哈爾濱工業大學

实验报告

课程名称：数字图像处理

班级：1604104、1604105

同组人：田昊宇 1160400417

宋健 1160400518

分工：田昊宇：C++程序设计

宋健：Python 程序设计

实验一 透视变换实验

一．公式推导

设原图像中点的坐标为$\left(x,y,1\right)^T$，变换后图像中对应点的坐标为$\left(x',y',1\right)^T$

则两点坐标之间的关系如下：

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$x' = \frac{u}{w}, \mathrm{y}' = \frac{v}{w}$$

即原图像中的点的坐标经过几何变换后，得到生成图像中点的坐标，实验内容为求取几何变换矩阵 $\boldsymbol{H}$，矩阵中参数h_{33}恒为 1，其他 8 个参数是待求参数，原图像中一个点和生成图像中一个点的对应关系可以形成两个方程，即：

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \frac{1}{w} \begin{pmatrix} u \\ v \\ w \end{pmatrix} = \frac{1}{w} \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$\begin{cases} x' = \dfrac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}} \\ y' = \dfrac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}} \end{cases}$$

$$\begin{cases} \left(h_{31}x + h_{32}y + h_{33}\right)x' = h_{11}x + h_{12}y + h_{13} \\ \left(h_{31}x + h_{32}y + h_{33}\right)y' = h_{21}x + h_{22}y + h_{23} \end{cases}$$

一对对应点可以形成两个约束方程，有四对对应点就可以求出矩阵 $\boldsymbol{H}$ 的全部参数。设这四对点的坐标为$\left(x_i,y_i\right),\left(x_i',y_i'\right) i=1\sim4$

可以将约束方程改写成以下形式：

$$\begin{aligned} x_i h_{11} + y_i h_{12} + h_{13} - x_i x_i' h_{31} - y_i x_i' h_{32} = x_i' \\ x_i h_{21} + y_i h_{22} + h_{23} - x_i y_i' h_{31} - y_i y_i' h_{32} = y_i' \end{aligned}$$

将待求参数转为列向量：

$$\boldsymbol{h} = \begin{pmatrix} h_{11} & h_{12} & h_{13} & h_{21} & h_{22} & h_{23} & h_{31} & h_{32} \end{pmatrix}^T$$

将上面得到的 8 个方程写成矩阵形式：

$$\begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x_1' & -y_1x_1' \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x_4' & -y_4x_4' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y_1' & -y_1y_1' \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y_4' & -x_4y_4' \end{pmatrix} \boldsymbol{h} = \begin{pmatrix} x_1' \\ \vdots \\ x_4' \\ y_1' \\ \vdots \\ y_4' \end{pmatrix}$$

由此可以解出所有参数：

$$\boldsymbol{h} = \begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x_1' & -y_1x_1' \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x_4' & -y_4x_4' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y_1' & -y_1y_1' \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y_4' & -x_4y_4' \end{pmatrix}^{-1} \begin{pmatrix} x_1' \\ \vdots \\ x_4' \\ y_1' \\ \vdots \\ y_4' \end{pmatrix}$$

转换回方阵就可以得到几何变换矩阵 ***H***。

二． 程序源码（C++）

1. 透视几何变换矩阵求解函数

```
Mat MyGetPerspectiveMatrix(Point2f srcPoints[4],Point2f dstPoints[4])
{
  Mat H = Mat::ones(3,3,CV_32FC1);
  float Xsrc[4],Ysrc[4];
  float Xdst[4],Ydst[4];
  int i;
  //将Point2F数组坐标数据存入数组
  for (i=0;i<4;i++)
  {
    Xsrc[i] = srcPoints[i].x;
    Ysrc[i] = srcPoints[i].y;
    Xdst[i] = dstPoints[i].x;
    Ydst[i] = dstPoints[i].y;
  }
  //定义矩阵A（8*8）,B（8*1）,h（8*1）
  Mat matA = Mat::zeros(8,8,CV_32FC1);
  Mat matB = Mat::zeros(8,1,CV_32FC1);
  Mat matH = Mat::zeros(8,1,CV_32FC1);
  //矩阵A,B赋值
  for (i=0;i<4;i++)
  {
    matA.at<float>(i,0) = Xsrc[i];
    matA.at<float>(i,1) = Ysrc[i];
    matA.at<float>(i,2) = 1;
```

```
      matA.at<float>(i+4,3) = Xsrc[i];
      matA.at<float>(i+4,4) = Ysrc[i];
      matA.at<float>(i+4,5) = 1;
      matA.at<float>(i,6) = -Xsrc[i]*Xdst[i];
      matA.at<float>(i,7) = -Ysrc[i]*Xdst[i];
      matA.at<float>(i+4,6) = -Xsrc[i]*Ydst[i];
      matA.at<float>(i+4,7) = -Ysrc[i]*Ydst[i];

      matB.at<float>(i,0) = Xdst[i];
      matB.at<float>(i+4,0) =Ydst[i];
    }
    //求解参数
    matH = matA.inv(DECOMP_LU) * matB;
    //存入矩阵
    H.at<float>(0,0) = matH.at<float>(0,0);
    H.at<float>(0,1) = matH.at<float>(1,0);
    H.at<float>(0,2) = matH.at<float>(2,0);
    H.at<float>(1,0) = matH.at<float>(3,0);
    H.at<float>(1,1) = matH.at<float>(4,0);
    H.at<float>(1,2) = matH.at<float>(5,0);
    H.at<float>(2,0) = matH.at<float>(6,0);
    H.at<float>(2,1) = matH.at<float>(7,0);
    return H;
  }
```

2. 透视变换函数

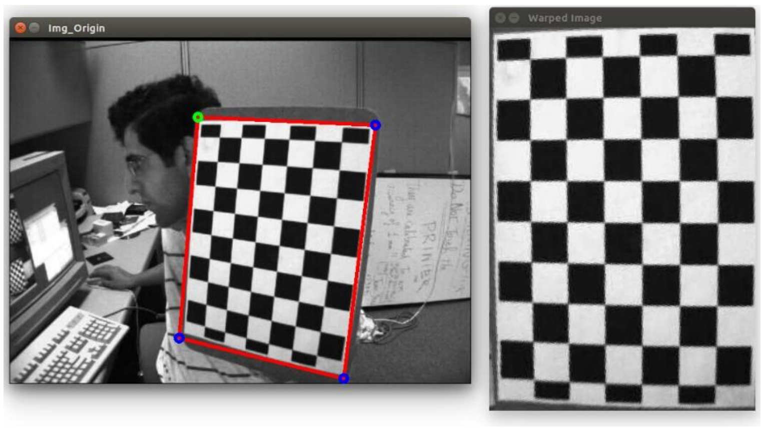
```
void MakeWarpPerspective(Mat &src_Image,Mat &dst_Image,Mat
transform_Mat,Size WarpImageSize)
{
  //规定生成图像的大小
  dst_Image=Mat::zeros(WarpImageSize,CV_8UC3);
  int src_rows,src_cols;
  float u,v,w;
  //原图像行数和列数
  src_rows = src_Image.rows;
  src_cols = src_Image.cols;
  //原图像与生成图像中的点的齐次坐标向量
  Mat dst_point=Mat::ones(3,1,CV_32FC1);
  Mat src_point=Mat::ones(3,1,CV_32FC1);
  //像素坐标
  Point2i pt_src,pt_dst;
  for(int i = 0; i < WarpImageSize.width; i++)
  {
    for(int j = 0; j < WarpImageSize.height; j++)
    {
      dst_point.at<float>(0,0) = (float)i;
      dst_point.at<float>(1,0) = (float)j;
      //对该点进行变换
      src_point = (transform_Mat.clone()).inv()*dst_point;
      u = src_point.at<float>(0,0);
      v = src_point.at<float>(1,0);
      w = src_point.at<float>(2,0);
      //齐次坐标转换为实际像素坐标
      pt_src.x = floor(u/w);
      pt_src.y = floor(v/w);
      pt_dst = Point2i(i,j);
      //未找到该点对应点时绘制为黑色
      if((pt_src.x < 0)||(pt_src.y < 0))
      {
        dst_Image.at<Vec3b>(pt_dst)[0] = 0;
        dst_Image.at<Vec3b>(pt_dst)[1] = 0;
        dst_Image.at<Vec3b>(pt_dst)[2] = 0;
      }
      else if((pt_src.x >= src_cols)||(pt_src.y >= src_rows))
      {
        dst_Image.at<Vec3b>(pt_dst)[0] = 0;
        dst_Image.at<Vec3b>(pt_dst)[1] = 0;
        dst_Image.at<Vec3b>(pt_dst)[2] = 0;
      }
      else
        dst_Image.at<Vec3b>(pt_dst) = src_Image.at<Vec3b>(pt_src);
    }
  }
}
```

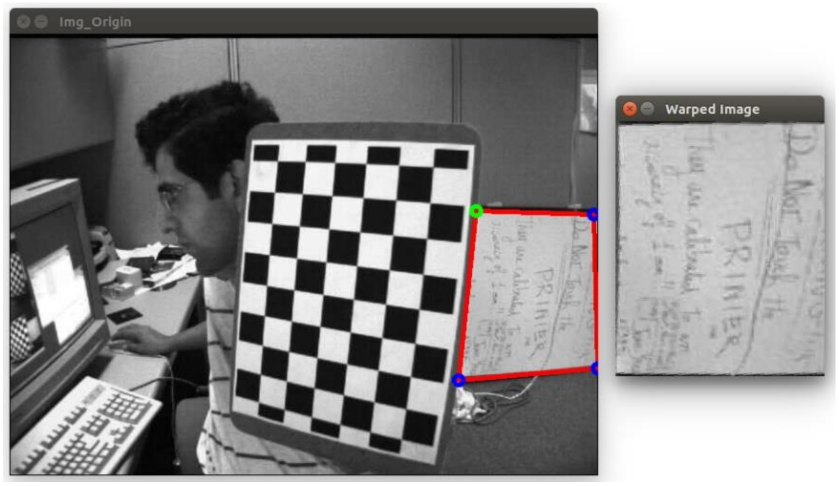
3. 主函数与鼠标操作函数

```
#include "opencv2/opencv.hpp"
#include "opencv2/imgproc.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/calib3d.hpp"
#include <iostream>

using namespace std;
using namespace cv;

float Image_Gain = 1.5;  //放大倍数
Mat MyGetPerspectiveMatrix(Point2f srcPoints[4],Point2f dstPoints[4]);
void MakeWarpPerspective(Mat &src_Image,Mat &dst_Image,Mat
transform_Mat,Size WarpImageSize);
//鼠标操作函数
void onMouse(int event, int x, int y, int, void*);
//原图像与变换后图像的像素点坐标
Point2f OriginPoints[4];
```

```
Point2f WarpPoints[4];
//是否拖拽点的标识
bool dragging;
//选中的点的编号
int selected_corner_index = 0;

int main()
{
  Mat Img_Origin;
  //读取并显示原始图像
  Img_Origin = imread("/home/thy/Pictures/right06.jpg");
  imshow("Img_Origin",Img_Origin);
  //原始图像的行数、列数
  float OriginCols,OriginRows;
  OriginCols = (float)Img_Origin.cols;
  OriginRows = (float)Img_Origin.rows;
  //原图像上初始选定的四个点
  OriginPoints[0] = Point2f(OriginCols*0.33,OriginRows*0.33);
  OriginPoints[1] = Point2f(OriginCols*0.67,OriginRows*0.33);
  OriginPoints[2] = Point2f(OriginCols*0.67,OriginRows*0.67);
  OriginPoints[3] = Point2f(OriginCols*0.33,OriginRows*0.67);
  //变换后图像的初始坐标（初始化为400*300，之后根据框选点坐标调整）
  WarpPoints[0] = Point2f(0,0);
  WarpPoints[1] = Point2f(400,0);
  WarpPoints[2] = Point2f(400,300);
  WarpPoints[3] = Point2f(0,300);

  Size WarpImageSize;  //变换后图像大小
  Mat ShowOriginImage;  //显示图像（原始图像上加标记）
  Mat WarpImage;  //变换后图像
  Mat PerspectiveMatrix;  //变换矩阵
  setMouseCallback("Img_Origin", onMouse, 0);  //设置鼠标操作函数
  char c;  //读取按键输入
  //框选四边形的四边长度
  float Length_Left,Length_Right,Length_Top,Length_Bottom;
  //图像长度
  float ImageWidth,ImageHeight;
  for(int time_current=0;;time_current+=10)
  {
    ShowOriginImage=Img_Origin.clone();  //复制原图像
    //绘制四边形边框
    line(ShowOriginImage,OriginPoints[0],OriginPoints[1],Scalar(0,0,255),3);
    line(ShowOriginImage,OriginPoints[1],OriginPoints[2],Scalar(0,0,255),3);
    line(ShowOriginImage,OriginPoints[2],OriginPoints[3],Scalar(0,0,255),3);
    line(ShowOriginImage,OriginPoints[3],OriginPoints[0],Scalar(0,0,255), 3);
    //标记四个角点（左上角点绿色，其他角点蓝色）
    circle(ShowOriginImage, OriginPoints[0], 5, Scalar(0, 255, 0), 3);
    for(int i=1;i<4;i++)
      circle(ShowOriginImage, OriginPoints[i], 5, Scalar(255, 0, 0), 3);
    imshow("Img_Origin",ShowOriginImage);  //显示
        //计算一边两端点之间的距离作为边长
    Length_Left = norm(OriginPoints[0] - OriginPoints[3]);
    Length_Right = norm(OriginPoints[1] - OriginPoints[2]);
    Length_Top = norm(OriginPoints[0] - OriginPoints[1]);
    Length_Bottom = norm(OriginPoints[3] - OriginPoints[2]);
    //顶边与底边的较大值作为宽度，左边右边的较大值作为高度
    ImageWidth = Image_Gain *(float)max(Length_Top,Length_Bottom);
    ImageHeight = Image_Gain *(float)max(Length_Left,Length_Right);
    //得到生成图像对应四点坐标
    WarpPoints[0].x = 0;
    WarpPoints[0].y = 0;
    WarpPoints[1].x = ImageWidth;
    WarpPoints[1].y = 0;
    WarpPoints[2].x = ImageWidth;
    WarpPoints[2].y = ImageHeight;
    WarpPoints[3].x = 0;
    WarpPoints[3].y = ImageHeight;
    //定义生成图像尺寸
    WarpImageSize = Size(cvRound(WarpPoints[2].x), cvRound(WarpPoints[2].y));
    //求出变换矩阵
    PerspectiveMatrix = MyGetPerspectiveMatrix(OriginPoints, WarpPoints);
    //求出变换后图像
    MakeWarpPerspective(Img_Origin,WarpImage,PerspectiveMatrix,WarpImageSize)
;
    //显示变换后图像
    imshow("Warped Image", WarpImage);
    if(time_current%1000 == 0)
    {
      cout << "PerspectiveMatrix" << endl;
      cout << PerspectiveMatrix << endl; //每1000ms输出一次变换矩阵
      cout << endl;
    }
    c = waitKey(10);
    if(c == 'q')
      break;  //键盘输入q时退出
  }
  return 0;
}
void onMouse(int event, int x, int y, int, void*)
{
    float dist;  //距离
    for (int i = 0; i < 4; ++i)
    {
        //点击点与前一次选取的角点的距离
        dist = (float)norm(OriginPoints[i]-Point2f(x,y));
        if ((event == EVENT_LBUTTONDOWN) & (dist < 10))
        {
            //左键按下且与角点距离小于10时记录选取的点序号并标记为正在拖动
            selected_corner_index = i;
            dragging = true;
        }
    }
    if (event == EVENT_LBUTTONUP)
        dragging = false;  //左键松开，标记为不在拖动
    if ((event == EVENT_MOUSEMOVE) && dragging)
    {
        //鼠标移动且标识为正在拖动时，不断更新正在拖动点的坐标值
        OriginPoints[selected_corner_index].x = (float) x;
        OriginPoints[selected_corner_index].y = (float) y;
    }
}
```

