95+20

Image Enhancement (October 2011)

Ali Ghezawi

ECE 572 – Digital Image Processing

***Abstract*—We explored estimation of noise and its removal in space and frequency. Much of it dealt with salt and pepper (SAP) noise. Spatial noise (horizontal lines) were also considered as well as Rayleigh noise in a real photograph. Adaptive median dominated 'artificial' SAP (being maximum or minimum intensity) whereas the median won in removing real-world SAP. The contraharmonic was best at removing only salt or only pepper noise – even if it were real-world. Arithmetic and geometric means handled Rayleigh noise decently. A frequency notch filtered did wonders against horizontal scan lines. We also looked at the affine and perspective transform and serial versus composite transforms, using homogeneous coordinates. The affine preserved parallel lines whereas the perspective only did so parallel to the projection plane.**

I. Denoising

A. Estimation of Noise

FIGURE 1 shows an original image with noise in it. It also shows the portion of that image of relatively constant intensity used to measure the characteristics of the noise. The structure of the histogram in fig. 1, produced from the slice in red, was a Rayleigh probability density function (PDF). Equation (1) shows the form of this distribution.

$$p_Z(z) = \begin{cases} \frac{2}{b}(z-a)\exp\left(-\frac{(z-a)^2}{b}\right), & z \geq a \\ 0 & , z < a \end{cases} \quad (1)$$

With (2) and (3), a and b were estimated to be 63 and 17.59 respectively. The estimated mean was 66.79, and the unbias variance estimate was 3.775.

$$\text{var}\{Z\} = b\frac{4-\pi}{4} \quad (2)$$

$$\text{E}\{Z\} = a + \sqrt{b\frac{\pi}{4}} \quad (3)$$

The ideal model matched decently with the measurements. A Rayleigh is nonzero until a, and the first nonzero element in the histogram was 64, quite close to the approximated a.

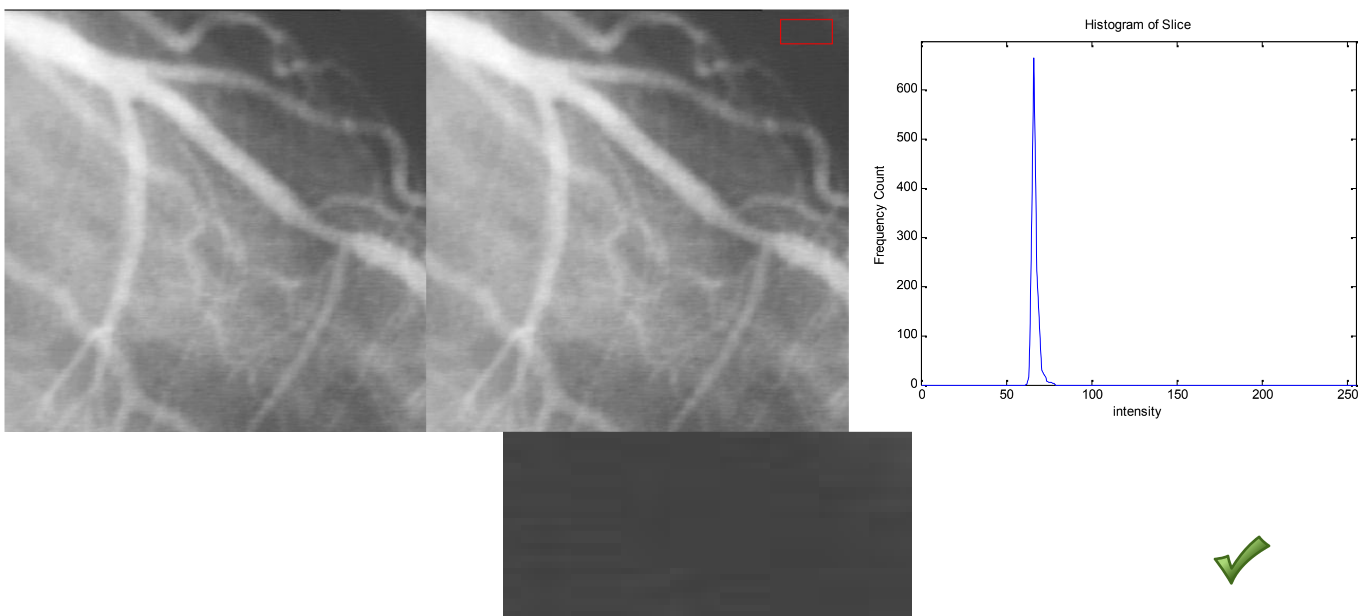
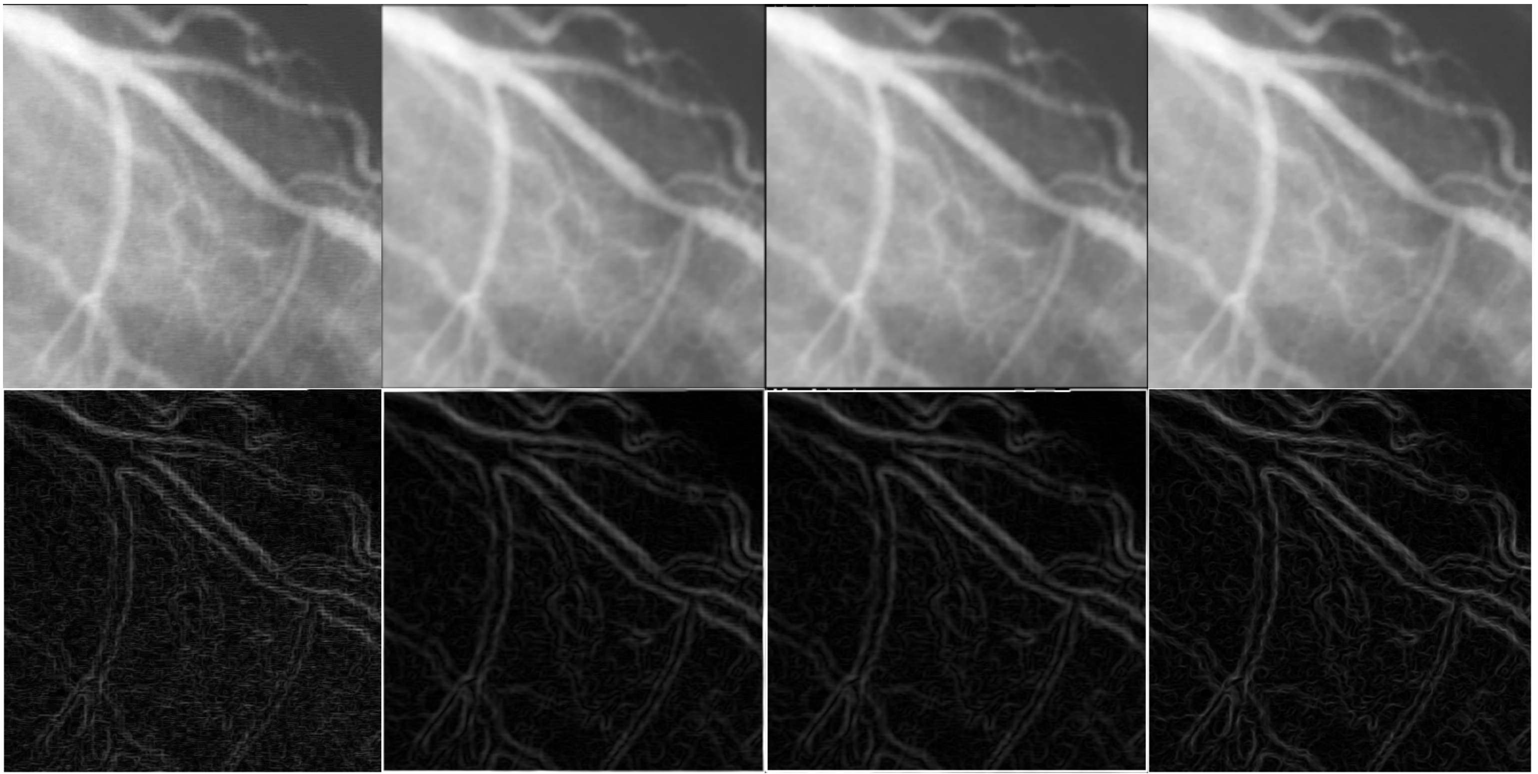


Fig. 1. Row one shows the original image, the original image with the rectangle used for noise estimation highlighted, and the histogram of that slice. Row 2 shows the slice enlarged.



Original 5x5 Arithmetic Mean 5x5 Geometric mean 5x5 Median

Fig. 2. Row one is an angiogram image described by the text below each column. Row two is the Sobel filtering of the image above it.

B. Edge Detection and Spatial Noise Removal

The mask sizes were increased until the Sobel filter produced decent results. For all filters, a 5x5 mask was chosen to provide decent performance under the constraint of using the lowest size possible in an effort to reduce blur.

Figure 2 shows an image with noise in it, various filtered versions of that image, and the use of a Sobel edge detector on each of the previously mentioned images. The arithmetic and geometric mean filters are good at removing Rayleigh and Gaussian noise. The geometric filter, however, tends to leave more detail compared to the arithmetic filter. The geometric filter is also horrible at removing pepper noise since a single 0 in a neighborhood makes the output zero. The median filter is best at removing salt-and-pepper (SP) noise. To the naked eye, all four images looked almost exactly the same except for a

(a) Original (b) 5x5 Median (c) A-Median 3x3 to 7x7 (d) A-Median 7x7 to 17x17

(e) 5x5 CH $Q = 2$ (f) 5x5 CH $Q = 1$ (g) 5x5 CH $Q = -1$ (h) 5x5 CH $Q = -2$

Fig. 3. Results for section *C.* Note: A-median stands for adaptive-median, and CH stands for contraharmonic.

small amount of noticeable blurring in the filtered images compared to the default image.

The Sobel filter revealed more than a naked-eye viewing did. The original image's edge image had so much Rayleigh noise that the sensitive Sobel filter produced unclear edges for the structures in the background, and the main structures in the forefront had scratchy-looking boundaries. Both the geometric and arithmetic filters produced much better edge images with finer detail in the background as well as edges not having a scratchy appearance. The median filter did everything the other two filters did except all edges had a much finer, less blurry look.

C. Contraharmonic and Adaptive Median Filtering

Figure 3 shows the results of applying contraharmonic (CH), median, and adaptive median filtering to an image corrupted with salt noise. The median filter did a fair job at removing the salt noise, expected since it is good at removing both salt and pepper noise. However, it created an odd cartoony blurring effect. The CH filter with Q = -1 actually removed the salt noise better than the median filter yet with a blur more comparable to a mean filter and more pleasing than the median's blur. In CH, a negative Q removed salt noise extremely well whereas a positive Q amplified it, still blurring the image. A positive Q would have worked against pepper noise. The adaptive median filter did almost nothing, visibly removing only a few slices from some of the bars of salty noise, or slightly reducing the intensity of a few isolated dots of noise. The adaptive median filter only works if salt noise is a constant value above most values in the image and pepper noise is a constant value below most values in the image. In this image, most of the salt noise must have been dwarfed in magnitude per each neighborhood by either other noise or other pixels in the image. Hence, almost no noise filtered out entirely. What noise that did filter out, however, supported the fact that noise dwarfed other noise, thus causing slicing in

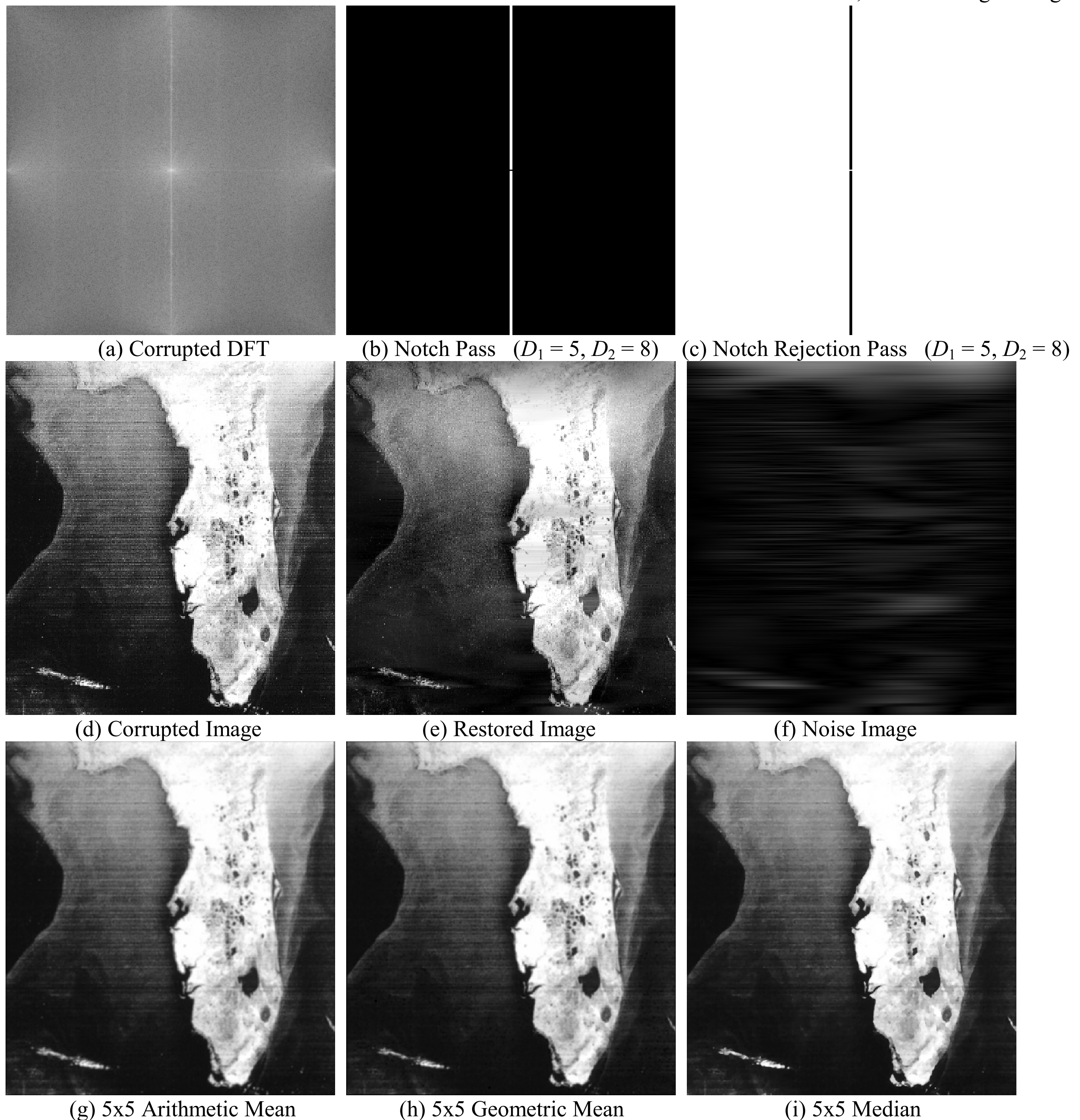


Fig. 4. Results for section *D*.

previously solid bars of noise.

