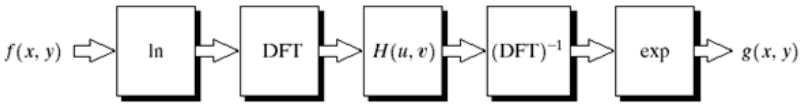
7. Homomorphic Filter and Bandreject Filter

1. Homomorphic Filter (bridge, goldhill):

Algorithm:

The process of homomorphic filter is similar to the Pass Filters in Lab6:



The image is first transformed by **DFT** to obtain a complex array of each pixel. The **real** and **imaginary** parts of each complex number are multiplied by $H(u,v)$ respectively. The formula of $H(u,v)$ is:

$$H(\mathrm{u},\mathrm{v}) = (\gamma_H - \gamma_L)[1 - e^{-c[D^2(u,v)/D_0^2]}] + \gamma_L$$

Where $\gamma H > 1 \; and \; \gamma L < 1$, and D_0 represents the radius of the passband. The calculation method of $D(u,v)$ is also the distance between two points, through the formula:

$$D(u,v) = \sqrt{\left(u - \frac{P}{2}\right)^2 + \left(v - \frac{Q}{2}\right)^2}$$

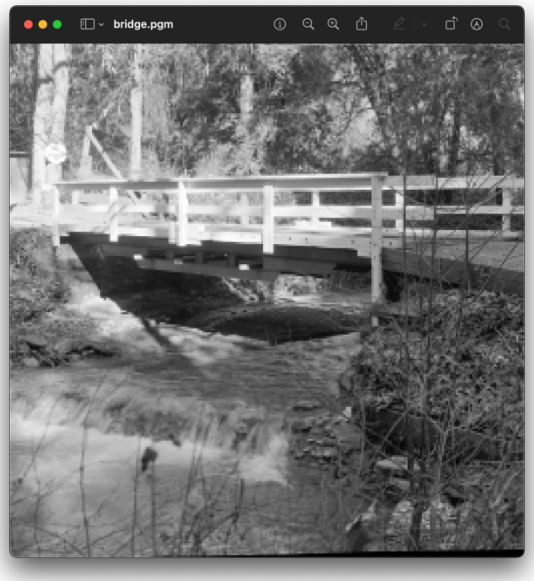
Where, P and Q are the length and width of the image.

In the following processing, I set $\gamma H = 1.5, \gamma L = 0.75, D_0 = 30, and \; c = 1.$

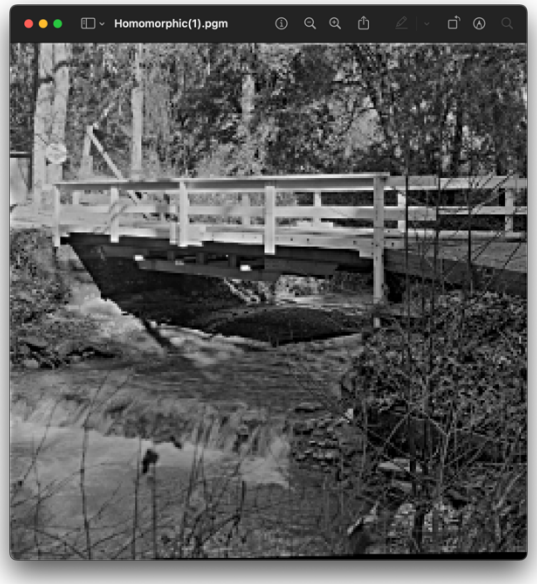
Results (including pictures):

Process result of "bridge.pgm":

Source Image:



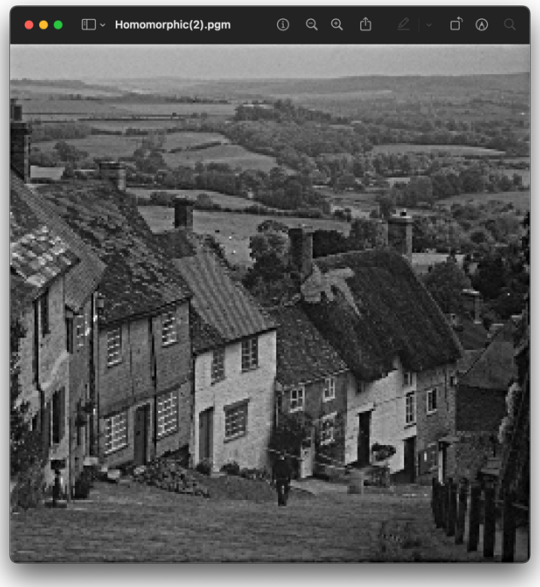
Result of Homomorphic filter:



Process result of "goldhill.pgm":

Source Image:

Result of Homomorphic filter:



Discussion:

Homomorphic filtering is essentially a frequency domain filter, which combines frequency filtering and spatial grayscale transformation. Typically, the illuminated part of the image is a slowly changing part, concentrated in the low frequency part. The reflection part belongs to the fast-changing part and tends to concentrate on the high frequency part.

Our model is in the shape of an inverse Gaussian filter that attenuates the effects of low-frequency components (illumination) and emphasizes high-frequency components (reflections). The final result is a simultaneous compression of dynamic range and enhancement of contrast, making textures more prominent.

