计算机视觉与模式识别
作业一 报告

密码	LFJCVPR3120305208
姓名	罗福杰
班级	硕0078班
学号	3120305208
Email	1626027173@qq.com
日期	2020-12-9

目录

1 图像变换

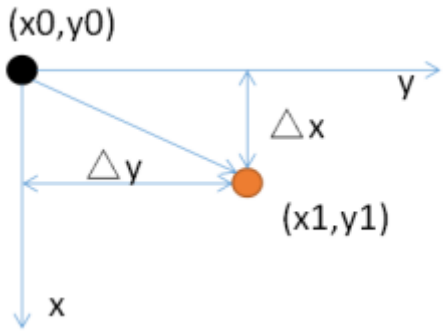
1.1 图像的参数化几何变换原理

图像几何变换又称为图像空间变换，它将一副图像中的坐标位置映射到另一幅图像中的新坐标位置。通过图像的几何变换就是确定这种空间映射关系，以及映射过程中的变化参数。下面介绍图像的平移、镜像、缩放和旋转变换的原理。

1.1.1 图像平移变换

图像的平移变换就是将图像所有的像素坐标分别加上指定的水平偏移量和垂直偏移量。平移变换根据是否改变图像大小分为两种，直接丢弃或者通过加目标图像尺寸的方法使图像能够包含这些点。

假设原来的像素的位置坐标为（x0,y0），经过平移量（Δx， Δy）后，坐标变为（x1,y1），如下所示：



用数学式子表示可以表示为：

$$x1 = x0 + \Delta x;$$
$$y1 = y0 + \Delta y;$$

用矩阵表示为：

$$\begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix}$$

为了适应像素、拓展适应性，这里使用了三维的向量进行表示，其中：

$\begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$称为平移变换矩阵（因子），$\Delta x$ 和 Δy 为平移量。

1.1.2 图像的镜像变换

图像的镜像变换分为两种：水平镜像和垂直镜像。水平镜像以图像垂直中线为轴，将图像的像素进行对换，也就是将图像的左半部和右半部对调。垂直镜像则是以图像的水平中线为轴，将图像的上半部分和下班部分对调。

设一副图像的像素为（height*width），则变换后的图像的坐标位置为：（x1,y1）。则，水平镜像的原理：

$$\begin{cases} x1 = width - x0 - 1 \\ y1 = y0 \end{cases}$$

其逆变换的原理是：

$$\begin{cases} x0 = width - x1 - 1 \\ y0 = y1 \end{cases}$$

垂直镜像的原理是：

$$\begin{cases} x1 = x0 \\ y1 = height - y0 - 1 \end{cases}$$

其逆变换是：

$$\begin{cases} x0 = x1 \\ y0 = height - y1 - 1 \end{cases}$$

1.1.3 图像缩放

图像的缩放是将图像的尺寸变小或变大的过程，也就是减少或增加原图像数据的像素个数。简单来说，就是通过增加或删除像素点来改变图像的尺寸。当图像缩小时，图像会变得更加清晰，当图像放大时，图像的质量会有所下降，因此需要进行插值处理。

缩放的原理是：设水平缩放系数为 sx，垂直缩放系数为 sy，(x0,y0)为缩放前坐标，(x1,y1)为缩放后坐标，其缩放的坐标映射关系：

$$\begin{cases} x1 = x0 \times sx \\ y1 = y0 \times sy \end{cases}$$

矩阵表示的形式为：

$$\begin{bmatrix} x1 & y1 & 1 \end{bmatrix} = \begin{bmatrix} x0 & y0 & 1 \end{bmatrix} \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

这是向前映射，在缩放的过程改变了图像的大小，使用向前映射会出现映射重叠和映射不完全的问题，所以这里更关心的是向后映射，也就是输出图像通过向后映射关系找到其在原图像中对应的像素。

$$\begin{cases} x0 = \dfrac{x1}{sx} \\ y0 = \dfrac{y1}{sy} \end{cases}$$

1.1.4 图像旋转

图像的旋转就是让图像按照某一点旋转指定的角度。图像旋转后不会变形，但是其“垂直对称抽”和“水平对称轴”都会发生改变，旋转后图像的坐标和原图像坐标之间的关系已不能通过简单的加减乘法得到，而需要通过一系列的复杂运算。而且图像在旋转后其宽度和高度都会发生变化，其坐标原点会发生变化。

图像所用的坐标系不是常用的笛卡尔，其左上角是其坐标原点，X 轴沿着水平方向向右，Y 轴沿着竖直方向向下。而在旋转的过程一般使用旋转中心为坐标原点的笛卡尔坐标系，所以图像旋转的第一步就是坐标系的变换。设旋转中心为(x0,y0)，(x′,y′)是旋转后的坐标，(x,y)是旋转后的坐标，则坐标变换如下：

$$\begin{cases} x' = x - x0 \\ y' = -y + y0 \end{cases}$$

矩阵表示为：

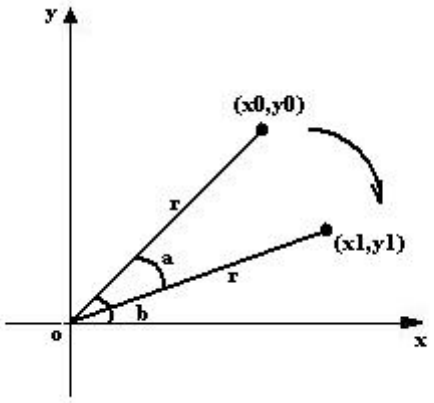
$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ -x0 & y0 & 1 \end{bmatrix}$$

在最终的实现中，常用到的是有缩放后的图像通过映射关系找到其坐标在原

图像中的相应位置，这就需要上述映射的逆变换：

$$\begin{bmatrix} x & y & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ x0 & y0 & 1 \end{bmatrix}$$

坐标系变换到以旋转中心为原点后，接下来就要对图像的坐标进行变换。



上图所示，将坐标(x0,y0)顺时针方向旋转 a,得到(x1,y1)。

旋转前的表示：

$$\begin{cases} x0 = r\cos b \\ y0 = r\sin b \end{cases}$$

旋转后有：

$$\begin{cases} x1 = r\cos(b-a) = r\cos b\cos a + r\sin b\sin a = x0\cos a + y0\sin a \\ y1 = r\sin(b-a) = r\sin b\cos a - r\cos b\sin a = -x0\sin a + y\cos a \end{cases}$$

矩阵的形式有：

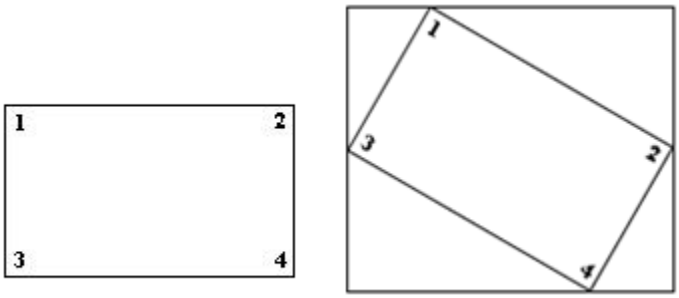
$$\begin{bmatrix} x1 & y1 & 1 \end{bmatrix} = \begin{bmatrix} x0 & y0 & 1 \end{bmatrix} \begin{bmatrix} \cos a & -\sin a & 0 \\ \sin a & \cos a & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

其逆变换是：

$$\begin{bmatrix} x0 & y0 & 1 \end{bmatrix} = \begin{bmatrix} x1 & y1 & 1 \end{bmatrix} \begin{bmatrix} \cos a & \sin a & 0 \\ \text{-}\sin a & \cos a & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

由于在旋转的时候是以旋转中心为坐标原点的，旋转结束后还需要将坐标原点移到图像左上角，也就是还要进行一次变换。这里需要注意的是，旋转中心的坐标(x0,y0)是在以原图像的左上角为坐标原点的坐标系中得到，而在旋转后由于

图像的宽和图像的高度发生了变化，也就导致了旋转后图像的坐标原点和旋转前的发生了变换。



上边两图，可以清晰的看到，旋转前后图像的左上角，也就是坐标原点发生了变换。

在求图像旋转后左上角的坐标前，先来看看旋转后图像的宽和高。从上图可以看出，旋转后图像的宽和高与原图像的四个角旋转后的位置有关。

设 top 为旋转后最高点的纵坐标，down 为旋转后最低点的纵坐标，left 为旋转后最左边点的横坐标，right 为旋转后最右边点的横坐标。旋转后的宽和高为 newWidth,newHeight，则可得到下面的关系：

$$\begin{cases} newWidth = right - left \\ newHeight = top - down \end{cases}$$

也就很容易的得出旋转后图像左上角坐标(left,top)（以旋转中心为原点的坐标系）；故在旋转完成后要将坐标系转换为以图像的左上角为坐标原点，可由下面变换关系得到：

$$\begin{cases} x' = x + left \\ y' = -y + top \\ (x', y')\text{为变换后的坐标；} \\ (x, y)\text{为变换前的坐标.} \end{cases}$$

矩阵表示：

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ left & top & 1 \end{bmatrix}$$

其逆变换为：

$$\begin{bmatrix} x & y & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ -left & top & 1 \end{bmatrix}$$

综合以上，也就是说原图像的像素坐标要经过三次的坐标变换:

(1). 将坐标原点由图像的左上角变换到旋转中心

(2). 以旋转中心为原点，图像旋转角度 a

(3). 旋转结束后，将坐标原点变换到**旋转后图像的左上角。**

可以得到下面的旋转公式：

$$\begin{bmatrix} x & y & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ -x_0 & y_0 & 1 \end{bmatrix} \begin{bmatrix} \cos a & -\sin a & 0 \\ \sin a & \cos a & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ left & top & 1 \end{bmatrix}$$

注：(x’,y’)旋转后的坐标，(x,y)原坐标，(x0,y0)旋转中心,a 旋转的角度(顺时针);这种由输入图像通过映射得到输出图像的坐标，是向前映射。常用的向后映射是其逆运算：

$$\begin{bmatrix} x & y & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ -left & top & 1 \end{bmatrix} \begin{bmatrix} \cos a & \sin a & 0 \\ -\sin a & \cos a & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ x_0 & y_0 & 1 \end{bmatrix}$$

1.2 图像的向前变换与图像的逆向变换

图像的几何变换改变了像素的空间位置，建立一种原图像像素与变换后图像像素之间的映射关系，通过这种映射关系能够实现下面两种计算：

- 原图像任意像素计算该像素在变换后图像的坐标位置；
- 变换后图像的任意像素在原图像的坐标位置；

对于第一种计算，只要给出原图像上的任意像素坐标，都能通过对应的映射关系获得到该像素在变换后图像的坐标位置。将这种输入图像坐标映射到输出的过程称为**“向前变换”**或**“向前映射”（forward warping）**。

反过来，知道任意变换后图像上的像素坐标，计算其在原图像的像素坐标，将输出图像映射到输入的过程称为**“向后变换”**或**“向后映射”（inverse warping）**。

在上一节的原理推导中，已经给出了常用图像变换操作（平移、镜像、缩放、旋转）的向前变换和向后变换的表达式。

1.3 图像的下采样原理与图像的内插方法原理

1.3.1 图像的上采样和下采样

- **下采样（subsampled）**（或称为缩小图像或降采样（downsampled））的主要目的有两个：1、使得图像符合显示区域的大小；2、生成对应图像的缩略图。

  其原理是：对于一幅图像 I 尺寸为 M*N，对其进行 s 倍下采样，即得到(M/s)*(N/s)尺寸的得分辨率图像，当然 s 应该是 M 和 N 的公约数才行，如果考虑的是矩阵形式的图像，就是把原始图像 s*s 窗口内的图像变成一个像素，这个像素点的值就是窗口内所有像素的均值：

$$P_k = \sum_{i \in win(k)} \frac{I_i}{s^2}$$

- **上采样（upsampling）**（或称为放大图像或图像插值（interpolating））的主要目的是放大原图像，从而可以显示在更高分辨率的显示设备上。

  其原理是：图像放大几乎都是采用内插值方法，即在原有图像像素的基础上在像素点之间采用合适的插值算法插入新的元素。

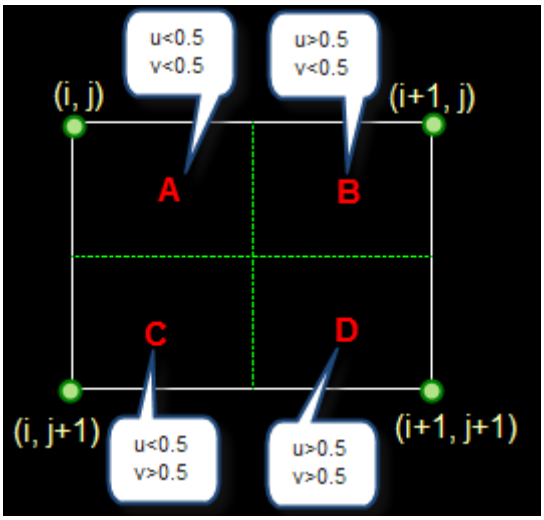
无论缩放图像（下采样）还是放大图像（上采样），采样方式有很多种。如最近邻插值，双线性插值，均值插值，中值插值等方法。

