *
 * Created: 09/30/11
 *
 *************************************************************/
#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cmath>
#include <cstdlib>

using namespace std;


/** Wiener filtering
 *@param input inputImage;
 *@param D_0 D_0 of the Gaussian low-pass filter
 *@param K constant used as a replacement of power-spectrum constant
 *@return imag eafter wiener filtering
 */
Image wiener(Image input,float D_0, float K) {
   Image Huv;
  int P;
  int Q;
  int nr,nc,nchan;
  Image img2;
  int i,j,k;
  Image mag,phase;
  Image cropped;
  Image F;
  Image outimg;
  float Duv;
  Image H_two;
  Image denom;

  //pad the image
  nr = input.getRow();
  nc = input.getCol();

  //find the right padding size.
  //padding size should be greater than twice the width of an image
  //AND it should be power of 2
  P=2;
  while(true) {
    if(P>=(nr*2)&&P>=(nc*2)) break; //check if condition is met
    P = P *2; //otherwise, multiply it by 2
  }
  Q=P;


  //creates a padded image
  img2.createImage(P,Q);
  for(i=0;i<nr;i++)
    for(j=0;j<nc;j++)
      img2(i,j) = input(i,j);

  //creates an estimation of blur kernel
  Huv.createImage(P,Q); //creates an image
  for(int u=0;u<P;u++) {
    for(int v=0;v<Q;v++) {
      //use equation
      //H(u,v) = e^(-D(u,v)^2 / (2D0^2)
      Duv = sqrt(pow((u-P/2),2) + pow((v-Q/2),2));
      Huv(u,v) = exp(-pow(Duv,2)/(2 * pow(D_0,2)));
    }
  }

  //Fourier transform
  mag.createImage(P,Q);
  phase.createImage(P,Q);
  outimg.createImage(P,Q);

  fft(img2,mag,phase);

  //apply inverse filter
  H_two = Huv*Huv; //H_two is |H|^2
  denom = H_two*Huv;
  denom = denom +K;
  F = (H_two * mag)/denom;

  //F = F  * Huv;
  //inverse fourier transform
  ifft(outimg,F,phase);


  //crop the image
   cropped = subImage(outimg,0,0,nr-1,nc-1);

  return cropped;
}

 /**
 * Inverse filtering
 * @param input inputImage;
 * @param D_O D_0 of the Gaussian low-pass filter
 * @return image after inverse filtering
 */
Image invFilter(Image input,float D_0) {
  Image Huv;
  int P;
  int Q;
  int nr,nc,nchan;
  Image img2;
  int i,j,k;
  Image mag,phase;
  Image cropped;
  Image F;
  Image outimg;
  float Duv;
  Image Hinv;

  //pad the image
  nr = input.getRow();
  nc = input.getCol();

  //find the right padding size.
  //padding size should be greater than twice the width of an image
  //AND it should be power of 2
  P=2;
  while(true) {
    if(P>=(nr*2)&&P>=(nc*2)) break; //check if condition is met
    P = P *2; //otherwise, multiply it by 2
  }
  Q=P;


  //creates a padded image
  img2.createImage(P,Q);
  for(i=0;i<nr;i++)
```

```
    for(j=0;j<nc;j++)
      img2(i,j) = input(i,j);

  //creates an estimation of blur kernel
  Huv.createImage(P,Q); //creates an image
  Hinv.createImage(P,Q); //creates an image of invese
  for(int u=0;u<P;u++) {
    for(int v=0;v<Q;v++) {
      //use equation
      //H(u,v) = e^(-D(u,v)^2 / (2D0^2)
      Duv = sqrt(pow((u-P/2),2) + pow((v-Q/2),2));
      Huv(u,v) = exp(-pow(Duv,2)/(2.0 * pow(D_0,2)));

      //if smaller than 0.01, make it 1.
      //some elements are very small, so diviing produces very larget values.
       if(Huv(u,v)<0.01) Huv(u,v)=1;
      //if(Huv(u,v)>0.01) {
      //        Hinv(u,v) = 1 / Huv(u,v);
      // }else
      //        Hinv(u,v) = 0;
      //
    }
  }

  //Fourier transform
  mag.createImage(P,Q);
  phase.createImage(P,Q);
  outimg.createImage(P,Q);

  fft(img2,mag,phase);

  //apply inverse filter

  F = mag / Huv;

  //F = F  * Huv;
  //inverse fourier transform
  ifft(outimg,F,phase);


  //crop the image
   cropped = subImage(outimg,0,0,nr-1,nc-1);

  return cropped;
}
```

```
/*************************************************************
 * freqFilter.cpp: frequency filters
 *
 *   - notch: notch filter
 *
 * Author: Sanghyeb Lee (C) slee91@utk.edu
 *
 * Created: 09/30/11
 *
 *************************************************************/

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cmath>
#include <cstdlib>

using namespace std;

/**
 * notch filter. It uses a vertical rectangular line to remove noise
 * @param inimg input image
 * @param D_0 radius of the circle in the middle. Vertical line will not be drawn i
n this radius
 * @param width of the line
 * @return image after notch frequency filtering
 */
Image notch(Image inimg,float D_0,float width) {
  Image Huv;
  int P;
  int Q;
  int nr,nc,nchan;
  Image img2;
  int i,j,k;
  Image mag,phase;
  Image cropped;
  Image F;
  Image outimg;
  float Duv;
  Image H_two;
  Image denom;

  //pad the image
  nr = inimg.getRow();
  nc = inimg.getCol();

  //find the right padding size.
  //padding size should be greater than twice the width of an image
  //AND it should be power of 2
  P=2;
  while(true) {
    if(P>=(nr*2)&&P>=(nc*2)) break; //check if condition is met
    P = P *2; //otherwise, multiply it by 2
  }
  Q=P;

  //creates a padded image
  img2.createImage(P,Q);
  for(i=0;i<nr;i++)
    for(j=0;j<nc;j++)
      img2(i,j) = inimg(i,j);

  //creates an filter by drawing a vertical line
  Huv.createImage(P,Q); //creates an image
  Huv = Huv+1; //set it to 1;

  for(int v=(Q-width)/2;v<(Q+width)/2;v++) {
    for(int u=0;u<P;u++) {
      //do not draw the vertical line within the radius
      if(abs((P/2)-u)<D_0) Huv(u,v) = 1;
      else {

        Huv(u,v) = 0;
      }

    }
  }

    Image res = rescale(Huv);
    writeImage(res,"noisefrequency.pgm");


  //Fourier transform
  mag.createImage(P,Q);
  phase.createImage(P,Q);
  outimg.createImage(P,Q);

  fft(img2,mag,phase);

  //apply inverse filte
  F = (mag * Huv);

  //F = F  * Huv;
  //inverse fourier transform
  ifft(outimg,F,phase);


  //crop the image
   cropped = subImage(outimg,0,0,nr-1,nc-1);

  return cropped;
}
```

```
/*********************************************************
 * geotran.cpp - geometric transformation
 *
 *   - geotranpoly2: geometric transformation using 2nd polynomial approximation
 *
 * Author: Sanghyeb(Sam) Lee (C) slee91@utk.edu
 *
 * Created: 10/10/11
 *********************************************************/
#include "Image.h"
#include "Dip.h"
#include "utility.h"
#include <iostream>
#include <cstdlib>

using namespace std;

void printMatrix(Image &img);

