

作业二：传统插值算法实现

全部代码可以在<https://github.com/shuitatata/24Fall-ImageProcessing.git>中获取到。

实现功能

- 基本功能：
 - 手动实现最近邻、双线性插值、双三次插值算法
- 进阶功能：
 - 实现可选参数的图像旋转、平移、斜切

最近邻算法

最近邻算法的实现比较简单，重点在于找到变换后像素与变换前像素的坐标对应关系。设变换后像素坐标为 i, j ，则变换前的坐标为：

```
origin_i = round(i/scale)
origin_j = round(j/scale)
origin_i = min(origin_i, h-1)
origin_j = min(origin_j, w-1)
```

完整的处理过程如下：

```
def resize_nearest(image, scale):
    """
    Nearest neighbor interpolation
    """
    h, w = image.shape[:2]
    new_h, new_w = int(h*scale), int(w*scale)
    new_image = np.zeros((new_h, new_w, 3), dtype=np.uint8)
    for i in range(new_h):
        for j in range(new_w):
            origin_i = round(i/scale)
            origin_j = round(j/scale)
            origin_i = min(origin_i, h-1)
            origin_j = min(origin_j, w-1)
            new_image[i, j] = image[origin_i, origin_j]
    return new_image
```

双线性插值算法

与最近邻算法相比，双线性插值中每个变换后像素由四个变换前像素组成。一个自然的想法是仿照最近邻算法，使用 for 循环遍历变换后图像中的每一个像素点，再按照距离进行加权求和：

```
def resize_bilinear(image, scale):
    """
    Bilinear interpolation
    """
    h, w = image.shape[:2]
    new_h, new_w = int(h*scale), int(w*scale)
```

```

new_image = np.zeros((new_h, new_w, 3), dtype=np.uint8)
for i in range(new_h):
    for j in range(new_w):
        origin_i = i/scale
        origin_j = j/scale
        x1, y1 = int(origin_i), int(origin_j)
        x2, y2 = min(x1+1, h-1), min(y1+1, w-1)
        u, v = origin_i-x1, origin_j-y1
        new_image[i, j] = (1-u)*(1-v)*image[x1, y1] + u*(1-v)*image[x2, y1] +
(1-u)*v*image[x1, y2] + u*v*image[x2, y2]
    return new_image.astype(np.uint8)

```

但是这种写法中的 for 循环会带来比较大的性能开销，因此我们考虑使用向量化方法进行加速：

```

def resize_bilinear(image, scale):
    """
    Bilinear interpolation
    """
    h, w = image.shape[:2]
    new_h, new_w = int(h*scale), int(w*scale)

    row_indices = np.arange(new_h).reshape(-1, 1).repeat(new_w, axis=1) / scale
    col_indices = np.arange(new_w).reshape(1, -1).repeat(new_h, axis=0) / scale

    x1 = np.floor(row_indices).astype(int)
    y1 = np.floor(col_indices).astype(int)
    x2 = np.clip(x1 + 1, 0, h - 1)
    y2 = np.clip(y1 + 1, 0, w - 1)

    u = row_indices - x1
    v = col_indices - y1

    # 将u复制为三维
    u = np.expand_dims(u, axis=-1)
    v = np.expand_dims(v, axis=-1)

    print(u.shape)

    new_image = (1 - u) * (1 - v) * image[x1, y1] + \
        u * (1 - v) * image[x2, y1] + \
        (1 - u) * v * image[x1, y2] + \
        u * v * image[x2, y2]

    return new_image.astype(np.uint8)

```

经过实验，在较大的图片上，第二种方法确实可以带来明显的效率提升。

双三次插值算法

双三次插值算法的大体思路与双线性插值算法相似，不同点在于使用了三次函数而非线性函数进行插值，这意味着双三次插值算法中每一个像素点都是由原始图像中的16个像素点计算得来。

同样，为了提升性能，我们尝试使用向量化方法来加速计算，首先将三次插值函数的确定与计算过程抽象为 cubic 函数：

```
def cubic(p:list, x):
    p = p.astype(np.float32)
    a = p[..., 1, :]
    b = (p[..., 2, :] - p[..., 0, :]) / 2
    c = p[..., 0, :] + 2 * p[..., 2, :] - (5 * p[..., 1, :] + p[..., 3, :]) / 2
    d = (-p[..., 0, :] + 3 * p[..., 1, :] - 3 * p[..., 2, :] + p[..., 3, :]) / 2
    return a + b * x + c * x ** 2 + d * x ** 3
```

