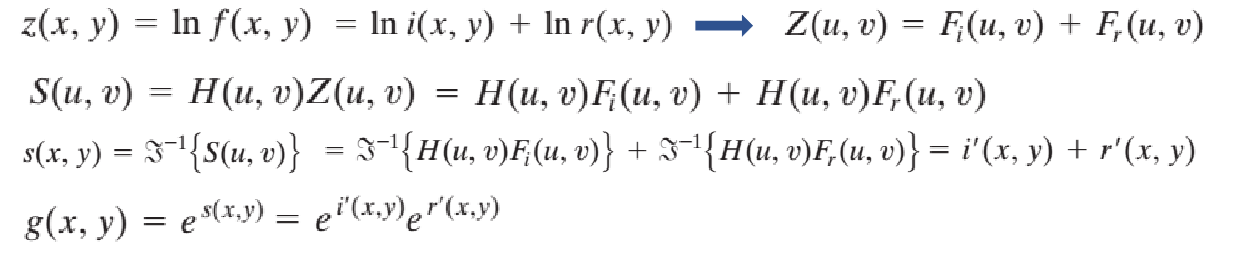
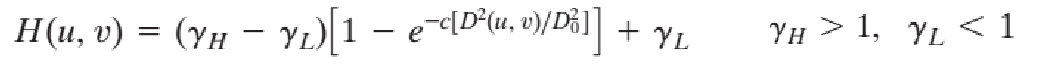
**Lab 7**

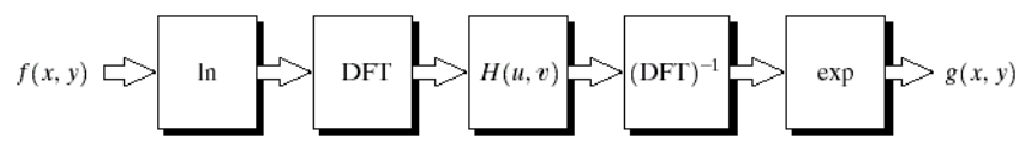
Use two images for each operation to do the following operations and write down their advantages and disadvantages and explain your results:

1. **Homomorphic filter (bridge, goldhill):**

**Algorithm:**







**Results (including pictures):**

Source: result:

Source: Result:

**Discussion:**

In the original image, many low gray features are hazy due to the high gray "hot spots" that dominate the dynamic range of the display.

In the processed image, "hot spot" is clearer, and more details can be seen in the image. For the dynamic range of the display, it is possible to allow the low gray level to become more visible by reducing the influence of the dominant illumination component (hot spot). Similarly, because the high frequency is enhanced by homomorphic filtering, the reflection component (edge information) of the image is significantly sharpened. The enhanced image has been greatly improved compared with the original image.

**Codes:**

Image\* homomorphicFilter(Image\* image) {

Image\* outimage;

int big, count = 1;

int Height = image->Height;

int Width = image->Width;

if (Height > Width) {

big = Height;

}

else {

big = Width;

}

while (count <= big) {

count \*= 2;

}

int size = count \* count;

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

struct \_complex\* dst = (struct \_complex\*)malloc(sizeof(struct \_complex) \* Width \* Height);

int x, y;

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

if (x >= Height || y >= Width) {

src[x \* count + y].x = 0.0;

src[x \* count + y].y = 0.0;

}

else {

src[x \* count + y].x = 1.0 \* image->data[x \* Width + y];

src[x \* count + y].y = 0.0;

}

}

}

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

src[x \* count + y].x += 1.0;

}

}

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

src[x \* count + y].x = log(src[x \* count + y].x);

}

}

fft(src, src, 1, count);

int D0 = 80;

double c = 1.0;

double L = 0.75;

double H = 2.0;

double D, HUV;

for (int u = 0; u < count; u++) {

for (int v = 0; v < count; v++) {

D = pow((double)u - count / 2, 2) + pow((double)v - count / 2, 2);

HUV = (H - L) \* (1 - pow(E, (-1.0 \* c) \* (D / (D0 \* D0)))) + L;

src[u \* count + v].x \*= HUV;

src[u \* count + v].y \*= HUV;

