**Lab 6**

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

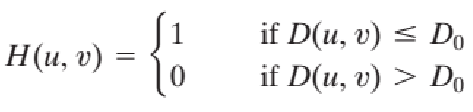
1. **IDLPF (lena, camera):**

**Algorithm:**

Fill the original image with the necessary number of zeros to form an image of size P × Q (where P = M, Q = 2N).

Fourier transform the filled image and move it to the center of the transform.

The filter is generated and multiplied by the real part and imaginary part obtained by Fourier transform respectively.

(This is the filter)

D = sqrt(pow((double)u - 2 \* Height / 2, 2) + pow((double)v - 2 \* Width / 2, 2));

if (D > D0)

src[u \* 2 \* Width + v].x \*= 0.0;

src[u \* 2 \* Width + v].y \*= 0.0;

The real part and imaginary part obtained are inversely the real part after Fourier and moved from the center to the four corners.

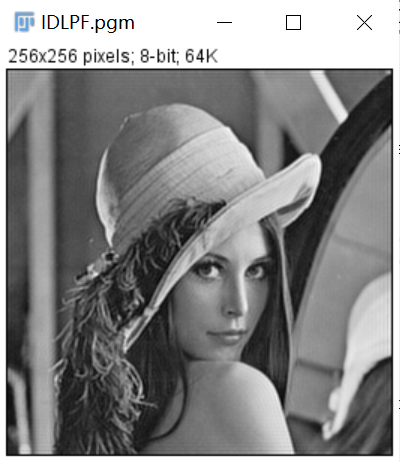
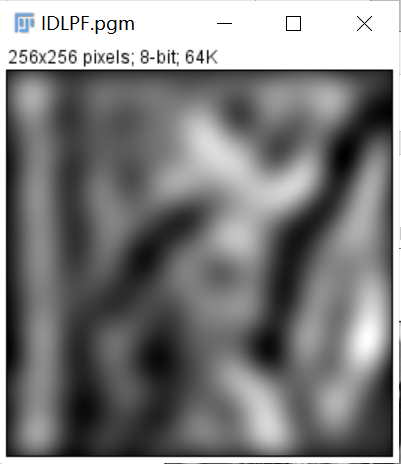
Take upper left quadrant M × N region to get the final result.

**Results (including pictures):**

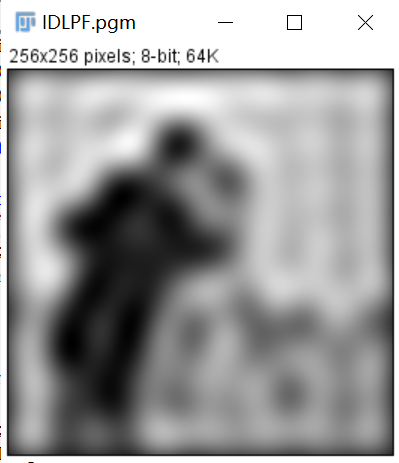
Source: result(D0=60):

Result(D0=200): Result(D0=15):

Source: Result(D0=15):

(Here I do not show the different result of camera.pgm in different D0 because I have showed the contrast in lean.pgm.)

**Discussion:**

The smaller D0 is, the more blurred the visual feeling it brings intuitively. The larger D0, the smaller the power of filter removal, and the closer the spatial sinc function is to an impact that will not cause blur when convoluted with the image.

**Codes:**

Image \*IDLPF(Image \*image){

Image\* temp, \*outimage;

int Height = image->Height;

int Width = image->Width;

int size = 2 \* Width \* 2 \* Height;

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 < 2 \* Height; x++) {

for (y = 0; y < 2 \* Width; y++) {

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

src[x \* 2 \* Width + y].x = 0.0;

src[x \* 2 \* Width + y].y = 0.0;

}

else {

src[x \* 2 \* Width + y].x = 1.0 \* image->data[x \* Width + y];

src[x \* 2 \* Width + y].y = 0.0;

