1、傅里叶变换将一个时域或空间域的信号分解为不同频率的正弦和余弦波，从而将信号转换到频域，这意味着任何复杂的波形都可以表示为一系列简单的频率成分的叠加。傅里叶系数则表示每个频率成分的幅度和相位，反映了信号在特定频率上的强度和特征，这有助于从不同的频率上分析其特征，能够“升维思考”。

2、

(1)顺时针旋转 30 度

原理：将输入图像绕笛卡尔坐标系的原点逆时针旋转θ角度，则变换后图像坐标为

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

同时旋转后由于坐标只能取整数，故旋转后图像会存在无颜色数据的点，我们需要对其进行插值处理。

分析：对于所给的彩色图像，首先进行读取：

```
input_image = imread('实验图像.bmp');
```

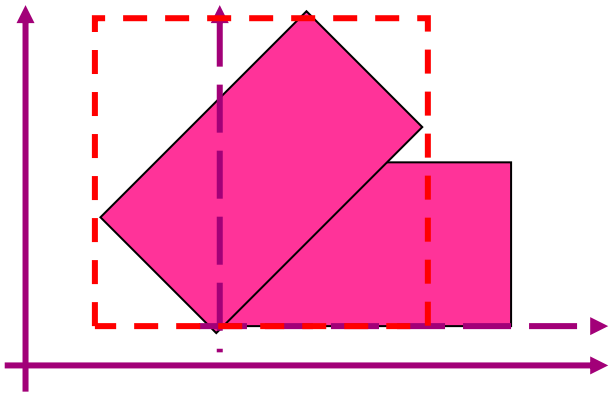
之后将其转换为灰度图，获取原图像的宽和高，对每一个像素进行计算，得到灰度图 gray_image：

```
[row,col,channels]=size(input_image);
gray_image = uint8(zeros(row,col));
%转为灰度图
for i=1:row
    for j=1:col
        gray_image(i,j)=0.299*input_image(i,j,1)+...
            0.587*input_image(i,j,2)+0.114*input_image(i,j,3);
    end
end
```

设定旋转角度，并转换为弧度制：

```
degrees = 30; % 顺时针旋转 30 度
theta = deg2rad(degrees); % 转换为弧度制
```

由于图像旋转后，若要包含图像的全部信息，则新图像会变大，可以通过原图像的四个角点旋转后的坐标来确定(如下图所示)，并确定旋转后的新原点坐标：



```
% 计算旋转后的四个边界点
corners = [1, 1; col, 1; 1, row; col, row];
new_corners = zeros(4, 2);

for i = 1:4
    new_corners(i, :) = calculate_pixels(theta, corners(i, 1), corners(i, 2));
end

% 新的宽度和高度
min_corner = min(new_corners);
max_corner = max(new_corners);
new_width = max_corner(1)-min_corner(1)+1;
new_height = max_corner(2)-min_corner(2)+1;

% 计算新图像的原点
origin_x = min_corner(1)-1;
origin_y = min_corner(2)-1;

% 计算旋转后坐标
function A = calculate_pixels(theta, x, y)
n = [cos(theta), -sin(theta); sin(theta), cos(theta)];
A = round(n * [x; y]);
end
```

通过旋转公式来建立新图像坐标[u,v]到原图像[x,y]的变换，遍历新图像中的每一个点对其像素进行赋值，并变换后在原图像中且仍空白的像素进行双线性插值；

```
rotate_img = uint8(zeros(new_height, new_width));

for i = 1:new_width
    for j = 1:new_height
        % 计算原灰度图像中对应的坐标
        P = calculate_inverse_pixels(theta, i + origin_x, j + origin_y);
        x = P(1);
        y = P(2);
        % 检查原坐标是否在灰度图像范围内
        if x >= 1 && x <= col && y >= 1 && y <= row
            % 使用双线性插值计算像素值
            rotate_img(j, i) = bilinear_interpolation(gray_image, x, y);
        end
    end
end

% 计算逆旋转后原图中的坐标
function A = calculate_inverse_pixels(theta, x, y)
```

```
n = [cos(theta), sin(theta); -sin(theta), cos(theta)];
A = n * [x; y]; % 保留浮点数
end
```