三. 实验结果及分析（C++）





四．程序源码（Python）

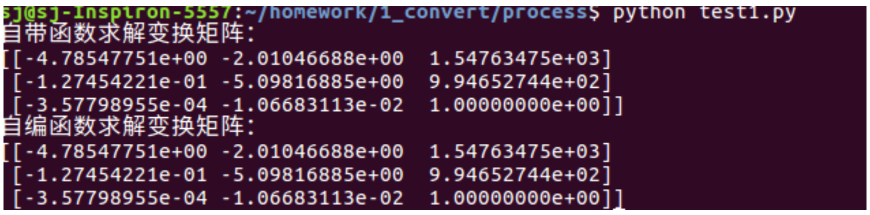
```python
import cv2 as cv
import numpy as np
import matplotlib.pylab as plt
import math
from PIL import Image

#原图像四点
orign_point1 = (244,189)
orign_point2 = (102,527)
orign_point3 = (443,528)
orign_point4 = (324,187)

def Getlen(point1,point2):
    x1,y1 = point1
    x2,y2 = point2
    distance = math.sqrt((x1-x2)**2+(y1-y2)**2)
    return distance

def SelectPoint(x,y):
    Point0 = (x,y)
    ditance = 10
    if Getlen(Point0,orign_point1) < ditance:
        global orign_point1
        orign_point1 = Point0
    elif Getlen(Point0,orign_point2) < ditance:
        global orign_point2
        orign_point2 = Point0
    elif Getlen(Point0,orign_point3) < ditance:
        global orign_point3
        orign_point3 = Point0
    elif Getlen(Point0,orign_point4) < ditance:
        global orign_point4
        orign_point4 = Point0

def onMouse(event,x,y,flags,param):
    if event == cv.EVENT_LBUTTONDOWN:
        SelectPoint(x,y)
    elif flags == cv.EVENT_FLAG_LBUTTON:
        SelectPoint(x,y)
    elif event == cv.EVENT_LBUTTONUP:
        SelectPoint(x,y)
```

```
def GetWarpMatric_seif(pos1,pos2):
    assert pos1.shape[0] == pos2.shape[0] and pos1.shape[0] >= 4
    nums = pos1.shape[0]
    # A*warpMatrix=B
    A = np.zeros((2*nums, 8))
    B = np.zeros((2*nums, 1))
    for i in range(0,nums):
        A.itemset((2*i,0), pos1[i][0])
        A.itemset((2*i,1), pos1[i][1])
        A.itemset((2*i,2), 1)
        A.itemset((2*i,6),-pos1[i][0]*pos2[i][0])
        A.itemset((2*i,7),-pos1[i][1]*pos2[i][0])
        A.itemset((2*i+1,3), pos1[i][0])
        A.itemset((2*i+1,4), pos1[i][1])
        A.itemset((2*i+1,5), 1)
        A.itemset((2*i+1,6), -pos1[i][0]*pos2[i][1])
        A.itemset((2*i+1,7), -pos1[i][1]*pos2[i][1])
        B.itemset((2*i,0),pos2[i][0])
        B.itemset((2*i+1,0),pos2[i][1])
    A = np.mat(A)
    #求出 a_11, a_12, a_13, a_21, a_22, a_23, a_31, a_32
    warpMatrix = A.I * B
    #之后为结果的后处理
    warpMatrix = np.array(warpMatrix).T[0]
    warpMatrix = np.insert(warpMatrix, warpMatrix.shape[0], values=1.0, 
axis=0)
    #插入 a_33 = 1
    warpMatrix = warpMatrix.reshape((3, 3))
    return warpMatrix

def imagePers(image):
    #图像透射需要 3*3 矩阵，通过函数 M = cv2.getPerspectiveTransform(pts1,pts2),
    #其中 pts 需要变换前后的 4 个点对应位置。
    #最后通过透射函数 warpPerspective 进行变换
    row,col = image.shape[:2]

    pos1 = np.float32([orign_point1,orign_point2,orign_point3,orign_point4])
    #目标图像 size
    len_left = Getlen(orign_point1,orign_point2)
    len_right = Getlen(orign_point3,orign_point4)
    len_top = Getlen(orign_point1,orign_point4)
    len_bottom = Getlen(orign_point2,orign_point3)
    image_width = int(max(len_top,len_bottom))
    image_height = int(max(len_left,len_right))
```

```python
    size = (image_width,image_height)
    #目标点
    warp_point1 = (0,0)
    warp_point2 = (0,image_height)
    warp_point3 = (image_width,image_height)
    warp_point4 = (image_width,0)
    pos2 = np.float32([warp_point1,warp_point2,warp_point3,warp_point4])
    #求取变换矩阵
    M = GetWarpMatric_seif(pos1,pos2)
    #透射变换
    dst = cv.warpPerspective(image,M,size)
    return dst,M

Image_Original = cv.imread('666.jpg')
print(Image_Original.shape)
cv.namedWindow("Image_Original",cv.WINDOW_AUTOSIZE)
cv.setMouseCallback("Image_Original",onMouse)
cv.namedWindow("Perspective_image",cv.WINDOW_AUTOSIZE)
while(True):
    Image = Image_Original.copy()
    cv.line(Image,orign_point1,orign_point2,(0,0,255),3)
    cv.line(Image,orign_point2,orign_point3,(0,0,255),3)
    cv.line(Image,orign_point3,orign_point4,(0,0,255),3)
    cv.line(Image,orign_point1,orign_point4,(0,0,255),3)
    cv.circle(Image,orign_point1,8,(255,0,0),-1)
    cv.circle(Image,orign_point2,5,(255,0,0),-1)
    cv.circle(Image,orign_point3,5,(255,0,0),-1)
    cv.circle(Image,orign_point4,5,(255,0,0),-1)
    Perspective_image,M_Perspective = imagePers(Image_Original)
    cv.imshow("Image_Original",Image)
    cv.imshow("Perspective_image",Perspective_image)
    # 按下 ESC 键退出
    if cv.waitKey(20) == 27:
       break
cv.destroyAllWindows()
```

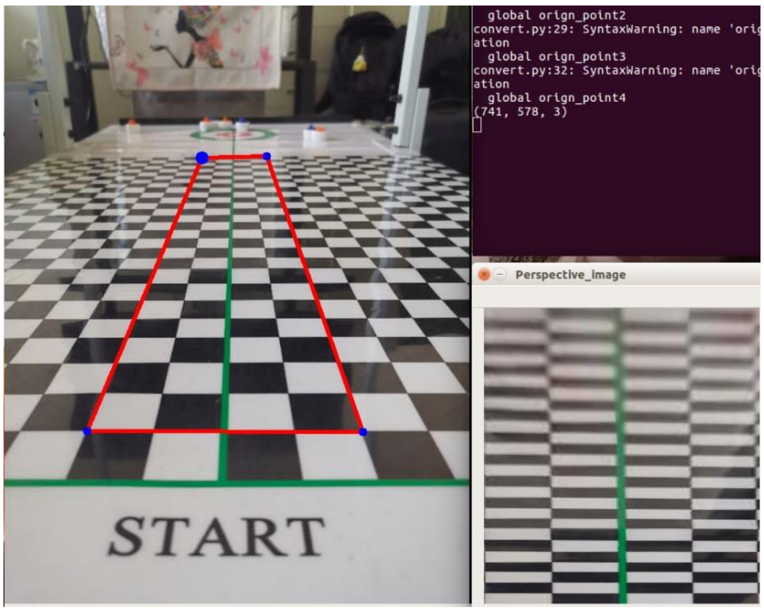
五．实验结果及分析（Python）

1. 透视变换矩阵求取结果：

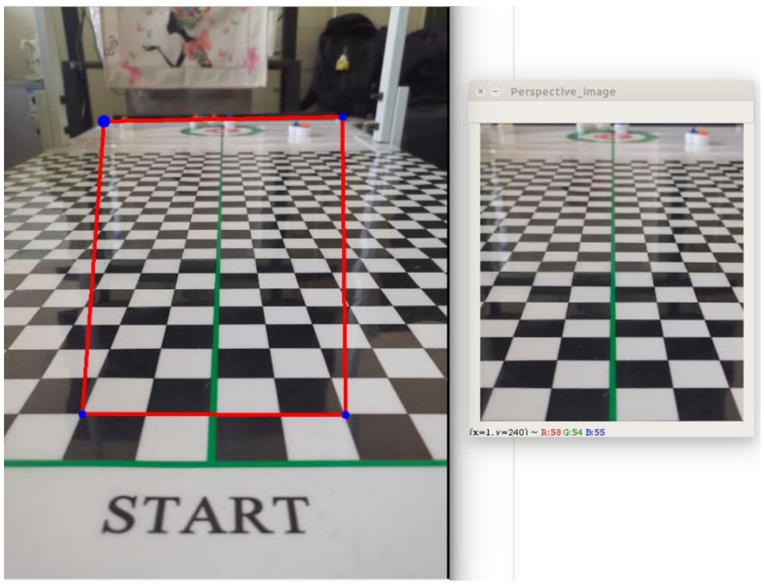


```
sj@sj-Inspiron-5557:~/homework/1_convert/process$ python test1.py
自带函数求解变换矩阵:
[[-4.78547751e+00 -2.01046688e+00  1.54763475e+03]
 [-1.27454221e-01 -5.09816885e+00  9.94652744e+02]
 [-3.57798955e-04 -1.06683113e-02  1.00000000e+00]]
自编函数求解变换矩阵:
[[-4.78547751e+00 -2.01046688e+00  1.54763475e+03]
 [-1.27454221e-01 -5.09816885e+00  9.94652744e+02]
 [-3.57798955e-04 -1.06683113e-02  1.00000000e+00]]
```