}

}

fft(src, src, -1, count);

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

src[x \* count + y].x = exp(src[x \* count + y].x);

}

}

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

src[x \* count + y].x -= 1.0;

}

}

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

if (x < Height && y < Width) {

dst[x \* Width + y].x = src[x \* count + y].x;

dst[x \* Width + y].y = src[x \* count + y].y;

}

}

}

outimage = CreateNewSizeImage(image, Width, Height, "homomorphicFilter");

outimage->data = Normal(getResult(dst, Width \* Height, 0), Width \* Height, 255, 0);

return(outimage);

}

1. **Sinusoidal noise (lena):**

**Algorithm:**



**Results (including pictures):**

Source: result:

**Discussion:**

After adding the noise, the image, the image is covered with horizontal and vertical stripes.

**Codes:**

Image\* sinusoidalNoise(Image\* image) {

Image\* temp, \*outimage;

int big, count = 1;

int Height = image->Height;

int Width = image->Width;

if (Height > Width) {

big = Height;

}

else {

big = Width;

}

while (count <= big) {

count \*= 2;

}

int size = count \* count;

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

struct \_complex\* src2 = (struct \_complex\*)malloc(sizeof(struct \_complex) \* size);

struct \_complex\* dst = (struct \_complex\*)malloc(sizeof(struct \_complex) \* Width \* Height);

int x, y;

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

if (x >= Height || y >= Width) {

src[x \* count + y].x = 0.0;

src[x \* count + y].y = 0.0;

src2[x \* count + y].x = 0.0;

src2[x \* count + y].y = 0.0;

}

else {

src[x \* count + y].x = 1.0 \* image->data[x \* Width + y] + 20 \* sin((double)y) + 20 \* sin((double)x);

src[x \* count + y].y = 0.0;

src2[x \* count + y].x = 1.0 \* image->data[x \* Width + y] + 20 \* sin((double)y) + 20 \* sin((double)x);

src2[x \* count + y].y = 0.0;

}

}

}

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

if (x < Height && y < Width) {

dst[x \* Width + y].x = src[x \* count + y].x;

dst[x \* Width + y].y = src[x \* count + y].y;

}

}

}

temp = CreateNewSizeImage(image, Width, Height, "sinusoidalNoise");

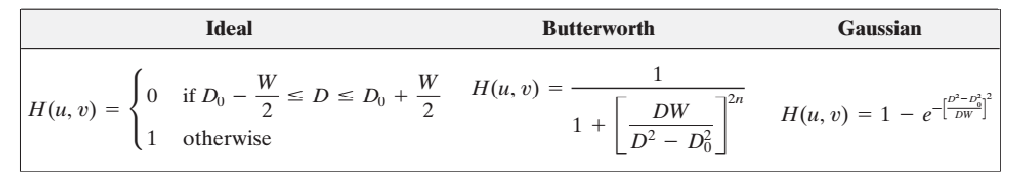
temp->data = Normal(getResult(dst, Height \* Width, 0), Height \* Width, 255, 0);

return(temp);

}

1. **Bandreject filter (lena, lenaWithNoise,** **cameraWithNoise):**

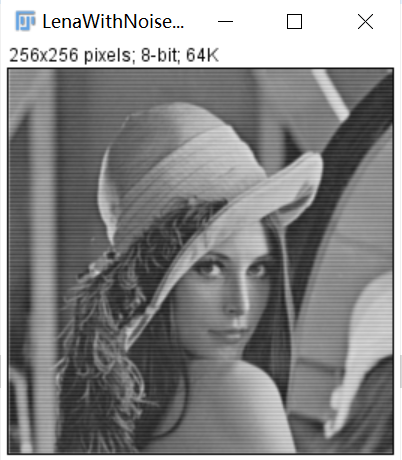
**Algorithm:**

**Results (including pictures):**

Source: result:

Source: result:

Source: result:

**Discussion:**

Here I just use ideal bandreject filter.

The filter can remove the sinusoidal noise but after processing, the image seems to be distorted.

For thelenaWithNoise.pgm, the noise of the horizontal line shape is removed, but the result is also distorted.

For thecameraWithNoise.pgm, the noise does not seem to be well processed and the image becomes blurred.

