第 7 章 几何变换与图像特征

7.1 几何变换

图像的几何变换最常见的有两个类别，一类称为仿射变换(Affine Transform)、另外一类称为单应性变换(Homography)。

仿射变换

我们最常见的的图像变换多数都是仿射变换，主要包括图像的平移、放缩、旋转、错切操作。仿射变换常常是上述几种变换组合在一起，所以仿射变换相对来说比较复杂，但是既然仿射变换是组合变换，自然也可以拆分为不同的简单变换。基本的变换操作可以表示如下：

$$\begin{vmatrix} x_2 \\ y_2 \end{vmatrix} = A \times \begin{vmatrix} x_1 \\ y_1 \end{vmatrix} + B$$

当平移操作时：

$$A = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix},\ B = \begin{vmatrix} b_1 \\ b_2 \end{vmatrix}$$

当旋转操作时：

$$A = \begin{vmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{vmatrix},\ B = \begin{vmatrix} 0 \\ 0 \end{vmatrix}$$

当放缩操作时：

$$A = \begin{vmatrix} a_{11} & 0 \\ 0 & a_{22} \end{vmatrix},\ B = \begin{vmatrix} 0 \\ 0 \end{vmatrix}$$

当错切操作时：

$$A = \begin{vmatrix} 1 & k \\ 0 & 1 \end{vmatrix},\ B = \begin{vmatrix} 0 \\ 0 \end{vmatrix}$$

在 OpenCV 中为了方便，把 A 跟 B 合并位一个 2x3 的矩阵，描述如下：

$A = \begin{vmatrix} a & b & t_x \\ c & d & t_y \end{vmatrix}$ 其中最后一列参数是表示平移

假设原图上的点(x, y)使用上述的仿射变换得到映射点 (x_t, y_t)，使用下面的公式表示：

$$\begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

当平移时旋转参数跟错切参数为 0，放缩参数为 1，平移参数为非零参数。在欧式变换中，旋转跟平移参数不为零、放缩参数为 1、错切参数为 0。

在 OpenCV 中想调用仿射变换实现图像的变换，调用的函数功能为 warpAffine，如下所示：

```
void cv::warpAffine(
    InputArray  src,
    OutputArray dst,
    InputArray  M,
    Size    dsize,
    int     flags = INTER_LINEAR,
    int     borderMode = BORDER_CONSTANT,
    const Scalar &  borderValue = Scalar()
)
```

参数解释如下
src 输入图像
dst 输出图像
M 仿射变换矩阵，2x3 大小的矩阵
dsize 输出图像的大小
flags 变换过程中的图像插值方式
borderMode 对边缘的处理方式
borderValue 当边缘选择为常量边缘的时候，该值起作用，否则默认为 0

7.1.1 仿射变换演示

- 图像平移

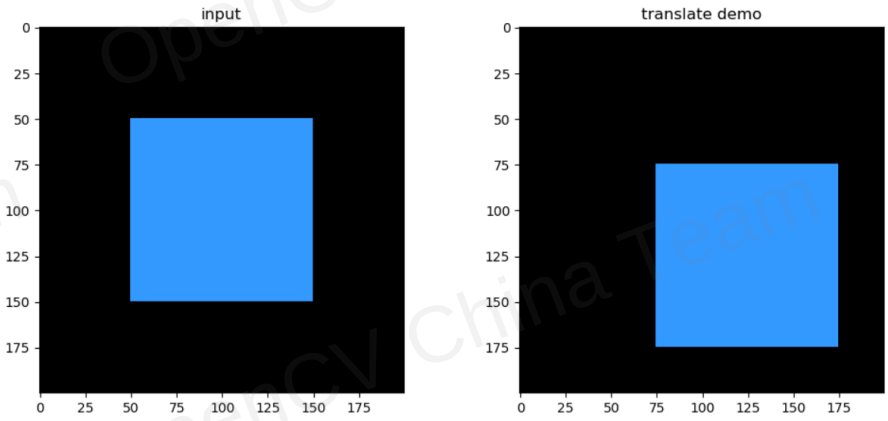


图 7. 1-1

读入一张图像，平移到(25, 25)，代码实现如下：

```
Mat image = imread("D:/blue-square.png");

// 平移到(25, 25)
float warpMatValues[] = { 1.0, 0.0, 25.0, 0.0, 1.0, 25.0 };
Mat warpMat = Mat(2, 3, CV_32F, warpMatValues);

// Warp Image
Mat result;
Size outDim = image.size();
warpAffine(image, result, warpMat, outDim);
```

- 图像放缩，宽度放大两倍

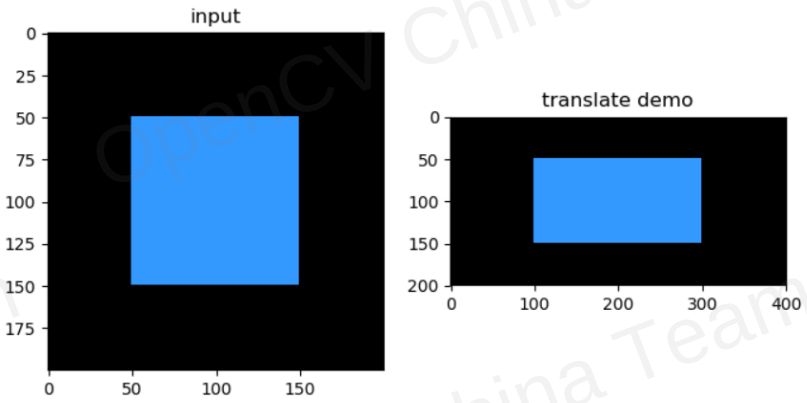


图 7.1-2

```
// 宽度放大两倍
float warpMatValues2[] = { 2.0, 0.0, 0.0, 0.0, 1.0, 0.0 };
warpMat = Mat(2, 3, CV_32F, warpMatValues2);
warpAffine(image, result, warpMat, Size(outDim.width * 2, outDim.height));
```

对图像宽高同时放大两倍，需要修改变换矩阵代码如下：

```
float warpMatValues2[] = { 2.0, 0.0, 0.0, 0.0, 2.0, 0.0 };
warpMat = Mat(2, 3, CV_32F, warpMatValues2);
```

- 基于左上角坐标原点实现旋转图像

根据角度参数θ旋转一个点 P(x,y)，旋转之后得到坐标为$P(x',y')$，相关的转换公式为：

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

所以仿射变换的旋转矩阵对应为：

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \end{bmatrix}$$

相关代码实现如下：

```
// 旋转图像
float angleInRadians = 30;
angleInRadians = 30 * 3.14 / 180.0;

float cosTheta = cos(angleInRadians);
float sinTheta = sin(angleInRadians);

// 定义旋转矩阵
float warpMatValues4[] = { cosTheta, sinTheta, 0.0, -sinTheta, cosTheta, 0.0 };
warpMat = Mat(2, 3, CV_32F, warpMatValues4);

// 仿射变换
warpAffine(image, result, warpMat, outDim);
```

运行结果如下：

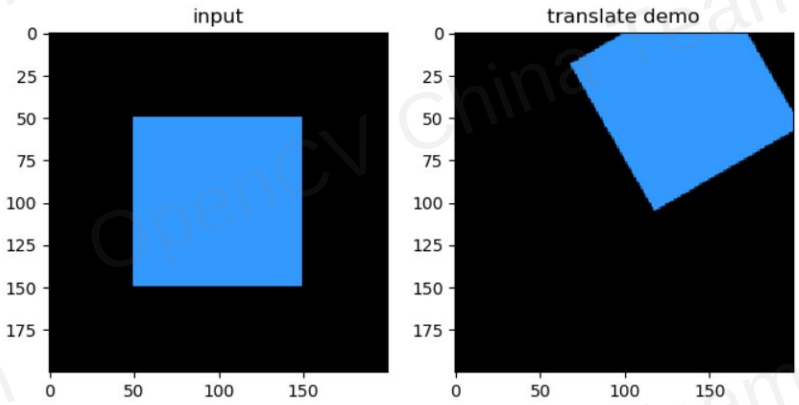


图 7.1-3

- 基于中心点实现图像旋转

旋转一个点 P(x, y)基于给定的中心点 $P(x_c, y_c)$，这个就是基于中心的旋转。用仿射变换矩阵可以表示为：

$$\begin{bmatrix} \cos\theta & \sin\theta & x_c(1-\cos\theta) - y_c\sin\theta \\ -\sin\theta & \cos\theta & x_c\sin\theta + y_c(1-\cos\theta) \end{bmatrix}$$

对上述矩阵的快速推导可以分为三步

1. 把原来坐标移动到 (x_c, y_c)，可以通过 (x, y) 减去 (x_c, y_c) 得到

2. 基于变换之后的点 $(x - x_c, y - y_c)$ 实现旋转

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x - x_c \\ y - y_c \end{bmatrix}$$

3. 重新变换回到 (x_c, y_c)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x - x_c \\ y - y_c \end{bmatrix} + \begin{bmatrix} x_c \\ y_c \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_c(1-\cos\theta) - y_c\sin\theta \\ x\sin\theta + y_c(1-\cos\theta) \end{bmatrix}$$

代码实现如下：

```
float centerX = image.size().width/2;
float centerY = image.size().height/2;

float tx = (1-cosTheta) * centerX - sinTheta * centerY;
float ty = sinTheta * centerX + (1-cosTheta) * centerY;

float warpMatValues5[] = { cosTheta, sinTheta, tx, -sinTheta, cosTheta, ty };
warpMat = Mat(2,3,CV_32F,warpMatValues5);

// 基于中心的旋转
warpAffine(image, result, warpMat, outDim);
```

运行结果如下：

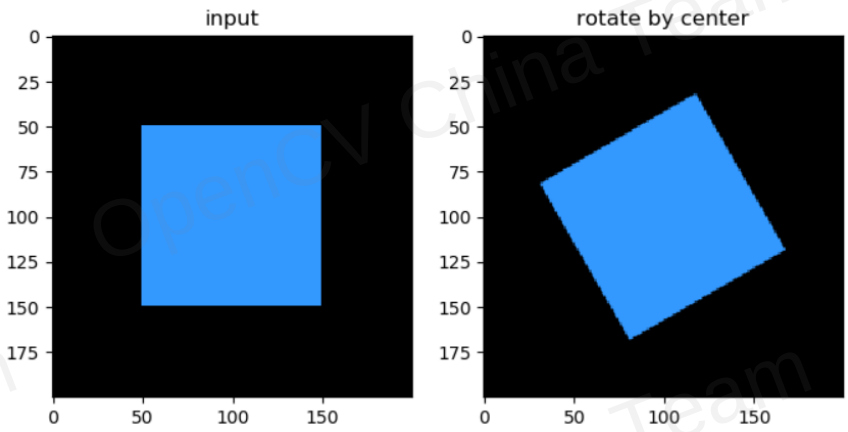


图 7.1-4

- 图像错切

X 方向的错切，首先定义仿射变换矩阵，定义为：

```
// 错切变换
float shearAmount = 0.1;
float warpMatValues6[] = { 1, shearAmount, 0, 0, 1.0, 0 };
warpMat = Mat(2, 3, CV_32F, warpMatValues6);

// Warp image
warpAffine(image, result, warpMat, outDim);
```

