Lab 7

1. Use a homomorphic filter to enhance bridge.pgm and goldhill.pgm
2. Add sinusoidal noise to lena.pgm and use bandreject filter to recover lena.pgm
3. There are two noisy images lenaWithNoise.pgm and cameraWithNoise.pgm. The task is to get the noise patterns and to design filters to remove or reduce the noise to get uncorrupted or better quality lena and camera images.

Algorithm:

1. Homographic filter

   The formula of homographic filter is shown below

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

   input image (result of DFT), high frequency γ_H (γ_H>1), low frequency γ_L (γ_L<1), control constanct c, cutoff frequency D_0

```
for (i = 0; i < image.Height; i++) {
    for (j = 0; j < image.Width; j++) {
        dist = sqrt((i -image.Height / 2)^2) +(j – image.Width / 2)^2);
        filter = (γH - γL) * (1 - exp((-c) * (dist / D0, 2)^2) + γL;
        image[i][j]*= filter;
    }
}
return image
```

2. Add sinusoidal noise

   To add the periodic noise, use the formula

$$g(x,y) = F(x,y) + Asin(A*x) + Asin(A*y)$$

   Input original image, amplitude c

```
for (i = 0; i < image.Height; i++) {
    for (j = 0; j < image.Width; j++) {
        res = image [i][j] + c * sin(c * i) + c * sin(c * j);
        outimage[i][j] = res;
    }
}
return outimage
```

3. Ideal band reject filter

   The formula of ideal band reject filter is shown below

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

   input image (result of DFT), , cutoff frequency D_0, band width W

```
for (i = 0; i < image.Height; i++) {
    for (j = 0; j < image.Width; j++) {
        dist = sqrt((i -image.Height / 2)^2) +(j – image.Width / 2)^2);
        if (dist >=(D0-width/2) and dist<=(D0 + width / 2)) {
            image[i][j] = 0;
        }
```

```
        }
    }
    return image
```

4. Butterworth band reject filter

   The formula of butterworth band reject filter is shown below

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

   input image (result of DFT), cutoff frequency D_0, band width W, order of filter n

```
    for (i = 0; i < image.Height; i++) {
        for (j = 0; j < image.Width; j++) {
            dist = sqrt((i -image.Height / 2)^2) +(j – image.Width / 2)^2);
            image[i][j] *= 1 / (1 + pow(dist * W /(dist^2 - D0^2), 2n));
        }
    }
    return image
```

5. Gaussian band reject filter

   The formula of Gaussian band reject filter is shown below

$$H(u,v) = 1 - e^{-[\frac{D^2 - {D_0}^2}{DW}]^2}$$

   input image (result of DFT), cutoff frequency D_0, band width W

```
    for (i = 0; i < image.Height; i++) {
        for (j = 0; j < image.Width; j++) {
            dist = sqrt((i -image.Height / 2)^2) +(j – image.Width / 2)^2);
            image[i][j] *= 1 - exp(-pow(dist^2-D0^2) / (dist * width), 2));
        }
    }
    return image
```

6. Arithmetic mean filter

$$\hat{f}(x,y) = \frac{1}{mn} \sum_{(s,t) \in S_{xy}} g(s,t)$$

   Input: an unsigned char array that stores the source image data, the height, width of mask m,n

   Output: an unsigned char array that stores the output image data

```
    for (i = m / 2; i < inimage. Height -m / 2; i++) {
        for (j = n / 2; j < inimage.Width - n / 2; j++) {
            temp = 0;
            for (p = -1; p < m-1; p++) {
                for (q = -1; q < n-1; q++) {
                    temp = temp+tempin[(i + p)][(j + q)];
                }
            }
            outimage[i][j] = temp /(m * n);
        }
    }
```

return outImage;

7. Geometric filter

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

Input: an unsigned char array that stores the source image data, the height, width of mask m,n