**Codes:**

Image\* IdealbandrejectFilter(Image\* image) {

Image\* temp, \*outimage;

int big, count = 1;

int Height = image->Height;

int Width = image->Width;

if (Height > Width) {

big = Height;

}

else {

big = Width;

}

while (count <= big) {

count \*= 2;

}

int size = count \* count;

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

struct \_complex\* src2 = (struct \_complex\*)malloc(sizeof(struct \_complex) \* size);

struct \_complex\* dst = (struct \_complex\*)malloc(sizeof(struct \_complex) \* Width \* Height);

int x, y;

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

if (x >= Height || y >= Width) {

src[x \* count + y].x = 0.0;

src[x \* count + y].y = 0.0;

}

else {

src[x \* count + y].x = 1.0 \* image->data[x \* Width + y];

src[x \* count + y].y = 0.0;

}

}

}

fft(src, src, 1, count);

int D0 = 180;

double W = 180;

double D, H;

for (int u = 0; u < count; u++) {

for (int v = 0; v < count; v++) {

D = sqrt(pow((double)u - count / 2, 2) + pow((double)v - count / 2, 2));

if ((D >= (D0 - W / 2)) && (D <= (D0 + (W / 2)))) {

src[u \* count + v].x = 0.0;

src[u \* count + v].y = 0.0;

}

}

}

fft(src, src, -1, count);

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

if (x < Height && y < Width) {

dst[x \* Width + y].x = src[x \* count + y].x;

dst[x \* Width + y].y = src[x \* count + y].y;

}

}

}

outimage = CreateNewSizeImage(image, Width, Height, "IdealbandrejectFilter");

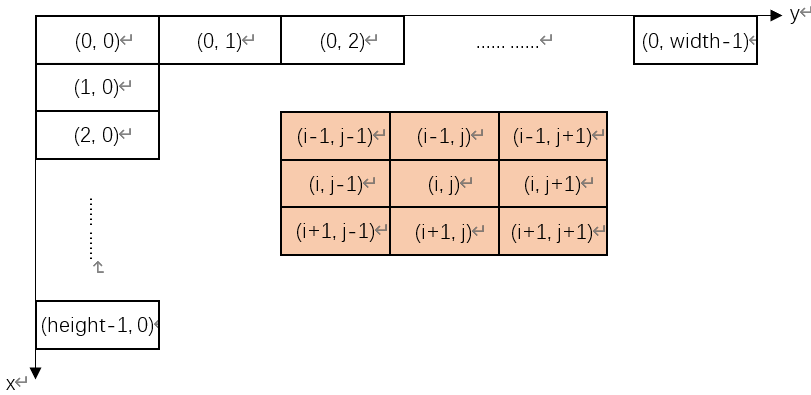
outimage->data = Normal(getResult(dst, Width \* Height, 0), Width \* Height, 255, 0);

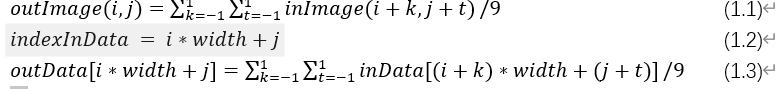
return(outimage);

}

1. **3×3 arithmetic mean filter(lenaD1,** **lenaD2):**

**Algorithm:**





**Results (including pictures):**

Source: result:

Source: result:

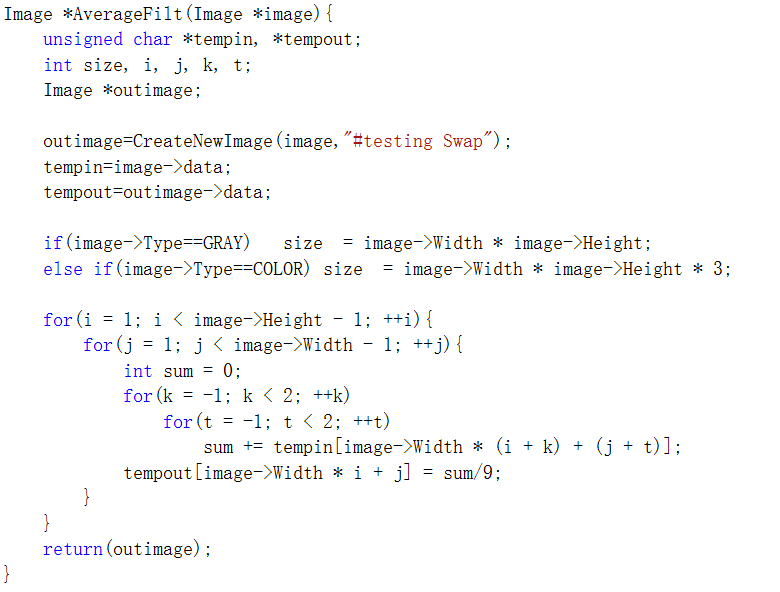
 

**Discussion:**

From equation (1.1), 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.

**Codes:**



1. **3×3 geometric mean filter, me (lenaD1,** **lenaD2):**

**Algorithm:**

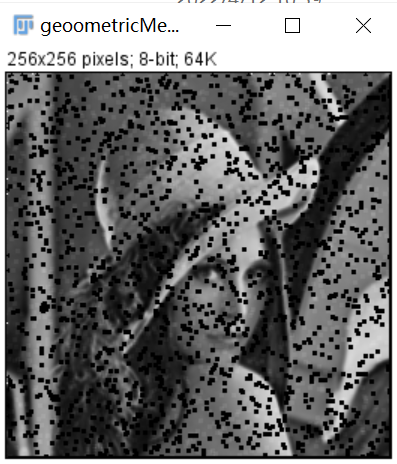


**Results (including pictures):**

Source: result:

Source: result:

**Discussion:**

The smoothing achieved by geometric mean filter can be compared with arithmetic mean filter, but less image details are lost in this process

**Codes:**

Image\* geoometricMeanFilter(Image\* image) {

Image\* temp, \*outimage;

int big, count = 1;

int Height = image->Height;

int Width = image->Width;

if (Height > Width) {

big = Height;

}

else {

big = Width;

}

while (count <= big) {

count \*= 2;

}

int size = count \* count;//PQ 4mn

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

struct \_complex\* src2 = (struct \_complex\*)malloc(sizeof(struct \_complex) \* size);

struct \_complex\* dst = (struct \_complex\*)malloc(sizeof(struct \_complex) \* Width \* Height);

int x, y;

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

if (x >= Height || y >= Width) {

src[x \* count + y].x = 0.0;

src[x \* count + y].y = 0.0;

src2[x \* count + y].x = 0.0;

src2[x \* count + y].y = 0.0;

}

else {

src[x \* count + y].x = 1.0 \* image->data[x \* Width + y];

src[x \* count + y].y = 0.0;

src2[x \* count + y].x = 1.0 \* image->data[x \* Width + y];

src2[x \* count + y].y = 0.0;

}

}

}

double arry[9];

for (int x = 0; x < count; x++) {

for (int y = 0; y < count; y++) {

if (x != 0 && x != count - 1 && y != 0 && y != count - 1) {

arry[0] = src2[(x - 1) \* count + y - 1].x;

arry[1] = src2[(x - 1) \* count + y].x;

arry[2] = src2[(x - 1) \* count + y + 1].x;

arry[3] = src2[x \* count + y - 1].x;

arry[4] = src2[x \* count + y].x;

arry[5] = src2[x \* count + y + 1].x;

arry[6] = src2[(x + 1) \* count + y - 1].x;

arry[7] = src2[(x + 1) \* count + y].x;

arry[8] = src2[(x + 1) \* count + y + 1].x;

src[x \* count + y].x = pow((arry[0] \* arry[1] \* arry[2] \* arry[3] \* arry[4] \* arry[5] \* arry[6] \* arry[7] \* arry[8]), (1.0 / 9.0));

}

}

}

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

if (x < Height && y < Width) {

dst[x \* Width + y].x = src[x \* count + y].x;

dst[x \* Width + y].y = src[x \* count + y].y;

}

}

}

temp = CreateNewSizeImage(image, Width, Height, "geoometricMeanFilter");

temp->data = Normal(getResult(dst, Height \* Width, 0), Height \* Width, 255, 0);

return(temp);