函数 value = bilinear_interpolation(image, x, y)为计算双线性插值后的像素大小，输入新图像上的点[u,v]进过变换后在原图像中的坐标[x,y](为浮点数)以及原图像矩阵，获取该坐标最近的四个像素点，先通过行插值计算，计算结果再进行列插值，通过两次插值计算出新图像的灰度值 f(x，y)，返回像素大小 value：

```
% 双线性插值函数
function value = bilinear_interpolation(image, x, y)
% 获取周围的四个整数像素点横纵坐标
x1 = floor(x);
x2 = ceil(x);
y1 = floor(y);
y2 = ceil(y);
x1 = max(1, min(x1, size(image, 2)));
x2 = max(1, min(x2, size(image, 2)));
y1 = max(1, min(y1, size(image, 1)));
y2 = max(1, min(y2, size(image, 1)));
% 计算各点的权重
if x1==x2
    R1=image(y1, x1);
    R2=image(y2, x1);
else
    R1 = (x2 - x) * image(y1, x1) + (x - x1) * image(y1, x2);
    R2 = (x2 - x) * image(y2, x1) + (x - x1) * image(y2, x2);
end

if y1==y2
    value=R1;
else
    value = (y2 - y) * R1 + (y - y1) * R2;
end
% 保证值在 [0, 255] 之间
value = uint8(max(0, min(255, value)));
end
```

结果：图像被成功顺时针旋转了 30°，同时经过双线性插值后的图像平滑度较好。

顺时针旋转30°的图像

原图

(2) 基于最近邻和双线性插值将图像分别放大 2 倍和 4 倍

原理：将原图像的横纵坐标[x,y]分别乘对应的放大系数，即得新的坐标[u,v]，图像的放大操作中，需对尺寸放大后所多出来的空格填入适当的值，此处采用最近邻插值和双线性插值两种方法分别进行插值。

分析：首先将原彩色图转换为灰度图，代码同(1)，接着给定放大倍数，并计算

新图像的宽和高，由于放大不会改变图像的原点，故新图像的原点仍为(0,0)：

```
% 倍数
n=2;

new_row=round(n*row);
new_col=round(n*col);
```

接着计算放大后像素在原图像中的位置，并进行最近邻插值，即找到原图中距离计算后的坐标最近的像素点，直接进行赋值，将变换后的结果保存在 new_img1 中：

```
new_img1=uint8(zeros(new_row,new_col));

for i=1:new_col
    for j=1:new_row
        [x,y]=inverse_pixels(n,i,j);
        new_img1(j,i)=neighbor_interpolation(gray_image,x,y);
    end
end

% 计算放大后像素在原图像中的位置
function [x,y]=inverse_pixels(num,x1,y1)
x=x1/num;
y=y1/num;
end

% 最近邻插值
function pixel=neighbor_interpolation(img,x,y)
[r,c]=size(img);
x1=max(1,round(x));
y1=max(1,round(y));

x1=min(x1,c);
y1=min(y1,r);
pixel=img(y1,x1);
end
```

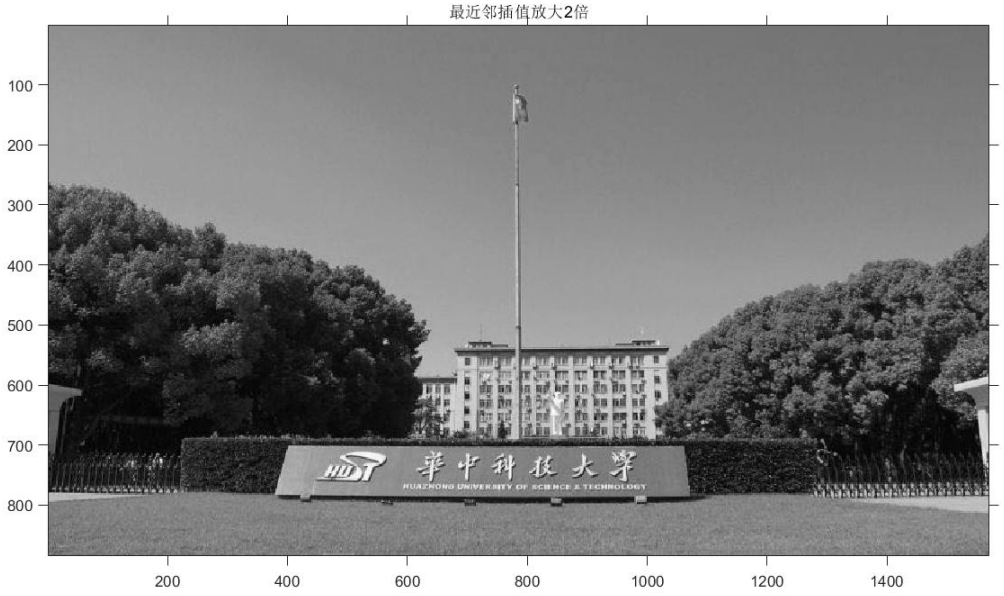
重复上述步骤，对空白点进行双线性插值，原理及代码同(1)，将结果保存在 new_img2 中：