2. 透视变换结果

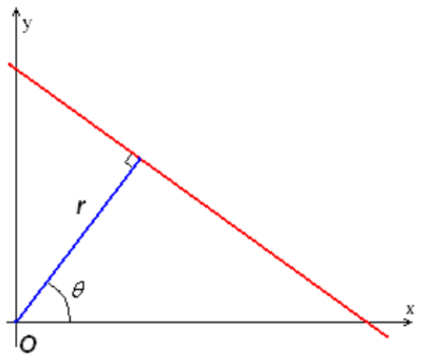


```
  global orign_point2
convert.py:29: SyntaxWarning: name 'orig
ation
  global orign_point3
convert.py:32: SyntaxWarning: name 'orig
ation
  global orign_point4
(741, 578, 3)
```



实验二 Hough 变换直线检测

一．数学基础



如图，像素平面内的直线可以表示成以下形式：

$$x\cos\theta + y\sin\theta = \rho$$

其中 ρ 是原点到直线的垂线段的长度， θ 是这条垂线段与 x 轴的夹角。由此可以将原图像中每一个点与一对参数(ρ,θ)对应。我们将参数(ρ,θ)变化的空间定义为霍夫空间。

对于原图像内任意一点，都有无数条直线经过该点，经过该点的所有直线定义为一个“直线簇”，该直线簇在霍夫空间上可以映射成一条曲线，这条曲线是 ρ 关于 θ 的三角函数曲线。也就是说，原图像上的一条线可以对应于霍夫空间上一点，原图像上的一点对应于霍夫空间内一条曲线。

对于图像的直线检测问题，将图像阈值化后的点映射到霍夫空间上，每一点对应于一条曲线，霍夫空间上的图像是多条曲线的叠加。寻找霍夫空间图像中的像素最大值，像素值最大的点表示经过该点的曲线数最多，也就是在原图像上该直线经过的点最多，这条直线最有可能是图像中的直线。

二．程序思路

1. 主程序

（1）读取原图像并显示。

（2）选取四组对应点，求取透视变换矩阵及逆变换矩阵，做透视变换后图像。

（3）将透视变换后的图像分离 HSV 三通道，阈值化处理，筛选其中的绿色部分再将三通道合并。

（4）做霍夫变换直线检测，得到一组直线的两端点坐标。

（5）将坐标用透视变换逆矩阵变换到原图像上，标记直线。

2. 霍夫变换

（1)将图像中所有的选中的点坐标提取出来存入一个类型为 Point2i 的 vector 中用于后面的计算， 这些点到原点距离的最大值作为霍夫空间中距离 ρ 的最大

值。

（2）定义霍夫空间空白图像，角度范围-90° 到 180° ，距离范围从 0 到距离最大值，类型 CV_32FC1。

（3）遍历 vector 内所有点，绘制霍夫空间内图像。对于每一个点，求取对应于 0° 到 180° 每一个角度（间距 1° ），求取距离 ρ （若 ρ 小于 0 则更正为其相反数，再将角度-180° ），将该组(ρ,θ)参数对应的点在霍夫空间图像中找出，并将该点像素值加一。

（4）将霍夫空间内像素值最大的若干个点找出，并找出它们的(ρ,θ)参数。

（5）根据参数求出该直线在图像上的两个端点坐标（分为垂直于 x 轴，垂直于 y 轴，斜线三种情况讨论）

（6）将若干条直线的端点坐标作为函数返回值返回。

三．程序源码（C++）

1. 霍夫变换提取直线函数

```
void FindHoughLines(Mat src,vector<Vec4i> &lines,int line_num)
{
    Mat ImgThre;
    threshold(src, ImgThre, 128, 255, THRESH_BINARY); //图像二值化
    Size ImgSize = ImgThre.size();
    vector<Point2i> SelectPoints;
    Point2i tempPoint;
    int px,py;
    int tempPixel;
    float Rmax = 0;
    for (px = 0;px < ImgSize.width; px++)
    {
    for (py = 0;py < ImgSize.height; py++)
    {
        tempPoint.x = px;
        tempPoint.y = py;
        tempPixel = ImgThre.at<uchar>(tempPoint);
        if(tempPixel > 0)
        {
            SelectPoints.push_back(tempPoint); //提取所有白色点的坐标
            if(sqrt(px*px + py*py) > Rmax)
                Rmax = sqrt(px*px + py*py); //计算白色点到原点的最大距离
        }
    }
    }
    int numPoints = SelectPoints.size();//总点数
    int cnt_point;
    int line_angle;
    float line_angle_rad;
```

```
int Rho,tmp;
int Angle_Range = 270; //角度范围-90～+180
Rmax = floor(Rmax);
int R_Range = Rmax;
Mat HoughSpaceImage = Mat::zeros(R_Range,Angle_Range,CV_32FC1);
for(cnt_point = 0; cnt_point < numPoints; cnt_point++)
{
    for(line_angle = 0; line_angle < 180; line_angle++)
    {
        px = SelectPoints[cnt_point].x;
        py = SelectPoints[cnt_point].y;
        //计算每个点、每个角度对应的直线
        line_angle_rad = line_angle*CV_PI/180.0;
        Rho = floor(px*cos(line_angle_rad)+py*sin(line_angle_rad));
        if(Rho > 0)
        {
            tmp = HoughSpaceImage.at<float>(Rho,line_angle+90);
            HoughSpaceImage.at<float>(Rho,line_angle+90) = tmp + 1;
        }
        else //如果出现距离小于零，角度反向
        {
            tmp = HoughSpaceImage.at<float>(-Rho,line_angle-90);
            HoughSpaceImage.at<float>(-Rho,line_angle-90) = tmp + 1;
        }
    }
}
Mat HoughSpaceImage_Show;
HoughSpaceImage.convertTo(HoughSpaceImage_Show,CV_8U);
imshow("HoughSpaceImage_Show",HoughSpaceImage_Show);

double a,b;
Point2i pmin,pmax;
vector<Point2i> EndPoint(4);
vector<Point2i> EndPointSelect;
Vec4i tmpVec4i;
for(int i = 0;i < line_num; )
{
    minMaxLoc(HoughSpaceImage,&a,&b,&pmin,&pmax); //选取像素值最大的点
    Rho = pmax.y; //纵坐标为距离
    line_angle = pmax.x - 90; //横坐标减90为角度
    if(Rho > 0)
    {
        cout << "Rho = " << Rho << endl;
        cout << "line_angle = " << line_angle << endl;
        cout << "NumberofPoints="<<HoughSpaceImage.at<float>(pmax)<<endl;
        i++;
```

```
            if(line_angle%90 !=0) //如果不垂直于X轴或Y轴
            {
                line_angle_rad = line_angle*CV_PI/180.0;
                //计算和XY各两条边界的四个交点
                EndPoint[0].x = 0;
                EndPoint[0].y = Rho/sin(line_angle_rad);
                EndPoint[1].x = ImgSize.width - 1;
                EndPoint[1].y = (Rho -
ImgSize.width*cos(line_angle_rad))/sin(line_angle_rad);
                EndPoint[2].x = Rho/cos(line_angle_rad);
                EndPoint[2].y = 0;
                EndPoint[3].x = (Rho -
ImgSize.height*sin(line_angle_rad))/cos(line_angle_rad);
                EndPoint[3].y = ImgSize.height - 1;
                //选取满足条件的两个端点（满足在图像尺寸以内）
                for(int j = 0; j < 4; j++)
                {
                    if((EndPoint[j].x < ImgSize.width)&&(EndPoint[j].x >= 0))
                    {
                    if((EndPoint[j].y<ImgSize.height)&&(EndPoint[j].y>=0))
                        EndPointSelect.push_back(EndPoint[j]);
                        //储存满足条件的点
                    }
                }
            }
            else if(line_angle == 0)
            //垂直于X轴时，横坐标为原点到直线的距离，纵坐标为两边界
            {
                EndPointSelect[0].x = Rho;
                EndPointSelect[1].x = Rho;
                EndPointSelect[0].y = 0;
                EndPointSelect[1].y = ImgSize.height - 1;
            }
            else if(line_angle == 90)
            //垂直于Y轴时，纵坐标为原点到直线的距离，横坐标为两边界
            {
                EndPointSelect[0].y = Rho;
                EndPointSelect[1].y = Rho;
                EndPointSelect[0].x = 0;
                EndPointSelect[1].x = ImgSize.width - 1;
            }
            //将选取的点存入Vec4i
            tmpVec4i[0] = EndPointSelect[0].x;
            tmpVec4i[1] = EndPointSelect[0].y;
            tmpVec4i[2] = EndPointSelect[1].x;
            tmpVec4i[3] = EndPointSelect[1].y;
            cout << tmpVec4i << endl << endl;
            //添加到lines
            lines.push_back(tmpVec4i);
            EndPointSelect.clear();
        }
        HoughSpaceImage.at<float>(pmax) = 0;
    }
}
```

2. 主函数

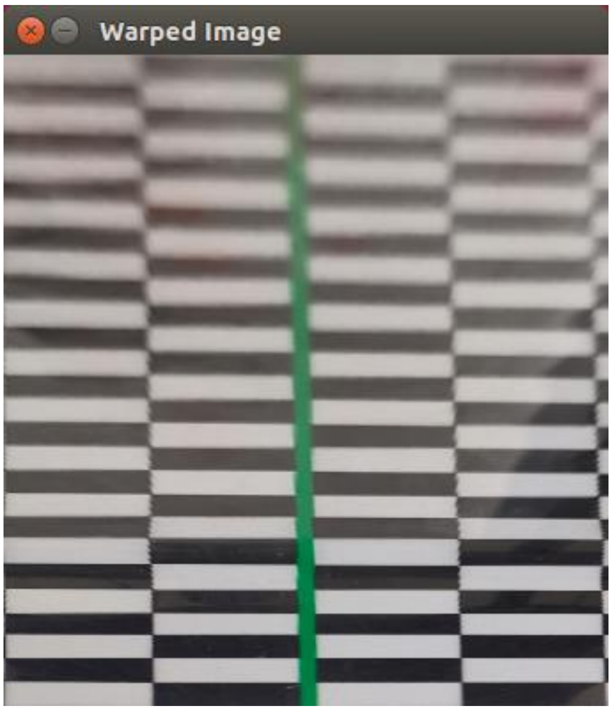
```
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;
static void onMouse(int event, int x, int y, int, void*);
void FindHoughLines(Mat src,vector<Vec4i> &lines,int line_num);

int main()
{
    //读取原图像
    Mat Img_origin;
    Img_origin = imread("/home/thy/Pictures/SrcImg.png");
    imshow("Origin Image",Img_origin);
    //透视变换四个原点、四个目标点
    vector<Point2f> OriginPoints(4);
     vector<Point2f> WarpPoints(4);
    //原图像行数列数
    float OriginCols,OriginRows;
    OriginCols = (float)Img_origin.cols;
    OriginRows = (float)Img_origin.rows;
    //原图像中四个点坐标
    OriginPoints[0] = Point2f(244,189);
    OriginPoints[1] = Point2f(325,187);
    OriginPoints[2] = Point2f(443,528);
    OriginPoints[3] = Point2f(102,527);

    Size WarpImageSize;
    Mat WarpImage;
    Mat PerspectiveMatrix;


    float Length_Left,Length_Right,Length_Top,Length_Bottom;
    float ImageWidth,ImageHeight;
    //上下左右四边长度，选取透视变换后图像的尺寸
    Length_Left = norm(OriginPoints[0] - OriginPoints[3]);
    Length_Right = norm(OriginPoints[1] - OriginPoints[2]);
    Length_Top = norm(OriginPoints[0] - OriginPoints[1]);
```

```
Length_Bottom = norm(OriginPoints[3] - OriginPoints[2]);
ImageWidth = (float)max(Length_Top,Length_Bottom);
ImageHeight = (float)max(Length_Left,Length_Right);
//目标点坐标
WarpPoints[0].x = 0;
WarpPoints[0].y = 0;
WarpPoints[1].x = ImageWidth;
WarpPoints[1].y = 0;
WarpPoints[2].x = ImageWidth;
WarpPoints[2].y = ImageHeight;
WarpPoints[3].x = 0;
WarpPoints[3].y = ImageHeight;
//计算变换矩阵，生成图像
WarpImageSize = Size(cvRound(WarpPoints[2].x), cvRound(WarpPoints[2].y));
PerspectiveMatrix = findHomography(OriginPoints, WarpPoints);
warpPerspective(Img_origin,WarpImage,PerspectiveMatrix,WarpImageSize);
imshow("Warped Image", WarpImage);

//转为HSV
Mat ImgHSV;
cvtColor(WarpImage, ImgHSV, COLOR_BGR2HSV);
vector<Mat> hsvSplit;
split(ImgHSV, hsvSplit);//分离HSV三通道
equalizeHist(hsvSplit[2],hsvSplit[2]);
merge(hsvSplit,ImgHSV);
Mat ImgThresholded;
//筛选绿色，二值化
inRange(ImgHSV, Scalar(35,43,46), Scalar(77, 255, 255), ImgThresholded);
imshow("Green Part", ImgThresholded);

//用库函数寻找图像中的直线
vector<Vec4i> lines;
HoughLinesP(ImgThresholded, lines, 1, CV_PI/180, 80, 20, 20 );
Mat ImgPaint = Img_origin.clone();

Mat Hinv; //变换矩阵的逆矩阵
Hinv = findHomography(WarpPoints, OriginPoints);
Hinv.convertTo(Hinv, CV_32FC1); //转为32位float
Mat LineStart = Mat::ones(3,1,CV_32FC1); //起始点齐次坐标
Mat LineEnd = Mat::ones(3,1,CV_32FC1); //终止点齐次坐标
Mat LineStartOrigin = Mat::zeros(3,1,CV_32FC1);
//起始点齐次坐标（映射回原图）
Mat LineEndOrigin = Mat::zeros(3,1,CV_32FC1);
//终止点齐次坐标（映射回原图）
float k1,k2;
Point Start,End;
```