其中 p 存放着原始图像中的像素点，用于计算三次函数的参数，x为待求点离p[1]的距离。

接下来我们需要对每一个变换后的像素点，计算一个 patch，即原图像中一个 4×4 大小的区域，用于计算变换后像素的值：

```
x_new = np.linspace(0, h-1, new_h)
y_new = np.linspace(0, w-1, new_w)
x, y = np.meshgrid(x_new, y_new)

x_indices = np.clip(np.add.outer(x0, np.arange(-1, 3)), 0, h-1) #(h, w, 4)
y_indices = np.clip(np.add.outer(y0, np.arange(-1, 3)), 0, w-1) #(h, w, 4)

patch = image[x_indices[:, :, :, None], y_indices[:, :, None, :], :] #(h, w, 4, 3)
```

此时我们就得到了每一个像素点所对应的patch，每个patch是一个 $4 \times 4 \times 3$ 的数组，最后一个维度表示3个通道。

最后我们只需要对每个patch进行插值即可，由于之前已经整理为了向量的形式，我们避免了龟速的for循环，可以利用np自带的向量加速：

```
col = np.zeros((new_h, new_w, 4, 3), dtype=np.float32)
for m in range(4):
    # 横向插值
    col[:, :, m, :] = cubic(patch[:, :, m, :, :], dy[:, :, None])
    # 纵向插值
    new_image = np.clip(cubic(col, dx[:, :, None]), 0, 255)
```

斜切、平移、旋转操作

为了尽可能地复用，我将这三个操作抽象为了两个函数：get_xxxx_matrix() 和 apply_transform()。前一个函数负责根据参数生成变换矩阵，后一个参数则对图像运用变换矩阵。

为了加快计算速度，我采用了双线性插值而非双三次插值。

获取变换矩阵的函数比较简单：

```
def get_rotation_matrix(angle):
    angle = np.deg2rad(angle)
    return np.array([[np.cos(angle), -np.sin(angle), 0], [np.sin(angle),
np.cos(angle), 0], [0, 0, 1]])

def get_translation_matrix(tx, ty):
    return np.array([[1, 0, tx], [0, 1, ty], [0, 0, 1]])

def get_shear_matrix(kx, ky):
    return np.array([[1, kx, 0], [ky, 1, 0], [0, 0, 1]])
```

而根据变换矩阵进行变换的大体思路为：

- 生成变换后图像的坐标网格
- 计算变换矩阵的逆
- 二者相乘计算在原图像中的坐标
- 根据原图像中的坐标进行双线性插值

```
def apply_transform(image, matrix, output_shape=None):
    h, w, c = image.shape
    if output_shape is None:
        output_shape = (h, w)
    new_h, new_w = output_shape
    new_image = np.zeros((new_h, new_w, c), dtype=np.uint8)

    def valid_coord(x, y):
        return (0 <= x) & (x < h) & (0 <= y) & (y < w)

    # 生成目标图像的坐标网格
    x, y = np.meshgrid(np.arange(new_h), np.arange(new_w), indexing='ij') #
    Shape: (new_h, new_w)
    coords = np.stack([x, y, np.ones_like(x)], axis=-1).reshape(-1,
3).astype(np.float32) # Shape: (new_h*new_w, 2)

    # 计算逆变换矩阵
    try:
        inverse_matrix = np.linalg.inv(matrix)
    except LinAlgError:
        inverse_matrix = matrix
    print(inverse_matrix.shape)

    # 计算在原图中的坐标（逆映射）
    original_coords = coords @ inverse_matrix.T

    original_coords = original_coords.reshape(new_h, new_w, 3)
    print(image.shape, new_h)

    # 计算四角坐标
    x_orig = original_coords[:, :, 0] / original_coords[:, :, 2]
    y_orig = original_coords[:, :, 1] / original_coords[:, :, 2]
    x1 = np.floor(x_orig).astype(int)
    y1 = np.floor(y_orig).astype(int)
    x2 = x1 + 1
    y2 = y1 + 1
```

```

# 计算插值权重
u = (x_orig - x1)[..., None] # Shape: (new_h, new_w, 1)
v = (y_orig - y1)[..., None] # Shape: (new_h, new_w, 1)

# 使用双线性插值
x1_valid = np.clip(x1, 0, h - 1)
y1_valid = np.clip(y1, 0, w - 1)
x2_valid = np.clip(x1 + 1, 0, h - 1)
y2_valid = np.clip(y1 + 1, 0, w - 1)
print(image.shape)
print(x1.shape)
new_image = ((1 - u) * (1 - v) * np.where(valid_coord(x1, y1)[..., None],
image[x1_valid, y1_valid], 0) +
             u * (1 - v) * np.where(valid_coord(x1, y2)[..., None],
image[x1_valid, y2_valid], 0) +
             (1 - u) * v * np.where(valid_coord(x2, y1)[..., None],
image[x2_valid, y1_valid], 0) +
             u * v * np.where(valid_coord(x2, y2)[..., None], image[x2_valid,
y2_valid], 0))

return new_image.astype(np.uint8)

