姚慧峰 U20221513 人工智能 2204 班

姚慧峰

一.傅里叶变换和傅里叶系数的理解：

1.1 傅里叶变换

傅里叶变换是一种数学工具，用于将信号或函数从时域转换到频域。这种变换揭示了信号的频率成分，即信号由哪些频率的正弦波或余弦波组成，以及每个频率成分的幅度和相位。

傅里叶变换的物理含义：

频率成分分析：任何复杂的信号，无论是声音、光、电信号还是其他类型的波形，都可以被分解为不同频率的正弦波和余弦波的组合。傅里叶变换提供了一种方法来识别这些组成波的频率、幅度和相位。

信号处理：在信号处理中，傅里叶变换用于滤波、数据压缩、噪声降低等。通过将信号转换到频域，可以更容易地识别和处理特定的频率成分。

图像分析：在图像处理中，傅里叶变换用于分析图像中的模式和特征。例如，通过分析图像的傅里叶变换，可以识别图像中的周期性结构或边缘。

通信系统：在无线通信中，傅里叶变换用于调制和解调信号。通过将信息编码到不同频率的载波上，可以在同一信道上传输多个信号。

1.2 傅里叶系数

傅里叶系数是傅里叶级数或傅里叶变换的结果，它们描述了信号在各个频率上的分量。对于周期信号，傅里叶级数提供了一个表示信号的三角级数，其中每个系数对应于一个特定的频率成分。

傅里叶系数的物理含义：

幅度：傅里叶系数的绝对值表示相应频率成分的幅度或强度。幅度较大的系数意味着信号在该频率上有较强的能量。

相位：傅里叶系数的相位表示相应频率成分的相位偏移。

能量分布：在物理系统中，傅里叶系数的平方可以表示信号在各个频率上的能量分布。

二.图片的旋转与放大

2.1 彩色图转灰度图

2.1.1 转换方法

我查阅了相关资料，彩色图转灰度图有多钟方法，其中我采用了加权平均的方法。这是最常用的一种方法，它基于人眼对不同颜色敏感度的调查结果来确定 RGB 三个通道的权重。由于人眼对绿色的敏感度最高，对红色次之，对蓝色最低，因此通常使用的权重是：

红色通道权重：0.299

绿色通道权重：0.587

蓝色通道权重：0.114

转换公式为：**灰度值=0.299*R+0.587*G+0.114*B**

这种方法能够较准确地反映原图的亮度信息。

2.1.2 实现代码

```
% 转换为灰度图
%使用自带库
gray_img1 = rgb2gray(test);
%编程转化为灰度图
% 初始化灰度图像矩阵
gray_img2 = zeros(size(test, 1), size(test, 2));
%遍历每个像素
for i = 1:size(test, 1)
for j = 1:size(test, 2)
% 获取 RGB 值并采用加权平均的方法求灰度值
r = test(i, j, 1);
g = test(i, j, 2);
b = test(i, j, 3);
gray_img2(i, j) = 0.299 * r + 0.587 * g + 0.114 * b;
end
end
gray_img2=uint8(gray_img2);
% 显示灰度图
figure;
subplot(1, 2, 1);
imshow(gray_img1);
title('Image 1');
subplot(1, 2, 2);
imshow(gray_img2);
title('Image 2');
```

2.1.3 结果对比与分析

图一　　自带库与手写代码灰度图转换比较

其中左侧是 MATLAB 自带库转换的结果，右边是我们利用上述方法得到的结果，我们经过比较发现两张图片基本没有差异，即便是放大后细节方面的东西区别也不是太大。

2.2 旋转

2.2.1 方法

我们采用反向查找的方法进行图片的旋转，这样做可以避免图片某些像素点存在空值。具体方法就是求出新图像每一个像素点旋转前在原图片中的位置，再将原图片中对应像素赋值给新图片。其中我们利用旋转矩阵 R 来得到原图片中的位置。

其中 $$R=\begin{matrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{matrix}$$。

