5. Fourier Transform and Reconstruction

1. 2D Fourier Transform (lena, bridge):

- DFT:

Algorithm:

The two-dimensional discrete Fourier transform formula is as follows:

$$F(u,v) = \frac{1}{\sqrt{MN}} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \mathrm{e}^{-j2\pi(ux/M+vy/N)}$$

where $f(x,y)$ is the original pixel, and $F(u,v)$ is the result after Fourier transform. According to Euler's formula $C = |C|ej\theta$, each value of $F(u,v)$ is a complex number, which is calculated by the real part R and Imaginary part I.

The formula for the Fourier spectrum:

$$\mathrm{F(u,v)} = |\mathrm{F(u,v)}|^2 = [\mathrm{R}^2(\mathrm{u,v}) + \mathrm{I}^2(\mathrm{u,v})]$$

After the transformation, we want the four frequencies to meet at the center of the image, which requires a Fourier translation:

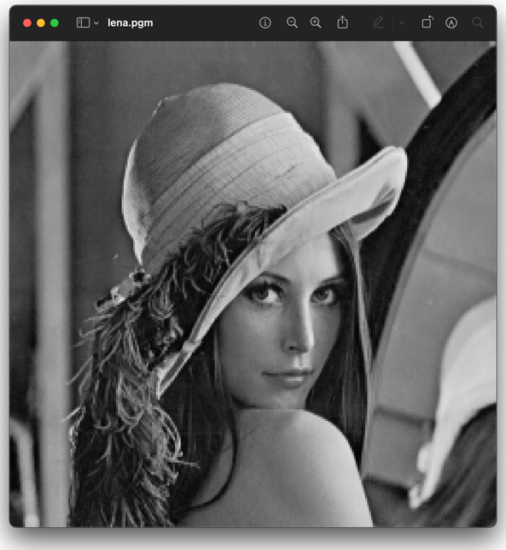
$$f(x,y)(-1)^{x+y} \Leftrightarrow F\left(u - \frac{M}{2}, v - \frac{N}{2}\right)$$

So multiplying each $f(x,y)$ term by $(-1)^{x+y}$ achieves the purpose

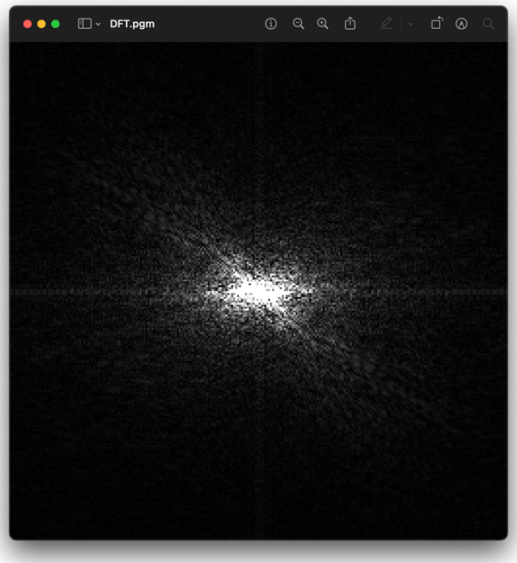
Results (including pictures):

Process result of "lena.pgm":

Source Image:

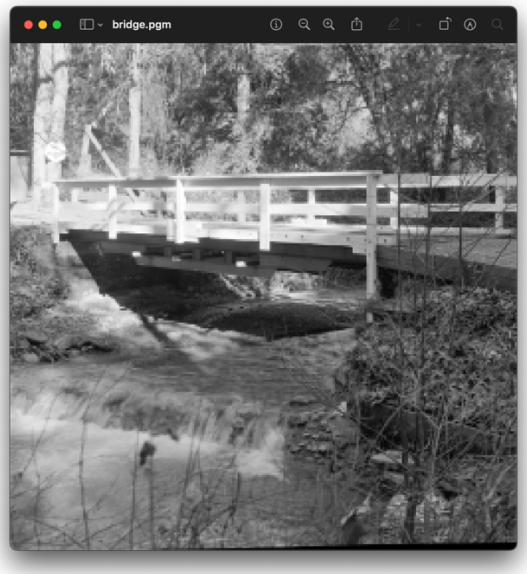


Result of Fourier spectrum:

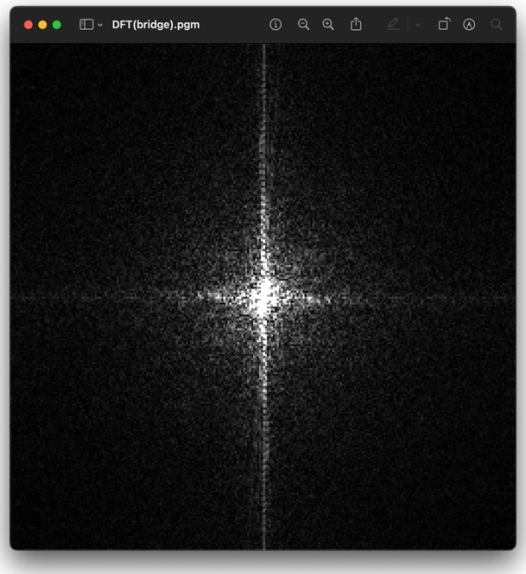


Process result of "bridge.pgm":

Source Image:



Result of Fourier spectrum:



Discussion:

The frequency of an image is an indicator that characterizes the intensity of grayscale changes in an image, and is the gradient of grayscale in plane space. The right image is the corresponding spectrum, the middle part is very bright and the gray value is very high. This means that the lower frequency waves will have very large intensities, while the higher frequency waves will have very small intensities. However, in the process of testing, it is found that the execution speed is really very slow, and the time complexity of the algorithm is $O(n^4)$.

Codes:

```
// Algorithms Code:
Image *DFT(Image *image, float *real_array, float *imaginary_array) {
    unsigned char *tempin, *tempout;
    float real, imaginary;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function");
    tempin = image->data;
    tempout = outimage->data;

    printf("DFT algorithm is executing...\n");
    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++) {
            real = 0;
            imaginary = 0;

            for(int m = 0; m < image->Height; m++) {
                for(int n = 0; n < image->Width; n++) {
                    float temp = (float)i * m / (float)image->Height + (float)j * n / (float)image->Width;
                    int offset = (m + n) % 2 == 0 ? 1 : -1;
                    real += tempin[image->Width * m + n] * cos(-2 * pi * temp) * offset;
                    imaginary += tempin[image->Width * m + n] * sin(-2 * pi * temp) * offset;
                }
            }
            real_array[image->Width * i + j] = real;
            imaginary_array[image->Width * i + j] = imaginary;

            int temp = (int)(sqrt(real*real + imaginary*imaginary) / sqrt((float)image->Height*(float)image->Width));
            if(temp < 0) temp = 0;
            if(temp > 255) temp = 255;
            tempout[image->Width * i + j] = temp;
        }
    }
    printf("DFT Finished!\n");
    return (outimage);
}
```

- iDFT:

Algorithm:

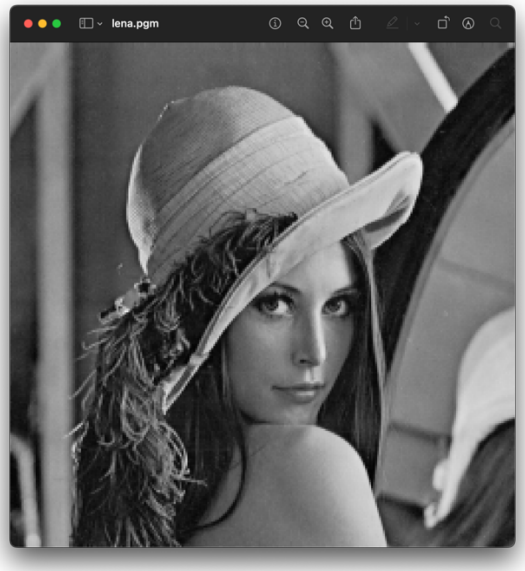
Inverse transform the image using the real part R and imaginary part I obtained from the DFT. The formula of iDFT is:

$$f(x,y) = \frac{1}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) e^{j2\pi(ux/M+vy/N)}$$