```

实验结果

插值算法

原图像分辨率为 512×512 ，均放大 4×4 倍

原图



手动实现



从左到右分别为最近邻、双线性 and 双三次。可以观察到最近邻相较于其他两者，边缘有更明显的锯齿，不够平滑。

官方实现



几何变换

原图



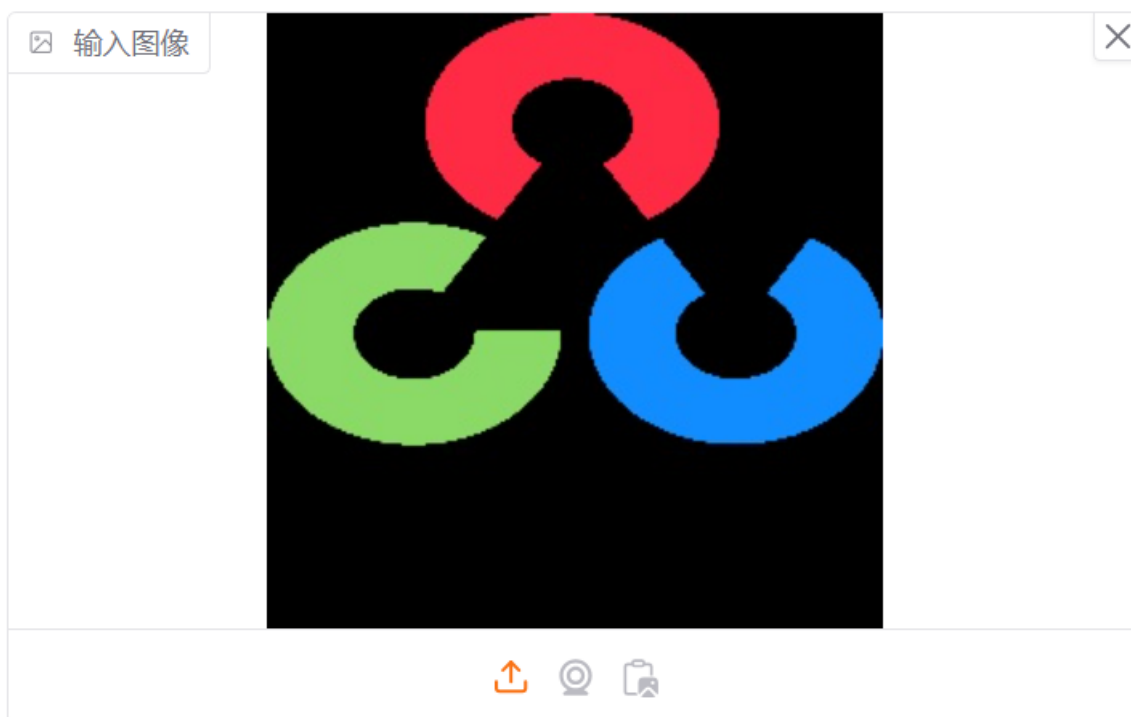
平移、斜切、旋转



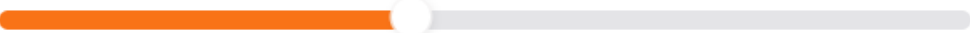
这三张图片是对原图分别进行平移、斜切和绕原点旋转的结果。为了更清晰地表示出图像的范围，在操作前为图片增加了一个白色的边框。