Output: an unsigned char array that stores the output image data

```
for (i = m / 2; i < inimage. Height -m / 2; i++) {
    for (j = n / 2; j < inimage.Width - n / 2; j++) {
        temp = 1;
        for (p = -1; p < m-1; p++) {
            for (q = -1; q < n-1; q++) {
                temp*=tempin[(i + p)][(j + q)];
            }
        }
        outimage[i][j] = pow(temp ,1/(m * n));
    }
}
return outimage
```

8. Median filter

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

Input: an unsigned char array that stores the source image data, the size of mask

Output: an unsigned char array that stores the output image data

```
for (i = m / 2; i < inimage. Height -m / 2; i++) {
    for (j = n / 2; j < inimage.Width - n / 2; j++) {
        temp = 1;
        for (p = -1; p < m-1; p++) {
            for (q = -1; q < n-1; q++) {
                tempArr[temp++]=inimage[(i + p)][(j + q)];
            }
        }
        outimage[i ][j] = Median(tempArr,size*size);
    }
}
}
return outimage
```

9. Alpha-trimmed mean filter

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

Input: an unsigned char array that stores the source image data, the height, width of mask m,n, adjust parameter d

Output: an unsigned char array that stores the output image data

```
for (i = m / 2; i < inimage. Height -m / 2; i++) {
    for (j = n / 2; j < inimage.Width - n / 2; j++) {
        temp = 0;
```

```
                for (p = -1; p < m-1; p++) {
                    for (q = -1; q < n-1; q++) {
                        temp = temp+tempin[(i + p)][(j + q)];
                    }
                }
                outimage[i][j] = temp /(m * n-d);
            }
        }
    return outimage
```

10. Adaptive median filter

Consider the following notation:

$$
\begin{aligned}
z_{\min} &= \text{minimum intensity value in } S_{xy} \\
z_{\max} &= \text{maximum intensity value in } S_{xy} \\
z_{\text{med}} &= \text{median of intensity values in } S_{xy} \\
z_{xy} &= \text{intensity value at coordinates } (x, y) \\
S_{\max} &= \text{maximum allowed size of } S_{xy}
\end{aligned}
$$

The adaptive median-filtering algorithm works in two stages, denoted stage A and stage B, as follows:

Stage A:

$$
\begin{aligned}
&A1 = z_{\text{med}} - z_{\min} \\
&A2 = z_{\text{med}} - z_{\max} \\
&\text{If } A1 > 0 \text{ AND } A2 < 0, \text{ go to stage } B \\
&\text{Else increase the window size} \\
&\text{If window size} \leq S_{\max} \text{ repeat stage } A \\
&\text{Else output } z_{\text{med}}
\end{aligned}
$$

Stage B:

$$
\begin{aligned}
&B1 = z_{xy} - z_{\min} \\
&B2 = z_{xy} - z_{\max} \\
&\text{If } B1 > 0 \text{ AND } B2 < 0, \text{ output } z_{xy} \\
&\text{Else output } z_{\text{med}}
\end{aligned}
$$

Discussion:

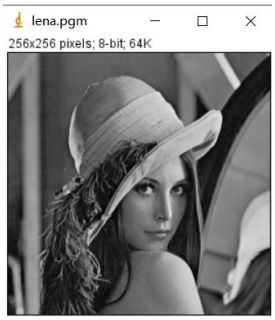
1. After conducting the homographic filter, it can be seen that the overall brightness range becomes smaller, and the details in the dark appear. The disadvantage is that the frequency domain transformation is involved, the calculation is complex, and the details will inevitably be lost after filtering
2. Periodic noise is a kind of noise that exists in the spatial domain and is related to a specific frequency. If the amplitude of the sine wave in the spatial domain is strong enough, then the pulse pair of each sine wave in the image will be seen in the spectrum of the image.
3. Band reject filters can be used to remove periodic noise. Ideal band reject filter has the best performance, than is Butterworth band reject filter. Gaussian band reject filter is the worst.
4. Median, alpha-trimmed mean and adaptive median filters can remove salt and pepper noise, and adaptive median filters are the best at removing salt and pepper noise. Geometric mean filter will make the image damaged by pepper noise. The larger kernel size would generate blurrier images.

