图像处理与可视化 Homework 07

学号: 21307140051
姓名: 雍崔扬

Problem 1

编程实现:

- ① 仿射变换
- ② 局部仿射变换
- ③ 基于 FFD 的空间变换

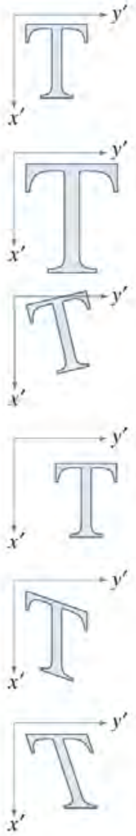
1.1 仿射变换

仿射变换 (affine transformation) 保留二维空间中的点、直线和平面.
它包含缩放变换 (scaling)、平移变换 (translation)、旋转变换 (rotation) 和剪切变换 (shearing)
我们可用**齐次坐标** (homogeneous coordinates) 来表示仿射变换:

$$\text{2D:} \begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \\ 1 \end{bmatrix} = A \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

TABLE 2.3
Affine transformations based on Eq. (2-45).

Transformation Name	Affine Matrix, A	Coordinate Equations	Example
Identity	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x$ $y' = y$	
Scaling/Reflection (For reflection, set one scaling factor to −1 and the other to 0)	$\begin{bmatrix} c_x & 0 & 0 \\ 0 & c_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = c_x x$ $y' = c_y y$	
Rotation (about the origin)	$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x\cos\theta - y\sin\theta$ $y' = x\sin\theta + y\cos\theta$	
Translation	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x + t_x$ $y' = y + t_y$	
Shear (vertical)	$\begin{bmatrix} 1 & s_v & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x + s_v y$ $y' = y$	
Shear (horizontal)	$\begin{bmatrix} 1 & 0 & 0 \\ s_h & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x$ $y' = s_h x + y$	



Python 代码:

```python
def apply_affine_transformation(transformation_types, params=None):
```

```
    """
    Apply a series of affine transformations (scaling, rotation, translation,
shear_vertical, shear_horizontal)
    by multiplying their corresponding transformation matrices together.

    Parameters:
    - transformation_types: A list of transformation types (e.g., ['scaling',
'rotation', 'translation']).
    - params: A list of parameter dictionaries, each corresponding to a
transformation in `transformation_types`.
        Example:
        [
            {'cx': 2, 'cy': 2},       # for scaling
            {'theta': 45},            # for rotation
            {'tx': 10, 'ty': 20},     # for translation
            {'sv': 0.5},              # for shear_vertical
            {'sh': 0.5}               # for shear_horizontal
        ]

    Returns:
    - A: The combined affine transformation matrix as a result of all the
transformations.
    """

    # Initialize the result matrix as the identity matrix
    A_total = np.eye(3)

    # Process each transformation type and apply the corresponding transformation
matrix
    for transformation_type, param in zip(transformation_types, params):
        if transformation_type == 'scaling':
            cx = param.get('cx', 1)
            cy = param.get('cy', 1)
            A = np.array([
                [cx, 0, 0],
                [0, cy, 0],
                [0, 0, 1]
            ])
        elif transformation_type == 'rotation':
            theta = np.radians(param.get('theta', 0))  # Convert to radians
            A = np.array([
                [np.cos(theta), -np.sin(theta), 0],
                [np.sin(theta), np.cos(theta), 0],
                [0, 0, 1]
            ])
        elif transformation_type == 'translation':
            tx = param.get('tx', 0)
            ty = param.get('ty', 0)
            A = np.array([
                [1, 0, tx],
                [0, 1, ty],
                [0, 0, 1]
            ])
        elif transformation_type == 'shear_vertical':
            sv = param.get('sv', 0)
            A = np.array([
                [1, sv, 0],
                [0, 1, 0],
```

```
                [0, 0, 1]
            ])
        elif transformation_type == 'shear_horizontal':
            sh = param.get('sh', 0)
            A = np.array([
                [1, 0, 0],
                [sh, 1, 0],
                [0, 0, 1]
            ])
        else:
            raise ValueError(f"Unknown transformation type: 
{transformation_type}")

        # Apply the transformation matrix by multiplying with the cumulative 
result matrix
        A_total = np.dot(A_total, A)

    return A_total
```