运行结果如下：

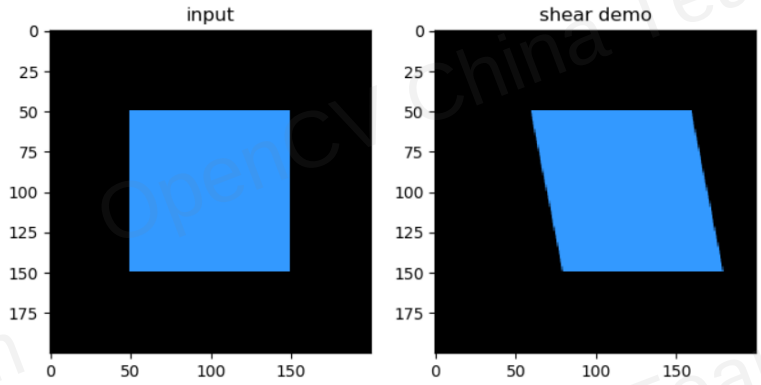


图 7.1-5

- 复杂变换

现在我们可以实现一个多个变换操作包括旋转、放缩、错切、平移。我们可以一个一个来分别操作完成，但是一个更有效的方法是合并为一个仿射变换矩阵，先把旋转、放缩、错切矩阵相乘，然后加上平移坐标参数即可。

我们来做一个实验，首先对图像放大 1.1 倍、错切-0.1、旋转 10°、在 X 方向平移 10 个像素。相关的代码实现如下：

```
float scaleAmount = 1.1;
float warpMatValues7[] = { scaleAmount, 0.0, 0, scaleAmount };
Mat scaleMat = Mat(2, 2, CV_32F, warpMatValues7);

// Shear
shearAmount = -0.1;
float warpMatValues8[] = { 1, shearAmount, 0, 1.0 };
Mat shearMat = Mat(2, 2, CV_32F, warpMatValues8);

// Rotate by 10 degrees about (0,0)
angleInRadians = 10.0 * 3.14 / 180.0;
cosTheta = cos(angleInRadians);
sinTheta = sin(angleInRadians);

float warpMatValues9[] = { cosTheta, sinTheta, -sinTheta, cosTheta };
Mat rotMat = Mat(2, 2, CV_32F, warpMatValues9);

float warpMatValues10[] = { 10,0 };
Mat translateVector = Mat(2, 1, CV_32F, warpMatValues10);

// First scale is applied, followed by shear, followed by rotation.
Mat scaleShearRotate = rotMat * shearMat * scaleMat;

hconcat(scaleShearRotate, translateVector, warpMat);

float warpMatValues11[] = { 50,50,50,149,149,50,149,149 };
hconcat(translateVector, translateVector, translateVector);
hconcat(translateVector, translateVector, translateVector);
Mat outPts = scaleShearRotate * Mat(4, 2, CV_32F, warpMatValues11).t() + translateVector;

// Warp image
warpAffine(image, result, warpMat, outDim);
```

运行结果如下：

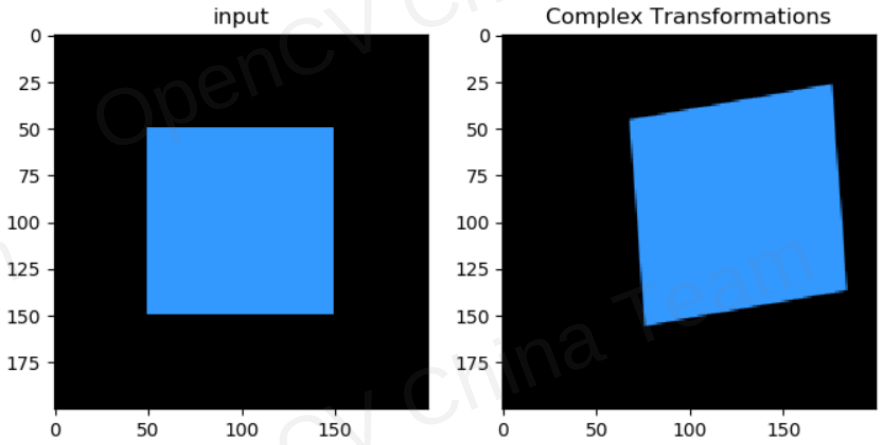


图 7.1-6

- 使用三点相关估算复杂变换

上面的复杂变换可以看出来它们计算过程非常复杂，有没有一个简单的方法可以做到，答案是肯定的，我们知道仿射变换有 6 个自由度

- 平移两个参数(tx, ty)
- 放缩两个参数(sx, sy)
- 错切一个参数
- 旋转角度一个参数

这个意思是说如果我们知道两个图像之间是仿射变换关系，那么我们至少需要原图与目标图像上各三个点，这样我们就可以恢复它们之间的仿射变换。

下面我们考虑原图上三个角点坐标分别位（50，50）、（50、149）、（149，50），在目标图像中对应的三个点坐标为（68，45）、（76，155）、（176，27），我们可以使用 estimateAffine2D 函数来估计它们之间的仿射变换矩阵 H。然后根据 H 实现仿射变换，代码实现如下：

```
vector<Point2f> srcPoints{ Point2f(50,50),Point2f(50,149),Point2f(149,50) };
vector<Point2f> dstPoints{ Point2f(68,45),Point2f(76,155),Point2f(176,27) };
Mat estimatedMat = estimateAffine2D(srcPoints, dstPoints);
warpAffine(image, result, estimatedMat, outDim);
```

运行结果如下：

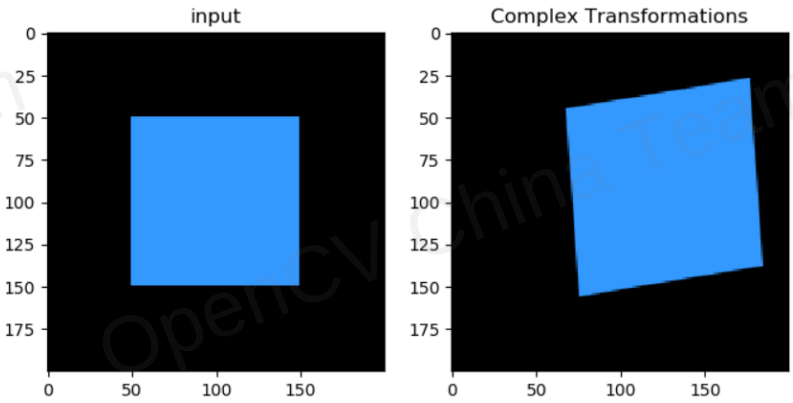


图 7.1-7

仿射变换的局限性

考虑仿射变换从正方形到梯形，原图与梯形图像图示如下：

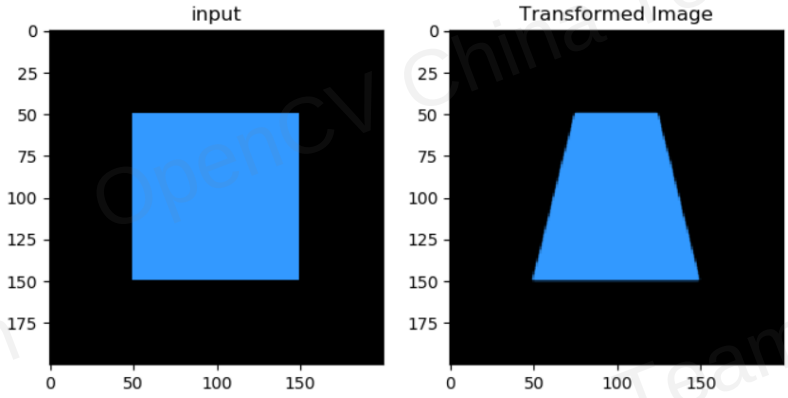


图 7.1-8

绘制梯形的相关代码如下：

```
// Transformed image
Mat imgT(200, 200, CV_8UC3, Scalar(0, 0, 0));
Point dstPoints2[] = { Point(75,50),Point(50,149),Point(149,149),Point(124,50) };
fillConvexPoly(imgT, dstPoints2, 4, Scalar(255, 153, 50), LINE_AA);
```

现在我们知道了正方形的四个点，我们尝试把正方形仿射变换到梯形，从正方形的四个点变换到梯形的四个点(75, 50)、(50, 149)、(149, 149)、(124, 50)。我们知道一个仿射变换不能实现从正方形到梯形的转换。但是理论上我们还是可以通过估算函数 estimateAffine2D 得到一个仿射变换矩阵 H，代码如下：

```
srcPoints = vector<Point2f> { Point2f(50,50),Point2f(50,149),Point2f(149,149),Point2f(149,50) };
dstPoints = vector<Point2f> { Point2f(75,50),Point2f(50,149),Point2f(149,149),Point2f(124,50) };
estimatedMat = estimateAffine2D(srcPoints, dstPoints);
// Warp image
Mat imA;
warpAffine(image, imA, estimatedMat, outDim);
```

运行结果如下

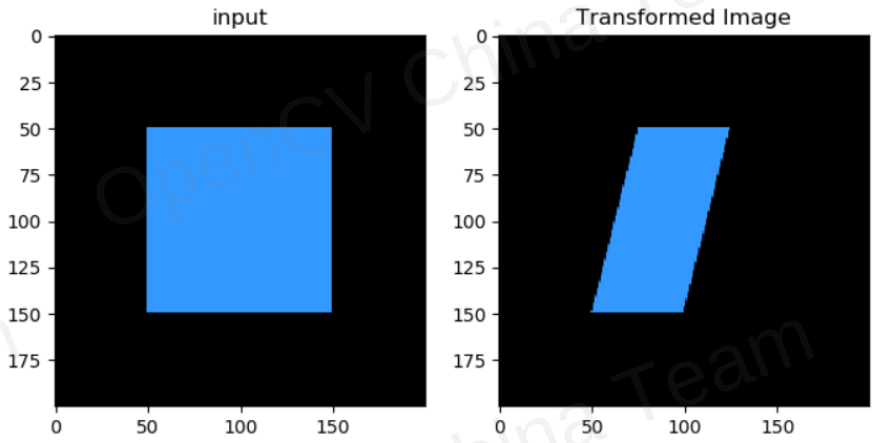


图 7.1-9

从上面可以看出来，仿射变换无法完成从正方形到梯形的转换，这个时候我们需要另外一种变换来完成，它就是单应性变换。

7.1.2 单应性变换

什么是单应性变换

考虑下面两张图像，如下图所示（图 7.1.2-1），红色点表示两张图像上相同位置标记，在计算机视觉术语中我们称它们为相关点。图像中显示的四个相关点使用四种不同的颜色标记分别为红、绿、黄、橙。