Codes:

```
132 void Homomorphic(Image *image, float *real_array, float *imaginary_array) {
133     unsigned char *tempin, *tempout;
134     float height = image->Height, width = image->Width;
135     float real[(int)(height*width)], imaginary[(int)(height*width)];
136     Image *outimage;
137     outimage = CreateNewImage(image, (char*)"#testing function");
138     tempin = image->data;
139     tempout = outimage->data;
140 
141     for(int i = 0; i < height; i++) {
142         for(int j = 0; j < width; j++) {
143             float dis = sqrt(pow((float)i - height/2, 2) + pow((float)j - width/2, 2));
144             float H = (1.5 - 0.75) * (1 - pow(M_E, pow(dis / 30, 2) * -1)) + 0.75;
145             real[(int)(i*height + j)] = real_array[(int)(i*height + j)] * H;
146             imaginary[(int)(i*height + j)] = imaginary_array[(int)(i*height + j)] * H;
147         }
148     }
149     printf("Homomorphic Finished!\n");
150     outimage = iDFT(image, real, imaginary);
151     SavePNMImage(outimage, (char*)"Homomorphic.pgm");
152 }
```

2. Add sinusoidal noise and use bandreject filter to recover(lena):

Algorithm:

(1) Algorithm of adding noise:

$$p_{i,j}(out) = p_{i,j}(in) + noise_{i,j},$$
$$noise_{i,j} = A(\sin(iu) + \sin(jv)).$$

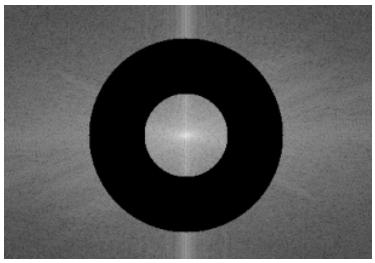
Where A is the amplitude, and u, v are the sine frequencies determined with respect to the x and y axes, respectively.

(2) Bandreject filter:

According to our spectral analysis of the noised image, we can see that the noise is distributed around the center. So we need a filter that blocks the passage of signals in certain frequency ranges and allows signals in other frequency ranges to pass:

$$H(u,v) = \begin{cases} 1, & D(u,v) < D_0 - \frac{W}{2} \\ 0, D_0 - \frac{W}{2} <= D(u,v) <= D_0 + \frac{w}{2} \\ 1, D(u,v) > D_0 + \frac{W}{2} \end{cases}$$

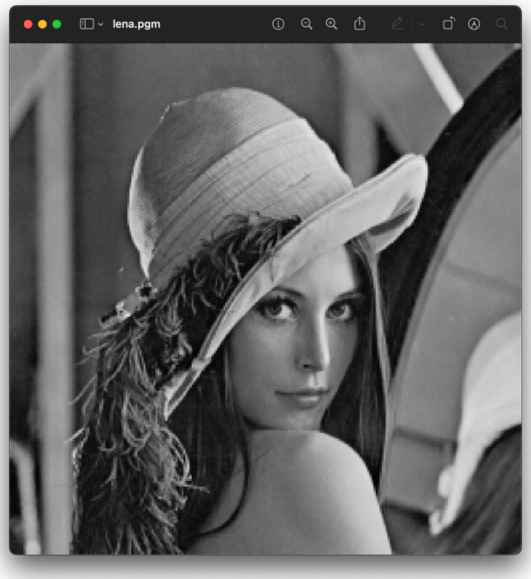
Where W is the width of the annular band. So the principle of the filter will look like this:



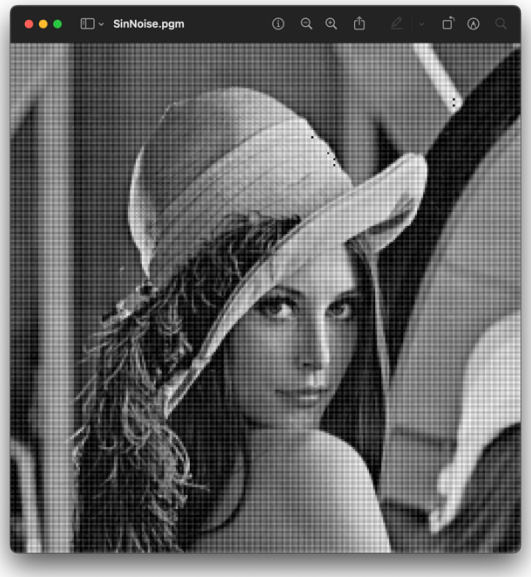
Results (including pictures):

Process result of “lena.pgm”:

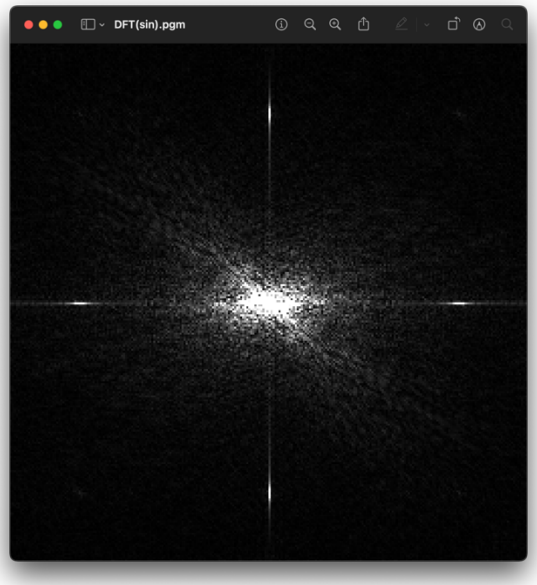
Source Image:

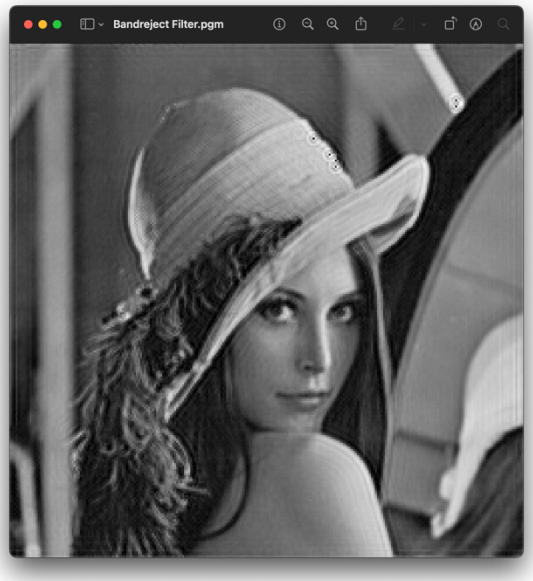


After adding sinusoidal noise:



The spectrum of the noisy image:





Restoring result:

Discussion:

We found that the periodic noise is distributed around the center of the spectrogram. Therefore, we can construct our band-reject filter according to the location and distance of the noise distribution to prevent the signals in these frequency ranges from passing through to achieve the purpose of noise reduction. And the restored image turns out to be back to normal. However, similar to the ideal LPF, its band boundary change is too steep, so it may cause a certain ringing phenomenon.

Codes:

(1) The code of adding sinusoidal noise:

```
154 void Sinusoidal(Image *image) {
155     unsigned char *tempin, *tempout;
156     Image *outimage;
157     outimage = CreateNewImage(image, (char*)"#testing function");
158     tempin = image->data;
159     tempout = outimage->data;
160 
161     for(int i = 0; i < image->Height; i++) {
162         for(int j = 0; j < image->Width; j++) {
163             float noise = 20 * (sin(i * 40) + sin(j * 40));
164             tempout[image->Height * i + j] = tempin[image->Height * i + j] + noise;
165         }
166     }
167     SavePNMImage(outimage, (char*)"SinNoise.pgm");
168 }
```

(2) The code of band-reject filter:

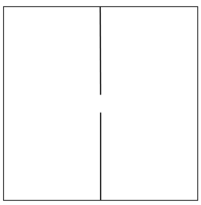
```
void BandrejectFilter(Image *image, float *real_array, float *imaginary_array) {
    unsigned char *tempin, *tempout;
    float height = image->Height, width = image->Width;
    float real[(int)(height*width)], imaginary[(int)(height*width)];
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function");
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 0; i < height; i++) {
        for(int j = 0; j < width; j++) {
            float dis = sqrt(pow((float)i - height/2, 2) + pow((float)j - width/2, 2));
            float H;
            if(dis > 75 && dis < 110) H = 0;
            else H = 1;
            real[(int)(i*height + j)] = real_array[(int)(i*height + j)] * H;
            imaginary[(int)(i*height + j)] = imaginary_array[(int)(i*height + j)] * H;
        }
    }
    printf("Bandreject Filter Finished!\n");
    outimage = iDFT(image, real, imaginary);
    SavePNMImage(outimage, (char*)"Bandreject Filter.pgm");
}
```

3. Enhance the noisy images (LenaWithNoise, cameraWithNoise):

- 3.1 LenaWithNoise:

Algorithm:

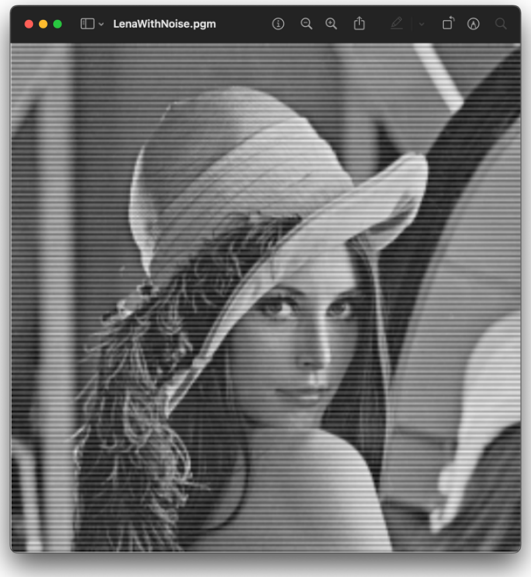
The image of “LenaWithNoise”shows” horizontal scan lines. From the DFT spectrum we find that there are vertical energy pulses in the center, so we want to isolate these noises to improve the image. Vertical notch reject filter is considered, it shows like this (white=1, black=0):