D. Frequency Filtering

Figure 4.d shows an image corrupted by horizontal scanning lines, estimated by viewing the spatial image. A battery of spatial techniques was used to demonstrate the weakness of spatial filters toward this type of noise: arithmetic, geometric, and the nonlinear median filters all failed to remove the horizontal lines, the outputs of which are shown in fig. 4.g through fig. 4.i. And as fig. 5 shows, the noise was approximately Gaussian, meaning arithmetic and mean filters should have done a decent job, but they did not, indicating that sometimes frequency methods are needed.

The horizontal nature of the noise indicated vertical bars were to blame in the frequency domain. Thus, after viewing fig. 4.a and after a bit of tweaking, we produced a notch filter to target the main contributing frequency components of the noise while leaving the important DC component untouched. The filter had the form of

$$H(u,v)=\begin{cases}0, \left|u-\left\lfloor \frac{r}{2}\right\rfloor\right|>D_1 \wedge \left|v-\left\lfloor \frac{c}{2}\right\rfloor\right|\leq D_2 \\ 1, \text{otherwise}\end{cases} \quad (4)$$

where r and c were the number of rows and columns respectively in the DFT, D_1 equaled 5 and D_2 equaled 8. Figure 4.e shows the restoration, and fig. 4.d shows the noise filtered out in the spatial domain by applying a notch pass filter of the same form as (4) save the zero being a one and the one being a zero. Figures 4.b and 4.c show these filters.

II. Deblurring

Figures 6.a through 6.c show the noising and blurring process applied to Lena with the corruption quantified by the increasing mean-squared error (MSE) and peak signal-to-noise ratio (PSR) as well as subjectively by viewing the images.

The remainder in fig. 6 shows adaptive median filters with various maximum mask sizes being used on fig. 6.c. Since the adaptive median filter automatically stops at the smallest mask size needed, using too large an S_{max} was not an issue, shown empirically in the non-effect of increasing S_{max} above nine. However, if S_{max} was too small, some of the salt and pepper (SAP) noise staid. These results demonstrated the power of adaptive median filtering on SAP equal to zero and the maximum intensity. The PSR came closer to 14.3dB after the filtering. For its superior PSR as well as to be safe (since being too safe had no negative effect here), an S_{max} of 11 was chosen for the deblurring filters.

The first row shows the results of using an inverse filter on the denoised Lena. The inverse filter theoretically should have struggled on this type of deblurring since the convolution kernel presented an ill-posed problem. And since deblurring

(a) Original MSE = 0 PSR = ∞dB | (b) 3x3 Gaussian Blur MSE = 86.95 PSR = 14.3dB | (c) (b) with SAP q = .25 MSE = 9373 PSR = 4.2dB

(d) s_{max} = 5 MSE = 244.6 PSR = 12.1dB | (e) s_{max} = 7 MSE = 145.6 PSR = 13.2dB | (f) s_{max} = 9 MSE = 142.6 PSR = 13.3dB | (g) s_{max} = 11 MSE = 142.6 PSR = 13.3dB

Fig. 6. The first row contains images showing the blurring and noising process. The second row shows results from adaptive median filtering.

need to explain difference between inverse and forward transform -5

effectively amplified the remaining noise if H was small, which it was far away from the origin especially, a parameter to prevent inverse filtering outside a radius of R was used. In that case, the frequency content from the original image was used. Figure 7.a shows the result of using a pure inverse filter, offering terrible results visually and based on PSR. With more conservative Rs chosen, the PSR increased. Despite the lowered PSR and the periodic noise, fig. 7.b has better sharpness especially around the feathers in the hat compared to fig. 6.g. The Wiener filter was used too, shown in the row below in fig. 7. Similar to the previous results, PSR decreased relative to simply having a denoised image, yet details in the feathered region were more striking and closer to the original. The Wiener filter, at the least, beat the unadulterated inverse filter, and due to its lack of periodic noise after filtering, it may even beat the cutoff inverse filter. However, since it offered a constant K to be added and attenuate every frequency, the image appeared slightly darkened too, but that is the same device that allowed it not to amplify noise as badly.

Figure 8 shows the same procedure applied to the angiogram image. It reaffirmed the previous results.

III. Fundamental Transforms

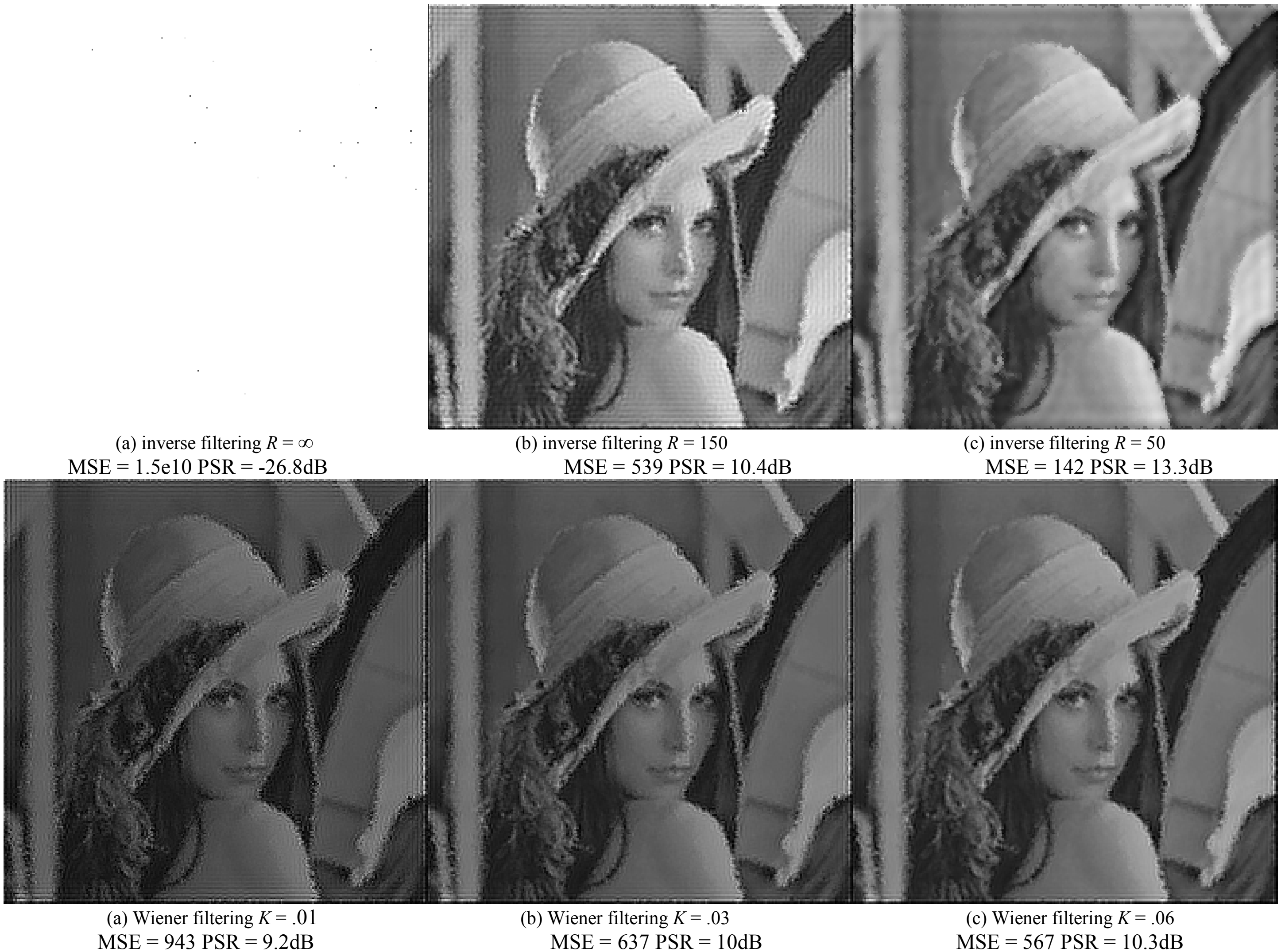
Figure 9 shows the results of performing affine and perspective transforms on the clock image. Figure 9.a is the original image, represented by the identity transform. Figures 9.b through 9.f were the results of performing the affine or perspective transform on the letter before it. Figure 9.g was the same series of transforms except performed in a single transform via a composite matrix.

The serial operations not only lost certain data in the series when the image moved out of frame, but it was also less efficient. On the other hand, the composite transform kept as much in the frame as possible and gained computational efficiency since the transform matrix need application only once instead of five times. The edges also seem smoother on the final result.

The affine transform seemed to preserve parallel lines whereas the perspective transform ravaged them if they were not parallel to the projection plane. In the perspective transform, the four old points were the corners and the four new points were the corner, save the bottom right had its coordinates multiplied by two.

IV. Adaptive Local Noise Reduction

The adaptive filter uses the variance of the random variable representing the noise added. Thus, since we knew the characteristics of the corrupting noise as SAP with a certain probability, the variance could be computed by the follwing estimate. The probability mass function (PMF) for each



(a) inverse filtering $R = \infty$
MSE = 1.5e10 PSR = -26.8dB

(b) inverse filtering $R = 150$
MSE = 539 PSR = 10.4dB

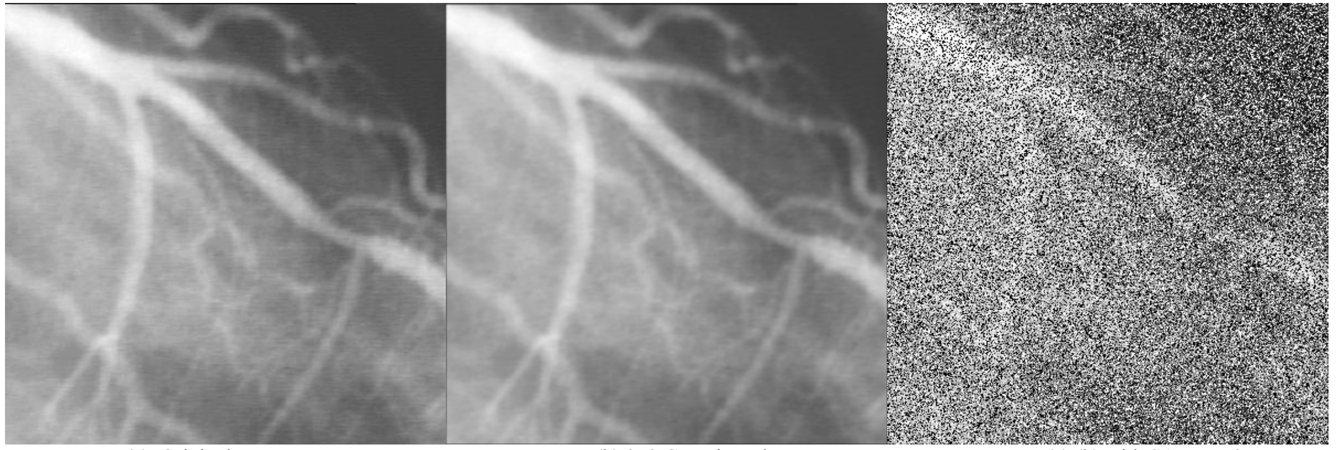
(c) inverse filtering $R = 50$
MSE = 142 PSR = 13.3dB

(a) Wiener filtering $K = .01$
MSE = 943 PSR = 9.2dB

(b) Wiener filtering $K = .03$
MSE = 637 PSR = 10dB

(c) Wiener filtering $K = .06$
MSE = 567 PSR = 10.3dB

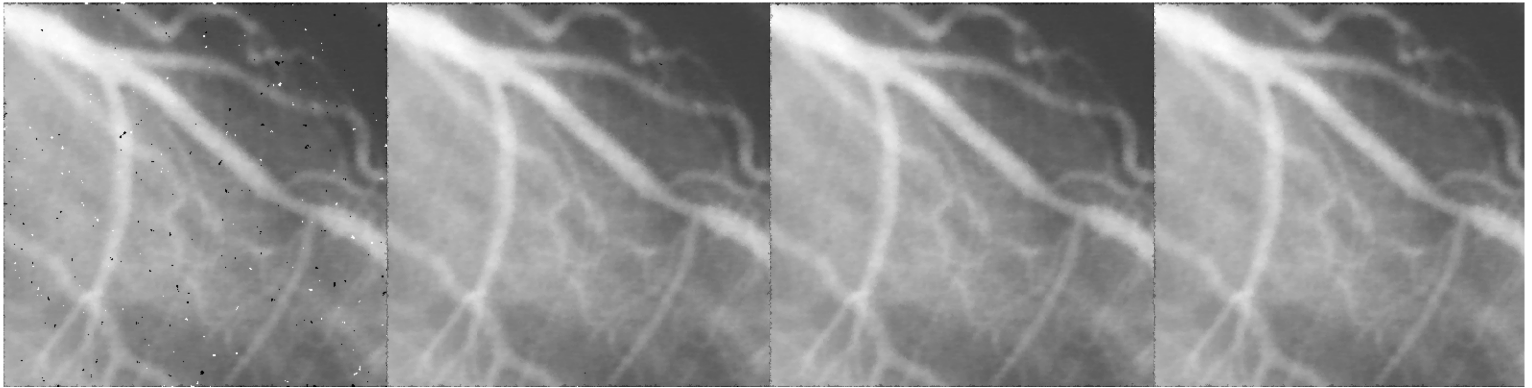
Fig. 7. The first row contains deblurring via inverse filtering. The second contains deblurring via Wiener filtering.