1.2 局部仿射

给定 n 个局部区域 R_i，分别对应仿射变换 T_i
记 (x, y) 到区域 R_i 的距离为 $d_i(x, y)$
定义 $(x, y) \notin \bigcup_{i=1}^{n} \mathrm{R}_i$ 关于区域 R_i 的权重为:
(良好的权重定义要求同一点对应的权重之和为 1)

$$w_i(x, y) := \frac{\frac{1}{(d_i(x,y))^p}}{\sum_{k=1}^{n} \frac{1}{(d_k(x,y))^p}}$$

其中 p 的一个比较好的选择是 $p = 1.5$
则局部仿射变换 T 由以下公式给出:

$$T(x, y) := \begin{cases} T_i(x, y) & \text{if } (x, y) \in \mathrm{R}_i \text{ for a certain } i = 1, \ldots, n \\ \sum_{i=1}^{n} w_i(x, y) T_i(x, y) & \text{otherwise} \end{cases}$$

基本思想: 区域控制 $+$ 全局影响.
优化方法:

- 将区域控制调整为点控制 (即将区域 R_i 调整为单个控制点 $\phi_i = (x_i, y_i)$)
- 将全局影响调整为局部影响 (即将全局性权重调整为局部性权重，使得控制点只对局部区域有影响，减少计算开销)

Python 代码:

```
def inversely_transform_image_local(image, region, transformation, 
output_shape=None, p=1.5, interpolation_method="bilinear"):

    if output_shape is None:
        output_shape = image.shape

    height, width = output_shape
    # the boundary of image
    source_coords = np.zeros((height, width, 2))
    for y in range(height):
        source_coords[y, 0, :] = [y, 0]
```

```
        source_coords[y, width-1, :] = [y, width-1]
    for x in range(width):
        source_coords[0, x, :] = [0, x]
        source_coords[height-1, x, :] = [height-1, x]

    A_inv = np.linalg.inv(transformation)
    y, x = np.meshgrid(np.arange(1, height-1), np.arange(1, width-1),
indexing="ij")
    region_coords = np.stack((y.ravel(), x.ravel(), np.ones_like(x.ravel())),
axis=-1)
    region_coords = region_coords @ A_inv.T
    region_coords = region_coords[..., :2]
    region_coords = region_coords.reshape((height-2, width-2, 2))

    y_min, y_max = region[0]
    x_min, x_max = region[1]
    mask = (
        (region_coords[..., 0] >= x_min) & (region_coords[..., 0] <= x_max) &
        (region_coords[..., 1] >= y_min) & (region_coords[..., 1] <= y_max)
    )
    valid_indices = np.argwhere(mask)
    for idx in valid_indices:
        i, j = idx
        source_coords[i+1, j+1, :] = region_coords[i, j, :]

    invalid_indices = np.argwhere(~mask)
    for idx in invalid_indices:
        i, j = idx
        i = i + 1
        j = j + 1
        top_distance = i
        bottom_distance = height - 1 - i
        left_distance = j
        right_distance = width - 1 - j
        x_project = np.clip(j, x_min + 1, x_max + 1)
        y_project = np.clip(i, y_min + 1, y_max + 1)
        region_distance = np.linalg.norm(np.array([
            np.maximum(1, abs(y_project - i)),
            np.maximum(1, abs(x_project - j))
        ]))
        distance = np.array([top_distance, bottom_distance, left_distance,
right_distance, region_distance])
        weight = 1 / (distance ** p)
        weight = weight / np.sum(weight)
        source_coords[i, j] = (weight[0] + weight[1] + weight[2] + weight[3]) *
np.array([i, j]
                              ) + weight[4] * region_coords[i-1, j-1]

    # Interpolate pixel values from the source image
    transformed_image = interpolate(image, source_coords,
method=interpolation_method)

    return transformed_image
```