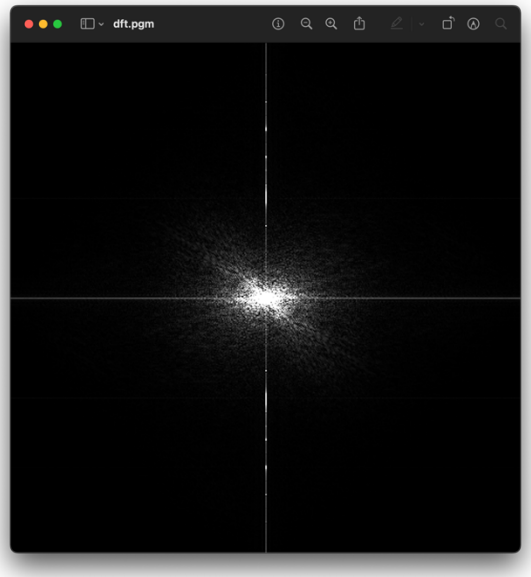
Results (including pictures):

Process result of “lena.pgm”:

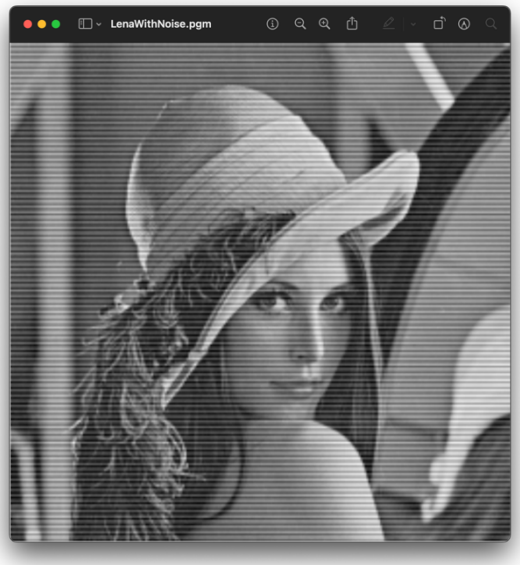
Source Image:



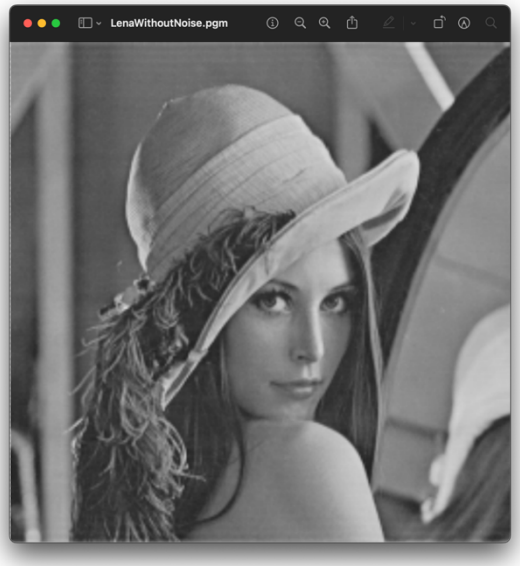
DFT Spectrum of the image:



Source image:



After using vertical notch reject filter:



Discussion:

This periodic interference is relatively easy to eliminate. By isolating the frequency on the vertical axis, the processed result is greatly improved compared to the original image.

Codes:

```
194 void noiseCancel1(Image *image, float *real_array, float *imaginary_array) {
195     unsigned char *tempin, *tempout;
196     float height = image->Height, width = image->Width;
197     float real[(int)(height*width)], imaginary[(int)(height*width)];
198     Image *outimage;
199     outimage = CreateNewImage(image, (char*)"#testing function");
200     tempin = image->data;
201     tempout = outimage->data;
202 
203     for(int i = 0; i < height; i++) {
204         for(int j = 0; j < width; j++) {
205             float dis = sqrt(pow((float)i - height/2, 2) + pow((float)j - width/2, 2));
206             float H;
207             if(j > 240 && j < 246 && dis >= 45) H = 0;
208             else H = 1;
209             real[(int)(i*height + j)] = real_array[(int)(i*height + j)] * H;
210             imaginary[(int)(i*height + j)] = imaginary_array[(int)(i*height + j)] * H;
211         }
212     }
213     outimage = iDFT(image, real, imaginary);
214     SavePNMImage(outimage, (char*)"LenaWithoutNoise.pgm");
215 }
```

- 3.2 cameraWithNoise:

Algorithm:

Method selected: **Adapted mean filter**

We define: $\boldsymbol{Z_{min}}$: min value in the mask, $\boldsymbol{Z_{max}}$: max value in the mask, $\boldsymbol{Z_{xy}}$: the central value in the mask, and $\boldsymbol{Z_{med}}$: the median value in the mask.

And the adaptive median filter has two processing procedures, denoted as: A and B respectively.