(a) Original
MSE = 0 PSR = ∞dB

(b) 3x3 Gaussian Blur
MSE = 22.4 PSR = 17.3dB

(c) (b) with SAP q = .25
MSE = 9091 PSR = 4.27dB

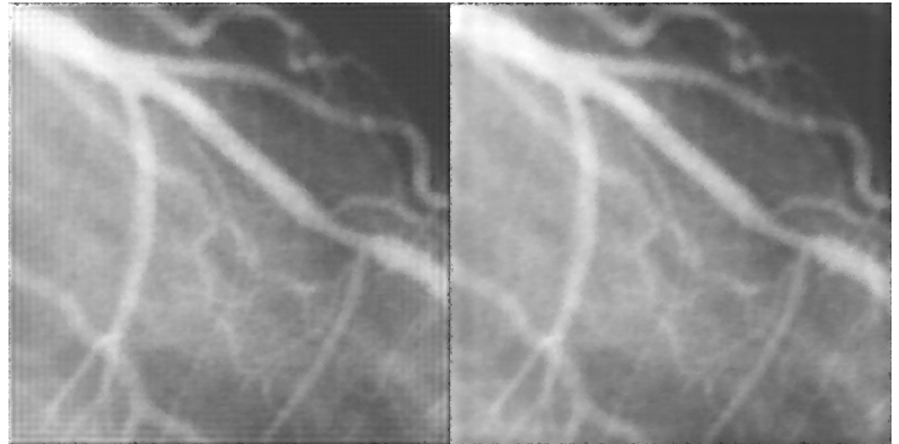


(d) $s_{max} = 5$
MSE = 110 PSR = 13.8dB

(e) $s_{max} = 7$
MSE = 30.97 PSR = 16.6 dB

(f) $s_{max} = 9$
MSE = 29.7 PSR = 16.7 dB

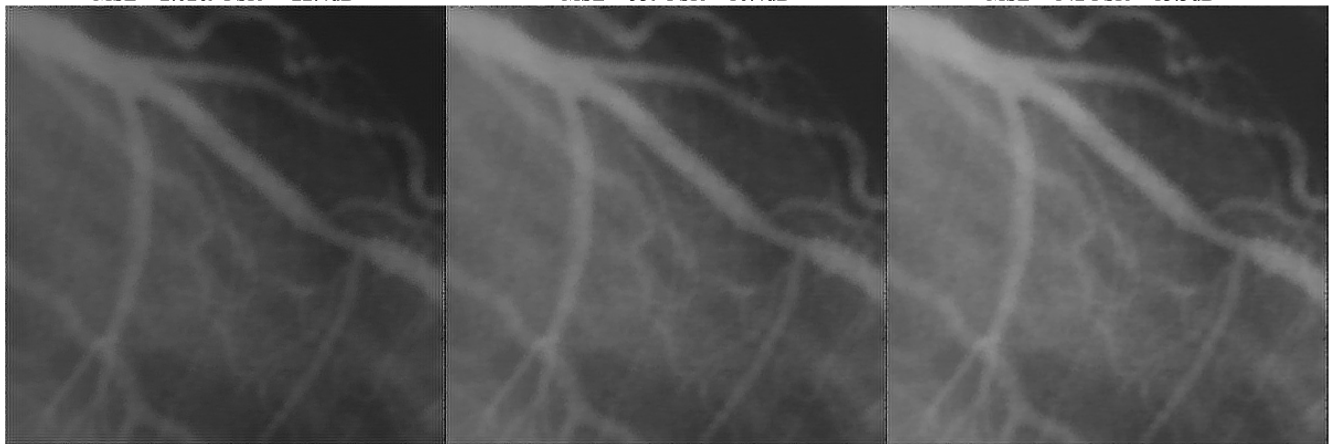
(g) $s_{max} = 11$
MSE = 29.7 PSR = 16.7dB



(a) inverse filtering $R = \infty$
MSE = 2.02e9 PSR = -22.4dB

(b) inverse filtering $R = 150$
MSE = 539 PSR = 10.4dB

(c) inverse filtering $R = 50$
MSE = 142 PSR = 13.3dB



(a) Wiener filtering K = .01
MSE = 268 PSR = 11.9dB

(b) Wiener filtering K = .03
MSE = 190 PSR = 12.7dB

(c) Wiener filtering K = .06
MSE = 210 PSR = 12.46dB

Fig. 8. Repeating the denoising process on angiogram image.

pixel was given by

$$P_{G(x,y)}\left(g\left(x,y\right)\right)=\begin{cases} q, g\left(x,y\right)=L \\ q, g\left(x,y\right)=0 \\ 1-2q, g\left(x,y\right)=f\left(x,y\right) \end{cases} \quad (5)$$

Thus, the noise's PMF was given by

$$P_{H_{(x,y)}}\left(\eta_{(x,y)}\right)=\begin{cases} q,\eta=L-f\left(x,y\right) \\ q,\eta=-f\left(x,y\right) \\ 1-2q,\eta=0 \end{cases} \quad (6)$$

where the subscript on H reminds us that the PMF depends on location due to the dependence on the original pixel value. In practice, this dependence is hard to work with. Thus, we replaced $f(x,y)$ by an approximated mean of the entire image, m. In doing so, the dependence on location vanished, and the PMF became

$$P_H\left(\eta\right)=\begin{cases} q,\eta=L-m \\ q,\eta=-m \\ 1-2q,\eta=0 \end{cases}$$

The variance then was computed simply as

$$\sigma_H^2=q\left(\left(L-m\right)^2+m^2\right)-\left(q\left(L-2m\right)\right)^2 \quad (7)$$

In practice, q was approximated by moving through g, the corrupted image, counting whether the pixel was L or 0. If not, that pixel then was used to compute an estimated m. Using this estimate, the variance for q = 0.25 was 8147, and for q = 0.10 was 3242. Figure 10 shows the original image, corrupted images, output from the adaptive median filtering, and output

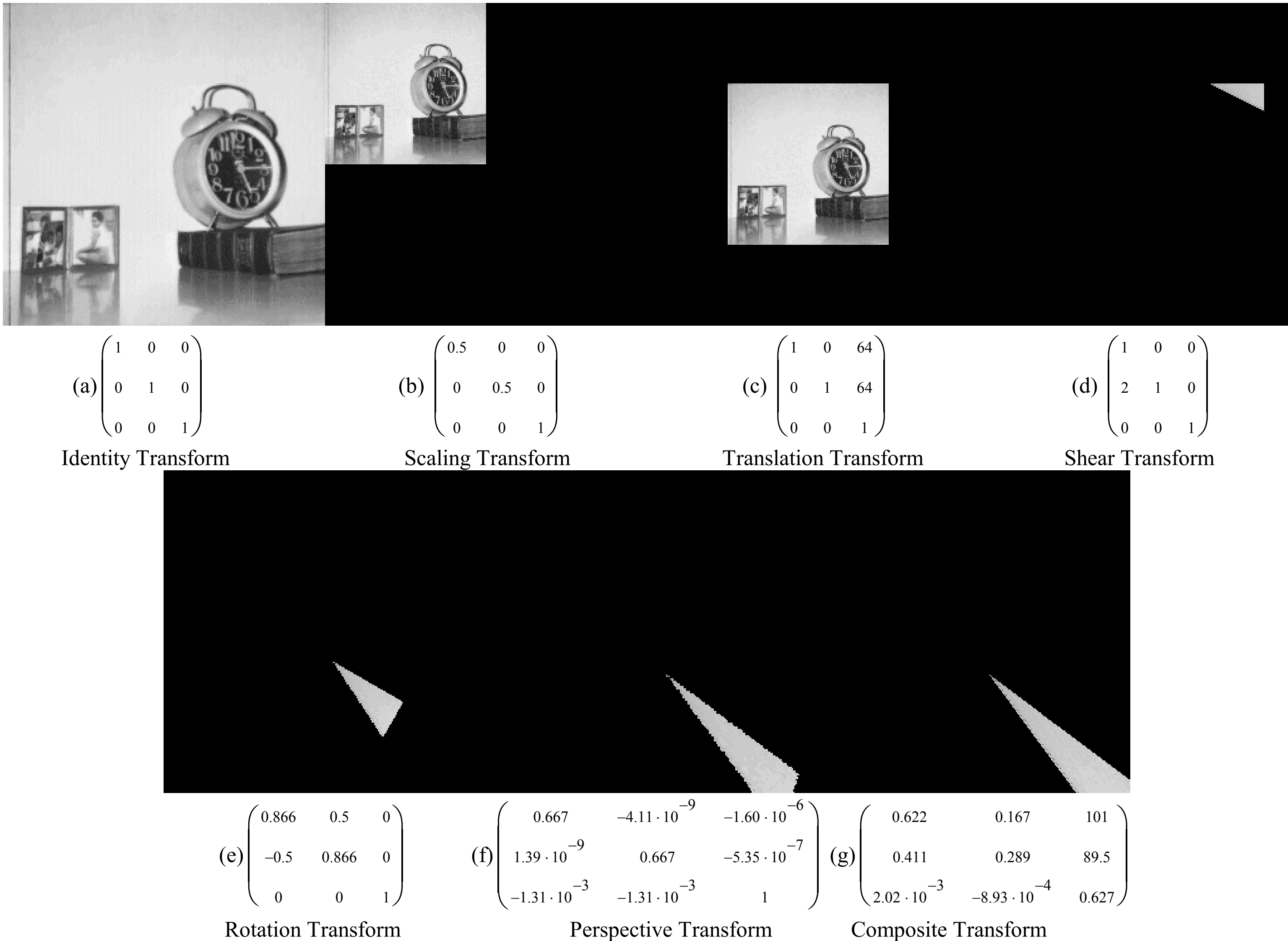


Fig. 9. Affine and perspective transforms. Except for (a) and (g), each letter transforms on the image from the previous letter.

from the adaptive local noise reduction filter. The adaptive median filter performed far better than the adaptive local noise reduction filter, perhaps due to the intense variance of SAP. The PSR of the adaptive median was much higher than the PSR of the competitor.

+10

V. Geometric Transformation with Polynomials

Figure 11 shows the original image, the original tie points, and the image after using polynomial geometric correction of order two. The new tie points were the same as the old tie points, except all points along the horizon were brought to a level of 116.

+10

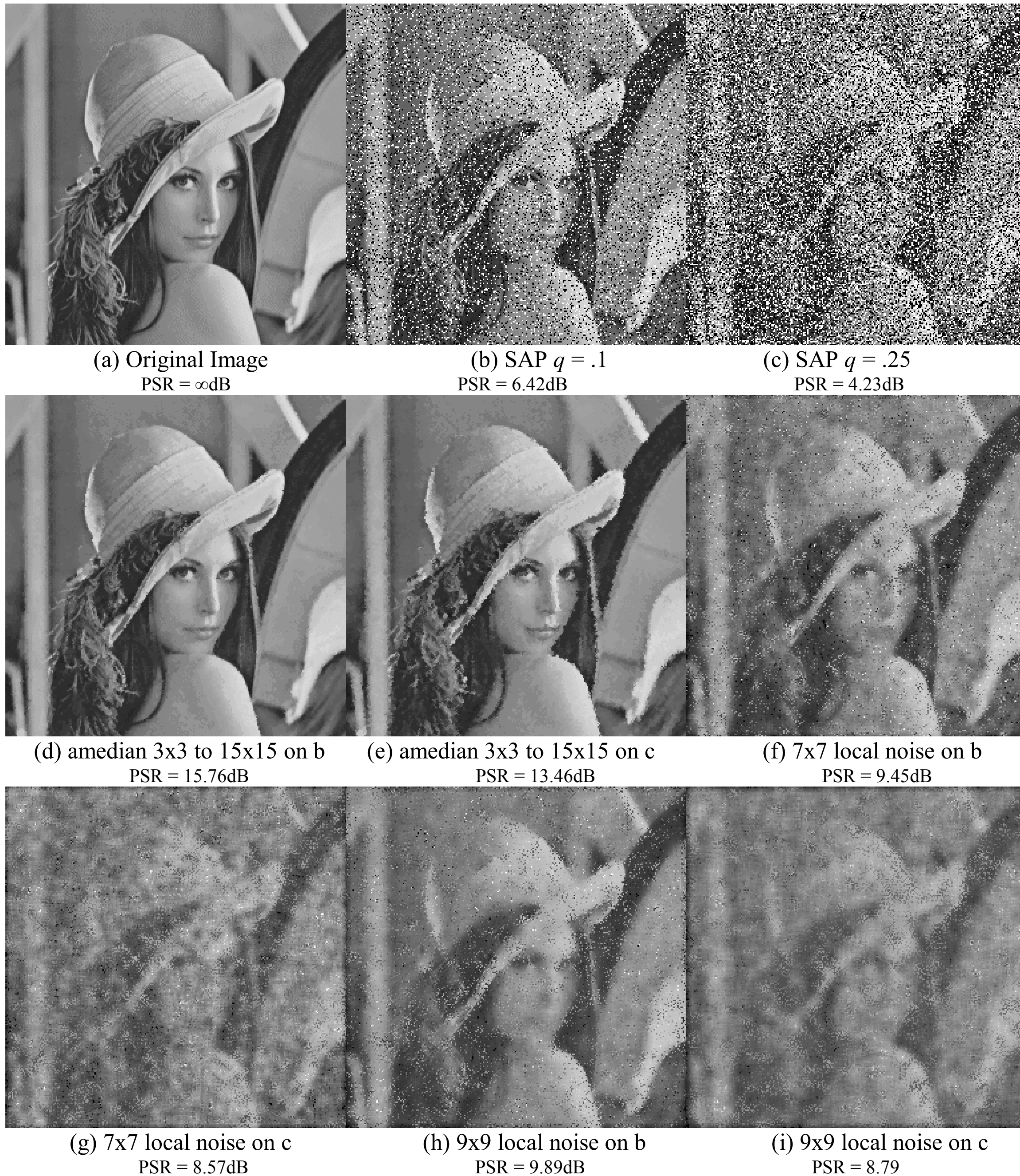
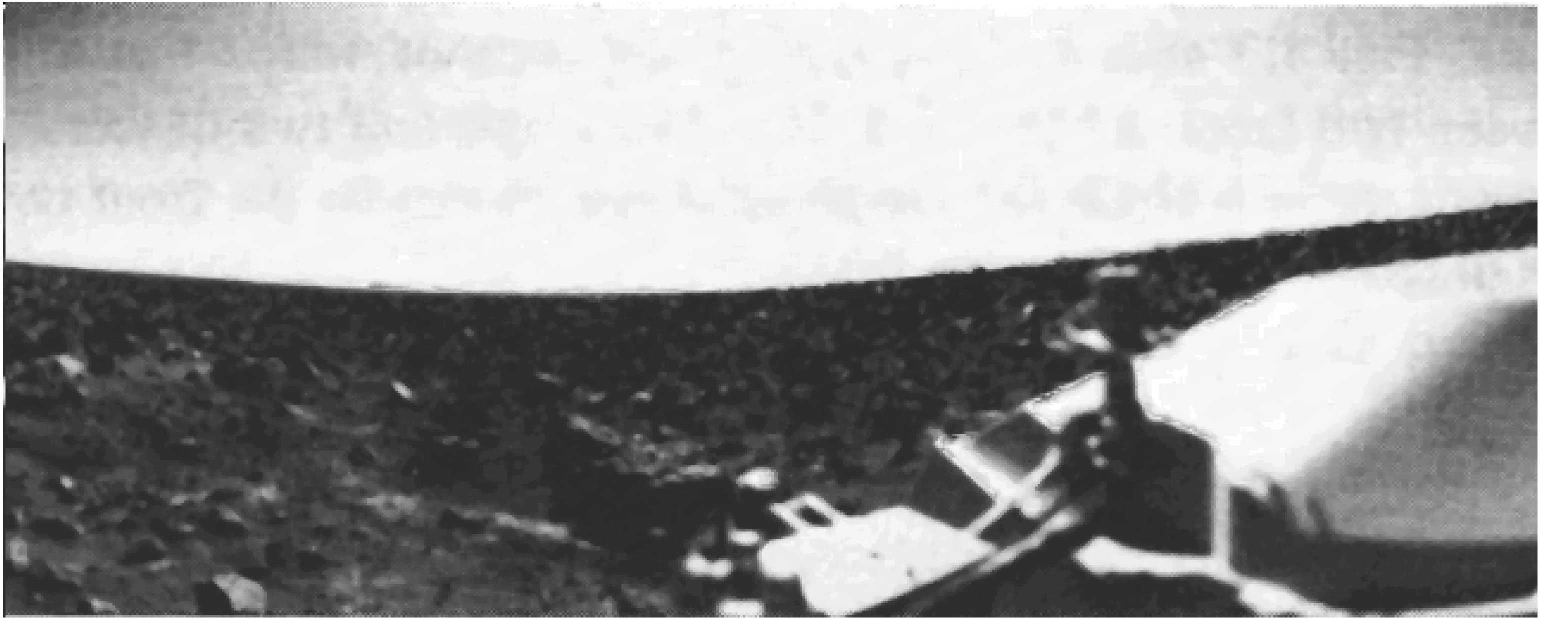
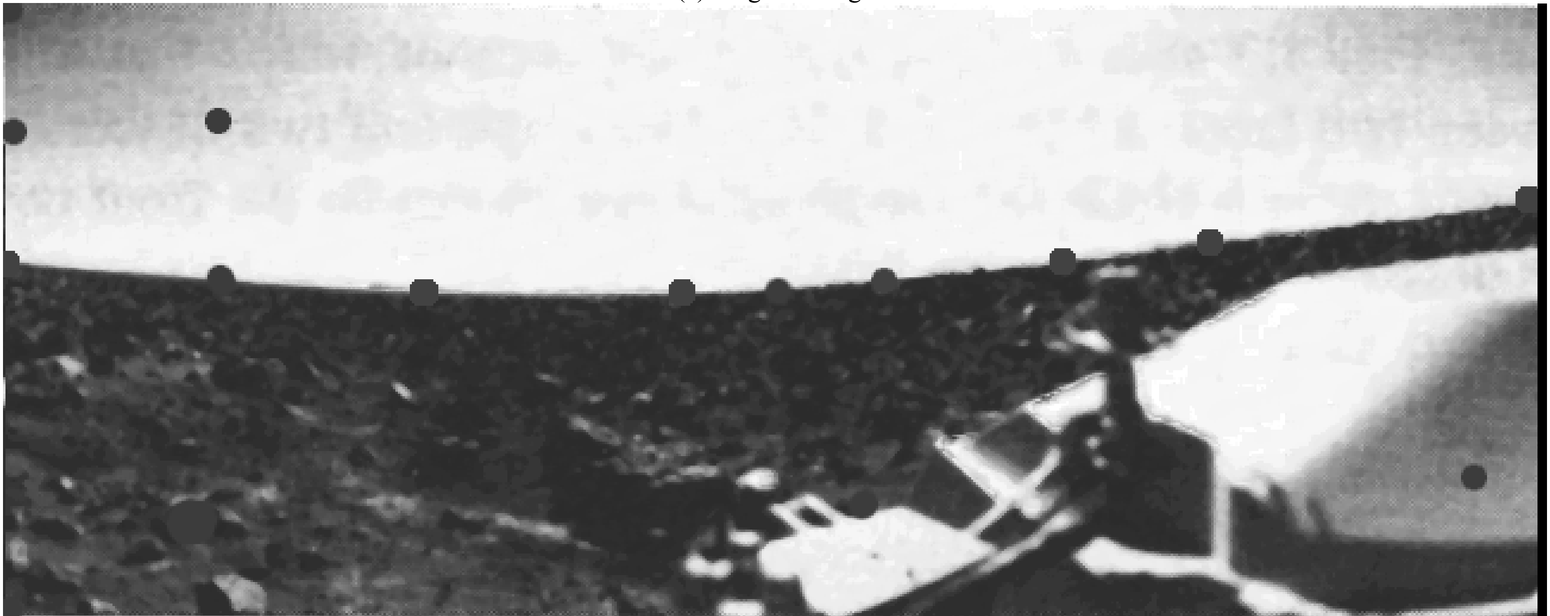