Results (compare the results with the original image):

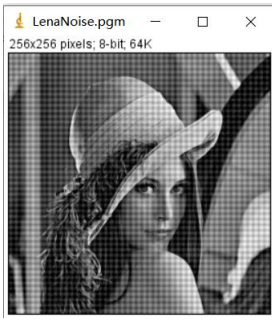
1. Homographic filter

origin	after
	
	

2. Add sinusoidal noise

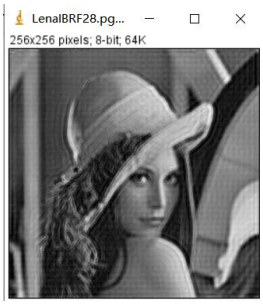


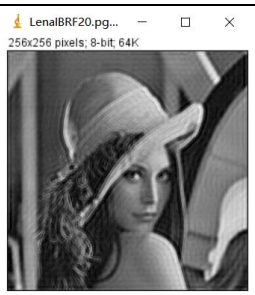
Before



After

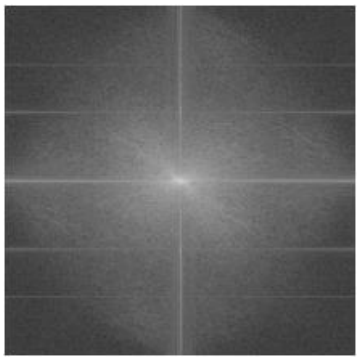
3. Band reject filters

	Ideal	Butterworth	Gaussian
W=5			
W=10			

W=20			

4. Question2

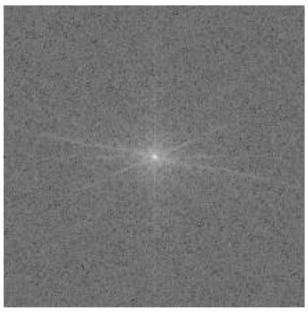
1) LenaWithNoise.pgm

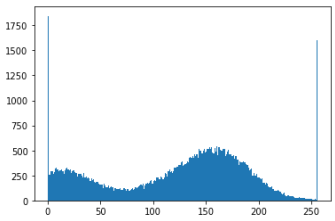


Origin Spectum

We can see that the image has a periodic noise, we can use band reject to remove it. Here I use the Butterworth band reject filter, the result is shown below, and the algorithm of which is shown previously.

2) CamaraWithNoise.pgm

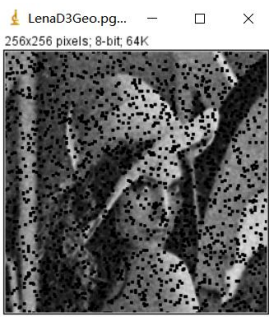
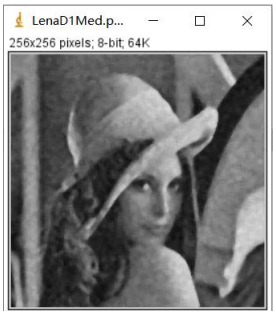




Origin Spectrum Histogram

We can see that the image has a pepper noise, we can use mean or median filters to smooth is. Here I use the median filter, the result is shown below, and the algorithm of which is shown previously.

5. Mean and median filters

	LenaD1	LenaD2	LenaD3
origin			
Arithmetic mean			
geometric mean			
median			

Alpha-trimmed mean			
adaptive median			

Codes:

1. Homographic filter

```
Image* HomoFilter(Image* image, float high, float low, float cutoff, float c) {
    int i, j;
    Image* inimage, * outimage;
    float dist;
    float filter;
    struct _complex* in = (struct _complex*)malloc(sizeof(struct _complex) * image->Height * image->Width);
    struct _complex* out = (struct _complex*)malloc(sizeof(struct _complex) * image->Height * image->Width);


    for (i = 0; i < image->Height * image->Width; i++) {
        in[i].x = 1.0 * image->data[i];
        in[i].y = 0.0;
    }

    FFT(in, out, 1, image->Height, image->Width);

    for (i = 0; i < image->Height; i++) {
        for (j = 0; j < image->Width; j++) {
            dist = sqrt(pow((i - (double)image->Height / 2), 2) + pow((j - (double)image->Width / 2), 2));
            filter = (high - low) * (1 - exp((-c) * (pow(dist / cutoff, 2)))) + low;
            out[i * image->Width + j].x *= filter;
        }
    }

    FFT(out, out, -1, image->Height, image->Width);

    outimage = CreateNewImage(image, "#HomoFilter", image->Height, image->Width);

    for (int i = 0; i < image->Height * image->Width; i++) {
        outimage->data[i] = (out[i].x < 0) ? 0 : ((out[i].x > 255) ? 255 : out[i].x);
    }

    return (outimage);
}
```

2. Add sinusoidal noise

```
Image* AddSinNoise(Image* image,float c) {
    int i, j;
    Image * outimage;
    unsigned char *tempin, * tempout;
    float dist;
    float filter;
    int temp, min=255, max=0;
    float res;


    outimage = CreateNewImage(image, "#AddSinNoise", image->Height, image->Width);
    tempin = image->data;
    tempout = outimage->data;

    for (i = 0; i < image->Height; i++) {
        for (j = 0; j < image->Width; j++) {
            res = tempin[i * image->Width + j] + c * sin(c * i) + c * sin(c * j);
            res = (res < 0) ? 0 : ((res > 255) ? 255 : res);
            tempout[i * image->Width + j] = res;
        }
    }

    return (outimage);
}
```

3. Ideal band reject filter

```
for (i = 0; i < image->Height; i++) {
    for (j = 0; j < image->Width; j++) {
        dist = sqrt(pow((i - (double)image->Height / 2), 2) + pow((j - (double)image->Width / 2), 2));
        if (dist >(cutoff-width/2)&& dist<(cutoff + width / 2)) {
            out[i * image->Width + j].x = 0;
            out[i * image->Width + j].y = 0;
        }
    }
}
```

4. Butterworth band reject filter

```
for (i = 0; i < image->Height; i++) {
    for (j = 0; j < image->Width; j++) {
        dist = sqrt(pow((i - (double)image->Height / 2), 2) + pow((j - (double)image->Width / 2), 2));
        out[i * image->Width + j].x *= 1 / (1 + pow(dist * width / (pow(dist, 2) - pow(radius, 2)), 2 * order));
        out[i * image->Width + j].y *= 1 / (1 + pow(dist * width / (pow(dist, 2) - pow(radius, 2)), 2 * order));
    }
}
```

5. Gaussian band reject filter

```
for (i = 0; i < image->Height; i++) {
    for (j = 0; j < image->Width; j++) {
        dist = sqrt(pow((i - (double)image->Height / 2), 2) + pow((j - (double)image->Width / 2), 2));
        out[i * image->Width + j].x *= 1 - exp(-pow((pow(dist, 2) - pow(radius, 2)) / (dist * width), 2));
        out[i * image->Width + j].y *= 1 - exp(-pow((pow(dist, 2) - pow(radius, 2)) / (dist * width), 2));
    }
}
```

6. Median filter

```
Image* MedianFilter(Image* image, int size) {
    Image* outImage;
    int i, j, p, q, count = 0, temp, tempArr[26];
    unsigned char* tempin, * tempout;


    outImage = CreateNewImage(image, "#MedianFilter", image->Width, image->Height);
    tempout = outImage->data;
    tempin = image->data;

    for (i = size / 2; i < image->Width - size / 2; i++) {
        for (j = size / 2; j < image->Height - size / 2; j++) {
            temp = 0;
            for (p = -1; p < size - 1; p++) {
                for (q = -1; q < size - 1; q++) {
                    tempArr[temp++] = tempin[(i + p) * image->Width + (j + q)];
                }
            }
            tempout[i * outImage->Width + j] = Median(tempArr, size * size);
        }
    }
    return outImage;
}
```

