山东大学 计算机科学与技术 学院

计算机视觉 课程实验报告

学号：201900161140	姓名：张文浩	10.16
实验题目：图像统计特征		

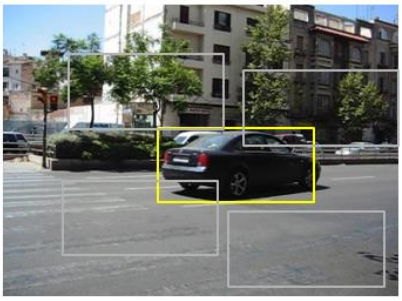
实验过程中遇到和解决的问题：

（记录实验过程中遇到的问题，以及解决过程和实验结果。可以适当配以关键代码辅助说明，但不要大段贴代码。）

基于直方图的目标跟踪

- 实现基于直方图的目标跟踪：已知第t帧目标的包围矩形，计算第t+1帧目标的矩形区域。
- 选择适当的测试视频进行测试：给定第1帧目标的矩形框，计算其它帧中的目标区域。

t



t+1

实验原理：

通过搜索下一帧图像中当前帧的位置周围一定区域的直方图，计算直方图之间的显示度，找到最相近的一个框，就判定为下一帧中目标的矩形区域。

实验步骤：

1. 初始化，启动函数加载视频，并交互框出目标图像

鼠标左键按下开始绘制目标矩形，松开左键绘制结束，关键代码如下：

```
while (true)
{
    if (!leftButtonDownFlag) //判定鼠标左键没有按下，采取播放视频，否则暂停
    {
        video >> image;
    }
    if (!image.data || waitKey(pauseTime) == 27)  //图像为空或Esc键按下退出播放
    {
        break;
    }
    if (originalPoint != processPoint && !leftButtonDownFlag)
    {
        rectangle(image, originalPoint, processPoint, Scalar(0, 255, 0), 2);
    }
    imshow("target", image);
    if (flag == 1) {
        destroyWindow("target");
        break;
    }

}
video.release();
```

2. 绘制 RGB 颜色直方图

调用 opencv 自带的计算直方图的函数，分别计算三个颜色通道的直方图

```
//计算图像的直方图(红色通道部分)
cv::calcHist(&rectImage, nimages, &channels[0], cv::Mat(), outputHist_red, dims, &histSize[0], &ranges[0], uni, accum);
//计算图像的直方图(绿色通道部分)
cv::calcHist(&rectImage, nimages, &channels[1], cv::Mat(), outputHist_green, dims, &histSize[1], &ranges[1], uni, accum);
//计算图像的直方图(蓝色通道部分)
cv::calcHist(&rectImage, nimages, &channels[2], cv::Mat(), outputHist_blue, dims, &histSize[2], &ranges[2], uni, accum);
```

计算完毕，绘制直方图

```
for (int i = 0; i < histSize[0]; i++)
{
    float value_red = outputHist_red.at<float>(i);
    float value_green = outputHist_green.at<float>(i);
    float value_blue = outputHist_blue.at<float>(i);
    //分别画出直线
    cv::line(histPic, cv::Point(i * scale, histSize[0]), cv::Point(i * scale, histSize[0] - value_red * rate_red), cv::Scalar(0, 0, 255));
    cv::line(histPic, cv::Point((i + 256) * scale, histSize[0]), cv::Point((i + 256) * scale, histSize[0] - value_green * rate_green), cv::Scalar(0, 255, 0));
    cv::line(histPic, cv::Point((i + 512) * scale, histSize[0]), cv::Point((i + 512) * scale, histSize[0] - value_blue * rate_blue), cv::Scalar(255, 0, 0));
}
cv::imshow("histgram", histPic);
```