一个单应性变换（3x3 的矩阵）是从一个图像中相关点映射到另外一张图像上的相关点，对单应性变换矩阵 H 可以表示如下：

$$H = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix}$$

考虑第一张图像上相关点 (x_1, y_1) 到第二张图像上相关点 (x_2, y_2)。它们之间基于单应性矩阵的映射关系可以表示为如下：

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = H \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

基于单应性矩阵的图像对齐

当所有相关点在同一个平面时候上面的等式成立。换句话说就是把第一张图像中书本使用单应性变换得到第二张中图书实现对齐：

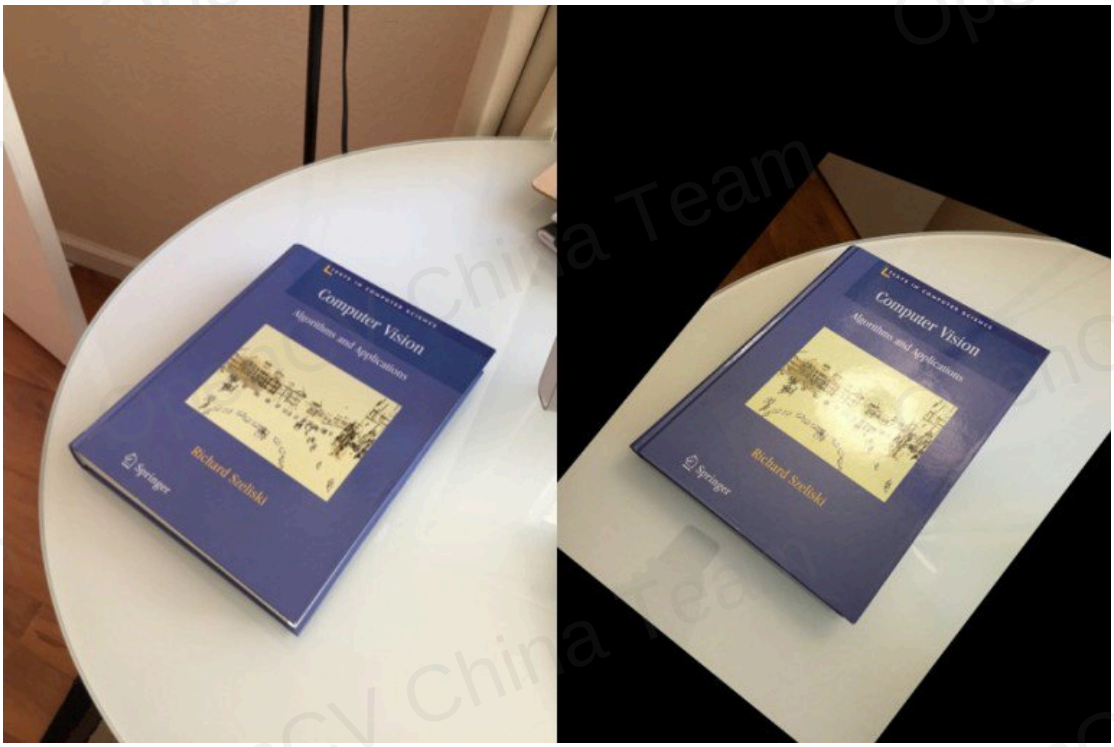


图 7.1.2-2

但是当这些对应点不再同一个平面会如何？它们不会被一个单应性矩阵对齐。如果图像中有两个平面的话，就应该有两个单应性矩阵，每个对应一个。

计算单应性矩阵

计算两个图像之间的单应性矩阵，你至少需在两个图像之间需要四个相关点，如果有超过四个相关点，那就更好了。OpenCV 有一个很稳定的估算方法拟合所有的相关点，这些相关点通过 SURF/SIFT 特征匹配自动发现，但是本节我们还是通过手动选择一些相关点。

估算单应性矩阵

```
Mat cv::findHomography(
    InputArray          srcPoints,
    InputArray          dstPoints,
    OutputArray         mask,
    int         method = 0,
    double      ransacReprojThreshold = 3
)
```

使用单应性矩阵实现原图到目标图像的变换，函数功能

```
void cv::warpPerspective(
    InputArray src,
    OutputArray dst,
    InputArray M,
    Size dsize,
    int flags = INTER_LINEAR,
    int borderMode = BORDER_CONSTANT,
    const Scalar &      borderValue = Scalar()
)
```

参数解释

Src 表示输入图像

Dst 表示输出图像

M 表示单应性矩阵

Dsize 表示变换之后图像大小

Flags 表示变换时图像的插值方式

borderMode 表示边缘处理方式

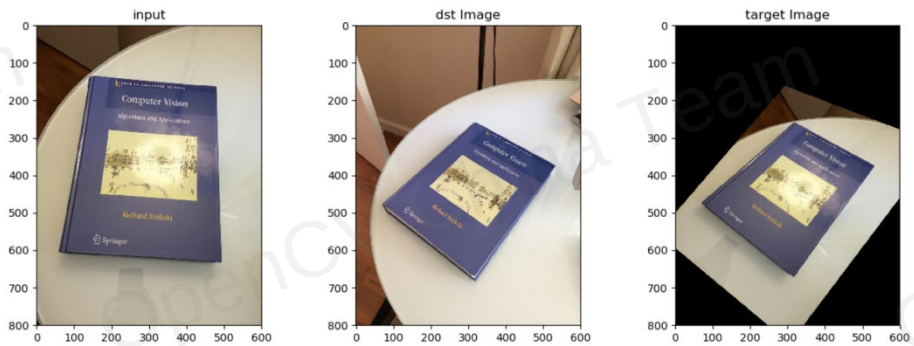
Scalar 当 BORDER_CONSTANTS 的时候

单应性矩阵变换演示

代码实现部分演示了如何从一张图像基于四点计算单应性矩阵，实现从一张图像到另外一张图像的变换。这里四个点是图中书的四个角点，使用单应性矩阵估算变换矩阵 H，然后基于 H 实现变换。代码实现如下：

```
// Read source image.
Mat im_src = imread("D:/book2.jpg");
// Four corners of the book in source image
vector<Point2f> pts_src{ Point2f(141, 131), Point2f(480, 159), Point2f(493, 630),Point2f(64, 601) };

// Read destination image.
Mat im_dst = imread("D:/book1.jpg");
// Four corners of the book in destination image.
vector<Point2f> pts_dst{ Point2f(318, 256), Point2f(534, 372),Point2f(316, 670),Point2f(73, 473) };

// Calculate Homography
Mat h = findHomography(pts_src, pts_dst);

// Warp source image to destination based on homography
Mat im_out;
warpPerspective(im_src, im_out, h, im_dst.size());
```

运行结果如下（**图 7.1.2-3**）：



单应性矩阵应用

单应性矩阵最重要的应用之一就是图像对齐拼接跟全景图生成，可以看下面的一些应用例子。

1. 全景图-单应性应用

通过前面的知识学习，我们知道单应性矩阵可以帮助我们实现一张图像到另外一张图的交换。但是这里还有个重要的告诫就是图像必须在同一个平面之上，而且必须被对齐正确。并不是说任意两张图像都可以通过单应性矩阵实现正确的对齐跟拼接，全景图拼接基本的方法是对齐，使用单应性矩阵自动拼接，实现无缝全景图展示。关于全景图拼接的内容我们在本章后续会继续。

2. 基于单应性的透视校正

使用单应性实现透视矩阵，主要步骤如下：

- 首先写一个用户接口实现对原图的四个点的选取
- 根据宽高选择输出图像大小，然后对输出目标图像选择四个点
- 计算单应性矩阵 H
- 应用单应性矩阵实现从原图到目标图像的透视校正

3. 虚拟广告牌

在许多视频广告中，虚拟广告牌。例如足球或者篮球比赛，放置广告牌都可以虚拟的改变，广告主可以根据需求调整广告内容，在这些应用中，我们可以检测广告牌的四个角点，作为目标点，然后对视频中一帧设置四个目标输出点作为相关点，根据广告牌跟视频帧对应位置相关点计算单应性矩阵 H，然后基于单应性矩阵完成变换，实现虚拟广告牌。广告牌图示如下：

视频中场景图像如下：

最终替换左侧的广告牌效果如下：

7.2 图像特征

上一节中我们提到单应性矩阵实现不同图像中对象对齐，如果要实现这样的操作就要从对象总选取合适的相关点，这些相关点又被称为图像特征的关键点，更进一步这些关键点生成描述子就可以发现图像对象匹配的相关点。而寻找图像关键点跟描述子的过程被称为图像的特征提取。常见的图像特征提取方法有 SIFT、SUFR、ORB 等，其中 SIFT 特征提取方法在 2004 提出，它同时具备迁移、尺度、旋转不变性，但是速度慢，而且必须要专利授权；比 SIFT 更快的一个特征提取方法是 SURF 特征提取，同样它也需要专利授权；从 OpenCV3 开始 SIFT 特征跟 SURF 特征被从发布版本中移除，在扩展模块中作为非免费功能由开发者自行决定是否需要在商业场景中使用。2011 年 ORB 算法被提出，它是速度比 SIFT 跟 SURF 都快，同时又免费商业应用的特征提取算法。

ORB 特征提取算法

ORB 主要是在快速关键点检测算法跟 BRIEF 描述子算法上改进而成的，它的关键点检测主要通过 FAST 算法发现关键点，然后通过 Harris 角点检测与金字塔提取多尺度特征。对于特征描述子，ORB 使用 BRIEF 描述子，但是 BRIEF 描述子在匹配时候本身不具备旋转不变性，稳定性不高，因此 ORB 通过改进 BRIEF 描述子对点对旋转生成多个查找表实现对特征描述子的计算。在描述子匹配阶段，采用多探针的 LSH 方法来替代传统 LSH，论文上说 ORB 描述子的

性能比 SURF 好，而且计算量要求低，可以用在低功耗设备上实现全景图拼接。

在 OpenCV 中使用 ORB

首先是要创建一个 ORB 特征检测器，OpenCV 中的函数功能为

```
static Ptr<ORB> cv::ORB::create (
    int     nfeatures = 500,
    float   scaleFactor = 1.2f,
    int     nlevels = 8,
    int     edgeThreshold = 31,
    int     firstLevel = 0,
    int     WTA_K = 2,
    ORB::ScoreType  scoreType = ORB::HARRIS_SCORE,
    int     patchSize = 31,
    int     fastThreshold = 20
)
```

该函数参数比较多，建议直接使用默认参数即可，其中 nfeatures=500 表示创建的 ORB 特征检测器支持 500 点检测。ORB 特征检测属于 feature2D 模块，它几个会经常用到的重要函数，分别是 orb.detect()、orb.compute()、orb.detectAndCompute()。这些函数功能及其参数解释分别为：