}

}

}

fft(src, src, 1, 2 \* Width);

int D0 = 60;

double D;

for (int u = 0; u < 2\*Height; u++) {

for (int v = 0; v < 2 \* Width; v++) {

D = sqrt(pow((double)u - 2 \* Height / 2, 2) + pow((double)v - 2 \* Width / 2, 2));

if (D > D0) {

src[u \* 2 \* Width + v].x \*= 0.0;

src[u \* 2 \* Width + v].y \*= 0.0;

}

}

}

fft(src, src, -1, 2 \* Width);

for (x = 0; x < 2 \* Width; x++) {

for (y = 0; y < 2 \* Width; y++) {

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

dst[x \* Width + y].x = src[x \* 2 \* Width + y].x;

dst[x \* Width + y].y = src[x \* 2 \* Width + y].y;

}

}

}

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

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

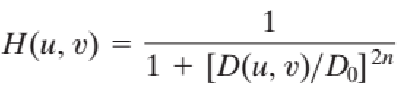
return(outimage);

}

1. **BLPF (lena, camera):**

**Algorithm:**

The processing steps is just like IDLPF, the only difference is about the filter whose algorithm is showed below:



D = sqrt(pow((double)u - 2 \* Height / 2, 2) + pow((double)v - 2 \* Width / 2, 2));

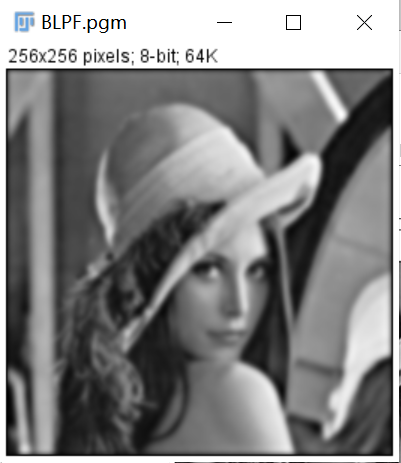
H = 1 / (1.0 + pow(D / D0, 2 \* n));

src[u \* 2 \* Width + v].x \*= H;

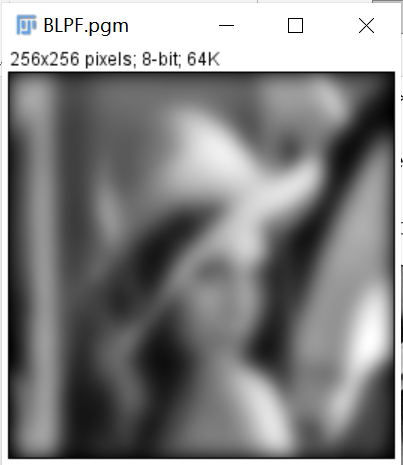
src[u \* 2 \* Width + v].y \*= H;

**Results (including pictures):**

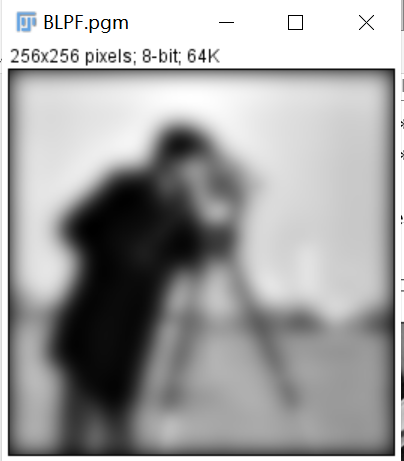
Source: result(D0=60):

Result(D0=200): Result(D0=15):

Source: Result(D0=15):

(Here I do not show the different result of camera.pgm in different D0 because I have showed the contrast in lean.pgm.)

