School of Data Science, Fudan University

Image Processing HW6

李裕硕

May 29, 2022

1 实现空间变换算法

1.1 仿射变换

code:

```
def my_transform(im_ary, MA, target_m=1000, target_n=1000):
    """
    基于仿射变换的前向图变换
    :param im_ary: 输入图像
    :param MA: 仿射变换的变换矩阵
    :param target_m: 目标图像的行数
    :param target_n: 目标图像的列数
    :return:
    """
    trans_matrix = MA
    # 输入图像的形状
    m, n, q = im_ary.shape
    new_img = np.zeros((target_m, target_n, 3), dtype = 'int32')
    sum_changed = np.zeros((target_m, target_n))
    # 计算坐标（i, j）变换之后的新坐标，并通过前向图变换给新坐标赋值
    for i in tqdm(range(m)):
        for j in range(n):
            # 输入图像的坐标向量
            vector = np.array([i, j, 1]).reshape(3, 1)
            # 变换后得到的目标图像的坐标向量
            newvect = np.dot(trans_matrix, vector)
            new_i = int(np.round(newvect[0]))
            new_j = int(np.round(newvect[1]))
            # 边界判断
            if new_i >= target_m or new_j >= target_n or new_i < 0 or new_j < 0:
                continue
            # 前向图赋值
            if sum_changed[new_i, new_j] == 0:
                sum_changed[new_i, new_j] += 1
                new_img[new_i, new_j, 0] = int(im_ary[i, j, 0])
                new_img[new_i, new_j, 1] = int(im_ary[i, j, 1])
                new_img[new_i, new_j, 2] = int(im_ary[i, j, 2])
    return new_img
```

Figure 1.1: Forward affine transformation

```python
def Shear(alpha=0,beta=0):
    # 错切
    trans_matrix1 = np.eye(3)
    trans_matrix1[1,0] = alpha
    trans_matrix2 = np.eye(3)
    trans_matrix2[0,1] = beta
    return np.dot(trans_matrix1, trans_matrix2)

def Scaling(sx=1,sy=1):
    # 放缩
    trans_matrix = np.eye(3)
    trans_matrix[0,0] = sx
    trans_matrix[1,1] = sy
    return trans_matrix

def Translation(x=0,y=0):
    # 平移
    trans_matrix = np.eye(3)
    trans_matrix[0,2] = x
    trans_matrix[1,2] = y
    return trans_matrix

def Rotation(theta=0):
    # 旋转
    trans_matrix = np.eye(3)
    trans_matrix[0,0] = np.cos(theta)
    trans_matrix[0,1] = np.sin(theta)
    trans_matrix[1,1] = np.cos(theta)
    trans_matrix[1,0] = -np.sin(theta)
    return trans_matrix
```

Figure 1.2: Linear transformations

```python
def Affine(parameters):
    # 仿射
    t1 = Shear(alpha=parameters['shear_a'], beta=parameters['shear_b'])
    t2 = Scaling(sx=parameters['scale_a'],sy=parameters['scale_b'])
    t3 = Translation(x=parameters['trans_a'],y=parameters['trans_b'])
    t4 = Rotation(theta=parameters['theta'])
    trans_matrix = np.dot(t1,np.dot(t2,np.dot(t3,t4)))
    return trans_matrix
```

Figure 1.3: Affine transformation

实现了包括错切、放缩、平移、旋转在内的仿射变换，为此我们使用正向图变换进行一些实验查看这些变换的效果。

放缩：

旋转：

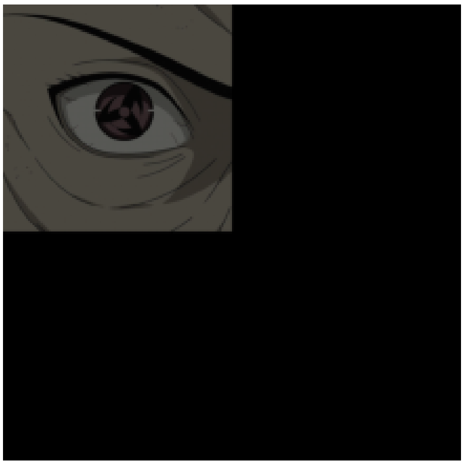

Figure 1.4: Scaling transformation

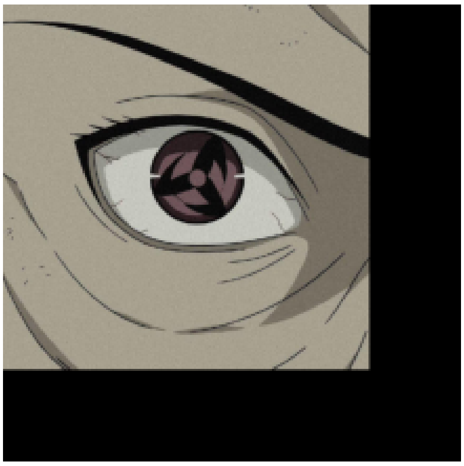

Figure 1.5: Scaling transformation

图形放大的情形中，我们扩大图像展示范围。正向图变换因为无法处理目标图片无像素值点的插值问题，导致有些像素点取值为 0（黑色），因此图片会变暗。而图形缩小的情形中，因为没有插值问题，图像形状和色彩保留完好。