Fig. 10. Comparison between adaptive local noise reduction and adaptive median filtering.



(a) Original Image



(b) Original Tie points

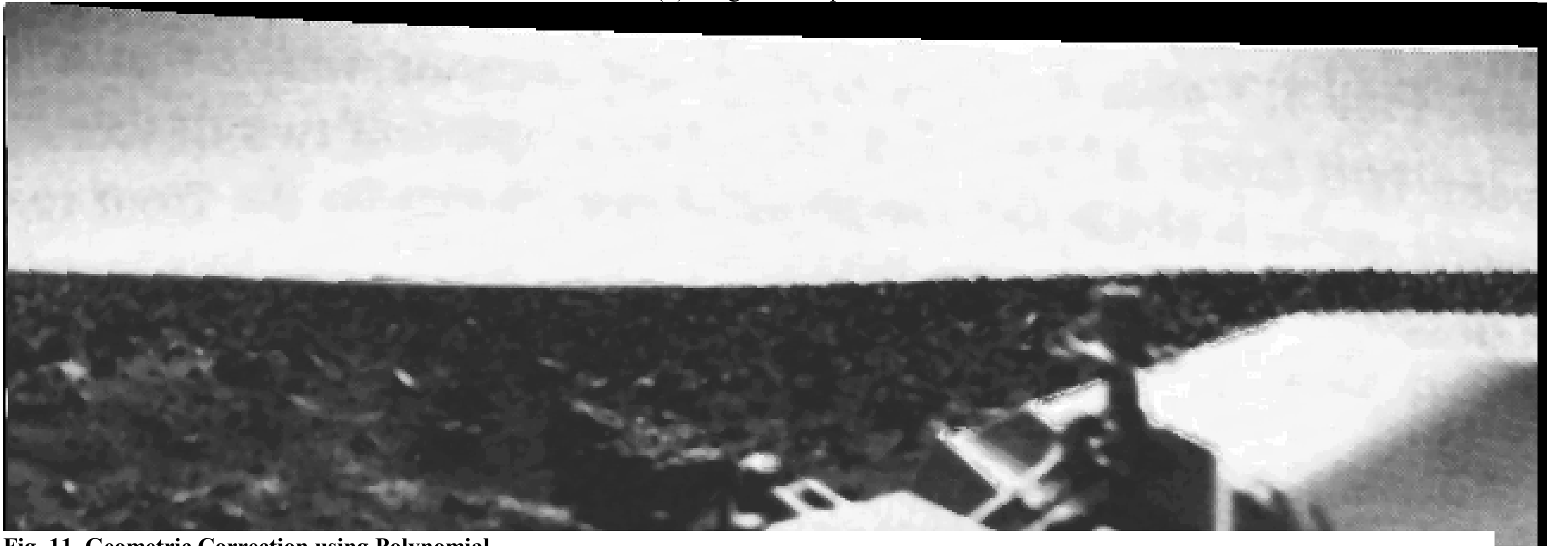


Fig. 11. Geometric Correction using Polynomial

VI. CODE

A. Dip.h

```
/********************************************************************
 * Dip.h - header file of the Image processing library
 *
 * Author: Hairong Qi, hqi@utk.edu, ECE, University of Tennessee
 * Author: Ali Ghezawi, aghezawi@utk.edu (all except cs())
 *
 * Created: 01/22/06
 *
 * Modification:
 *  9/24-25/11 added convolution and frequency stuff
 *  9/5/11 added gaussian noise function and point-based image enhancement
 *  8/22/11 added oil, swirl, edge, pixelMatrixMult
 ********************************************************************/

#ifndef DIP_H
#define DIP_H

#include "Image.h"

#define PI 3.1415926f


/*
 * geometric correction
 */
/*
 * geometric correction via polynomial
 * @param1 input image to correct
 * @param2 order of polynomial
 * @param3 new tie points (u,v) each row is a new one
 * @param4 old tie points (x,y) each row is a new one
 * @return geometrically corrected image
 */
Image polynomial(const Image& IN_IMG, const int& ORDER, const Image& TIE_NEW, const Image&
TIE_OLD);

/*
 * transformations
 */
/*
 * Takes an image and a transform matrix and applies it to each pixel using inverse
transform
 * @param1 image to transform
 * @param2 transform matrix (3x3)
 */
Image transform(const Image& IN_IMG, const Image& TRANSFORM);


/*
 * rotates image
 * @param1 input image to rotates
 * @param2 amount to rotate (radians)
 * @return rotated image
 */
Image rotation(const Image& IN_IMG, const float& THETA);

/*
 * translates image
 * @param1 input image to translate
 * @param2 x translation
 * @param3 y translation
 * @return translated image
 */
Image translation(const Image& IN_IMG, const float& TRANS_X, const float& TRANS_Y);

/*
 * shear image
 * @param1 input image to shear
 * @param2 x shear
 * @param3 y shear
 * @return shear image
 */
Image shear(const Image& IN_IMG, const float& SHEAR_X, const float& SHEAR_Y);

/*
 * scale image
 * @param1 input image to scale
 * @param2 x scale
 * @param3 y scale
 * @return scale image
 */
Image scaling(const Image& IN_IMG, const float& SCALE_X, const float& SCALE_Y);

/*
 * creates rotation matrix
 * @param1 angle to rotate
 * @return rotation matrix
 */
Image rotationMatrix(const float& THETA);

/*
 * creates translation matrix
 * @param2 x translation
 * @param3 y translation
 * @return translation matrix
 */
Image translationMatrix(const float& TRANS_X, const float& TRANS_Y);

/*
 * creates shear matrix
 * @param2 x shear
 * @param3 y shear
 * @return shear matrix
 */
Image shearMatrix(const float& SHEAR_X, const float& SHEAR_Y);
```

```
/*
 * creates scaling matrix
 * @param2 x scaling
 * @param3 y scaling
 * @return scaling matrix
 */
Image scalingMatrix(const float& SCALE_X, const float& SCALE_Y);



/*
 * Performs perspective transform
 * @param1 input image to transform
 * @param2 4 original points (each row is an x,y set)
 * @param3 4 new points (each row is an x,y set)
 * @return transformed image
 */
Image perspective(const Image& IN_IMG, const Image& ORIGINAL, const Image& NEW);


/*
 * Creates perspective matrix from 4 original/new points
 * @param1 4 original points (each row is a set of points)
 * @param2 4 new points (each row...)
 * @return matrix
 */
Image perspectiveMatrix(const Image& ORIGINAL, const Image NEW);

//returns z by z identity
Image identity(int Z);

/*
 * deblurring
 */
// inverse filtering
Image invFilter(const Image&, //input image to deblur
    const Image&,             //mask responsible for blur
    const bool&,              // whether mask is freq or spatial
    const Image&,             // if isFreq, can provide phase of mask too else give
Image(1,1) to ignore
    const float&,             // cutoff radius in freq domain before inv filtering is not
used
    const bool&);             // whether to ignore freq content outside circle defined by
above argument or use input image's content untouched.

// approximate wiener filtering (k)
Image wiener(const Image&,   //corrupted image to deblur
    const Image&,            // mask responsbile for the blur
    const bool&,             // whether mask is freq or spatial
    const Image&,            // if isFreq, can provide phase of mask too else give
Image(1,1) to ignore
    const float);            // K = ratio of power spectrum of noise to g

// calculates MSE and psnr
void accuracy(const Image&,  // uncorrupted image
    const Image&);           // f^, recovered image from g


/*
 * Frequency filtering
 */
//uses high pass guassian as base to do homomorphic filtering
Image homomorphic(const Image&,       // input image to filter
    const float&,                     // high frequency
    const float&,                     // low frequency
    const float&);                    // cutoff frequency

// filters image in freq domain based on spatial mask
Image freqSpatialMaskFilter(const Image&,     // image to filter (regular size)
    const Image&,                             // mask (spatial domain, regular size)
size(arg1) > size(arg2)
    const bool& isCONV = true);               // whether to convolve or correlalate

// convolves IN_IMG with MASK in frequency domain using lp butterworth
Image lpButterworth(const Image&,    //image to convolve mask with
    const float&,                    //cutoff freq
    const int&);                     //order

// convolves IN_IMG with MASK in frequency domain using lp gaussian
Image lpGaussian(const Image&,    // image to convolve mask with
    const float&);                // cutoff freq

// convolves IN_IMG with MASK in frequency domain using hp butterworth
Image hpButterworth(const Image&,    //image to convolve mask with
    const float&,                    //cutoff freq
    const int&);                     //order

// convolves IN_IMG with MASK in frequency domain using hp gaussian
Image hpGaussian(const Image&,    // image to convolve mask with
    const float&);                // cutoff freq

// computes the 'convolution' and outputs power ratio
Image filter(const Image&,                 //regular-sized image to 'convolve' with
    const Image&,                          //padded mask in frequency domain
    const bool& SUPPRESS = true,           // whether to suppress the power ratio output
    const Image& MASK_PHASE = Image(1,1),  // mask phase, defaults to 1by1, indicating not
to use it
    const bool& isCONV = true,             //whether to convole or correlate
    const int& EXTRACT_OFFSET = 0);        //offset in extracting the final image to get
'same' convolution.

Image padPow2(const Image&,            //input image to resize with zeros (also makes sure
power of 2)
    const Image& GOAL = Image(1,1));   //defines a goal row by col to resize to (1,1) means
none
```

```
// ideal notch rejection filter filtering all components defined within radius2 and radius
1
Image notch(const Image&,            //in image
    const float&,                    //d for row
    const float&,                    //d for col
    const bool& isREJECT = true);    // isREJECT

/*
 * Spatial filtering
 */
Image adaptiveLocalNoiseFilter(const Image&,  // corrupted image
    const int&,                               // mask size
    const float&);                            // variance of noise

Image amean(const Image&,  // image to perform regular average filter on
    const int&);           // mask defining neighborhood size

Image um(const Image& IN_IMG,  // input image to filter
    const int& MASK_SIZE);     // filter size for average

Image laplacian(const Image& IN_IMG); // input image to filter

Image sobel(const Image&,          // image to filter
    const bool& APPROX = false);   // whether to approximate (for speed)

Image lpGaussian(const Image&, // image to filter
    const int&,                // mask size for gassuain mask
    const float&);             // standard deviation of gaussian filter

Image median(const Image&,     // image to filter
    const int&);               // mask size for median search

// computes median of a single neighborhood.
float median(const Image&,    // image to compute the median within
    const int&,               // mask size (area to compute median within)
    const int&,               // row to center around
    const int&,               // col to center around
    const int&);              // channel to work in

// computes geometric filtering
Image gmean(const Image&,  // in image
    const int&);           // mask size

// contraharmonic filtering
Image contrah(const Image& IN_IMG,   // in image
    const float& Q,                  // q in filter
    const int& MASK_SIZE);           // neighborhood size

// raise image to a power
Image powImage(const Image& IN_IMG,  // in imaeg
    const float& POWER);             // power to raise to

// adaptive median filtering
Image amedian(const Image&,  // in image
    const int&,              // beginning mask size (3)
    const int&);             // final mask size

// returns adaptive median filtering for a single pixel
float amedianNeigh(const Image&,  //image to work in
    int,                          // mask size to start at
    const int&,                   // last mask size to end at
    const int&,                   // row in image to work at
    const int&);                  // col to work at

/*
 * Convolution
 */
// convolves IN_IMG with MASK, dividing each neighborhood multiplication by COEF
// NOTE: assume flipping the mask results in the mask
Image conv(const Image&,             //image to convolve mask with
    const Image&,                    //mask
    const bool&);                    //normalize mask first?

// sum(sum(MASK*neighborhoodOfPixel)). Mask nbyn, n%2 !=0
float pixelMatrixMult(const Image&, //input image where operation is taking place
    const Image&,                   //mask that defines the area of the operation
    const int&,                     //row in param1 where operation is centered around
    const int&,                     //col in param1 where operation is centered around
    const int&);                    //the channel under which the operation is taking place

/*
 * Line
 */
// returns black image with white vertical line in it
Image line(const int&, // rows in image
    const int&,        // cols in image
    const int&,        // row coordinate of top of line
    const int&,        // col coordinate of top of line
    const int&,        // length of line (vertical)
    const int&);       // thickness of line (horizontal)

/*
 * White Square
 */
// returns arg1 with white square at center sized arg2 by arg2.
Image square(const int&, // row# of picture
    const int&,          // col# of picture
    const int&);        // length of square to put at the center of image.

// calculates a scaling factor then does log transfer on input. Input will then
// contain a scaled, log-transformed image.
void automaticLog(Image&);

/*
 * noise
 */
// puts salt and pepper noise in image
Image sapNoise(const Image&,   // input image to corrupt
    const float&);            // higher q == higher noise.

// Returns a single sample of a Gaussian random variable with mean 0
float gaussianNoise(const float&);

// returns input image with noise from gaussianNoise added to it.
Image addGaussNoise(const Image&, const float&);    // input image

// returns estimate for SAP variance
float sapVar(const Image&,         // corrupted image
    const bool& SHOW_VAR = false); //whether to show computed variance

/*
 * point-based image enhancement processing
 * note: all can work for RGB pictures, but the
 * algorithms are designed for graycale.
 */
// samples an image by S
Image sampling(const Image&,  // input image
    const int&);              // S

// quantizes an image's levels
Image quantization(const Image&,  // input image
    const int&);                  // # of levels, > 1

// performs log transformation on each pixel
Image logtran(const Image&,       // input image
    const float&);                // scaling factor

// performs power law (gamma) transformation on each pixel
Image powerlaw(const Image&,  // input image
    const float&,             // scaling factor
    const float&);            // gamma

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

// performs threshold effect
Image threshold(const Image&,      // input image
    const float&,                  // bottom threshold
    const float&);                 // top threshold

// performs bitplane slicing
Image bitplane(const Image&,     // input image
    const int&);                  // BIT to slice

// performs histogram equalization
Image histeq(const Image& ); // input image

// saves histogram of image to a file
void saveHist(const Image&,   // input image to take hist of
    const int&,               // what channel to operate in
    char*);                   // filename to save hist data to

// performs localized histogram equalization. Each pixel
// has histeq() done to it using a ROW by COL box around
// that pixel. ROW & COL must be odd & > 1
Image localHistEq(const Image&,     // input image
    const int&,                     // ROW
    const int&);                    // COL

// Swirls an image about its center.
Image swirl(const Image&,            // Input image
    const float&);                   // swirl coefficient

// outputs an image with edges highlighted.
Image edge(const Image& IN_IMG);     // Input image
// Makes an image look as if it were painted with oil paints.
Image oil(const Image&,              // Input image
    const int&);                     // arg2 = (s-1)/2 of search sqr
#end
```