**Discussion:**

The smaller D0 is, the more blurred the visual feeling it brings intuitively. Compared with IDLPF, BLPF processed images have no visible ringing phenomenon

**Codes:**

Image \*BLPF(Image \*image){

Image \*outimage;

int Height = image->Height;

int Width = image->Width;

int size = 2 \* Width \* 2 \* Height;

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 < 2 \* Height; x++) {

for (y = 0; y < 2 \* Width; y++) {

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

src[x \* 2 \* Width + y].x = 0.0;

src[x \* 2 \* Width + y].y = 0.0;

}

else {

src[x \* 2 \* Width + y].x = 1.0 \* image->data[x \* Width + y];

src[x \* 2 \* Width + y].y = 0.0;

}

}

}

fft(src, src, 1, 2 \* Width);

int D0 = 60,n=2;

double D, H;

for (int u = 0; u < 2\*Height; u++) {

for (int v = 0; v < 2\*Width; v++) {

D = sqrt(pow((double)u - 2 \* Height / 2, 2) + pow((double)v - 2 \* Width / 2, 2));

H = 1 / (1.0 + pow(D / D0, 2 \* n));

src[u \* 2 \* Width + v].x \*= H;

src[u \* 2 \* Width + v].y \*= H;

}

}

fft(src, src, -1, 2 \* Width);

for (x = 0; x < 2 \* Width; x++) {

for (y = 0; y < 2 \* Width; y++) {

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

dst[x \* Width + y].x = src[x \* 2 \* Width + y].x;

dst[x \* Width + y].y = src[x \* 2 \* Width + y].y;

}

}

}

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

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

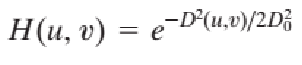
return(outimage);

}

1. **GLPF (lena, camera):**

**Algorithm:**

The processing steps is just like IDLPF, the only difference is about the filter whose algorithm is showed below:



D = pow((double)u - 2\*Height / 2, 2) + pow((double)v - 2 \* Width / 2, 2);

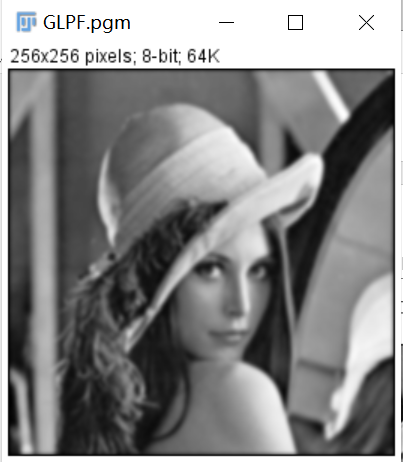
H = pow(E, (-1.0 \* D) / (2 \* D0 \* D0));

src[u \* 2 \* Width + v].x \*= H;

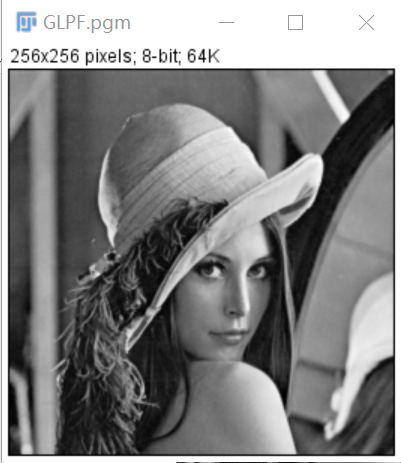
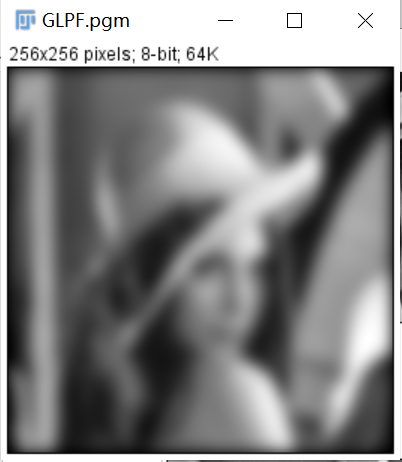
src[u \* 2 \* Width + v].y \*= H;

**Results (including pictures):**

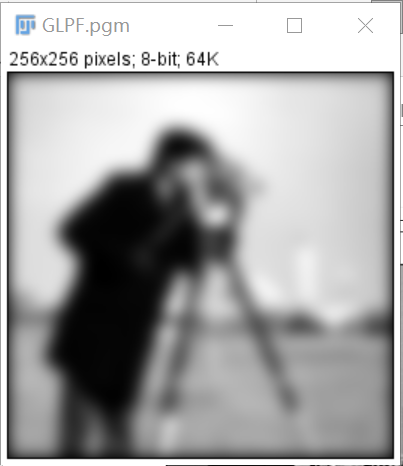
Source: result(D0=60):

Result(D0=200): Result(D0=15):

Source: Result(D0=15):

(Here I do not show the different result of camera.pgm in different D0 because I have showed the contrast in lean.pgm.)