7. Arithmetic mean filter

```
Image* AriAverageFilter(Image* image, int size) {
    Image* outImage;
    int i, j, p, q, temp;
    unsigned char* tempin, * tempout;

    outImage = CreateNewImage(image, "#AriAverageFilter", image->Width, image->Height);
    tempout = outImage->data;
    tempin = image->data;

    for (i = size / 2; i < image->Width - size / 2; i++) {
        for (j = size / 2; j < image->Height - size / 2; j++) {
            temp = 0;
            for (p = -1; p < size - 1; p++) {
                for (q = -1; q < size - 1; q++) {
                    temp = temp + tempin[(i + p) * image->Width + (j + q)];
                }
            }
            tempout[i * outImage->Width + j] = temp / (size * size);
        }
    }
    return outImage;
}
```

8. Geometric mean filter

```
Image* GeoAverageFilter(Image* image, int size) {
    Image* outImage;
    int i, j, p, q;
    unsigned char* tempin, * tempout;
    double temp;
    outImage = CreateNewImage(image, "#GeoAverageFilter", image->Width, image->Height);
    tempout = outImage->data;
    tempin = image->data;

    for (i = size / 2; i < image->Width - size / 2; i++) {
        for (j = size / 2; j < image->Height - size / 2; j++) {
            temp = 1;
            for (p = -1; p < size - 1; p++) {
                for (q = -1; q < size - 1; q++) {
                    temp *= (double)tempin[(i + p) * image->Width + (j + q)];
                }
            }
            tempout[i * outImage->Width + j] = pow(temp ,1.0 / (double)(size * size));
        }
    }

    for (int i = 0; i < image->Height * image->Width; i++) {
        tempout[i] = (tempout[i] < 0) ? 0 : ((tempout[i] > 255) ? 255 : tempout[i]);
    }
    return outImage;
}
```

9. Alpha trimmed mean filter

```
Image* AlphaTriMeanFilter(Image* image, int size, float d) {
    Image* outImage;
    int i, j, p, q, temp;
    unsigned char* tempin, * tempout;

    outImage = CreateNewImage(image, "#AlphaTriMeanFilter", image->Width, image->Height);
    tempout = outImage->data;
    tempin = image->data;

    for (i = size / 2; i < image->Width - size / 2; i++) {
        for (j = size / 2; j < image->Height - size / 2; j++) {
            temp = 0;
            for (p = -1; p < size - 1; p++) {
                for (q = -1; q < size - 1; q++) {
                    temp = temp + tempin[(i + p) * image->Width + (j + q)];
                }
            }
            tempout[i * outImage->Width + j] = temp / (size * size-d);
        }
    }
    return outImage;
}
```

10. Adaptive median filter

```
Image* AdapMedFilter(Image* image, int size, int smax) {
    Image* outImage;
    int i, j, p, q, count = 0, temp, tempArr[26], min, max, mid, z, a1, a2, b1, b2, n;
    unsigned char* tempin, * tempout;
    int len = size * size;

    outImage = CreateNewImage(image, "#MedianFilter", image->Width, image->Height);
    tempout = outImage->data;
    tempin = image->data;

    for (i = size / 2; i < image->Width - size / 2; i++) {
        for (j = size / 2; j < image->Height - size / 2; j++) {
            temp = 0;

            for (p = -1; p < size - 1; p++) {
                for (q = -1; q < size - 1; q++) {
                    tempArr[temp++] = tempin[(i + p) * image->Width + (j + q)];
                }
            }
            n = size;
            qsort(tempArr, len, sizeof(int), cmp);
            mid = Median(tempArr, size * size);
            min = tempArr[0];
            max = tempArr[len - 1];
            z = tempin[i * image->Width + j];
            a1 = mid - min;
            a2 = mid - max;
            if (a1 > 0 && a2 < 0) {
                b1 = z - min;
                b2 = z - max;
                if (b1 > 0 && b2 < 0)
                    tempout[i * outImage->Width + j] = z;
                else
                    tempout[i * outImage->Width + j] = mid;
            }
            else {
                n += 2;
                if (n > smax) {
                    tempout[i * image->Width + j] = z;
                    break;
                }
            }
        }
    }
    return outImage;
}
```