Lab 6

1 . Filter lena.pgm and camera.pgm using IDLPF, BLPF, GLPF and compare the image qualities with different cutoff frequencies

2. Use HPF and thresholding to sharpen fingerprint1.pgm and fingerprint2.pgm

Algorithm:

1. ILPF

   Input: the complex number out of the image obtained after fft, the cutoff frequency d

   Output: the complex number of the image

   The formula is $H(u,v) = \begin{cases} 1, & if\ D(u,v) \le D_0 \\ 0, & if\ D(u,v) > D_0 \end{cases}$,where D(u,v) is the distance between the current pixel and the center of the image, D_0 is the cutoff frequency

```
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 > d) {
            out[i][j].x = 0;
            out[i][j].y = 0;
        }
    }
}
```

   return out

2. BLPF

   Input: the complex number out of the image obtained after fft, the cutoff frequency d, the order n

   Output: the complex number of the image

   The formula is $H(u,v) = \frac{1}{1+[\frac{D(u,v)}{D_0}]^{2n}}$, where D(u,v) is the distance between the current pixel and the center of the image, D_0 is the cutoff frequency, n is the given order.

```
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][j].x *= 1 / (1 + pow(dist / radius, 2 * order));
        out[i][j].y *= 1 / (1 + pow(dist / radius, 2 * order));
    }
}
```

   Return out

3. GLPF

   Input: the complex number out of the image obtained after fft, the cutoff frequency d, the order n

   Output: the complex number of the image

   The formula is $H(u,v) = e^{-\frac{D^2(u,v)}{2{D_0}^2}}$

```
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 *= exp((-pow(dist / radius, 2 * order) / (2 * pow(order, 2))));
            out[i * image->Width + j].y *= exp((-pow(dist / radius, 2 * order) / (2 * pow(order, 2))));
        }
    }
    return out
```

After conducting the filter operation, use the inverse Fourier transform to show the image.

4. The formula of high pass filter is $H_{HP}(u,v) = 1 - H_{LP}(u,v)$, the following conduction is same as the low pass filter.
5. Threshold filter

   Input: array of image f, threshold t

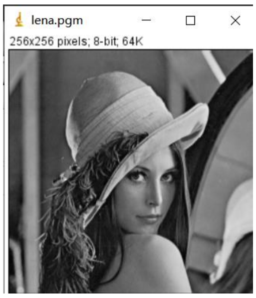
   Output: array of output image

```
    for (i = 0; i <image size; i++) {
        if (in[i] > threshold) out[i] = 255;
        else out[i] =in[i];
    }
return out
```

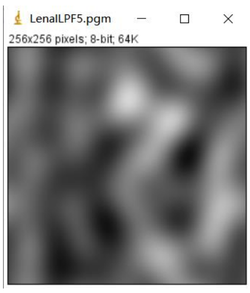
Results (compare the results with the original image):

Paste the result images and the original ones.