**Discussion:**

The smaller D0 is, the more blurred the visual feeling it brings intuitively.

GLPF is blurring but no ringing and slightly sharper than BLPF.

No ringing artifacts of GLPF.

**Codes:**

Image \*GLPF(Image \*image){

Image \*outimage;

int Height = image->Height;

int Width = image->Width;

int size = 2 \* Width \* 2 \* Height;

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 < 2 \* Height; x++) {

for (y = 0; y < 2 \* Width; y++) {

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

src[x \* 2 \* Width + y].x = 0.0;

src[x \* 2 \* Width + y].y = 0.0;

}

else {

src[x \* 2 \* Width + y].x = 1.0 \* image->data[x \* Width + y];

src[x \* 2 \* Width + y].y = 0.0;

}

}

}

fft(src, src, 1, 2 \* Width);

int D0 = 60;

double D, H;

for (int u = 0; u < 2\*Height; u++) {

for (int v = 0; v < 2\*Width; v++) {

D = pow((double)u - 2\*Height / 2, 2) + pow((double)v - 2 \* Width / 2, 2);

H = pow(E, (-1.0 \* D) / (2 \* D0 \* D0));

src[u \* 2 \* Width + v].x \*= H;

src[u \* 2 \* Width + v].y \*= H;

}

}

fft(src, src, -1, 2 \* Width);

for (x = 0; x < 2 \* Width; x++) {

for (y = 0; y < 2 \* Width; y++) {

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

dst[x \* Width + y].x = src[x \* 2 \* Width + y].x;

dst[x \* Width + y].y = src[x \* 2 \* Width + y].y;

}

}

}

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

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

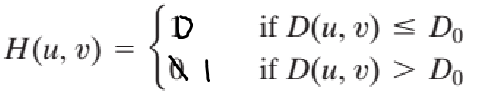
return(outimage);

}

1. **IHPF (fingerprint1,fingerprint2):**

**Algorithm:**

The processing steps is just like IDLPF, the only difference is about the filter whose algorithm is showed below:

(This is the filter)

D = sqrt(pow((double)u - 2 \* Height / 2, 2) + pow((double)v - 2 \* Width / 2, 2));

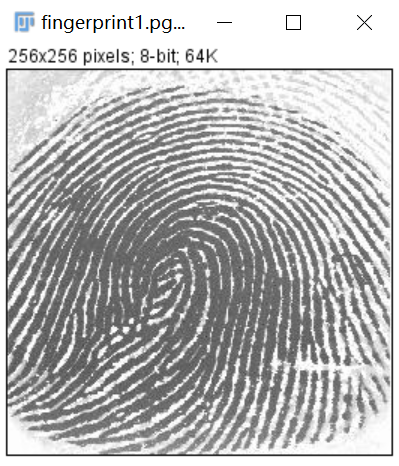
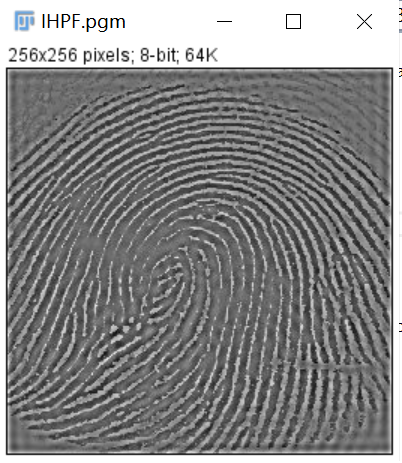
if (D <= D0)

src[u \* 2 \* Width + v].x \*= 0.0;