下面主要介绍近邻插值和双线性插值的原理。

1.3.2 近邻插值的原理

最近邻插值是取采样点周围四个相邻像素点中距离最近的**“一个邻点的灰度值”**作为该点灰度值的方法。

在待求象素的四邻象素中，将距离待求象素最近的邻象素灰度值赋给待求的象素。设 i+u, j+v(i, j 为正整数， u, v 为大于零小于 1 的小数，下同)为待求象素坐标，则待求象素灰度的值 f(i+u, j+v) 如下图所示：

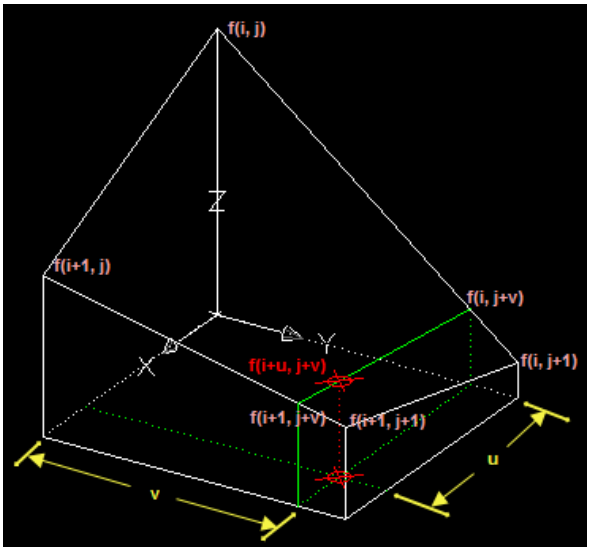


如果(i+u, j+v)落在 A 区，即 u<0.5, v<0.5，则将左上角象素的灰度值赋给待求象素，同理，落在 B 区则赋予右上角的象素灰度值，落在 C 区则赋予左下角象素的灰度值，落在 D 区则赋予右下角象素的灰度值。

最邻近元法计算量较小，但可能会造成插值生成的图像灰度上的不连续，在灰度变化的地方可能出现明显的锯齿状。

1.3.3 双线性插值的原理

双线性内插法是利用待求象素四个邻象素的灰度在两个方向上作线性内插，如下图所示：



对于 (i, j+v)，f(i, j) 到 f(i, j+1) 的灰度变化为线性关系，则有：

$$f(i, j+v) = [f(i, j+1) - f(i, j)] * v + f(i, j)$$

同理对于 (i+1, j+v) 则有：

$$f(i+1, j+v) = [f(i+1, j+1) - f(i+1, j)] * v + f(i+1, j)$$

从 f(i, j+v) 到 f(i+1, j+v) 的灰度变化也为线性关系，由此可推导出待求象素灰度的计算式如下：

$$f(i+u, j+v) = (1-u) * (1-v) * f(i, j) + (1-u) * v * f(i, j+1) + u * (1-v) * f(i+1, j) + u * v * f(i+1, j+1)$$

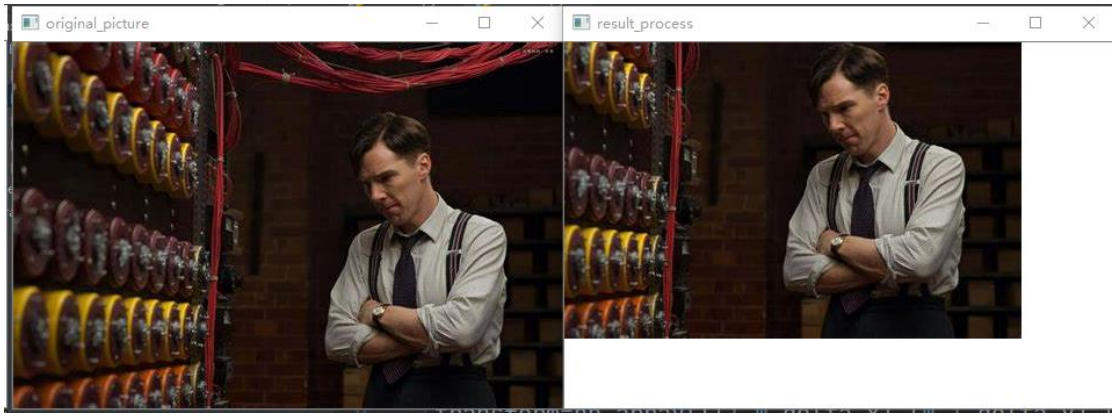
双线性内插法的计算比最邻近点法复杂，计算量较大，但没有灰度不连续的缺点，结果基本令人满意。它具有低通滤波性质，使高频分量受损，图像轮廓可能会有一点模糊。

1.4 图像的几何变换实验结果

1.4.1 图像的平移变换

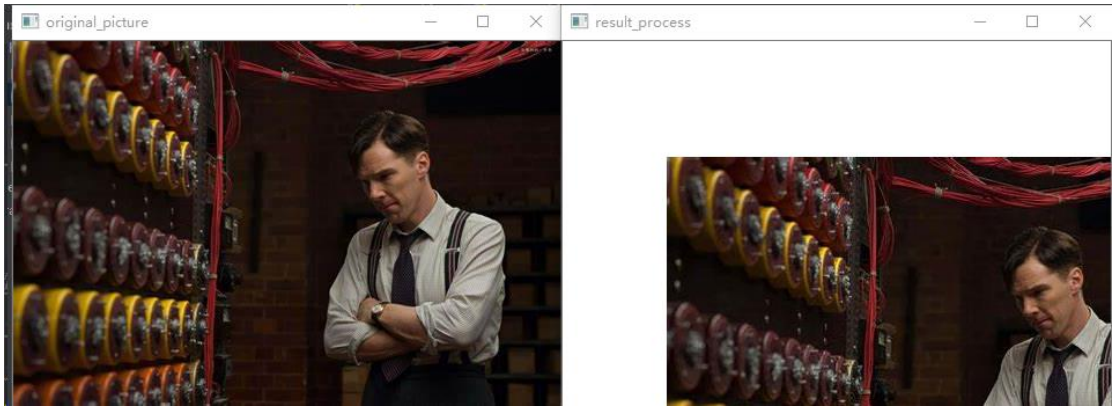
通过 1.1.1 的图像平移的原理可知：由 $\begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix}$ 构建平移矩阵；并实现图像平移的结果如下：

当平移量为（-80,-60）时，图像的平移结果如下：



由结果可见，图像发生了相应的位置平移；

当平移量为（100，90）时，图像的平移结果如下：



由结果可见，图像发生了相应的位置平移；

1.4.2 图像的旋转变换

由 1.1.4 可知，图像的旋转变换原理是：

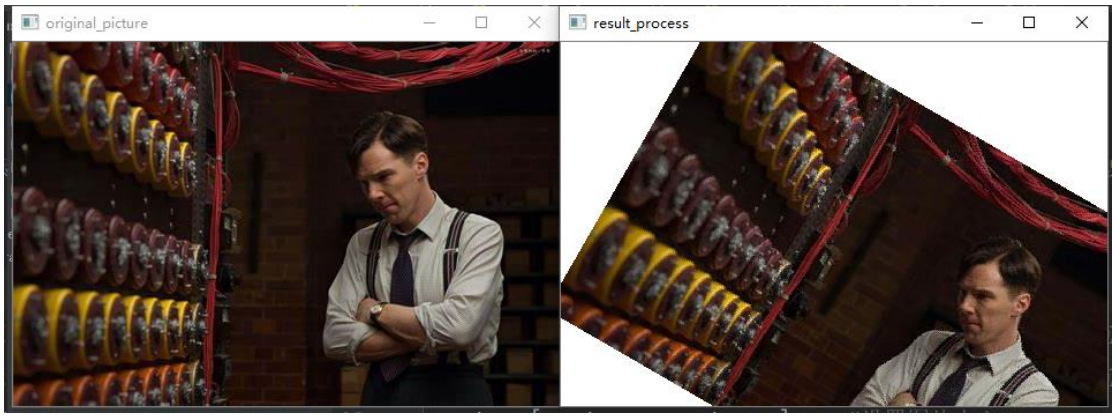
$$\begin{bmatrix} x1 & y1 & 1 \end{bmatrix} = \begin{bmatrix} x0 & y0 & 1 \end{bmatrix} \begin{bmatrix} \cos a & -\sin a & 0 \\ \sin a & \cos a & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

所以，在实验过程中，构建旋转矩阵，即可实现旋转变换：

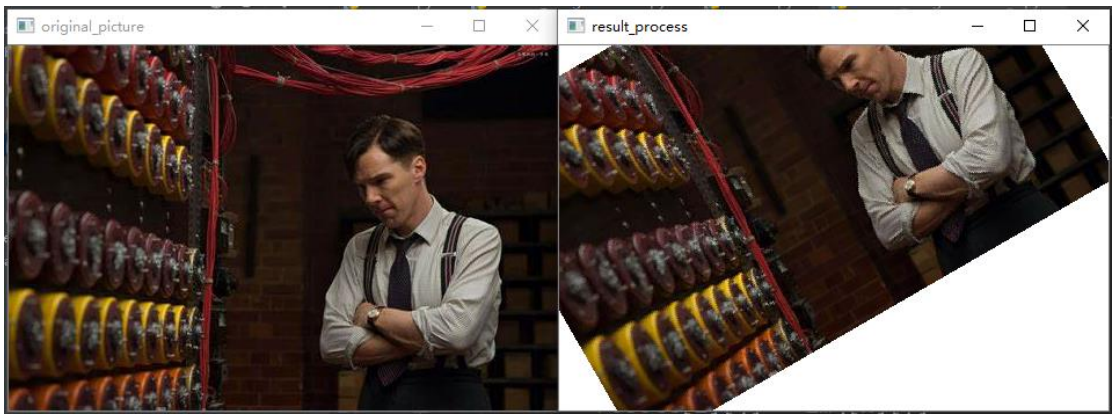
参数输入：

```
图像的高度： 315 图像的宽度： 474
请输入旋转中心x：（请在图像的宽度范围内输入）：250
请输入旋转中心y：（请在图像的宽度范围内输入）：150
请输入旋转角度：30
```

结果如下：




```
图像的高度： 315 图像的宽度： 474
请输入旋转中心x：（请在图像的宽度范围内输入）：200
请输入旋转中心y：（请在图像的宽度范围内输入）：100
请输入旋转角度：-30
```



结果分析：由以上的结果可以，图像按照了变换要求发生了旋转变换。在变换结果中，只取了和原图像像素大小的区域，所以结果中，有部分旋转到区域以外的原图像信息没有呈现出来。

1.4.3 图像的欧式变换

图像的欧式变换也称为等距变换，包括对图像的平移变换和旋转变换；其变换矩阵是：

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \varepsilon\cos(\theta) & -\varepsilon\sin(\theta) & t_x \\ \varepsilon\sin(\theta) & -\varepsilon\cos(\theta) & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

变换矩阵可以简化为：

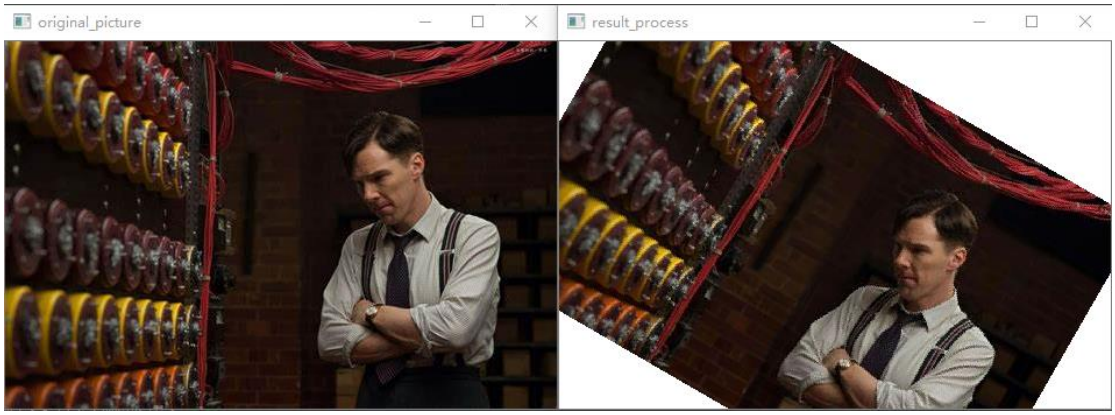
$$x' = H_E x = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} x$$

即是一个旋转矩阵和平移矩阵组成的一个新的变换矩阵，其中 ε =1，为旋转的方向因子。编程实现，其结果如下：

当输入的变换矩阵参数如下时：

```
图像的高度： 315 图像的宽度： 474
请输入x方向偏移量：20
请输入y方向偏移量：20
请输入旋转中心x：（请在图像的宽度范围内输入）：150
请输入旋转中心y：（请在图像的宽度范围内输入）：150
请输入旋转角度：30
```