A:

$A1 = Z_{med} - Z_{min}$,

$A2 = Z_{med} - Z_{max}$,

$IF\ A1 > 0\ and\ A2 < 0\ then\ goes\ to\ B,$ (Z_{med} is not a noise)

$otherwise,$ expand the size of the mask. (Z_{med} is a noise)

B:

$B1 = Z_{xy} - Z_{min},$

$B2 = Z_{max} - Z_{xy},$

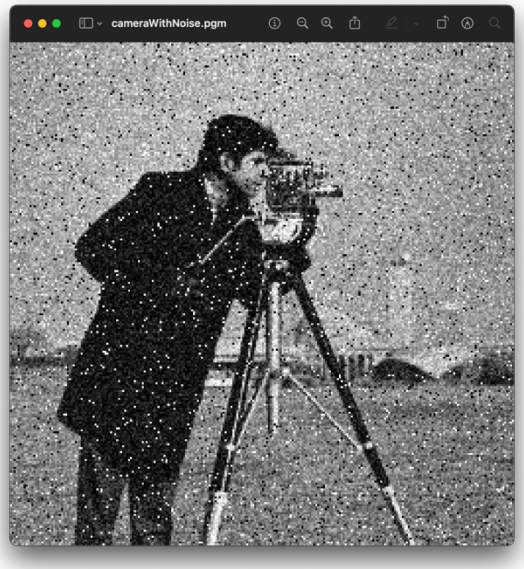
$IF\ B1 > 0\ and\ B2 < 0\ then\ output\ Z_{xy},$ (Z_{xy} is not a noise)

$otherwise\ output\ Z_{med}.$ (Z_{xy} is not a noise)

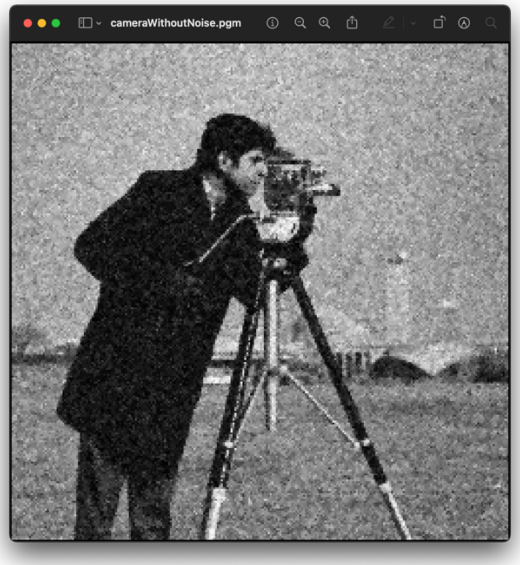
Results (including pictures):

Process result of "lena.pgm":

Result of GLPF($D_0 = 60$):



Result of GLPF($D_0 = 30$):



Discussion:

There are sporadic black and white dots in the picture, clearly the type of salt and pepper noise. The more appropriate way for salt and pepper noise is median filtering, but after practice, I found that the adaptive filter works better, and its loss of details and noise reduction are better.

Compared with the conventional median filter, the adaptive median filter can better protect the edge details in the image. Since it will check whether the central pixel and the median pixel in the mask are noises or not.

Codes:

```
344 void AdaptiveMedian(Image *image) {
345     unsigned char *tempin, *tempout, Sudoku[9], enlarge[25];
346     Image *outimage;
347     outimage = CreateNewImage(image, (char*)"#testing Function");
348     tempin = image->data;
349     tempout = outimage->data;
350
351     for(int i = 1; i < image->Height-1; i++) {
352         for(int j = 1; j < image->Width-1; j++){
353             int num = 0;
354             for(int x = -1; x <= 1; x++) {
355                 for(int y = -1; y <= 1; y++) {
356                     Sudoku[num++] = tempin[(image->Width)*(i+x) + (j+y)];
357                 }
358             }
359             // Use Insertion Sort:
360             for(int m = 1; m < 9; m++) {
361                 int currNum = Sudoku[m];
362                 int n = m;
```

```
                while(n >= 1 && Sudoku[n-1] > currNum) {
                    Sudoku[n] = Sudoku[n-1];
                    n--;
                }
                Sudoku[n] = currNum;
            }
            // Case 1: the median one is not a noise:
            if(Sudoku[0] < Sudoku[4] && Sudoku[4] < Sudoku[8]) {
                // check whether the current central pixel is a noise:
                int temp = tempin[image->Height * i + j];
                if(Sudoku[0] < temp && temp < Sudoku[8]) tempout[image->Height * i + j] = temp;
                else tempout[image->Height * i + j] = Sudoku[4];
            }

            // Case 2: the median one is a noise, so expand the mask:
            else {
                int num = 0;
                for(int x = -2; x <= 2; x++) {
                    for(int y = -2; y <= 2; y++) {
                        enlarge[num++] = tempin[(image->Width)*(i+x) + (j+y)];
                    }
                }
                // Use Insertion Sort:
                for(int m = 1; m < 25; m++) {
                    int currNum = enlarge[m];
                    int n = m;
                    while(n >= 1 && enlarge[n-1] > currNum) {
                        enlarge[n] = enlarge[n-1];
                        n--;
                    }
                    enlarge[n] = currNum;
                }
                // check whether the current central pixel is a noise:
                int temp = tempin[image->Height * i + j];
                if(enlarge[0] < temp && temp < enlarge[8]) tempout[image->Height * i + j] = temp;
                else tempout[image->Height * i + j] = enlarge[12];
            }
        }
    }
    SavePNMImage(outimage, (char*)"AdaptiveMedian.pgm");
}
// Algorithms End.
```