3. 因为采用 HSV 颜色格式最终效果比 RGB 的效果好，所以现在将图像的 RGB 颜色格式换成 HSV 的颜色格式。

在 HSV 颜色格式下，再次计算刚才抠出来的目标矩形的直方图，方便后续计算相似度

```
//进行原图直方图的计算
calcHist(&srcHsvImage, 1, channels, Mat(), srcHist, 2, histSize, ranges, true, false);
```

对直方图做归一化处理

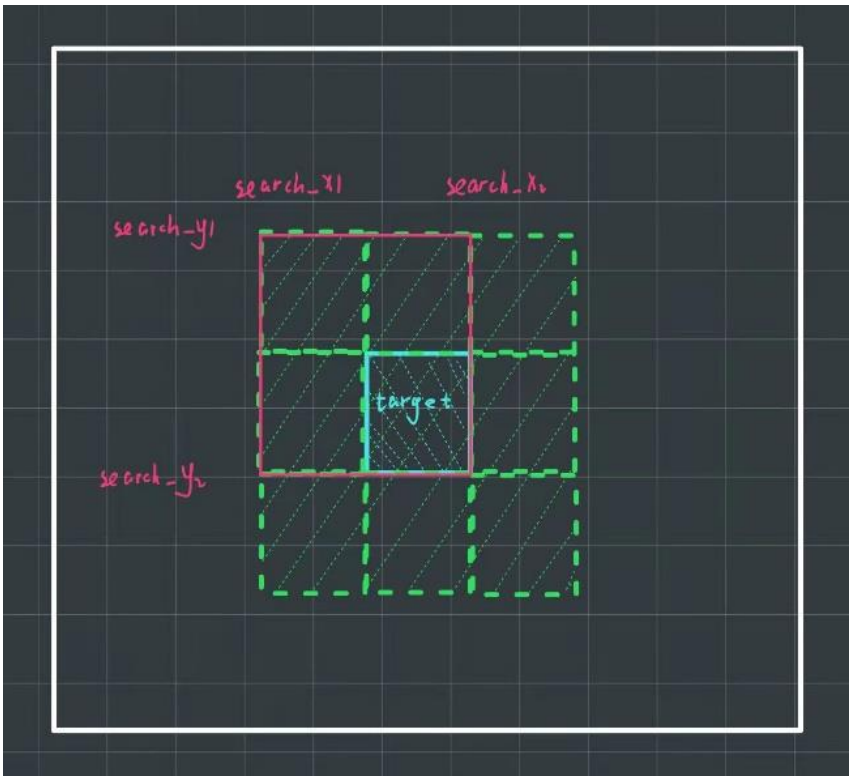
```
//归一化
normalize(srcHist, srcHist, 0, 1, NORM_MINMAX);
```

4. 扫描搜索区域，绘制目标矩形。

首先确定搜索范围，如果搜索范围太大，就会增大计算量，造成视频卡顿；如果搜索范围太小，在目标物体移动太快的时候无法追踪到物体。

经过多次尝试，最终将搜索范围定为了目标矩形的三倍大小

```
//目标搜索区域设定为原区域的周围且面积为原来三倍
int search_x1 = originalPoint.x - width;
int search_x2 = originalPoint.x + width;
int search_y1 = originalPoint.y - height;
int search_y2 = originalPoint.y + height;
//越界检查
```



如原理图所示，利用两个 for loop 遍历粉色的搜索区域

在两个 for loop 中,每次计算搜索到的矩形区域的直方图,利用函数 compHist 计算当前搜索的矩形区域与目标区域的相似性，cursim 越小，说明越相似，最后找到最相似的矩形区域，作为下一帧的目标区域。

```
//抠出当前搜索到的图像
Mat image_compare;
image_compare = image(Rect(preStart, preEnd));
double cursim = compHist(srcHist, image_compare);
//计算当前搜索图像与抠出图像的相似性
if (minsim > cursim) {  //如果当前搜索图像的相似性更高，就暂时把当前搜索图像作为下一帧的显示图像，直到搜索到更相似的
    get1 = preStart;
    get2 = preEnd;
    minsim = cursim;
}
```