B. *Project4.cpp*

```
#include "Dip.h"
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <cstdio>

using namespace std;

#define USAGE "Project4 inImg outImg taskNumber[optional]\n"

int main(int argc, char **argv)
{
    /*
     * Manage input
     */
    char taskNumber[3];
    if (argc < 3)
    {
        cout << USAGE;
        exit(3);
    }
    else if(argc != 4)
    {
        cout << "1: PDF Estimation" << endl;
        cout << "2: Spatial removal of noise (1)" << endl;
        cout << "3: Spatial removal of noise (2)" << endl;
        cout << "4: Frequency removal of noise" << endl;
        cout << "5: Deblurring" << endl;
        cout << "6: Fundamental transforms" << endl;
        cout << "7: adaptive local noise reduction" << endl;
        cout << "8: geometric correction using polynomial approximation" << endl;


        cout << endl << "Selection : ";

        cin >> taskNumber;
        cout << endl;
    }
    else
    {
        taskNumber[0] = argv[3][0];
        taskNumber[1] = argv[3][1];
    }
```

```
    if(taskNumber[1] != '\0' || taskNumber[0] > '8' || taskNumber[0] < '1')
    {
        cout << "Invalid selection." << endl;
        exit(3);
    }

    const Image IN_IMG = readImage(argv[1]);

    // for naming certain outputs
    const int NAME_SIZE = strlen(argv[2]);
    char* outputName = new char[NAME_SIZE + 40];
    strcpy(outputName, argv[2]);
    outputName[NAME_SIZE - 4] = '\0';

    if(taskNumber[0] == '1') // pdf estimation
    {
        int points[4];
        cout << "Enter coordinate of bottom left then top right part of rectangle to grab: ";
        cin >> points[0] >> points[1] >> points[2] >> points[3];

        Image slice(points[0] - points[2] + 1, points[3] - points[1] + 1);
        for(int row = points[2]; row < points[0] + 1; ++row)
            for(int col = points[1]; col < points[3] + 1; ++col)
                slice(row - points[2], col - points[1]) = IN_IMG(row, col);

        saveHist(slice, 0, "noiseHistogram.txt");

        // compute mean
        float mean = 0;
        for(int row = 0; row < slice.getRow(); ++row)
            for(int col = 0; col < slice.getCol(); ++col)
                mean += slice(row, col);
        mean /= slice.getRow()*slice.getCol();

        // compute variance (divide by n - 1)
        float var = 0;
        for(int row = 0; row < slice.getRow(); ++row)
            for(int col = 0; col < slice.getCol(); ++col)
                var += pow(slice(row, col) - mean, 2.0f);
        var /= slice.getRow()*slice.getCol() - 1;

        cout << "Mean: " << mean << endl << "Var: " << var << endl;

        // save the slice
        strcat(outputName, "slice.pgm");
        writeImage(slice, outputName);
    }
    else if(taskNumber[0] == '2') // spatial removal using amean, gmean, median on angiogram
    {
        int sizes[3];
        cout << "input mask size for amean, gmean, and median: ";
        cin >> sizes[0] >> sizes[1] >> sizes[2];

        // output sobel of original image
        strcat(outputName, "originalSobel.pgm");
        writeImage(sobel(IN_IMG), outputName);
        outputName[NAME_SIZE - 4] = '\0';

        // amean + then sobel
        Image filtered = amean(IN_IMG, sizes[0]);
        strcat(outputName, "amean.pgm");
        writeImage(filtered, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        strcat(outputName, "ameanSobel.pgm");
        writeImage(sobel(filtered), outputName);
        outputName[NAME_SIZE - 4] = '\0';

        //gmean + then sobel
        filtered = gmean(IN_IMG, sizes[1]);
        strcat(outputName, "gmean.pgm");
        writeImage(filtered, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        strcat(outputName, "gmeanSobel.pgm");
        writeImage(sobel(filtered), outputName);
        outputName[NAME_SIZE - 4] = '\0';

        //median + then sobel
        filtered = median(IN_IMG, sizes[2]);
        strcat(outputName, "median.pgm");
        writeImage(filtered, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        strcat(outputName, "medianSobel.pgm");
        writeImage(sobel(filtered), outputName);
    }
    else if(taskNumber[0] == '3') // spatial removal using contraharmonic, median, and adaptive median  (wizard of
oz)
    {
        int sizes[3], q, maxMask;
        cout << "Enter mask size (or starting mask_size) for contraharmonic, median, and adaptive median filters:
";
        cin >> sizes[0] >> sizes[1] >> sizes[2];
        cout << "Enter Q: ";
        cin >> q;
        cout << "Enter maximum mask size for adaptive median filter: ";
        cin >> maxMask;

        // contraharmonic
        Image filtered = contrah(IN_IMG, q, sizes[0]);
        strcat(outputName, "contraharmonic.pgm");
        writeImage(filtered, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        // median
        filtered = median(IN_IMG, sizes[1]);
        strcat(outputName, "median.pgm");
        writeImage(filtered, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        // adaptive median
        filtered = amedian(IN_IMG, sizes[2], maxMask);
        strcat(outputName, "adaptiveMedian.pgm");
        writeImage(filtered, outputName);
    }
    else if(taskNumber[0] == '4') // frequency removal of noise (florida)
    {
        float d1, d2;
        cout << "Enter a row d, col d: ";
        cin >> d1 >> d2;

        // get fft of input
        Image padImg = padPow2(IN_IMG);
        Image mag(padImg.getRow(), padImg.getCol(), padImg.getChannel());
        Image phase(padImg.getRow(), padImg.getCol(), padImg.getChannel());
        fft(padImg, mag, phase);
        strcat(outputName, "INPUT_FOURIER.pgm");
        automaticLog(mag);
        writeImage(mag, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        // do notch filtering and output them
        Image restored = notch(IN_IMG, d1, d2, true);
        Image noise = notch(IN_IMG, d1, d2, false);

        strcat(outputName, "restored.pgm");
        writeImage(restored, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        strcat(outputName, "noise.pgm");
        writeImage(noise, outputName);

        // do spatial filtering & pdf estimation using 1 & 2
    }
    else if(taskNumber[0] == '5') // deblurring
    {
        const int MASK_SIZE = 3;
        const float SD = 1.5;
        float cutoff;
        int smax[5];
        float k[3];
        cout << "Enter 5 smax for amedian: ";
        cin >> smax[0] >> smax[1] >> smax[2] >> smax[3] >> smax[4];
        cout << "Enter three K for wiener filter: ";
        cin >> k[0] >> k[1] >> k[2];
        cout << "Enter cutoff for inv filter: ";
        cin >> cutoff;

        // blur
        const Image BLURRED = lpGaussian(IN_IMG, MASK_SIZE, SD);
        strcat(outputName, "blurred.pgm");
        writeImage(BLURRED, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        cout << "Blur" << endl;
        accuracy(IN_IMG, BLURRED);
        cout << endl;

        // add noise
        const Image G = sapNoise(BLURRED, .25);
        strcat(outputName, "blurredNoise.pgm");
        writeImage(G, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        cout << "blur and noise" << endl;
        accuracy(IN_IMG, G);
        cout << endl;

        // create convolution mask for inv filter and wiener filter
        Image mask(MASK_SIZE, MASK_SIZE);
        const float SHIFT = (MASK_SIZE-1)/2;
        const float SDSD2 = 2*SD*SD;
        float sum = 0.0f;
        for(int row = 0; row < mask.getRow(); ++row)
            for(int col = 0; col < mask.getCol(); ++col)
            {
                mask(row, col) = std::exp(-(std::pow(row-SHIFT,2)+std::pow(col-SHIFT,2))/SDSD2)/PI/SDSD2;
                sum += mask(row, col);
            }
        mask = mask/sum;

        // do for each smax
        for(int smaxTrial = 0; smaxTrial < 5; ++smaxTrial)
        {
            char str[15];

            cout << "smax: " << smax[smaxTrial] << endl;

            // filter using amedian
            const Image NOISE_GONE = amedian(G, 3, smax[smaxTrial]);
            strcat(outputName, "amedian");
            sprintf(str,"%d", smax[smaxTrial]);
            strcat(outputName, str);
            strcat(outputName, ".pgm");
            writeImage(NOISE_GONE, outputName);
            outputName[NAME_SIZE - 4] = '\0';
            cout << "Noise gone:" << endl;
            accuracy(IN_IMG, NOISE_GONE);

            // inv filter
            Image INV = invFilter(NOISE_GONE, mask, false, Image(1,1), cutoff, false);
            strcat(outputName, "invFilter");
            strcat(outputName, str);
            strcat(outputName, ".pgm");
            writeImage(INV, outputName);
            outputName[NAME_SIZE - 4] = '\0';

            cout << "inv filter" << endl;
            accuracy(IN_IMG, INV);

            // do for each k. name = basename(s#)(k#)
            for(int kTrial = 0; kTrial < 3; ++kTrial)
            {
                char str2[15];
                const Image WIENER = wiener(NOISE_GONE, mask, false, Image(1,1), k[kTrial]);
                sprintf(str2,"%f", k[kTrial]);
                strcat(outputName, "wiener");
                strcat(outputName, str);
                strcat(outputName, "_");
                strcat(outputName, str2);
                strcat(outputName, ".pgm");
                writeImage(WIENER, outputName, 1);
                outputName[NAME_SIZE - 4] = '\0';

                cout << "wiener" << k[kTrial] << endl;
                accuracy(IN_IMG, WIENER);
            }
            cout << endl;
        }
    }
    else if(taskNumber[0] == '6')
    {
        // define perspective points
        Image original(4,2);
        Image new1(4,2);
        // top left corner
        original(0,0) = new1(0,0) = 0;
        original(0,1) = new1(0,1) = 0;
        // top right corner
        original(1,0) = new1(1,0) = 0;
        original(1,1) = new1(1,1) = IN_IMG.getCol() - 1;
        // bottom left corner
        original(2,0) = new1(2,0) = IN_IMG.getRow() - 1;
        original(2,1) = new1(2,1) = 0;
        // bottom right corner
        original(3,0) = IN_IMG.getRow() - 1;
        original(3,1) = IN_IMG.getCol() - 1;
        new1(3,0) = 2*(IN_IMG.getRow() - 1);
        new1(3,1) = 2*(IN_IMG.getCol() - 1);

        // sequential
        const Image SCALED = scaling(IN_IMG, .5, .5);
        writeImage(SCALED, "1shrunk.pgm");
        cout << scalingMatrix(.5, .5) << endl << endl;

        const Image SHIFTED = translation(SCALED, IN_IMG.getRow()/4, IN_IMG.getCol()/4);
        writeImage(SHIFTED, "2shifted.pgm");
        cout << translationMatrix(IN_IMG.getRow()/4, IN_IMG.getCol()/4) << endl << endl;

        const Image SHEARED = shear(SHIFTED, 0, 2);
        writeImage(SHEARED, "3sheared.pgm");
        cout << shearMatrix(0, 2) << endl << endl;

        const Image ROTATED = rotation(SHEARED, 30.0f/180.0f*PI);
        writeImage(ROTATED, "4rotated.pgm");
        cout << rotationMatrix(30.0f/180.0f*PI) << endl << endl;

        const Image PERSPECTIVED = perspective(ROTATED, original, new1);
        writeImage(PERSPECTIVED, "5perspective.pgm");
        cout << perspectiveMatrix(original, new1) << endl << endl;

        // form compositve matrix
        Image composite = perspectiveMatrix(original, new1)->* rotationMatrix(30.0f/180.0f*PI) ->* shearMatrix(0,
2)
            ->* translationMatrix(IN_IMG.getRow()/4, IN_IMG.getCol()/4) ->* scalingMatrix(.5, .5);

        cout << composite << endl;
        const Image ALL = transform(IN_IMG, composite);
        writeImage(ALL, "6all.pgm");
    }
    else if(taskNumber[0] == '7')
    {
        char str[10];
        int maskSize;

        cout << "Enter adaptive local noise filter mask size: ";
        cin >> maskSize;

        // create noise imag
        const Image SAP25 = sapNoise(IN_IMG, .25f);
        const Image SAP10 = sapNoise(IN_IMG, .10f);
```

```
        //output noise statistics
        cout << ".25 " << endl;
        accuracy(IN_IMG, SAP25);
        cout << ".10" << endl;
        accuracy(IN_IMG, SAP10);

        // write noise images
        sprintf(str,"%f", .25f);
        strcat(outputName, "SAPnoise");
        strcat(outputName, str);
        strcat(outputName, ".pgm");
        writeImage(SAP25, outputName);
        outputName[NAME_SIZE - 4] = '\0';
        sprintf(str,"%f", .10f);
        strcat(outputName, "SAPnoise");
        strcat(outputName, str);
        strcat(outputName, ".pgm");
        writeImage(SAP10, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        // do local noise reduction
        const Image LOCAL_NOISE_REDUCTION25 = adaptiveLocalNoiseFilter(SAP25, maskSize, sapVar(SAP25, true));
        const Image LOCAL_NOISE_REDUCTION10 = adaptiveLocalNoiseFilter(SAP10, maskSize, sapVar(SAP10, true));

        // do amedian (smax = 15 because larger is almost always better since it automatically uses smallest
possible)
        const Image AMEDIAN25 = amedian(SAP25, 3, 15);
        const Image AMEDIAN10 = amedian(SAP10, 3, 15);

        //write images
        sprintf(str,"%f", .25f);
        strcat(outputName, "amedian");
        strcat(outputName, str);
        strcat(outputName, ".pgm");
        writeImage(AMEDIAN25, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        sprintf(str,"%f", .10f);
        strcat(outputName, "amedian");
        strcat(outputName, str);
        strcat(outputName, ".pgm");
        writeImage(AMEDIAN10, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        sprintf(str,"%f", .25f);
        strcat(outputName, "localNoise");
        strcat(outputName, str);
        strcat(outputName, ".pgm");
        writeImage(LOCAL_NOISE_REDUCTION25, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        sprintf(str,"%f", .10f);
        strcat(outputName, "localNoise");
        strcat(outputName, str);
        strcat(outputName, ".pgm");
        writeImage(LOCAL_NOISE_REDUCTION10, outputName);

        // output performance
        cout << ".25 => amedian then local noise reduction. Repeat for .10" << endl;
        accuracy(IN_IMG, AMEDIAN25);
        accuracy(IN_IMG, LOCAL_NOISE_REDUCTION25);
        accuracy(IN_IMG, AMEDIAN10);
        accuracy(IN_IMG, LOCAL_NOISE_REDUCTION10);
    }
    else if(taskNumber[0] == '8')
    {
        // for 2nd degree, need 9 eqns
        Image xy(20, 2);
        Image uv(20, 2);

        // pick points by putting it in source code
        const int HORZ = 116;
        xy(0, 0) = 116; xy(0,1) = 370; uv(0, 0) = HORZ; uv(0,1) = 370;
        xy(1, 0) = 109; xy(1,1) = 0; uv(1, 0) = HORZ; uv(1,1) = 0;
        xy(2, 0) = 57; xy(2,1) = 823; uv(2, 0) = HORZ; uv(2,1) = 823;
        xy(3, 0) = 82; xy(3,1) = 641; uv(3, 0) = HORZ; uv(3,1) = 641;
        xy(4, 0) = 124; xy(4,1) = 175; uv(4, 0) = HORZ; uv(4,1) = 175;
        xy(5, 0) = 100; xy(5,1) = 507; uv(5, 0) = HORZ; uv(5,1) = 507;
        xy(6, 0) = 121; xy(6,1) = 285; uv(6, 0) = HORZ; uv(6,1) = 285;
        xy(7, 0) = 69; xy(7,1) = 731; uv(7, 0) = HORZ; uv(7,1) = 731;
        xy(8, 0) = 108; xy(8,1) = 445; uv(8, 0) = HORZ; uv(8,1) = 445;
        xy(13, 0) = 116; xy(13,1) = 92; uv(13, 0) = HORZ; uv(13,1) = 92;
        xy(14, 0) = 121; xy(14,1) = 325; uv(14, 0) = HORZ; uv(14,1) = 325;

        // other points
        xy(15, 0) = 50; xy(15,1) = 90; uv(15, 0) = 50; uv(15,1) = 90;
        xy(16, 0) = 54; xy(16,1) = 5; uv(16, 0) = 54; uv(16,1) = 5;
        xy(17, 0) = 218; xy(17,1) = 79; uv(17, 0) = 218; uv(17,1) = 79;

        // more
        xy(18, 0) = 200; xy(18,1) = 620; uv(18, 0) = 200; uv(18,1) = 620;
        xy(19, 0) = 210; xy(19,1) = 360; uv(19, 0) = 210; uv(19,1) = 360;

        //corners
        xy(9, 0) = 0; xy(9,1) = 0; uv(9, 0) = 0; uv(9,1) = 0;
        xy(10, 0) = IN_IMG.getRow()-1; xy(10,1) = 0; uv(10, 0) = IN_IMG.getRow()-1; uv(10,1) = 0;
        xy(11, 0) = 0; xy(11,1) = IN_IMG.getCol() -1; uv(11, 0) = 0; uv(11,1) = IN_IMG.getCol() -1;
        xy(12, 0) = IN_IMG.getRow()-1; xy(12,1) = IN_IMG.getCol() -1; uv(12, 0) = IN_IMG.getRow()-1; uv(12,1) =
IN_IMG.getCol() -1;

        const Image CORRECTED = polynomial(IN_IMG, 2, uv, xy);

        strcat(outputName, ".pgm");
        writeImage(CORRECTED, outputName);
    }
    else
        cout << "main::error in selection code" << endl;

    // success!
    return 0;
}
```