旋转：

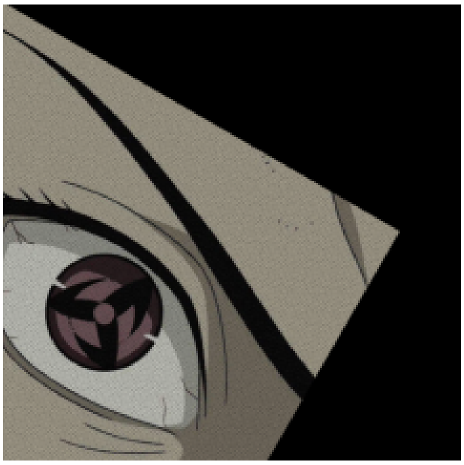

Figure 1.6: Rotation transformation

以原点为中心顺时针旋转 theta 角度。
错切：

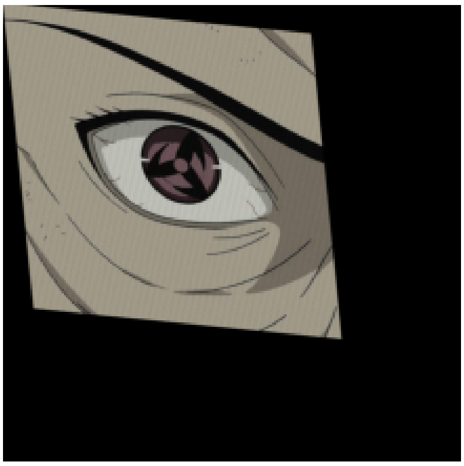

Figure 1.7: Rotation transformation

平移：

在参考图片和浮动图片上选择一一对应的区域，进行局部仿射变换，其他非区域内的点的

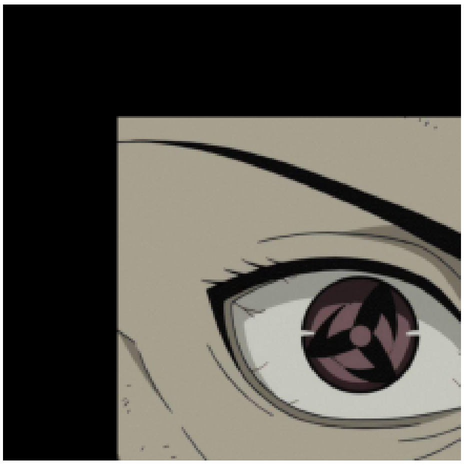

Figure 1.8: Translation transformation

将上述线性空间变换整合就得到仿射变换：

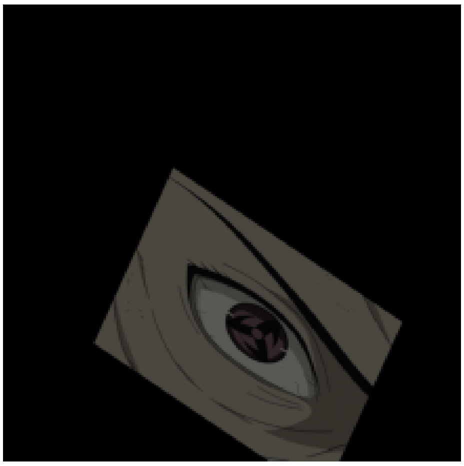

Figure 1.9: Translation transformation

1.2 局部仿射变换

在参考图片和浮动图片上选择一一对应的区域，进行局部仿射变换，其他非区域内的点的变换通过以下公式得到：

$$T(X)=\begin{cases} G_i(X), & X\in U_i,\ i=1\ldots n \\ \sum_{i=1}^{n} w_i(X)G_i(X_i), & X\notin \bigcup_{i=1}^{n} U_i \end{cases} \qquad w_i(X)=\left(1/d_i(X)^e\right)\Big/\left(\sum_{i=1}^{n} 1/d_i(X)^e\right)$$