/**
 * Geometric transformation using 2nd degree polynomial approximation
 * 2nd degre polynomial equation
 *  x_est = a000 +a101v + a110 u + a202v^2 + a220 u^2 + a211 uv
 *  y_est = b000 +b101v + b110 u + a202v^2 + a220 u^2 + a211 uv
 * @param inimg The input matrix.
 * @param original an array of tie points in the original image
 * @param corrected an array of tie points in the corrected image
 * @param size size of the array
 * @return Image with geometric transformation
 */
Image geotranpoly2(Image& inimg, Point original[],Point corrected[],int size) {
  int i,j,k;
  int nc,nr,nchan;
  Image outimg;
  float x_est;
  float y_est;
  int x,y;

  //(1) use 2nd degree polynomial approximation to model
  // inverse of the distortion
  // 2nd degree polynomial equation
  // x_est = a000 +a101v + a110 u + a202v^2 + a220 u^2 + a211 uv
  // y_est = b000 +b101v + b110 u + a202v^2 + a220 u^2 + a211 uv
  // W = [1 v u v^2 u^2 uv]

  //(1) set up matrix W,X,Y
  Image W(size,6);
  Image X(size,1);
  Image Y(size,1);
  Image W_T;
  Image W_Inverse;
  Image A;
  Image B;
  Image temp;

  for(i=0;i<size;i++) {

    //set up (1-a) W
    W(i,0) = 1; //a000
    W(i,1) = corrected[i].y;   //a101v
    W(i,2) = corrected[i].x;   //a110u
    W(i,3) = pow(corrected[i].y,2); //a202v^2
    W(i,4) = pow(corrected[i].x,2); //a220u^2
    W(i,5) = corrected[i].x * corrected[i].y; //a211 uv

    //set up (1-b) X
    X(i,0) = original[i].x;

    //set up (1-c) Y
    Y(i,0) = original[i].y;

  }
  cout<<"matrix W"<<endl;
  printMatrix(W);

  //(2) Solve the coefficients of the polynomial
  // A = W-1X
  // B = W-1Y
  // W-1 = (WtW)-1Wt
  W_T = transpose(W);
  cout<<"matrix W_T"<<endl;
  printMatrix(W_T);
  temp = W_T ->* W;
  cout<<"matrix W_T * W"<<endl;
  printMatrix(temp);

  W_Inverse = inverse(temp) ->* W_T;
  A = W_Inverse ->* X;
  B = W_Inverse ->* Y;
  //check calculation
  for(int i=0;i<size;i++) {
  x_est = A(0,0)*W(i,0) + A(1,0)*W(i,1) + A(2,0)*W(i,2) + A(3,0)*W(i,3) + A(4,0) *
W(i,4)+ A(5,0) * W(i,5);
  y_est = B(0,0)*W(i,0) + B(1,0)*W(i,1) + B(2,0)*W(i,2) + B(3,0)*W(i,3) + B(4,0) *
W(i,4)+ B(5,0) * W(i,5);
  cout<<"EST: "<<y_est<<", ACTUAL: "<<Y(i,0)<<endl;
  cout<<"EST: "<<x_est<<", ACTUAL: "<<X(i,0)<<endl;
  }


  //(3) For each (u,v) in the corrected image,
  //(4) Find the corresponding (x,y) in the original image and use its intensity as
 the intensity at (u,v)
  nr = inimg.getRow();
  nc = inimg.getCol();
  nchan = inimg.getChannel();
  outimg.createImage(nr,nc,nchan);

  for(i=0;i<nr;i++) {
    for(j=0;j<nc;j++) {
      //work out x_est, y_est using polynomail equation
      // x_est = a000 +a101v + a110 u + a202v^2 + a220 u^2 + a211 uv
      // y_est = b000 +b101v + b110 u + a202v^2 + a220 u^2 + a211 uv
      x_est = A(0,0) + A(1,0)*j + A(2,0)*i + A(3,0)*pow(j,2) + A(4,0) * pow(i,2) +
A(5,0) * i * j;
      y_est = B(0,0) + B(1,0)*j + B(2,0)*i + B(3,0)*pow(j,2) + B(4,0) * pow(i,2) +
B(5,0) * i * j;

      x = boundaryCheckF(round(x_est),0,nr-1);
      y = boundaryCheckF(round(y_est),0,nc-1);
      //Assign its intensitiy
      for(k=0;k<nchan;k++) {
        cout<<"Assigning ("<<i<<","<<j<<") = ("<<x<<","<<y<<")"<<endl;
        outimg(i,j,k) = inimg(x,y,k);
      }

    }
  }

  return outimg;

}

void printMatrix(Image& inimg) {
  int nr;
```

```
  int nc;

  nr = inimg.getRow();
  nc = inimg.getCol();
  cout<<nr<<" X "<<nc<<endl;
  for(int i=0;i<nr;i++) {
    cout<<"[ ";
    for(int j=0;j<nc;j++) {
      cout<<inimg(i,j)<<" ";
    }
    cout<<"]"<<endl;
  }
}
```

```
#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cmath>
#include <cstdlib>


/**
 * Add salt-and-pepper noise to an image
 * @param inimg The input image.
 * @param q The probability. 0<q<1.
 * For each pixel in the image, generate a random number, say r.
 *   If r<q, change the pixel's intensity to zero.
 *   If r>1-q, change the pixel's intensity to L
 * The higher the q, the worse the noise
 * @return Image corrupted by salt and pepper noise.
 */
Image sapNoise(Image &inimg, float q) {
  Image outimg;
  int i,j,k;
  int nr,nc,nchan;
  double r;

  //work out image statistics
  nr = inimg.getRow();
  nc = inimg.getCol();
  nchan = inimg.getChannel();

  outimg.createImage(nr,nc);

  // add SAP noise
  srand(time(0));         // so that a different seed nr is generated
  for (i=0; i<nr; i++)
    for (j=0; j<nc; j++)
      for (k=0; k<nchan; k++) {
        r = rand()/(RAND_MAX+1.0);
        outimg(i,j,k) = inimg(i,j,k);
        if (r < q)
          outimg(i,j,k) = 0;
        if (r > 1-q)
          outimg(i,j,k) = L;
      }

  return outimg;
}
```

```
/*************************************************************
 * spatialFilter.cpp: spatial Filters
 *
 *  - amean: arithmetic average spatial filter
 *  - amedian: adaptive median filter
 *  - gmean: geometric average spatial filter
 *  - sobel: sobel spatial filter
 *  - contrah: contra-harmonic filter
 * Author: Sanghyeb Lee (C) slee91@utk.edu
 *
 * Created: 09/30/11
 *
 *************************************************************/

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cmath>
#include <cstdlib>

using namespace std;

/**
 * Contraharmonic filter
 * @param inimg input image
 * @param kernelSize kernel size
 * @param Q order of the filter
 * @return image with contraharmonic filtering effect
 */
Image contrah(Image& inimg,int kernelSize,float Q) {
  int nr, nc, ntype, nchan;
  Image outimg;
  int row, column, i, j, k;
  float one_MN;
  int radius;
  double sum;
  double denominator;
  double nominator;

  // allocate memory
  nr = inimg.getRow();
  nc = inimg.getCol();
  ntype = inimg.getType();
  nchan = inimg.getChannel();