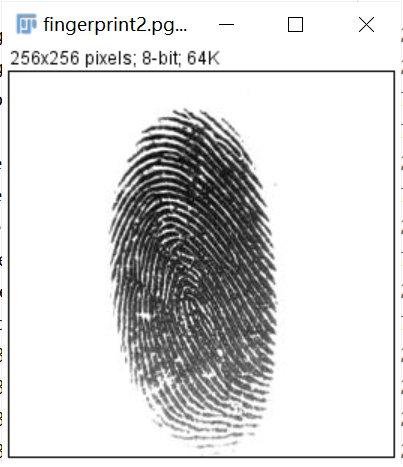
src[u \* 2 \* Width + v].y \*= 0.0;

**Results (including pictures):**

Source: result(D0=60):

Source: result(D0=60):

**Discussion:**

IHPF has the same ringing property as ILPF. The larger the D0, the clearer the edge and the smaller the distortion.

**Codes:**

Image \*IHPF(Image \*image){

Image\* temp, \*outimage;

int Height = image->Height;

int Width = image->Width;

int size = 2 \* Width \* 2 \* Height;

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 < 2 \* Height; x++) {

for (y = 0; y < 2 \* Width; y++) {

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

src[x \* 2 \* Width + y].x = 0.0;

src[x \* 2 \* Width + y].y = 0.0;

}

else {

src[x \* 2 \* Width + y].x = 1.0 \* image->data[x \* Width + y];

src[x \* 2 \* Width + y].y = 0.0;

}

}

}

fft(src, src, 1, 2 \* Width);

int D0 = 60;

double D;

for (int u = 0; u < 2 \* Height; u++) {

for (int v = 0; v < 2 \* Width; v++) {

D = sqrt(pow((double)u - 2 \* Height / 2, 2) + pow((double)v - 2 \* Width / 2, 2));

if (D <= D0) {

src[u \* 2 \* Width + v].x \*= 0.0;

src[u \* 2 \* Width + v].y \*= 0.0;

}

}

}

fft(src, src, -1, 2 \* Width);

for (x = 0; x < 2 \* Width; x++) {

for (y = 0; y < 2 \* Width; y++) {

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

dst[x \* Width + y].x = src[x \* 2 \* Width + y].x;

dst[x \* Width + y].y = src[x \* 2 \* Width + y].y;

}

}

}

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

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

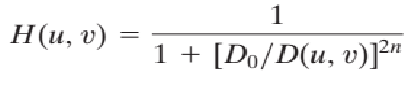
return(outimage);

}

1. **BHPF (fingerprint1,fingerprint2):**

**Algorithm:**

The processing steps is just like IDLPF, the only difference is about the filter whose algorithm is showed below:



D = sqrt(pow((double)u - 2 \* Height / 2, 2) + pow((double)v - 2 \* Width / 2, 2));

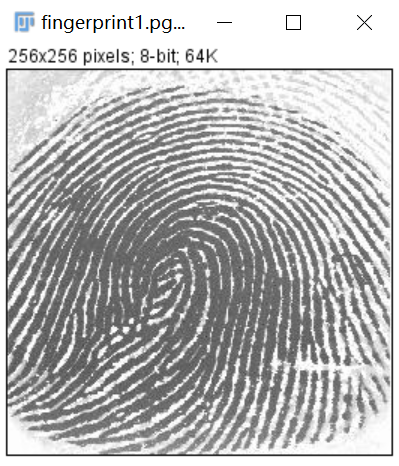
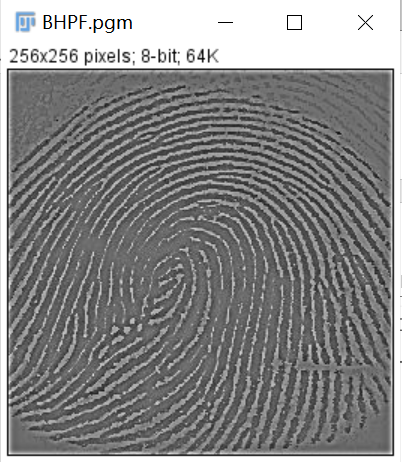
H = 1 / (1.0 + pow(D0/D, 2 \* n));

src[u \* 2 \* Width + v].x \*= H;