C. addNose.cpp

```
/*************************************************************
 * addNoise.cpp: has Gaussian Noise generator
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 09/5/11
 *
 *************************************************************/

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

/*
 * generates a random gauss number with mean = 0 and standard D = SD
 * @param1 standard deviation of noise
 * @return gaussian random variable
 */
float gaussianNoise(const float& SD)
{
    float a;
    do{
        a = SD*std::sqrt(-
2*log(rand()/float(RAND_MAX))*cos(2*PI*rand()/float(RAND_MAX)));
    }while(a - a);

    return a;
}

/*
 * Adds noise from GaussNoise to an image.
 * @param IN_IMG: input image
 * @param sd: standard deviation of noise
 * @return gaussian-noise corrupted image
 */
Image addGaussNoise(const Image& IN_IMG, const float& SD)
{
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());

    for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                outImg(row, col, chan) = IN_IMG(row, col, chan) + gaussianNoise(SD);

    return outImg;
}

/*
 * s&p by qi and slightly modified by ali
 */
Image sapNoise(const Image& IN_IMG, const float& q)
{
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());

    std::srand(time(0));
    for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
            {
                float r = std::rand()/(RAND_MAX + 1.0f);
                if(r < q)
                    outImg(row, col, chan) = 0;
                else if(r > 1 - q)
                    outImg(row, col, chan) = L;
                else
                    outImg(row, col, chan) = IN_IMG(row, col, chan);
            }

    return outImg;
}

/*
 * Estimates the variance of an image corrupted by SAP
 * @param1 image corrupted by sap
 * @param2 showvar boolean
 * @return estimated variance
 */
float sapVar(const Image& IN_IMG, const bool& SHOW_VAR)
{
    float mean = 0,
        qCount = 0,
        q,
        var;

    for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                if(IN_IMG(row, col, chan) == L || !IN_IMG(row,col, chan))
                    ++qCount;
                else
                    mean += IN_IMG(row, col, chan);

    mean /= IN_IMG.getRow()*IN_IMG.getCol() - qCount;
    q = qCount/(IN_IMG.getRow()*IN_IMG.getCol())/2; // divide by 2 because
p[salt]=p[pepper] = q

    // compute estimate of variance from equation (7) in report
    var = q*((L-mean)*(L-mean) + mean*mean) - std::pow(q*(L-2.0f*mean),2.0f);

    if(SHOW_VAR)
        std::cout << var << std::endl;

    return var;
```

APPROVED

D. *deblurr.cpp*

```
/***************************************************************
 * deblurr.cpp: deblurrs
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 10/12/11
 *
 ***************************************************************/
#include <complex>
#include "Dip.h"


/*
 * Deblurrs using a known blurring function (spatial or freq)
 * @param1 input image to deblurr
 * @param2 mask responsible for blurring (spatial or frequency)
 * @param3 whether arg2 is freq or space
 * @param4 phase of freq mask (use img(1,1) if you don't want to include phase in freq
filter)
 * @param5 radius in which to inverse transform
 * @param6 whether to simply ignore freq outside param5 or use input image's freq
content there
 * @return deblurred image
 */
Image invFilter(const Image& IN_IMG, const Image& MASK, const bool& isFreq, const
Image& PHASE, const float& CUTOFF, const bool& IGNORE)
{
    Image hMag;
    Image hPhase;
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());

    // if spatial, find freq
    if(!isFreq)
    {
        Image padMask = padPow2(MASK, IN_IMG);

        hMag = Image(padMask.getRow(), padMask.getCol(), padMask.getChannel());
        hPhase = Image(padMask.getRow(), padMask.getCol(), padMask.getChannel());

        fft(padMask, hMag, hPhase);
    }
    else // else copy freq inputs as mask/phase for H
    {
        if(PHASE.getRow() != 1) // if phase provided, use it
            hPhase = PHASE;
        else // ignore it if not, use phases of zero for no effect
            hPhase = Image(MASK.getRow(), MASK.getCol(), MASK.getChannel());
        hMag = MASK;
    }

    // create fft of input image
    Image padImg = padPow2(IN_IMG);

    // fft the image
    Image inMag(padImg.getRow(), padImg.getCol(), padImg.getChannel());
    Image inPhase(padImg.getRow(), padImg.getCol(), padImg.getChannel());
    fft(padImg, inMag, inPhase);

    // define padded output mag and phase
    Image outMag(hMag.getRow(), hMag.getCol(), hMag.getChannel());
    Image outPhase(hMag.getRow(), hMag.getCol(), hMag.getChannel());

    // inverse filter.
    const int ROW_SHIFT = (hMag.getRow() - 1)/2;
    const int COL_SHIFT = (hMag.getCol() - 1)/2;
    for(int chan = 0; chan < outMag.getChannel(); ++chan)
        for(int row = 0; row < outMag.getRow(); ++row)
            for(int col = 0; col < outMag.getCol(); ++col)
            {
                const float DIST = std::abs(std::complex<float>(row - ROW_SHIFT, col -
COL_SHIFT));
                if(hMag(row, col, chan) && DIST < CUTOFF)
                {
                    outMag(row, col, chan) = inMag(row, col, chan)/hMag(row, col, chan);
                    outPhase(row, col, chan) = inPhase(row, col, chan) - hPhase(row, col,
chan);
                }
                else if(!IGNORE)
                {
                    outMag(row, col, chan) = inMag(row, col, chan);
                    outPhase(row, col, chan) = inPhase(row, col, chan);
                }
            }

    // find ifft of padded output
    ifft(padImg, outMag, outPhase);

    for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                outImg(row, col, chan) = padImg(row, col, chan);

    return outImg;
}

Image wiener(const Image& IN_IMG, const Image& MASK, const bool& isFreq, const Image&
PHASE, const float K)
{
    Image hMag;
    Image hPhase;
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());

    // if spatial, find freq
    if(!isFreq)
    {
        Image padMask = padPow2(MASK, IN_IMG);

        hMag = Image(padMask.getRow(), padMask.getCol(), padMask.getChannel());
        hPhase = Image(padMask.getRow(), padMask.getCol(), padMask.getChannel());

        fft(padMask, hMag, hPhase);
    }
    else // else copy freq inputs as mask/phase for H
    {
        if(PHASE.getRow() != 1) // if phase provided, use it
            hPhase = PHASE;
        else // ignore it if not, use phases of zero for no effect
            hPhase = Image(MASK.getRow(), MASK.getCol(), MASK.getChannel());
        hMag = MASK;
    }

    // create fft of input image
    Image padImg = padPow2(IN_IMG);

    // fft the image
    Image inMag(padImg.getRow(), padImg.getCol(), padImg.getChannel());
    Image inPhase(padImg.getRow(), padImg.getCol(), padImg.getChannel());
    fft(padImg, inMag, inPhase);

    // define padded output mag and phase
    Image outMag(hMag.getRow(), hMag.getCol(), hMag.getChannel());
    Image outPhase(hMag.getRow(), hMag.getCol(), hMag.getChannel());

    // calculate power spectrum
    Image hPower = hMag*hMag;

    // wiener filter
    for(int chan = 0; chan < outMag.getChannel(); ++chan)
        for(int row = 0; row < outMag.getRow(); ++row)
            for(int col = 0; col < outMag.getCol(); ++col)
            {
                // use definition with h* in top to avoid division by zero + cut down on
operations
                outMag(row, col, chan) = inMag(row, col, chan)*hMag(row, col,
chan)/(hPower(row, col, chan) + K);
                // g times h* => mult the mags and subtract phase of h
                outPhase(row, col, chan) = inPhase(row, col, chan) - hPhase(row, col,
chan);
            }

    // find ifft of padded output
    ifft(padImg, outMag, outPhase);

    for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                outImg(row, col, chan) = padImg(row, col, chan);

    return outImg;
}

/*
 * outputs MSE and pSNR between a perfect original and a fixed image from recovery
 * @param1 perfect image
 * @param2 image recovered from corrupted param1
 * @return mse and snr
 */
void accuracy(const Image& ORIGINAL, const Image& FIXED)
{
    double mse = 0.0f;
    double psnr;

    for(int chan = 0; chan < ORIGINAL.getChannel(); ++chan)
        for(int row = 0; row < ORIGINAL.getRow(); ++row)
            for(int col = 0; col < ORIGINAL.getCol(); ++col)
                mse += std::pow(ORIGINAL(row, col, chan) - FIXED(row, col, chan), 2.0f);

    mse /= (ORIGINAL.getRow()*ORIGINAL.getCol());
    psnr = 10*std::log10(double(L)/std::sqrt(mse));

    std::cout << "MSE: " << mse << std::endl << "PSNR: " << psnr << " dB" << std::endl;
}
```