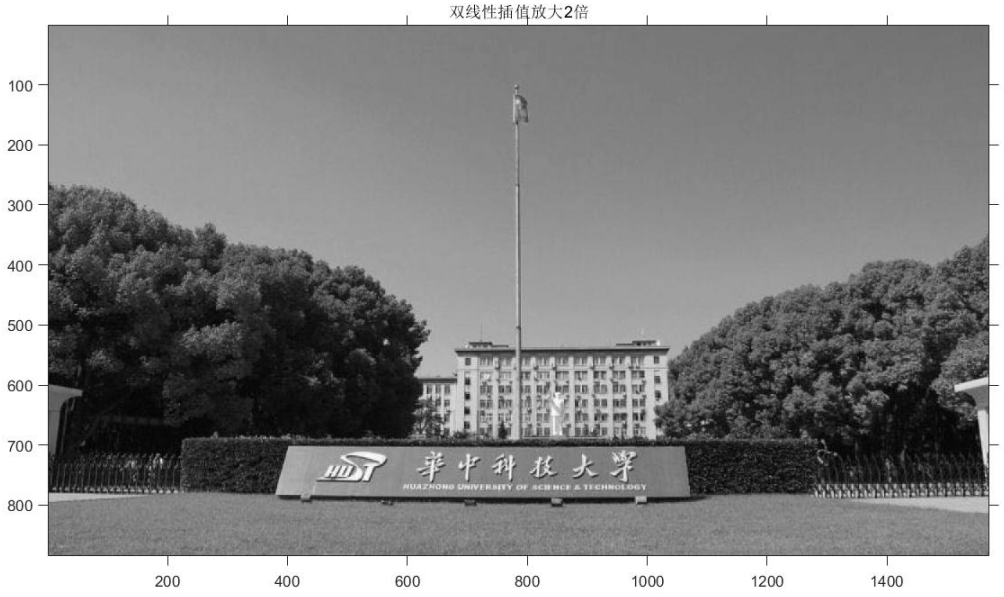
```
new_img2=uint8(zeros(new_row,new_col));

for i=1:new_col
    for j=1:new_row
        [x,y]=inverse_pixels(n,i,j);
        new_img2(j,i)=bilinear_interpolation(gray_image,x,y);
```