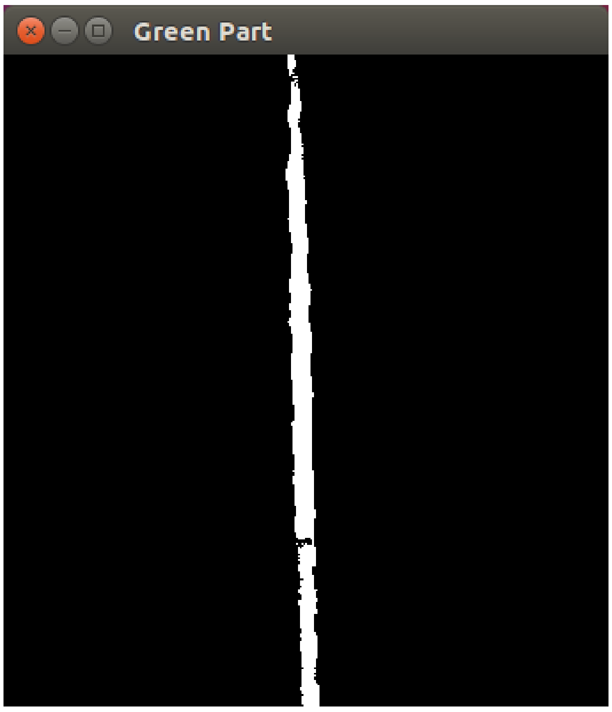
```
    for(int i = 0; i < lines.size(); i++ )
    {
        Vec4i l = lines[i];
        LineStart.at<float>(0,0) = (float)l[0]; //提取Vec4i中元素
        LineStart.at<float>(1,0) = (float)l[1];
        LineEnd.at<float>(0,0) = (float)l[2];
        LineEnd.at<float>(1,0) = (float)l[3];
        LineStartOrigin = Hinv * LineStart;//逆变换
        LineEndOrigin = Hinv * LineEnd;
        k1 = 1/(LineStartOrigin.at<float>(2,0));//计算比例因数
        k2 = 1/(LineEndOrigin.at<float>(2,0));
        Start.x = floor((LineStartOrigin.at<float>(0,0))*k1);
        //还原出在原图上坐标
        Start.y = floor((LineStartOrigin.at<float>(1,0))*k1);
        End.x = floor((LineEndOrigin.at<float>(0,0))*k2);
        End.y = floor((LineEndOrigin.at<float>(1,0))*k2);
        line(ImgPaint, Start, End, Scalar(0,0,255), 1);
    }
    imshow("Image(find lines)(1)", ImgPaint);

    vector<Vec4i> linesNew; //用自定义Hough变换函数寻找10条直线
    FindHoughLines(ImgThresholded,linesNew,10);
    Mat ImgPaint_Mine = Img_origin.clone();
    for(int i = 0; i < linesNew.size(); i++ )
    {
        Vec4i l = linesNew[i];
        LineStart.at<float>(0,0) = (float)l[0];
        LineStart.at<float>(1,0) = (float)l[1];
        LineEnd.at<float>(0,0) = (float)l[2];
        LineEnd.at<float>(1,0) = (float)l[3];
        LineStartOrigin = Hinv * LineStart;
        LineEndOrigin = Hinv * LineEnd;
        k1 = 1/(LineStartOrigin.at<float>(2,0));
        k2 = 1/(LineEndOrigin.at<float>(2,0));
        Start.x = floor((LineStartOrigin.at<float>(0,0))*k1);
        Start.y = floor((LineStartOrigin.at<float>(1,0))*k1);
        End.x = floor((LineEndOrigin.at<float>(0,0))*k2);
        End.y = floor((LineEndOrigin.at<float>(1,0))*k2);
        line(ImgPaint_Mine, Start, End, Scalar(0,0,255), 1);
    }

    imshow("Image(find lines)(2)", ImgPaint_Mine);
    while(waitKey(0)!='q');
    return 1;
}
```