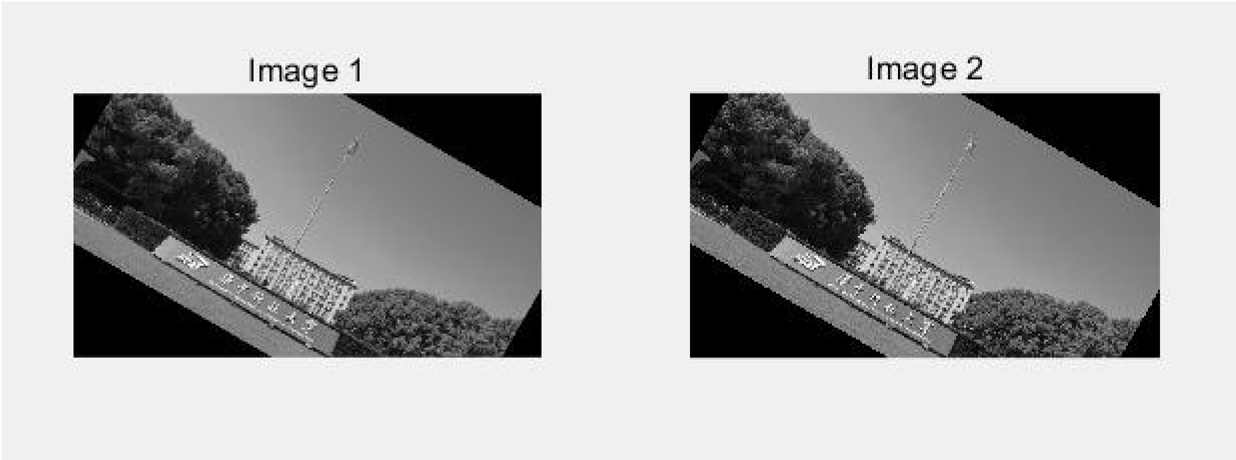
在旋转后图片像素点确定时，我们采用了双性线性插值法，保证了图片的边缘更加平滑。

2.2.2 代码

```
% 旋转图像
%使用库函数
rotated_img1 = imrotate(gray_img1, -30, 'bilinear', 'crop');
%自己编程
[ h, w ] = size(gray_img1);
rotated_img2=uint8(zeros(h, w));
angle=-30;
angle=deg2rad(angle);
RotMatrix = [ cos(angle), -sin(angle); sin(angle), cos(angle) ];%旋转矩阵
halfH = floor( h / 2 );
halfW = floor( w / 2 );
%逐个像素计算其旋转前的位置及像素
```

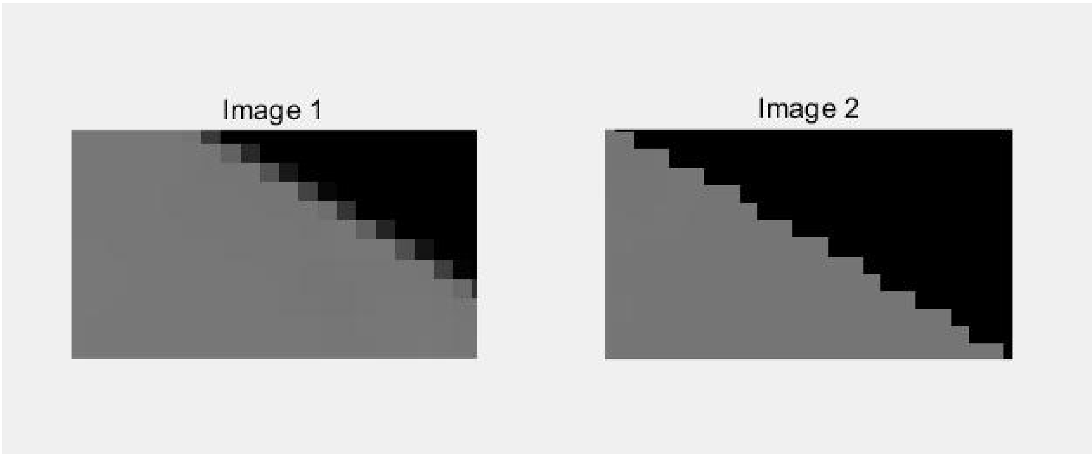
```
for i = 1 : h
for j = 1 : w
coord = [ i-halfH, j-halfW ] * RotMatrix;%中心化旋转
ir = round(coord(1)) + halfH;
jr = round(coord(2)) + halfW;
if ir > 0 && ir <= h && jr > 0 && jr <= w
ir_floor = floor(ir);
jr_floor = floor(jr);
ir_ceil = ceil(ir);
jr_ceil = ceil(jr);
% 计算插值权重
u = ir - ir_floor;
v = jr - jr_floor;
% 获取四个相邻像素
Q11 = double(gray_img2(ir_floor, jr_floor));
Q21 = double(gray_img2(ir_ceil, jr_floor));
Q12 = double(gray_img2(ir_floor, jr_ceil));
Q22 = double(gray_img2(ir_ceil, jr_ceil));
% 双线性插值
rotated_img2(i, j) = (1-u)*(1-v)*Q11 + u*(1-v)*Q21 + (1-u)*v*Q12 + u*v*Q22;
end
end
end
% 显示旋转后的图像
figure;
subplot(1, 2, 1);
imshow(rotated_img1);
title('Image 1');
subplot(1, 2, 2);
imshow(rotated_img2);
title('Image 2');
```