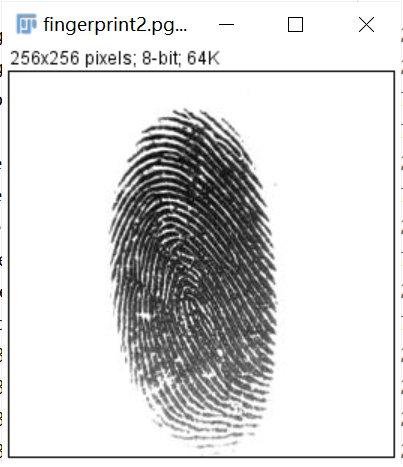
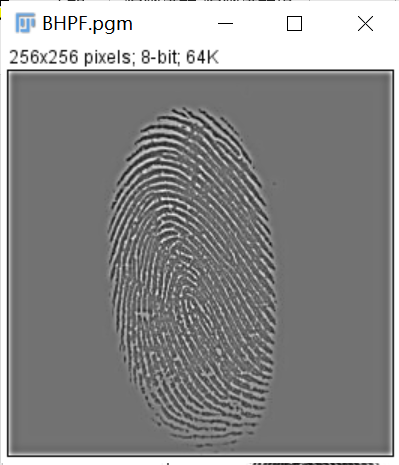
src[u \* 2 \* Width + v].y \*= H;

**Results (including pictures):**

Source: result(D0=60):

Source: result(D0=60):

**Discussion:**

BHPF is smoother than IHPF. Similarly, the larger the D0, the clearer the edge and the smaller the distortion.

**Codes:**

Image \*BHPF(Image \*image){

Image \*outimage;

int Height = image->Height;

int Width = image->Width;

int size = 2 \* Width \* 2 \* Height;

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 < 2 \* Height; x++) {

for (y = 0; y < 2 \* Width; y++) {

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

src[x \* 2 \* Width + y].x = 0.0;

src[x \* 2 \* Width + y].y = 0.0;

}

else {

src[x \* 2 \* Width + y].x = 1.0 \* image->data[x \* Width + y];

src[x \* 2 \* Width + y].y = 0.0;

}

}

}

fft(src, src, 1, 2 \* Width);

int D0 = 60, n = 2;

double D, H;

for (int u = 0; u < 2 \* Height; u++) {

for (int v = 0; v < 2 \* Width; v++) {

D = sqrt(pow((double)u - 2 \* Height / 2, 2) + pow((double)v - 2 \* Width / 2, 2));

H = 1 / (1.0 + pow(D0/D, 2 \* n));

src[u \* 2 \* Width + v].x \*= H;

src[u \* 2 \* Width + v].y \*= H;

}

}

fft(src, src, -1, 2 \* Width);

for (x = 0; x < 2 \* Width; x++) {

for (y = 0; y < 2 \* Width; y++) {

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

dst[x \* Width + y].x = src[x \* 2 \* Width + y].x;

dst[x \* Width + y].y = src[x \* 2 \* Width + y].y;

}

}

}

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

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

return(outimage);

}

1. **GHPF (fingerprint1,fingerprint2):**

**Algorithm:**

The processing steps is just like IDLPF, the only difference is about the filter whose algorithm is showed below:



D = pow((double)u - 2\*Height / 2, 2) + pow((double)v - 2 \* Width / 2, 2);

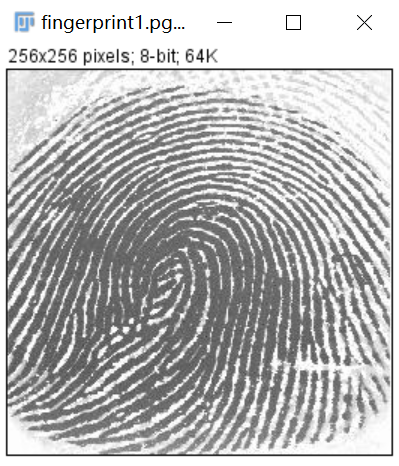
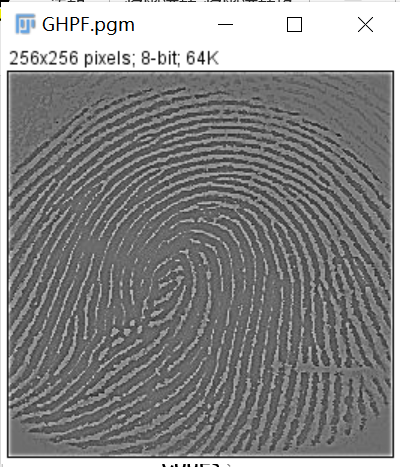
H = 1 - pow(E, (-1.0 \* D) / (2 \* D0 \* D0));

src[u \* 2 \* Width + v].x \*= H;