1.3 FFD

使用径向基函数会导致每个控制点对变换都有全局影响，这可能会导致局部形变的建模变得困难.
此外，当控制点的数量较多时，径向基函数样条的计算代价也会变得非常昂贵.
一个替代方案是使用**自由形状变换** (Free-Form Deformation, FFD)
(点控制 + 局部影响 + 规则化控制点)
其基本思想是通过操作一个控制点网格来变形物体.
由此产生的形变控制了 3D 物体的形状，并产生平滑连续的变换.
与允许控制点任意配置的径向基函数样条不同，
基于样条的 FFD 需要一个规则的控制点网格，且控制点之间均匀间隔.

考虑一个基于 B-样条的 FFD 模型.
我们将二维图像表示为一个网格图，每个网格交叉点代表一个控制点.
设 x 轴方向上网格间距为 δ_x，分为 n_x 段，而 y 轴方向上网格间距为 δ_y，分为 n_y 段.
(医学图像配准通常选择 $\delta_x = \delta_y = 4$)
记控制点网格为 $\Phi = [\phi_{i,j}]$ (其中 $0 \le i \le n_x, 0 \le j \le n_y$)
为配准图像中的控制点，我们设其在周围四个控制点的网格范围内移动.
根据 B-样条理论，点 (x, y) 到 $(x + \Delta x, y + \Delta y)$ 的位移量可表示为:

$$
\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \sum_{l=0}^{3}\sum_{k=0}^{3} B_l(u)B_k(v)\phi_{i+l,j+k} \quad \text{where} \begin{cases} i = \left\lfloor \frac{x}{\delta_x} \right\rfloor - 1 \\ u = \frac{x}{\delta_x} - \left\lfloor \frac{x}{\delta_x} \right\rfloor \\ j = \left\lfloor \frac{y}{\delta_y} \right\rfloor - 1 \\ v = \frac{y}{\delta_y} - \left\lfloor \frac{y}{\delta_y} \right\rfloor \end{cases}
$$

三次 B-样条函数的定义为: (其中 $u \in [0, 1)$)
(事实上它们是三次 B-样条函数的 4 段, 这样拆分可以方便代码实现)

$$
\begin{aligned}
B_0(u) &:= \frac{1}{6}(1-u)^3 \\
B_1(u) &:= \frac{1}{6}(3u^3 - 6u^2 + 4) \\
B_2(u) &:= \frac{1}{6}(-3u^3 + 3u^2 + 3u + 1) \\
B_3(u) &:= \frac{1}{6}u^3
\end{aligned}
$$

Python 代码:

```
def inversely_transform_image_ffd(image, n_x, n_y, shift_dict, interpolation_method="bilinear"):
    """
    Perform inverse FFD to transform the image based on control point displacements.
    """
    # Initialize control points
    def return_no_shift():
        return np.zeros(2)

    # Use defaultdict to store the shifts of control points (default value is zero displacement)
    control_points = defaultdict(lambda: np.zeros(2))  # Default is [0, 0] for control points
    for (i, j), (delta_y, delta_x) in shift_dict.items():
```

```
        control_points[(i, j)] = np.array([delta_y, delta_x])  # Store
displacement for control points

    # Get image dimensions
    height, width = image.shape
    ly = height / n_y  # Grid cell height
    lx = width / n_x   # Grid cell width

    # Prepare a mesh grid for source coordinates
    y, x = np.meshgrid(np.arange(height), np.arange(width), indexing="ij")
    source_coords = np.stack((y.ravel(), x.ravel()), axis=-1)
    source_coords = source_coords.reshape((height, width, 2)).astype(np.float64)

    for x_idx in range(width):  # Iterate over all pixels in the image
        ix = np.floor(x_idx / lx) - 1
        u = (x_idx / lx) - np.floor(x_idx / lx)
        beta_u = beta(u)

        for y_idx in range(height):
            iy = np.floor(y_idx / ly) - 1
            v = (y_idx / ly) - np.floor(y_idx / ly)
            beta_v = beta(v)

            # Sum over the neighboring control points using the B-spline weights
            for l in range(4):
                for k in range(4):
                    source_coords[y_idx, x_idx, :] += beta_u[l] * beta_v[k] *
control_points[(ix + l,

             iy + k)]

    # Perform interpolation on the transformed coordinates
    transformed_image = interpolate(image, source_coords,
method=interpolation_method)
    return transformed_image
```

Problem 2

将 Problem 1 中的空间变换应用于基于反向的图像变换过程.