2.2.3 结果分析



图二　自带库与手写代码旋转结果比较

从结果上看，我们手动编程的结果比 MATLAB 自带库的结果在边缘处理上要差了一点，我们放大来看，结果如下：



图三　自带库与手写代码旋转结果放大边缘比较

可以看出，MATLAB 自带库边缘更加平滑，有一个过渡的阶段，我们的代码在这一点上需要改进。

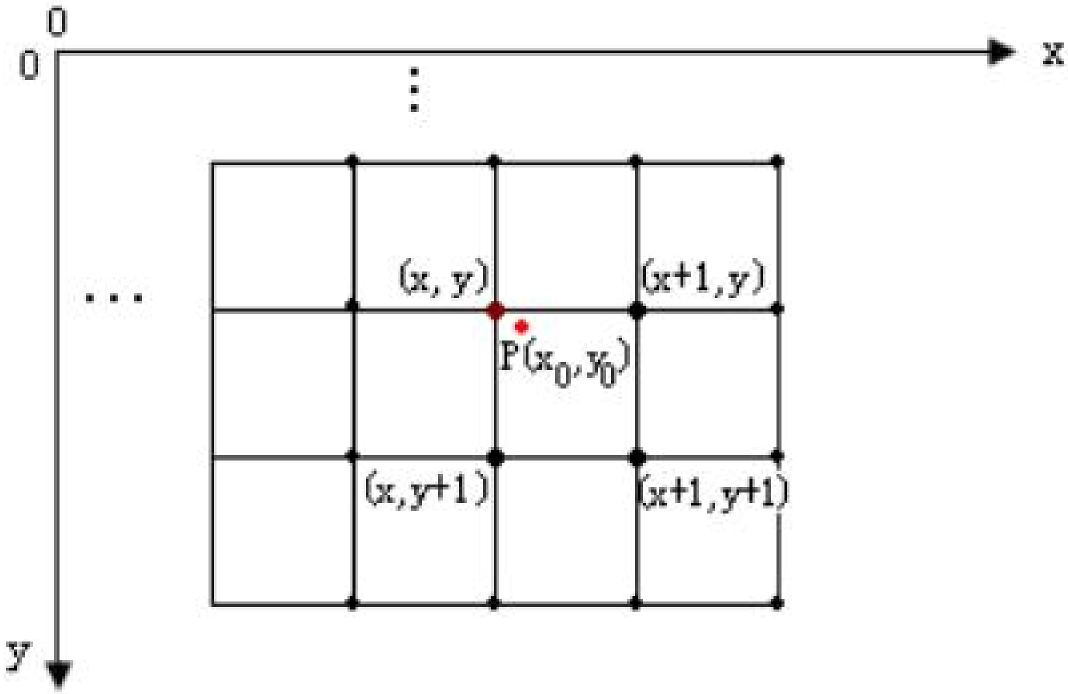
2.3 放大

2.3.1 方法

图片放大所需要注意的问题就是放大后图片空值的补缺问题，我们在这里使用了两种方法进行插值，分别是最近邻法和双线性插值法。

1.最近邻法：

计算与点 P(x0，y0)临近的四个点，将与点 P(x0，y0)最近的整数坐标点(x，y)的灰度值取为 P(x0，y0) 点灰度近似值。



图四　　最近邻法

2. 双线性插值法：

根据点 P(x0，y0)的四个相邻点的灰度值，通过两次插值计算出灰度值 f(x0，y0) 。公式为：

$$
\begin{aligned}
f(x_0,y_0)=f(x,y)+\alpha[f(x+1,y)-f(x,y)]+\\
\beta[f(x,y+1)-f(x,y)]+\\
\alpha\beta[f(x+1,y+1)+f(x,y)-\\