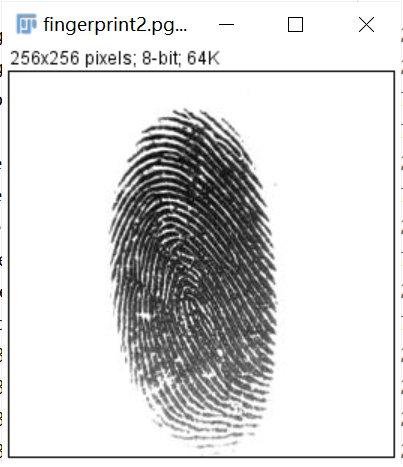
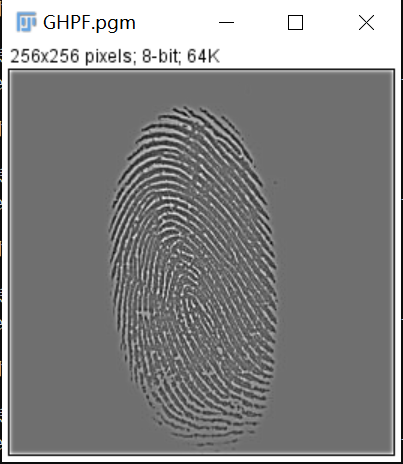
src[u \* 2 \* Width + v].y \*= H;

**Results (including pictures):**

Source: result(D0=60):

Source: result(D0=60):

**Discussion:**

The result is smoother than that of the first two filters. Even if Gaussian filter is used for filtering small objects and thin lines, the result is clear.

**Codes:**

Image \*GHPF(Image \*image){

Image \*outimage;

int Height = image->Height;

int Width = image->Width;

int size = 2 \* Width \* 2 \* Height;

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 < 2 \* Height; x++) {

for (y = 0; y < 2 \* Width; y++) {

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

src[x \* 2 \* Width + y].x = 0.0;

src[x \* 2 \* Width + y].y = 0.0;

}

else {

src[x \* 2 \* Width + y].x = 1.0 \* image->data[x \* Width + y];

src[x \* 2 \* Width + y].y = 0.0;

}

}

}

fft(src, src, 1, 2 \* Width);

int D0 = 60;

double D, H;

for (int u = 0; u < 2 \* Height; u++) {

for (int v = 0; v < 2 \* Width; v++) {

D = pow((double)u - 2 \* Height / 2, 2) + pow((double)v - 2 \* Width / 2, 2);

H = 1-pow(E, (-1.0 \* D) / (2 \* D0 \* D0));

src[u \* 2 \* Width + v].x \*= H;

src[u \* 2 \* Width + v].y \*= H;

}

}

fft(src, src, -1, 2 \* Width);

for (x = 0; x < 2 \* Width; x++) {

for (y = 0; y < 2 \* Width; y++) {

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

dst[x \* Width + y].x = src[x \* 2 \* Width + y].x;

dst[x \* Width + y].y = src[x \* 2 \* Width + y].y;

}

}

}

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

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

return(outimage);

}

1. **Thresholding(fingerprint1,fingerprint2):**

**Algorithm:**

Set all negative values to black(0) and all positive values to white(255).

if (src[u \* Width + v].x > 0) src[u \* Width + v].x = 255;