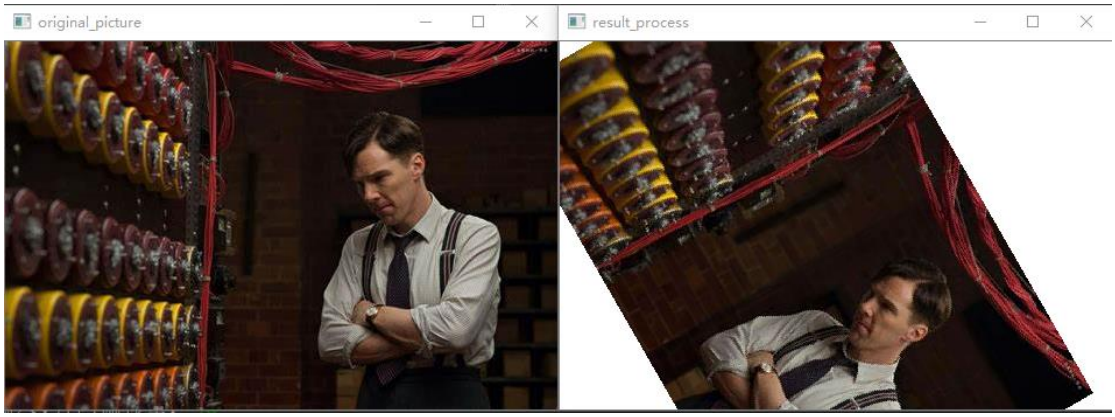
变换结果如下：



当输入的变换矩阵参数如下时：

```
图像的高度： 315 图像的宽度： 474
请输入x方向偏移量：40
请输入y方向偏移量：40
请输入旋转中心x：（请在图像的宽度范围内输入）：150
请输入旋转中心y：（请在图像的宽度范围内输入）：150
请输入旋转角度：60
```

变换结果如下：



结果分析：通过以上两组欧式变换结果可知，图像在输入的变换参数中发生了相应的变换。

1.4.4 图像的相似变换

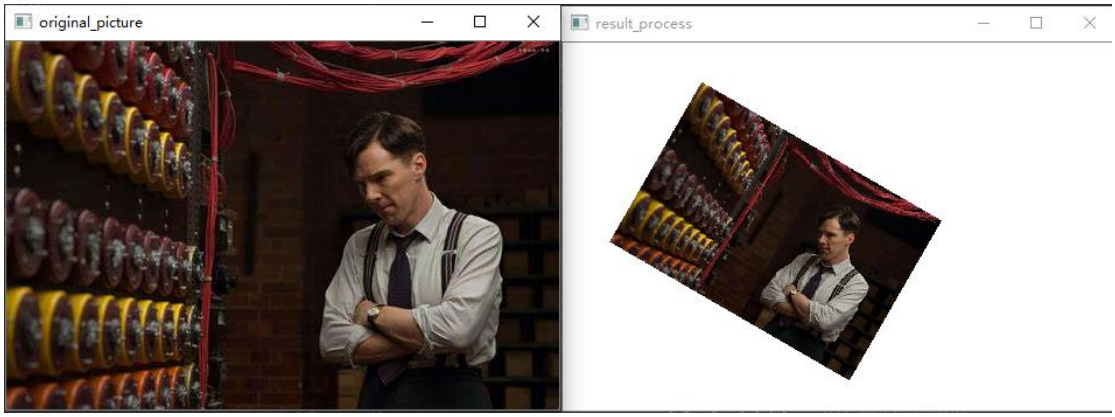
图像的相似变换是在等距变换的基础上增加缩放的因子，其变换矩阵相似矩阵的自由度比等距变换多了一个缩放因子 s，具体表示如下：

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s\cos(\theta) & -s\sin(\theta) & t_x \\ s\sin(\theta) & -s\cos(\theta) & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

当输入的变换矩阵参数如下时：

```
图像的高度： 315 图像的宽度： 474
请输入x方向偏移量：20
请输入y方向偏移量：20
请输入旋转中心x：（请在图像的宽度范围内输入）：150
请输入旋转中心y：（请在图像的宽度范围内输入）：150
请输入旋转角度：30
请输入相似比例：0.5
```

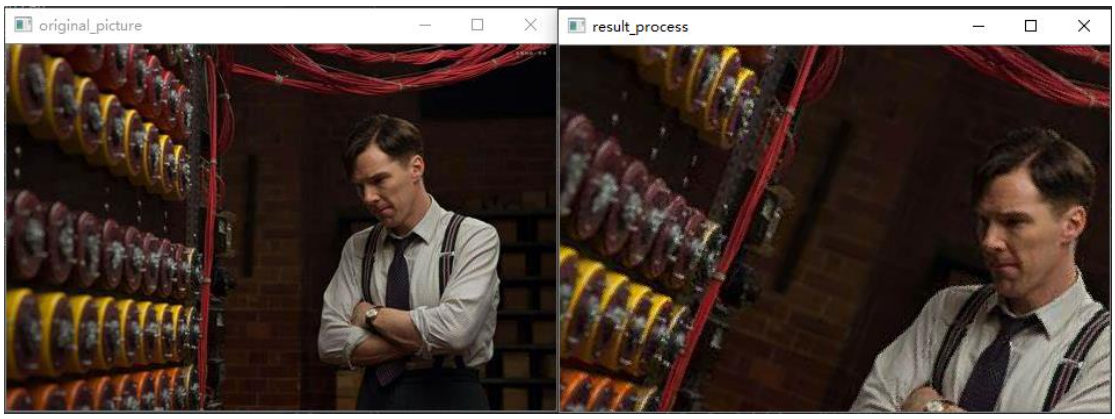
变换结果如下：



当输入的变换矩阵参数如下时：

```
图像的高度： 315 图像的宽度： 474
请输入x方向偏移量：20
请输入y方向偏移量：20
请输入旋转中心x：（请在图像的宽度范围内输入）：150
请输入旋转中心y：（请在图像的宽度范围内输入）：150
请输入旋转角度：20
请输入相似比例：1.5
```

变换结果如下：



结果分析：图像的相似变换就是在欧式变换的基础上增加了缩放的因子，在以上的结果可见，设置该缩放因子即可实现相似变换。

1.4.5 图像的仿射变换

图像的仿射变换：Affine transformation，仿射变换，包括平移变换、旋转变换和各向不同性缩放变换（Non-inotropic Scaling）。同性缩放变换只有一个缩放因子 s，而不同性缩放变换有两个缩放因子 s_x 和 s_y。下面先介绍不同性缩放变换和剪切变换。

不同性缩放变换矩阵：

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

其中，s_x 控制沿 x 轴，s_y 控制着沿 y 轴倾斜，特别地：

- 当 $s_x = s_y$ 时，为各向同性缩放变换。
- 当 s_x =-1，s_y =1 时，为垂直翻转。
- 当 s_x =1，s_y =-1 时，为水平翻转。
- 当 s_x =-1，s_y =-1 时，为水平垂直翻转。

剪切变换（Shear Transformation）：剪切变换矩阵如下表达：

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & a_x & 0 \\ a_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

剪切变换由两个变量 a_x 和 a_y。其中，a_x 控制沿 x 轴倾斜，a_y 控制沿 y 轴倾斜。当 $a_x = -a_y = \tan(\theta)$ 时，剪切变换退化成旋转变换。

仿射变换（Affine Transformation）由平移、剪切变换和非同性缩放变换复合而成。仿射变换矩阵：

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & a_x & t_x \\ a_y & s_y & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

由以上仿射变换矩阵可知，由六个变量：s_x、s_y、a_x、a_y、t_x、t_y，即自由度为

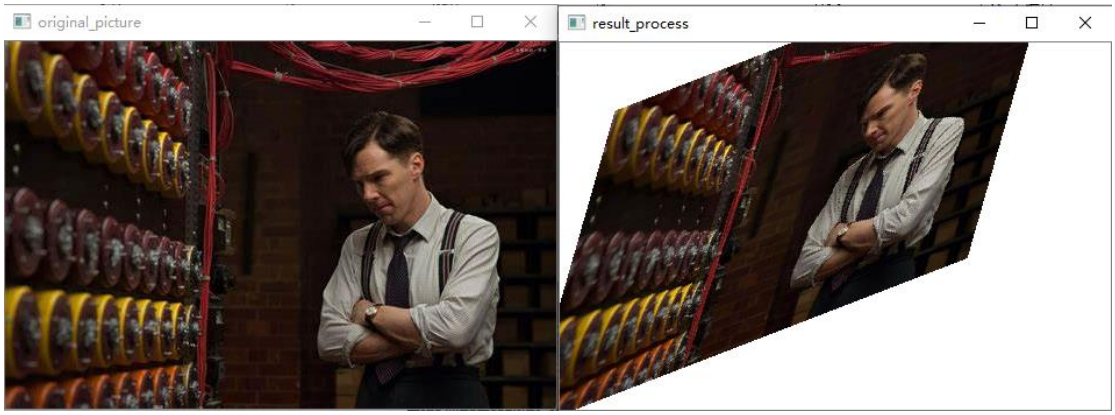
6，其中s_x、s_y控制着缩放翻转，a_x、a_y控制旋转、倾斜，t_x、t_y控制图像的平移。可以把s_x、s_y、a_x、a_y组成一个仿射矩阵A。

由以上的原理，编程实现，通过输入参数：

当输入的变换矩阵参数如下时：

```
图像的高度： 315 图像的宽度： 474
请输入仿射矩阵s_x：1.3
请输入仿射矩阵a_x：0.5
请输入仿射矩阵a_y：0.4
请输入仿射矩阵s_y：1.4
请输入x方向偏移量t_x：-100
请输入y方向偏移量t_y：-90
```

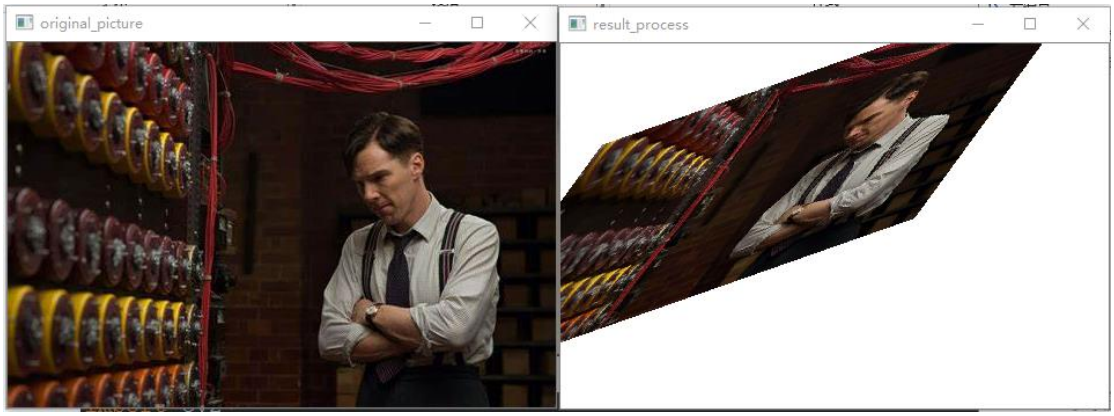
变换结果如下：



当输入的变换矩阵参数如下时：

```
图像的高度： 315 图像的宽度： 474
请输入仿射矩阵s_x：2
请输入仿射矩阵a_x：0.7
请输入仿射矩阵a_y：1.1
请输入仿射矩阵s_y：1.5
请输入x方向偏移量t_x：-200
请输入y方向偏移量t_y：-150
```

变换结果如下：



结果分析：仿射变换是在原来图像平移、旋转变换的基础上，再加上各向不同性缩放变换得到的，将这些变换因子组成仿射变换的变换矩阵，编程时直接输入相应的变换因子即可实现图像的仿射变换。

1.4.5 图像的射影变换

在前面所介绍的图像变换（欧式变换、相似变换、仿射变换）中，都是在 2 维平面上的变换，他们的变换矩阵第三行恒为[0,0,1]。下面介绍投影变换：

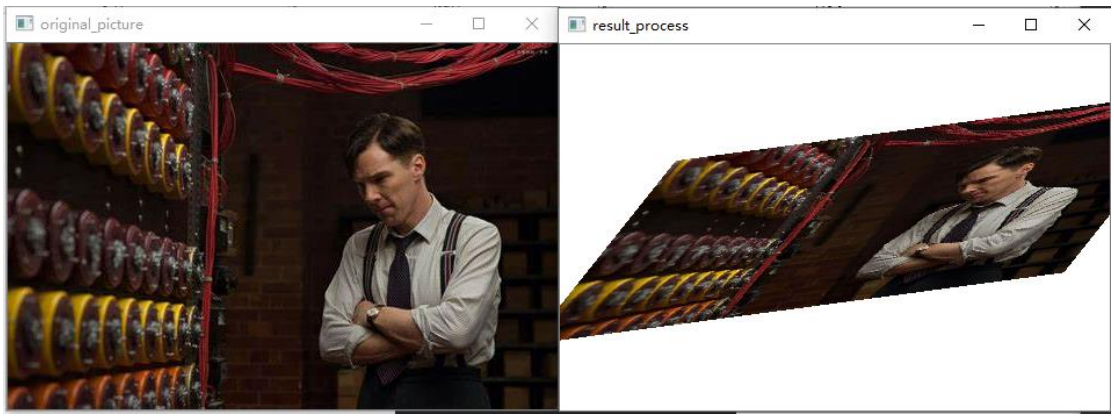
射影变换（projection transform）：当图像中的点的齐次坐标的一般非奇异线性变换。有些文献中把射影变换矩阵称为单应性矩阵变换矩阵如下所示：