4. Compare and analyze the five noise canceling algorithms (lenaD1, D2, D3):

Algorithms:

(1) Arithmetic mean filter:

(i-1, j-1)	**(i-1, j)**	**(i-1, j+1)**
(i, j-1)	**(i, j)**	**(i, j+1)**
(i+1, j-1)	**(i+1, j)**	**(i+1, j+1)**

Each pixel in the picture is surrounded by **eight** pixels except for those at the edges which are ignored. And we recalculate the value of each pixel by taking the average of the **nine** pixels in figure. So we have the algorithm: $\hat{f}(x,y) = \frac{1}{mn} \sum_{(s,t)\in S_{xy}} g(s,t)$

$$Pixel(i,j) = data[i * Width + j], \quad i,j \geq 1, i < Height - 1, and\ j < Weight - 1.$$

$$newPixel(i,j) = \sum_{x=-1}^{1} \sum_{y=-1}^{1} originalPixel(i+x, j+y)/9 = \sum_{x=-1}^{1} \sum_{y=-1}^{1} data[(i+x) * Width + (j+y)]/9$$

(2) Geometric mean filter:

The algorithm is changed to:

$$\hat{f}(x,y)=\left[\prod_{(s,t)\in S_{xy}} g(s,t)\right]^{\frac{1}{mn}}$$

(3) Median filter:

Similar to the Arithmetic mean filter, but the value of each pixel is replaced by the **median** of the mask instead of the average. I store the values of 9 surrounded pixels into an array and use the **Insertion Sort** method to find its median, $array[4]$, which will be assigned to $Pixel(i,j)$.

(4) Alpha-trimmed mean filter:

$$\hat{f}(x,y)=\frac{1}{MN-2d}\sum_{(s,t)\in S_{xy}} g_r(x,y)$$

The modified alpha mean filter class removes the highest and lowest values, that is, sorts the data within the filtering range, removes d data in order from large to small, removes d data in order from small to large, and calculates the remaining data. mean.

(5) Adaptive mean filter:

We define: $\boldsymbol{Z_{min}}$: min value in the mask, $\boldsymbol{Z_{max}}$: max value in the mask, $\boldsymbol{Z_{xy}}$: the central value in the mask, and $\boldsymbol{Z_{med}}$: the median value in the mask.

And the adaptive median filter has two processing procedures, denoted as: A and B.

A:

$A1 = Z_{med} - Z_{min}$,

$A2 = Z_{med} - Z_{max}$,

$IF\ A1 > 0\ and\ A2 < 0\ then\ goes\ to\ B,$ (Z_{med} is not a noise)

$otherwise,$ expand the size of the mask. (Z_{med} is a noise)

B:

$B1 = Z_{xy} - Z_{min}$,

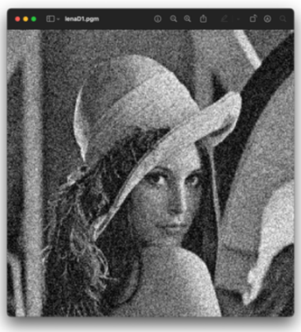
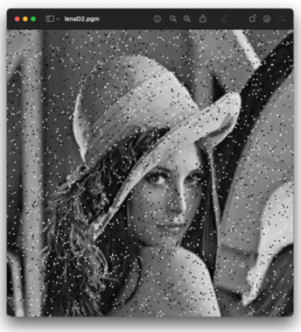
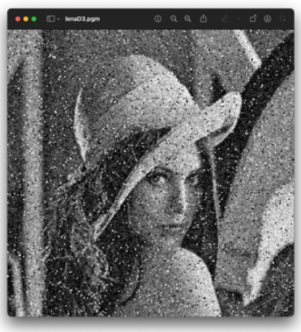
$B2 = Z_{max} - Z_{xy}$,

$IF\ B1 > 0\ and\ B2 < 0\ then\ output\ Z_{xy}$, ($Z_{xy}$ is not a noise)

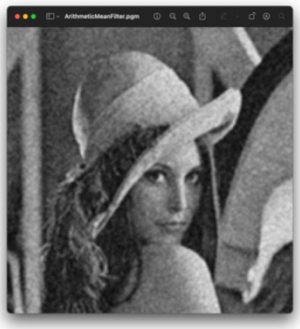
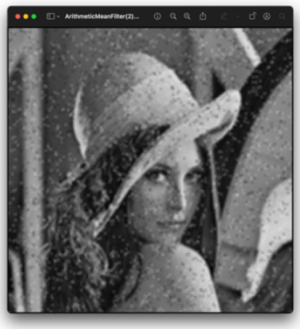
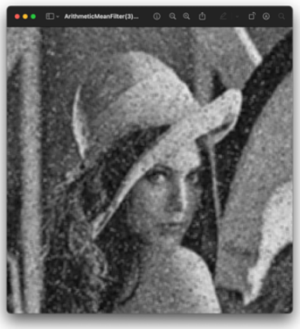
$otherwise\ output\ Z_{med}$. ($Z_{xy}$ is not a noise)

Results (including pictures – please refer to the attaching files if some of them seem blurry):