  //create an output image
  outimg.createImage(nr,nc,nchan);

  radius = kernelSize/2; //radius of the kernel
  //for each pixel at the image
  for (row=0; row<nr; row++) {
    for (column=0; column<nc; column++) {
      for (k=0; k<nchan; k++) {
        // work out the contraharmonic mean in the subimage window
        sum=0;
        denominator=0;
        nominator=0;
        for(i=-radius;i<=radius;i++) {
          for(j=-radius;j<=radius;j++) {
            //be careful not to go out of the range
            if( (i+row)>=0 && (i+row)<nr && (j+column)>=0 && (j+column)<nc) {
              denominator = denominator + pow(inimg(i+row,i+column,k),Q);
              nominator   = nominator + pow(inimg(i+row,i+column,k),Q+1);
            }

          }
        }

        //assign the resulting sum
        sum = nominator / denominator;

        outimg(row,column,k) = (float)sum;
      }
    }
  }

  return outimg;


}
/**
 * Sobel edge detector
 * @param inimg input image
 * @return resulting image with sobel edge detection filter
 */
Image sobel(Image& inimg) {
  Image hkernel(3,3); //kernel for detecting horizontal lines
  Image vkernel(3,3); //kernel for detecting vertical lines
  Image hresult;
  Image vresult;
  Image outimg;
  int nc,nr,nchan;
  int i,j,k;

  //setup horizontal kernel
  hkernel(0,0) = -1;
  hkernel(1,0) = 0;
  hkernel(2,0) = 1;
  hkernel(0,1) = -2;
  hkernel(1,1) = 0;
  hkernel(2,1) = 2;
  hkernel(0,2) = -1;
  hkernel(1,2) = 0;
  hkernel(2,2) = 1;

  //setup vertical kernel
  vkernel(0,0) = -1;
  vkernel(1,0) = -2;
  vkernel(2,0) = -1;
  vkernel(0,1) = 0;
  vkernel(1,1) = 0;
  vkernel(2,1) = 0;
  vkernel(0,2) = 1;
  vkernel(1,2) = 2;
  vkernel(2,2) = 1;

  //apply horizontal kernel
  hresult = conv(inimg,hkernel);

  //apply vertical kernel
  vresult = conv(inimg,vkernel);

  nr = inimg.getRow();
  nc = inimg.getCol();
  nchan = inimg.getChannel();

  outimg.createImage(nr,nc,nchan);

  //work out the output image by summing up the absolute values
  for(i=0;i<nr;i++) {
    for(j=0;j<nc;j++) {
      for(k=0;k<nchan;k++) {
        outimg(i,j,k) = fabs(hresult(i,j,k)) + fabs(vresult(i,j,k));
```

```
      }
    }
  }

  return outimg;
}



int sort(const void *x, const void *y) {
  return (*(float*)x - *(float*)y);
}

/**
 * Adaptive Median filter
 * @param inimg Input image
 * @param (optional) startKernelSize starting kernel size, default is 3
 * @param (optional) maxKernelSize maximum kernel size, default is 21
 * @return image with median filter effect
 */
Image amedian(Image& inimg,int startKernelSize,int maxKernelSize) {

  int nr, nc, ntype, nchan;
  Image outimg;
  int i, j, k;
  int kernelSize = startKernelSize; //starting kernelSize
  int arraySize = maxKernelSize*maxKernelSize;
  float array[arraySize]; //array for holding pixel values.
  int arrayCount=0;
  float tempHolder;
  float A1,A2,z_min,z_max,z_med; //variable for adaptive median filtering
  float B1,B2, z_fin; //variable for adaptive median filtering

  int radius = kernelSize /2;

  // allocate memory
  nr = inimg.getRow();
  nc = inimg.getCol();
  ntype = inimg.getType();
  nchan = inimg.getChannel();

  outimg.createImage(nr, nc, ntype);

  //for each position (i,j,k) in the image
  for (i=0; i<nr; i++) {
    for (j=0; j<nc; j++) {
      for (k=0; k<nchan; k++) {
        //default kernelSize
        kernelSize=startKernelSize;
        radius = kernelSize/2;

      REPEAT_STAGEA:
        //STAGE(A)
        //gather all the pixel values in the window
        arrayCount=0; //set the number of element array to 0
        for(int xi=-radius;xi<(-radius+kernelSize);xi++) {
          for(int xj=-radius;xj<(-radius+kernelSize);xj++) {
            if( (xi+i)>=0 && (xi+i)<nr && (xj+j)>=0 && (xj+j)<nc) {
              //put the pixel intensity into the array
              array[arrayCount]=inimg(xi+i,xj+j,k);
              arrayCount++;
            }
          }
        }

        //sort the array of pixel values
        qsort(array,arrayCount,sizeof(float),sort);
        z_med = array[arrayCount/2]; //median value
        z_max = array[arrayCount-1]; //maximum value
        z_min = array[0]; //minimum value

        //print out the value of array
        //for(int zz=0;zz<arrayCount;zz++)
        //        cout<<array[zz]<<" ";
        //      cout<<"End of array"<<endl;

        //work out A1 and A2
        A1 = z_med - z_min;
        A2 = z_med - z_max;

        //
        if(A1>0 && A2<0) {
          //STAGE(B)
          B1 = inimg(i,j,k) - z_min; // B1 = z_xy - z_min
          B2 = inimg(i,j,k) - z_max; // B2 = z_xy - z_max
          if(B1>0 && B2<0) z_fin = inimg(i,j,k); //output z_xy
          else z_fin = z_med; //output z_med
        } else {

          //increase the window size

          kernelSize=kernelSize+2;
          radius = kernelSize/2;

          if(kernelSize<=maxKernelSize) //repeat stage A
            goto REPEAT_STAGEA;
          else z_fin = z_med; //else output z_med
        }


        outimg(i,j,k) = z_fin;
      }
    }
  }

  return outimg;


}
/**
 * Geometric average mean filter
 * Each restored pixel is given by the product of the pixels in the subimage window
, raised to the power 1/mn.
 * @param inimg Input image
 * @param kernelSize size
 * @return image with geometric average spatil filter effect
 */
Image gmean(Image& inimg, int kernelSize) {
  int nr, nc, ntype, nchan;
  Image outimg;
  int row, column, i, j, k;
  float one_MN;
  int radius;
  double sum;

  // allocate memory
  nr = inimg.getRow();
  nc = inimg.getCol();
  ntype = inimg.getType();
  nchan = inimg.getChannel();


  //create an output image
  outimg.createImage(nr,nc,nchan);
```

```
  radius = kernelSize/2; //radius of the kernel
  one_MN = 1.0 / (kernelSize*kernelSize); // 1/MN

  //for each pixel at the image
  for (row=0; row<nr; row++) {
    for (column=0; column<nc; column++) {
      for (k=0; k<nchan; k++) {
        // work out the product of the pixels in the subimage window
        sum=1;
        for(i=-radius;i<=radius;i++) {
          for(j=-radius;j<=radius;j++) {
            //be careful not to go out of the range
            if( (i+row)>=0 && (i+row)<nr && (j+column)>=0 && (j+column)<nc) {
              sum = sum * inimg(i+row,j+column,k);
            }