2.1 图像内插

内插 (interpolation) 是用已知数据来估计未知位置的值的过程，
它通常在图像放大、缩小、旋转和几何校正等任务中使用.

赋值的方法有以下几种:

- **① 最邻近内插 (nearest neighbor interpolation):**
  将原图像中最近邻的灰度作为新图像中待求位置的灰度.
  这种方法简单，但会产生一些人为失真，例如严重的直边失真.
- **② 双线性内插 (bilinear interpolation):**
  使用尺寸为 $M \times N$ 的原图像 f 中的 4 个最近邻的灰度来计算新图像 g 中待求位置 (x, y) 的灰度.
  取 x, y 的小数部分为 dx, dy，记四个邻近点的灰度值为 $I_{11}, I_{12}, I_{22}, I_{21}$ (左上, 右上, 右下, 左下)

$$
\begin{gathered}
dx = x - \lfloor x \rfloor \\
dy = y - \lfloor y \rfloor \\
I_{11} = f(\lfloor x \rfloor, \lfloor y \rfloor) \\
I_{12} = f(\lfloor x \rfloor, \min\{\lfloor y \rfloor + 1, N-1\}) \\
I_{21} = f(\min\{\lfloor x \rfloor + 1, M-1\}, \lfloor y \rfloor) \\
I_{22} = f(\min\{\lfloor x \rfloor + 1, M-1\}, \min\{\lfloor y \rfloor + 1, N-1\}) \\
\hline
g(x,y) = I_{11}(1-dx)(1-dy) + I_{12}(1-dx)dy + I_{21}dx(1-dy) + I_{22}dxdy
\end{gathered}
$$

- **③ 双三次内插 (bicubic interpolation):**
  使用原图像 f 中的 16 个最近邻的灰度来计算新图像中待求位置 (x, y) 的灰度.

  $$g(x,y) = \sum_{i,j=0}^{3} a_{ij} f(x_i, y_j)$$

  其中 16 个系数 a_{ij} $(i,j = 0,1,2,3)$ 由点 (x,y) 的 16 个最近邻点 (x_i, x_j) $(i,j = 0,1,2,3)$ 的梯度和 Hessian 矩阵求出.
  BiCubic 基函数为:

  $$
  \begin{gathered}
  W(t) := \begin{cases} \frac{3}{2}|t|^3 - \frac{5}{2}|t|^2 + 1 & \text{if } 0 \le |t| \le 1 \\ -\frac{1}{2}|t|^3 + \frac{5}{2}|t|^2 - 4|t| + 2 & \text{if } 1 < |t| < 2 \\ 0 & \text{otherwise} \end{cases} \\
  a_{ij} = W(x - x_i) W(y - y_j) \ (i,j = 0,1,2,3)
  \end{gathered}
  $$