四．实验结果及分析（C++）

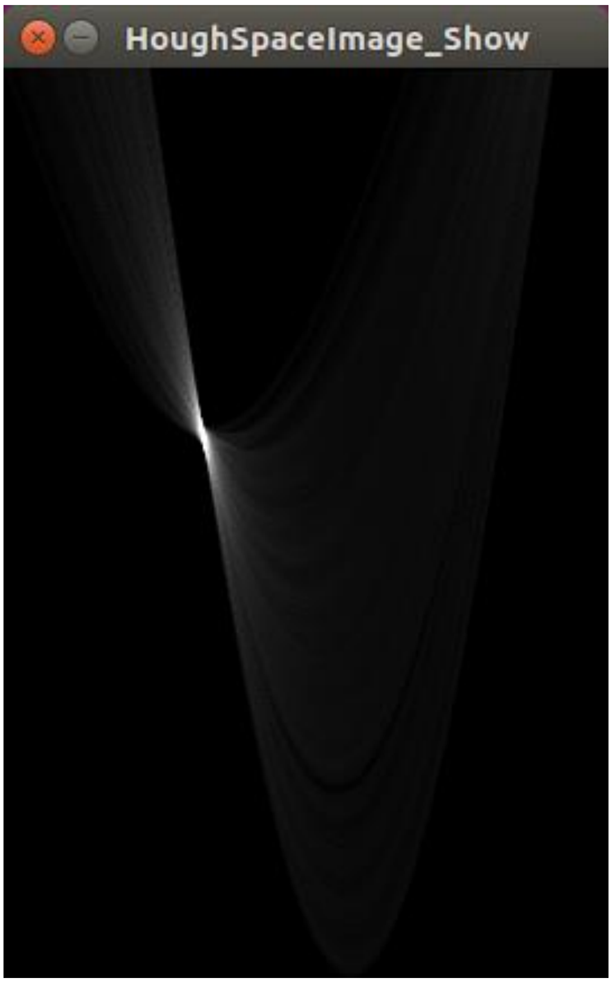
1. 透视变换



2. 提取绿色、阈值化处理

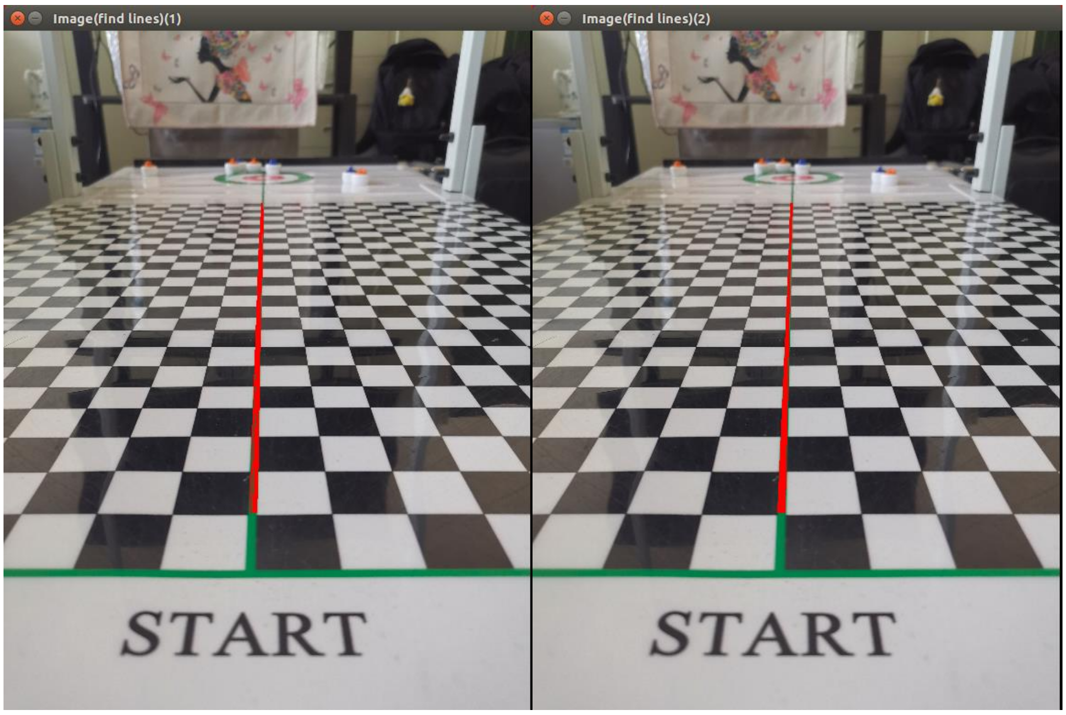


3. 霍夫空间图像



4. 提取直线结果

（1）调用库函数结果　　　　　　（2）自定义霍夫变换结果



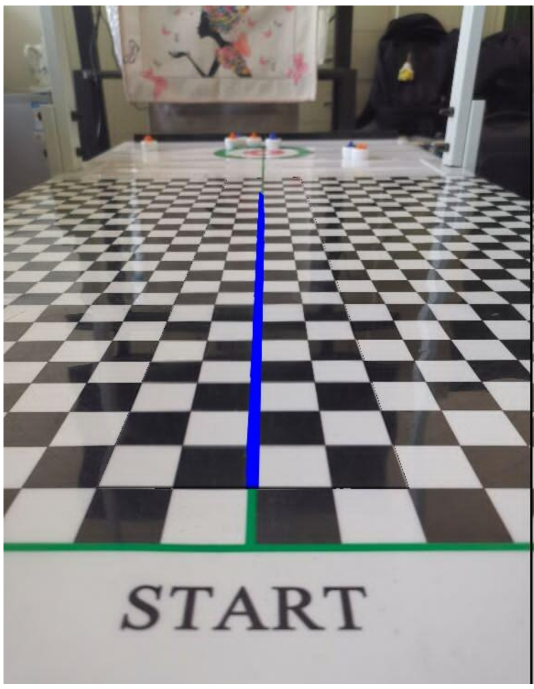
五．程序源码（Python）

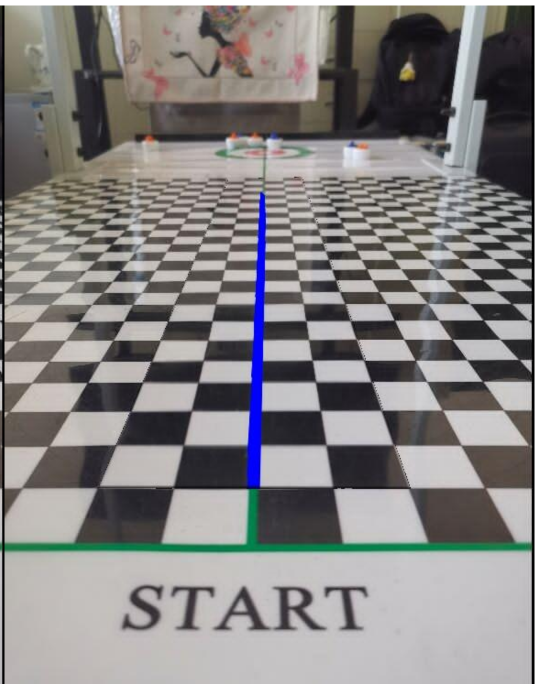
```
import cv2 as cv
import numpy as np
import matplotlib.pylab as plt
import math
from PIL import Image
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def Getlen(point1,point2):
    x1,y1 = point1
    x2,y2 = point2
    distance = math.sqrt((x1-x2)**2+(y1-y2)**2)
    return distance

def imagePers(image):
    #图像透射需要 3*3 矩阵，通过函数 M = cv2.getPerspectiveTransform(pts1,pts2),
    #其中 pts 需要变换前后的 4 个点对应位置。
    #最后通过透射函数 warpPerspective 进行变换
    row,col = image.shape[:2]
    #原图像四点
    orign_point1 = (244,189)
    orign_point2 = (102,527)
    orign_point3 = (443,528)
    orign_point4 = (324,187)
    pos1 = np.float32([orign_point1,orign_point2,orign_point3,orign_point4])
    #目标图像 size
    len_left = Getlen(orign_point1,orign_point2)
    len_right = Getlen(orign_point3,orign_point4)
    len_top = Getlen(orign_point1,orign_point4)
    len_bottom = Getlen(orign_point2,orign_point3)
    image_width = int(max(len_top,len_bottom))
    image_height = int(max(len_left,len_right))
    size = (image_width,image_height)
    #目标点
    warp_point1 = (0,0)
    warp_point2 = (0,image_height)
    warp_point3 = (image_width,image_height)
    warp_point4 = (image_width,0)
    pos2 = np.float32([warp_point1,warp_point2,warp_point3,warp_point4])
    #求取变换矩阵
    M = cv.getPerspectiveTransform(pos1,pos2)
    #透射变换
    dst = cv.warpPerspective(image,M,size)
    return dst,M

def select_green(frame):
    lower_green=np.array([35,43,46])
    upper_green=np.array([77,255,255])
    #转换到 hsv 空间
    hsv = cv.cvtColor(frame, cv.COLOR_BGR2HSV)
    #提取绿色
    mask = cv.inRange(hsv, lower_green, upper_green)
    #cv.imshow('Mask', mask)
    res = cv.bitwise_and(frame, frame, mask=mask)
    #cv.imshow('select_green_Image', res)
    return res,mask

def line_detect_possible_demo(image):
    gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    #边缘检测
    #edges = cv.Canny(gray, 50,150 , apertureSize=3)
    #自动检测可能的直线，返回的是一条条线段
    #lines = cv.HoughLinesP(edges, 1, np.pi/180, 80, minLineLength=20,
maxLineGap=20)
    lines = cv.HoughLinesP(gray, 1, np.pi/180, 80, minLineLength=20,
maxLineGap=20)
    return lines

#霍夫极坐标变换：直线检测
def HTLine (Image,image,stepTheta=1,stepRho=1):
    #宽、高
    rows,cols = image.shape
    #图像中可能出现的最大垂线的长度
    L =  round(math.sqrt(pow(rows-1,2.0)+pow(cols-1,2.0)))+1
    #初始化投票器
    numtheta = int(180.0/stepTheta)
    numRho = int(2*L/stepRho + 1)
    accumulator = np.zeros((numRho,numtheta),np.int32)
    #建立字典
    accuDict={}
    for k1 in range(numRho):
        for k2 in range(numtheta):
            accuDict[(k1,k2)]=[]
    #投票计数
    for y in range(rows):
        for x  in range(cols):
            if(image[y][x] == 255):#只对边缘点做霍夫变换
```

```
                for m in range(numtheta):
                    #对每一个角度，计算对应的 rho 值
                    rho =
x*math.cos(stepTheta*m/180.0*math.pi)+y*math.sin(stepTheta*m/180.0*math.pi)
                    #计算投票哪一个区域
                    n = int(round(rho+L)/stepRho)
                    #投票加 1
                    accumulator[n,m] += 5
                    #记录该点
                    accuDict[(n,m)].append((x,y))
    #计数器的二维直方图方式显示
    rows,cols = accumulator.shape
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    X,Y = np.mgrid[0:rows:1, 0:cols:1]
    surf = ax.plot_wireframe(X,Y,accumulator,cstride=1,
rstride=1,color='gray')
    ax.set_xlabel(u"$\\rho$")
    ax.set_ylabel(u"$\\theta$")
    ax.set_zlabel("accumulator")
    ax.set_zlim3d(0,np.max(accumulator))
    #计数器的灰度级显示
    grayAccu = accumulator/float(np.max(accumulator))
    grayAccu = 255*grayAccu
    grayAccu = grayAccu.astype(np.uint8)
    #只画出投票数大于 60 直线
    voteThresh = 10
    for r in range(rows):
        for c in range(cols):
            if accumulator[r][c] > voteThresh:
                points = accuDict[(r,c)]
                cv.line(Image,points[0],points[len(points)-1],(255),2)
    #cv.imshow('accumulator',grayAccu)
    #cv.imwrite('accumulator.jpg',grayAccu)
    #显示原图
    #cv.imshow("image",Image)
    #plt.show()

    #cv.imwrite('image.jpg',Image)
    return Image

src = cv.imread('666.jpg')
#print(src.shape)
cv.namedWindow("Orignal_Image",cv.WINDOW_AUTOSIZE)
cv.imshow("Orignal_Image",src)

Perspective_image,M_Perspective = imagePers(src)
Selectimage,Selectimage_gray = select_green(Perspective_image)

edge = cv.Canny(Selectimage_gray,50,200)
Perspective_image = HTLine(Perspective_image,edge)

M_inv = np.linalg.inv(M_Perspective)
Image_mark =
cv.warpPerspective(Perspective_image,M_inv,(src.shape[1],src.shape[0]))

for i in range(Image_mark.shape[0]):
    for j in range(Image_mark.shape[1]):
        (b,g,r) = Image_mark[i,j]
        if (b < 10) and (g < 10) and (r < 10):
            Image_mark[i,j] = src[i,j]
#Marked_Image = Image.blend(src,Image_mark,0.3)
cv.namedWindow("result2",cv.WINDOW_AUTOSIZE)
cv.imshow("result2", Image_mark)
cv.imwrite("hough_self0.jpg",Image_mark)
cv.waitKey(0)
cv.destroyAllWindows()
```

六. 实验结果与分析（Python）

（1）调用库函数结果　　　　（2）自定义霍夫变换结果





实验三 特征点检测实验

一．特征点检测方法

（一）SIFT

尺度不变特征转换(Scale-invariant feature transform 或 SIFT)是一种计算机视觉的算法用来侦测与描述影像中的局部性特征，它在空间尺度中寻找极值点，并提取出其位置、尺度、旋转不变量。其应用范围包含物体辨识、机器人地图感知与导航、影像缝合、3D 模型建立、手势辨识、影像追踪和动作比对。

局部影像特征的描述与侦测可以帮助辨识物体，SIFT 特征是基于物体上的一些局部外观的兴趣点而与影像的大小和旋转无关。对于光线、噪声、些微视角改变的容忍度也相当高。基于这些特性，它们是高度显著而且相对容易撷取，在母数庞大的特征数据库中，很容易辨识物体而且鲜有误认。使用 SIFT 特征描述对于部分物体遮蔽的侦测率也相当高，甚至只需要 3 个以上的 SIFT 物体特征就足以计算出位置与方位。在现今的电脑硬件速度下和小型的特征数据库条件下，辨识速度可接近即时运算。SIFT 特征的信息量大，适合在海量数据库中快速准确匹配。

1．特点：

（1）SIFT 特征是图像的局部特征，其对旋转、尺度缩放、亮度变化保持不变性，对视角变化、仿射变换、噪声也保持一定程度的稳定性；

（2）独特性好，信息量丰富，适用于在海量特征数据库中进行快速、准确的匹配；

（3）多量性，即使少数的几个物体也可以产生大量的 SIFT 特征向量；

（4）高速性，经优化的 SIFT 匹配算法甚至可以达到实时的要求；

（5）可扩展性，可以很方便的与其他形式的特征向量进行联合。

2．可以解决的问题：

（1）目标的旋转、缩放、平移

（2）图像仿射/投影变换

（3）光照影响

（4）目标遮挡

（5）杂物场景

（6）噪声

3．步骤：

（1）尺度空间极值检测：搜索所有尺度上的图像位置。通过高斯微分函数来识别潜在的对于尺度和旋转不变的兴趣点。

（2）关键点定位：在每个候选的位置上，通过一个拟合精细的模型来确定位置和尺度。关键点的选择依据于它们的稳定程度。

（3）方向确定：基于图像局部的梯度方向，分配给每个关键点位置一个或多个方向。所有后面的对图像数据的操作都相对于关键点的方向、尺度和位置进行变换，从而提供对于这些变换的不变性。

（4）关键点描述：在每个关键点周围的邻域内，在选定的尺度上测量图像局部的梯度。这些梯度被变换成一种表示，这种表示允许比较大的局部形状的变形和光照变化。

（二）SURF

SURF算法（Speeded Up robust Feature,快速鲁棒特征）是基于SIFT算法（Scale-invariant Feature Transform 尺度不变特征变换）改进来的一个算法，可以做到在图像有一定程度的尺度缩放和方向旋转情况下，依然快速的检测和提取特征。并不影响识别结果。对于亮度变换、仿射变换、噪声也有不错的鲁棒性。SURF 算法主要包含以下几个模块。

（1）建立积分图像。通过计算卷积获得积分图像,使得只需要访问四个点的值就可快速计算一个矩形区域的像素点值的和。

（2）构建尺度空间。这是 SURF 算法具有尺度不变特征的重要原因。通过尺度空间可以提取在尺度变化之后的图像特征，用于后续匹配。

（3）利用 Hessian 算子找特征点,并计算特征点的方向。这是 SURF 算法的特征具有旋转不变形的原因。当特征点带上了方向信息，就可以在图片发生旋转平移变化之后,仍保持稳定的识别效果

（三）Harris

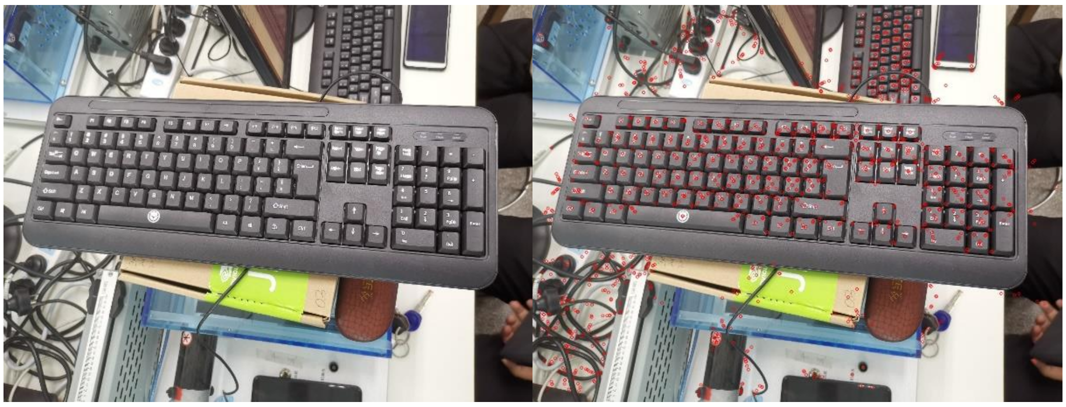
普遍认为，角点是二维图像亮度变化剧烈的点或图像边缘曲线上曲率极大值的点。这些点在保留图像图形重要特征的同时, 可以有效地减少信息的数据量, 使其信息的含量很高, 有效地提高了计算的速度, 有利于图像的可靠匹配, 使得实时处理成为可能。其在三维场景重建、运动估计、目标跟踪、目标识别、图像配准与匹配等计算机视觉领域起着非常重要的作用。