Figure 1.10: Local Affine transformation

参考图片和浮动图片及其相应的区域选择如下所示：

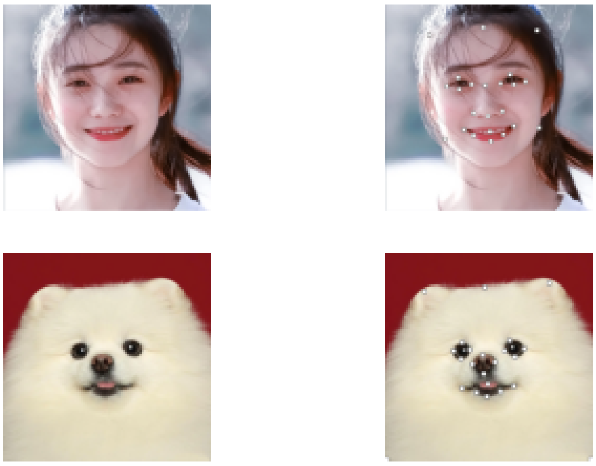

Figure 1.11: Reference and float images

利用前向图变换，从浮动图像变换到参考图片。code：

```python
for i in tqdm(range(target_m)):
    for j in range(target_n):
        liule = False
        # 前向图中的坐标向量
        point = np.array([i,j])
        # G中存储浮动图像选择区域中心点的坐标向量
        for item in G:
            symbol = point == item
            if symbol.sum()==2:
                liule = True
                break
        # 区域中心点直接赋值
        if liule == True:
            continue
        # 计算坐标向量到选择点的距离
        distances = []
        for k in range(23):
            distances.append(np.linalg.norm(point-G[k])**2)
        distances = np.array(distances)
        indexs = np.argsort(distances) # 根据距离大小顺序获得下标
        # 根据局部仿射变换公式计算新的坐标向量
        w_G = []
        new_distances = []
        for iter in range(4): #取前四个点
            index = indexs[iter]
            w_G.append(1/distances[index]*G[index])
            new_distances.append(distances[index])
        sums = np.sum(1/np.array(new_distances))
        w_G = np.array(w_G)/sums
        # 新的坐标向量
        newvector = np.round(np.sum(w_G,axis=0))
        new_i = int(np.round(newvector[0]))
        new_j = int(np.round(newvector[1]))
        if new_i >= target_m or new_j >= target_n or new_i < 0 or new_j < 0:
            continue
        #前向图给新的坐标向量赋予像素值
        new_image[new_i,new_j] = im_ary2[i,j]
```

Figure 1.12: Forward local affine transformation

结果如下：

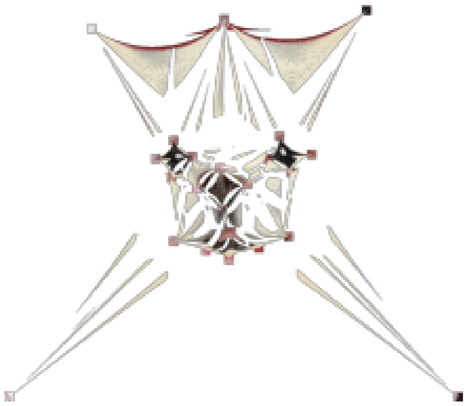

Figure 1.13: Forward local affine transformation result

2 反向图像变化

code：

```python
def get_coord(shape):
    """
    获得浮动图片的所有坐标
    :param shape: 图片形状
    :return:
    """
    # 只考虑二维图片
    m,n = shape
    seq_x = np.arange(m,dtype='int32')
    seq_y = np.arange(n,dtype='int32')
    x, y = np.meshgrid(seq_x,seq_y)
    # 坐标矩阵
    coord = np.stack((np.transpose(x),np.transpose(y)))
    return coord
```

Figure 2.1: Get coordinates

```python
def linear_interpolation(new_coord, float_img):
    """
    线性插值拟合（从浮动图像到参考图像）
    :param new_coord: 坐标矩阵
    :param float_img: 浮动图像
    :return: 参考图像
    """
    coord_floor = np.floor(new_coord).astype(int)
    q_coord = new_coord - coord_floor
    # 可以广播到三通道图像
    get_img = lambda a,b: float_img[:,np.clip(a, 0, float_img.shape[1] - 1),np.clip(b, 0, float_img.shape[2] - 1)]
    # 双线性插值
    a = (1-q_coord[0]) * (1-q_coord[1]) * get_img(coord_floor[0], coord_floor[1])
    b = (1-q_coord[0]) * q_coord[1] * get_img(coord_floor[0], coord_floor[1]+1)
    c = q_coord[0] * (1-q_coord[1]) * get_img(coord_floor[0]+1, coord_floor[1])
    d = q_coord[0] * q_coord[1] * get_img(coord_floor[0]+1, coord_floor[1]+1)
    return a+b+c+d
```

Figure 2.2: Linear interpolation

```python
def Trans_coord(M, coord):
    shape = coord.shape
    MA = M[:2,:2]
    Ts = M[:2,2].reshape((2,1))
    new_coord = np.dot(MA, coord.reshape(shape[0],-1)) + Ts
    return new_coord.reshape(shape)
```