E. *freqFilter.cpp*

```
/****************************************************************
 * freqFilter.cpp: has frequency filters in it
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 09/24/11
 *          09/25/11 added freqSpatialMaskFilter
 *
 ****************************************************************/

#include <complex>
#include "Dip.h"


/*
 * convolves IN_IMG with MASK in frequency domain using lp butterworth
 * @param1 image to convolve mask with
 * @param2 cutoff freq
 * @param3 order
 * @return convolved image
 */
Image lpButterworth(const Image& IN_IMG, const float& D_0, const int& N)
{
    // generate filter
    const int ROW = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
    const int COL = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    Image mask(ROW, COL, IN_IMG.getChannel());
    const float ROW_SHIFT = (mask.getRow()-1)/2;
    const float COL_SHIFT = (mask.getCol()-1)/2;
    for(int row = 0; row < mask.getRow(); ++row)
        for(int col = 0; col < mask.getCol(); ++col)
            mask(row, col) = 1/(1 + std::pow(std::abs(std::complex<float>(row -
ROW_SHIFT, col - COL_SHIFT))/D_0, 2*N));

    // filter
    std::cout << D_0 << " and " << N <<  ": ";

    return filter(IN_IMG, mask, false);
}

/*
 * convolves IN_IMG with MASK in frequency domain using lp gaussian
 * @param1 image to convolve mask with
 * @param2 cutoff freq
 * @return convolved image
 */
Image lpGaussian(const Image& IN_IMG, const float& D_0)
{
    // generate filter
    const int ROW = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
    const int COL = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    Image mask(ROW, COL, IN_IMG.getChannel());
    const float ROW_SHIFT = (mask.getRow()-1)/2;
    const float COL_SHIFT = (mask.getCol()-1)/2;
    for(int row = 0; row < mask.getRow(); ++row)
        for(int col = 0; col < mask.getCol(); ++col)
            mask(row, col) = std::exp(-std::pow(std::abs(std::complex<float>(row -
ROW_SHIFT, col - COL_SHIFT))/D_0, 2)/2);

    // filter
    std::cout << D_0 << ": ";

    return filter(IN_IMG, mask, false);
}

/*
 * convolves IN_IMG with MASK in frequency domain using hp gaussian
 * @param1 image to convolve mask with
 * @param2 cutoff freq
 * @return convolved image
 */
Image hpGaussian(const Image& IN_IMG, const float& D_0)
{
    // generate filter
    const int ROW = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
    const int COL = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    Image mask(ROW, COL, IN_IMG.getChannel());
    const float ROW_SHIFT = (mask.getRow()-1)/2;
    const float COL_SHIFT = (mask.getCol()-1)/2;
    for(int row = 0; row < mask.getRow(); ++row)
        for(int col = 0; col < mask.getCol(); ++col)
            mask(row, col) = 1 - std::exp(-std::pow(std::abs(std::complex<float>(row -
ROW_SHIFT, col - COL_SHIFT))/D_0, 2)/2);

    // filter
    std::cout << D_0 << ": ";

    return filter(IN_IMG, mask, false);
}

/*
 * convolves IN_IMG with MASK in frequency domain using lp butterworth
 * @param1 image to convolve mask with
 * @param2 cutoff freq
 * @param3 order
 * @return convolved image
 */
Image hpButterworth(const Image& IN_IMG, const float& D_0, const int& N)
{
    // generate filter
    const int ROW = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
    const int COL = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    Image mask(ROW, COL, IN_IMG.getChannel());
    const float ROW_SHIFT = (mask.getRow()-1)/2;
    const float COL_SHIFT = (mask.getCol()-1)/2;
    for(int row = 0; row < mask.getRow(); ++row)
        for(int col = 0; col < mask.getCol(); ++col)
            mask(row, col) = 1/(1 + std::pow(D_0/std::abs(std::complex<float>(row -
ROW_SHIFT, col - COL_SHIFT)), 2*N));

    // filter
    std::cout << D_0 << " and " << N <<  ": ";

    return filter(IN_IMG, mask, false);
}

/*
 * computes the 'convolution' and outputs power ratio
 * @param1 regular-sized image to 'convolve' with
 * @param2 padded mask in frequency domain (power of 2 >= 2*row and 2*col)
 * @param3 whether to suppress power ratio calculation
 * @param4 the phase of the mask (Image(1,1) means ignore phase of mask)
 * @param5 whether the operation is convolution or correlation
 * @return filtered image, regular size
 */
Image filter(const Image& IN_IMG, const Image& PAD_MASK, const bool& SUPPRESS, const
Image& MASK_PHASE, const bool& isCONV, const int& EXTRACT_OFFSET)
{
    float PT = 0, P = 0;

    Image padImg = padPow2(IN_IMG);

    // fft the image
    Image mag(padImg.getRow(), padImg.getCol(), padImg.getChannel());
    Image phase(padImg.getRow(), padImg.getCol(), padImg.getChannel());
    fft(padImg, mag, phase);

    // 'convolve'
    for(int row = 0; row < PAD_MASK.getRow(); ++row)
        for(int col = 0; col < PAD_MASK.getCol(); ++col)
        {
            PT += mag(row,col)*mag(row,col);
            padImg(row, col) = mag(row, col)*PAD_MASK(row, col); // padImg represents
output img freq
            P += padImg(row,col)*padImg(row,col);
        }

    // inverse transform
    Image padOutImg(padImg.getRow(), padImg.getCol(), padImg.getChannel());
    if(MASK_PHASE.getRow() == 1) // meaning no mask phase given (for butterworht etc.)
        ifft(padOutImg, padImg, phase);
    else if(isCONV)
    {
        Image outPhase = phase + MASK_PHASE;
        ifft(padOutImg, padImg, outPhase);
    }
    else
    {
        Image outPhase = phase - MASK_PHASE;
        ifft(padOutImg, padImg, outPhase);
    }

    // truncate results
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());
    const int END_ROW = EXTRACT_OFFSET + IN_IMG.getRow(); // to get 'same' convolution
    const int END_COL = EXTRACT_OFFSET + IN_IMG.getCol();  // to get 'same' convolution
    for(int row = EXTRACT_OFFSET; row < END_ROW; ++row)
        for(int col = EXTRACT_OFFSET; col < END_COL; ++col)
            outImg(row - EXTRACT_OFFSET, col - EXTRACT_OFFSET) = padOutImg(row, col);

    // output power ratio
    if(!SUPPRESS)
        std::cout << 100*P/PT << std::endl;

    return outImg;
}

/*
 * filters image in freq domain based on spatial mask
 * @param1 image (regular size)
 * @param2 mask (spatial, regular size, must be square)
 */
Image freqSpatialMaskFilter(const Image& IN_IMG, const Image& MASK, const bool& isCONV)
{
    // create frequency version of mask then pass to filter()
    Image padMask = padPow2(MASK, IN_IMG);

    Image magMask(padMask.getRow(), padMask.getCol(), padMask.getChannel());
    Image phaseMask(padMask.getRow(), padMask.getCol(), padMask.getChannel());

    fft(padMask, magMask, phaseMask);

    return filter(IN_IMG, magMask, true, phaseMask, isCONV, int(MASK.getRow()/2));
}

/*
 * Resizes an image to ROW by COL
 * @param1 input image to resize 2 fold if a power of 2 else the next highest power of
2
 * @param2 arg1 will resize to 2*arg1 (or closest pwr of 2) Image(1,1) means no goal
 */
Image padPow2(const Image& IN_IMG, const Image& GOAL)
{
    int padRow, padCol;
    if(GOAL.getRow() == 1)
    {
        padRow = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
        padCol = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    }
    else
    {
        padRow = std::pow(2,
float(std::ceil(std::log(float(GOAL.getRow()*2))/std::log(float(2)))));
        padCol = std::pow(2,
float(std::ceil(std::log(float(GOAL.getCol()*2))/std::log(float(2)))));
```

```
    }

    Image padImg(padRow, padCol, IN_IMG.getChannel());

    for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                padImg(row, col) = IN_IMG(row, col);

    return padImg;
}

Image homomorphic(const Image& IN_IMG, const float& HI, const float& LO, const float&
D0)
{
    // generate filter
    const int ROW = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
    const int COL = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    Image mask(ROW, COL, IN_IMG.getChannel());
    const float ROW_SHIFT = (ROW-1)/2;
    const float COL_SHIFT = (COL-1)/2;
    for(int row = 0; row < ROW; ++row)
        for(int col = 0; col < COL; ++col)
            mask(row, col) = (HI-LO)*(1 - std::exp(-
std::pow(std::abs(std::complex<float>(row - ROW_SHIFT, col - COL_SHIFT))/D0, 2)))+ LO;

    // filter
    return filter(IN_IMG, mask);
}

Image notch(const Image& IN_IMG, const float& D_ROW, const float& D_COL, const bool&
isREJECT)
{
    // generate filter
    const int ROW = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
    const int COL = std::pow(2,
float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    Image mask(ROW, COL, IN_IMG.getChannel());
    const float ROW_SHIFT = ROW/2;
    const float COL_SHIFT = COL/2;

    float band = 1.0f;

    if(isREJECT)
    {
        mask.initImage(1);
        band = 0.0f;
    }

    for(int row = 0; row < ROW; ++row)
        for(int col = 0; col < COL; ++col)
            if(std::abs(row - ROW_SHIFT) > D_ROW && std::abs(col - COL_SHIFT) <= D_COL)
                mask(row, col) = band;

    if(isREJECT)
        writeImage(mask*255, "NOTCH_REJECTION.pgm");
    else
        writeImage(mask*255, "NOTCH_PASS.pgm");

    return filter(IN_IMG, mask);
}
```

F. geometric.cpp

```
/**************************************************************
 * geometric.cpp: geometric correction
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 10/16/11
 *
 **************************************************************/

#include "Dip.h"

/*
 * geometric correction via polynomial
 * @param1 input image to correct
 * @param2 order of polynomial
 * @param3 new tie points (u,v) each row is a new one
 * @param4 old tie points (x,y) each row is a new one
 * @return geometrically corrected image
 */
Image polynomial(const Image& IN_IMG, const int& ORDER, const Image& TIE_NEW, const
Image& TIE_OLD)
{
    const int MIN_EQUATIONS = (ORDER + 1)*(ORDER + 1);
    const int EQUATIONS = TIE_NEW.getRow();

    if(TIE_NEW.getRow() < MIN_EQUATIONS || TIE_OLD.getRow() < MIN_EQUATIONS)
    {
        std::cout << "polynomial::not enough points" << std::endl;
        return IN_IMG;
    }
    else if(TIE_NEW.getRow() !=TIE_OLD.getRow())
    {
        std::cout << "polynomial::need same number of points" << std::endl;
        return IN_IMG;
    }

    // construct W
    Image W(EQUATIONS, MIN_EQUATIONS);

    for(int eqn = 0; eqn < EQUATIONS; ++eqn)
        for(int u = 0; u <= ORDER; ++u)
            for(int v = 0; v <= ORDER; ++v)
                W(eqn, u*(ORDER + 1) + v) = pow(TIE_NEW(eqn, 0), u)*pow(TIE_NEW(eqn,
1), v);

    // construct X and Y
    Image X(EQUATIONS, 1);
    Image Y(EQUATIONS, 1);
    for(int eqn = 0; eqn < EQUATIONS; ++eqn)
    {
        X(eqn,0) = TIE_OLD(eqn, 0);
        Y(eqn, 0) = TIE_OLD(eqn, 1);
    }

    // compute A and B
    const Image INVERSE = pinv(W);
     Image A = INVERSE ->* X;
    const Image B = INVERSE ->* Y;

    // map u,v to x,y
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());
    for(int row = 0; row < IN_IMG.getRow(); ++row)
        for(int col = 0; col < IN_IMG.getCol(); ++col)
        {
            float x = 0;
            float y = 0;

            for(int u = 0; u <= ORDER; ++u)
                for(int v = 0; v <= ORDER; ++v)
                {
                    const int PLACE = u*(ORDER + 1) + v;
                    x += A(PLACE, 0)*pow(float(row), u)*pow(float(col), v);
                    y += B(PLACE, 0)*pow(float(row), u)*pow(float(col), v);
                }

            if(x >= 0 && y >= 0 && x < IN_IMG.getRow() && y < IN_IMG.getCol())
                for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
                    outImg(row, col, chan) = IN_IMG(x, y, chan);
        }

        return outImg;
}
```