1. Lena



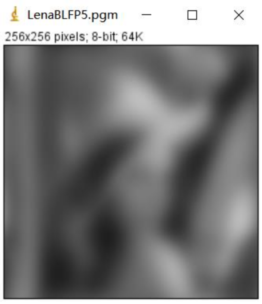
Source



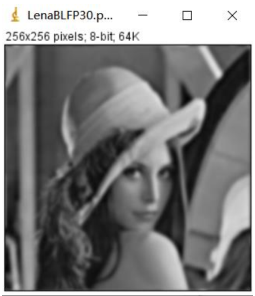
ILPF, 5

ILPF, 30

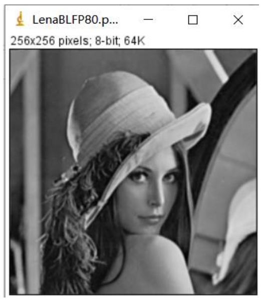
ILPF, 80



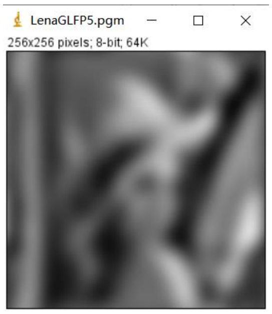
BLPF, 5, n=2



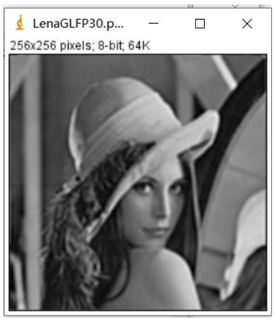
BLPF, 30, n=2



BLPF, 80, n=2



GLFP, 5, n=2

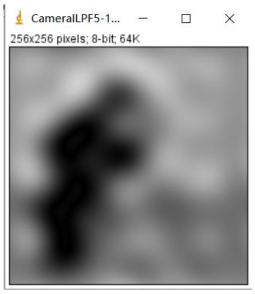


GLPF, 30, n=2

GLPF, 80, n=2

2. Camera

Source

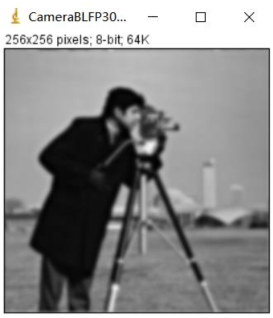


ILPF, 5

ILPF, 30

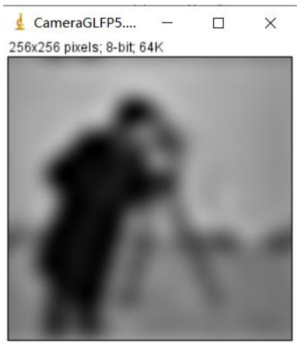
ILPF, 80

BLPF, 5, n=2



BLPF, 30, n=2

BLPF, 80, n=2



GLPF, 5, n=2

GLPF, 30, n=2

GLPF, 80, n=2

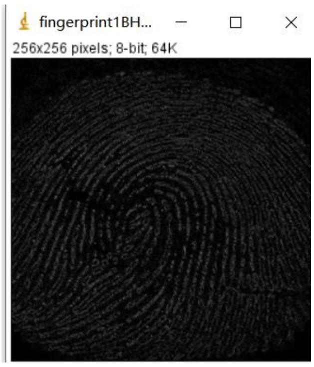
3. Fingerprint1

Source

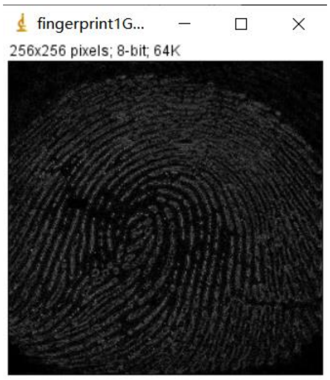
IHPF, 30

IHPF, 60

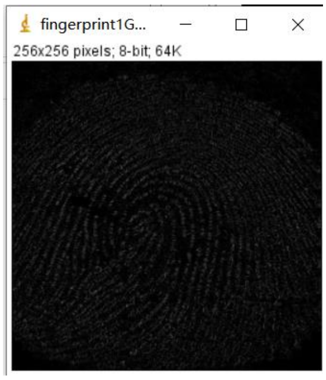
BHPF, 30, n=2



BHPF, 60, n=2



GHPF, 30, n=2



GHPF, 60, n=2

With threshold filter

IHPF, 30 threshold=20

IHPF, 60 threshold=20

BHPF, 30 threshold=20

BHPF, 60 threshold=20

GHPF, 30 threshold=20

GHPF, 60 threshold=20

4. Fingerprint2

Source

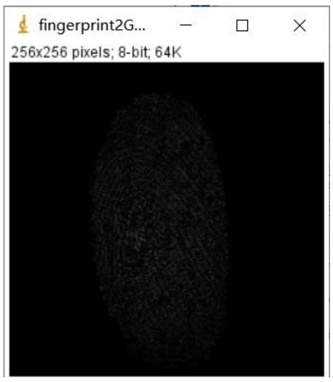
IHPF, 30

IHPF, 60

BHPF, 30, n=2

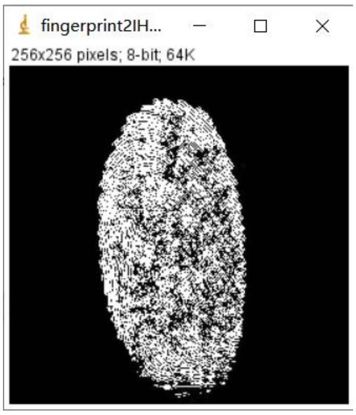
BHPF, 60, n=2

GHPF, 30, n=2



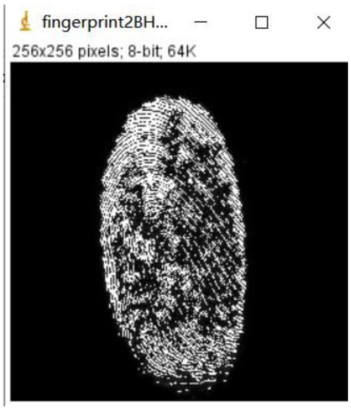
GHPF, 60, n=2

With threshold filter



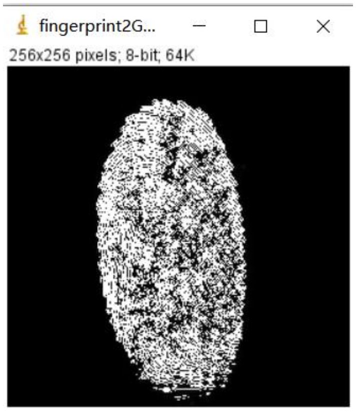
IHPF, 30 threshold=20

IHPF, 60 threshold=20

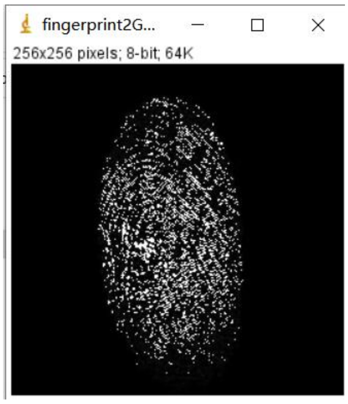


BHPF, 30, n=2

BHPF, 60, n=2



GHPF, 30, n=2



GHPF, 60, n=2

Discussion:

1. An ideal low-pass filter with steep variation will cause the filtered image to "ring".
2. Where, D0 is the cutoff frequency of Butterworth low-pass filter, and parameter n is the order of Butterworth low-pass filter. The larger n is, the steeper the shape of filter is, that is, the more obvious ringing phenomenon is.
3. With the increase of cut-off frequency, the blur of Gaussian low-pass filter becomes weaker and weaker. The smoothing effect is slightly worse than that of second-order BLPF with the same cutoff frequency. No ringing occurs, which is better than BLPF
4. Setting a shrehlod filter can help sharpen the image and see more detail infomation

Codes:

You don't need to paste all the codes. Just show the pieces of codes that present the algorithm displayed above.

1. ILPF

```
Image* ILPF(Image* image, float radius) {
    int i, j;
    unsigned char* tempin, * tempout;
    Image* inimage, * outimage;

    tempin = image->data;
    float dist;