- **检测关键点**

```
virtual void cv::Feature2D::detect(
    InputArray  image,
    std::vector< KeyPoint > &   keypoints,
    InputArray  mask = noArray()
)
```

image 表示输入图像
keypoints 表示检测到的关键点
mask 表示寻找关键点的区域，默认是全部

- **计算描述子**

```
virtual void cv::Feature2D::compute(
    InputArray  image,
    std::vector< KeyPoint > &   keypoints,
    OutputArray     descriptors
)
```

Image 表示输入图像
Keypoints 输入的关键点，不能被计算描述子的关键点将会自动删除
Descriptors 得到描述子信息

- **检测关键点与计算描述子**

```
virtual void cv::Feature2D::detectAndCompute(
    InputArray  image,
    InputArray  mask,
    std::vector< KeyPoint > &   keypoints,
    OutputArray     descriptors,
    bool    useProvidedKeypoints = false
)
```

上面的函数检测关键点与计算描述子一起完成输出。

- 绘制关键点

```
void cv::drawKeypoints  (
    InputArray  image,
    const std::vector< KeyPoint > & keypoints,
    InputOutputArray    outImage,
    const Scalar &  color = Scalar::all(-1),
    DrawMatchesFlags    flags = DrawMatchesFlags::DEFAULT
)
```

image 表示输入图像

keypoints 表示检测到的关键点

outImage 绘制完关键点的输出图像

color 关键点的颜色

flags 绘制关键点的方法标记，在 DrawMatchesFlags 中声明

使用 ORB 特征检测，实现关键点检测，代码实现如下：

```
Mat img = imread("D:/book.jpeg");
// Convert to grayscale
Mat imgGray;
cvtColor(img, imgGray, COLOR_BGR2GRAY);

// Initiate ORB detector
Ptr<ORB> orb = ORB::create();

// find the keypoints with ORB
vector<KeyPoint> kp;
orb->detect(imgGray, kp, Mat());

// compute the descriptors with ORB
Mat des;
orb->compute(imgGray, kp, des);

// draw only keypoints location,not size and orientation
Mat img2;
drawKeypoints(img, kp, img2, Scalar(0, 255, 0), DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
```

运行结果如下：

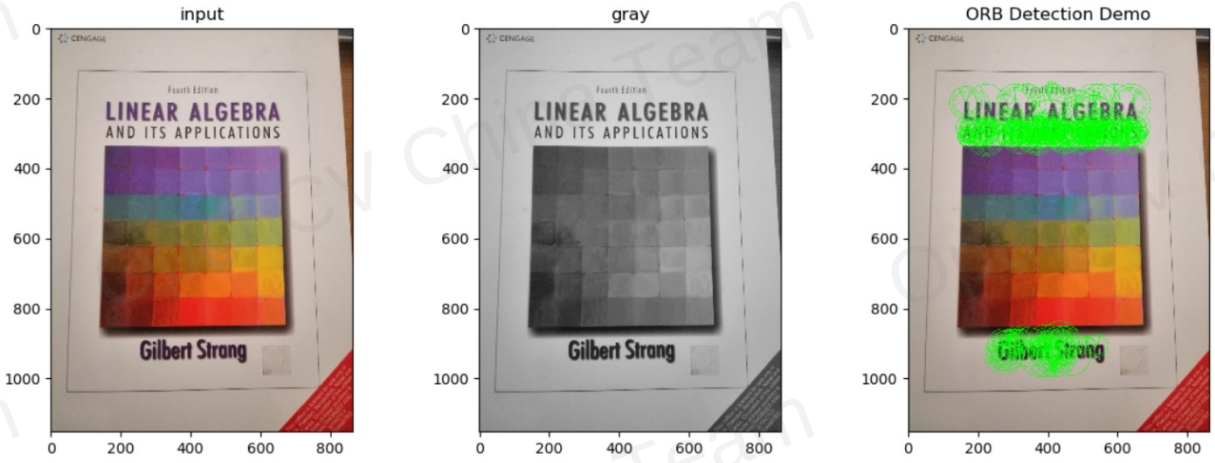


图 7.2-1

7.3 特征匹配

当使用 ORB 特征提取算法提取到图像特征之后，你就可以通过特征匹配实现不同的计算机视觉任务，这些特征就好像图像的 DNA 一样。在本节中，我们将会明白特征匹配是如何通过不同的算法实现一张图像到另外一张图像匹配的。在 OpenCV 中有两种特征匹配方法，它们是

- 暴力匹配
- FLANN 匹配

它们在代码实现上都继承了 OpenCV 超类 Descriptor Matcher 接口

这里我们看一个例子，通过计算图像的 ORB 描述子，实现两张图像之间的特征匹配，首先需要加载图像，计算 ORB 描述子，然后使用不同的匹配方法。

ORB 描述子计算

首先计算图像的 ORB 特征描述子，代码实现如下：

```
01 Mat img1 = imread("D:/book.jpeg", IMREAD_GRAYSCALE);
02 Mat img2 = imread("D:/book_scene.jpeg", IMREAD_GRAYSCALE);
03
04 // Initiate ORB detector
05 Ptr<ORB> orb = ORB::create();
06
07 // find the keypoints and descriptors with ORB
08 vector<KeyPoint> kp1, kp2;
09 Mat des1, des2;
10
11 orb->detectAndCompute(img1, Mat(), kp1, des1);
12 orb->detectAndCompute(img2, Mat(), kp2, des2);
```

暴力匹配

暴力匹配很简单，就是用第一个图像的描述子片段，去第二个图像描述子中进行全局匹配搜索，计算最小距离，最后返回最相似的。OpenCV 中暴力匹配在使用前，首先需要创建暴力匹配器，通过 cv::BFMatcher()创建，函数功能如下：

```
01 static Ptr<BFMatcher> cv::BFMatcher::create (
02     int     normType = NORM_L2,
03     bool    crossCheck = false
04 )
```

该方法有两个参数

normType 可以声明不同的距离度量方式，默认的 NORM_L2 方式对 SIFT、SURF 效果比较好，对于基于二进制字符串匹配的描述子如 ORB、BRIEF、BRISK 等，NORM_HAMMING 效果比较好。

crossCheck 默认是 false，如果为 true 则表示则返回(i,j)值，表示 A 图中的第 i 个特征描述子，对应 B 图中第 j 个特征描述子，反之亦然。这个就表示两个特征描述子在对应 A 与 B 两个特征描述子合集中是相互匹配的，保持了一致性。

创建了暴力匹配器之后，就可以调用函数方法实现匹配，暴力匹配有两个重要的匹配方法，分别为：

- BFMatcher.match() 返回最好匹配结果

- BFMatcher.knnMatch()返回最好的 K 个匹配结果，其中 K 是用户可以声明的参数

完成了匹配之后，就需要绘制匹配结果，类似绘制关键点的函数 drawKeypoints()一样，使用 drawMatches()就可以实现匹配结果绘制。它从通过从第一个图像绘制线到第二个图像显示最好的匹配结果。此外还有个绘制函数 cv::drawMatchesKnn 意思绘制全部最好的 K 个匹配线，架设 k=2 表示每个关键点绘制两条匹配线。

这里看一个简单的例子，创建暴力匹配，使用汉明距离作为距离度量，为了获取更好的结果，把 crossCheck 选项开启。使用 BFMatcher.match()方法去获取最好的匹配，升序排列，然后绘制前 10 个最好的匹配（当然你也可以绘制更多）。代码实现如下：

```
01 // create BFMatcher object
02 BFMatcher bf(NORM_HAMMING, true);
03
04 vector <DMatch> matches;
05 bf.match(des1, des2, matches, Mat());
06
07 // Sort them in the order of their distance
08 sort(matches.begin(), matches.end());
09
10 // Draw first 10 matches.
11 matches = vector<DMatch>(matches.begin(), matches.begin() + 10);
12 Mat img3;
13 vector<char> match_mask(matches.size(), 1);
14 drawMatches(img1, kp1, img2, kp2, matches, img3, Scalar::all(-1),
15             Scalar::all(-1), match_mask, DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
```

运行结果如下：

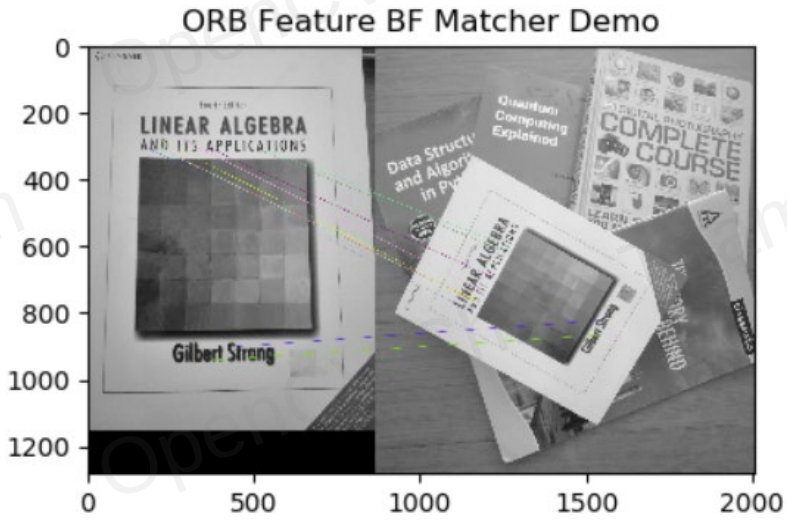


图 7.3-1

基于 FLANN 的匹配

FLANN 是 **F**ast **L**ibrary for **A**pproximate **N**earest **N**eighbors 首字母的缩写，它对高维特征优化了快速最近邻搜索算法，速度比 BFMatcher 快。

算法参数声明

对 FLANN 匹配方法来说，需要声明使用算法的方法参数，实现的算法有如下：

Algorithms	ID
FLANN_INDEX_LINEAR	0
FLANN_INDEX_KDTREE	1
FLANN_INDEX_KMEANS	2
FLANN_INDEX_COMPOSITE	3
FLANN_INDEX_KDTREE_SINGLE	4
FLANN_INDEX_HIERARCHICAL	5
FLANN_INDEX_LSH	6
FLANN_INDEX_SAVED	254
FLANN_INDEX_AUTOTUNED	255