          }
        }

        //raise it to the power 1/mn
        //cout<<"Assiging (1)"<<sum<<" at ("<<row<<","<<column<<")"<<endl;
        sum = pow(sum,one_MN);
        //assign the resulting sum
        //cout<<"Assiging (2)"<<sum<<" at ("<<row<<","<<column<<")"<<endl;

        outimg(row,column,k) = (float)sum;
      }
    }
  }

  return outimg;

}
/**
 * Arithmetic average mean filter
 * @param inimg Input image
 * @param kernelSize size
 * @return image with arithmetic average spatil filter effect
 */
Image amean(Image& inimg, int kernelSize) {

  Image filter(kernelSize,kernelSize);
  int nr, nc, ntype, nchan;
  Image outimg;
  int i, j, k;
  float element;

  // allocate memory
  nr = inimg.getRow();
  nc = inimg.getCol();
  ntype = inimg.getType();
  nchan = inimg.getChannel();

  //cout<<"BEFORE"<<endl;
  //(1) work out the filter
  element = 1.0/(kernelSize*kernelSize);
  for(i=0;i<kernelSize;i++)
    for(j=0;j<kernelSize;j++)
      filter(i,j) = element; //set the value

  //(2) apply filter using conv operator.
  outimg = conv(inimg,filter);
  //cout<<"AFTER"<<endl;
  return outimg;
}
```

```
// test code for adaptive median filter

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testadaptive inimg outimg kernelSize noise\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  int kernelSize;
  float noise;

  // check if the number of arguments on the command line is correct
  if (argc < 3) {
    cout << Usage;
    exit(3);
  }

  //read in kernel sizes
  kernelSize = atoi(argv[3]);
  noise   = atof(argv[4]);

  // read in image
  inimg = readImage(argv[1]);

  // test the contrast stretching function
  outimg = adaptive(inimg, kernelSize, noise);

  // output the image
  writeImage(outimg, argv[2]);

  return 0;
}
```

```
// test code for adding sap noise

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testamean inimg outimg probability\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  double probability;

  // check if the number of arguments on the command line is correct
  if (argc < 4) {
    cout << Usage;
    exit(3);
  }

  // read in command-line arguments
  probability = atof(argv[3]);

  // read in image
  inimg = readImage(argv[1]);

  // test sap noise function
  outimg = sapNoise(inimg,probability);

  // output the image
  writeImage(outimg, argv[2]);

  return 0;
}
```

```
// test code for arithmetic mean

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testamean inimg outimg kernelSize\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  int kernelSize;

  // check if the number of arguments on the command line is correct
  if (argc < 4) {
    cout << Usage;
    exit(3);
  }

  // read in command-line arguments
  kernelSize = atoi(argv[3]);

  // read in image
  inimg = readImage(argv[1]);

  // test the contrast stretching function
  outimg = amean(inimg, kernelSize);

  // output the image
  writeImage(outimg, argv[2]);

  return 0;
}
```

```
// test code for adaptive median filter

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testamedian inimg outimg startSize endSize\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  int startKernel;
  int endKernel;

  // check if the number of arguments on the command line is correct
  if (argc < 5) {
    cout << Usage;
    exit(3);
  }

  //read in kernel sizes
  startKernel = atoi(argv[3]);
  endKernel   = atoi(argv[4]);

  // read in image
  inimg = readImage(argv[1]);

  // test the contrast stretching function
  outimg = amedian(inimg,startKernel,endKernel);

  // output the image
  writeImage(outimg, argv[2]);

  return 0;
}
```

```
// test code for contra harmonic mean

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testcontrah inimg outimg kernelSize Q\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  int kernelSize;
  float Q;

  // check if the number of arguments on the command line is correct
  if (argc < 5) {
    cout << Usage;
    exit(3);
  }

  // read in command-line arguments
  kernelSize = atoi(argv[3]);
  Q = atof(argv[4]);

  // read in image
  inimg = readImage(argv[1]);

  // test the contrast stretching function
  outimg = contrah(inimg, kernelSize,Q);

  // output the image
  writeImage(outimg, argv[2]);

  return 0;
}
```

```
//Test code for task1.4 flordia satellite image.

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "./testflorida inimg outimg\n"

int main(int argc, char **argv)
{
  Image inimg, mag, phase, outimg, img2, img3;
  Image maglog;

  int imgsize = 4;
  int i, j;
  int P,Q;
  int nr,nc;

  if (argc < 3) {
    cout << Usage;
    exit(3);
  }

  // create the image
  inimg = readImage(argv[1]);

  nr = inimg.getRow();
  nc = inimg.getCol();

  if(nr>nc) { //pad with zero.
    P = 2*nr;
    Q = 2*nr;
  }else {
    Q = 2*nc;
    P = 2*nc;
  }

  P = 2048;
  Q = 2048;
  // for padding
  img2.createImage(P, Q);

  for (i=0; i<nr; i++)
    for (j=0; j<nc; j++)
      img2(i,j) = inimg(i,j);

  // perform FFT and IFFT
  mag.createImage(P, Q);
  phase.createImage(P,Q);
  img3.createImage(P, Q);
  maglog.createImage(P,Q);


  fft(img2, mag, phase);
  cout << "Magnitude after padding:\n" << mag << endl;

 // magnitude after log transformation
  //  maglog = logtran(mag);

  for (i=0; i<P; i++)      // new: perform log transformation to compress the dynam
ic range
    for (j=0; j<Q; j++)    // new: you should be able to call the logtran() you des
igned in proj2
      maglog(i,j) = log(1+fabs(mag(i,j)));

  // inverse Fourier transform
  ifft(outimg, mag, phase);

  ifft(img3, mag, phase);
  cout << "IFFT image with padding:\n" << img3 << endl;

  // you need to apply subImage to cut the upper-left corner
  // to obtain the resulting image of the right size

 // output image
  cout << "Write magnitude image ..." << endl;
  writeImage(mag, "testmag.pgm", 1);
  cout << "Write magnitude image AFTER log transformation ..." << endl;
  writeImage(maglog, "testmaglog.pgm", 1);
  cout << "Write phase image ..." << endl;
  writeImage(phase, "testphase.pgm", 1);
  cout << "Write image after inverse Fourier transform ..." << endl;
  writeImage(outimg, "testifft.pgm");
  return 0;
}
```

```
// test code for contrast stretching

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
#include <fstream>

using namespace std;

#define Usage "testgeotran inimg outimg originalpoint_file correctedpoint_file\n"

/**
 * read in a text file and convert it into an array of Point
 * @param filename name of the text file
 * @param pointer to pointer to an array of point
 */
int parsePointArray(char* filename,Point **array) {
  int size;


  ifstream fin(filename);

  //how many points?
  fin>>size;

  //creates an array
  (*array) = new Point[size];
  for(int i=0;i<size;i++) {
    fin>>((*array)[i].x);
    fin>>((*array)[i].y);
  }

  return size;
}

/**
 * prints list of point array
 * @param pointer to an array of Point
 */
void printPointArray(Point **array,int size) {
  for(int i=0;i<size;i++) {
    cout<<(*array)[i].x<<","<<(*array)[i].y<<endl;
  }
}

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  float m, b;
  Point* original;
  Point* corrected;
  int size;

  // check if the number of arguments on the command line is correct
  if (argc < 5) {
    cout << Usage;
    exit(3);
  }

  // read in image
  inimg = readImage(argv[1]);

