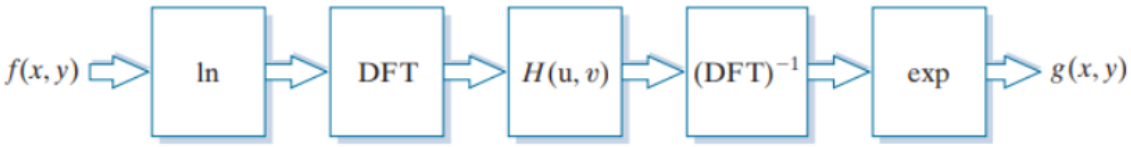
Class Lab 7

Use two images for each operation to do the following operations and write down their advantages and disadvantages and explain your results:

1. Homomorphic Filter (bridge, goldhill):

Algorithm:



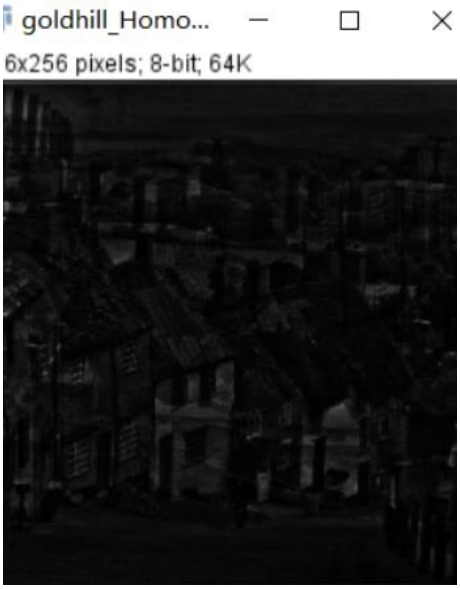
$$H(u,v) = (\gamma_H - \gamma_L)\left[1 - e^{-cD^2(u,v)/D_0^2}\right] + \gamma_L$$

Results (including pictures):

Result of processing "goldhill.pgm":

Source Image:

Result after homomorphic Filter:



Result of processing “Bridge.pgm”:

Source Image:

Result after homomorphic Filter:

Discussion:

A good deal of control can be gained over the illumination and reflectance components with a homomorphic filter. This control requires specification of a filter transfer function that affects the low- and high-frequency components of the Fourier transform in different, controllable ways. The net result is simultaneous dynamic range compression and contrast enhancement.

Codes:

```
int size = image->Height * image->Width;
int width = image->Width, height = image->Height;

struct _complex *src = (struct _complex *)malloc(sizeof(struct _complex) * size);

for (int i = 0; i < size; ++i) {···

fft(src, src, 1, height, width);

for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        double des = sqrt(pow(i - (double)height / 2, 2) + pow(j - (double)width / 2, 2));
        double p = -c * pow(des / radius, 2);
        src[i * width + j].x *= (gamma1 - gamma2) * (1 - exp(p)) + gamma2;
    }
}

fft(src, src, -1, height, width);

outimage = CreateNewImage(image, height, width, (char *)"#Homomorphic img");

for (int i = 0; i < size; i++) {···
```

2. Sinusoidal Noise (Lena):

Algorithm:

$$g(x,y) = f(x,y) + \alpha \sin(i+j)$$

Results (including pictures):

Result of processing "lena.pgm":

Source Image:

Result after add noise:

Codes:

```
Image *AddSinNoiseImage(Image *image, float sigma) {        WYX, 3天前 • lab7 …
    Image *outimage;
    outimage = CreateNewImage(image, image->Height, image->Width, (char *)"#Add Sin Noise");

    int size = image->Height * image->Width;
    int width = image->Width, height = image->Height;

    int *src = (int *)malloc(sizeof(int) * size);

    for (int i = 0; i < size; ++i) {…

    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            src[i * width + j] += sigma * sin(i + j);
        }
    }

    for (int i = 0; i < size; i++) {…

    return (outimage);
}
```

3. Ideal Notch Band Reject Filter (lena):

Algorithm:

$$d1 = \sqrt{(i-x)^2 + (j-y)^2}$$

$$d2 = \sqrt{(i-(height-x))^2 + (j-(width-y))^2}$$

$$\begin{cases} 0 & des1 < d0 \ or \ des2 < d0 \\ 1 & otherwise \end{cases}$$

Results (including pictures):

Result of processing "lena.pgm":

Source Image:

Result after IDealNotchBandreject Filter:

Discussion:

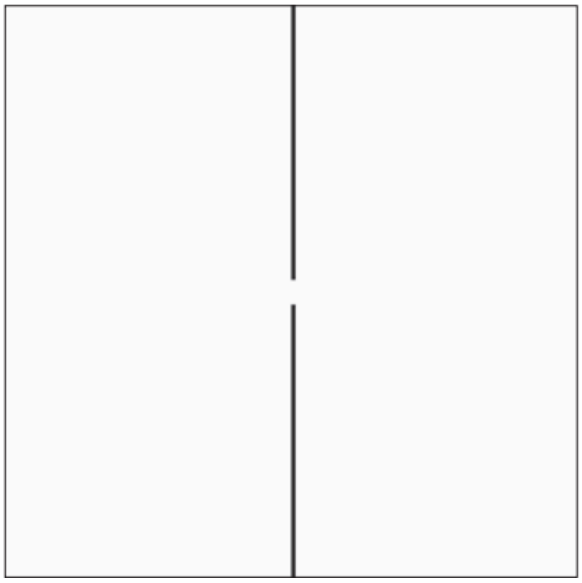
Most of the sinusoidal periodic noise can be removed by using an ideal notch filter

Codes:

```
Image *outimage;

int size = image->Height * image->Width;
int width = image->Width;
int height = image->Height;
struct _complex *src = (struct _complex *)malloc(sizeof(struct _complex) * size);

for (int i = 0; i < size; ++i) {···

fft(src, src, 1, height, width);

for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        double des1 = sqrt(pow(i - x, 2) + pow(j - y, 2));
        double des2 = sqrt(pow(i - (height - x), 2) + pow(j - (width - y), 2));
        // double p = -0.5 * pow((double)(pow(des, 2) - pow(c, 2)) / (des * w), 2);
        if (des1 < d0 || des2 < d0) {
            src[i * width + j].x = 0;
        }
    }
}

fft(src, src, -1, height, width);
outimage = CreateNewImage(image, height, width, (char *)"#Bandreject img");
// outimage->data = Normal(getResult(src, size), size, 255);

for (int i = 0; i < size; i++) {···

return (outimage);
```

4. Rectangle Notch Band Reject Image(LenaWithNoise)

Algorithm:



Results (including pictures):

Result of processing “LenaWithNoise.pgm”:

Source Image:

Result after RectNotchBandrejectFilter:

Discussion:

This is the sinusoidal noise. We use Rectangle Notch Band Reject Filter.

Codes:

```
int size = image->Height * image->Width;
int width = image->Width;
int height = image->Height;
struct _complex *src = (struct _complex *)malloc(sizeof(struct _complex) * size);        You, 12秒

for (int i = 0; i < size; ++i) {···

fft(src, src, 1, height, width);

for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        if (i < Rectheight && j < (width + Rectwidth) / 2 && j > (width - Rectwidth) / 2 || i > h
            src[i * width + j].x = 0;
            src[i * width + j].y = 0;
        }
    }
}

fft(src, src, -1, height, width);
outimage = CreateNewImage(image, height, width, (char *)"#RectNotchBandreject img");
// outimage->data = Normal(getResult(src, size), size, 255);

for (int i = 0; i < size; i++) {···

return (outimage);
```

5. Ad Med Filter(cameraWithNoise, lenaD1, lenaD2, lenaD3)

Algorithm:

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

The adaptive median-filtering algorithm uses two processing levels, denoted level A and level B, at each point (x, y):

Level A: If $z_{\min} < z_{\text{med}} < z_{\max}$, go to Level B
Else, increase the size of S_{xy}
If $S_{xy} \leq S_{\max}$, repeat level A
Else, output z_{med}.
Level B: If $z_{\min} < z_{xy} < z_{\max}$, output z_{xy}
Else output z_{med}.

Results (including pictures):

Result of processing "cameraWithNoise.pgm":

Source Image:

Result after AdMedFilter:

Result of processing “lenaD1.pgm”:

Source Image:

Result after AdMedFilter:

Result of processing “lenaD2.pgm”:

Source Image:

Result after AdMedFilter:

Result of processing “lenaD3.pgm”:

Source Image:

Result after AdMedFilter:

Discussion:

CameraWithNoise.pgm is added the salt-and-pepper noise. We use Adaptive Median Filter. Noise removal performance was similar to the median filter. However, the adaptive filter did a much better job of preserving sharpness and detail. The connector fingers are less distorted, and some other features that were either obscured or distorted beyond recognition by the median filter appear sharper and better. We can observe the results of filtering Lenad2.pGM using an adaptive median filter with a maximum window of 9.