参数有

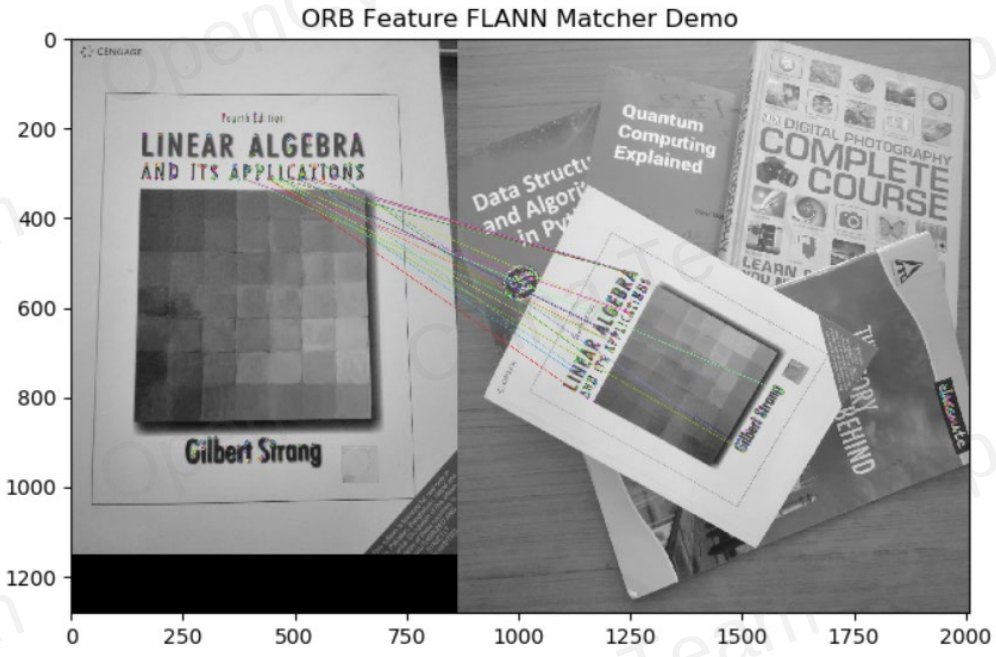
flann::IndexParams 声明使用的算法

flann::SearchParams 声明指定的 index 在树中递归遍历的次数，值越高精度越好，但是耗时更多。

基于 ORB 特征描述子的 FLANN 匹配代码实现如下：

```
01 FlannBasedMatcher matcher(new flann::LshIndexParams(6, 12, 1), new flann::SearchParams(50));
02 std::vector< std::vector<DMatch> > matches_vec;
03 matcher.knnMatch(des1, des2, matches_vec, 2);
04 matches_vec = vector< vector<DMatch> >(matches_vec.begin(), matches_vec.begin() + 10);
05 drawMatches(img1, kp1, img2, kp2, matches_vec, img3);
```

运行结果如下：



随机采样一致性(RANSAC)

最小二乘方法在描述子匹配输出的点对质量很好，理想情况下是图像没有噪声污染与像素迁移与光线恒定，但是实际情况下图像特别容易受到光线、噪声导致像素迁移，从而产生额外的多余描述子匹配，这些点对可以分为 outlier 跟 inlier 两类，基于 RANSAC（Random Sample

Consensus）可以很好的过滤掉 outlier 点对，使用合法的点对得到最终的变换矩阵 H。RANSAC 算法基本思想是，它会从给定的数据中随机选取一部分进行模型参数计算，然后使用全部点对进行计算结果评价，不断迭代，直到选取的数据计算出来的错误是最小，比如低于 0.5% 即可，完整的算法流程步骤如下：

1. 选择求解模型要求的最少要求的随机点对
2. 根据选择随机点对求解/拟合模型得到参数
3. 根据模型参数，对所有点对做评估，分为 outlier 跟 inlier
4. 如果所有 inlier 的数目超过预定义的阈值，则使用所有 inlier 重新评估模型参数，停止迭代
5. 如果不符合条件则继续 1~4 循环。

通常迭代次数 N 会选择一个比较高的值，OpenCV 中默认迭代次数为 200，确保有一个随机选择点对不会有 outlier 数据，

7.4 应用-基于特征的图像对齐

基于单应性矩阵实现文档对齐

在本节中，我们将使用 OpenCV 实现基于特征的图像对齐。

本节将通过一个简单的例子来演示图像对齐，首先使用手机拍一张文档照片作为模板。这里使用的技术是基于特征的匹配，首先检测一张图像上稀疏特征点然后再用它去匹配另外一张图像中的特征点，然后基于这些匹配特征点实现图像校正变换。

什么是图像对齐/配准

在许多应用中，我们有相同场景的两张图像，但是它们没有对齐。换句话说，一张图像上得到的特征(角点)不能准确对应到另外一张图像上，因为拍摄的时候总会有变化，得到完全一致的图像很难。图像对齐就是让这两张图像实现对一场景下目标很好的对齐，实现完整一致性。如下所示

图 7.4-1

上图左侧是下载模板图像，中间是手机拍摄填写好的图像，右侧是使用左侧图像对齐以后得到的结果。

图像对齐应用

图像对齐有大量的应用场景，在许多文档处理中，第一步就是用模板对齐扫描的照片文档。例如，你想写一个自动表单读取器，第一步就是通过模板对表单实现自动对齐，然后读取指定的文档内容，实现值域读取。

在多媒体应用中，两张图像可能来自一组多个扫描仪，通过本节提到的技术实现图像配准，此外最好玩的应用可能是创建全景图，这种情况下，两张图像不是一个平面上，它是一个 3D 场景。简单来说，三维图像对齐需要深度信息，但是这里通过旋转移动相机镜头位置，我们可以使用图像对齐技术实现两张图像的全景图对齐。

发现单应性矩阵

图像对齐的核心是 3x3 单应性矩阵，关于单应性矩阵这里就不再赘述。我们重点关注如何发现单应性矩阵。看到本章第一节内容，我们知道 4 个或者更多的相关点对，在 OpenCV 中我们就可以计算出两张图像之间的单应性变换矩阵 H，函数为 findHomography。函数功能如下：

```
Mat cv::findHomography(
    InputArray  srcPoints,
    InputArray  dstPoints,
    int      method = 0,
    double  ransacReprojThreshold = 3,
    OutputArray      mask = noArray(),
    const int   maxIters = 2000,
    const double     confidence = 0.995
)
```

其中

srcPoints 表示原来平面的坐标点

dstPoints 表示目标平面的坐标点

method 表示计算单应性矩阵方法，默认 0 表示最小二乘，

- RANSAC 表示随机采样一致性
- LMEDS 表示最小中值
- RHO 表示改进的随机采样一致性

ransacReprojThreshold 表示最大允许的重映射错误，只有 RANSAC 跟 RHO 时该参数才有效

mask 表示可以选择的点，默认全部

maxIters 表示最大迭代次数，当 RANSAC 有效

confidence 表示置信度，值在 0~1 之间

如何自动发现相关点

在许多计算机视觉应用中，我们需要鉴别那些是图像中稳定的感兴趣点，这些点通常被称为关键点或者特征点。OpenCV 已经实现了几个不同的特征点检测算法包括 SIFT、SURF、ORB 等。

这里我们采样 ORB 特征检测器，因为它的开发者是我前实验室同事 Vincent Rabaud！开玩笑！我们使用 ORB 是因为 SIFT、SURF 是由专利费用的，实际场景使用需要付费，ORB 速度快，精准而且免费。

ORB 关键点如下图所示的圆圈

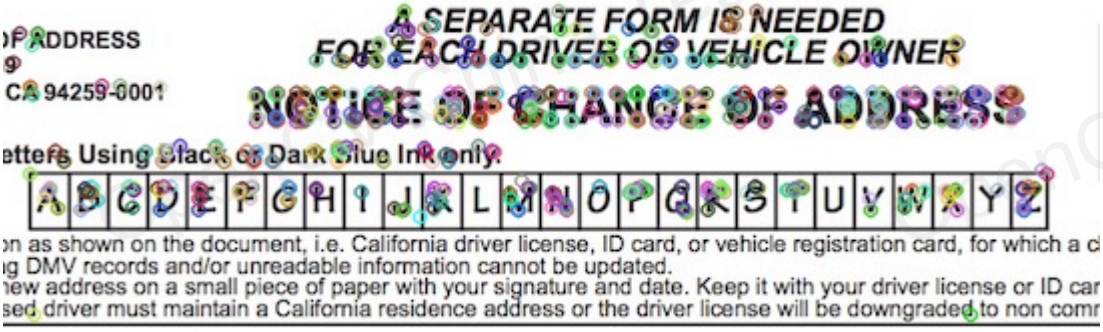


图 7.4-2

ORB 是 Oriented Fast and Rotated BRIEF 算法缩写，下面让我们看看什么是 Fast 与 BRIEF，一个特征点检测器有如下两个部分

- 特征点定位，定位要寻找图像中那些对迁移、放缩、形变相对比较稳定的角点坐标，ORB 中定位使用的算法角 FAST
- 特征描述子，特征点定位帮助我们找到了那些感兴趣的特征关键点，ORB 特征检测的第二步是对每个特征点信息进行编码，描述子就是用来描述每个特征点信息的，本质上描述子就是一组数字，理论上不同两张图像中，相同位置的对象特征点描述子应该完全一致，但是实际上受到各种因素干扰，它们会很相似，很难做到完全相同。ORB 中用来完成描述子生成的部分叫 BRIEF。

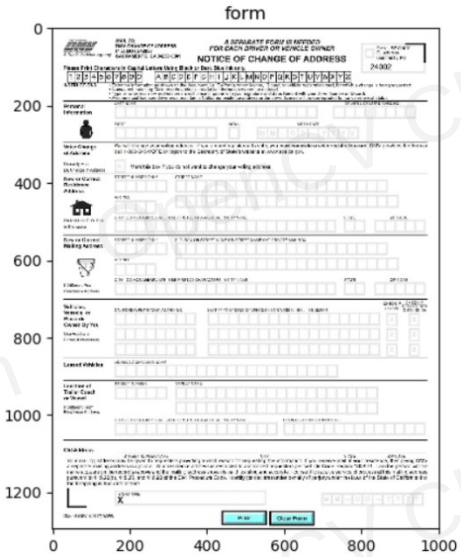
注意：在许多计算机视觉识别应用中，我们解决识别问题主要分为两个步骤，第一步是定位检测，第二步是识别。一个典型的例子就是人脸识别系统，首先需要的就是完成人脸检测，实现人脸定位，常见就是一个矩形框标记，但是人脸检测并不知道他是谁，所以检测完成的任务就是定位发现；接下来就是人脸识别算法，把检测到的人脸图片作为输入，识别算法就是最终确定他是谁，完成人脸识别。

当两张图像相关特征已知的情况下，两张图像之间的单应性矩阵是可以计算出来的。因此一个匹配算法用来实现一张图像特征到另外一张图像特征匹配，一张图像特征的描述子到另外一张图像上描述子相似程度被计算出来，得到好的匹配描述子输出。

基于特征的图像对齐实现步骤

第一步，读取模板表单图像与拍摄图像，并显示，相关代码如下：

```
Mat imReference = imread("D:/form.jpg");
plt::figure_size(800, 600);
displayImage(imReference);
plt::title("form");
plt::show();

Mat img = imread("D:/scanned-form.jpg");
plt::figure_size(800, 600);
displayImage(img);
plt::title("input image");
plt::show();
```