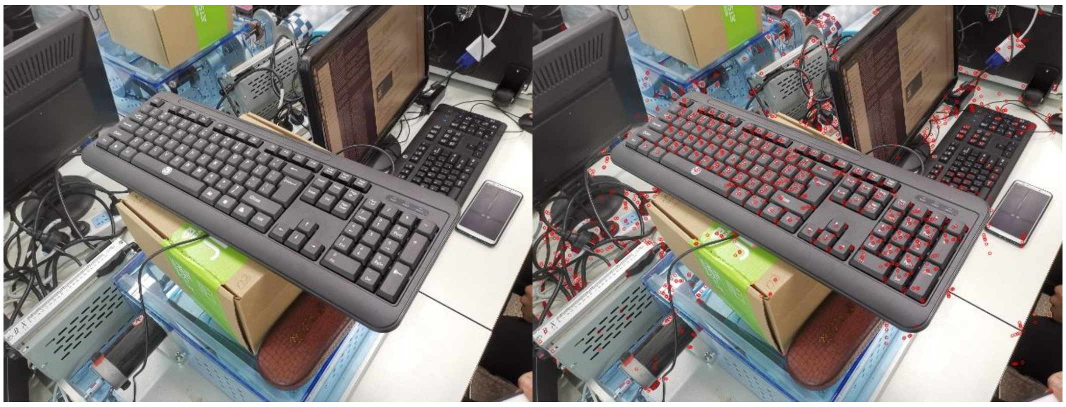
Harris 角点检测算法是 Harris 和 Stephens 于 1988 年在 Moravec 算法的基础上提出基于信号的点特征提取方法，其原理为:如果某一点向任一方向小小偏移都会引起灰度的很大变化, 这就说明该点是角点。考虑的是用一个高斯窗或矩形窗在图像上移动, 由模板窗口取得原图像衍生出 2 ×2 的局部结构矩阵。该方法计算图像 X 方向和 Y 方向上的梯度自相关函数相联系的矩阵 M 及 M 的两个特征值，而矩阵 M 的特征值是自相关函数的一阶曲率，若两个曲率值都很高，则认为该点为角点。

二．程序源码（C++）

```cpp
#include "opencv2/opencv.hpp"
#include "opencv2/core/core.hpp"
#include "opencv2/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/xfeatures2d.hpp"
#include <iostream>

using namespace std;
using namespace cv;
using namespace cv::xfeatures2d;

int main()
{
    Mat Img_origin1,Img_origin2;
    Img_origin1 = imread("/home/thy/Pictures/image05.jpg");
    Img_origin2 = imread("/home/thy/Pictures/image06.jpg");

    Mat Img_gray1,Img_gray2;
    cvtColor(Img_origin1,Img_gray1,CV_RGB2GRAY);
    cvtColor(Img_origin2,Img_gray2,CV_RGB2GRAY);

    int minHessian = 4000;
    Ptr<SURF> detector = SURF::create(minHessian);
    std::vector<KeyPoint> keypoints_1, keypoints_2;
    detector->detect(Img_gray1, keypoints_1);
    detector->detect(Img_gray2, keypoints_2);
    //-- Draw keypoints
    Mat img_keypoints_1; Mat img_keypoints_2;
    drawKeypoints(Img_origin1,keypoints_1,img_keypoints_1, Scalar(0,0,255));
    drawKeypoints(Img_origin2,keypoints_2,img_keypoints_2, Scalar(0,0,255));
    imwrite("/home/thy/Pictures/img_keypoints_1.jpg",img_keypoints_1);
    imwrite("/home/thy/Pictures/img_keypoints_2.jpg",img_keypoints_2);
    namedWindow("Image With KeyPoints 1",0);
    namedWindow("Image With KeyPoints 2",0);
    imshow("Image With KeyPoints 1",img_keypoints_1);
    imshow("Image With KeyPoints 2",img_keypoints_2);

    while(waitKey(0)!='q');
    return 1;
}
```

三．实验结果及分析（C++）



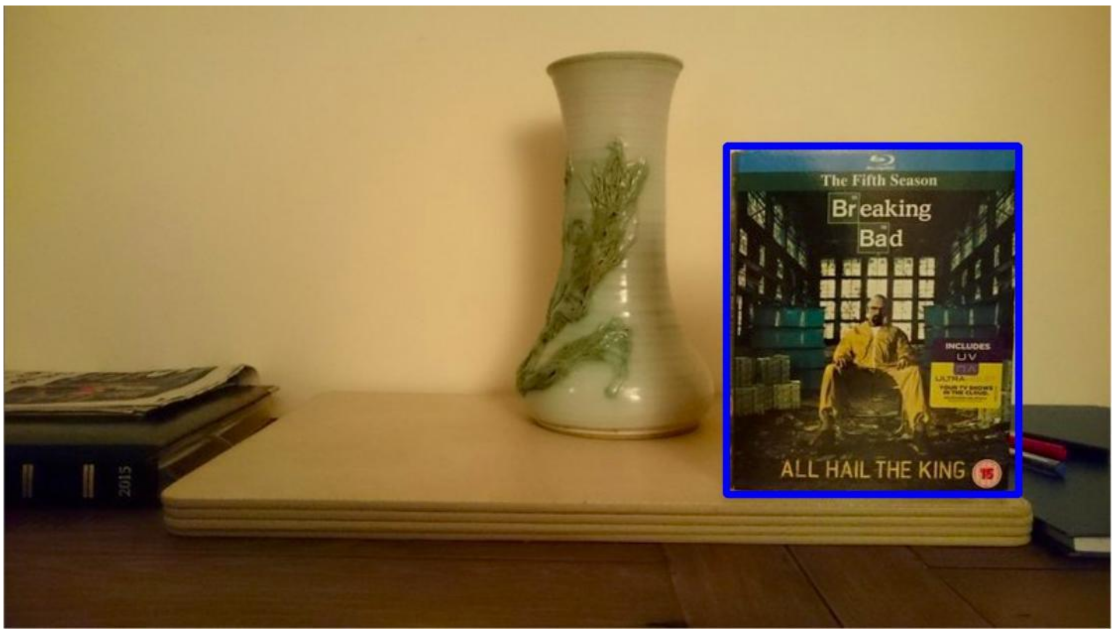


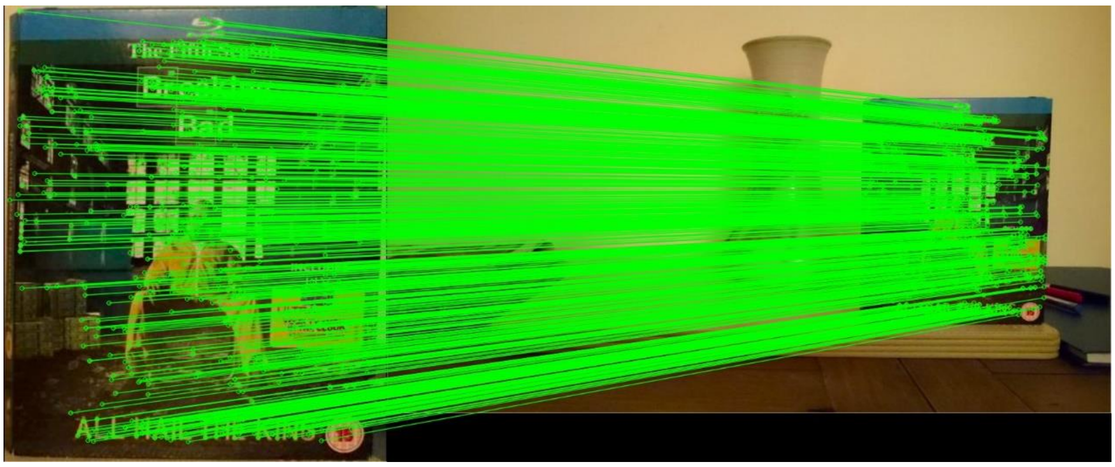
四．程序源码（Python）

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt

MIN_MATCH_COUNT = 10

Image1 = cv.imread('logo.jpg')
Image2 = cv.imread('book.jpg')
img1 = cv.cvtColor(Image1,cv.COLOR_RGB2GRAY)
img2 = cv.cvtColor(Image2,cv.COLOR_RGB2GRAY)

# 使用 SIFT 检测角点
sift = cv.xfeatures2d.SIFT_create()
# 获取关键点和描述符
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# 定义 FLANN 匹配器
index_params = dict(algorithm = 1, trees = 5)
search_params = dict(checks = 50)
flann = cv.FlannBasedMatcher(index_params, search_params)
# 使用 KNN 算法匹配
matches = flann.knnMatch(des1,des2,k=2)
```

```
# 去除错误匹配
good = []
for m,n in matches:
    if m.distance < 0.7*n.distance:
        good.append(m)

# 单应性
if len(good)>MIN_MATCH_COUNT:
    # 改变数组的表现形式，不改变数据内容，数据内容是每个关键点的坐标位置
    src_pts = np.float32([ kp1[m.queryIdx].pt for m in good ]).reshape(-
1,1,2)
    dst_pts = np.float32([ kp2[m.trainIdx].pt for m in good ]).reshape(-
1,1,2)
    # findHomography 函数是计算变换矩阵
    # 参数 cv2.RANSAC 是使用 RANSAC 算法寻找一个最佳单应性矩阵 H，即返回值 M
    # 返回值：M 为变换矩阵，mask 是掩模
    M, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC,5.0)
    print(M)
    # ravel 方法将数据降维处理，最后并转换成列表格式
    matchesMask = mask.ravel().tolist()
    # 获取 img1 的图像尺寸
    h,w = img1.shape
    # pts 是图像 img1 的四个顶点
    pts = np.float32([[0,0],[0,h-1],[w-1,h-1],[w-1,0]]).reshape(-1,1,2)
    # 计算变换后的四个顶点坐标位置
    dst = cv.perspectiveTransform(pts,M)

    # 根据四个顶点坐标位置在 img2 图像画出变换后的边框
    Image2_poly =
cv.polylines(Image2.copy(),[np.int32(dst)],True,(255,0,0),5, cv.LINE_AA)

else:
    print("Not enough matches are found - %d/%d") %
(len(good),MIN_MATCH_COUNT)
    matchesMask = None

# 显示匹配结果
draw_params = dict(matchColor = (0,255,0), # draw matches in green color
                singlePointColor = None,
                matchesMask = matchesMask, # draw only inliers
                flags = 2)
Image3 = cv.drawMatches(Image1,kp1,Image2,kp2,good,None,**draw_params)
cv.imshow( 'Matches',Image3)
cv.imshow('Detect_edge',Image2_poly)

cv.waitKey()
cv.destroyAllWindows()
```

五．实验结果与分析（Python）





实验四 图像匹配实验

一．图像匹配优化求解理论基础

1. 图像特征点匹配方法

特征点的匹配主要有两种方式：Brute-force Matcher（暴力匹配）和 Flann-based Matcher（快速近似最近邻搜索算法）。

暴力方法找到点集 1 中每个 descriptor 在点集 2 中距离最近的 descriptor，找寻到的距离最小就认为匹配，返回距离最近的关键点

FLANN (Fast Approximate Nearest Neighbor Search Library)，快速最近邻逼近搜索函数库。即实现快速高效匹配。特征匹配记录下目标图像与待匹配图像的特征点（KeyPoint），并根据特征点集合构造特征量（descriptor），对这个特征量进行比较、筛选，最终得到一个匹配点的映射集合。我们也可以根据这个集合的大小来衡量两幅图片的匹配程度。

2. 匹配的筛选

根据每一组匹配的点得到所有点对的距离最小值，筛选所有点对，距离小于距离最小值的 2 倍的被认为是有效的点对，用于求解透视变换矩阵。

3. 透视变换矩阵求解优化

筛选后匹配点对的个数多于 4 个，所以存在多解，需要根据一定指标求得最优的变换矩阵。对任何一个变换矩阵，优化的目标函数是该矩阵在变换过程中的错误率，定义为：

$$E=\sum_i\left[\left(x_i'-\frac{h_{11}x_i+h_{12}y_i+h_{13}}{h_{31}x_i+h_{32}y_i+1}\right)^2+\left(y_i'-\frac{h_{21}x_i+h_{22}y_i+h_{23}}{h_{31}x_i+h_{32}y_i+1}\right)^2\right]$$

从所有点对中任取 4 个，共有若干种组合，对每一个组合求得变换矩阵，根据变换矩阵求得错误率，错误率最小的变换矩阵是最优变换矩阵。

二．程序源码（C++）

1. 自定义映射变换矩阵求取函数