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

当输入的变换矩阵参数如下时：

```
图像的高度： 315 图像的宽度： 474
请输入射影矩阵h11：0.7
请输入射影矩阵h12：-0.1
请输入射影矩阵h13：100
请输入射影矩阵h21：-0.2
请输入射影矩阵h22：1
请输入射影矩阵h23：100
请输入射影矩阵h31：0.002
请输入射影矩阵h32：0.0003
```

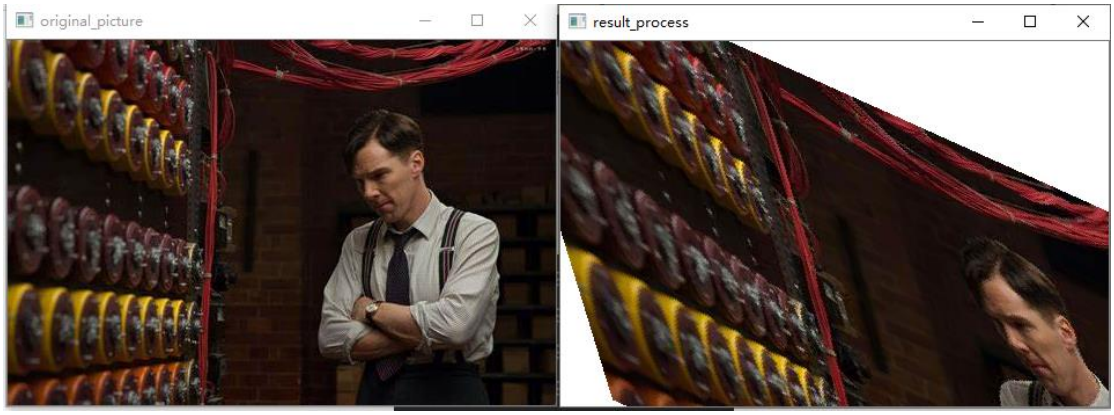
变换结果如下：



当输入的变换矩阵参数如下时：

```
图像的高度： 315 图像的宽度： 474
请输入射影矩阵h11：0.8
请输入射影矩阵h12：-0.2
请输入射影矩阵h13：-100
请输入射影矩阵h21：-0.
请输入射影矩阵h22：0.7
请输入射影矩阵h23：-80
请输入射影矩阵h31：0.005
请输入射影矩阵h32：0.008
```

变换结果如下：



结果分析：可以看出，射影变换就是对图像的旋转+平移+缩放+切变+射影，相比前三种变换图像的形变更为自由，原图中的平行线经过变换之后已经不在平行，而可能相交于一点，射影变换就是把理想点（平行直线在无穷远处相交）变换到图像上。

1.5 图像的高斯金字塔与拉普拉斯金字塔

1.5.1 实验要求

完成图像的高斯金字塔表示与拉普拉斯金字塔表示，讨论前置低通滤波与抽样频率的关系

1.5.2 基本概念

图像金字塔是图像中多尺度表达的一种，最主要用于图像的分割，是一种以多分辨率来解释图像的有效但概念简单的结构。

图像金字塔最初用于机器视觉和图像压缩，一幅图像的金字塔是一系列以金字塔形状排列的分辨率逐步降低，且来源于同一张原始图的图像集合。其通过梯次向下采样获得，直到达到某个终止条件才停止采样。金字塔的底部是待处理图像的高分辨率表示，而顶部是低分辨率的近似。我们将一层一层的图像比喻成金字塔，层级越高，则图像越小，分辨率越低。

一般情况下常用的有：高斯金字塔、拉普拉斯金字塔；

- 高斯金字塔(Gaussianpyramid): 用来向下采样，主要的图像金字塔；
- 拉普拉斯金字塔(Laplacianpyramid): 用来从金字塔低层图像重建上层未采样图像，在数字图像处理中也即是预测残差，可以对图像进行最大程度的还原，配合高斯金字塔一起使用。

两者的简要区别：高斯金字塔用来向下降采样图像，而拉普拉斯金字塔则用来从金字塔底层图像中向上采样重建一个图像。

下采样的基本步骤：

- 对图像 Gi 进行高斯卷积核（高斯滤波）；其中，高斯核卷积运算（高斯滤波）就是对整幅图像进行加权平均的过程，每一个像素点的值，都由其本身和邻域内的其他像素值（权重不同）经过加权平均后得到。
- 删除所有的偶数行和列。

上采样的基本步骤：

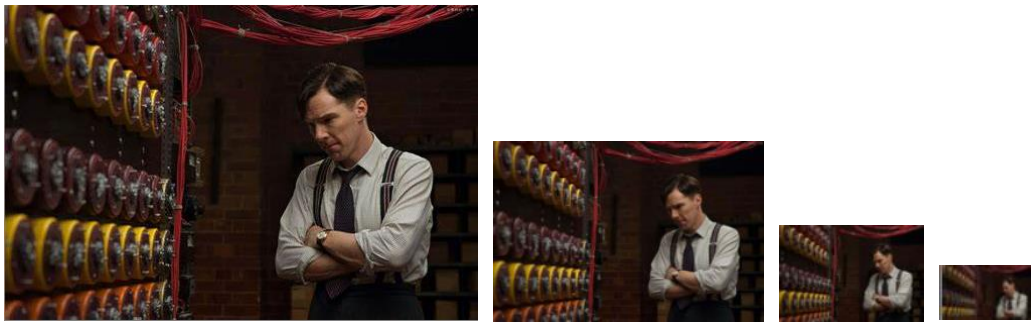
- 在图像向上取样是由小图像不断放图像的过程。它将图像在每个方向上

扩大为原图像的 2 倍，新增的行和列均用 0 来填充；

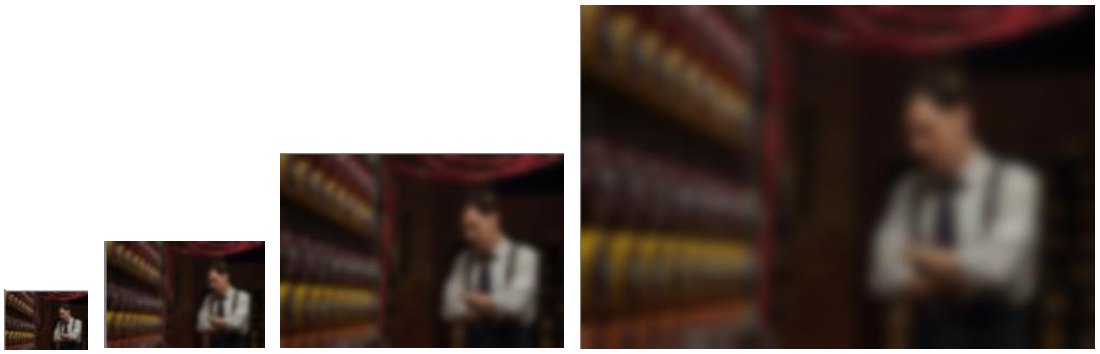
- 并使用与“向下取样”相同的卷积核乘以 4，再与放大后的图像进行卷积运算，以获得“新增像素”的新值。

1.5.3 高斯金字塔实验结果

高斯金字塔实现的**下采样结果**如下：



高斯金字塔实现的**上采样结果**如下：



结果分析：

通过以上的变换结果可知：下采样会逐渐丢失图像的信息，向上采用得到的图像即为放大后的图像，但是与原来的图像相比会发觉比较模糊。即**向上采样和向下采样不是互逆操作**，经过两种操作后，无法恢复原有的图像。

1.5.4 拉普拉斯金字塔原理和实验结果

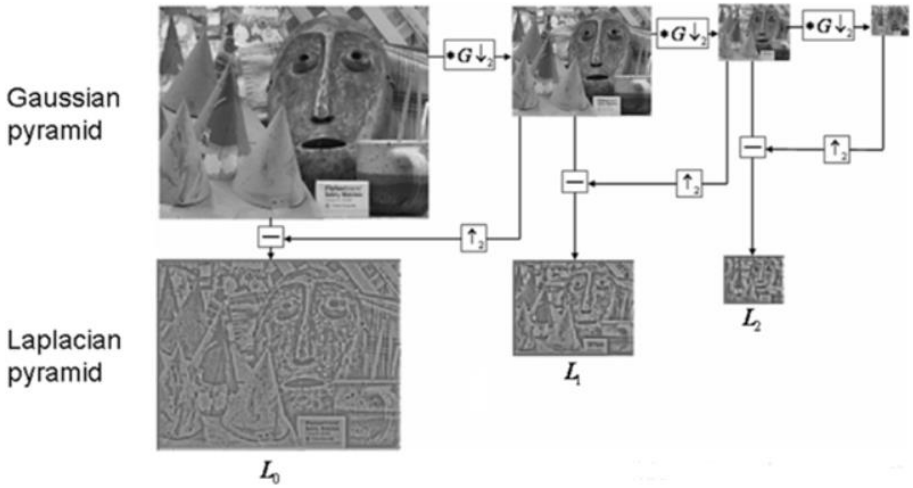
前面提到的均是高斯金字塔(使用高斯核)，下面介绍拉普拉斯(Laplacian) 金字塔，拉普拉斯(Laplacian) 金字塔是在高斯金字塔的基础上新的金字塔。

拉普拉斯(Laplacian) 金字塔的表达式：

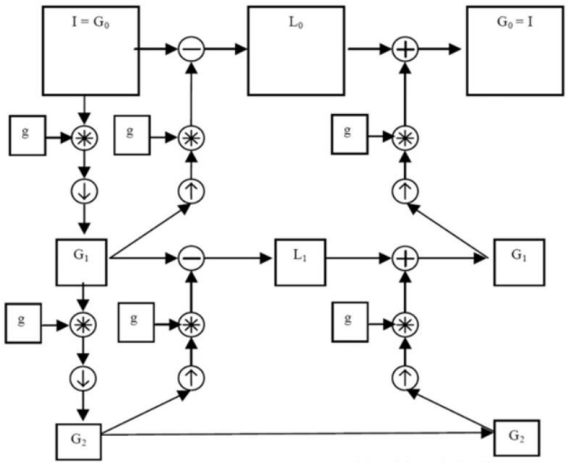
$$L_i = G_i - PyrUp(PyrDown(G_i))$$

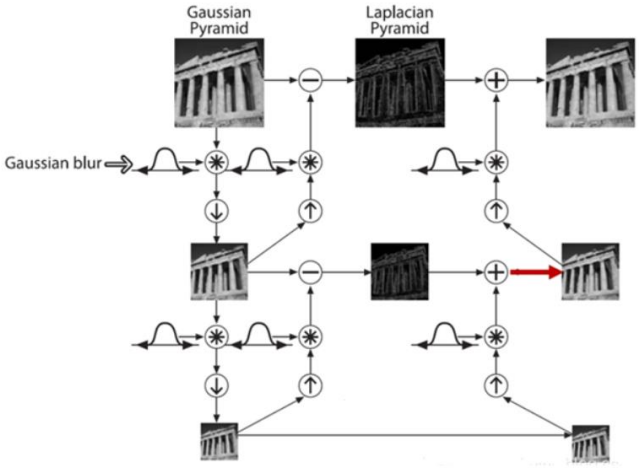
其中，G_i 是原始图像、L_i 是拉普拉斯金字塔图像。

拉普拉斯每一层表示如下图所示：



下图高斯金字塔和拉普拉斯金字塔 交叉使用得到不同的图像。





实验结果如下：

原图像

拉普拉斯金字塔第 0 层

拉普拉斯金字塔第 1 层

拉普拉斯金字塔第 2 层

拉普拉斯金字塔第 3 层

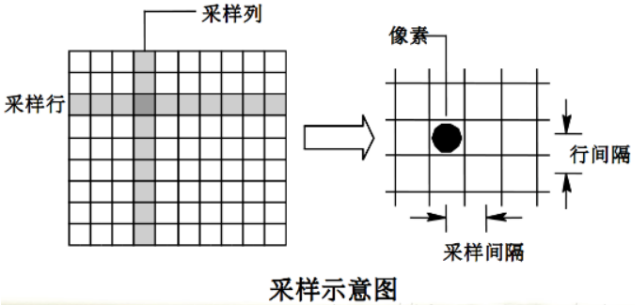
结果分析：

由以上的原理可知，拉普拉斯(Laplacian) 金字塔是在高斯金字塔的基础上新的金字塔。在高斯金字塔的运算过程中，图像经过下采样操作后会丢失部分高频细节信息。拉普拉斯金字塔就是记录高斯金字塔每一级下采样后再上采样与下采样前的差异，目的是为了能够完整的恢复出每一层级的下采样前图像。