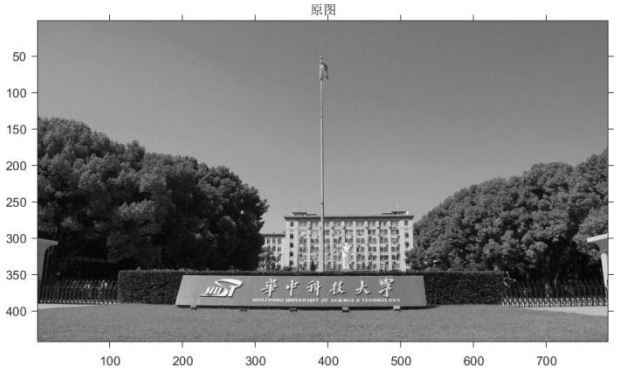
```
    end
end

% 双线性插值
function value = bilinear_interpolation(image, x, y)
% 获取周围的四个整数像素点横纵坐标
x1 = floor(x);
x2 = ceil(x);
y1 = floor(y);
y2 = ceil(y);
x1 = max(1, min(x1, size(image, 2)));
x2 = max(1, min(x2, size(image, 2)));
y1 = max(1, min(y1, size(image, 1)));
y2 = max(1, min(y2, size(image, 1)));
% 计算各点的权重
if x1==x2
    R1=image(y1, x1);
    R2=image(y2, x1);
else
    R1 = (x2 - x) * image(y1, x1) + (x - x1) * image(y1, x2);
    R2 = (x2 - x) * image(y2, x1) + (x - x1) * image(y2, x2);
end

if y1==y2
    value=R1;
else
    value = (y2 - y) * R1 + (y - y1) * R2;
end
%disp([x,y,x1,x2,y1,y2]);
% 保证值在 [0, 255] 之间
value = uint8(max(0, min(255, value)));
end
```