f(x,y+1)-f(x+1,y)]
\end{aligned}
$$

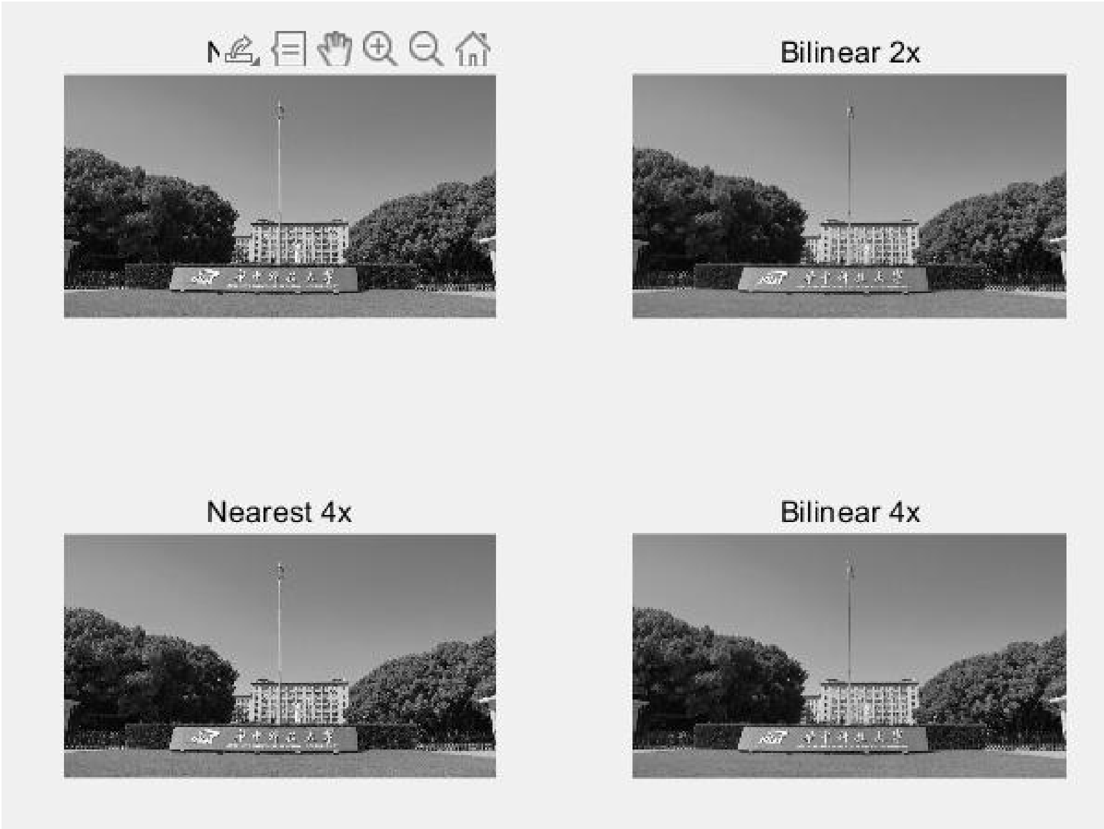
2.3.1 代码

```
% 最近邻插值放大 2 倍
nearest_img_2x = imresize(gray_img1, 2, 'nearest');
% 双线性插值放大 2 倍
bilinear_img_2x = imresize(gray_img1, 2, 'bilinear');
% 最近邻插值放大 4 倍
nearest_img_4x = imresize(gray_img1, 4, 'nearest');
% 双线性插值放大 4 倍
bilinear_img_4x = imresize(gray_img1, 4, 'bilinear');
% 显示放大后的图像
subplot(2,2,1), imshow(nearest_img_2x), title('Nearest 2x');
subplot(2,2,2), imshow(bilinear_img_2x), title('Bilinear 2x');
subplot(2,2,3), imshow(nearest_img_4x), title('Nearest 4x');
```