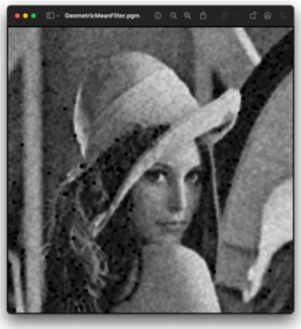
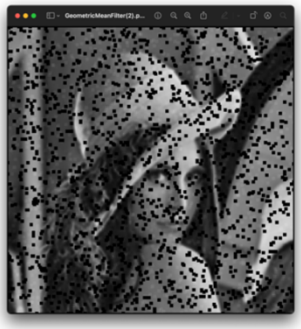
The source images(D1, D2, D3):

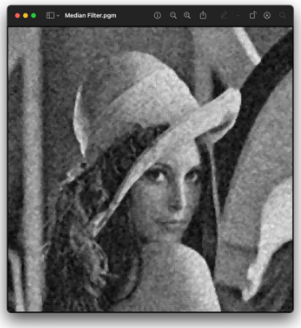
After using Arithmetic mean filter (D1, D2, D3):

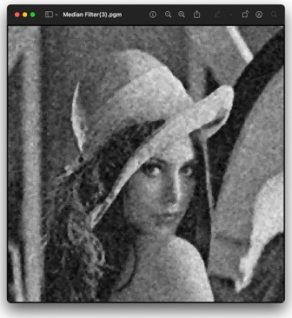




After using Geometric mean filter (D1, D2, D3):

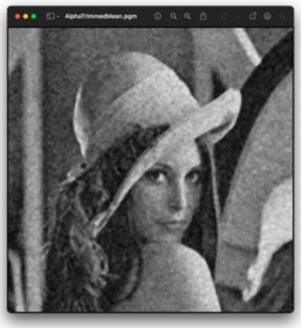




After using Median filter (D1, D2, D3):

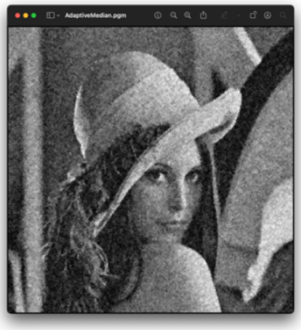


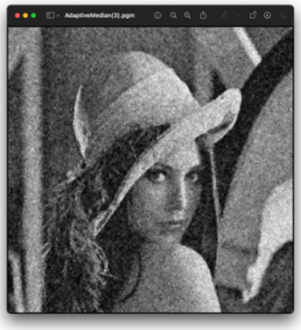


After using Alpha-trimmed mean filter (D1, D2, D3):



After using Adaptive mean filter (D1, D2, D3):





Discussion:

Image D1 is polluted by **Gaussian** noise, image D2 is polluted by **salt** and **pepper** noise, and image D3 is polluted by a mixture of **Gaussian** and **salt** and **pepper** noise.

i. **Arithmetic mean filtering** has a little effect on Gaussian noise and salt and pepper noise, but it will smooth the image at the same time.
ii. **Geometric mean filter** loses less image detail than the arithmetic mean filter while handling the Gaussian and salt noise, but it cannot deal with pepper noise at all.
iii. **Median filter** has a certain effect on Gaussian noise and is good at handling salt and pepper noise, and it will keep more details compared with the Arithmetic mean filtering.
iv. **Alpha-trimmed mean filter** is good at removing noise in the images that are polluted by salt and pepper noise along with other types of noise.
v. **Adaptive mean filter** has the overall best performance to reduce to noise especially the salt and pepper noise. It also better protects the edge details in the image compared to the normal median filter.

Codes:

(1) Arithmetic mean filter:

```
void ArithmeticMeanFilter(Image *image) {
    unsigned char *tempin, *tempout, Sudoku[9];
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing Function");
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 1; i < image->Height-1; i++) {
        for(int j = 1; j < image->Width-1; j++){
            int num = 0;
            for(int x = -1; x <= 1; x++) {
                for(int y = -1; y <= 1; y++) {
                    Sudoku[num++] = tempin[(image->Width)*(i+x) + (j+y)];
                }
            }
            int sum = 0;
            for(int k = 0; k < 9; k++) {
                sum += Sudoku[k];
            }
            sum /= 9;
            if(sum > 255) sum = 255;
            if(sum < 0) sum = 0;
            tempout[image->Height * i + j] = sum;
        }
    }
    SavePNMImage(outimage, (char*)"ArithmeticMeanFilter.pgm");
}
```

(2) Geometric mean filter:

```
void GeometricMeanFilter(Image *image) {
    unsigned char *tempin, *tempout, Sudoku[9];
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing Function");
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 1; i < image->Height-1; i++) {
        for(int j = 1; j < image->Width-1; j++){
            int num = 0;
            for(int x = -1; x <= 1; x++) {
                for(int y = -1; y <= 1; y++) {
                    Sudoku[num++] = tempin[(image->Width)*(i+x) + (j+y)];
                }
            }
            float product = 1.0;
            for(int k = 0; k < 9; k++) {
                product *= Sudoku[k];
            }
            product = pow(product, 1.0/9.0);
            int temp = product;
            if(temp > 255) temp = 255;
            if(temp < 0) temp = 0;
            tempout[image->Height * i + j] = temp;
        }
    }
    SavePNMImage(outimage, (char*)"GeometricMeanFilter.pgm");
}
```