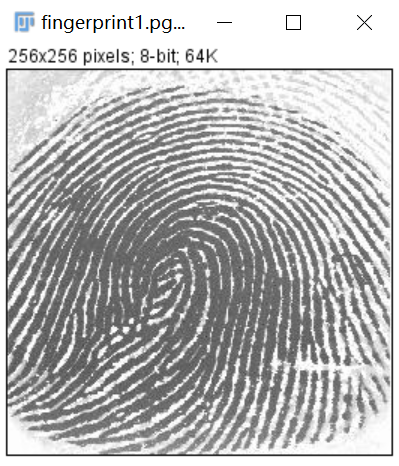
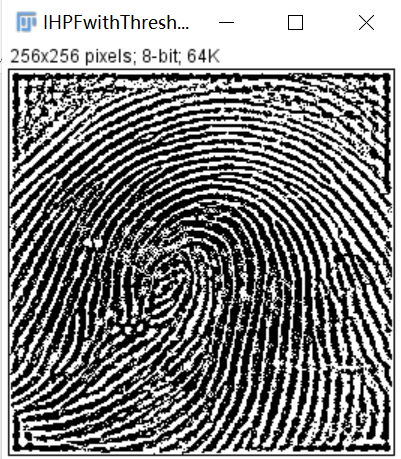
if (src[u \* Width + v].x < 0) src[u \* Width + v].x = 0;

if (src[u \* Width + v].y > 0) src[u \* Width + v].y = 255

if (src[u \* Width + v].y < 0) src[u \* Width + v].y = 0;

**Results (including pictures):**

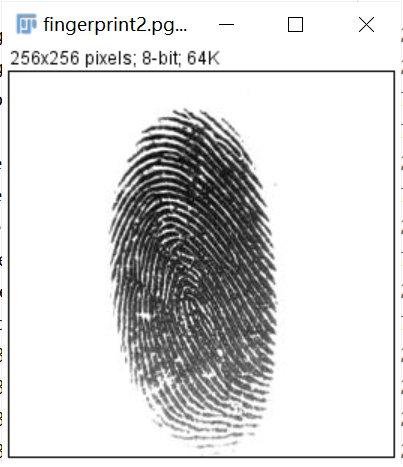
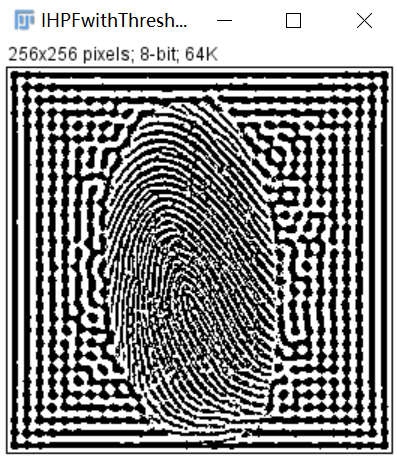
Source: result(IHPF with thresholding):

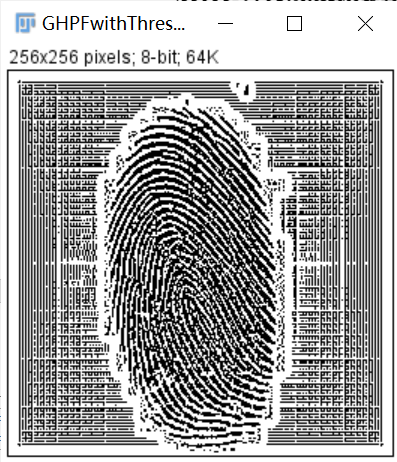
Result(BHPF with thresholding): Result(GHPF with thresholding):

Source: result(IHPF with thresholding):

result(BHPF with thresholding): result(GHPF with thresholding):

**Discussion:**

After threshold processing, the ridge of fingerprint image becomes clear and the pollution has been significantly reduced.

**Codes:**

void thresholding(struct \_complex\* src, int Height, int Width) {

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

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

if (src[u \* Width + v].x > 0) {

src[u \* Width + v].x = 255;

}

if (src[u \* Width + v].x < 0) {

src[u \* Width + v].x = 0;

}

if (src[u \* Width + v].y > 0) {

src[u \* Width + v].y = 255;

}

if (src[u \* Width + v].y < 0) {

src[u \* Width + v].y = 0;

}

}

}

return;

}