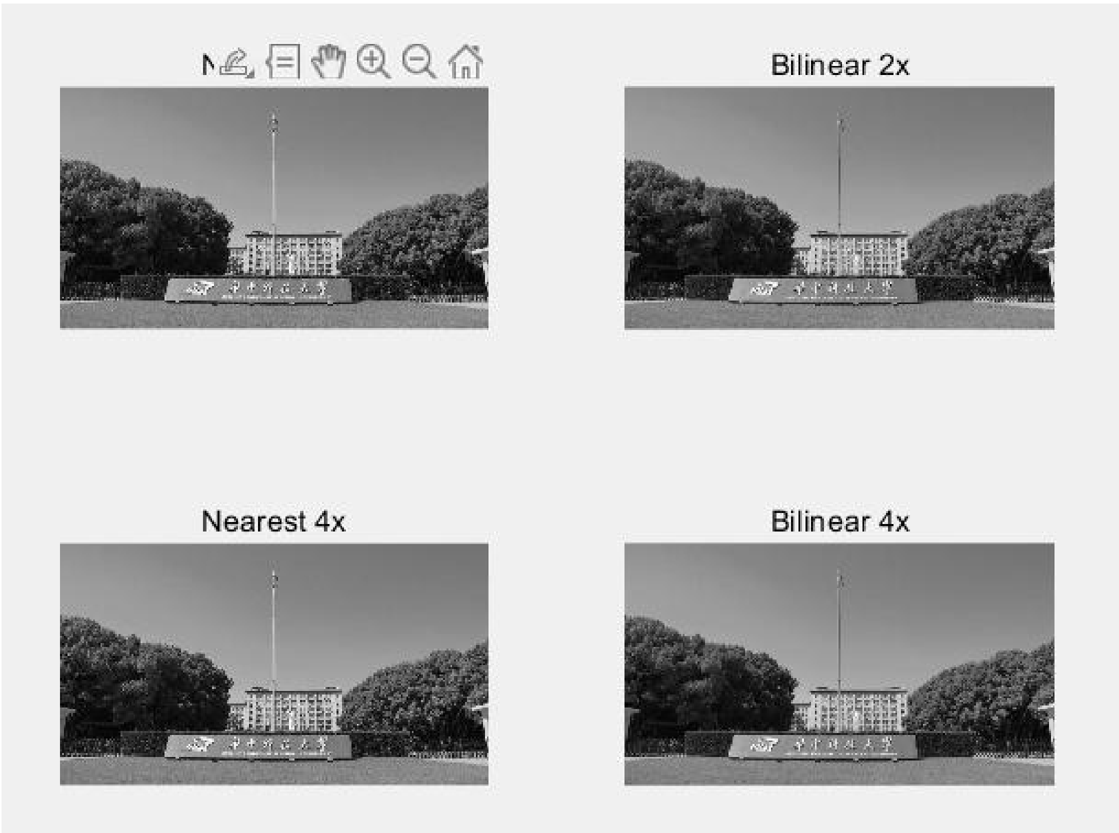
```
subplot(2,2,4), imshow(bilinear_img_4x), title('Bilinear 4x');

%不使用自带库放大
nearest_img_2x=nearest_neighbor_upscaling(gray_img2, 2);
bilinear_img_2x=bilinear_upscaling(gray_img2, 2);
nearest_img_4x=nearest_neighbor_upscaling(gray_img2, 4);
bilinear_img_4x=bilinear_upscaling(gray_img2, 4);
figure;
imshow(bilinear_img_4x);
figure;
imshow(gray_img2);
% 显示放大后的图像
subplot(2,2,1), imshow(nearest_img_2x), title('Nearest 2x');
subplot(2,2,2), imshow(bilinear_img_2x), title('Bilinear 2x');
subplot(2,2,3), imshow(nearest_img_4x), title('Nearest 4x');
subplot(2,2,4), imshow(bilinear_img_4x), title('Bilinear 4x');

function upscaled_img = nearest_neighbor_upscaling(img, scale_factor)
[rows, cols] = size(img);
upscaled_rows = rows * scale_factor;
upscaled_cols = cols * scale_factor;
upscaled_img = uint8(zeros(upscaled_rows, upscaled_cols));
for i = 1:upscaled_rows
for j = 1:upscaled_cols
% 计算最近邻像素的坐标
nearest_row = floor((i - 1) / scale_factor) + 1;
nearest_col = floor((j - 1) / scale_factor) + 1;
% 复制像素值
upscaled_img(i, j) = img(nearest_row, nearest_col);
end
end
end
function upscaled_img = bilinear_upscaling(img, scale_factor)
[rows, cols] = size(img);
upscaled_rows = rows * scale_factor;
upscaled_cols = cols * scale_factor;
upscaled_img = uint8(zeros(upscaled_rows, upscaled_cols));
for i = 1:upscaled_rows
for j = 1:upscaled_cols
% 计算原图中的坐标
x = (i - 1) / scale_factor + 1;
y = (j - 1) / scale_factor + 1;
% 找到四个最近邻像素的坐标
x1 = floor(x);
```

```matlab
y1 = floor(y);
x2 = ceil(x);
y2 = ceil(y);
% 检查边界
if x2 > rows
x2 = rows;
end
if y2 > cols
y2 = cols;
end
% 计算插值权重
u = x - x1;
v = y - y1;
% 获取四个像素点的值
Q11 = double(img(x1, y1));
Q21 = double(img(x2, y1));
Q12 = double(img(x1, y2));
Q22 = double(img(x2, y2));
% 计算插值
fxy = (1-u)*(1-v)*Q11 + u*(1-v)*Q21 + (1-u)*v*Q12 + u*v*Q22;
% 将插值结果赋值给放大后的图像
upscaled_img(i, j) = uint8(fxy);
end
end
end
```