```
Mat MyFindHomography(vector<Point2f> &src,vector<Point2f> &dst)
{
    int NumMatches;
    NumMatches = src.size();
    cout << NumMatches << endl;
    int n1,n2,n3,n4; //四个点的序号
    vector<Vec4i> SelectPoints; //选点的组合
    Vec4i tmpVec4i;
    for(n1 = 0; n1 < NumMatches; n1++)
    {
        for(n2 = n1 + 1; n2 < NumMatches; n2++)
        {
            for(n3 = n2 + 1; n3 < NumMatches; n3++)
            {
                for(n4 = n3 + 1; n4 < NumMatches; n4++)
                {
                    tmpVec4i[0] = n1;
                    tmpVec4i[1] = n2;
                    tmpVec4i[2] = n3;
                    tmpVec4i[3] = n4;
                    SelectPoints.push_back(tmpVec4i);
                    //将新的组合存入vector中
                }
            }
        }
    }
    cout << SelectPoints.size() << endl;
    int i;
    vector<Point2f> SrcPt(4);
    vector<Point2f> DstPt(4);
    Mat H, tmpH;
    float Error = 0,ErrorMin = 10000000;
    //当前组合求得的矩阵，最佳矩阵，错误率，最小错误率
    H = Mat::ones(3,3,CV_32FC1);
    //遍历所有组合
    for(i = 0; i < SelectPoints.size(); i++)
    {
        tmpVec4i = SelectPoints[i];
        n1 = tmpVec4i[0];
        n2 = tmpVec4i[1];
        n3 = tmpVec4i[2];
        n4 = tmpVec4i[3];
        SrcPt[0] = src[n1];
        SrcPt[1] = src[n2];
        SrcPt[2] = src[n3];
        SrcPt[3] = src[n4];
        DstPt[0] = dst[n1];
        DstPt[1] = dst[n2];
        DstPt[2] = dst[n3];
        DstPt[3] = dst[n4];
        tmpH = findHomography(SrcPt,DstPt,CV_RANSAC);
        Error = CalculateError(src,dst,tmpH);
        if(Error < ErrorMin)
        {
            H = tmpH;
            ErrorMin = Error;
            cout << ErrorMin << endl;
            //如果当前错误率小于最小错误率，则更新最小错误率和矩阵数值
        }
    }
    return H;
}
```

2. 错误率计算函数

```
float CalculateError(vector<Point2f> &src,vector<Point2f> &dst,Mat H)
{
    int N = src.size();
    int i;
    float Result = 0,Error_x,Error_y,w;
    Mat CoorSrc = Mat::ones(3,1,CV_32FC1);
    Mat CoorTmp;
    for(i = 0;i < N;i++) //遍历所有点
    {
        CoorSrc.at<float>(0,0) = src[i].x;
        CoorSrc.at<float>(1,0) = src[i].y;
        CoorSrc.at<float>(2,0) = 1;
        CoorSrc.convertTo(CoorSrc,CV_32FC1);
        H.convertTo(H,CV_32FC1);
        CoorTmp = H*CoorSrc;
        w = CoorTmp.at<float>(2,0);
        Error_x = dst[i].x - CoorTmp.at<float>(0,0)/w; //x坐标偏差
        Error_y = dst[i].y - CoorTmp.at<float>(1,0)/w; //y坐标偏差
        Result += Error_x*Error_x + Error_y*Error_y; //当前点错误率
    }
    return Result;
}
```

3. 主函数

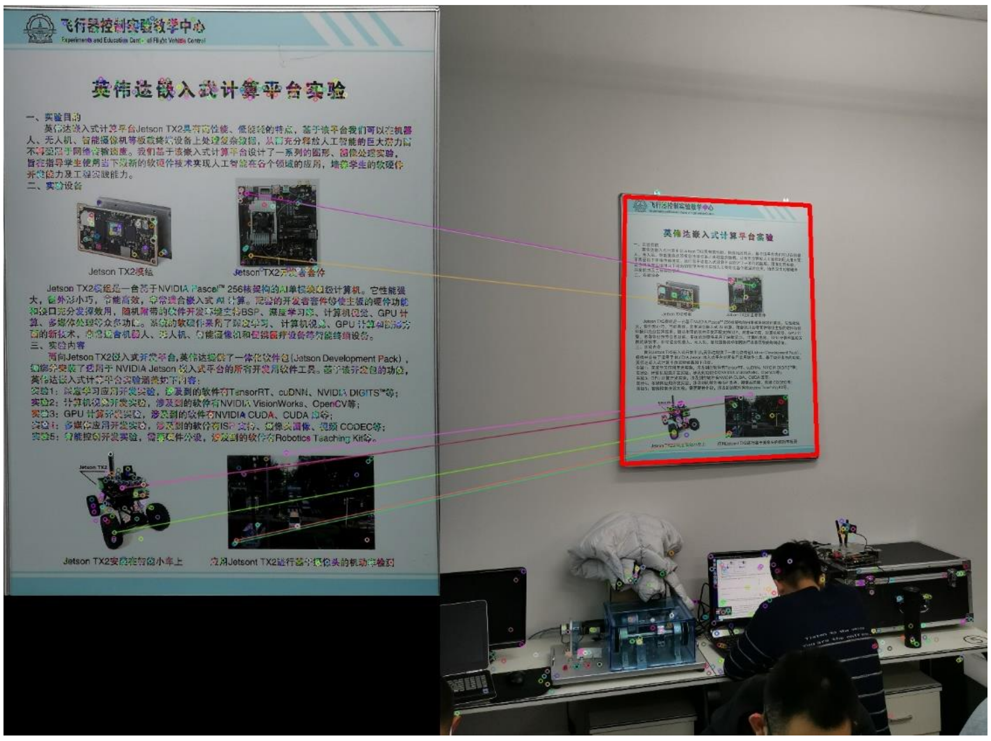
```
#include "opencv2/opencv.hpp"
#include "opencv2/core/core.hpp"
#include "opencv2/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/xfeatures2d.hpp"
#include <iostream>

using namespace std;
using namespace cv;
using namespace cv::xfeatures2d;
Mat MyFindHomography(vector<Point2f> &src,vector<Point2f> &dst);
float CalculateError(vector<Point2f> &src,vector<Point2f> &dst,Mat H);
```

```
int main()
{
    Mat Img_origin1,Img_origin2;
    Img_origin1 = imread("/home/thy/Pictures/image11.jpg");
    Img_origin2 = imread("/home/thy/Pictures/image12.jpg");

    Mat Img_gray1,Img_gray2;
    cvtColor(Img_origin1,Img_gray1,CV_RGB2GRAY);
    cvtColor(Img_origin2,Img_gray2,CV_RGB2GRAY);

    int minHessian = 4000;
    //调用SURF特征点检测器
    Ptr<SURF> detector = SURF::create(minHessian);
    std::vector<KeyPoint> keypoints_1, keypoints_2;
    //识别灰度图像中的特征点
    detector->detect(Img_gray1, keypoints_1);
    detector->detect(Img_gray2, keypoints_2);
    Mat img_keypoints_1; Mat img_keypoints_2;
    //绘制特征点
    drawKeypoints(Img_origin1,keypoints_1, img_keypoints_1, Scalar::all(-1));
    drawKeypoints(Img_origin2,keypoints_2, img_keypoints_2, Scalar::all(-1));
    imwrite("/home/thy/Pictures/img_keypoints_1.jpg",img_keypoints_1);
    imwrite("/home/thy/Pictures/img_keypoints_2.jpg",img_keypoints_2);
    Ptr<SURF>extractor = SURF::create();
    Mat descriptors1, descriptors2;
    extractor->compute(Img_origin1, keypoints_1, descriptors1);
    extractor->compute(Img_origin2, keypoints_2, descriptors2);
    //基于Flann的特征点匹配
    Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("FlannBased");
    std::vector< DMatch > matches;
    matcher->match(descriptors1, descriptors2, matches);
    //求取最小距离
    double min_dist = 2000;
    for( int i = 0; i < descriptors1.rows; i++ )
    {
        double dist = matches[i].distance;
        if( dist < min_dist )
            min_dist = dist;
    }
    //筛选匹配
    std::vector< DMatch > good_matches;
    for( int i = 0; i < descriptors1.rows; i++ )
    {
        if( matches[i].distance < 2*min_dist )
            good_matches.push_back( matches[i]);
    }
    Mat imgMatches;
    drawMatches(Img_origin1,keypoints_1,Img_origin2,keypoints_2,good_matches,
imgMatches);

    vector<Point2f> obj;
    vector<Point2f> scene;
    for (size_t i = 0; i < good_matches.size(); ++i)
    {
        obj.push_back(keypoints_1[ good_matches[i].queryIdx].pt);
        scene.push_back(keypoints_2[ good_matches[i].trainIdx].pt);
    }
    cout << obj.size()  << endl;
    //调用函数求解优化的透视变换矩阵
    Mat H = findHomography( obj, scene, CV_RANSAC );
    //自定义函数求解透视变换矩阵
    Mat H1 = MyFindHomography(obj, scene);
    cout << "H: " << H << endl;
    cout << "H1: " << H1 << endl;
    //待识别目标的四个角点
    vector<Point2f> obj_corners(4);
    obj_corners[0] = cvPoint(0,0);
    obj_corners[1] = cvPoint(Img_origin1.cols,0);
    obj_corners[2] = cvPoint(Img_origin1.cols,Img_origin1.rows);
    obj_corners[3] = cvPoint(0,Img_origin1.rows);
    vector<Point2f> scene_corners(4);
    //将图像映射到场景图像中，并显示匹配结果
    Mat dst1;
    warpPerspective(Img_origin1, dst1, H, Img_origin1.size());
    perspectiveTransform( obj_corners, scene_corners, H);
    namedWindow("WarpPerspective Result01",0);
    imshow("WarpPerspective Result01",dst1);
    imwrite("/home/thy/Pictures/WarpPerspective_Result01.jpg",dst1);
    Mat imgMatches01;
    imgMatches.copyTo(imgMatches01);
    line(imgMatches01, scene_corners[0] + Point2f( Img_origin1.cols, 0),
scene_corners[1] + Point2f( Img_origin1.cols, 0),Scalar(0,0,255),5);
    line(imgMatches01, scene_corners[1] + Point2f( Img_origin1.cols, 0),
scene_corners[2] + Point2f( Img_origin1.cols, 0),Scalar(0,0,255),5);
    line(imgMatches01, scene_corners[2] + Point2f( Img_origin1.cols, 0),
scene_corners[3] + Point2f( Img_origin1.cols, 0),Scalar(0,0,255),5);
    line(imgMatches01, scene_corners[3] + Point2f( Img_origin1.cols, 0),
scene_corners[0] + Point2f( Img_origin1.cols, 0),Scalar(0,0,255),5);
    namedWindow("Match Result01",0);imshow( "Match Result01", imgMatches01);
    imwrite("/home/thy/Pictures/Match_Result01.jpg",imgMatches01);
    warpPerspective(Img_origin1, dst1, H1, Img_origin1.size());
    perspectiveTransform( obj_corners, scene_corners, H1);
    namedWindow("WarpPerspective Result02",0);
    imshow("WarpPerspective Result02",dst1);
    imwrite("/home/thy/Pictures/WarpPerspective_Result02.jpg",dst1);
    Mat imgMatches02;
    imgMatches.copyTo(imgMatches02);
    line( imgMatches02, scene_corners[0] + Point2f( Img_origin1.cols, 0),
scene_corners[1] + Point2f( Img_origin1.cols, 0),Scalar(0,0,255),5);
    line( imgMatches02, scene_corners[1] + Point2f( Img_origin1.cols, 0),
scene_corners[2] + Point2f( Img_origin1.cols, 0),Scalar(0,0,255),5);
    line( imgMatches02, scene_corners[2] + Point2f( Img_origin1.cols, 0),
scene_corners[3] + Point2f( Img_origin1.cols, 0),Scalar(0,0,255),5);
    line( imgMatches02, scene_corners[3] + Point2f( Img_origin1.cols, 0),
scene_corners[0] + Point2f( Img_origin1.cols, 0),Scalar(0,0,255),5);
    namedWindow("Match Result02",0);imshow( "Match Result02", imgMatches02);
    imwrite("/home/thy/Pictures/Match_Result02.jpg",imgMatches02);