2.2 仿射变换

```
def inversely_transform_image(image, A, output_shape=None,
interpolation_method="bilinear"):
    """
    Inversely transform an image using a specified affine transformation

    Parameters:
        image (np.ndarray): Input image as a 2D array.
        A: Affine transformation
        output_shape (tuple): Shape of the output image (height, width).
        interpolation_method (str): Interpolation method ('nearest', 'bilinear',
'bicubic').

    Returns:
        np.ndarray: Transformed image with the specified output shape.
    """
    if output_shape is None:
        output_shape = image.shape
        print(output_shape)

    # Compute the inverse of the transformation matrix
    A_inv = np.linalg.inv(A)

    # Generate the coordinate grid for the target image
    height, width = output_shape
    y, x = np.meshgrid(np.arange(height), np.arange(width), indexing="ij")
    target_coords = np.stack((y.ravel(), x.ravel(), np.ones_like(x.ravel())),
axis=-1)
```

```
    # Apply the inverse transformation to map target coordinates back to source
coordinates
    source_coords = target_coords @ A_inv.T
    source_coords = source_coords[..., :2] # Normalize homogeneous coordinates

    # Reshape the source coordinates to match the target shape
    source_indices = source_coords.reshape((height, width, 2))

    # Interpolate pixel values from the source image
    transformed_image = interpolate(image, source_indices,
method=interpolation_method)

    return transformed_image
```

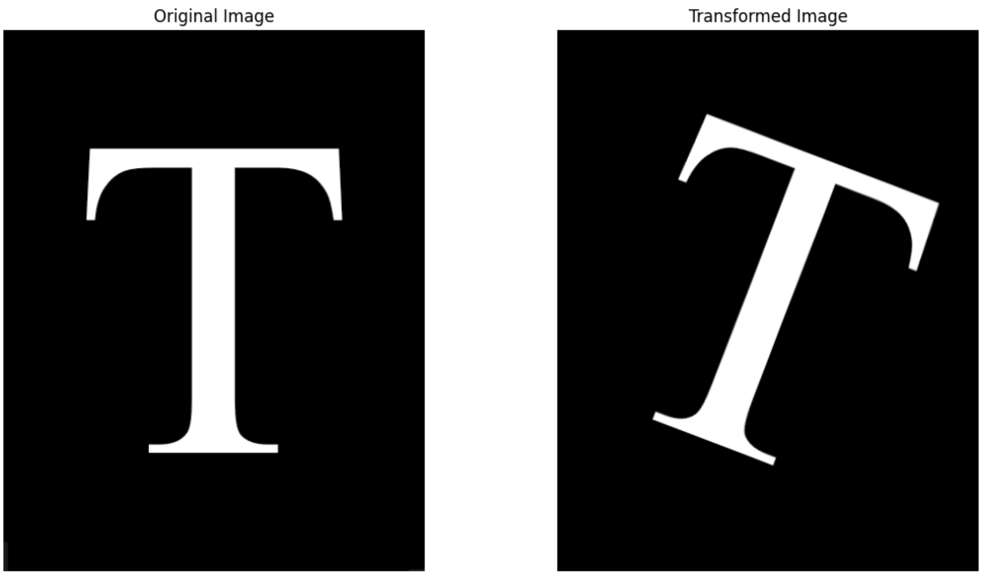
函数调用:

```
if __name__ == "__main__":
    image_path = 'DIP Fig 02.36(a)(letter_T).tif'
    image_name = 'DIP Fig 02.36(a)(letter_T)'
    image = Image.open(image_path).convert('L')  # Convert the image to
grayscale ('L' mode)
    image = np.array(image)  # Convert the grayscale image to a numpy array

    # Affine transformation type
    # ('scaling', 'rotation', 'translation', 'shear_vertical',
'shear_horizontal')
    # Example:
        # [
        #     {'cx': 2, 'cy': 2},        # for scaling
        #     {'theta': 45},             # for rotation
        #     {'tx': 10, 'ty': 20},      # for translation
        #     {'sv': 0.5},               # for shear_vertical
        #     {'sh': 0.5}                # for shear_horizontal
        # ]
    height, width = image.shape
    option = 2
    if option == 1:
        transformation_types = ['translation', 'rotation', 'translation']
        params = [{'tx': 0.5 * height, 'ty': 0.5 * width},
                {'theta': -21},
                {'tx': -0.5 * height, 'ty': -0.5 * width}]
    else:
        transformation_types = ['translation', 'scaling', 'translation']
        params = [{'tx': 0.5 * height, 'ty': 0.5 * width},
                {'cx': 0.7, 'cy': 1.3},
                {'tx': -0.5 * height, 'ty': -0.5 * width}]

    interpolation_method="bicubic"

    # Get the combined affine transformation matrix
    A = apply_affine_transformation(transformation_types, params)
    print(A)

    # Output shape is the same as the input image
    output_shape = np.array(image.shape)

    # Apply the inverse transformation to the image
```

