一、傅里叶变换和傅里叶系数的物理含义：

傅里叶变换是将一个**时域信号**（或空间域信号）转化为**频域信号**的工具。简单来说，傅里叶变换用于将信号分解为不同频率成分的叠加。这种分解可以让我们分析信号的频率结构，物理过程可以看作是信号在频率轴上的重新表达，它告诉我们信号中包含哪些频率成分以及每个频率成分的强度。

傅里叶系数是傅里叶变换或傅里叶级数展开中的一组参数，这些系数描述了信号的不同频率成分的**幅度**和**相位**。每个傅里叶系数对应于信号在某个特定频率上的贡献，可以被理解为信号的“频谱”，即信号是由哪些频率成分组成，以及它们的相对强度和相位。

另外的，在**数字图像处理**中，傅里叶变换和傅里叶系数具有重要的物理含义，帮助我们理解和分析图像中的空间频率成分，特别是在图像的**边缘检测**、**图像压缩**、**去噪**和**滤波**等领域。

在图像处理中，傅里叶变换将图像从**空间域**转换到**频率域**。图像在空间域是像素的二维排列，而在频率域中，每个像素表示图像中某一特定频率的强度。这个频率可以理解为图像中亮度或颜色变化的速率。傅里叶变换可以帮助我们将图像分解成不同的频率成分，从而理解图像是如何在不同的空间尺度上变化的。低频成分对应于图像的全局特征，而高频成分对应于局部的细节特征。

傅里叶系数在图像中的物理意义与信号处理类似：图像的每个细节（如边缘或纹理）都可以被看作是不同频率的叠加。傅里叶系数告诉我们图像在哪些频率上有强烈的变化，以及这些变化在图像中的位置。

二、

将所给图像转化为灰度图像：

```python
def load_image_as_gray(image_path):
    img = cv2.imread(image_path)
    if img is None:
        raise ValueError(f"无法加载图像：{image_path}")
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return gray_img
```

要将图像顺时针旋转，首先一点是图像是绕着图像的中心点进行旋转的，我先算出旋转后图像的尺寸，算出旋转前后图像的中心点，根据旋转的角度遍历新图像的每个像素，将新图像坐标映射到原图像坐标，在这里采用最近邻插值对一些像素进行处理，如果原图坐标在图像范围内，则将原图像素值付给旋转后图像，从而得到旋转后的图像。

```python
# 2. 计算旋转后的图像尺寸
def calculate_new_dimensions(h, w, angle):
    radians = math.radians(angle)
    new_w = int(abs(h * math.sin(radians)) + abs(w * math.cos(radians)))
    new_h = int(abs(h * math.cos(radians)) + abs(w * math.sin(radians)))
    return new_w, new_h

# 3. 进行最近邻插值，将图像中旋转的点映射到新图像
def rotate_image_manual(image, angle):
    h, w = image.shape
    new_w, new_h = calculate_new_dimensions(h, w, angle)

    rotated_image = np.zeros((new_h, new_w), dtype=np.uint8)

    # 计算旋转角度的弧度值
    radians = math.radians(angle)
    cos_angle = math.cos(radians)
    sin_angle = math.sin(radians)

    # 找到原图像中心和新图像中心
    cx, cy = w // 2, h // 2
    new_cx, new_cy = new_w // 2, new_h // 2

    # 遍历新图像的每个像素，找到它在原图中的位置
    for new_y in range(new_h):
        for new_x in range(new_w):
            # 将新图像坐标映射到原图像坐标（逆旋转）
            original_x = int((new_x - new_cx) * cos_angle + (new_y - new_cy) * sin_angle + cx)
            original_y = int(-(new_x - new_cx) * sin_angle + (new_y - new_cy) * cos_angle + cy)

            # 如果原图坐标在图像范围内，则将原图像素值赋给旋转后的图像
            if 0 <= original_x < w and 0 <= original_y < h:
                rotated_image[new_y, new_x] = image[original_y, original_x]

    return rotated_image
```

采用最近邻插值对图像进行放大，就是计算新图像的每个像素在原图像中的位置，并简单地选择最接近的原图像素值作为插值结果。

```python
# 最近邻插值方法，放大图像
def nearest_neighbor_resize(image, scale_factor):
    h, w = image.shape
    new_h, new_w = int(h * scale_factor), int(w * scale_factor)
    resized_image = np.zeros((new_h, new_w), dtype=np.uint8)

    for new_y in range(new_h):
        for new_x in range(new_w):
            # 映射到原图像的最近邻坐标（找到最近的整数坐标）
            original_x = int(new_x / scale_factor)
            original_y = int(new_y / scale_factor)

            # 赋值最近邻的像素值给新的像素点
            resized_image[new_y, new_x] = image[original_y, original_x]

    return resized_image
```