将代码中变量 n 的值改为 4，即可对图像用两种方法分别放大 4 倍，原理分析同上。

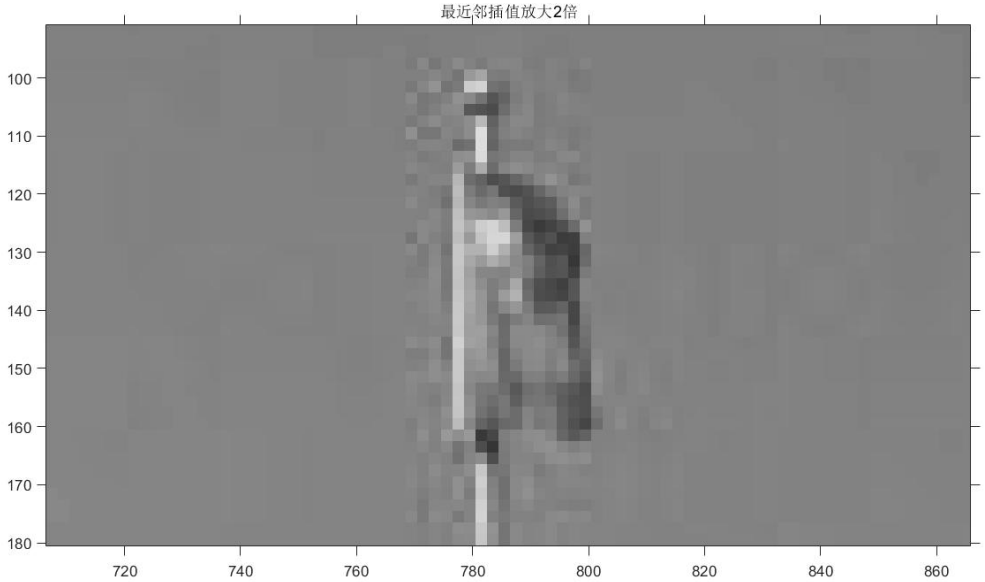
结果：

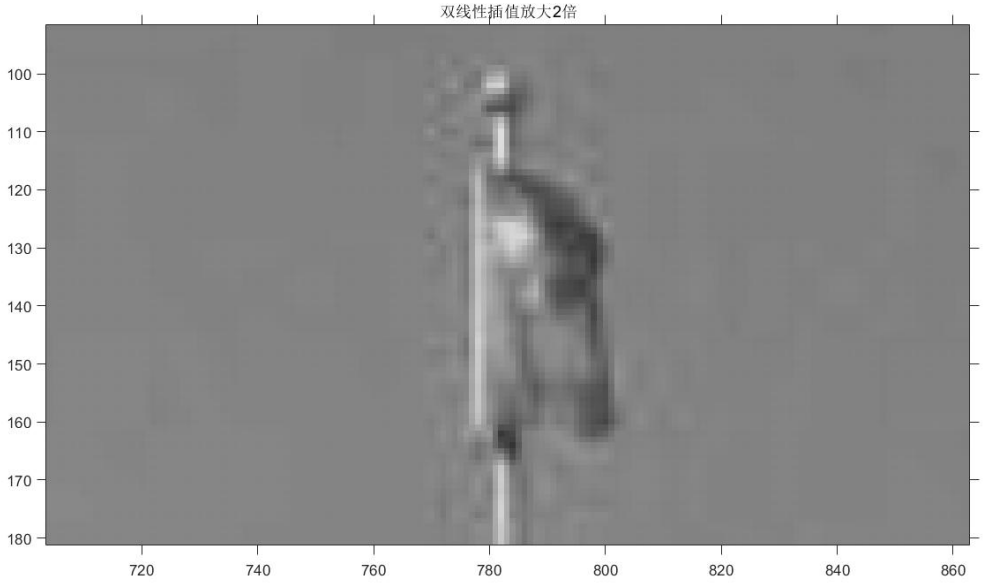

最近邻插值放大2倍
100
200
300
400
500
600
700
800
200
400
600
800
1000
1200
1400


双线性插值放大2倍
100
200
300
400
500
600
700
800
200
400
600
800
1000
1200
1400


原图
50
100
150
200
250
300
350
400
100
200
300
400
500
600
700

将最近邻插值和双线性插值得到的图片进行放大，可以发现双线性插值可以更好地保留图像的特征，在细节上更加平滑，如下图红旗的位置所示：





3、展开傅里叶变换，提取傅里叶变换图像（将频率原点移至图像中心）

原理：

一维离散傅里叶变换

对于一个长度为 N 的序列 $x[n]$，其DFT $X[k]$ 定义为：

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-2\pi i \frac{kn}{N}}, \quad k = 0, 1, \ldots, N-1$$

二维离散傅里叶变换

对于一个大小为 $M \times N$ 的二维信号 $f(m, n)$，其DFT $F(u, v)$ 定义为：

$$F(u,v)=\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}f(m,n)\cdot e^{-2\pi i\left(\frac{um}{M}+\frac{vn}{N}\right)},\quad u=0,1,\ldots,M-1,\quad v=0,1,\ldots,N-1$$

由于直接对图像进行二维傅里叶变换非常耗时，故将其转化为两次一维傅里叶变换

$$F(x,v)=\frac{1}{N}\sum_{y=0}^{N-1}f(x,y)\exp[-j2\pi vy/N]$$

$$F(u,v)=\frac{1}{N}\sum_{x=0}^{N-1}F(x,v)\exp[-j2\pi ux/N]$$

分析：首先读取图像并转换为灰度图，同 2，接着调用手动实现的二维傅里叶变换 my_dft2(img)，第一部分循环遍历每一行，使用 my_dft1d 函数计算一维傅里叶变换，第二部分循环遍历每一列，计算结果矩阵 F 的列的一维傅里叶变换：

```
function F = my_dft2(img)
[M, N] = size(img);
F = zeros(M, N);
% 对每一行进行 DFT
for m = 1:M
    F(m, :) = my_dft1d(img(m, :));
End

% 对每一列进行 DFT
for n = 1:N
    F(:, n) = my_dft1d(F(:, n)); % 对每一列进行一维傅里叶变换
end
end
```

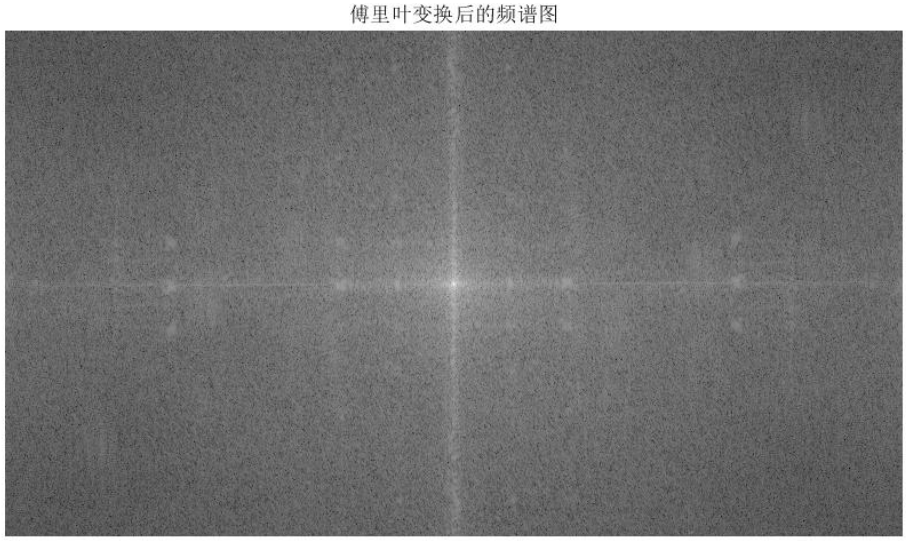
上述代码中的一维傅里叶变换函数 my_dft1d(x)，：

```
function X = my_dft1d(x)
N = length(x);
X = zeros(1, N);
% 计算傅里叶变换
for k = 1:N
    for n = 1:N
        X(k) = X(k) + x(n) * exp(-2 * pi * 1i * (k-1) * (n-1) / N);
    end
end
end
```