1.5.5 讨论前置低通滤波与抽样频率的关系

低通滤波(Low-pass filter) 是一种过滤方式，规则为低频信号能正常通过，而超过设定临界值的高频信号则被阻隔、减弱。在数字图像处理领域，从频域看，低通滤波可以对图像进行“平滑去噪”处理。所以，低通滤波可以减少图像变换的幅度。

图像的抽样频率即是图像的采样频率。所谓采样，就是把一幅连续图像在空间上分割成 M×N 个网格，每个网格用一亮度值来表示。一个网格称为一个像素。M×N 的取值满足采样定理。图像采样示意图如下所示：



采样示意图

对于一幅图像，采样点数越多，图像质量越好；当采样点数减少时，图上的块状效应就逐渐明显。而低通滤波的目的是减少图像变化的幅度。

2 特征检测

2.1 特征检测的要求

- 基于高斯一阶微分的图像梯度（幅值图与方向图），分析高斯方差对图像梯度的影响；
- 掌握 canny 边缘检测原理，完成图像的边缘检测实验，展示每个环节的处理结果（梯度图、NMS、边缘链接）；
- 掌握 harris 角点检测原理，完成图像的角点检测实验，分析窗口参数对角点检测影响，讨论角点检测的不变性、等变性与定位精度等。

2.2 实验原理

2.2.1 Canny 边缘检测原理

John Canny 于 1986 年提出 Canny 算子，是一种先平滑后求导数的方法。其步骤和原理如下：

(1). **对原始图像进行灰度化：**Canny 算法通常处理的图像为灰度图，因此如果摄像机获取的是彩色图像，那首先就得进行灰度化。对一幅彩色图进行灰度化，就是根据图像各个通道的采样值进行加权平均。

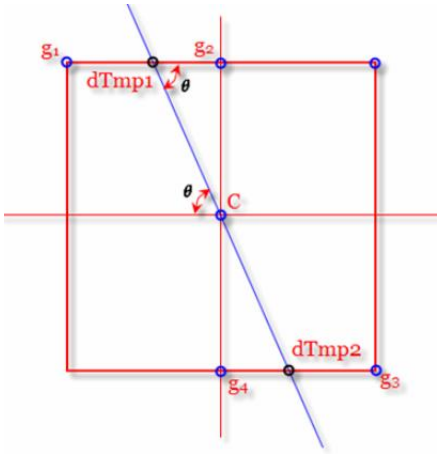
(2). **对图像进行高斯滤波：**图像高斯滤波的实现可以用两个一维高斯核，分别进行两次加权实现，也可以通过一个二维高斯核一次卷积实现。图像高斯滤波：就是根据待滤波的像素点及其邻域点的灰度值，按照一定的参数规则进行加权平均，这样可以有效滤去理想图像中叠加的高频噪声。

(3). **用一阶偏导的有限差分来计算梯度的幅值和方向：**关于图像灰度值得梯度可使用一阶有限差分来进行近似，这样就可以得图像在 x 和 y 方向上偏导数的两个矩阵。常用的梯度算子有如下几种：Roberts 算子、Sobel 算子、Prewitt 算子等。

(4). **对梯度幅值进行非极大值抑制：**图像梯度幅值矩阵中的元素值越大，说明图像中该点的梯度值越大，但这不能说明该点就是边缘（这仅仅是属于图像增

强的过程）。在 Canny 算法中，非极大值抑制是进行边缘检测的重要步骤，通俗意义上是指**寻找像素点局部最大值**，将非极大值点所对应的灰度值置为 0，这样可以剔除掉一大部分非边缘的点。

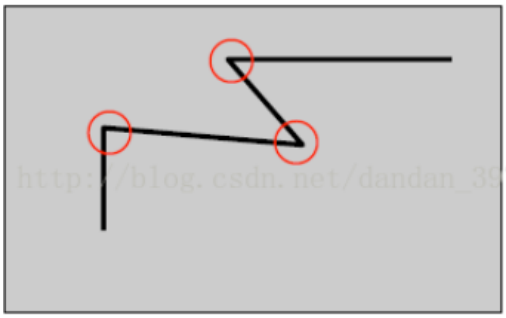
非极大值抑制原理：根据下图可知，要进行非极大值抑制，就首先要确定像素点 C 的灰度值在其 8 值邻域内是否为最大。下图中蓝色的线条方向为 C 点的梯度方向，这样就可以确定其局部的最大值肯定分布在这条线上，也即出了 C 点外，梯度方向的交点 dTmp1 和 dTmp2 这两个点的值也可能会是局部最大值。因此，判断 C 点灰度与这两个点灰度大小即可判断 C 点是否为其邻域内的局部最大灰度点。如果经过判断，C 点灰度值小于这两个点中的任一个，那就说明 C 点不是局部极大值，那么则可以排除 C 点为边缘。这就是非极大值抑制的工作原理。



(5). **用双阈值算法检测和连接边缘：**Canny 算法中减少假边缘数量的方法是采用双阈值法。选择两个阈值，根据高阈值得到一个边缘图像，这样一个图像含有很少的假边缘，但是由于阈值较高，产生的图像边缘可能不闭合，未解决这样一个问题采用了另外一个低阈值。在高阈值图像中把边缘链接成轮廓，当到达轮廓的端点时，该算法会在断点的 8 邻域点中寻找满足低阈值的点，再根据此点收集新的边缘，直到整个图像边缘闭合。

2.2.2 Harris 角点检测原理

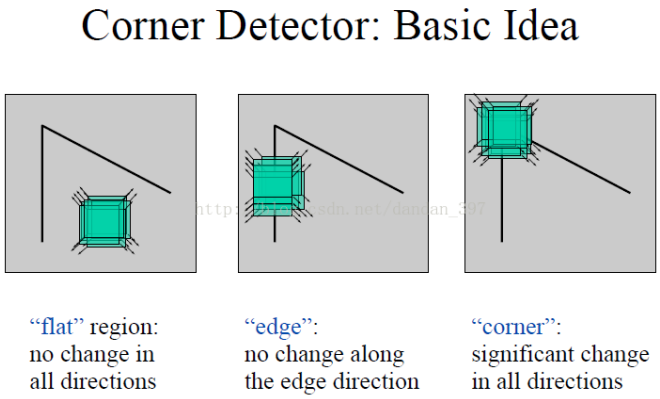
角点：通常意义上来说，角点就是极值点，即在某方面属性特别突出的点，是在某些属性上强度最大或者最小的孤立点、线段的终点。 对于图像而言，如下图所示圆圈内的部分，即为图像的角点，其是物体轮廓线的连接点。



角点在保留图像图形重要特征的同时，可以有效地减少信息的数据量，使其信息的含量很高，有效地提高了计算的速度，有利于图像的可靠匹配，使得实时处理成为可能。

角点检测的基本思想：

取某个像素的一个邻域窗口，当这个窗口在各个方向上进行小范围移动时，观察窗口内平均的像素灰度值的变化（即，E(u,v)，Window-averaged change of intensity）。从下图可知，我们可以将一幅图像大致分为三个区域（‘flat’，‘edge’，‘corner’），这三个区域变化是不一样的。



当窗口发生[u,v]移动时，那么滑动前与滑动后对应的窗口中的像素点灰度变化描述如下：

$$E(u,v)=\sum_{x,y} w(x,y)\left[I(x+u,y+v)-I(x,y)\right]^2$$

其中：

- u,v 是窗口在水平、竖直方向上的偏移；
- (x,y)是窗口内所对应的像素坐标位置，窗口有多大，就有多少个位置；
- $w(x,y)$ 是窗口函数，最简单情形就是窗口内的所有像素所对应的 w 权重系数均为 1。

根据上述表达式，当窗口处在平坦区域上滑动，可以想象的到，灰度不会发生变

化，那么 $E(u,v)=0$；如果窗口处在比纹理比较丰富的区域上滑动，那么灰度变化会很大。算法最终思想就是计算灰度发生较大变化时所对应的位置，当然这个较大是指针任意方向上的滑动，并非单指某个方向。

下面对 $E(u,v)$ 表达式进一步化简：

对 I(x+u,y+v)进行二维泰勒级数展开，我们取一阶近似，有

$$E(u,v) \approx \sum_{x,y} w(x,y)[\underbrace{I(x,y)+uI_x+vI_y}_{I(x+u,y+v)\ \text{First order approx}} - I(x,y)]^2$$

$$= \sum_{x,y} w(x,y)[uI_x+vI_y]^2$$

$$= (u \quad v)\sum_{x,y} w(x,y)\begin{bmatrix} I_xI_x & I_xI_y \\ I_xI_y & I_yI_y \end{bmatrix}\begin{pmatrix} u \\ v \end{pmatrix}$$

图中蓝线圈出的部分我们称之为结构张量（structure tensor），用 M 表示。我们的目的是寻找这样的像素点，它使得我们的 u，v 无论怎样取值，E(u,v)都是变化比较大的，这个像素点就是我们要找的角点。不难发现，上式近似处理后的 E(u,v)是一个二次型，而由下述定理可知：

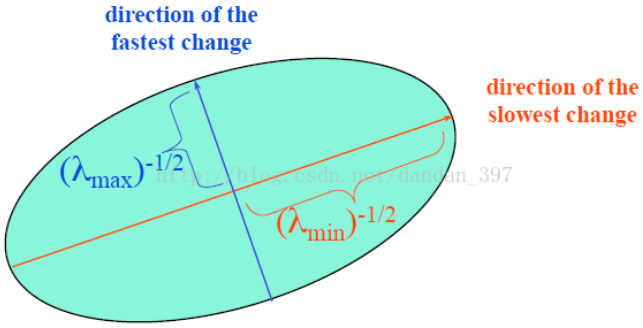
定理1 任给二次型 $f=\sum_{i,j=1}^{n} a_{ij}x_ix_j\ (a_{ij}=a_{ji})$，总有

正交变换 $x=Py$，使 f 化为标准形

$$f=\lambda_1y_1^2+\lambda_2y_2^2+\cdots+\lambda_ny_n^2,$$

其中 $\lambda_1,\lambda_2,\cdots,\lambda_n$ 是 f 的矩阵 $A=(a_{ij})$ 的特征值.

令 E(u,v)=常数，我们可用一个椭圆来描绘这一函数。

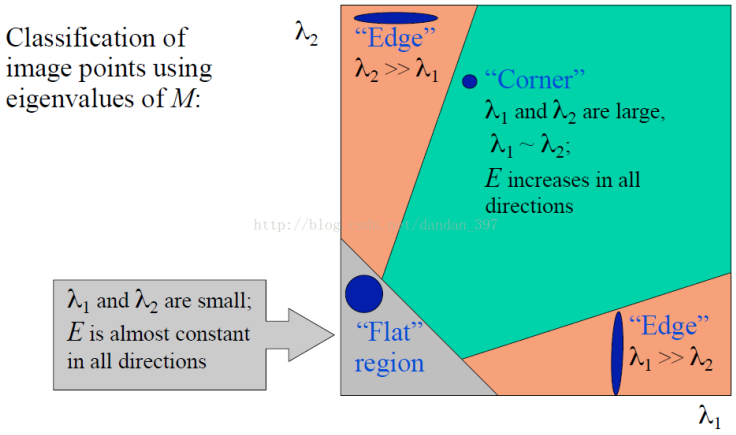


椭圆的长短轴是与结构张量 M 的两个特征值相对应的量。通过判断的情况我们就可以区分出‘flat’，‘edge’，‘corner’这三种区域，因为最直观的印象：

- corner：在水平、竖直两个方向上变化均较大的点，即 Ix、Iy 都较大；
- edge：仅在水平或仅在竖直方向有较大的点，即 Ix 和 Iy 只有其一较大；

■ flat：在水平、竖直方向的变化量均较小的点，即 Ix、Iy 都较小；

而结构张量 M 是由 Ix，Iy 构成，它的特征值正好可以反映 Ix，Iy 的情况。因此可以得到结论：



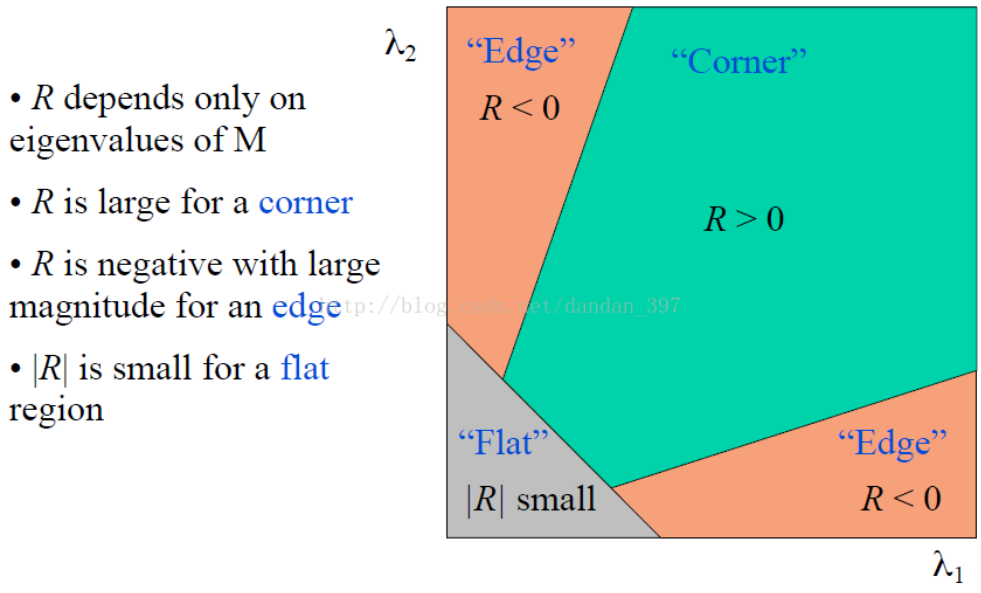
这样通过判断两个变量的值来判断角点不是很方便。于是定义角点响应函数 R（corner response function）：

$$R = \det M - k\left(\operatorname{trace} M\right)^2$$

$$\det M = \lambda_1\lambda_2$$
$$\operatorname{trace} M = \lambda_1 + \lambda_2$$