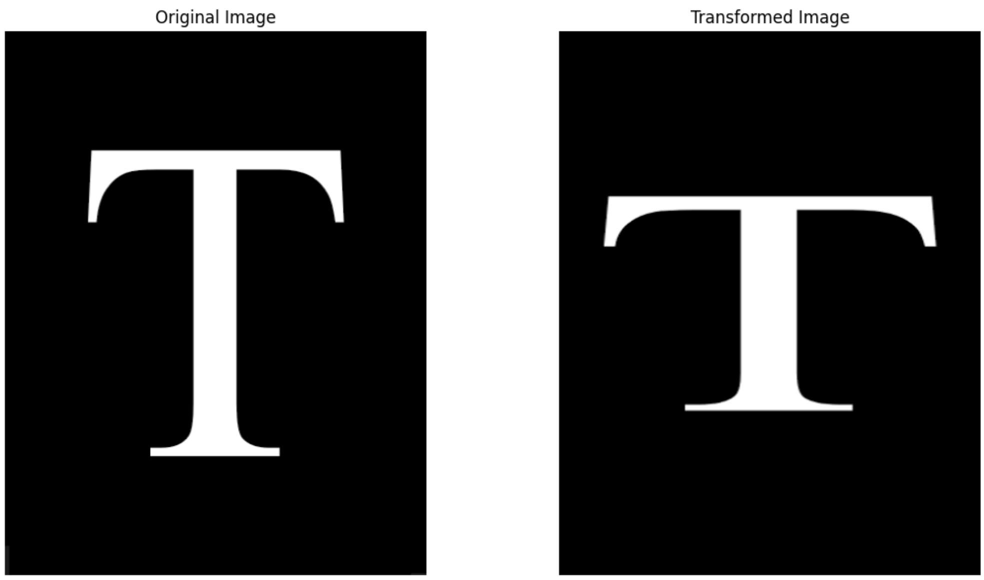
```
    transformed_image = inversely_transform_image(
        image, A=A, output_shape=output_shape,
interpolation_method=interpolation_method
    )

    # Plot the original and processed images side by side for comparison
    plt.figure(figsize=(12, 6))

    # Plot the original image
    plt.subplot(1, 2, 1)
    plt.imshow(image, cmap='gray')
    plt.title("Original Image")
    plt.axis('off')  # Hide axis ticks

    # Plot the transformed image
    plt.subplot(1, 2, 2)
    plt.imshow(transformed_image, cmap='gray')
    plt.title("Transformed Image")
    plt.axis('off')  # Hide axis ticks

    # Display the comparison
    plt.tight_layout()
    save_path = f"Comparision-{image_name}.png"
    plt.savefig(save_path)
    plt.show()
```

运行结果:

- ① 绕图像中心逆时针旋转 21°



- ② 以图像中心为原点，纵向缩小为原来的 0.7 倍，横向放大为原来的 1.3 倍



2.3 局部仿射

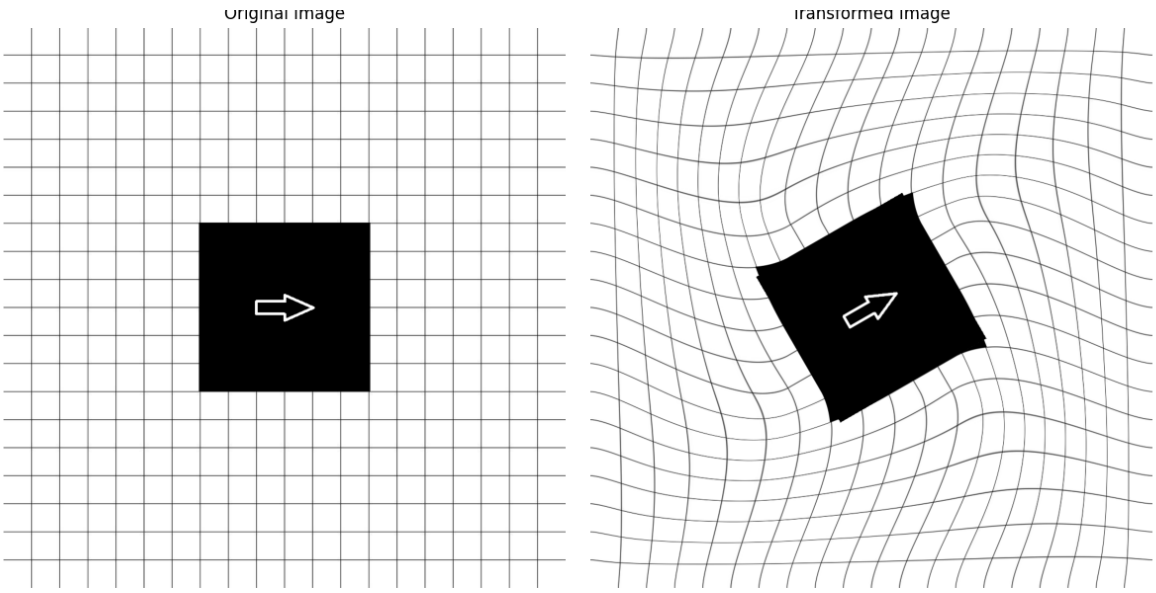
函数调用:

```
if __name__ == "__main__":
    image_path = 'grid.png'
    image_name = 'grid'
    image = Image.open(image_path).convert('L')  # Convert the image to 
grayscale ('L' mode)
    image = np.array(image)  # Convert the grayscale image to a numpy array

    # Affine transformation type ('scaling', 'rotation', 'translation', 
'shear_vertical', 'shear_horizontal')
    # Example:
        # [
        #     {'cx': 2, 'cy': 2},        # for scaling
        #     {'theta': 45},             # for rotation
        #     {'tx': 10, 'ty': 20},      # for translation
        #     {'sv': 0.5},               # for shear_vertical
        #     {'sh': 0.5}                # for shear_horizontal
        # ]
    height, width = image.shape
    transformation_types = ['translation', 'rotation', 'translation']
    params = [{'tx': 0.5 * height, 'ty': 0.5 * width},
              {'theta': 30},
              {'tx': -0.5 * height, 'ty': -0.5 * width}]
    # Get the combined affine transformation matrix
    A = apply_affine_transformation(transformation_types, params)
    region = [[358, 665], [358, 665]]
    interpolation_method = "bicubic"

    # Output shape is the same as the input image
    output_shape = np.array(image.shape)

    # Apply the inverse transformation to the image
    transformed_image = inversely_transform_image_local(image, region=region,
                                                        transformation=A,
output_shape=output_shape,
```

```
                                                p=1.5,
interpolation_method=interpolation_method)

    # Plot the original and processed images side by side for comparison
    plt.figure(figsize=(12, 6))

    # Plot the original image
    plt.subplot(1, 2, 1)
    plt.imshow(image, cmap='gray')
    plt.title("Original Image")
    plt.axis('off')  # Hide axis ticks

    # Plot the transformed image
    plt.subplot(1, 2, 2)
    plt.imshow(transformed_image, cmap='gray')
    plt.title("Transformed Image")
    plt.axis('off')  # Hide axis ticks

    # Display the comparison
    plt.tight_layout()
    save_path = f"Comparision-{image_name}.png"
    plt.savefig(save_path)
    plt.show()
```

运行结果:

- 图像边框像素和黑色区域作为控制区域
- 图像边框像素不变 (或者说作用恒等变换)，黑色区域绕图像中心顺时针旋转 30°



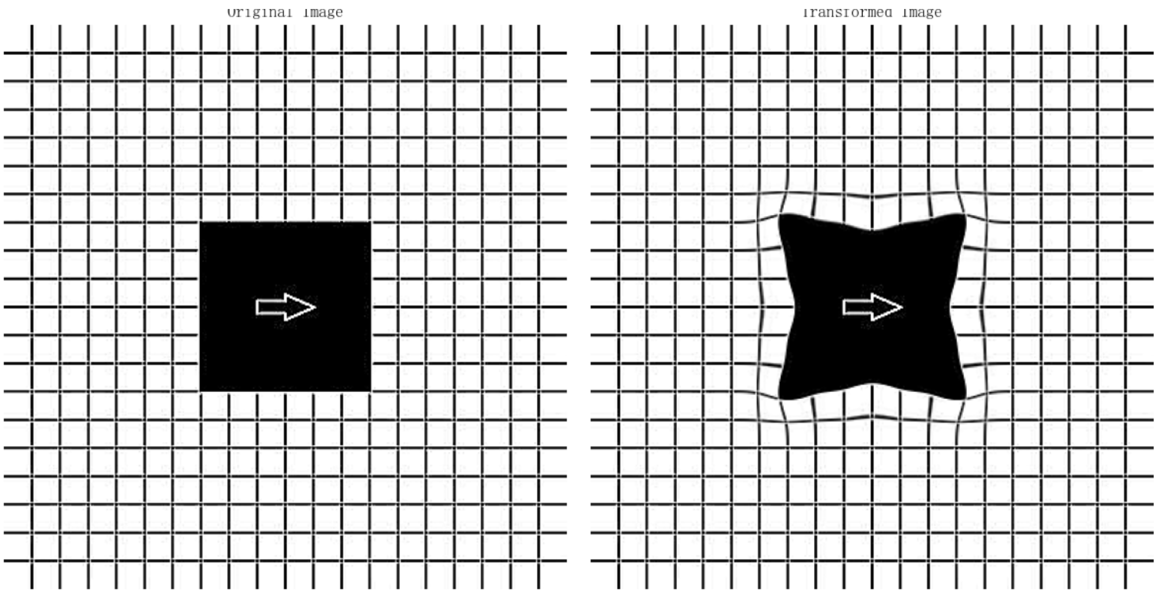
2.4 FFD

函数调用:

```
if __name__ == "__main__":
    # Image loading and preprocessing
    image_path = 'grid.png'
    image_name = 'grid'
    image = Image.open(image_path).convert('L')  # Convert to grayscale
```

```
    image = np.array(image)

    # Initialize the shift dictionary for control points
    n_x = 20
    n_y = 20
    height, width = image.shape
    lx = width / n_x
    ly = height / n_y

    # Define control points shift
    shift_dict = dict()
    shift_dict[(7, 7)] = np.array([15, 15])
    shift_dict[(7, 10)] = np.array([0, -15])
    shift_dict[(10, 7)] = np.array([-15, 0])
    shift_dict[(7, 13)] = np.array([-15, 15])
    shift_dict[(13, 7)] = np.array([15, -15])
    shift_dict[(13, 10)] = np.array([0, 15])
    shift_dict[(10, 13)] = np.array([15, 0])
    shift_dict[(13, 13)] = np.array([-15, -15])

    # Apply the inverse transformation to the image
    transformed_image = inversely_transform_image_ffd(image, n_x, n_y,
shift_dict,

 interpolation_method="bilinear")

    # Plot the original and processed images side by side for comparison
    plt.figure(figsize=(12, 6))

    # Plot the original image
    plt.subplot(1, 2, 1)
    plt.imshow(image, cmap='gray')
    plt.title("Original Image")
    plt.axis('off')  # Hide axis ticks

    # Plot the transformed image
    plt.subplot(1, 2, 2)
    plt.imshow(transformed_image, cmap='gray')
    plt.title("Transformed Image")
    plt.axis('off')  # Hide axis ticks

    # Display the comparison
    plt.tight_layout()
    save_path = f"Comparision-{image_name}.png"
    plt.savefig(save_path)
    plt.show()
```

运行结果:



The End