}

1. **3×3 median filter, (lenaD1,** **lenaD2):**

**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) =>

END

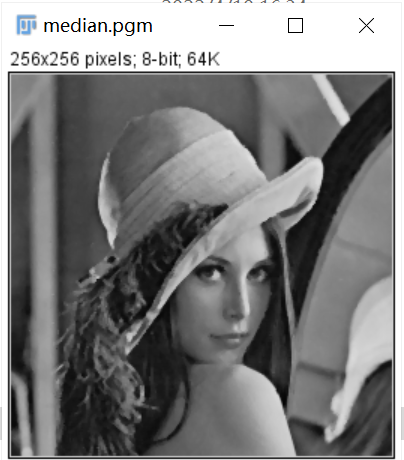
Function findMedian() use bubble sort to find the 5th smallest value of **local**.

**Results (including pictures):**

Source: result:

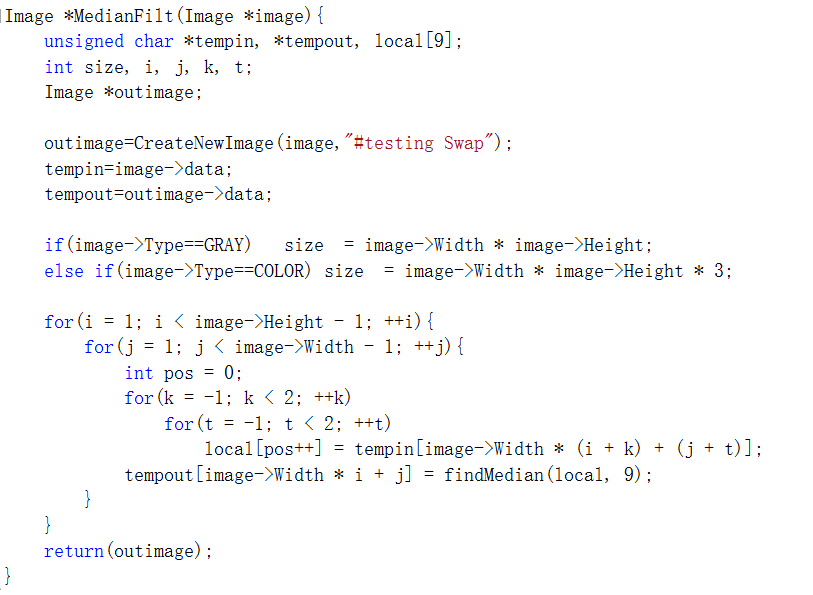
Source: result:

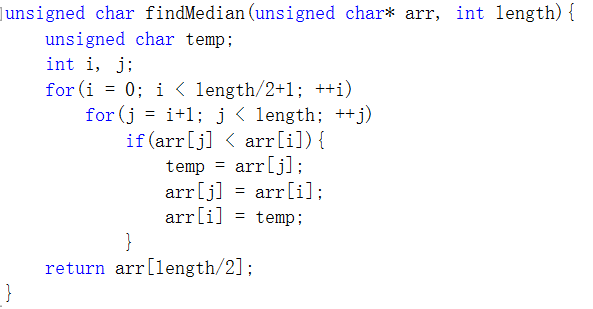
 

**Discussion:**

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.

**Codes:**

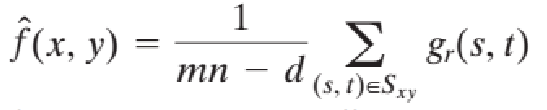




1. **Alpha-trimmed mean filter(lenaD1,** **lenaD2):**

**Algorithm:**

Delete d/2 lowest and d/ 2 highest intensity values of g(s, t) and average the remaining pixel values.



Source: result:

Source: result:

**Discussion:**

When d-0. it becomes mean filter. when d= mn-1. it becomes the median filter. When d becomes large, the filter approaches the median filter but still retains some smoothing capabilities.

It is useful to remove the combination of multiple types of noise.