与最近邻插值不同，双线性插值通过对目标像素周围的四个相邻像素进行加权平均来确定新像素的值，结果更加平滑。

```python
def bilinear_resize(image, scale_factor):
    new_h, new_w = int(h * scale_factor), int(w * scale_factor)

    resized_image = np.zeros((new_h, new_w), dtype=np.uint8)

    for new_y in range(new_h):
        for new_x in range(new_w):
            # 计算新图像坐标在原图像中的浮点位置
            src_x = new_x / scale_factor
            src_y = new_y / scale_factor

            # 获取原图像中该点的四个相邻像素的坐标
            x0 = int(src_x)
            y0 = int(src_y)
            x1 = min(x0 + 1, w - 1)  # 防止越界
            y1 = min(y0 + 1, h - 1)  # 防止越界

            # 计算权重，表示当前位置与相邻像素的距离
            dx = src_x - x0
            dy = src_y - y0

            # 获取四个相邻像素的值
            top_left = image[y0, x0]
            top_right = image[y0, x1]
            bottom_left = image[y1, x0]
            bottom_right = image[y1, x1]

            # 根据水平距离 dx 进行线性插值
            top = top_left * (1 - dx) + top_right * dx
            bottom = bottom_left * (1 - dx) + bottom_right * dx

            # 根据垂直距离 dy 进行线性插值
            value = top * (1 - dy) + bottom * dy

            # 赋值给新图像，确保值在 0 到 255 之间
            resized_image[new_y, new_x] = np.clip(value, 0, 255)

    return resized_image
```

原始灰度图和旋转后的灰度图，如下所示：

785x422 像素

900x755 像素

用两种方法放大后的图像如下:

1580x884 像素

3140x1768 像素

三、

对所得灰度图展开傅里叶变换，提取傅里叶变换图像（将频率原点移至图像中心），在这里，我先实现一维 DFT，它用于将图像的一行或一列转换到频域。

```
# 一维傅里叶变换
def dft1d(signal):
    N = len(signal)
    n = np.arange(N)
    k = n.reshape((N, 1))  # 创建列向量
    exponent = -2j * np.pi * k * n / N  # 指数项
    dft_matrix = np.exp(exponent)  # 生成DFT矩阵
    return np.dot(dft_matrix, signal)  # 进行矩阵乘法，得到DFT结果
```

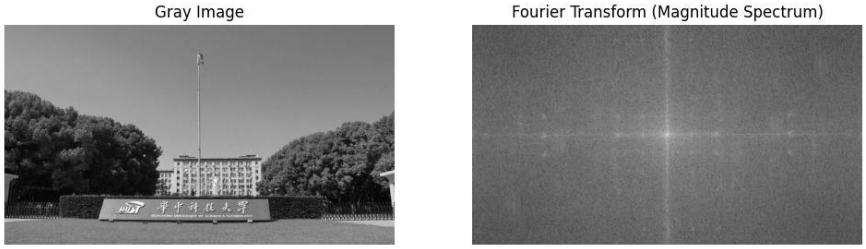
再进一步实现二维的 DFT，通过将二维 DFT 分解为两次一维 DFT（先对行进行，再对列进行），可以大幅优化运算。

```
# 二维傅里叶变换
def dft2d(image):
    M, N = image.shape  # 获取图像尺寸

    # 对每一行进行1D DFT
    dft_rows = np.array([dft1d(row) for row in image])

    # 对每一列进行1D DFT
    dft_cols = np.array([dft1d(col) for col in dft_rows.T]).T  # 对列进行1D DFT，再转回来

    return dft_cols
```

再进一步实现频谱中心化

```
# 自己实现的fftshift
def fftshift_2d(dft_result):
    M, N = dft_result.shape
    return np.roll(np.roll(dft_result, M // 2, axis=0), N // 2, axis=1)
```

最终结果如图：



在变换的图像中，在中心有一条明显的十字交叉线。这是因为图像中存在明显的水平线条（如建筑物的顶部和底部的水平边缘）和垂直线条（如建筑物两侧的垂直边缘）。这些特征在频谱中分别映射为沿垂直轴和水平轴的频率分量，从而在频谱上形成了一个十字形。整体上这个结果展示了图像在频域中的信息，低频分量集中在图像的中心，表示图像的整体结构，而高频分量则对应于图像中的细节或边缘。