(k – empirical constant, k = 0.04-0.06)

针对三种区域，R 的取值：



至此，就可以通过判断 R 的值来判断某个点是不是角点了：

- 角点：R 为大数值整数；
- 边缘：R 为大数值负数；
- 平坦区：绝对值 R 是小数值。

Harris 角点检测算法步骤：

(1). 利用 Soble 计算出 XY 方向的梯度值；

(2). 计算出 Ix^2,Iy^2,Ix*Iy；

(3). 利用高斯函数对 Ix^2,Iy^2,Ix*Iy 进行滤波；

(4). 计算局部特征结果矩阵 M 的特征值和响应函数 C(i,j)=Det(M)-k(trace(M))^2 (0.04<=k<=0.06)；

(5). 将计算出响应函数的值 C 进行非极大值抑制，滤除一些不是角点的点，同时要满足大于设定的阈值。

2.3 实验结果

2.3.1 分析高斯方差对图像梯度的影响

基于高斯一阶微分的图像梯度（幅值图与方向图），分析高斯方差对图像梯度的影响：

原图像如下：

当高斯方差：$\sigma = 1$时的结果如下：

（从左到右以此是：高斯滤波后的图、幅度图、方向图）

当高斯方差：$\sigma = 5$时的结果如下：

当高斯方差：$\sigma = 0.5$时的结果如下：

结果分析：由以上的图像结果可以看出：

当高斯方差变大时，图像边界的梯度范围变大；

当高斯方差变小时，图像边界的梯度范围变小。

2.3.2 Canny 边缘检测结果

通过以上的原理以及 canny 边缘检测的步骤，编程得到实验结果如下：

原图

高斯滤波后的图像

梯度图

非极大值抑制（NMS）图

边缘链接图

2.3.3 Harris 角点检测结果

在其他参数不变的情况下，改变σ的值，观察角点检测的结果如下：

当$\sigma = 0.2$ 和$\sigma = 0.5$ 时的检测结果


当$\sigma=1$ 和$\sigma=2$ 时的检测结果


当$\sigma=5$ 和$\sigma=10$ 时的检测结果

结果分析：当只改变σ时，检测的效果随着σ的值变化而变化，σ在增大的过程中，检测的点数先减小后增加，并没有太明显的规律。

下面**验证参数α对结果的影响：**

在数学原理上：假设已经得到了矩阵 M 的特征值 $\lambda_1 \geq \lambda_2 \geq 0$，令 $\lambda_2=k\lambda_1, 0 \leqslant k \leqslant 1$。由特征值与矩阵 M 的直迹和行列式的关系可得：

$$detM=\prod_i \lambda_i \qquad traceM=\sum_i \lambda_i$$

从而可以得到角点的响应：

$$R=\lambda_2\lambda_2=\alpha(\lambda_2+\lambda_2)^2=\lambda^2(k-\alpha(1+k)^2)$$

假设 $R \geqslant 0$，则有：

$$0 \leq \alpha \leq \frac{k}{(1+k)^2} \leq 0.25$$

对于较小的 k 值，$R \approx \lambda^2(k-\alpha), \alpha<k$。

即增大α的值，将减小角点响应值 R，降低角点检测的灵性，减少被检测角点的数量；减小α值，将增大角点响应值 R，增加角点检测的灵敏性，增加被检测角点的数量。

实验结果如下：

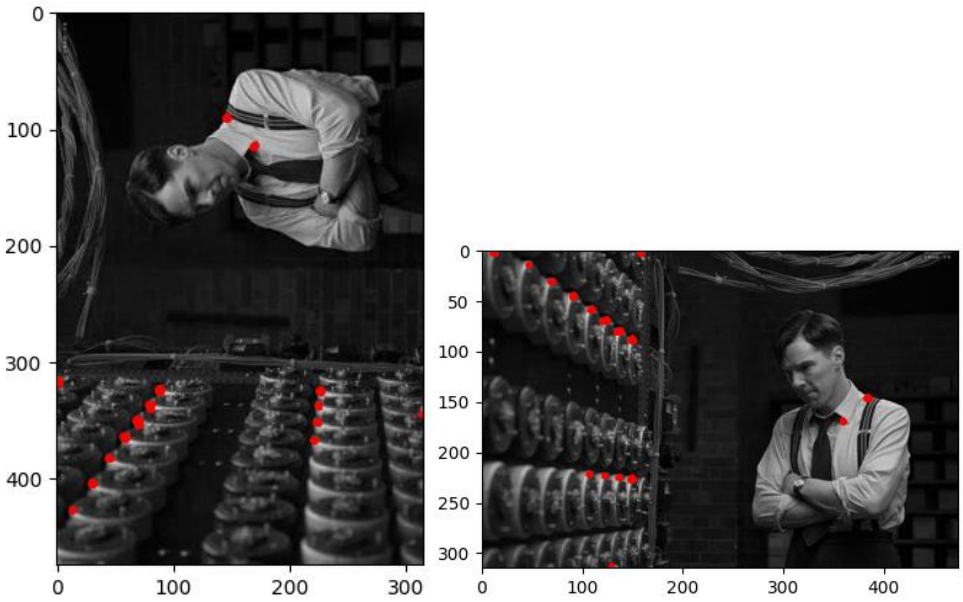
参数α=0.01 和α=0.05 时的检测结果

参数α=0.1 和α=0.2 时的检测结果

参数α=0.22 和α=0.25 时的检测结果

由以上的结果可以得知：随着α的增大，检测得到的角点个数在减少。当α大于一定值时，检测的个数为 0。由此可见验证了上面的结论：增大 α 的值，将减小角点响应值 R，降低角点检测的灵性，减少被检测角点的数量；减小 α 值，将增大角点响应值 R，增加角点检测的灵敏性，增加被检测角点的数量。

对图像进行欧式变换后，再进行检测：



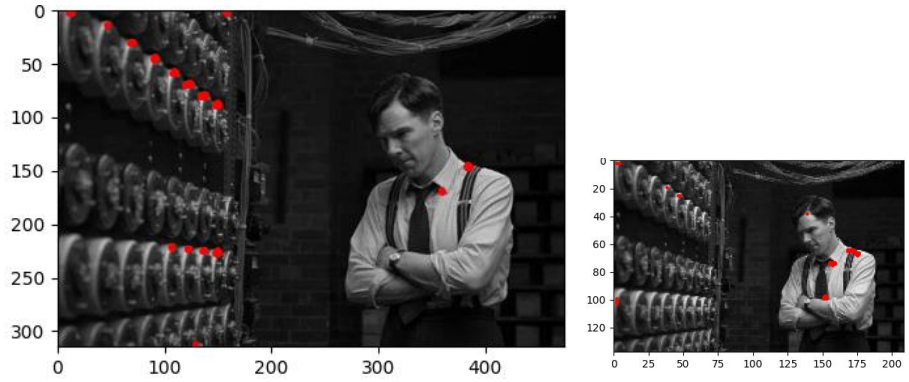
由此可见：Harris 角点检测算子具有旋转不变性的特点。

理论分析：Harris 角点检测算子使用的是角点附近的区域灰度二阶矩矩阵。而二阶矩矩阵可以表示成一个椭圆，椭圆的长短轴正是二阶矩矩阵特征值平方根的倒数。当特征椭圆转动时，特征值并不发生变化，所以判断角点响应值 R 也不发生变化，由此说明 **Harris 角点检测算子具有旋转不变性**。

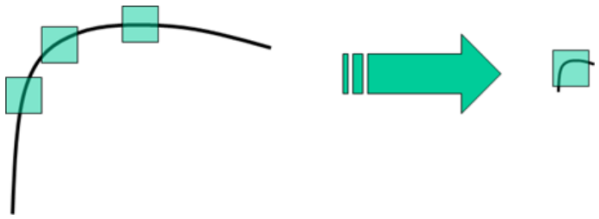
下面改变图像的亮度，（通过读取图像，然后增加 20 个像素值，大于 255 后置为 255），然后进行角点检测，得到以下的结果：

结果分析：由此可见，Harris 角点检测算子对亮度和对比度的变化不敏感。这是因为在进行 Harris 角点检测时，使用了微分算子对图像进行微分运算，而微分运算对图像密度的拉升或收缩和对亮度的抬高或下降不敏感。换言之，对亮度和对比度的仿射变换并不改变 Harris 响应的极值点出现的位置，但是，由于阈值的选择，可能会影响角点检测的数量。

下面改变图像的尺寸大小，得到角点检测的结果如下：



由结果可见：Harris 角点检测算子不具有尺度不变性。这是因为当图像被缩小时，在检测窗口尺寸不变的前提下，在窗口内所包含图像的内容是完全不同的。放大的图像可能被检测为边缘或曲线，而缩小的图像则可能被检测为一个角点。可以用下图得以解释：



3 代码附录

3.1 图像平移代码

```
1.  import cv2
2.  import numpy as np
3.  img = cv2.imread('picture.jpg')#读取原图片，左上为原点
4.  rows = img.shape[0] #取图片的行数，即高度
5.  cols = img.shape[1] #取图片的列数，即宽度
6.  result=np.zeros((rows,cols,3),dtype=np.uint8) #创建一样大小的转换结果
7.  delta_x=int(input("请输入 x 方向偏移量："))      #行的变化量，即平移量
8.  delta_y=int(input("请输入 y 方向偏移量："))      #列的变化取，即平移量
9.  transform=np.array([[1,0,delta_x],[0,1,delta_y],[0,0,1]])    #转换矩阵
10. cv2.imshow('original_picture',img)#显示原图片
11. for i in range(rows):
```

```
    for j in range(cols):
        img_pos=np.array([i,j,1])           #记录结果位置
        [x, y, z] = np.dot(transform, img_pos)          #转换为原图位置坐标
        x = int(x)                          #取整
        y = int(y)                          #取整
        if x >= rows or y >= cols or x < 0 or y < 0: #如果出界
            result[i][j] = 255                      #该点为白色
        else:
            result[i][j] = img[x][y]            #不出界把原图位置对应值取来
cv2.imshow('result_process', result)                    #显示结果
cv2.waitKey(0)                                  #按任意键继续
```

3.2 图像旋转代码

```
import cv2
import numpy as np
import math
img = cv2.imread('picture.jpg')#读取原图像，左上为原点
rows = img.shape[0] #取图片的行数，即高度
cols = img.shape[1] #取图片的列数，即宽度
print("图像的高度：",rows,"图像的宽度：",cols)
center_x=int(input("请输入旋转中心 x：（请在图像的宽度范围内输入）："))
center_y=int(input("请输入旋转中心 y：（请在图像的宽度范围内输入）："))
center=[center_x,center_y]    #设置图片中心
result=np.zeros((rows,cols,3),dtype=np.uint8) #创建一样大小的转换结果
beta=int(input("请输入旋转角度："))*math.pi/180
transform=np.array([[math.cos(beta),-math.sin(beta),0],
                    [math.sin(beta), math.cos(beta),0],
                    [0,0,1]])   #转换矩阵
cv2.imshow('original_picture',img)#显示原图片
for i in range(rows):
    for j in range(cols):
        img_pos=np.array([i-center[0],j-center[1],1])   #记录结果位置
        [x, y, z] = np.dot(transform, img_pos)          #转换为原图位置坐标
        x = int(x)+center[0]                        #取整
        y = int(y)+center[1]                        #取整
        if x >= rows or y >= cols or x < 0 or y < 0: #如果出界
            result[i][j] = 255                      #该点为白色
        else:
            result[i][j] = img[x][y]    #不出界把原图位置对应值取来
cv2.imshow('result_process', result)                    #显示结果
```