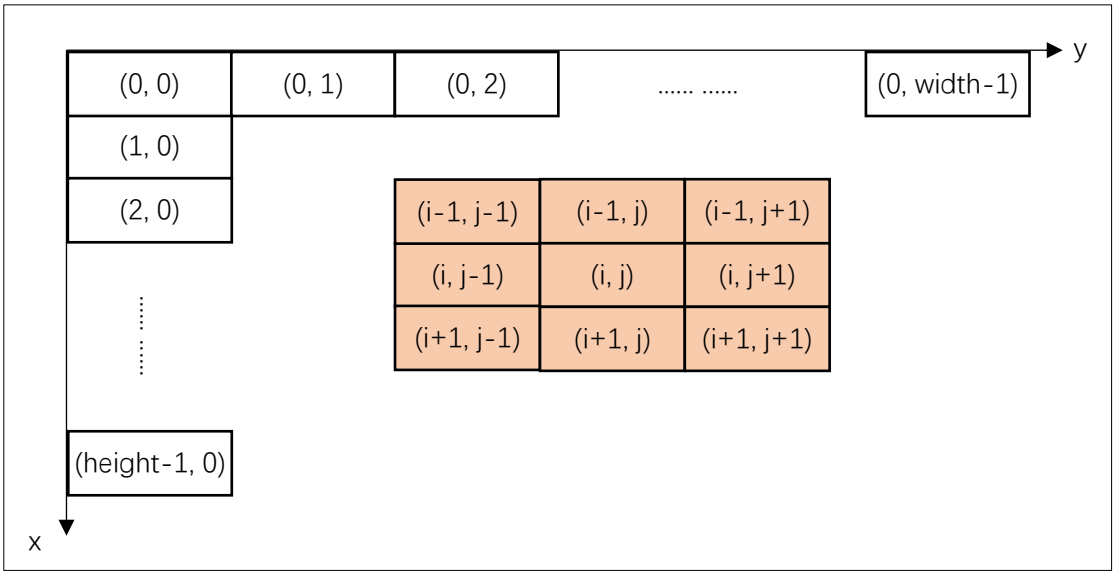
Codes:

```
memcpy(tempout, tempin, matrixWidth * matrixHeight);

int matrix[matrixWidth][matrixHeight];
memset(matrix, 0, sizeof(matrix));
for (int i = 0; i < matrixWidth; i++) {
    for (int j = 0; j < matrixHeight; j++, tempin++) {
        matrix[i][j] = *tempin;
    }
}

int pos = (smax - 1) / 2;
for (int i = pos; i < matrixWidth - pos; i++) {
    for (int j = pos; j < matrixHeight - pos; j++) {
        int size = n;
        while (size <= smax) {
            int lens = size * size;
            int arry[lens];
            int k = 0;
            for (int x = 0; x < size; x++) {…
            qsort(arry, lens, sizeof(int), cmp);
            int zmin = arry[0];
            int zmed = arry[(lens - 1) / 2];
            int zmax = arry[lens - 1];
            int z = matrix[i][j];
            int a1 = zmed - zmin;
            int a2 = zmed - zmax;
            if (a1 > 0 && a2 < 0) {
                int b1 = z - zmin, b2 = z - zmax;
                if (b1 > 0 && b2 < 0) {
                    tempout[i * matrixWidth + j] = z;
                } else {
                    tempout[i * matrixWidth + j] = zmed;
                }
                break;
            } else {
                size += 2;
                if (size > smax) {
                    tempout[i * matrixWidth + j] = z;
                    break;
                }
            }
        }
    }
```

6. Arithmetic Mean Filter (lenaD1, lenaD2, lenaD3)

Algorithm:

Algorithm:



$$outImage(i,j) = \sum_{k=-1}^{1}\sum_{t=-1}^{1} inImage(i+k, j+t)\ /9 \quad (1.1)$$

$$indexInData\ =\ i * width + j \quad (1.2)$$

$$outData[i * width + j] = \sum_{k=-1}^{1}\sum_{t=-1}^{1} inData[(i+k) * width + (j+t)]\ /9 \quad (1.3)$$

Results (including pictures):

Result of processing “lenaD1.pgm”:

Source Image:

Result after AverFilter:

Result of processing “lenaD2.pgm”:

Source Image:

Result after AverFilter:

Result of processing “lenaD3.pgm”:

Source Image:

Result after AverFilter:

Discussion:

we can conclude that differences between neighbor pixels become smaller by averaging, which leads to the smoother of image. The result images do turn out to be smoother. For example, the edges between objects are less obvious. This approach works better for lenaD3