G. spatialFilter.cpp

```
/**************************************************************
 * spatialFilter.cpp: has spatial filters in it
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 09/24/11
 *          09/25/11 added sobel/laplacian/unsharp masking/average filter
 *
 **************************************************************/

#include "Dip.h"
#include <cmath>
#include <algorithm>

/*
 * adpative local noise reduction
 * @param1 corrupted input image
 * @param2 neighborhood size
 * @param3 variance of noise
 * @return repaired image
 */
Image adaptiveLocalNoiseFilter(const Image& IN_IMG, const int& MASK_SIZE, const float&
VAR)
{
    const Image AVG = amean(IN_IMG, MASK_SIZE);
    const Image SQUARE = powImage(IN_IMG, 2);
    const Image SQUARE_AVERAGED = amean(SQUARE, MASK_SIZE);
    const Image AVG_SQUARED = powImage(AVG, 2);
    const Image IMG_VAR = SQUARE_AVERAGED - AVG_SQUARED;

    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());

    for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                if(IMG_VAR(row, col, chan))
                    outImg(row, col, chan) = IN_IMG(row, col, chan) - VAR/IMG_VAR(row,
col, chan)*(IN_IMG(row, col, chan) - AVG(row, col, chan));
                else
                    outImg(row, col, chan) = AVG(row, col, chan);

    return outImg;
}

/*
 * filters image with gaussian average filter
 * @param1 image to filter
 * @param2 size of gaussian mask used
 * @param3 standard deviation in the generation of the mask
 * @return filtered image
 */
Image lpGaussian(const Image& IN_IMG, const int& MASK_SIZE, const float& SD)
{
    Image mask(MASK_SIZE, MASK_SIZE);
    const float SHIFT = (MASK_SIZE-1)/2;

    const float SDSD2 = 2*SD*SD;
    for(int row = 0; row < mask.getRow(); ++row)
        for(int col = 0; col < mask.getCol(); ++col)
            mask(row, col) = std::exp(-(std::pow(row-SHIFT,2)+std::pow(col-
SHIFT,2))/SDSD2)/PI/SDSD2;

    std::cout << mask << std::endl;

    return conv(IN_IMG, mask, true);
}

/*
 * Applies median filtering to input image.
 * @param1 input image to filter
 * @param2 square neighborhood size to use during filtering
 */
Image median(const Image& IN_IMG, const int& MASK_SIZE)
{
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());

    for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                outImg(row, col, chan) = median(IN_IMG, MASK_SIZE, row, col, chan);

    return outImg;
}

/*
 * Returns the median of a neighborhood defined about a point in an image
 * @param1 image neighborhood resides in
 * @param2 square neighborhood size (odd)
 * @param3 & @param4 & parm5 row, col, chan to operate in/around
 */
float median(const Image& IN_IMG, const int& MASK_SIZE, const int& ROW, const int& COL,
const int& CHAN)
{
    int oddSize;
    if(MASK_SIZE%2)
        oddSize = MASK_SIZE;
    else
        oddSize = MASK_SIZE + 1;

    const int SHIFT =  (oddSize - 1)/2;

    const int ROW_END = SHIFT + ROW < IN_IMG.getRow() ? SHIFT + ROW + 1 :
IN_IMG.getRow();
    const int COL_END = SHIFT + COL < IN_IMG.getCol() ? SHIFT + COL + 1 :
IN_IMG.getCol();
    int rowStart;
    int colStart;
    if(MASK_SIZE%2)
    {
        rowStart = ROW - SHIFT > -1 ? ROW - SHIFT : 0;
        colStart = COL - SHIFT > -1 ? COL - SHIFT : 0;
    }
    else
    {
        rowStart = ROW - SHIFT + 1 > -1 ? ROW - SHIFT + 1 : 0;
        colStart = COL - SHIFT + 1 > -1 ? COL - SHIFT + 1 : 0;
    }

    const int VOTERS = (ROW_END - rowStart)*(COL_END - colStart);

    float* voters = new float[VOTERS];

    int voteNumber = 0;
    for(int row = rowStart; row < ROW_END; ++row)
        for(int col = colStart; col < COL_END; ++col)
            voters[voteNumber++] = IN_IMG(row, col, CHAN);

    std::sort(voters, voters + VOTERS);

    float pixelValue = VOTERS%2 ? voters[VOTERS/2] : (voters[VOTERS/2] +
voters[VOTERS/2 - 1])/2;

    delete [] voters;

    return pixelValue;
}

/*
 * Filters image with sobel mask
 * @param1 input image
 * @param2 whether to approximate (default false)
 * @return filtered image
 */
Image sobel(const Image& IN_IMG, const bool& APPROX)
{
    Image outImgX,
        outImg; // stores outImg and outImgy

    Image sobelxOry(3,3);

    // x first
    for(int row = 0; row < 3; ++row)
        for(int col = 0; col < 3; ++col)
            sobelxOry(row, col) = -1 + col + (col - 1)*(row%2);

    outImgX = conv(IN_IMG, sobelxOry, false);

    // now y
    for(int row = 0; row < 3; ++row)
        for(int col = 0; col < 3; ++col)
            sobelxOry(row, col) = -1 + row + (row - 1)*(col%2);

    outImg = conv(IN_IMG, sobelxOry, false);

    if(APPROX)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                outImg(row, col) = std::abs(outImg(row, col)) + std::abs(outImgX(row,
col));
    else
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                outImg(row, col) = std::sqrt(outImg(row, col)*outImg(row, col) +
outImgX(row, col)*outImgX(row, col));

    return outImg;
}

/*
 * Performs unsharp masking filtering
 * @param1 input image to filter
 * @param2 mask size of filter
 */
Image um(const Image& IN_IMG, const int& MASK_SIZE)
{
    return IN_IMG + (IN_IMG - amean(IN_IMG, MASK_SIZE));
}

/*
 * Performs Laplacian filtering
 * @param1 input image to filter
 * @return filtered image
 */
Image laplacian(const Image& IN_IMG)
{
    Image mask(3,3);
    for(int row = 0; row < 3; ++row)
        for(int col = 0; col < 3; ++col)
            mask(row,col) = ((row%2)|(col%2)) - 5*(row%2)*(col%2);

    return conv(IN_IMG, mask, false);
}

/*
 * filters image with standard average filter
 * @param1 image to filter
 * @param2 size of mask to use
 * @return filtered image
 */
Image amean(const Image& IN_IMG, const int& MASK_SIZE)
{
    Image mask(MASK_SIZE, MASK_SIZE);
    mask.initImage(1);

    return conv(IN_IMG, mask, true);
}

/*
 * Performs geometric averaging.
 * @param1 image to filter
 * @param2 size of mask to use
 * @return filtered image
 */
Image gmean(const Image& IN_IMG, const int& MASK_SIZE)
```

```
{
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());

    // initialize constant variables in the neighborhood computations
    int oddSize;
    if(MASK_SIZE%2)
        oddSize = MASK_SIZE;
    else
        oddSize = MASK_SIZE + 1;
    const int SHIFT =  (oddSize - 1)/2;
    const float ROOT = 1.0f/(MASK_SIZE*MASK_SIZE);

    // find product of neighborhoods.
    for(int row = 0; row < IN_IMG.getRow(); ++row)
        for(int col = 0; col < IN_IMG.getCol(); ++col)
        {
            const int ROW_END = SHIFT + row < IN_IMG.getRow() ? SHIFT + row + 1 :
IN_IMG.getRow();
            const int COL_END = SHIFT + col < IN_IMG.getCol() ? SHIFT + col + 1 :
IN_IMG.getCol();
            int rowStart;
            int colStart;
            if(MASK_SIZE%2)
            {
                rowStart = row - SHIFT > -1 ? row - SHIFT : 0;
                colStart = col - SHIFT > -1 ? col - SHIFT : 0;
            }
            else
            {
                rowStart = row - SHIFT + 1 > -1 ? row - SHIFT + 1 : 0;
                colStart = col - SHIFT + 1 > -1 ? col - SHIFT + 1 : 0;
            }

            bool isZero = false;

            for(int neigh_row = rowStart; neigh_row < ROW_END; ++neigh_row)
                for(int neigh_col = colStart; neigh_col < COL_END; ++neigh_col)
                {
                    if(IN_IMG(neigh_row, neigh_col))
                        outImg(row, col) += std::log(IN_IMG(neigh_row, neigh_col));
                    else
                        isZero = true;
                }

            if(isZero)
                outImg(row, col) = 0;
            else
            {
                outImg(row, col) = outImg(row, col)*ROOT;
                outImg(row, col) = std::exp(outImg(row, col));
            }
        }


    // take the root of image
    return outImg;
}

/*
 * Performs geometric averaging.
 * @param1 image to filter
 * @param2 Q in sum( g^(Q+1) )/sum(g^q)
 * @return filtered image
 */
Image contrah(const Image& IN_IMG, const float& Q, const int& MASK_SIZE)
{
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());

    // Find power image once to reduce redundant calculation of powers
    const Image IMG_Q = powImage(IN_IMG, Q);
    const Image IMG_Q_PLUS_1 = powImage(IN_IMG, Q + 1);

    Image mask(MASK_SIZE, MASK_SIZE, IN_IMG.getChannel());
    mask.initImage(1); // calculate sums in convolution
    const Image NUMERATOR = conv(IMG_Q_PLUS_1, mask, false);
    const Image DENOMINATOR = conv(IMG_Q, mask, false);

    for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
            for(int row = 0; row < IN_IMG.getRow(); ++row)
                for(int col = 0; col < IN_IMG.getCol(); ++col)
                    if( DENOMINATOR(row, col, chan) )
                        outImg(row, col, chan) = NUMERATOR(row, col,
chan)/DENOMINATOR(row, col, chan);
                    else
                        outImg(row, col, chan) = L; // i.e. division by zero => make
white

    return outImg;
}

/*
 * Returns image raised to power
 * @param1 input image
 * @param2 power to raise to
 * @return powered image
 */
Image powImage(const Image& IN_IMG, const float& POWER)
{
    Image powerImage(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());

    if( POWER )
        for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
            for(int row = 0; row < IN_IMG.getRow(); ++row)
                for(int col = 0; col < IN_IMG.getCol(); ++col)
                    powerImage(row, col, chan) = std::pow(IN_IMG(row, col, chan), POWER);
    else
        powerImage.initImage(1);  // assume 0^0 = 1

    return powerImage;
}

Image amedian(const Image& IN_IMG, const int& FIRST_MASK, const int& LAST_MASK)
{
    // do insult-filled error checking
    if(FIRST_MASK > LAST_MASK)
    {
        std::cout << "amedian::you're an idiot" << std::endl;
        return IN_IMG;
    }

    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());

        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                outImg(row, col) = amedianNeigh(IN_IMG, FIRST_MASK, LAST_MASK, row,
col);

    return outImg;
}

float amedianNeigh(const Image& IN_IMG, int maskSize, const int& LAST_MASK, const int&
ROW, const int& COL)
{
    // acquire ordered neighborhood
    int oddSize;
    if(maskSize%2)
        oddSize = maskSize;
    else
        oddSize = maskSize + 1;

    const int SHIFT =  (oddSize - 1)/2;

    const int ROW_END = SHIFT + ROW < IN_IMG.getRow() ? SHIFT + ROW + 1 :
IN_IMG.getRow();
    const int COL_END = SHIFT + COL < IN_IMG.getCol() ? SHIFT + COL + 1 :
IN_IMG.getCol();
    int rowStart;
    int colStart;
    if(maskSize%2)
    {
        rowStart = ROW - SHIFT > -1 ? ROW - SHIFT : 0;
        colStart = COL - SHIFT > -1 ? COL - SHIFT : 0;
    }
    else
    {
        rowStart = ROW - SHIFT + 1 > -1 ? ROW - SHIFT + 1 : 0;
        colStart = COL - SHIFT + 1 > -1 ? COL - SHIFT + 1 : 0;
    }

    const int VOTERS = (ROW_END - rowStart)*(COL_END - colStart);

    float* voters = new float[VOTERS];

    int voteNumber = 0;
    for(int row = rowStart; row < ROW_END; ++row)
        for(int col = colStart; col < COL_END; ++col)
            voters[voteNumber++] = IN_IMG(row, col);

    std::sort(voters, voters + VOTERS);

    float med;
    const float MAX = voters[VOTERS - 1];
    const float MIN = voters[0];
    //for even quantities, the average of the 2 nearest middle parts is problematic if
one is MIN or MAX. Thus, if
    //that is the case, set MED to MIN or MAX so that the a1/a2 test will fail. e.g.
    // a neighborhood has: 13 15 255 255 => median = (15+255)/2 = huge and visually
displeasing
    if(VOTERS%2) // odd = no problem
        med = voters[VOTERS/2];
    else if(voters[VOTERS/2] == MIN || voters[VOTERS/2] == MAX) //all below are even,
check for MIN/MAX issue
        med = voters[VOTERS/2];
    else if(voters[VOTERS/2 - 1] == MIN || voters[VOTERS/2 - 1] == MAX)
        med = voters[VOTERS/2 - 1];
    else
        med = (voters[VOTERS/2] + voters[VOTERS/2 - 1])/2;

    delete [] voters;

    if(med != MIN && med != MAX) // a1 and a2 test
        if(IN_IMG(ROW, COL) != MIN && IN_IMG(ROW, COL) != MAX) // b1 and b2 test
            return IN_IMG(ROW, COL);
        else
            return med;
    else if(++maskSize > LAST_MASK)
        return med;
    else
        return amedianNeigh(IN_IMG, maskSize, LAST_MASK, ROW, COL);
}
```

H. *transform.cpp*

```cpp
/***************************************************************
 * transform.cpp: transforms stuff
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 10/15/11
 *
 ***************************************************************/
#include <complex>
#include "Dip.h"

/*
 * Takes an image and a transform matrix and applies it to each pixel using inverse
transform
 * @param1 image to transform
 * @param2 transform matrix (3x3)
 */
Image transform(const Image& IN_IMG, const Image& TRANSFORM)
{
    const Image INVERSE = inverse(TRANSFORM);
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());
    Image newCoord(3,1);
    newCoord(2,0) = 1;
    for(int row = 0; row < IN_IMG.getRow(); ++row)
        for(int col = 0; col < IN_IMG.getCol(); ++col)
        {
            newCoord(0,0) = row;
            newCoord(1,0) = col;
            Image oldCoord = INVERSE ->* newCoord;
            oldCoord(0,0) /= oldCoord(2,0);
            oldCoord(1,0) /= oldCoord(2,0);

            if(oldCoord(0,0) >= 0 && oldCoord(1,0) >= 0 && oldCoord(0,0) <
IN_IMG.getRow() && oldCoord(1,0) < IN_IMG.getCol())
                for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
                    outImg(row, col, chan) = IN_IMG(oldCoord(0,0), oldCoord(1,0), chan);
        }

    return outImg;
}

/*
 * rotates image
 * @param1 input image to rotates
 * @param2 amount to rotate (radians)
 * @return rotated image
 */
Image rotation(const Image& IN_IMG, const float& THETA)
{
    return transform(IN_IMG, rotationMatrix(THETA));
}
/*
 * translates image
 * @param1 input image to translate
 * @param2 x translation
 * @param3 y translation
 * @return translated image
 */
Image translation(const Image& IN_IMG, const float& TRANS_X, const float& TRANS_Y)
{
    return transform(IN_IMG, translationMatrix(TRANS_X, TRANS_Y));
}
/*
 * shear image
 * @param1 input image to shear
 * @param2 x shear
 * @param3 y shear
 * @return shear image
 */
Image shear(const Image& IN_IMG, const float& SHEAR_X, const float& SHEAR_Y)
{
    return transform(IN_IMG, shearMatrix(SHEAR_X, SHEAR_Y));
}
/*
 * scale image
 * @param1 input image to scale
 * @param2 x scale
 * @param3 y scale
 * @return scale image
 */
Image scaling(const Image& IN_IMG, const float& SCALE_X, const float& SCALE_Y)
{
    return transform(IN_IMG, scalingMatrix(SCALE_X, SCALE_Y));
}
/*
 * creates rotation matrix
 * @param1 angle to rotate
 * @return rotation matrix
 */
Image rotationMatrix(const float& THETA)
{
    Image matrix = identity(3);
    matrix(0,0) = matrix(1,1) = std::cos(THETA);
    matrix(0,1) = std::sin(THETA);
    matrix(1,0) = -matrix(0,1);
    return matrix;
}
/*
 * creates translation matrix
 * @param2 x translation
 * @param3 y translation
 * @return translation matrix
 */
Image translationMatrix(const float& TRANS_X, const float& TRANS_Y)
{
    Image matrix = identity(3);
    matrix(0,2) = TRANS_X;
    matrix(1,2) = TRANS_Y;
    return matrix;
}
/*
 * creates shear matrix
 * @param2 x shear
 * @param3 y shear
 * @return shear matrix
 */
Image shearMatrix(const float& SHEAR_X, const float& SHEAR_Y)
{
    Image matrix = identity(3);
    matrix(0,1) = SHEAR_X;
    matrix(1,0) = SHEAR_Y;
    return matrix;
}
/*
 * creates scaling matrix
 * @param2 x scaling
 * @param3 y scaling
 * @return scaling matrix
 */
Image scalingMatrix(const float& SCALE_X, const float& SCALE_Y)
{
    Image matrix = identity(3);
    matrix(0,0) = SCALE_X;
    matrix(1,1) = SCALE_Y;
    return matrix;
}


/*
 * Performs perspective transform
 * @param1 input image to transform
 * @param2 4 original points (each row is an x,y set)
 * @param3 4 new points (each row is an x,y set)
 * @return transformed image
 */
Image perspective(const Image& IN_IMG, const Image& ORIGINAL, const Image& NEW)
{
    return transform(IN_IMG, perspectiveMatrix(ORIGINAL, NEW));
}

/*
 * Creates perspective matrix from 4 original/new points
 * @param1 4 original points (each row is a set of points)
 * @param2 4 new points (each row...)
 * @return matrix
 */
Image perspectiveMatrix(const Image& ORIGINAL, const Image NEW)
{
    // algorithm from: http://alumni.media.mit.edu/~cwren/interpolator/
    Image A(8,8);
    Image B(8,1);
    for(int twoRow = 0; twoRow < 8; twoRow += 2)
    {
        A(twoRow, 0) = ORIGINAL(twoRow/2, 0);
        A(twoRow, 1) = ORIGINAL(twoRow/2, 1);
        A(twoRow, 2) = 1;
        A(twoRow, 6) = -NEW(twoRow/2, 0)*ORIGINAL(twoRow/2, 0);
        A(twoRow, 7) = -NEW(twoRow/2, 0)*ORIGINAL(twoRow/2, 1);
        A(twoRow + 1, 3) = ORIGINAL(twoRow/2, 0);
        A(twoRow + 1, 4) = ORIGINAL(twoRow/2, 1);
        A(twoRow + 1, 5) = 1;
        A(twoRow + 1, 6) = -NEW(twoRow/2, 1)*ORIGINAL(twoRow/2, 0);
        A(twoRow + 1, 7) = -NEW(twoRow/2, 1)*ORIGINAL(twoRow/2, 1);

        B(twoRow, 0) = NEW(twoRow/2, 0);
        B(twoRow + 1, 0) = NEW(twoRow/2, 1);
    }

    Image LAMBDA = inverse(A) ->* B;
    Image matrix = identity(3);
    for(int a = 0; a < LAMBDA.getRow(); ++a)
        matrix(a/3, a%3) = LAMBDA(a, 0);
    return matrix;
}

Image identity(int Z)
{
    Image ident(Z,Z);
    for(int a = 0; a < ident.getRow(); ++a)
        ident(a,a) = 1;
    return ident;
}
```