**Codes:**

Image\* Alpha\_trimmedMeanFilter(Image\* image) {

Image\* temp, \*outimage;

int big, count = 1;

int Height = image->Height;

int Width = image->Width;

if (Height > Width) {

big = Height;

}

else {

big = Width;

}

while (count <= big) {

count \*= 2;

}

int size = count \* count;

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

struct \_complex\* src2 = (struct \_complex\*)malloc(sizeof(struct \_complex) \* size);

struct \_complex\* dst = (struct \_complex\*)malloc(sizeof(struct \_complex) \* Width \* Height);

int x, y;

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

if (x >= Height || y >= Width) {

src[x \* count + y].x = 0.0;

src[x \* count + y].y = 0.0;

src2[x \* count + y].x = 0.0;

src2[x \* count + y].y = 0.0;

}

else {

src[x \* count + y].x = 1.0 \* image->data[x \* Width + y];

src[x \* count + y].y = 0.0;

src2[x \* count + y].x = 1.0 \* image->data[x \* Width + y];

src2[x \* count + y].y = 0.0;

}

}

}

double arry[9], d = 4.0;

for (int x = 0; x < count; x++) {

for (int y = 0; y < count; y++) {

if (x != 0 && x != count - 1 && y != 0 && y != count - 1) {

arry[0] = src2[(x - 1) \* count + y - 1].x;

arry[1] = src2[(x - 1) \* count + y].x;

arry[2] = src2[(x - 1) \* count + y + 1].x;

arry[3] = src2[x \* count + y - 1].x;

arry[4] = src2[x \* count + y].x;

arry[5] = src2[x \* count + y + 1].x;

arry[6] = src2[(x + 1) \* count + y - 1].x;

arry[7] = src2[(x + 1) \* count + y].x;

arry[8] = src2[(x + 1) \* count + y + 1].x;

BubbleSort(arry, 9);

src[x \* count + y].x = (arry[0] + arry[1] + arry[2] + arry[3] + arry[4] + arry[5] + arry[6] + arry[7] + arry[8] - (arry[0] \* d / 2) - (arry[8] \* d / 2)) / (9 - d);

}

}

}

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

if (x < Height && y < Width) {

dst[x \* Width + y].x = src[x \* count + y].x;

dst[x \* Width + y].y = src[x \* count + y].y;

}

}

}

temp = CreateNewSizeImage(image, Width, Height, "Alpha\_trimmedMeanFilter");

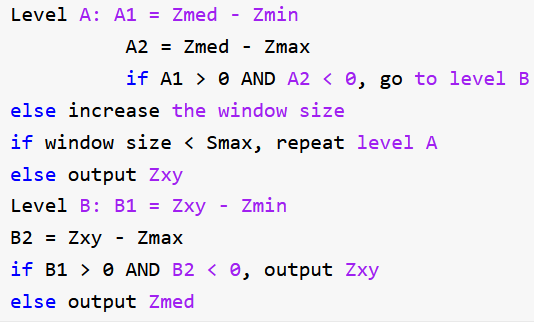
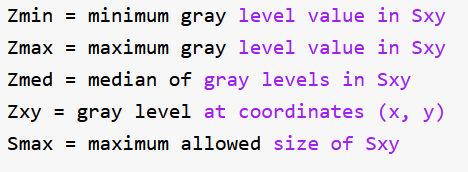
temp->data = Normal(getResult(dst, Height \* Width, 0), Height \* Width, 255, 0);

return(temp);

}

1. **adaptive median filter (lenaD1,** **lenaD2):**

**Algorithm:**

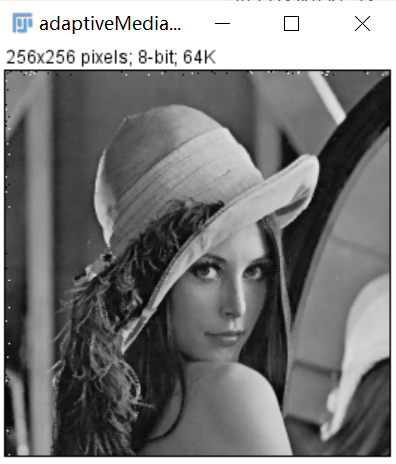


**Results (including pictures):**

Source: result:

Source: result:

**Discussion:**

The noise cancellation performance is similar to that of median filter. However, adaptive filters do better in maintaining clarity and detail. The connector has little distortion and uses a median value

Or distorted and unrecognizable features become sharper and clearer in Figure 5.14 (c).