Codes:

```
Image *AverFilterImage(Image *image, int number1, int number2) {
    unsigned char *tempin, *tempout;
    int zero1 = number1 - 1;
    int zero2 = number2 - 1;
    Image *outimage;
    outimage = CreateNewImage(image, image->Height, image->Width, (char *)"#Average Filter");
    tempin = image->data;
    tempout = outimage->data;

    int matrixWidth = image->Width + zero1;
    int matrixHeight = image->Height + zero2;

    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 0, sizeof(matrix));
    for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {···

    for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {
        for (int j = zero2 / 2; j < matrixHeight - zero2 / 2; j++) {
            int res = 0;
            for (int m = 0; m < number1; m++) {
                for (int n = 0; n < number2; n++) {
                    res += matrix[i - zero1 / 2 + m][j - zero2 / 2 + n];
                }
            }
            *tempout = res / (number1 * number2);
            tempout++;
        }
    }

    return (outimage);
}
```

7. Geometric Mean Filter (lenaD1, lenaD2, lenaD3)

Algorithm:

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

Results (including pictures):

Result of processing "lenaD1.pgm":

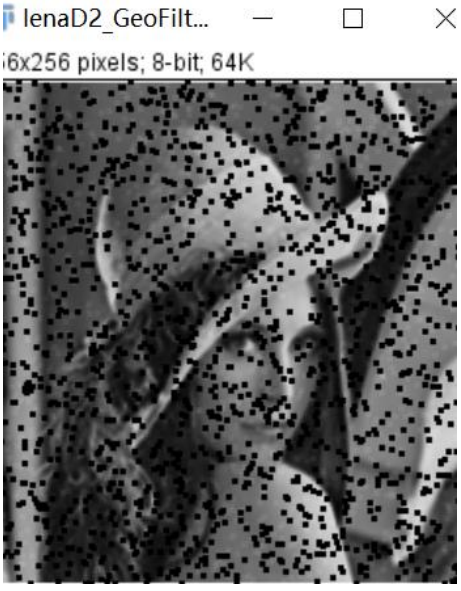
Source Image:

Result after GeoFilter:

Result of processing "lenaD2.pgm":

Source Image:

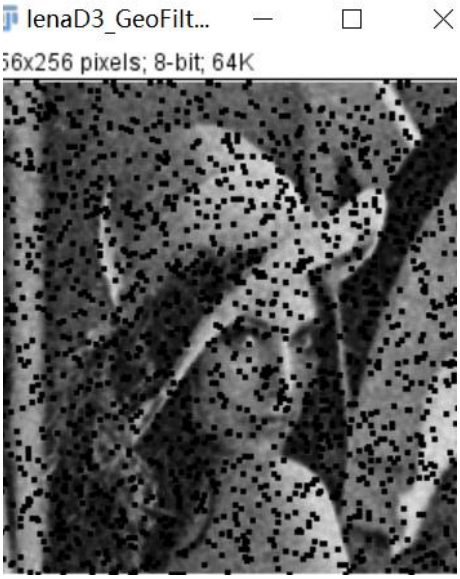
Result after GeoFilter:



Result of processing “lenaD3.pgm”:

Source Image:

Result after GeoFilter:



Discussion:

When processing gaussian noise, the effect is basically the same as arithmetic mean filter when the noise is small. However, if there is a lot of noise, the noise with a small gray value will be amplified, resulting in many black spots in the image and darkening the image. When processing salt and pepper noise, the pepper noise will be amplified, resulting in many black spots in the image, but the salt noise has a good filtering effect. This approach works better for lenaD1

Codes:

```
Image *GeoFilterImage(Image *image, int number1, int number2) {
    unsigned char *tempin, *tempout;
    int zero1 = number1 - 1;
    int zero2 = number2 - 1;
    Image *outimage;
    outimage = CreateNewImage(image, image->Height, image->Width, (char *)"#GeoFilter Filter");
    tempin = image->data;
    tempout = outimage->data;

    int matrixWidth = image->Width + zero1;
    int matrixHeight = image->Height + zero2;

    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 1, sizeof(matrix));
    for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {
        for (int j = zero2 / 2; j < matrixHeight - zero2 / 2; j++, tempin++) {
            matrix[i][j] = *tempin;
        }
    }

    for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {
        for (int j = zero2 / 2; j < matrixHeight - zero2 / 2; j++, tempout++) {
            double res = 1;
            for (int m = 0; m < number1; m++) {
                for (int n = 0; n < number2; n++) {
                    res *= pow(matrix[i - zero1 / 2 + m][j - zero2 / 2 + n], 1.0 / (number1 * number2
                }
            }
            if (res > 255)
                res = 255;
            if (res < 0)
                res = 0;
            *tempout = res;
            // printf("%d\n", *tempout);
        }
    }

    return (outimage);
}
```