  //construct array of points
  size = parsePointArray(argv[3],&original);
  size = parsePointArray(argv[4],&corrected);
  cout<<"SIZE: "<<size<<endl;
  cout<<"Original array"<<endl;
  printPointArray(&original,size);
  cout<<endl;
  cout<<"Corrected array"<<endl;
  printPointArray(&corrected,size);

  outimg =  geotranpoly2(inimg,original,corrected,size);


  // output the image
  writeImage(outimg, argv[2]);

  return 0;
}
```

```
// test code for geometric mean

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testgmean inimg outimg kernelSize\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  int kernelSize;

  // check if the number of arguments on the command line is correct
  if (argc < 4) {
    cout << Usage;
    exit(3);
  }

  // read in command-line arguments
  kernelSize = atoi(argv[3]);

  // read in image
  inimg = readImage(argv[1]);

  // test the contrast stretching function
  outimg = gmean(inimg, kernelSize);

  // output the image
  writeImage(outimg, argv[2]);

  return 0;
}
```

```
// test code for printing out a histogram for an image

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testhist inimg outimg startx starty endx endy histfile\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  Image subimg;

  int startx,starty,endx,endy; //region to crop

  // check if the number of arguments on the command line is correct
  if (argc < 8) {
    cout << Usage;
    exit(3);
  }

  //read in the region values
  startx = atoi(argv[3]);
  starty = atoi(argv[4]);
  endx = atoi(argv[5]);
  endy = atoi(argv[6]);

  //crop the image
  inimg = readImage(argv[1]);
  subimg = subImage(inimg,startx,starty,endx,endy);
  writeImage(subimg,argv[2]);

  printHistogram(subimg,argv[7]);
}
```

```
// test code for inverse filter

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testinvf inimg outimg D_0\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  int D_0;

  // check if the number of arguments on the command line is correct
  if (argc < 4) {
    cout << Usage;
    exit(3);
  }

  // read in command-line arguments
  D_0 = atoi(argv[3]);

  // read in image
  inimg = readImage(argv[1]);

  // test the contrast stretching function
  outimg = invFilter(inimg, D_0);

  // output the image
  writeImage(outimg, argv[2]);

  return 0;
}
```

```
// test code for mse

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testse original restored\n"

int main(int argc, char **argv)
{
  Image imgA,imgB;    // the original image
  Image dif;
  double sum_2;
  int nr,nc;
  double MSE;
  double PSNR;

  // check if the number of arguments on the command line is correct
  if (argc < 3) {
    cout << Usage;
    exit(3);
  }


  // read in image
  imgA = readImage(argv[1]);
  imgB = readImage(argv[2]);
  nr = imgA.getRow();
  nc = imgA.getCol();

  //work out MSE
  // e^2 = E{(f-f_e)^2}
  sum_2=0;
  dif = imgA - imgB;
  for(int i=0;i<nr;i++) {
    for(int j=0;j<nc;j++) {
      sum_2 += dif(i,j) * dif(i,j);
    }
  }
  cout<<"SUM^2: "<<sum_2<<endl;
  MSE = sum_2/(nr*nc);

  //work out PSNR
  PSNR = 10 * log10( L / sqrt(MSE));

  cout<<"MSE: "<<MSE<<endl;
  cout<<"PSNR: "<<PSNR<<endl;

  return 0;
}
```

```
// test code for wiener filter

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testwiener inimg outimg D_0 width\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  float D_0;
  float K;
  float width;

  // check if the number of arguments on the command line is correct
  if (argc < 5) {
    cout << Usage;
    exit(3);
  }

  // read in command-line arguments
  D_0 = atof(argv[3]);
  width = atof(argv[4]);
  // read in image
  inimg = readImage(argv[1]);

  // test the contrast stretching function
  outimg = notch(inimg, D_0,width);

  Image noise =  inimg - outimg;

  Image res = rescale(noise);
  writeImage(res,"res_noise.pgm");
  writeImage(noise,"noise.pgm");


  // output the image
  writeImage(outimg, argv[2]);

  return 0;
}
```

```
// test code for contrast stretching

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
#include <fstream>

using namespace std;

#define Usage "testcs inimg outimg originalpoint_file correctedpoint_file\n"

/**
 * read in a text file and convert it into an array of Point
 * @param filename name of the text file
 * @param pointer to pointer to an array of point
 */
int parsePointArray(char* filename,Point **array) {
  int size;


  ifstream fin(filename);

  //how many points?
  fin>>size;

  //creates an array
  (*array) = new Point[size];
  for(int i=0;i<size;i++) {
    fin>>((*array)[i].x);
    fin>>((*array)[i].y);
  }

  return size;
}

/**
 * prints list of point array
 * @param pointer to an array of Point
 */
void printPointArray(Point **array,int size) {
  for(int i=0;i<size;i++) {
    cout<<(*array)[i].x<<","<<(*array)[i].y<<endl;
  }
  }

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  float m, b;
  Point* original;
  Point* corrected;
  int size;

  // check if the number of arguments on the command line is correct
  if (argc < 5) {
    cout << Usage;
    exit(3);
  }

  // read in image
  inimg = readImage(argv[1]);

  //construct array of points
  size = parsePointArray(argv[3],&original);
  size = parsePointArray(argv[4],&corrected);
  cout<<"SIZE: "<<size<<endl;
  cout<<"Original array"<<endl;
  printPointArray(&original,size);
  cout<<endl;
  cout<<"Corrected array"<<endl;
  printPointArray(&corrected,size);

  if(size!=4) {
    cout<<"SIZE sould be 4"<<endl;
    return -1;
  }
  Image matrix = perspective_coefficient(original,corrected);
  outimg =  perspective(inimg,matrix);

  // output the image
  writeImage(outimg, argv[2]);

  return 0;
}
```

```
// test code for contrast stretching

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testtransform inimg outimg whichtransform arg1 arg2\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  float arg1, arg2,arg3,arg4;
  float arg5, arg6,arg7,arg8;
  int whichtransform;
  Image matrix(3,3);

  // check if the number of arguments on the command line is correct
  if (argc < 5) {
    cout << Usage;
    exit(3);
  }

  // read in command-line arguments
  whichtransform = atoi(argv[3]);
  arg1 = atof(argv[4]);

  //need two arguments unless rotating
  if(whichtransform!=2) arg2 = atof(argv[5]);
  // read in image
  inimg = readImage(argv[1]);

  switch(whichtransform) {
  case 1: //tranlation
    cout<<"Translation by "<<arg1<<" , "<<arg2<<endl;
    outimg = translate(inimg,arg1,arg2);
    break;
  case 2: //rotation
    cout<<"Rotation by "<<arg1<<endl;
    outimg = rotate(inimg,arg1);
    break;

  case 3: //shear
    cout<<"Shear by "<<arg1<<" , "<<arg2<<endl;
    outimg = shear(inimg,arg1, arg2);
    break;

  case 4://scale
    cout<<"Scale by "<<arg1<<" ,"<<arg2<<endl;
    outimg = scale(inimg,arg1, arg2);
    break;

  }

  // output the image
  writeImage(outimg, argv[2]);

  return 0;
}
```

```
// test code for contrast stretching

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testcs inimg"

extern Image composite;

/** apply composite matrix
 @param inimg input image
 @param matrix composite matrix
 @return transformed image
*/
Image apply_composite(Image& inimg, Image& matrix) {
  int i,j,k;
  int nr,nc,nchan;
  Image outimg; //output image