Even considering the high level of noise, the performance of the adaptive algorithm has been very good. The maximum window size allowed to be selected depends on the application, but a reasonable initial value can first be estimated by experiments using standard median filters of various sizes. This will establish an intuitive benchmark based on the expected performance of the adaptive algorithm.

**Codes:**

Image\* adaptiveMedianFilter(Image\* image) {

Image\* temp, \*outimage;

int big, count = 1;

int Height = image->Height;

int Width = image->Width;

if (Height > Width) {

big = Height;

}

else {

big = Width;

}

while (count <= big) {

count \*= 2;

}

int size = count \* count;

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

struct \_complex\* src2 = (struct \_complex\*)malloc(sizeof(struct \_complex) \* size);

struct \_complex\* dst = (struct \_complex\*)malloc(sizeof(struct \_complex) \* Width \* Height);

int x, y;

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

if (x >= Height || y >= Width) {

src[x \* count + y].x = 0.0;

src[x \* count + y].y = 0.0;

src2[x \* count + y].x = 0.0;

src2[x \* count + y].y = 0.0;

}

else {

src[x \* count + y].x = 1.0 \* image->data[x \* Width + y];// +20 \* sin(y) + 20 \* sin(x);

src[x \* count + y].y = 0.0;

src2[x \* count + y].x = 1.0 \* image->data[x \* Width + y];// +20 \* sin(y) + 20 \* sin(x);

src2[x \* count + y].y = 0.0;

}

}

}

adaptive(src, src2, count, count, 49);

for (x = 0; x < count; x++) {

for (y = 0; y < count; y++) {

if (x < Height && y < Width) {

dst[x \* Width + y].x = src[x \* count + y].x;

dst[x \* Width + y].y = src[x \* count + y].y;

}

}

}

temp = CreateNewSizeImage(image, Width, Height, "adaptiveMedianFilter");

temp->data = Normal(getResult(dst, Height \* Width, 0), Height \* Width, 255, 0);

return(temp);

}

void adaptive(struct \_complex\* out, struct \_complex\* in, int Height, int Width, int MaxSize) {

int count = floor(sqrt((double)MaxSize)) + 1;

int temp = (count - 1) / 2;

for (int x = 0; x < Height; x++) {

for (int y = 0; y < Width; y++) {

if (x > temp - 1 && x < Height - temp && y >temp - 1 && y < Width - temp) {

ad\_A(out, in, Width, x, y, 1, MaxSize);

}

}

}

}

void ad\_A(struct \_complex\* out, struct \_complex\* in, int Width, int X, int Y, int curSize, int MaxSize) {

int min, max, med, xy;

int A1, A2;

int temp = (curSize - 1) / 2;

int count = 0;

double\* arry = (double\*)malloc(sizeof(double)\* curSize \* curSize);

for (int i = X - temp; i <= X + temp; i++) {

for (int n = Y - temp; n <= Y + temp; n++) {

arry[count] = in[i \* Width + n].x;

count++;

}

}

BubbleSort(arry, curSize \* curSize);

min = arry[0];

max = arry[curSize \* curSize - 1];

med = arry[(curSize \* curSize - 1) / 2];

xy = in[X \* Width + Y].x;

A1 = med - min;

A2 = med - max;

if (A1 > 0 && A2 < 0) {

ad\_B(out, Width, X, Y, min, max, xy, med);

}

else {

curSize += 2;

if (curSize \* curSize > MaxSize) {

out[X \* Width + Y].x = med;

return;

}

else {

ad\_A(out, in, Width, X, Y, curSize, MaxSize);

return;

}

}

}

void ad\_B(struct \_complex\* out, int Width, int X, int Y, int min, int max, int xy, int med) {

int B1, B2;

B1 = xy - min;

B2 = xy - max;

if (B1 > 0 && B2 < 0) {

return;

}

else {

out[X \* Width + Y].x = med;

return;

}

}