2.3.1 结果分析

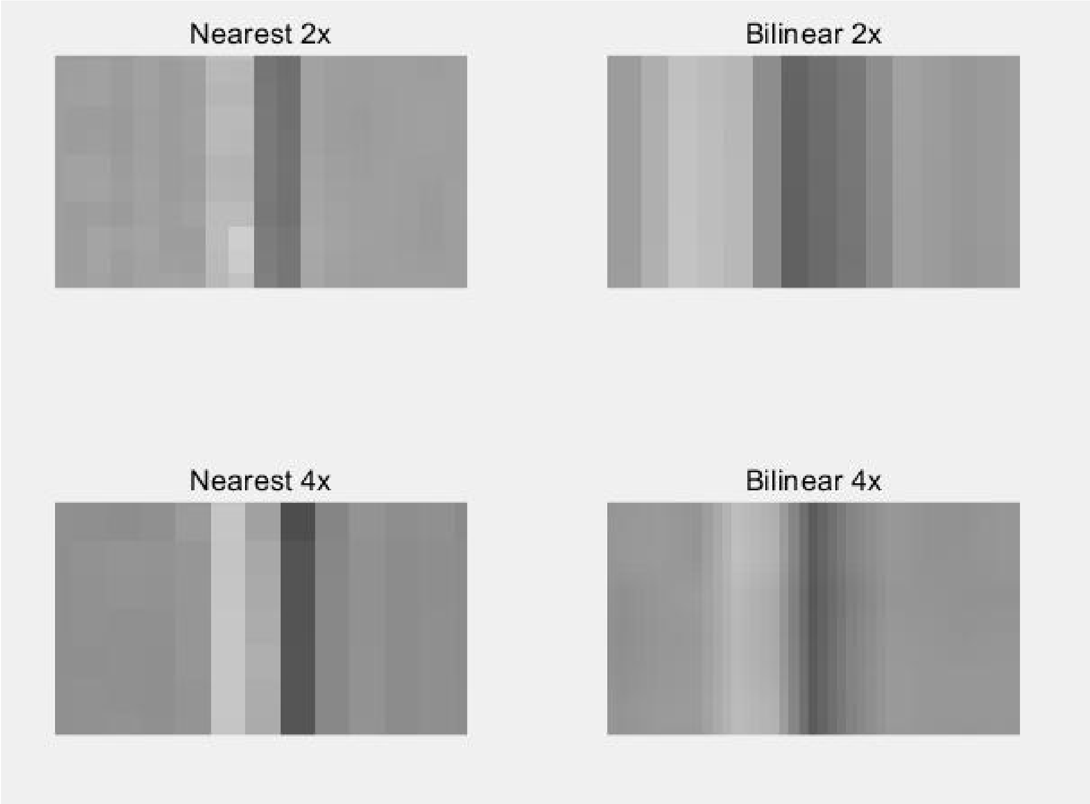


图五　　使用 MATLAB 自带库放大结果



图六　　手写代码放大结果

比较上述两幅图，我们可以看出二者并未有太大区别，但是放大后我们观察细节可以知道 MATLAB 自带库在细节方面做出的处理更好，我们在细节方面则略显不足。



图七　　手写代码在旗杆处放大结果对比图

而且对比最近邻法与双线性插值法，我们可以发现双线性插值法在细节方面做出的处理更加平滑，这种平滑可能会使细节方面的东西退化，最近邻法则是显得有很明显的人工处理痕迹。

三.图像的傅里叶变换

3.1 方法

我们对图片进行了二维离散傅里叶变换。直接转换的公式如下：

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

但是这样直接转换时我们编程会发现所需要的计算时间成本会很大，所以我们在这里利用了二维离散傅里叶变换的可分离性进行编程，我们利用矩阵相乘进行表示：

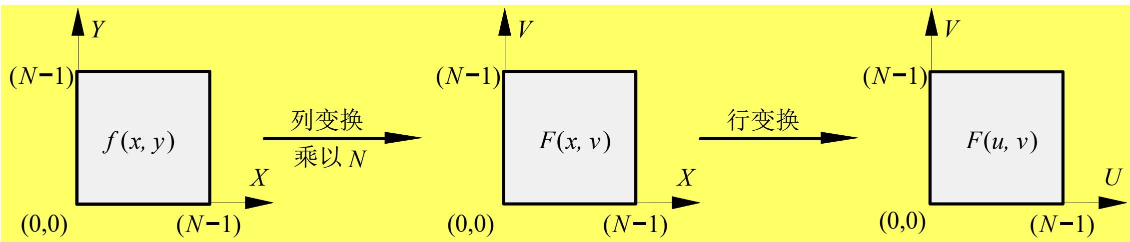
$$F=G_1fG_2$$

其中 G1 与 G2 分别为:

$$G_1=\begin{bmatrix} e^{-j2\pi\frac{0\cdot0}{M}} & e^{-j2\pi\frac{0\cdot1}{M}} & \cdots & e^{-j2\pi\frac{0\cdot(M-1)}{M}} \\ e^{-j2\pi\frac{1\cdot0}{M}} & e^{-j2\pi\frac{1\cdot1}{M}} & \cdots & e^{-j2\pi\frac{1\cdot(M-1)}{M}} \\ \cdots & \cdots & \cdots & \cdots \\ e^{-j2\pi\frac{(M-1)\cdot0}{M}} & e^{-j2\pi\frac{(M-1)\cdot1}{M}} & \cdots & e^{-j2\pi\frac{(M-1)\cdot(M-1)}{M}} \end{bmatrix}$$

$$G_2=\begin{bmatrix} e^{-j2\pi\frac{0\cdot0}{N}} & e^{-j2\pi\frac{0\cdot1}{N}} & \cdots & e^{-j2\pi\frac{0\cdot(N-1)}{N}} \\ e^{-j2\pi\frac{1\cdot0}{N}} & e^{-j2\pi\frac{1\cdot1}{N}} & \cdots & e^{-j2\pi\frac{1\cdot(N-1)}{N}} \\ \cdots & \cdots & \cdots & \cdots \\ e^{-j2\pi\frac{(N-1)\cdot0}{N}} & e^{-j2\pi\frac{(N-1)\cdot1}{N}} & \cdots & e^{-j2\pi\frac{(N-1)\cdot(N-1)}{N}} \end{bmatrix}$$