    while(waitKey(0)!='q');
    return 1;
}
```

三. 实验结果与分析（C++）

1. 调用库函数结果

（1）特征点匹配与透视变换结果

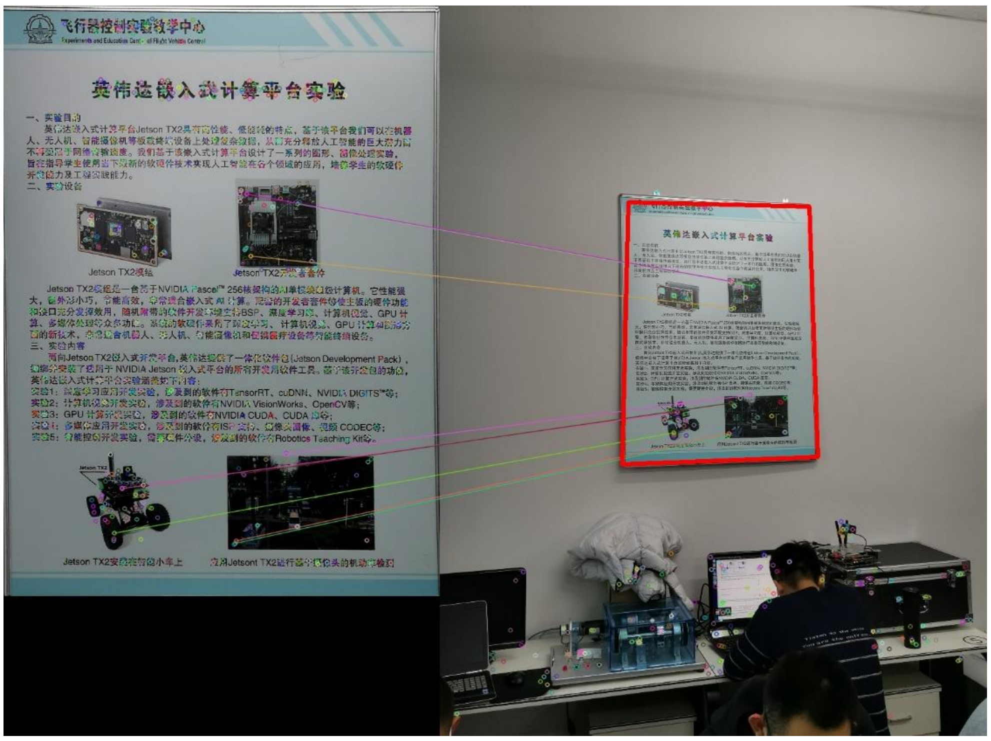


（2）透视变换矩阵求解结果

H: [0.4835133156085311, -0.02073829749311133, 300.6149995555653;
 0.05189759502360707, 0.4114429451983722, 312.2893194104079;
 9.818196405368791e-05, -5.456955389839364e-05, 1]

2. 自定义函数结果

（1） 特征点匹配与透视变换结果



（2）透视变换矩阵求解结果

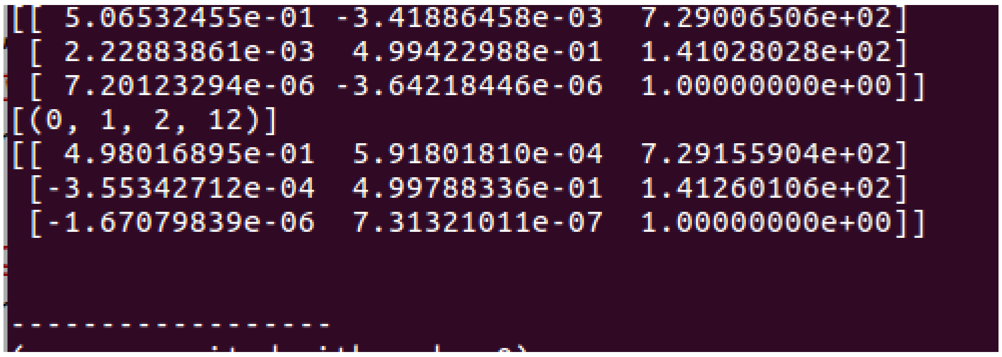
H1: [0.4507654203326859, -0.03455109309002561, 306.8784744029885;
 0.02848531905270124, 0.3826735463474425, 320.3763556842012;
 5.968633476358463e-05, -8.167646369384438e-05, 1]

四. 程序源码

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt

MIN_MATCH_COUNT = 10

def findHomography_self(src_pts, dst_pts):
    row = src_pts.shape[0]
    #print(row)
    Group_Suit = []
    Group_Error = 10000000
    for i in range(0,row-1):
        for j in range(i,row-1):
            for k in range(j,row-1):
                for g in range(k,row-1):
                    Orignalpoint1 =
(src_pts[i][0][0],src_pts[i][0][1])
                    Orignalpoint2 =
(src_pts[j][0][0],src_pts[j][0][1])
                    Orignalpoint3 =
(src_pts[k][0][0],src_pts[k][0][1])
                    Orignalpoint4 =
(src_pts[g][0][0],src_pts[g][0][1])
                    Warppoint1 = (dst_pts[i][0][0],dst_pts[i][0][1])
                    Warppoint2 = (dst_pts[j][0][0],dst_pts[j][0][1])
                    Warppoint3 = (dst_pts[k][0][0],dst_pts[k][0][1])
                    Warppoint4 = (dst_pts[g][0][0],dst_pts[g][0][1])
                    pos1 =
np.float32([Orignalpoint1,Orignalpoint2,Orignalpoint3,Orignalpoin
t4])
                    pos2 =
np.float32([Warppoint1,Warppoint2,Warppoint3,Warppoint4])
                    M_warp = cv.getPerspectiveTransform(pos1,pos2)
                    sum = 0
                    for z in range(0,row-1):
                        sum = sum + ((dst_pts[z][0][0]-
(M_warp[0][0]*src_pts[z][0][0]
                            + M_warp[0][1]*src_pts[z][0][1] +
M_warp[0][2])/
                            (M_warp[2][0]*src_pts[z][0][0]+
M_warp[2][1]*src_pts[z][0][1]
                            + M_warp[2][2]))**2
                            + (dst_pts[z][0][1]-
(M_warp[1][0]*src_pts[z][0][0]
                            + M_warp[1][1]*src_pts[z][0][1] +
M_warp[1][2])/
                            (M_warp[2][0]*src_pts[z][0][0]+
M_warp[2][1]*src_pts[z][0][1]
                            + M_warp[2][2]))**2)
                    if sum < Group_Error:
                        Group_Error = sum
                        Group_Suit.clear()
                        Group_Suit.append((i,j,k,g))
                        M_suit = M_warp
    print(Group_Suit)
    print(M_suit)

Image1 = cv.imread('logo.jpg')
Image2 = cv.imread('book.jpg')
img1 = cv.cvtColor(Image1,cv.COLOR_RGB2GRAY)
img2 = cv.cvtColor(Image2,cv.COLOR_RGB2GRAY)

# 使用 SIFT 检测角点
sift = cv.xfeatures2d.SIFT_create()
# 获取关键点和描述符
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# 定义 FLANN 匹配器
index_params = dict(algorithm = 1, trees = 5)
search_params = dict(checks = 50)
flann = cv.FlannBasedMatcher(index_params, search_params)
# 使用 KNN 算法匹配
matches = flann.knnMatch(des1,des2,k=2)

# 去除错误匹配
good = []
for m,n in matches:
    if m.distance < 0.1*n.distance:
        good.append(m)

# 单应性
if len(good)>MIN_MATCH_COUNT:
    # 改变数组的表现形式，不改变数据内容，数据内容是每个关键点的坐标位置
    src_pts = np.float32([ kp1[m.queryIdx].pt for m in
good ]).reshape(-1,1,2)
    dst_pts = np.float32([ kp2[m.trainIdx].pt for m in
good ]).reshape(-1,1,2)
    # findHomography 函数是计算变换矩阵
    # 参数 cv2.RANSAC 是使用 RANSAC 算法寻找一个最佳单应性矩阵 H，即返回值 M
    # 返回值：M 为变换矩阵，mask 是掩模
    M, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC,5.0)
    print(M)
    findHomography_self(src_pts, dst_pts)
    # ravel 方法将数据降维处理，最后并转换成列表格式
    matchesMask = mask.ravel().tolist()
    # 获取 img1 的图像尺寸
    h,w = img1.shape
    # pts 是图像 img1 的四个顶点
    pts = np.float32([[0,0],[0,h-1],[w-1,h-1],[w-1,0]]).reshape(-
1,1,2)
    # 计算变换后的四个顶点坐标位置
    dst = cv.perspectiveTransform(pts,M)

    # 根据四个顶点坐标位置在 img2 图像画出变换后的边框
    Image2_poly = 
cv.polylines(Image2.copy(),[np.int32(dst)],True,(255,0,0),5, 
cv.LINE_AA)

else:
    print("Not enough matches are found - %d/%d") % 
(len(good),MIN_MATCH_COUNT)
    matchesMask = None

# 显示匹配结果
draw_params = dict(matchColor = (0,255,0), # draw matches in 
green color
                   singlePointColor = None,
                   matchesMask = matchesMask, # draw only inliers
                   flags = 2)
Image3 = 
cv.drawMatches(Image1,kp1,Image2,kp2,good,None,**draw_params)
cv.imshow( 'Matches',Image3)
cv.imshow('Detect_edge',Image2_poly)

cv.waitKey()
cv.destroyAllWindows()
```

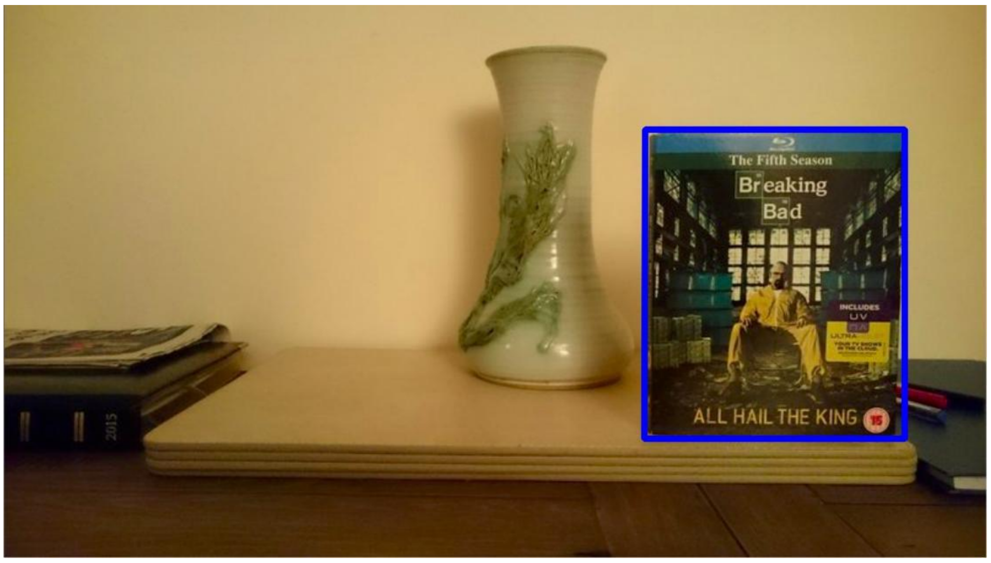
五．实验结果与分析

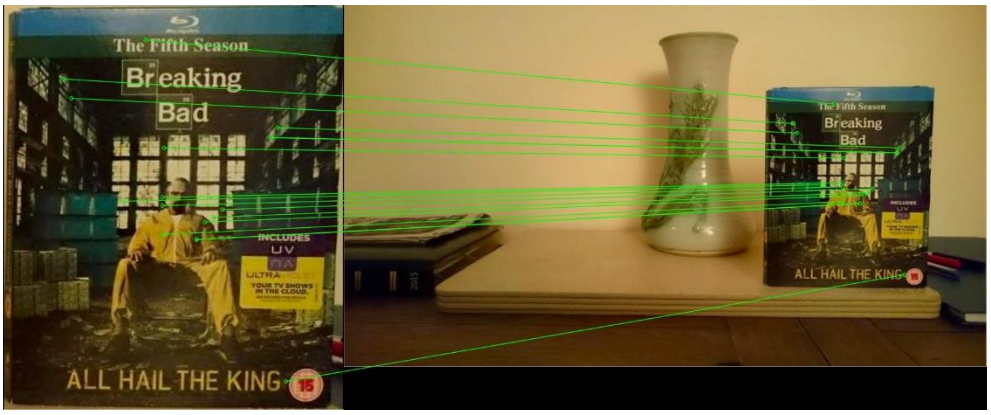
1. 变换矩阵的求取



```
[[ 5.06532455e-01 -3.41886458e-03  7.29006506e+02]
 [ 2.22883861e-03  4.99422988e-01  1.41028028e+02]
 [ 7.20123294e-06 -3.64218446e-06  1.00000000e+00]]
[(0, 1, 2, 12)]
[[ 4.98016895e-01  5.91801810e-04  7.29155904e+02]
 [-3.55342712e-04  4.99788336e-01  1.41260106e+02]
 [-1.67079839e-06  7.31321011e-07  1.00000000e+00]]


-------------------
```

2. 变换结果


The Fifth Season
Breaking Bad
INCLUDES UV
ALL HAIL THE KING


The Fifth Season
Breaking Bad
INCLUDES UV
YOUR TV SHOWS IN THE CLOUD.
ALL HAIL THE KING