  //construct matrix

  Image matrix_inverse; //inverse transformation matrix
  Image corrected(3,1);
  Image original;
  int x,y;


  cout<<matrix<<endl;

  matrix_inverse = inverse(matrix);

  //perform inverse transformation
  nr = inimg.getRow();
  nc = inimg.getCol();
  nchan = inimg.getChannel();
  outimg.createImage(nr,nc,nchan);


  //for each pixel in the new image
  for(i=0;i<nr;i++) {
    for(j=0;j<nc;j++) {

        corrected(0,0) = i;
        corrected(1,0) = j;
        corrected(2,0) = 1;

        //work out the position in the original image
        original = matrix_inverse ->* corrected;
        for(k=0;k<nchan;k++)  {
          x = round(original(0,0));
          y = round(original(0,1));
          if(x<0||x>=nr||y<0||y>=nc) outimg(i,j,k)=0;
             else outimg(i,j,k) = inimg(x,y,k);
        }
    }
  }

  return outimg;
}

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  float arg1, arg2,arg3,arg4;
  float arg5, arg6,arg7,arg8;
  int whichtransform;


  // check if the number of arguments on the command line is correct
  if (argc < 2) {
    cout << Usage;
    exit(3);
  }

  composite(0,0)=1;
  composite(1,1)=1;
  composite(2,2)=1;

  //applying sequentually
  inimg = readImage(argv[1]);

  // 1)Shrink
  outimg = scale(inimg,0.5,0.5);
  writeImage(outimg, "shrinnked.pgm");

  // 2)Translation to the center
  outimg = translate(outimg,64,64);
  writeImage(outimg,"Translate.pgm");

  // 3)Shear in the horizontal direction by 2
   outimg = shear(outimg,0,2);
  writeImage(outimg,"shear.pgm");

  // 4)Rotate clockwise by 30
  outimg = rotate(outimg,-0.523);
  writeImage(outimg,"rotate.pgm");

  //  5)Perspective transformation
  Image per(8,1);
  per(0,0) = 0.666232;
  per(4,0) = 0.666232;
  per(6,0) = -0.00130889;
  per(7,0) = -0.00130889;
  per(8,0) = 1;

  outimg = perspective(outimg,per);
  writeImage(outimg,"perspective.pgm");



  //STAGE 2) Composite matrix
  cout<<"Applying composite matrix"<<endl;
  cout<<composite<<endl;

  outimg = apply_composite(inimg,composite);
  writeImage(outimg,"composite.pgm");
  outimg = perspective(outimg,per);
  writeImage(outimg,"composite_perspective.pgm");

  //  perspectivereturn 0;
}
```

```
// test code for wiener filter

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testwiener inimg outimg D_0 K\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  float D_0;
  float K;

  // check if the number of arguments on the command line is correct
  if (argc < 4) {
    cout << Usage;
    exit(3);
  }

  // read in command-line arguments
  D_0 = atof(argv[3]);
  K   = atof(argv[4]);

  // read in image
  inimg = readImage(argv[1]);

  // test the contrast stretching function
  outimg = wiener(inimg, D_0,K);

  // output the image
  writeImage(outimg, argv[2]);

  return 0;
}
```

```
/*********************************************************
 * transform.cpp - affine and perpsective transformation function
 *
 *   - rotate: rotate affine transformation
 *   - translation: translation affine transformation
 *   - shear: share affine transformation
 *   - scale: scale affine transformation
 *
 * Author: Sanghyeb(Sam) Lee (C) slee91@utk.edu
 *
 * Created: 10/10/11
 *********************************************************/

#include "Image.h"
#include "Dip.h"
#include "utility.h"
#include <iostream>
#include <cmath>
#include <cstdlib>

using namespace std;

Image composite(3,3);


/** Determine coefficient matrix for perspective transformation
 @param original an array of original points
 @param corrected an array of corrected points
 @return coefficient matrix
 */
Image perspective_coefficient(Point original[],Point corrected[]) {
  int i,j;

 //Determine the coeffient of the matrix
  Image new_points(8,1);
  for(i=0;i<4;i++) {
    new_points((i*2),0) = corrected[i].x;
    new_points((i*2)+1,0) = corrected[i].y;

   }


  //determine the matrix A

  Image A(8,8);
  A(0,0) = original[0].x;
  A(0,1) = original[0].y;
  A(0,2) = 1;
  A(0,6) = -original[0].x * corrected[0].x;
  A(0,7) = -corrected[0].x * original[0].y;

  A(1,3) = original[0].x;
  A(1,4) = original[0].y;
  A(1,5) = 1;
  A(1,6) = -original[0].x * corrected[0].y;
  A(1,7) = -original[0].y * corrected[0].y;
  //
  A(2,0) = original[1].x;
  A(2,1) = original[1].y;
  A(2,2) = 1;
  A(2,6) = -original[1].x * corrected[1].x;
  A(2,7) = -corrected[1].x * original[1].y;

  A(3,3) = original[1].x;
  A(3,4) = original[1].y;
  A(3,5) = 1;
  A(3,6) = -original[1].x * corrected[1].y;
  A(3,7) = -original[1].y * corrected[1].y;
  //
  A(4,0) = original[2].x;
  A(4,1) = original[2].y;
  A(4,2) = 1;
  A(4,6) = -original[2].x * corrected[2].x;
  A(4,7) = -corrected[2].x * original[2].y;

  A(5,3) = original[2].x;
  A(5,4) = original[2].y;
  A(5,5) = 1;
  A(5,6) = -original[2].x * corrected[2].y;
  A(5,7) = -original[2].y * corrected[2].y;
  //
  A(6,0) = original[3].x;
  A(6,1) = original[3].y;
  A(6,2) = 1;
  A(6,6) = -original[3].x * corrected[3].x;
  A(6,7) = -corrected[3].x * original[3].y;

  A(7,3) = original[3].x;
  A(7,4) = original[3].y;
  A(7,5) = 1;
  A(7,6) = -original[3].x * corrected[3].y;
  A(7,7) = -original[3].y * corrected[3].y;


  //determine matrix consisting of the coefficients
  Image coefficients(8,1);
  Image A_inverse = inverse(A);

  coefficients = A_inverse ->* new_points;
  cout<<"new points"<<endl;
  cout<<new_points<<endl;

  cout<<"Coefficients"<<endl;
  cout<<coefficients<<endl;

  return coefficients;
}
/** Perspective transformation
 @param inimg input image
 @param matrix coefficient matrix
 @return Image with perspective transformation
*/
Image perspective(Image& inimg, Image& coefficients) {
  int i,j,k;
  int nr,nc,nchan;
  Image outimg; //output image
  int x,y;

  nr = inimg.getRow();
  nc = inimg.getCol();
  nchan = inimg.getChannel();
  outimg.createImage(nr,nc,nchan);

  //for each pixel in the new image
  for(i=0;i<nr;i++) {
    for(j=0;j<nc;j++) {
        //work out the position in the original image
        for(k=0;k<nchan;k++)  {
          x = round((coefficients(0,0)*i+coefficients(1,0)*j+coefficients(2,0)) / (
coefficients(6,0)*i + coefficients(7,0)*j + 1));
          y = round((coefficients(3,0)*i+coefficients(4,0)*j+coefficients(5,0)) / (
coefficients(6,0)*i + coefficients(7,0)*j + 1));
         if(x<0||x>=nr||y<0||y>=nc) outimg(i,j,k)=0;
            else outimg(i,j,k) = inimg(x,y,k);
```