调用该傅里叶变换函数并展示结果：

```
% 使用手动实现的 FFT 进行二维傅里叶变换
F_manual = my_dft2(img);

% 计算幅度谱并取对数
magnitude_spectrum_manual = log(1 + abs(fftshift(F_manual)));

% 显示原始图像和幅度谱
figure;
imshow(uint8(img));
title('原图');

figure;
imshow(magnitude_spectrum_manual, []);
title('傅里叶变换后的频谱图');
```

结果：



代码：

2. (1)

```
clc,clear;
input_image = imread('实验图像.bmp');

[row,col,channels]=size(input_image);
gray_image = uint8(zeros(row,col));
%转为灰度图
for i=1:row
for j=1:col
gray_image(i,j)=0.299*input_image(i,j,1)+...
0.587*input_image(i,j,2)+0.114*input_image(i,j,3);
end
end

degrees = 30; % 顺时针旋转 30 度
theta = deg2rad(degrees); % 转换为弧度制

% 计算旋转后的四个边界点
corners = [1, 1; col, 1; 1, row; col, row];
new_corners = zeros(4, 2);

for i = 1:4
new_corners(i, :) = calculate_pixels(theta, corners(i, 1), corners(i, 2));
end

% 新的宽度和高度
min_corner = min(new_corners);
max_corner = max(new_corners);
new_width = max_corner(1) - min_corner(1)+1;
new_height = max_corner(2) - min_corner(2)+1;

rotate_img = uint8(zeros(new_height, new_width));

% 计算新图像的原点
origin_x = min_corner(1)-1;
origin_y = min_corner(2)-1;

% 使用双线性插值
for i = 1:new_width
for j = 1:new_height
% 计算原灰度图像中对应的坐标
P = calculate_inverse_pixels(theta, i + origin_x, j + origin_y);
x = P(1);
```

```
y = P(2);
% 检查原坐标是否在灰度图像范围内
if x >= 1 && x <= col && y >= 1 && y <= row
rotate_img(j, i) = bilinear_interpolation(gray_image, x, y);
end
end
end

imshow(rotate_img);
title("顺时针旋转 30°的图像");

% 计算旋转后坐标，返回一个列向量
function A = calculate_pixels(theta, x, y)
n = [cos(theta), -sin(theta); sin(theta), cos(theta)];
A = round(n * [x; y]);
end

% 计算逆旋转后原图中的坐标
function A = calculate_inverse_pixels(theta, x, y)
n = [cos(theta), sin(theta); -sin(theta), cos(theta)];
A = n * [x; y];
end

% 双线性插值函数
function value = bilinear_interpolation(image, x, y)
% 获取周围的四个整数像素点横纵坐标
x1 = floor(x);
x2 = ceil(x);
y1 = floor(y);
y2 = ceil(y);
x1 = max(1, min(x1, size(image, 2)));
x2 = max(1, min(x2, size(image, 2)));
y1 = max(1, min(y1, size(image, 1)));
y2 = max(1, min(y2, size(image, 1)));
% 计算各点的权重
if x1==x2
R1=image(y1, x1);
R2=image(y2, x1);
else
R1 = (x2 - x) * image(y1, x1) + (x - x1) * image(y1, x2);
R2 = (x2 - x) * image(y2, x1) + (x - x1) * image(y2, x2);
end

if y1==y2
```

```
value=R1;
else
value = (y2 - y) * R1 + (y - y1) * R2;
end
% 保证值在 [0, 255] 之间
value = uint8(max(0, min(255, value)));
end
```