3.3 图像的欧式变换代码（平移+旋转）

```
#等距变换代码：等距变换=旋转+平移
import cv2
import numpy as np
import math
img = cv2.imread('picture.jpg')#读取原图像，左上为原点
rows = img.shape[0] #取图片的行数，即高度
cols = img.shape[1] #取图片的列数，即宽度
print("图像的高度：",rows,"图像的宽度：",cols)
delta_x=int(input("请输入 x 方向偏移量："))      #行的变化量，即平移量
delta_y=int(input("请输入 y 方向偏移量："))      #列的变化取，即平移量
center_x=int(input("请输入旋转中心 x：（请在图像的宽度范围内输入）："))
center_y=int(input("请输入旋转中心 y：（请在图像的宽度范围内输入）："))
center=[center_x,center_y]    #设置图片中心
result=np.zeros((rows,cols,3),dtype=np.uint8) #创建一样大小的转换结果
beta=int(input("请输入旋转角度："))*math.pi/180
transform=np.array([[math.cos(beta),-math.sin(beta),delta_x],
                    [math.sin(beta), math.cos(beta),delta_y],
                    [0,0,1]])   #转换矩阵
cv2.imshow('original_picture',img)#显示原图片
for i in range(rows):
    for j in range(cols):
        img_pos=np.array([i-center[0],j-center[1],1])   #记录结果位置
        [x, y, z] = np.dot(transform, img_pos)          #转换为原图位置坐标
        x = int(x)+center[0]                        #取整
        y = int(y)+center[1]                        #取整
        if x >= rows or y >= cols or x < 0 or y < 0: #如果出界
            result[i][j] = 255                          #该点为白色
        else:
            result[i][j] = img[x][y]            #不出界把原图位置对应值取来
cv2.imshow('result_process', result)                    #显示结果
cv2.waitKey(0)                                      #按任意键继续
```

3.4 图像的相似变换代码

```
import cv2
import numpy as np
import math
img = cv2.imread('picture.jpg')#读取图片成张量，左上为原点
rows = img.shape[0] #取图片的行数，即高度
cols = img.shape[1] #取图片的列数，即宽度
print("图像的高度：",rows,"图像的宽度：",cols)
```

```
delta_x=int(input("请输入 x 方向偏移量："))      #行的变化量，即平移量
delta_y=int(input("请输入 y 方向偏移量："))      #列的变化取，即平移量
center_x=int(input("请输入旋转中心 x：（请在图像的宽度范围内输入）："))
center_y=int(input("请输入旋转中心 y：（请在图像的宽度范围内输入）："))
center=[center_x,center_y]    #设置图片中心
result=np.zeros((rows,cols,3),dtype=np.uint8) #创建一样大小的转换结果
beta=int(input("请输入旋转角度："))*math.pi/180
s=1/float(input("请输入相似比例："))      #相似比
transform=np.array([[s*math.cos(beta),-s*math.sin(beta),delta_x],
                    [s*math.sin(beta), s*math.cos(beta),delta_y],
                    [0,0,1]])  #构建转换矩阵
cv2.imshow('original_picture',img)#显示原图片
for i in range(rows):
    for j in range(cols):
        img_pos=np.array([i-center[0],j-center[1],1])       #记录结果位置
        [x, y, z] = np.dot(transform, img_pos)          #转换为原图位置坐标
        x = int(x)+center[0]                        #取整
        y = int(y)+center[1]                        #取整
        if x >= rows or y >= cols or x < 0 or y < 0: #如果出界
            result[i][j] = 255                          #该点为白色
        else:
            result[i][j] = img[x][y]              #不出界把原图位置对应值取来
cv2.imshow('result_process', result)              #显示结果
cv2.waitKey(0)                                   #按任意键继续
```

3.5 图像的仿射变换代码

```
import cv2
import numpy as np
img = cv2.imread('picture.jpg')#读取图片成(400,400,3)的张量，左上为原点
rows = img.shape[0] #取图片的行数，即高度
cols = img.shape[1] #取图片的列数，即宽度
print("图像的高度：",rows,"图像的宽度：",cols)
center=[0,0]    #设置图片中心
result=np.zeros((rows,cols,3),dtype=np.uint8) #创建一样大小的转换结果
a11=float(input("请输入仿射矩阵 s_x："))
a12=float(input("请输入仿射矩阵 a_x："))
a21=float(input("请输入仿射矩阵 a_y："))
a22=float(input("请输入仿射矩阵 s_y："))
delta_x=int(input("请输入 x 方向偏移量 t_x："))      #行的变化量，即平移量
delta_y=int(input("请输入 y 方向偏移量 t_y："))      #列的变化取，即平移量
transform=np.array([[a11,a12,delta_x],
                    [a21, a22,delta_y],
                    [0,    0,     1]])   #转换矩阵
```

```
cv2.imshow('original_picture',img)#显示原图片
for i in range(rows):
    for j in range(cols):
        img_pos=np.array([i-center[0],j-center[1],1])          #记录结果位置
        [x, y, z] = np.dot(transform, img_pos)          #转换为原图位置坐标
        x = int(x)+center[0]                        #取整
        y = int(y)+center[1]                        #取整
        if x >= rows or y >= cols or x < 0 or y < 0: #如果出界
            result[i][j] = 255                          #该点为白色
        else:
            result[i][j] = img[x][y]            #不出界把原图位置对应值取来
cv2.imshow('result_process', result)            #显示结果
cv2.waitKey(0)                                  #按任意键继续
```

3.6 图像的投影变换代码

```
import cv2
import numpy as np
img = cv2.imread('picture.jpg')#读取图片成(400,400,3)的张量，左上为原点
rows = img.shape[0] #取图片的行数，即高度
cols = img.shape[1] #取图片的列数，即宽度
print("图像的高度：",rows,"图像的宽度：",cols)
center=[0,0]    #设置图片中心
result=np.zeros((rows,cols,3),dtype=np.uint8) #创建一样大小的转换结果
h11=float(input("请输入射影矩阵 h11："))
h12=float(input("请输入射影矩阵 h12："))
h13=float(input("请输入射影矩阵 h13："))
h21=float(input("请输入射影矩阵 h21："))
h22=float(input("请输入射影矩阵 h22："))
h23=float(input("请输入射影矩阵 h23："))
h31=float(input("请输入射影矩阵 h31："))
h32=float(input("请输入射影矩阵 h32："))
transform=np.array([[h11,h12,h13],
                    [h21, h22,h23],
                    [h31,h32,1]])   #转换矩阵
transform=np.linalg.inv(transform)
cv2.imshow('original_picture',img)#显示原图片
Z=1
for i in range(rows):
    for j in range(cols):
        img_pos=np.array([i-center[0],j-center[1],1])    #记录结果位置
        [x, y, z] = np.dot(transform, img_pos)          #转换为原图位置坐标
        x = int(x/Z)+center[0]                      #取整
        y = int(y/Z)+center[1]                      #取整
```

```
        if x >= rows or y >= cols or x < 0 or y < 0: #如果出界
            result[i][j] = 255                       #该点为白色
        else:
            result[i][j] = img[x][y]             #不出界把原图位置对应值取来
cv2.imshow('result_process', result)             #显示结果
cv2.waitKey(0)                                   #按任意键继续
```

3.7 高斯金字塔代码

```
import cv2
img=cv2.imread('picture.jpg')
cv2.imshow("original_picture",img)
level=3 #设置金字塔的层数
temp=img.copy()
gaosi_img=[]
for i in range(level):
    dst=cv2.pyrDown(temp)
    gaosi_img.append(dst)
    cv2.imshow("gsd"+str(i),dst)
    temp=dst.copy()
for i in range(level):
    dst=cv2.pyrUp(temp)
    gaosi_img.append(dst)
    cv2.imshow("gsu"+str(i),dst)
    temp=dst.copy()
cv2.waitKey(0)
cv2.destroyAllWindows()
```

3.8 拉普拉斯金字塔代码

```
   import cv2
import numpy as np
import matplotlib.pyplot as plt
# 读取原始图像
img = cv2.imread('picture.jpg')
img = cv2.resize(img,(512,256))
r1 = cv2.pyrDown(img) # 图像向下取样
r2 = cv2.pyrUp(r1) # 图像向上取样
LapPyr0 = img - r2  # 拉普拉斯第 0 层

r3 = cv2.pyrDown(r1) # 图像向下取样
r4 = cv2.pyrUp(r3) # 图像向上取样
LapPyr1 = r1 - r4  # 拉普拉斯第 1 层
```

```
14.
15. r5 = cv2.pyrDown(r3) # 图像向下取样
16. r6 = cv2.pyrUp(r5) # 图像向上取样
17. LapPyr2 = r3 - r6  # 拉普拉斯第 2 层
18.
19. r7 = cv2.pyrDown(r5) # 图像向下取样
20. r8 = cv2.pyrUp(r7) # 图像向上取样
21. LapPyr3 = r5 - r8  # 拉普拉斯第 3 层
22.
23. cv2.imshow('original', img)
24. cv2.imshow('LapPyr0', LapPyr0)
25. cv2.imshow('LapPyr1', LapPyr1)
26. cv2.imshow('LapPyr2', LapPyr2)
27. cv2.imshow('LapPyr3', LapPyr3)
28. cv2.waitKey()
29. cv2.destroyAllWindows()
```

3.9 分析高斯方差对图像梯度的影响代码

```
1.  import numpy as np
2.  import math
3.  import matplotlib.pyplot as plt
4.  import cv2 as cv
5.  #读取图片
6.  img=cv.imread('picture.jpg')
7.  #创建高斯矩阵核
8.  def create_gaussion_kernel(sigma1=1.0,sigma2=1.0):
9.      sum=0.0
10.     gaussian=np.zeros([5,5])
11.     for i in range(5):
12.         for j in range(5):
13.             gaussian[i, j] = math.exp(-
    1 / 2 * ((i - 2)**2 / (sigma1)**2+((j - 2)**2 / (sigma2)**2))) / (2 * math.p
    i * sigma1 * sigma2)
14.             sum=sum+gaussian[i,j]
15.
16.     gaussian=gaussian/sum
17.     return gaussian
18. def rgbTogray(rgb):
19.     return np.dot(rgb[..., :3], [0.299, 0.587, 0.114])
20.
21. #1 高斯滤波
22. def gaussion_filter(img,sigma1,sigma2):
23.     gaussian=create_gaussion_kernel(sigma1, sigma2)
```

```
    gray=rgbTogray(img)
    plt.figure(figsize=(3,3))
    plt.imshow(gray, cmap="gray")
    plt.title('original')
    W, H = gray.shape
    new_gray = np.zeros([W - 4, H - 4])
    for i in range(W - 4):
        for j in range(H - 4):
            new_gray[i, j] = np.sum(gray[i:i + 5, j:j + 5] * gaussian)  # 与高斯矩阵卷积实现滤波
    plt.figure(figsize=(3, 3))
    plt.imshow(new_gray, cmap="gray")
    plt.title('gaussion')
    return new_gray

#2 求一阶梯度幅值
def one_d(new_gray):
    W1, H1 = new_gray.shape
    dx = np.zeros([W1 - 1, H1 - 1])
    dy = np.zeros([W1 - 1, H1 - 1])
    d = np.zeros([W1 - 1, H1 - 1])
    for i in range(W1 - 1):
        for j in range(H1 - 1):
            dx[i, j] = new_gray[i, j + 1] - new_gray[i, j]
            dy[i, j] = new_gray[i + 1, j] - new_gray[i, j]
            d[i, j] = np.sqrt(np.square(dx[i, j]) + np.square(dy[i, j]))  # 图像梯度幅值作为图像强度值
    plt.figure(figsize=(3, 3))
    plt.imshow(d, cmap="gray")
    plt.title('Amplitude_Picture')
def one_drection(new_gray):
    W1, H1 = new_gray.shape
    dx = np.zeros([W1 - 1, H1 - 1])
    dy = np.zeros([W1 - 1, H1 - 1])
    d = np.zeros([W1 - 1, H1 - 1])
    for i in range(W1 - 1):
        for j in range(H1 - 1):
            dx[i, j] = new_gray[i, j + 1] - new_gray[i, j]
            dy[i, j] = new_gray[i + 1, j] - new_gray[i, j]
            d[i, j] = np.arctan((dy[i, j])/dx[i, j])  # 图像梯度方向作为图像强度值
    plt.figure(figsize=(3, 3))
    plt.imshow(d, cmap="gray")
    plt.title('Drection_Picture')
```

```
new_gray=gaussion_filter(img,1.0,1.0)
one_d(new_gray)
one_drection(new_gray)
new_gray=gaussion_filter(img,5.0,5.0)
one_d(new_gray)
one_drection(new_gray)
new_gray=gaussion_filter(img,0.5,0.5)
one_d(new_gray)
one_drection(new_gray)
plt.show()
```