(3) Median filter:

```
274 void MedianFilter(Image *image) {
275     unsigned char *tempin, *tempout, Sudoku[9];
276     Image *outimage;
277     outimage = CreateNewImage(image, (char*)"#testing Function");
278     tempin = image->data;
279     tempout = outimage->data;
280 
281     for(int i = 1; i < image->Height-1; i++) {
282         for(int j = 1; j < image->Width-1; j++){
283             int num = 0;
284             for(int x = -1; x <= 1; x++) {
285                 for(int y = -1; y <= 1; y++) {
286                     Sudoku[num++] = tempin[(image->Width)*(i+x) + (j+y)];
287                 }
288             }
289             // Use Insertion Sort:
290             for(int m = 1; m < 9; m++) {
291                 int currNum = Sudoku[m];
292                 int n = m;
293                 while(n >= 1 && Sudoku[n-1] > currNum) {
294                     Sudoku[n] = Sudoku[n-1];
295                     n--;
296                 }
297                 Sudoku[n] = currNum;
298             }
299             tempout[(image->Width)*i + j] = Sudoku[4];
300         }
301     }
302     SavePNMImage(outimage, (char*)"Median Filter.pgm");
303 }
```

(4) Alpha-trimmed mean filter:

```
305 void AlphaTrimmedMean(Image *image) {
306     unsigned char *tempin, *tempout, Sudoku[9];
307     Image *outimage;
308     outimage = CreateNewImage(image, (char*)"#testing Average Filter");
309     tempin = image->data;
310     tempout = outimage->data;
311 
312     for(int i = 1; i < image->Height-1; i++) {
313         for(int j = 1; j < image->Width-1; j++){
314             int num = 0;
315             for(int x = -1; x <= 1; x++) {
316                 for(int y = -1; y <= 1; y++) {
317                     Sudoku[num++] = tempin[(image->Width)*(i+x) + (j+y)];
318                 }
319             }
320             // Use Insertion Sort:
321             for(int m = 1; m < 9; m++) {
322                 int currNum = Sudoku[m];
323                 int n = m;
324                 while(n >= 1 && Sudoku[n-1] > currNum) {
325                     Sudoku[n] = Sudoku[n-1];
326                     n--;
327                 }
328                 Sudoku[n] = currNum;
329             }
330             // set the d/2 to 2:
331             int sum = 0, d = 2;
332             for(int k = d; k < 9-d; k++) {
333                 sum += Sudoku[k];
334             }
335             sum /= 9 - 2 * d;
336             if(sum > 255) sum = 255;
337             if(sum < 0) sum = 0;
338             tempout[image->Height * i + j] = sum;
339         }
340     }
341     SavePNMImage(outimage, (char*)"AlphaTrimmedMean.pgm");
342 }
```

(5) Adaptive mean filter:

```
void AdaptiveMedian(Image *image) {
    unsigned char *tempin, *tempout, Sudoku[9], enlarge[25];
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing Function");
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 1; i < image->Height-1; i++) {
        for(int j = 1; j < image->Width-1; j++){
            int num = 0;
            for(int x = -1; x <= 1; x++) {
                for(int y = -1; y <= 1; y++) {
                    Sudoku[num++] = tempin[(image->Width)*(i+x) + (j+y)];
                }
            }
            // Use Insertion Sort:
            for(int m = 1; m < 9; m++) {
                int currNum = Sudoku[m];
                int n = m;
                while(n >= 1 && Sudoku[n-1] > currNum) {
                    Sudoku[n] = Sudoku[n-1];
                    n--;
                }
                Sudoku[n] = currNum;
            }
            // Case 1: the median one is not a noise:
            if(Sudoku[0] < Sudoku[4] && Sudoku[4] < Sudoku[8]) {
                // check whether the current central pixel is a noise:
                int temp = tempin[image->Height * i + j];
                if(Sudoku[0] < temp && temp < Sudoku[8]) tempout[image->Height * i + j] = temp;
                else tempout[image->Height * i + j] = Sudoku[4];
            }

            // Case 2: the median one is a noise, so expand the mask:
            else {
                int num = 0;
                for(int x = -2; x <= 2; x++) {
                    for(int y = -2; y <= 2; y++) {
                        enlarge[num++] = tempin[(image->Width)*(i+x) + (j+y)];
                    }
                }
                // Use Insertion Sort:
                for(int m = 1; m < 25; m++) {
                    int currNum = enlarge[m];
                    int n = m;
                    while(n >= 1 && enlarge[n-1] > currNum) {
                        enlarge[n] = enlarge[n-1];
                        n--;
                    }
                    enlarge[n] = currNum;
                }
                // check whether the current central pixel is a noise:
                int temp = tempin[image->Height * i + j];
                if(enlarge[0] < temp && temp < enlarge[8]) tempout[image->Height * i + j] = temp;
                else tempout[image->Height * i + j] = enlarge[12];
            }
        }
    }
    SavePNMImage(outimage, (char*)"AdaptiveMedian.pgm");
}
// Algorithms End.
```