(2)

```
clc,clear;
input_image=imread("实验图像.bmp");

[row,col,channels]=size(input_image);
gray_image=uint8(zeros(row,col));
%转为灰度图
for i=1:row
for j=1:col
gray_image(i,j)=0.299*input_image(i,j,1)+...
0.587*input_image(i,j,2)+0.114*input_image(i,j,3);
end
end

% 倍数
n=2;

new_row=round(n*row);
new_col=round(n*col);

new_img1=uint8(zeros(new_row,new_col));

for i=1:new_col
for j=1:new_row
[x,y]=inverse_pixels(n,i,j);
new_img1(j,i)=neighbor_interpolation(gray_image,x,y);
end
end

new_img2=uint8(zeros(new_row,new_col));
```

```
for i=1:new_col
for j=1:new_row
[x,y]=inverse_pixels(n,i,j);
new_img2(j,i)=bilinear_interpolation(gray_image,x,y);
end
end

figure;
imshow(new_img1);
title("最近邻插值放大 2 倍");
axis on;

figure;
imshow(new_img2);
title("双线性插值放大 2 倍");
axis on;

figure;
imshow(gray_image);
title("原图");
axis on;

% 计算放大后像素在原图像中的位置
function [x,y]=inverse_pixels(num,x1,y1)
x=x1/num;
y=y1/num;
end

% 最近邻插值
function pixel=neighbor_interpolation(img,x,y)
[r,c]=size(img);
x1=max(1,round(x));
y1=max(1,round(y));

x1=min(x1,c);
y1=min(y1,r);
pixel=img(y1,x1);
end

% 双线性插值
function value = bilinear_interpolation(image, x, y)
% 获取周围的四个整数像素点横纵坐标
x1 = floor(x);
x2 = ceil(x);
```

```
y1 = floor(y);
y2 = ceil(y);
x1 = max(1, min(x1, size(image, 2)));
x2 = max(1, min(x2, size(image, 2)));
y1 = max(1, min(y1, size(image, 1)));
y2 = max(1, min(y2, size(image, 1)));
% 计算各点的权重
if x1==x2
R1=image(y1, x1);
R2=image(y2, x1);
else
R1 = (x2 - x) * image(y1, x1) + (x - x1) * image(y1, x2);
R2 = (x2 - x) * image(y2, x1) + (x - x1) * image(y2, x2);
end

if y1==y2
value=R1;
else
value = (y2 - y) * R1 + (y - y1) * R2;
end
%disp([x,y,x1,x2,y1,y2]);
% 保证值在 [0, 255] 之间
value = uint8(max(0, min(255, value)));
end
```

```
clc,clear;
img = imread('实验图像.bmp');
if size(img, 3) == 3
img = rgb2gray(img);
end
img = double(img);

% 二维傅里叶变换
F_manual = my_dft2(img);

% 计算幅度谱并取对数
magnitude_spectrum_manual = log(1 + abs(fftshift(F_manual)));

% 显示原始图像和幅度谱
```

```
figure;
imshow(uint8(img));
title('原图');

figure;
imshow(magnitude_spectrum_manual, []);
title('傅里叶变换后的频谱图');


function F = my_dft2(img)
[M, N] = size(img);

F = zeros(M, N);
% 对每一行进行 DFT
for m = 1:M
F(m, :) = my_dft1d(img(m, :));
end

% 对每一列进行 DFT
for n = 1:N
F(:, n) = my_dft1d(F(:, n));
end
end

% 实现一维离散傅里叶变换
function X = my_dft1d(x)
N = length(x);
X = zeros(1, N);
% 一维傅里叶变换
for k = 1:N
for n = 1:N
X(k) = X(k) + x(n) * exp(-2 * pi * 1i * (k-1) * (n-1) / N);
end
end
end
```