UI界面


界面共分为三列，最左边一列负责输入图像与改变处理参数。



缩放倍数 1.97 ↺

0.5  4

旋转角度 45 ↺

-180  180

垂直平移 100 ↺

-100  100

水平平移 -26 ↺

-100  100

水平错切 0 ↺

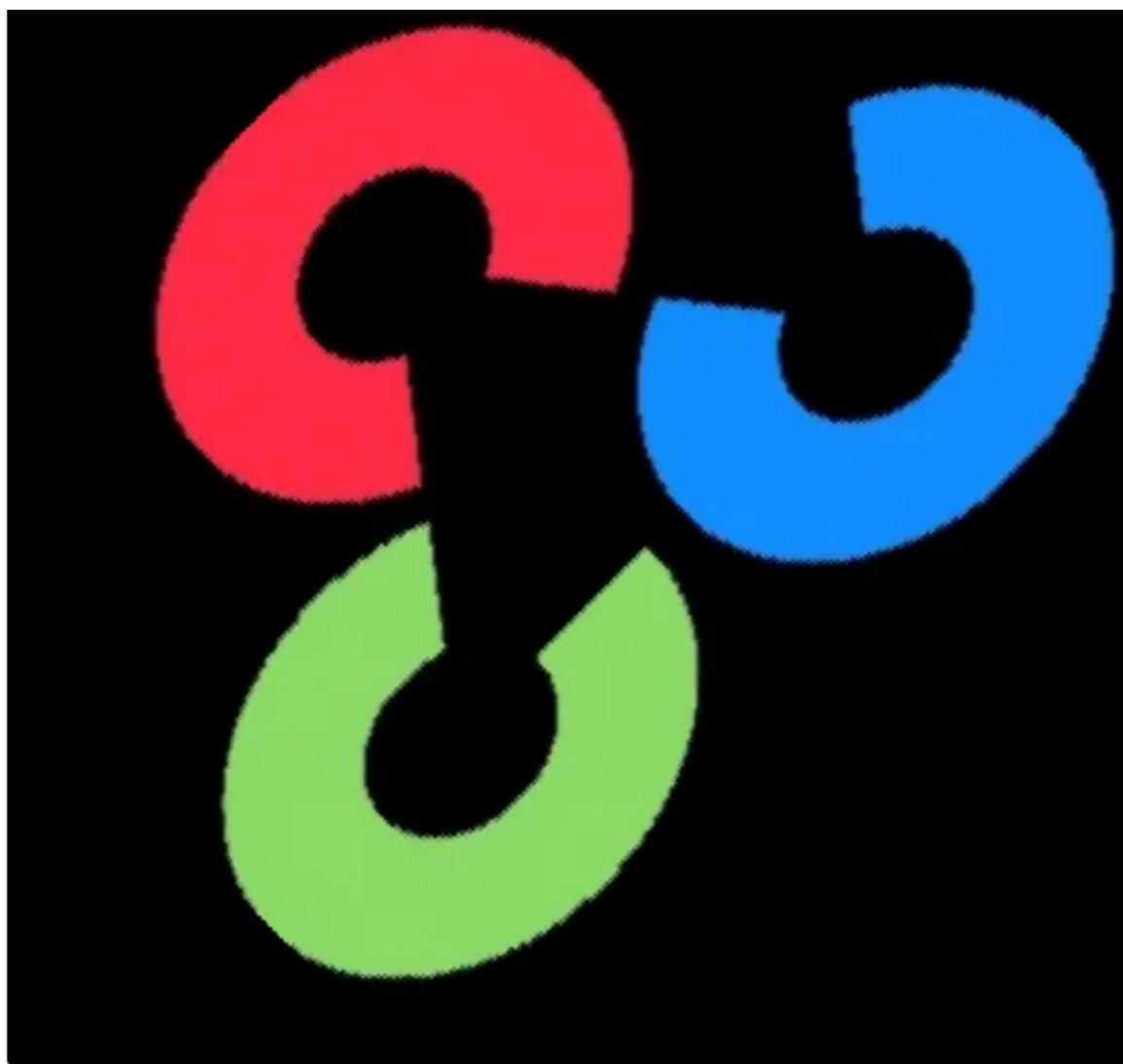
-1  1

垂直错切 0 ↺

-1  1

运行

中间列显示了手动输出的结果，包括三种方法缩放的结果与叠加几何变换后的结果：



手动实现



最近邻插值



双线性插值






最后一列是OpenCV官方实现的三种方法缩放的结果：



整体的UI界面如下：

作业二: 图像缩放工具

输入图像



缩放倍数

0.5

4

1.97

↕

旋转角度

-180

180

45

↕

垂直平移

-100

100

100

↕

水平平移

-100

100

-26

↕

水平错切

-1

1

0

↕

垂直错切

-1


1

0

↕


运行

双线性插值-手动实现




OpenCV/实现


最近邻插值



双线性插值




双三次插值




手动实现

最近邻插值



双线性插值



双三次插值