3.10 canny 边缘检测代码

```
import numpy as np
import math
import matplotlib.pyplot as plt
import cv2 as cv
#读取图片
img=cv.imread('picture.jpg')
#创建高斯矩阵核
sigma1=sigma2=1.0
sum=0.0

gaussian=np.zeros([5,5])
for i in range(5):
    for j in range(5):
        gaussian[i, j] = math.exp(-
1 / 2 * ((i - 2)**2 / (sigma1)**2+((j - 2)**2 / (sigma2)**2))) / (2 * math.p
i * sigma1 * sigma2)
        sum=sum+gaussian[i,j]
gaussian=gaussian/sum

#print(gaussian) #显示一下高斯核

def rgbTogray(rgb):
    return np.dot(rgb[..., :3], [0.299, 0.587, 0.114])

#1 高斯滤波

gray=rgbTogray(img)
plt.figure(1)
plt.subplot(1,1,1)
plt.imshow(gray, cmap="gray")
plt.title('original')
```

```
30.
31.
32. W, H = gray.shape
33. new_gray = np.zeros([W - 4, H - 4])
34. for i in range(W - 4):
35.     for j in range(H - 4):
36.         new_gray[i, j] = np.sum(gray[i:i + 5, j:j + 5] * gaussian)  # 与高斯矩阵卷积实现滤波
37. #plt.subplot(1,2,2)
38. plt.figure(2)
39. plt.imshow(new_gray, cmap="gray")
40. plt.title('gaussion')
41. plt.show()
42.
43. #2 求一阶梯度幅值
44. W1, H1 = new_gray.shape
45. dx = np.zeros([W1 - 1, H1 - 1])
46. dy = np.zeros([W1 - 1, H1 - 1])
47. d = np.zeros([W1 - 1, H1 - 1])
48. for i in range(W1 - 1):
49.     for j in range(H1 - 1):
50.         dx[i, j] = new_gray[i, j + 1] - new_gray[i, j]
51.         dy[i, j] = new_gray[i + 1, j] - new_gray[i, j]
52.         d[i, j] = np.sqrt(np.square(dx[i, j]) + np.square(dy[i, j]))  # 图像梯度幅值作为图像强度值
53. plt.figure(3)
54. plt.imshow(d, cmap="gray")
55. plt.title('d')
56. plt.show()
57.
58. #3 NMS
59. W2, H2 = d.shape
60. NMS = np.copy(d)
61. NMS[0, :] = NMS[W2 - 1, :] = NMS[:, 0] = NMS[:, H2 - 1] = 0
62. for i in range(1, W2 - 1):
63.     for j in range(1, H2 - 1):
64.         if d[i, j] == 0:
65.             NMS[i, j] = 0
66.         else:
67.             gradX = dx[i, j]
68.             gradY = dy[i, j]
69.             gradTemp = d[i, j]
70.             # 如果Y方向幅度值较大
71.             if np.abs(gradY) > np.abs(gradX):
```

```
                weight = np.abs(gradX) / np.abs(gradY)
                grad2 = d[i - 1, j]
                grad4 = d[i + 1, j]
                # 如果 x,y 方向梯度符号相同
                if gradX * gradY > 0:
                    grad1 = d[i - 1, j - 1]
                    grad3 = d[i + 1, j + 1]
                # 如果 x,y 方向梯度符号相反
                else:
                    grad1 = d[i - 1, j + 1]
                    grad3 = d[i + 1, j - 1]
            # 如果 X 方向幅度值较大
            else:
                weight = np.abs(gradY) / np.abs(gradX)
                grad2 = d[i, j - 1]
                grad4 = d[i, j + 1]
                # 如果 x,y 方向梯度符号相同
                if gradX * gradY > 0:
                    grad1 = d[i + 1, j - 1]
                    grad3 = d[i - 1, j + 1]
                # 如果 x,y 方向梯度符号相反
                else:
                    grad1 = d[i - 1, j - 1]
                    grad3 = d[i + 1, j + 1]
            gradTemp1 = weight * grad1 + (1 - weight) * grad2
            gradTemp2 = weight * grad3 + (1 - weight) * grad4
            if gradTemp >= gradTemp1 and gradTemp >= gradTemp2:
                NMS[i, j] = gradTemp
             else:
                 NMS[i, j] = 0
plt.figure(4)
plt.imshow(NMS, cmap = "gray")
plt.title("NMS")
plt.show()

#4 双阈值检测、链接边缘
W3, H3 = NMS.shape
DT = np.zeros([W3, H3])
# 定义高低阈值
TL = 0.2 * np.max(NMS)
TH = 0.3 * np.max(NMS)
for i in range(1, W3 - 1):
    for j in range(1, H3 - 1):
        if (NMS[i, j] < TL):
```

```
116.             DT[i, j] = 0
117.         elif (NMS[i, j] > TH):
118.             DT[i, j] = 1
119.         elif ((NMS[i - 1, j - 1:j + 1] < TH).any() or (NMS[i + 1, j - 1:j + 1]).any()
120.               or (NMS[i, [j - 1, j + 1]] < TH).any()):
121.             DT[i, j] = 1
122. plt.figure(5)
123. plt.imshow(DT, cmap="gray")
124. plt.title("link")
125. plt.show()
```

3.11 Harris 角点检测代码

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.  import cv2 as cv
4.  import math
5.
6.  #数据初始处理
7.  src = cv.imread('picture.jpg')
8.  #plt.imshow(src[:,:,[2,1,0]])
9.  #plt.show()
10. #准备图片成灰度图
11. im1=cv.cvtColor(src,cv.COLOR_BGR2GRAY).astype(float)
12. #plt.imshow(im1,cmap='gray')
13. #plt.show()
14. #定义函数
15. def create_gaussion_kernel(sigma1=1.0,sigma2=1.0): #高斯核滤波算子
16.     sum=0.0
17.     gaussian=np.zeros([5,5])
18.     for i in range(5):
19.         for j in range(5):
20.             gaussian[i, j] = math.exp(-1 / 2 * ((i - 2)**2 / (sigma1)**2+((j - 2)**2 / (sigma2)**2))) / (2 * math.pi * sigma1 * sigma2)
21.             sum=sum+gaussian[i,j]
22.
23.     gaussian=gaussian/sum
24.     return gaussian
25.
26. def gauss_5x5(a):#高斯滤波算子计算函数
27.     computer=np.ones(dtype=float,shape=(5,5))
28.     for i in range(-2,3):
```

```
29.         for j in  range(-2,3):
30.             computer[i+2,j+2]=pow(math.e,(-i*i-j*j)/(2*a*a))
31.     computer=computer/np.min(computer)
32.     computer=computer.astype(int)+1
33.     computer=np.where(computer==2,1,computer)
34.     computer[2,2]=56
35.     computer=computer.astype(float)
36.     return computer
37.
38. def convolution(computer,image_1):#卷积函数
39.     image_2=image_1.copy()
40.     size_i=int(computer.shape[0]/2)
41.     a=[0,0]
42.     for b in range(0,2):
43.         if computer.shape[b]%2==0:
44.             a[b]=0
45.         else:
46.             a[b]=1
47.     size_j=int(computer.shape[1]/2)
48.     sum_1=np.sum(computer)
49.     if sum_1==0:
50.         sum_1=1
51.     for i_1 in range(size_i,image_1.shape[0]-size_i):
52.         for j_1 in range(size_j,image_1.shape[1]-size_j):
53.             image_2[i_1,j_1]=np.sum(image_1[i_1-size_i:i_1+size_i+a[1],j_1-
    size_j:j_1+size_j+a[0]]*computer)/sum_1
54.     return image_2
55.
56. def edg(edg_cp_x,edg_cp_y,im): #求梯度模
57.     im_1=im.copy()
58.     im_a=convolution(edg_cp_x,im_1)
59.     im_b=convolution(edg_cp_y,im_1)
60.     im_2=np.sqrt(im_a*im_a+im_b*im_b)
61.     return im_2
62.
63. def point_extract(image,a,b=255):#返回角点检测地址
64.     image=image*255/np.max(image)      #灰度范围从 min-max 转到 0-255
65.     # image=np.absolute(image)
66.     arry=np.where(image>=a)
67.     arry=np.array(arry)
68.     return arry
69.
70. #%%进行滤波
71. #print(gauss_5x5(1))
```

```
detect=create_gaussion_kernel(1.0,1.0)
#print(detect)
im2=convolution(detect,im1)


#%%roberts 边缘算子计算 2x2
edg1_cp_x=np.array([1,0,0,-1]).reshape(2,2)
edg1_cp_y=np.array([0,-1,1,0]).reshape(2,2)
im5=edg(edg1_cp_x,edg1_cp_y,im2)
#%%sobel 边缘算子计算 3x3
edg2_cp_x=np.array([-1,0,1,-1,0,1,-1,0,1]).reshape(3,3)
edg2_cp_y=-edg2_cp_x.T
im6=edg(edg2_cp_x,edg2_cp_y,im2)
#%% Harris 角点检测
H_x=np.array([-1,0,1,-1,0,1,-1,0,1],dtype=float).reshape(3,3)
H_y=-H_x.T
H_i_x=convolution(H_x,im2)    #x 方向梯度
H_i_y=convolution(H_y,im2)    #y 方向梯度
H_A=convolution(detect,H_i_x*H_i_x)      #w（x,y）*Ix^2
H_B=convolution(detect,H_i_y*H_i_y)      #w（x,y）*Iy^2
H_C=convolution(detect,H_i_x*H_i_y)      #w（x,y）*Ix*Iy
M=np.array([[H_A,H_C],
            [H_C,H_B]])
M_T=M.transpose((2,3,0,1))  #调换索引值
M_det=np.linalg.det(M_T)    #求|M| 行列式的值
M_trace=np.trace(M)     #求 M 的迹
M_trace=M_trace.astype(float)   #转换成 float
im3=M_det-0.04*M_trace*M_trace      #角点相应函数
#%%归一化角点提取
l1=point_extract(im3,110)              #提取相应函数大于 110 的地址 给 l1

im7=np.float32(im1)                     #转换 float32
im4=cv.cornerHarris(im7,2,3,0.04)       #im7 未高斯模糊，其输入图像必须是 float32 最后一个参数是相应函数的 elpha，blockSize，表示邻域的大小=2；ksize，表示 Sobel()算子的孔径大小=3
l2=point_extract(im4,160)              #提取相应函数大于 160 的地址 给 l2


def harris(img,sigma1,sigma2,a,r):
    detect=create_gaussion_kernel(sigma1,sigma2)
    im2=convolution(detect,img)
    H_x=np.array([-1,0,1,-1,0,1,-1,0,1],dtype=float).reshape(3,3)
    H_y=-H_x.T
    H_i_x=convolution(H_x,im2)    #x 方向梯度
```

```
    H_i_y=convolution(H_y,im2)   #y 方向梯度
    H_A=convolution(detect,H_i_x*H_i_x)     #w（x,y）*Ix^2
    H_B=convolution(detect,H_i_y*H_i_y)     #w（x,y）*Iy^2
    H_C=convolution(detect,H_i_x*H_i_y)     #w（x,y）*Ix*Iy
    M=np.array([[H_A,H_C],
                [H_C,H_B]])
    M_T=M.transpose((2,3,0,1))  #调换索引值
    M_det=np.linalg.det(M_T)    #求|M| 行列式的值
    M_trace=np.trace(M)     #求 M 的迹
    M_trace=M_trace.astype(float)   #转换成 float
    im3=M_det-a*M_trace*M_trace      #角点相应函数
    #%%归一化角点提取
    l1=point_extract(im3,r)             #提取相应函数大于 110 的地址 给 l1
    plt.figure( figsize=(5, 5))
    plt.imshow(img, cmap='gray')
    plt.scatter(l1[1, :], l1[0, :], marker='o', color='red', s=5)
    plt.show()


# src = cv.imread('picture.jpg')
# im10=cv.cvtColor(src,cv.COLOR_BGR2GRAY).astype(float)
# src2 = cv.imread('picture.jpg')
# im11=(cv.cvtColor(src2,cv.COLOR_BGR2GRAY).astype(float)+20)

src = cv.imread('big1.JPG')
im10=cv.cvtColor(src,cv.COLOR_BGR2GRAY).astype(float)
src2 = cv.imread('big2.JPG')
im11=(cv.cvtColor(src2,cv.COLOR_BGR2GRAY).astype(float))

# src3 = cv.imread('picture6.jpg')
# im12=cv.cvtColor(src3,cv.COLOR_BGR2GRAY).astype(float)
# harris(im10,0.1,0.1,0.1,110)
# harris(im10,0.5,0.5,0.1,110)
# harris(im10,1.0,1.0,0.1,110)
# harris(im10,2.0,2.0,0.1,110)
# harris(im10,5.0,5.0,0.1,110)
# harris(im10,10.0,10.0,0.1,110)

# harris(im10,5.0,5.0,0.21,110)
# harris(im10,5.0,5.0,0.22,110)
# harris(im10,5.0,5.0,0.25,110)

harris(im10,5,5,0.05,110)
harris(im11,5,5,0.05,110)
```

```
158. #harris(im12,5,5,0.05,110)
159.
160.
161.
162. harris(im10,5,5,0.1,110)
163. harris(im10,5,5,0.2,110)
164. harris(im10,5.0,5.0,0.5,110)
165. harris(im10,5.0,5.0,0.8,110)
166.
167.
168.
169. #src = cv.imread('picture.jpg')
170. # im11=(cv.cvtColor(src,cv.COLOR_BGR2GRAY).astype(float)+10)
171. # harris(im11,1.0,1.0,0.04,110)
```