```
        }
      }
    }

    return outimg;



}
/** scale affine transformation
 @param inimg input image
 @param sx scale in x direction
 @param sy scale in y direction
 @return transformed image
*/
Image scale(Image& inimg, float sx, float sy) {
   int i,j,k;
  int nr,nc,nchan;
  Image outimg; //output image

  //construct matrix
  Image matrix(3,3);  //matrix
  Image matrix_inverse; //inverse transformation matrix
  Image corrected(3,1);
  Image original;
  int x,y;

  //sx 0 0
  //0 sy 0
  //0 0 1

  matrix(0,0) = sx;
  matrix(1,1) = sy;
  matrix(2,2) = 1;
    cout<<matrix<<endl;
 composite= matrix->*composite;
  matrix_inverse = inverse(matrix);

  //perform inverse transformation
  nr = inimg.getRow();
  nc = inimg.getCol();
  nchan = inimg.getChannel();
  outimg.createImage(nr,nc,nchan);


  //for each pixel in the new image
  for(i=0;i<nr;i++) {
    for(j=0;j<nc;j++) {

        corrected(0,0) = i;
        corrected(1,0) = j;
        corrected(2,0) = 1;

        //work out the position in the original image
        original = matrix_inverse ->* corrected;
        for(k=0;k<nchan;k++)  {
          x = round(original(0,0));
          y = round(original(0,1));
          if(x<0||x>=nr||y<0||y>=nc) outimg(i,j,k)=0;
             else outimg(i,j,k) = inimg(x,y,k);
        }
    }
  }

  return outimg;
}

/** shear affine transformation
 @param inimg input image
 @param hx shear in x direction
 @param hy shear in y direction
 @return transformed image
*/
Image shear(Image& inimg, float hx, float hy) {
   int i,j,k;
  int nr,nc,nchan;
  Image outimg; //output image

  //construct matrix
  Image matrix(3,3);  //matrix
  Image matrix_inverse; //inverse transformation matrix
  Image corrected(3,1);
  Image original;
  int x,y;

  //1 hx 0
  //hy 1 0
  //0 0 1
  matrix(0,0) = 1;
  matrix(0,1) = hx;
  matrix(1,0) = hy;
  matrix(1,1) = 1;
  matrix(2,2) = 1;

    cout<<matrix<<endl;
  composite= matrix->*composite;
  matrix_inverse = inverse(matrix);

  //perform inverse transformation
  nr = inimg.getRow();
  nc = inimg.getCol();
  nchan = inimg.getChannel();
  outimg.createImage(nr,nc,nchan);


  //for each pixel in the new image
  for(i=0;i<nr;i++) {
    for(j=0;j<nc;j++) {

        corrected(0,0) = i;
        corrected(1,0) = j;
        corrected(2,0) = 1;

        //work out the position in the original image
        original = matrix_inverse ->* corrected;
        for(k=0;k<nchan;k++)  {
          x = round(original(0,0));
          y = round(original(0,1));
          if(x<0||x>=nr||y<0||y>=nc) outimg(i,j,k)=0;
             else outimg(i,j,k) = inimg(x,y,k);
        }
    }
  }

  return outimg;
}

/** rotate affine transformation
 @param inimg input image
 @param rotation degree
 @return transformed image
*/
Image rotate(Image& inimg, float degree) {
```

```
   int i,j,k;
  int nr,nc,nchan;
  Image outimg; //output image

  //construct matrix
  Image matrix(3,3);  //matrix
  Image matrix_inverse; //inverse transformation matrix
  Image corrected(3,1);
  Image original;
  int x,y;

  matrix(0,0) = cos(degree);
  matrix(0,1) = -sin(degree);
  matrix(1,0) = sin(degree);
  matrix(1,1) = cos(degree);
  matrix(2,2) = 1;

  cout<<matrix<<endl;
  composite= matrix->*composite;

  matrix_inverse = inverse(matrix);

  //perform inverse transformation
  nr = inimg.getRow();
  nc = inimg.getCol();
  nchan = inimg.getChannel();
  outimg.createImage(nr,nc,nchan);


  //for each pixel in the new image
  for(i=0;i<nr;i++) {
    for(j=0;j<nc;j++) {

        corrected(0,0) = i;
        corrected(1,0) = j;
        corrected(2,0) = 1;

        //work out the position in the original image
        original = matrix_inverse ->* corrected;
        for(k=0;k<nchan;k++)  {
          x = round(original(0,0));
          y = round(original(0,1));

          if(x<0||x>=nr||y<0||y>=nc) outimg(i,j,k)=0;
          else outimg(i,j,k) = inimg(x,y,k);
        }
    }
  }

  return outimg;
}

/** translation transformation
 @param inimg input image
 @param tx move along x-axis
 @param ty move along y-axis
 @return transformed image
*/
Image translate(Image& inimg, float tx,float ty) {
  int i,j,k;
  int nr,nc,nchan;
  Image outimg; //output image

  //construct matrix
  Image matrix(3,3);  //matrix
  Image matrix_inverse; //inverse transformation matrix
  Image corrected(3,1);
  Image original;
  int x,y;

  matrix(0,0) = 1;
  matrix(0,2) = tx;
  matrix(1,1) = 1;
  matrix(1,2) = ty;
  matrix(2,2) = 1;

  cout<<matrix<<endl;
 composite= matrix->*composite;
  matrix_inverse = inverse(matrix);

  //perform inverse transformation
  nr = inimg.getRow();
  nc = inimg.getCol();
  nchan = inimg.getChannel();
  outimg.createImage(nr,nc,nchan);


  //for each pixel in the new image
  for(i=0;i<nr;i++) {
    for(j=0;j<nc;j++) {

        corrected(0,0) = i;
        corrected(1,0) = j;
        corrected(2,0) = 1;

        //work out the position in the original image
        original = matrix_inverse ->* corrected;
        for(k=0;k<nchan;k++)  {
          x = round(original(0,0));
          y = round(original(0,1));
          if(x<0||x>=nr||y<0||y>=nc) outimg(i,j,k)=0;
             else outimg(i,j,k) = inimg(x,y,k);
        }
    }
  }

  return outimg;
}
```

```
/*************************************************************
 * utility.cpp - commonly used supporting functions
 *
 *   - polarToCartesian : converts polar to cartesian coordinate
 *   - cartesianToPolar : converts cartesian to polar coordinate
 *   - boundaryCheck    : ensures that given number is within the given range
 *   - boundaryCheckF   : float-version of boundaryCheck
 *   - printHistogram   : print histogram
 * Author: Sang-hyeb(Sam) Lee (C) slee91@utk.edu
 *
 * Created: 01/11/08
 *
 * Modified:
 *  09/02/11: add boundaryCheckF() implementation by Sanghyeb Lee
 *            add printhisogram() implementation by Sanghyeb Lee
 *************************************************************/
#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <assert.h>
#include <cstring>
#include "utility.h"

using namespace std;