利用这个性质再次进行编程，我们会发现计算速度会大幅度提高，远超直接计算速度。



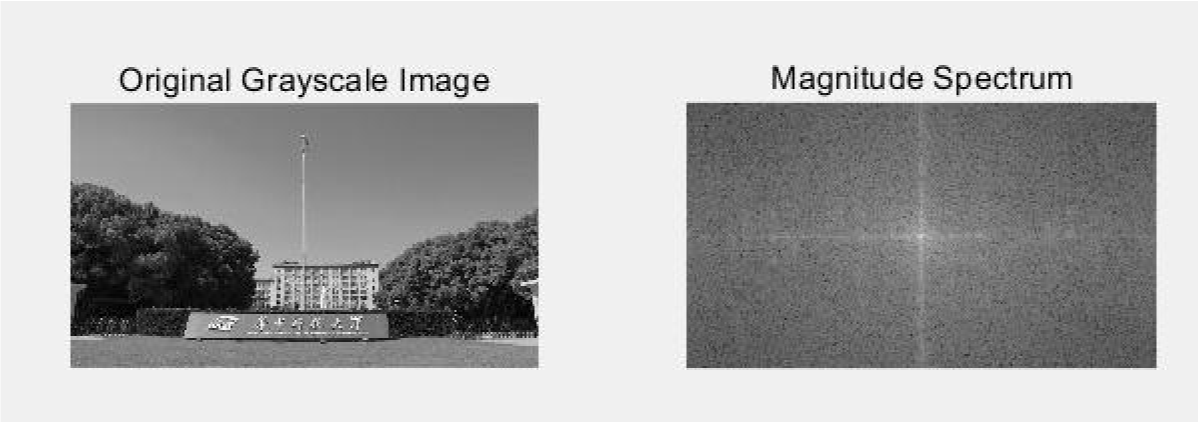
图八　　可分离性计算过程展示

3.2 代码

```
function magnitude_spectrum = perform_dft(gray_img)
% 下面是傅里叶正变换必备的一些矩阵:
[M,N]=size(gray_img);
Wm = exp(-1i*2*pi/M);
Wn = exp(-1i*2*pi/N); % 不同 G 中用不同的 W
Em = zeros(M);
En = zeros(N); % E 是辅助计算矩阵
Gm = zeros(M)+Wm;
```

```
Gn = zeros(N)+Wn; % G 是计算时要用的矩阵
F = zeros(M,N); % F 是转换到频域的结果
E = zeros(M,N);
% 对 Gm 的计算: 循环长度为 M
fprintf('二维离散傅里叶变换开始:\n');
for row = 0:M-1
for col = 0:M-1
Em(row+1,col+1) = row * col;
Gm(row+1,col+1) = Gm(row+1,col+1)^Em(row+1,col+1);
end
end
% 对 Gn 的计算: 循环长度为 N
for row = 0:N-1
for col = 0:N-1
En(row+1,col+1) = row * col;
Gn(row+1,col+1) = Gn(row+1,col+1)^En(row+1,col+1);
end
end
for row =1:M
%变换到图像中点
for col =1:N
E(row,col)=double(gray_img(row,col))*((-1)^(row+col));
end
end
F = real(Gm*E*Gn);

% 计算幅度谱并进行对数变换以增强可视化效果
magnitude_spectrum = log(1 + abs(F));

% 显示原始图像和傅里叶变换图像
figure;
subplot(1, 2, 1), imshow(gray_img), title('Original Grayscale Image');
subplot(1, 2, 2), imshow(magnitude_spectrum, []), title('Magnitude Spectrum');
end
```

3.2 结果分析



图九　　原图与傅里叶变换图像

根据得到的幅度谱图像我们可以看出，这幅图既含有低频成分，又含有高频成分，其中心更亮表明图中低频成分占主导地位，这与图片相符，因为图中大部分区域是表示低频成分的平滑区域。高频成分所占据的比例也含有不少，这主要包含了图中教学楼，左右两侧树木中边缘部分。在幅度谱中，较亮的部分通常对应于图像中的边缘和纹理区域。这些区域在空间域中变化较快，因此在频域中具有较高的频率成分。.

同时，我们也比较了 MATLAB 自带库中傅里叶变换的结果与我们手动编程的结果，我们直接对比了结果矩阵，最大误差小于 0.0001，基本可以忽略不计。