表单

输入图像

第二步，特征检测，代码实现如下：

```
int MAX_FEATURES = 500;
float GOOD_MATCH_PERCENT = 0.15f;

// Convert images to grayscale
Mat im1Gray, im2Gray;
cvtColor(img, im1Gray, COLOR_BGR2GRAY);
cvtColor(imReference, im2Gray, COLOR_BGR2GRAY);

// Variables to store keypoints and descriptors
std::vector<KeyPoint> keypoints1, keypoints2;
Mat descriptors1, descriptors2;

// Detect ORB features and compute descriptors.
Ptr<Feature2D> orb = ORB::create(MAX_FEATURES);
orb->detectAndCompute(im1Gray, Mat(), keypoints1, descriptors1);
orb->detectAndCompute(im2Gray, Mat(), keypoints2, descriptors2);
```

第三步，特征匹配，代码实现如下：

```
// Match features.
std::vector<DMatch> matches;
Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("BruteForce-Hamming");
matcher->match(descriptors1, descriptors2, matches, Mat());

// Sort matches by score
std::sort(matches.begin(), matches.end());

// Remove not so good matches
const int numGoodMatches = matches.size() * GOOD_MATCH_PERCENT;
matches.erase(matches.begin() + numGoodMatches, matches.end());

// Draw top matches
Mat imMatches;
drawMatches(img, keypoints1, imReference, keypoints2, matches, imMatches);
imwrite("matches.jpg", imMatches);
```

运行结果如下：

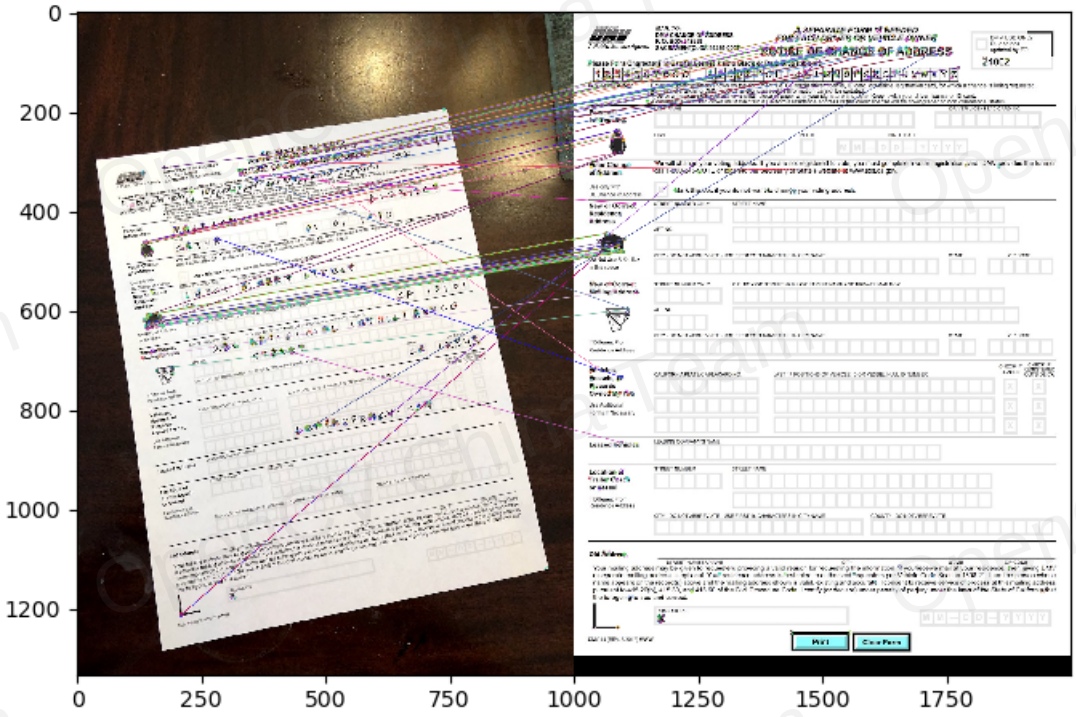


图 7.4-3

第四步，计算单应性矩阵

单应性矩阵计算最少需要两张图像中有四个相关点对或者更多匹配点。自动特征匹配并非100%的正确，大概有 20%~30%的匹配是错误的，在使用函数 findHomography 进行估算的时候需要尽量避免这些非正确匹配的错误影响，这里采用随机采样一致性来完成单应性矩阵估算。

```
// Extract location of good matches
std::vector<Point2f> points1, points2;

for( size_t i = 0; i < matches.size(); i++ ) {
    points1.push_back( keypoints1[ matches[i].queryIdx ].pt );
    points2.push_back( keypoints2[ matches[i].trainIdx ].pt );
}

// Find homography
Mat h = findHomography( points1, points2, RANSAC );
```

第五步，对齐图像，实现文档对齐校正

```
// Use homography to warp image
Mat im1Reg;
warpPerspective(img, im1Reg, h, imReference.size());
plt::figure_size(1000, 600);
plt::subplot(1, 2, 1);
displayImage(img);
plt::title("input");
plt::subplot(1, 2, 2);
displayImage(im1Reg);
plt::title("algined Image");
plt::show();
```

运行结果如下：

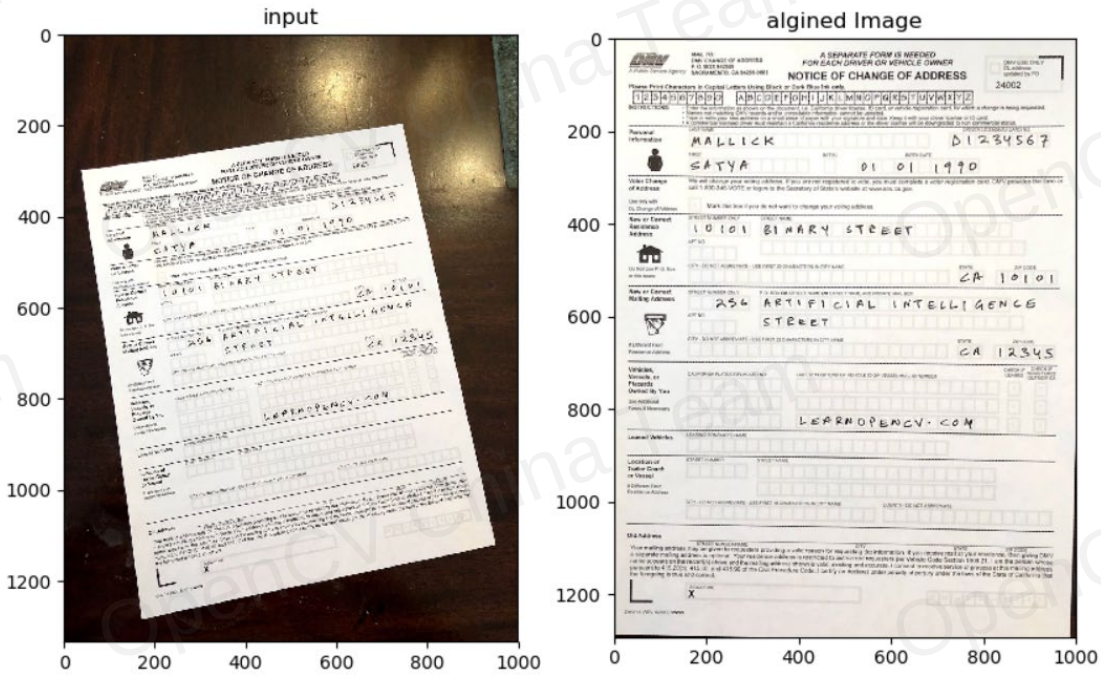


图 7.4-4

7.5 应用-创建全景图

全景图

你可能很好奇全景模式的图像是如何创建的，在一个简单的全景图像中有很多计算机视觉算法被应用，更好的全景模式图像就意味着更多的图像处理算法应用。全景图像生成主要依靠的是对相关图像的对齐与无缝拼接，涉及到 homography 跟 stitch 两个主要算法模块。

创建一个简单的全景图

本节我们如何基于已经学习的知识创建一个全景图像，警告，全景图创建有时候可能会失败！

我们经常使用的基于特征的图像拼接，因为是这项技术使用稀疏特征数据实现一张图像到另外一张图像的匹配。一个转换矩阵(homography)可以基于匹配到的相关特征数据计算得到，在前面的内容中我们已经看到可以实现从一张图像到另外一张图像的转换，实现图像对齐。

一旦第二张图像被参照第一张图像对齐，我们简单的拼接第一张图像跟第二张图像在一起就是可以得到一个简单的全景图像。我们将使用下面两张图像创建全景图像。

图 7.5-1

OpenCV 中实现两张图像的全景图创建步骤如下：

1. 发现两张图像的关键点与描述子
2. 通过描述子匹配发现相关特征点
3. 把第二张图像对齐到一张图像
4. 对第二张图像完成透视变换
5. 拼接两张图像生成一个全景图像

第一步，发现关键点与描述子

全景图像在第一个关键步骤是实现图像对齐，对齐主要是基于特征匹配，计算单应性矩阵，而计算单应性矩阵就需要在两张图像中自动检测跟匹配很多的相关特征点，OpenCV 中已经实现的特征检测算法有 SIFT、SURF、ORB 等，跟 7.4 中阐述的原因一致，我们还是选择 ORB 作为特征检测器，关于 ORB 特征更多解释前面已经有了，这里就不再赘述了。基于 ORB 特征检测器实现关键点与描述子提取的代码实现如下：

```
Mat im1 = imread("D:/scene1.jpg");
Mat im2 = imread("D:/scene3.jpg");

Mat im1Gray, im2Gray;
cvtColor(im1, im1Gray, COLOR_BGR2GRAY);
cvtColor(im2, im2Gray, COLOR_BGR2GRAY);

// Variables to store keypoints and descriptors
std::vector<KeyPoint> keypoints1, keypoints2;
Mat descriptors1, descriptors2;

// Detect ORB features and compute descriptors.
Ptr<Feature2D> orb = ORB::create(MAX_FEATURES);
orb->detectAndCompute(im1Gray, Mat(), keypoints1, descriptors1);
orb->detectAndCompute(im2Gray, Mat(), keypoints2, descriptors2);
```

第二步，发现匹配的相关点

如果两张图像的相关特征点已知的话，两张图像之间变换的单应性矩阵就可以被计算出来，因此匹配算法首先需要实现两张图像之间的特征匹配，自动找到匹配好的相关特征点。代码实现如下：

```
// Match features.
std::vector<DMatch> matches;
Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("BruteForce-Hamming");
matcher->match(descriptors1, descriptors2, matches, Mat());

// Sort matches by score
std::sort(matches.begin(), matches.end());

// Remove not so good matches
const int numGoodMatches = matches.size() * GOOD_MATCH_PERCENT;
matches.erase(matches.begin() + numGoodMatches, matches.end());


// Draw top matches
Mat imMatches;
drawMatches(im1, keypoints1, im2, keypoints2, matches, imMatches);
plt::figure_size(1000, 600);
displayImage(imMatches);
plt::title("matched keypoints");
plt::show();
```