/* Print histogram
 * it prints out histogram of given image to the given file
 * @param inimg input image
 */
void printHistogram(Image& inimg, char* outputfile) {
    Image outimg;
  int i, j, k;
  int nr, nc, ntype, nchan;
  int h[(int)L+1];
  ofstream file;

 //alloate memory
  nr = inimg.getRow();
  nc = inimg.getCol();
  ntype = inimg.getType();
  nchan = inimg.getChannel();
  memset(h,0, (int)(L+1) * sizeof(int));

  //open file for writing
  file.open(outputfile);
 //only accepts single-channel image.
  if (nchan>1) {
    cout << "printHistogram: can only handle single-channel image\n";
    exit(3);
  }
  outimg.createImage(nr, nc, ntype);

  //open file

  //compute histogram h
  for (i=0; i<nr; i++)
    for (j=0; j<nc; j++)
      h[(int)inimg(i,j,0)]++;

  //print histogram to the output file
  for(i=0;i<=L;i++) {
    file<<i<<" "<<h[i]<<endl;
  }

}

/**
 * converts cartesian coordinate to polar coordinate
 *  r = sqrt(x^2+y^2)
 *  a = atan(y/x)
 *  where r is the radius (distance from the center) and a is the angle
 *  and (x,y) is the cartesian co-ordinate of the raster point.
 * @param x the x-coordinate of the point (int)
 * @param y the y-coordinate of the point (int)
 * @param rp the address of the float variable to hold radius
 * @param ap the address of the float variable to hold angle
 */
void cartesianToPolar(int x,int y,float *rp,float *ap) {
  *rp = sqrt(pow(x,2)+pow(y,2));
  *ap = atan2f(y,x);
}
/**
 * converts polar coordinate to cartesian coordinate
 *  x = r * cos(a)
 *  y = -r * sin(a)
 *  where r is the radius (distance from the center) and a is the angle
 *  and (x,y) is the cartesian co-ordinate of the raster point.
 * @param r the radius (float)
 * @param a the angle  (float)
 * @param xp the address of the int variable to hold x-coordinate
 * @param yp the address of the int variable to hold y-coordinate
 */
void polarToCartesian(float r,float a,int* xp,int *yp) {
  *xp = roundf(r * cosf(a));
  *yp = roundf(r * sinf(a));
}

/**
 * check if the given int number is within the given range. Otherwise, change
 * the number to either maximum or minimum so that it is in the range.
 * @param x  the number to check  (float)
 * @param lower the lower limit of the range
 * @param upper the upper limit of the range
 * @return number in the range
 */
int boundaryCheck(int x,int lower,int upper) {
  if(x<lower) return lower;
  else if(x>upper) return upper;
  else return x;
}

/**
 * check if the given float number is within the given range. Otherwise, change
 * the number to either maximum or minimum so that it is in the range.
 * @param x  the number to check  (float)
 * @param lower the lower limit of the range
 * @param upper the upper limit of the range
 * @return number in the range
 */
float boundaryCheckF(float x,float lower,float upper) {
  if(x<lower) return lower;
  else if(x>upper) return upper;
  else return x;
}
```

```
/*************************************************************************
 * Dip.h - header file of the Image processing library
 *
 * Author: Hairong Qi, hqi@utk.edu, ECE, University of Tennessee
 *
 * Created: 01/22/06
 *
 * Modification:
 *************************************************************************/

#ifndef DIP_H
#define DIP_H

#include "Image.h"

#define PI 3.1415926

////////////////////////////////////////////
// point-based image enhancement processing

Image cs(Image &,                   // contrast stretching
         float,                     // slope
         float);                    // intercept


////////////////////////////////////////////
// geometric transformation
typedef struct Point { //structure for holding a point
  float x;
  float y;
} Point;


Image geotranpoly2(Image& inimg, //Geometric transformation using 2nd poly
                   Point original[], //an array of original points
                   Point corrected[], //an array of corrected points
                   int size); //size of array

////////////////////////////////////////////
// sptial filter image enhancement processing

Image amean(Image& inimg, //apply arithmetic average filter
                int kernelSize) ; //kernel size

Image gmean(Image& inimg, //apply geometric average filter
                int kernelSize) ; //kernel size

#define DEFAULT_KERNEL_SIZE 3 //default starting kernel size
#define MAXIMUM_KERNEL_SIZE 21 //default maximum kernel size

Image amedian(Image& inimg, //apply median filter
              int startKernel=DEFAULT_KERNEL_SIZE, //starting kernel size
              int maxKernel=MAXIMUM_KERNEL_SIZE); //maximum kernel size

Image conv(Image&,  //convlution operator
           Image&); //kernel

Image sobel(Image&); //sobel edge detector

Image contrah(Image& inimg, //contraharmonic mean filter
              int kernelSize,//kernel size
              float Q);//order of the filter

Image adaptive(Image& inimg, //adaptive local noise reduction
               int kernelSize, //filter size
               float noise); //noise


////////////////////////////////////////////
// frequency domain filter
Image invFilter(Image input, //inverse deblurring filter
                float D_0); //radius

Image wiener(Image input, //wiener filter function
             float D_0,  //radius
             float K);  //constant

Image notch(Image input, //notch reject filter
            float D_0,
            float width);

////////////////////////////////////////////
// Transformation
Image translate(Image& inimg, //original image
                float tx, //along x-axis
                float ty); //along y-axis

Image rotate(Image& inimg, //rotate
             float degree); //rotation degree

Image shear(Image& inimg, //share
            float hx, //shear in x direction
            float hy);//shear in y direction

Image scale(Image& inimg, //scale
            float hx, //shear in x direction
            float hy);//shear in y direction

Image perspective(Image& inimg,//perspective transformation
                  Image& coefficients//coefficient matrix
                  );

Image perspective_coefficient(Point original[], //work out the coefficient matrix f
or perspective transformation
                              Point corrected[]);


////////////////////////////////////////////
// MISC
//print out histogram to the given file
void printHistogram(Image& inimg, //input image
                    char* outputfile); //output filename
//Add salt-and-pepper noise to an image
Image sapNoise(Image &inimg, //input image
               float q); //probability to add SAP

#endif
```

```
/*****************************************************************
 * utility.h - header file of the utility functions
 *
 * Author: Sang-hyeb Lee, slee91@utk.edu, ECE, University of Tennessee
 *
 * Created: 08/31/11
 *
 * Modification:
 *
 *****************************************************************/
#ifndef UTILITY_H
#define UTILITY_H

////////////////////////////////////////////
// coordinate conversion

//polar to cartesian coordinate
void polarToCartesian(float r, //radius in polar coordinate
                      float a, //angle in polar coordinate
                      int* xp, //address of the variable to hold x-coordinate
                      int *yp); //address of the variable to hold y-coordinate

//cartesian to polar coordinate
void cartesianToPolar(int x, //x-coordinate
                      int y, //y-coordinate
                      float *rp, //address of the variable to hold radius
                      float *ap); //addresss of the variable to hold angle

////////////////////////////////////////////
// misc

//ensures that given int number is within the range.
int boundaryCheck(int x, //number to check for
                  int lower, //lower limit of the range
                  int upper); //upper limit of the range

//ensures that given float number is within the range.
float boundaryCheckF(float x, //number to check for
                  float lower, //lower limit of the range
                  float upper); //upper limit of the range

//print out histogram to the given file
void printHistogram(Image& inimg, //input image
                    char* outputfile); //output filename
#endif
```