    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));
            if (dist > radius) {
                out[i * image->Width + j].x = 0;
                out[i * image->Width + j].y = 0;
            }

        }
    }

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

    outimage = CreateNewImage(image,"#ILPF img", 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. BLFP

```
Image* BLPF(Image* image, float radius, float order) {
    int i, j;
    unsigned char* tempin, * tempout;
    Image * outimage;
    float dist;
    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);

    tempin = image->data;
    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));
            out[i * image->Width + j].x *=  1 / (1 + pow(dist / radius, 2 * order));
            out[i * image->Width + j].y *=  1 / (1 + pow(dist / radius, 2 * order));
        }
    }

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

    outimage = CreateNewImage(image, "#BLPF img", 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);
}
```

3. GLPF

```
Image* GLPF(Image* image, float radius, float order) {
    int i, j;
    unsigned char* tempin, * tempout;
    Image * outimage;
    float dist;
    tempin = image->data;

    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));
            out[i * image->Width + j].x *= exp((-pow(dist / radius, 2 * order) / (2 * pow(order, 2))));
            out[i * image->Width + j].y *= exp((-pow(dist / radius, 2 * order) / (2 * pow(order, 2))));
        }
    }

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

    outimage = CreateNewImage(image, "#GLPF img", 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);
}
```

4. IHPF

```
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 < radius) {
            out[i * image->Width + j].x = 0;
            out[i * image->Width + j].y = 0;
        }

    }
}
```

5. BHPF

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

6. GHPF

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

7. Threshold filter

```
Image* ThresholdFilter(Image* image, int threshold) {
    unsigned char* tempin, * tempout;
    Image* outimage;
    int i;
    tempin = image->data;
    outimage = CreateNewImage(image, "#GLPF img", image->Height, image->Width);
    tempout = outimage->data;
    for (i = 0; i < image->Width * image->Height; i++) {
        if (tempin[i] > threshold) tempout[i] = 255;
        else tempout[i] = tempin[i];
    }

    return (outimage);

}
```