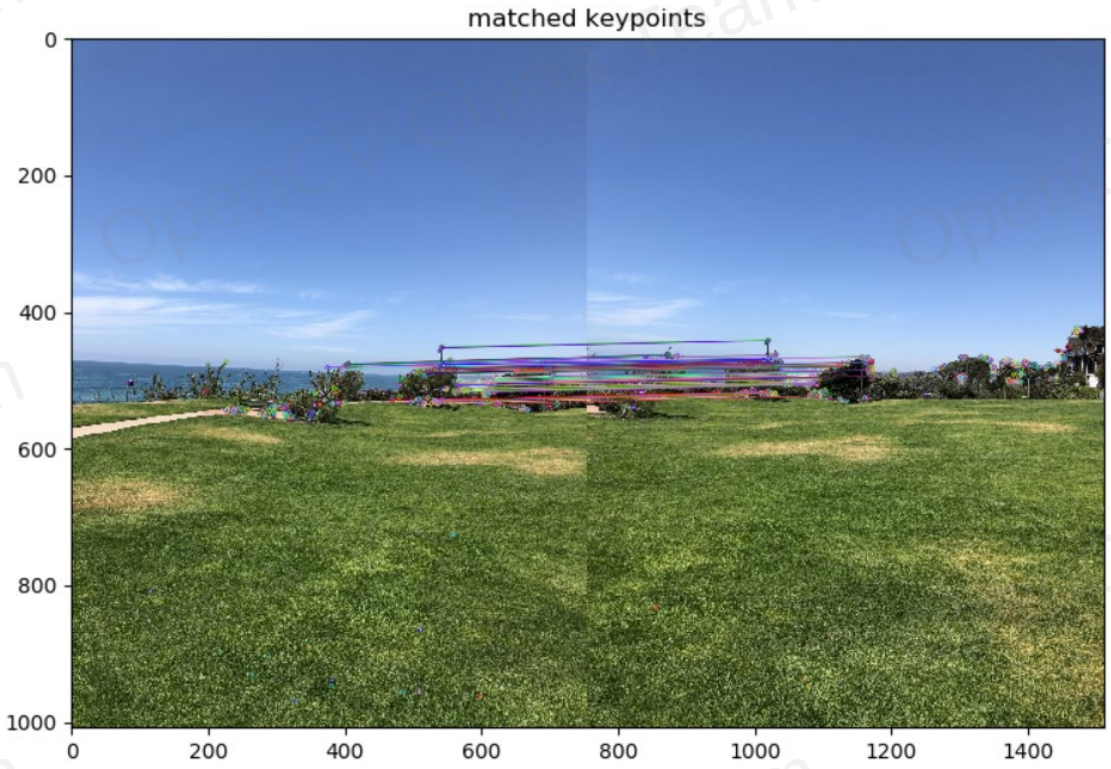


图 7.5-2

第三步，使用单应性矩阵实现图像对齐

自动特征匹配并非 100%的正确，大概有 20%~30%的匹配是错误的，在使用函数 findHomography 进行估算的时候需要尽量避免这些非正确匹配的错误影响，这里采用随机采样一致性来完成单应性矩阵估算。在完成匹配之后，输出的 DMatch 数据对象有如下几个属性数据：

matches.distance – 匹配的描述子之间距离，越低匹配程度越高
mathes.trainIdx – 训练描述子集中描述子索引
matches.queryIdx – 查询描述子集中描述子索引
matches.imgIdx – 训练图像索引
简单点说，queryIdx 跟一张图像相关，trainIdx 跟第二张图像相关，所以创建两个 list 把相关的点都放进去，然后基于这些点估算单应性矩阵，代码实现如下：

```
// Extract location of good matches
std::vector<Point2f> points1, points2;

for (size_t i = 0; i < matches.size(); i++) {
    points1.push_back(keypoints1[matches[i].queryIdx].pt);
    points2.push_back(keypoints2[matches[i].trainIdx].pt);
}

// Find homography
Mat h = findHomography(points2, points1, RANSAC);
```

第四步，图像变换
一旦计算得到单应性矩阵 H，就可以实现图像透视变换了，当使用透视变换函数的时候，需要声明变换以后图像的大小，这里我们需要声明一个不同于图像原来大小的输出图像尺寸。原因是图像后续的拼接是水平方向上的，我们应该声明图像宽度为两张图像宽度之和，高度保持不变，代码实现如下：

```
// Use homography to warp image
int im1Height = im1.rows;
int im1Width = im1.cols;
int im2Height = im2.rows;
int im2Width = im2.cols;
Mat im2Aligned;
warpPerspective(im2, im2Aligned, h, Size(im2Width + im1Width, im2Height));
```

运行结果如下：

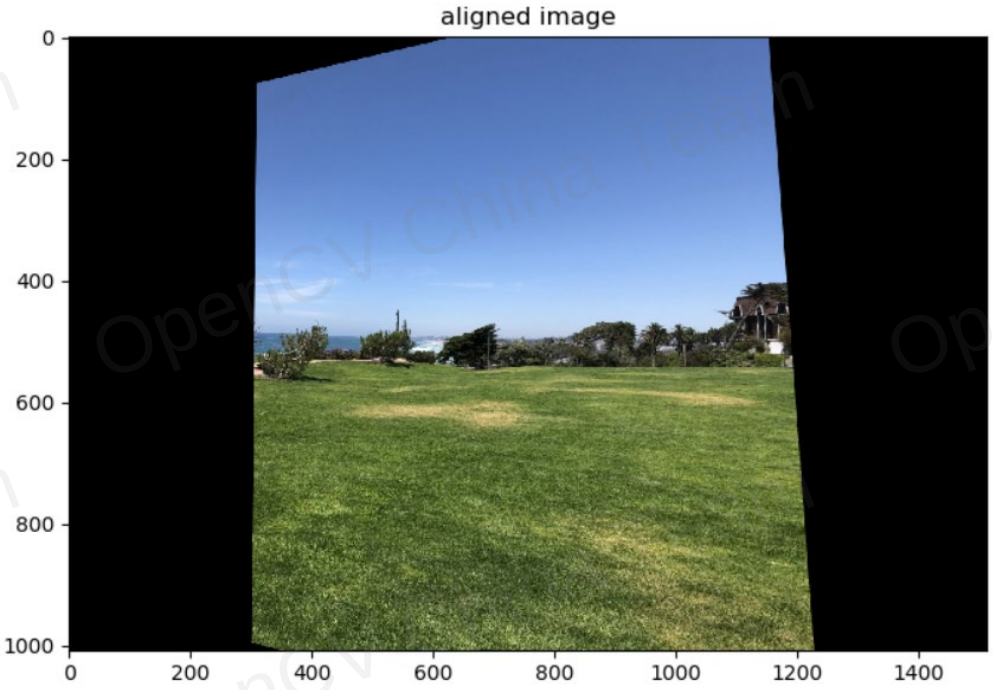


图 7.5-3

第五步，图像拼接，简单的把第一张图像跟透视变换后图像叠加，得到拼接图像，代码实现如下：

```
Mat stitchedImage = im2Aligned.clone();

Rect roi (0,0,im1.cols,im1.rows);
im1.copyTo(stitchedImage(roi));
```

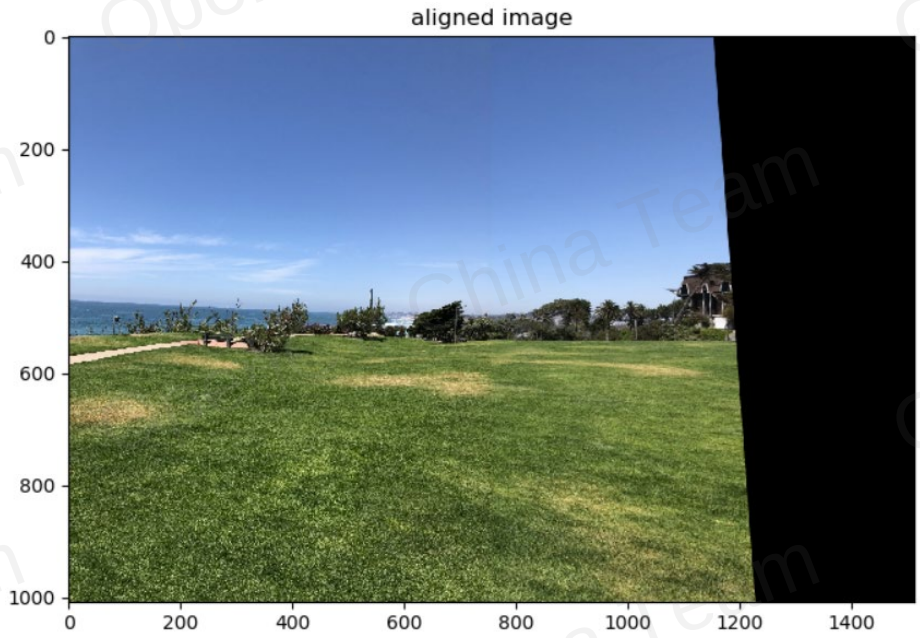


图 7.5-4

总结：

这种拼接方法有一些需要改进地方，主要包括

- 两张图像的拼接结果有缝隙存在，还不是无缝拼接
- 拼接后全景图像有亮度不均衡问题，需要亮度校正
- 该拼接方法很难扩展到多张图像的拼接

OpenCV 中有个专门的 API 函数可以一次实现多张图像的拼接，可以先了解一下，课后的作业就跟这个相关。

7.6 应用-发现已知对象

基于特征匹配实现对已经对象的检测，发现其在场景图像中位置并绘制出来。

首先需要加载两张图像，其中一张图像我们称为对象图像，是一本书；另外一张图像，称为场景图像，里面有第一张图像内容。代码实现如下：

```
int MAX_FEATURES = 1000;
int MIN_MATCH_COUNT = 10;
Mat img1 = imread("D:/book.jpeg");
Mat img1Gray;
cvtColor(img1, img1Gray, COLOR_BGR2GRAY);

Mat img2 = imread("D:/book_scene.jpeg");
Mat img2Gray;
cvtColor(img2, img2Gray, COLOR_BGR2GRAY);
```