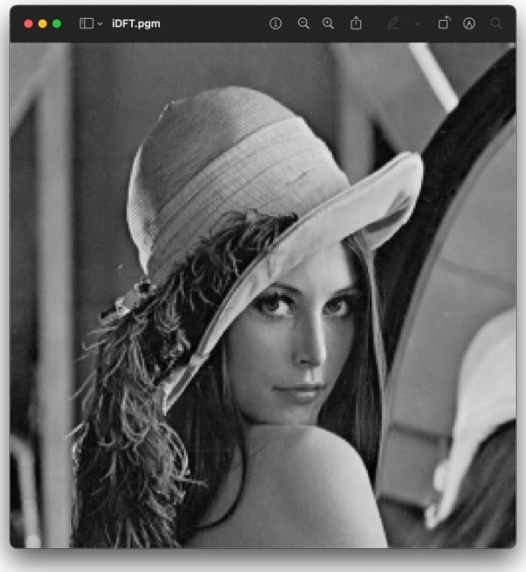
Results (including pictures):

Process result of "lena.pgm":

Source Image:

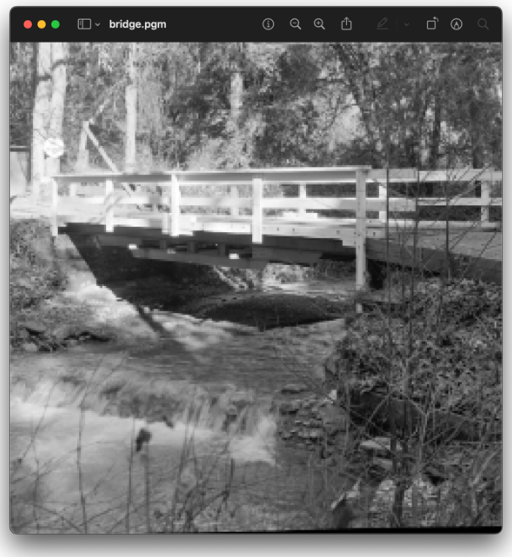


iDFT reconstruct:

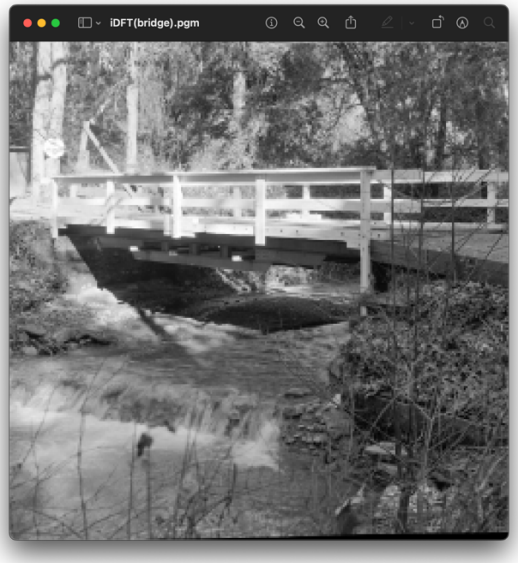


Process result of "bridge.pgm":

Source Image:



iDFT reconstruct:



Discussion:

It has been verified that the original image can be restored after the image is Fourier transformed and then inversely transformed.

Codes:

```
Image *iDFT(Image *image, float *real_array, float *imaginary_array) {
    unsigned char *tempin, *tempout;
    float real;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function");
    tempin = image->data;
    tempout = outimage->data;

    printf("iDFT algorithm is executing...\n");
    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++) {
            real = 0;

            for (int m = 0; m < image->Height; m++){
                for (int n = 0; n < image->Width; n++){
                    float temp = (float)i * m / (float)image->Height + (float)j * n / (float)image->Width;
                    real += real_array[image->Width * m + n] * cos(2 * pi * temp) -
                        imaginary_array[image->Width * m + n] * sin(2 * pi * temp);
                }
            }
            int offset = (i + j) % 2 == 0 ? 1 : -1;
            int temp = (int)(real / ((float)image->Height*(float)image->Width) * offset);
            if(temp < 0) temp = 0;
            if(temp > 255) temp = 255;
            tempout[image->Width * i + j] = temp;
        }
    }
    printf("iDFT Finished!\n");
    return (outimage);
}
```

2. Reconstruction only using the phase angle from 2D DFT (lena):

Algorithm:

According to the formulas of iDFT and phase angle:

$$\phi(u,v) = \arctan\left[\frac{I(u,v)}{R(u,v)}\right]$$

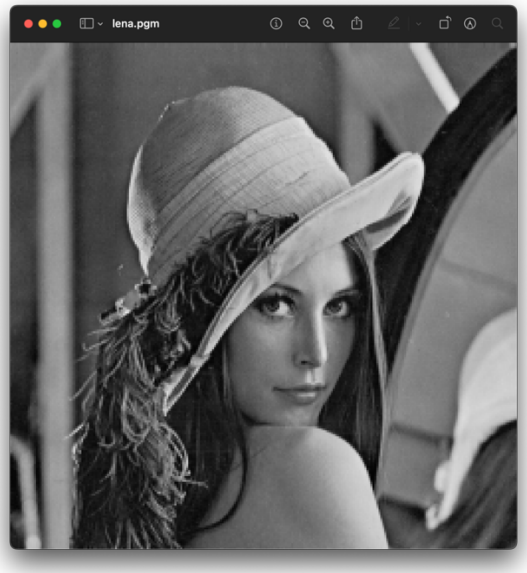
$$f(x,y) = \frac{1}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) e^{j2\pi(ux/M+vy/N)}$$

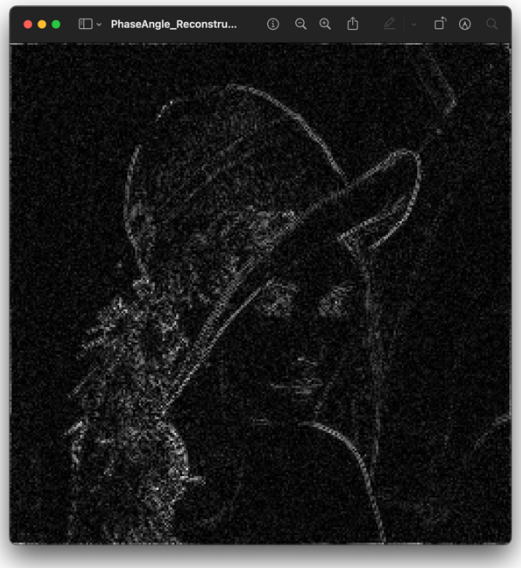
If we want to reconstruct the image only using the phase angle, we need to set $F(u,v)$ to 1.

Results (including pictures):

Process result of “lena.pgm”:

Source Image:





Result after phase angle reconstruct:

Discussion:

As can be seen from the features in the image, the phase information correlates with key shape properties in the image, although the grayscale information has been lost.

Codes:

```
Image *Reconstruct_phaseAngle(Image *image, float *real_array, float *imaginary_array) {
    unsigned char *tempin, *tempout;
    float *tempArray;
    float real, angle, size = image->Width * image->Height;
    int max = 0, min = 10000; // used to normalization
    Image *outimage;

    outimage = CreateNewImage(image, (char*)"#testing function");
    tempin = image->data;
    tempout = outimage->data;
    tempArray = (float*)malloc(size * sizeof(float));

    printf("Phase Angle Reconstruction is executing...\n");
    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++) {
            real = 0;

            for (int m = 0; m < image->Height; m++){
                for (int n = 0; n < image->Width; n++){
                    float temp = (float)i * m / (float)image->Height + (float)j * n / (float)image->Width;
                    angle = atan2(imaginary_array[image->Width * m + n], real_array[image->Width * m + n]);
                    real += cos(angle) * cos(2 * pi * temp) - sin(angle) * sin(2 * pi * temp);
                }
            }
            int temp = round(fabs(real));
            tempArray[image->Width * i + j] = temp;
            if(temp > max) max = temp;
            if(temp < min) min = temp;
        }
    }
    // Normalization:
    for(int i = 0; i < size; i++) {
        float temp = ((float)tempArray[i] - (float)min) / ((float)max - (float)min);
        tempout[i] = (int)(temp * 256);
    }
    printf("Phase Angle Reconstruction Finished!\n");
    return (outimage);
}
```

3. Reconstruction only using the magnitude from 2D DFT (lena):

Algorithm:

According to the formulas of iDFT and amplitude:

$$|F(u,v)| = [R^2(u,v) + I^2(u,v)]^{\frac{1}{2}}$$

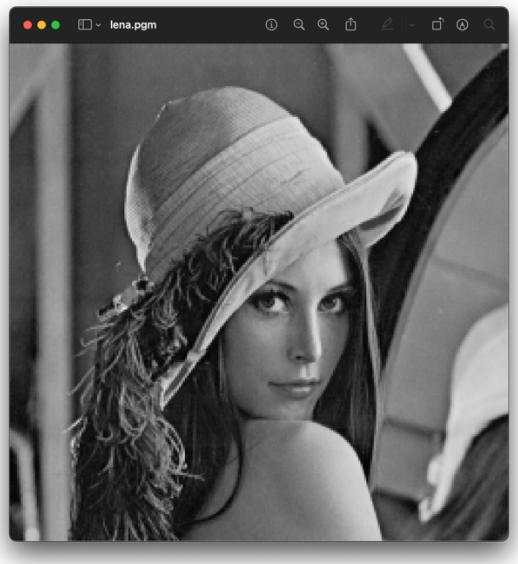
$$f(x,y) = \frac{1}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) e^{j2\pi(ux/M+vy/N)}$$

If we want to reconstruct the image only using the magnitude, we need to set $exponent$ to 0, that is, $e^{j2\pi(ux/M+vy/N)} = 1$.

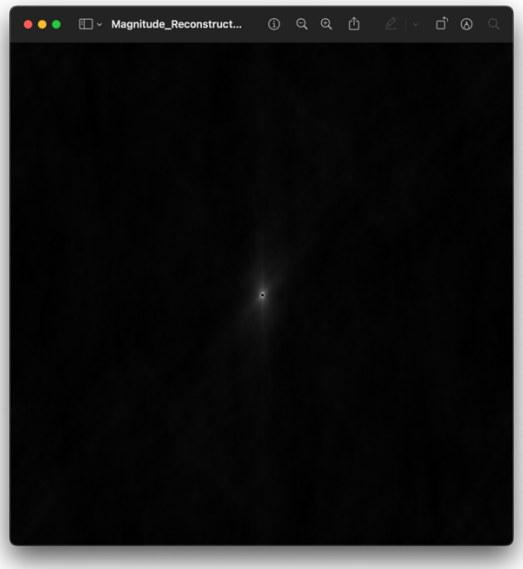
Results (including pictures):

Process result of "lena.pgm":

Source Image:



Result after magnitude reconstruct:



Discussion:

The resulting image uses only the spectrum obtained by inverse Fourier transform, which also means setting the phase angle to 0. So the output contains only grayscale information, and the DC term dominates. There is no shape information in the image because the phase has been set to zero.

Code:

```
Image *Reconstruct_magnitude(Image *image, float *real_array, float *imaginary_array) {
    unsigned char *tempin, *tempout;
    float *tempArray;
    float real, imaginary, size = image->Width * image->Height;
    int max = 0, min = 10000; // used to normalization

    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function");
    tempin = image->data;
    tempout = outimage->data;
    tempArray = (float*)malloc(size * sizeof(float));

    printf("Magnitude Reconstruction is executing...\n");
    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++) {
            real = 0;
            imaginary = 0;

            for (int m = 0; m < image->Height; m++){
                for (int n = 0; n < image->Width; n++){
                    float temp = (float)i * m / (float)image->Height + (float)j * n / (float)image->Width;
                    float mag = sqrt(pow(real_array[image->Width * m + n], 2) + pow(imaginary_array[image->Width * m + n], 2)) /
                        sqrt(size);
                    real += mag * cos(2 * pi * temp) * pow(-1, (float)m + (float)n);
                    imaginary += mag * sin(2 * pi * temp) * pow(-1, (float)m + (float)n);
                }
            }
            int temp = (int)sqrt(pow(real, 2) + pow(imaginary, 2));
            tempArray[image->Width * i + j] = temp;
            if(temp > max) max = temp;
            if(temp < min) min = temp;
        }
    }
    // Normalization:
    for(int i = 0; i < size; i++) {
        float temp = ((float)tempArray[i] - (float)min) / ((float)max - (float)min);
        tempout[i] = (int)(temp * 256);
    }
    printf("Magnitude Reconstruction Finished!\n");
    return (outimage);
}
```