8. Median Filter (lenaD1, lenaD2, lenaD3)

Algorithm:

Input: a unsigned char array that stores the source image data

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

For inImage(i, j):

Store its' 3 x 3 neighbor including itself in a unsigned char array **local**[9];

Find out the median of **local**;

Assign the median to outImage(i,j) => $outData[i * width + j] = findMedian(\boldsymbol{local}, 9)$

END

Results (including pictures):

Result of processing "lenaD1.pgm":

Source Image:

Result after MidFilterImage:

Result of processing "lenaD2.pgm":

Source Image:

Result after MidFilterImage:

Result of processing “lenaD3.pgm”:

Source Image:

Result after MidFilterImage:

Discussion:

The median filter also smooths the input image. But the result of median filter is sharper than that of average filter. Median filter simply substitutes the target pixel with the median of its 3 x 3 neighbor, that is, the operation does not operate all pixels in the neighbor. So the differences between pixels in the output image by median filter is larger than that by average filter. This approach works better for lenaD2

Codes:

```
Image *MidFilterImage(Image *image, int number1, int number2) {      You, 5天前 • lab6 …
    unsigned char *tempin, *tempout;
    int zero1 = number1 - 1;
    int zero2 = number2 - 1;
    Image *outimage;
    outimage = CreateNewImage(image, image->Height, image->Width, (char *)"#Mid Filter");
    tempin = image->data;
    tempout = outimage->data;

    int matrixWidth = image->Width + zero1;
    int matrixHeight = image->Height + zero2;

    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 0, sizeof(matrix));
    for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {···

    for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {
        for (int j = zero2 / 2; j < matrixHeight - zero2 / 2; j++) {
            int arry[number1 * number2], k = 0;
            for (int m = 0; m < number1; m++) {
                for (int n = 0; n < number2; n++, k++) {
                    arry[k] = matrix[i - zero1 / 2 + m][j - zero2 / 2 + n];
                }
            }
            qsort(arry, number1 * number2, sizeof(int), cmp);
            *tempout = arry[number1 * number2 / 2];
            tempout++;
        }
    }

    return (outimage);
}
```

9. Alpha-Trimmed Mean Filter (lenaD1, lenaD2, lenaD3)

Algorithm:

$$\hat{f}(x,y) = \frac{1}{mn - d} \sum_{(r,c) \in S_{xy}} g_R(r,c)$$

Results (including pictures):

Result of processing “lenaD1.pgm”:

Source Image:

Result after AlphaTrimmedMean:

Result of processing “lenaD2.pgm”:

Source Image:

Result after AlphaTrimmedMean:

Result of processing “lenaD3.pgm”:

Source Image:

Result after AlphaTrimmedMean:

Discussion:

The modified alpha mean filter is similar to the method used to evaluate a player's level by removing the highest and lowest scores. That is, the data within the filtering range are sorted, d data are removed from the order of large to small, d data are removed from the order of small to large, and the average value of the remaining data is calculated. Such filters are good at removing images that have been contaminated by salt and pepper noise along with other types of noise. This method has effect on many kinds of noise.

Codes:

```
Image *AlphaTrimmedMeanImage(Image *image, int number1, int number2, int d) {
    unsigned char *tempin, *tempout;
    int zero1 = number1 - 1;
    int zero2 = number2 - 1;
    Image *outimage;
    outimage = CreateNewImage(image, image->Height, image->Width, (char *)"#Alpha Trimmed Mean Filter
    tempin = image->data;
    tempout = outimage->data;

    int matrixWidth = image->Width + zero1;
    int matrixHeight = image->Height + zero2;

    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 1, sizeof(matrix));
    for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {···

    for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {      WYX, 4天前 • lab7 …
        for (int j = zero2 / 2; j < matrixHeight - zero2 / 2; j++) {
            int arry[number1 * number2], k = 0;
            for (int m = 0; m < number1; m++) {
                for (int n = 0; n < number2; n++, k++) {
                    arry[k] = matrix[i - zero1 / 2 + m][j - zero2 / 2 + n];
                }
            }
            qsort(arry, number1 * number2, sizeof(int), cmp);
            int res = 0;
            for (int k = d / 2; k < number1 * number2 - d / 2; k++) {
                res += arry[k];
            }
            res /= number1 * number2 - d;
            *tempout = res;
            tempout++;
        }
    }

    return (outimage);
}
```