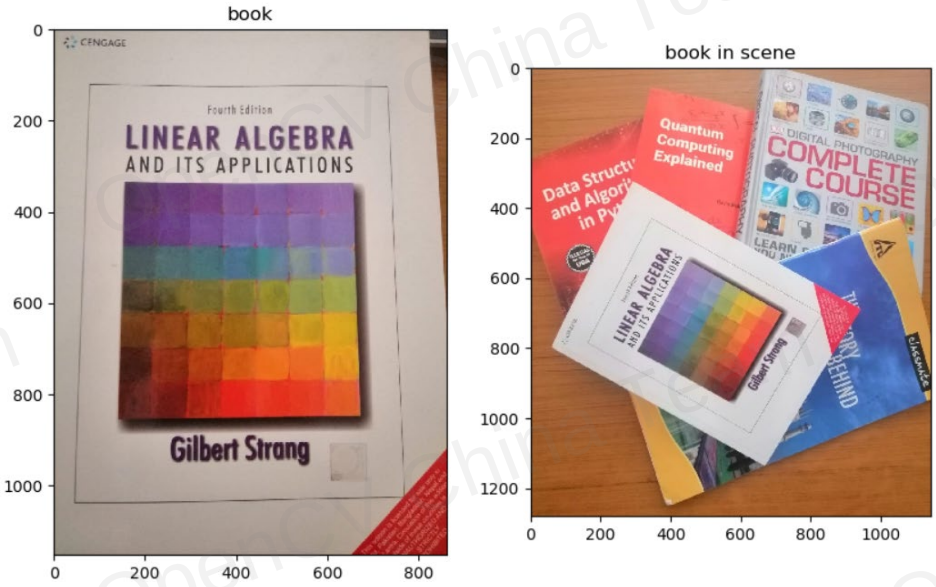


图 7.6-1

使用 ORB 特征检测发现特征点于描述子，代码实现如下：

```
// Initiate ORB detector
Ptr<ORB> orb = ORB::create(MAX_FEATURES);

// find the keypoints with ORB
vector<KeyPoint> keypoints1, keypoints2;
Mat descriptors1, descriptors2;

orb -> detectAndCompute(img1Gray,Mat(),keypoints1,descriptors1);
orb -> detectAndCompute(img2Gray,Mat(),keypoints2,descriptors2);
```

使用特征匹配发现相关匹配点，代码实现如下：

```
FlannBasedMatcher matcher(new flann::KDTreeIndexParams(5), new flann::SearchParams(50));
std::vector< std::vector<DMatch> > matches;
descriptors1.convertTo(descriptors1, CV_32F);
descriptors2.convertTo(descriptors2, CV_32F);
matcher.knnMatch(descriptors1, descriptors2, matches, 2);
```

过滤好的相关点匹配数据，代码实现如下：

```
//-- Filter matches using the Lowe's ratio test
const float ratio_thresh = 0.9f;
std::vector<DMatch> good;
for (size_t i = 0; i < matches.size(); i++) {
    if (matches[i][0].distance < ratio_thresh * matches[i][1].distance) {
        good.push_back(matches[i][0]);
    }
}

std::vector<Point2f> src_pts;
std::vector<Point2f> dst_pts;
for (size_t i = 0; i < good.size(); i++) {
    //-- Get the keypoints from the good matches
    src_pts.push_back(keypoints1[good[i].queryIdx].pt);
    dst_pts.push_back(keypoints2[good[i].trainIdx].pt);
}
```

基于好的匹配相关点，计算对象图像到场景中目标的单应性变换矩阵 H，基于 H 实现透视变换，得到对象在场景图像中的外接矩形四点坐标，实现对对象定位检测，代码实现如下：

```
//-- Draw matches
Mat img3;
drawMatches(img1, keypoints1, img2, keypoints2, good, img3, Scalar(0, 255, 0),
            Scalar::all(-1), std::vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
if (good.size() > MIN_MATCH_COUNT) {
    Mat H = findHomography(src_pts, dst_pts, RANSAC, 5.0);

    //-- Get the corners from the image_1 ( the object to be "detected" )
    std::vector<Point2f> obj_corners(4);
    obj_corners[0] = Point2f(0, 0);
    obj_corners[1] = Point2f((float)img1.cols, 0);
    obj_corners[2] = Point2f((float)img1.cols, (float)img1.rows);
    obj_corners[3] = Point2f(0, (float)img1.rows);
    std::vector<Point2f> scene_corners(4);
    perspectiveTransform(obj_corners, scene_corners, H);

    //-- Draw lines between the corners (the mapped object in the scene - image_2 )
    line(img3, scene_corners[0] + Point2f((float)img1.cols, 0),
         scene_corners[1] + Point2f((float)img1.cols, 0), Scalar(0, 0, 255), 10);
    line(img3, scene_corners[1] + Point2f((float)img1.cols, 0),
         scene_corners[2] + Point2f((float)img1.cols, 0), Scalar(0, 0, 255), 10);
    line(img3, scene_corners[2] + Point2f((float)img1.cols, 0),
         scene_corners[3] + Point2f((float)img1.cols, 0), Scalar(0, 0, 255), 10);
    line(img3, scene_corners[3] + Point2f((float)img1.cols, 0),
         scene_corners[0] + Point2f((float)img1.cols, 0), Scalar(0, 0, 255), 10);
} else {
    cout << "Not enough matches are found - " << good.size() << "/" << MIN_MATCH_COUNT;
}
```

运行结果如下：

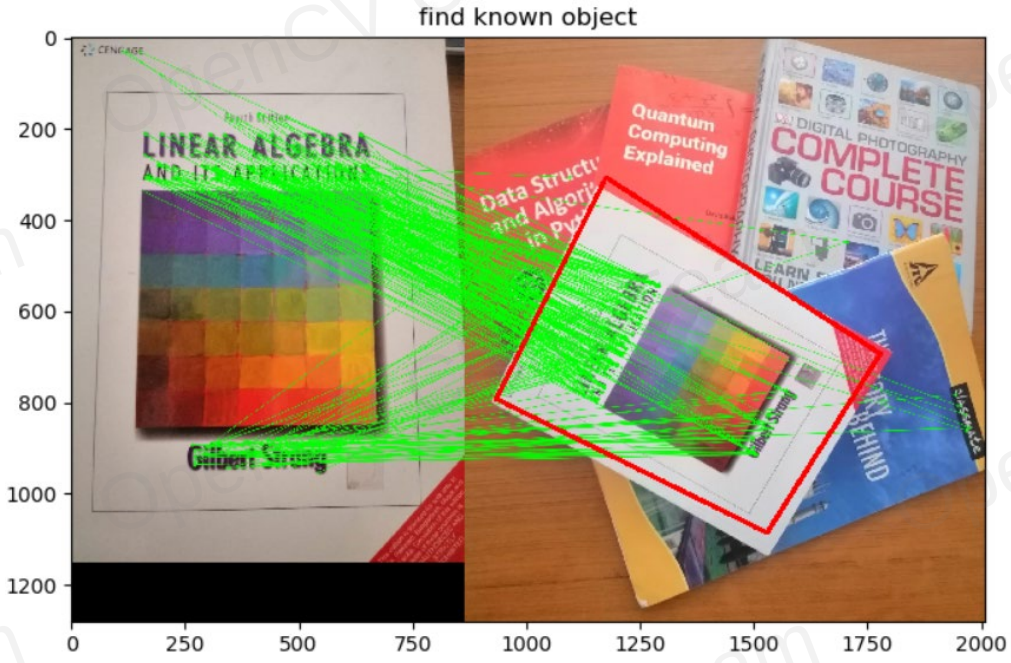


图 7.6-2

作业一：

全景图拼接

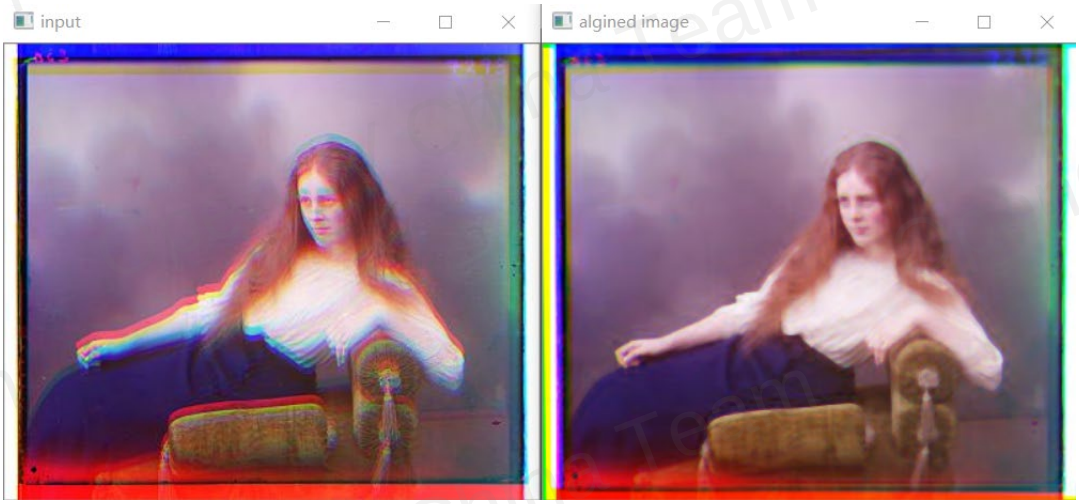
参考 OpenCV 官方的全景图拼接函数 API， 使用 Stitcher 类实现多张（超过两张）图像的全

景图拼接，并提交作业，阅读 OpenCV 官方全景图拼接原理并写下记录文档跟代码一同提交

```
// 设置拼接模式与参数
Mat result1, result2, result3;
Stitcher::Mode mode = Stitcher::PANORAMA;
Ptr<Stitcher> stitcher = Stitcher::create(mode);

// 拼接方式-多通道融合
auto blender = detail::Blender::createDefault(detail::Blender::MULTI_BAND);
stitcher->setBlender(blender);

// 拼接
auto affine_warper = makePtr<cv::AffineWarper>();
stitcher->setWarper(affine_warper);
Stitcher::Status status = stitcher->stitch(images, result1);

// 平面曲翘拼接
auto plane_warper = makePtr<cv::PlaneWarper>();
stitcher->setWarper(plane_warper);
status = stitcher->stitch(images, result2);

// 鱼眼拼接
auto fisheye_warper = makePtr<cv::FisheyeWarper>();
stitcher->setWarper(fisheye_warper);
status = stitcher->stitch(images, result3);

// 检查返回
if (status != Stitcher::OK) {
    cout << "Can't stitch images, error code = " << int(status) << endl;
    return EXIT_FAILURE;
}
imwrite("D:/result1.png", result1);
imwrite("D:/result2.png", result2);
imwrite("D:/result3.png", result3);
```

作业二：

早期的彩色相机是俄国人发明的，它有三个成像通道，分别是 RED、GREEN、BLUE，然后使用三个通道叠加得到 RGB 彩色图像，但是因为三个通道都是独立成像，所以在叠加的时候因为没有进行特征对齐导致成像效果如下图左侧所示，现在请使用本章所学知识，利用 OpenCV 实现对三个通道图像对其，完成对原始图像的对齐叠加，下图右侧所示为对齐后的结果。



作业提示：

1. 读入图像
2. 通道分离
3. 使用绿色通道作为基准图像，对齐红色跟蓝色通道
4. 对齐特征并合并
5. 输出上述的对比图像显示。