Figure 2.3: Affine transformation adjustment

实验：

```python
parameters = {}
parameters['theta'] = np.pi/6
parameters['scale_a'], parameters['scale_b'] = 0.8, 0.8
parameters['shear_a'], parameters['shear_b'] = 0.2, 0.2
parameters['trans_a'], parameters['trans_b'] = 300, 300
t4 = Affine(parameters)
MA4 = t4[:2,:]
new_img = my_transform(im_ary,MA4, target_m=1000, target_n=1000)
plt.imshow(new_img)
plt.axis("off")
plt.savefig('fig10.png')
```

Figure 2.4: Experiment

```python
t5 = np.linalg.inv(t4)
MA5 = t5[:2,:]
img_coord = get_coord(im_ary.shape[:2])
new_coord = Trans_coord(MA5, img_coord)
deformed_img = linear_interpolation(new_coord, new_ary)
t_im = np.transpose(deformed_img,(1,2,0)).astype('int32')
plt.imshow(t_im)
plt.axis("off")
plt.savefig('fig11.png')
```

Figure 2.5: Experiment

通过初始化仿射变换的参数，我们首先可以通过正向图像变换得到变换后的参考图像：

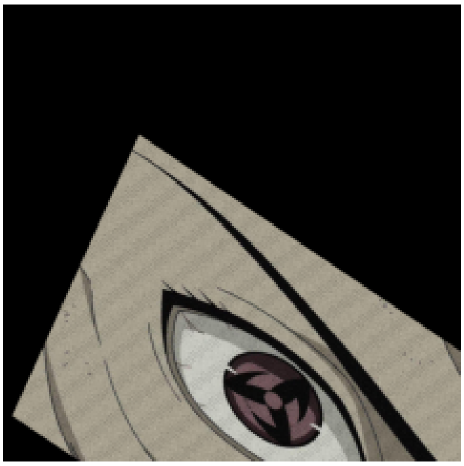

Figure 2.6: Forward affine transformation result

可以发现变换效果并不好，存在条带状阴影。由于仿射变换是线性变换，我们通过求仿射矩阵的逆矩阵，就可以应用于反向图像变换，得到相似的变换后图像，结果如下：

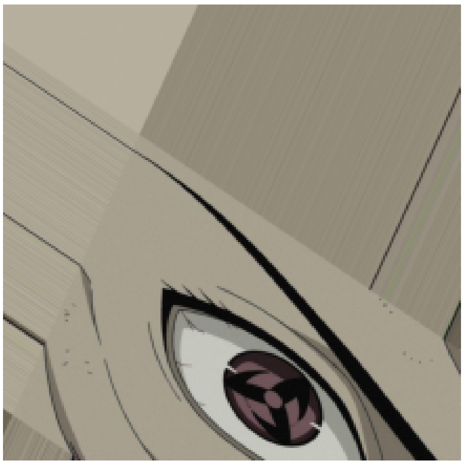

Figure 2.7: Backward affine transformation result

可以发现与前向方法得到的结果图片相比，反向方法完美保留了原浮动图像的特征，并且在没有像素点的位置，也运用了插值算法对原浮动图像的边界进行插值，得到相应像素值。

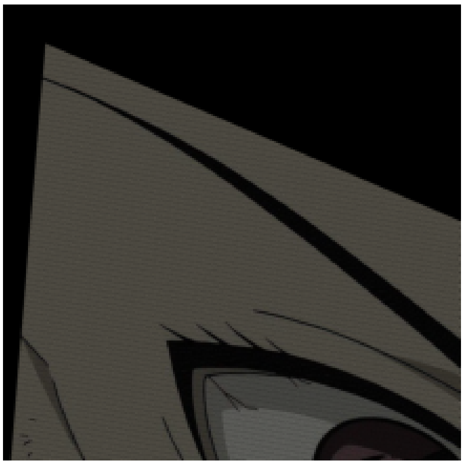

Figure 2.8: Forward affine transformation result

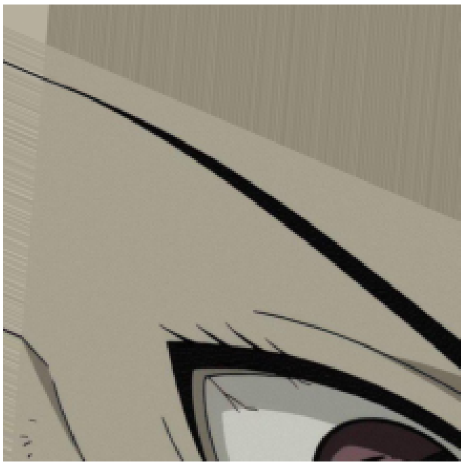

Figure 2.9: Backward affine transformation result

进行一次新的参数初始化，同样能发现反向图像变换的良好效果，结果如下：