在原始视频图像上刷新矩形,只有当与目标直方图很相似时才更新起点搜索区域，满足目标进行移动的场景

```
//在原始视频图像上刷新矩形，只有当与目标直方图很相似时才更新起点搜索区域，满足目标进行移动的场景
if (minsim < 0.3) {
    search_x1 = get1.x - width;
    search_x2 = get1.x + width;
    search_y1 = get1.y - height;
    search_y2 = get1.y + height;
```

绘制最相似的矩形区域

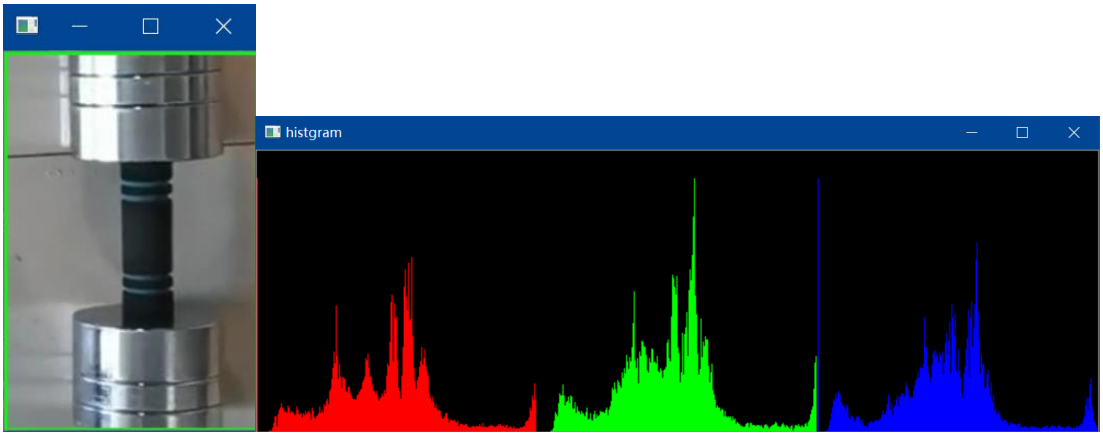
```
//绘制矩形
if (minsim < 0.5)
    rectangle(image, get1, get2, Scalar(0, 0, 255), 2);
```

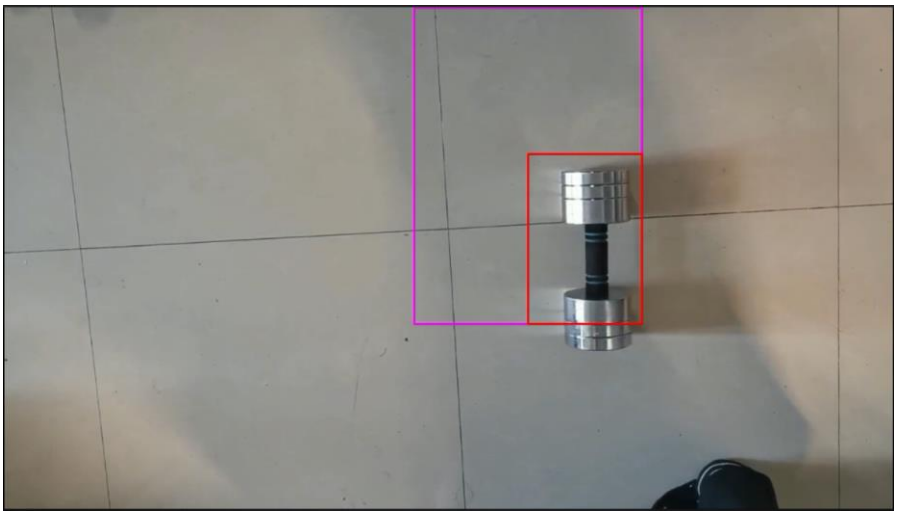
其中计算两个直方图的显示度的函数 compHist 实现如下；

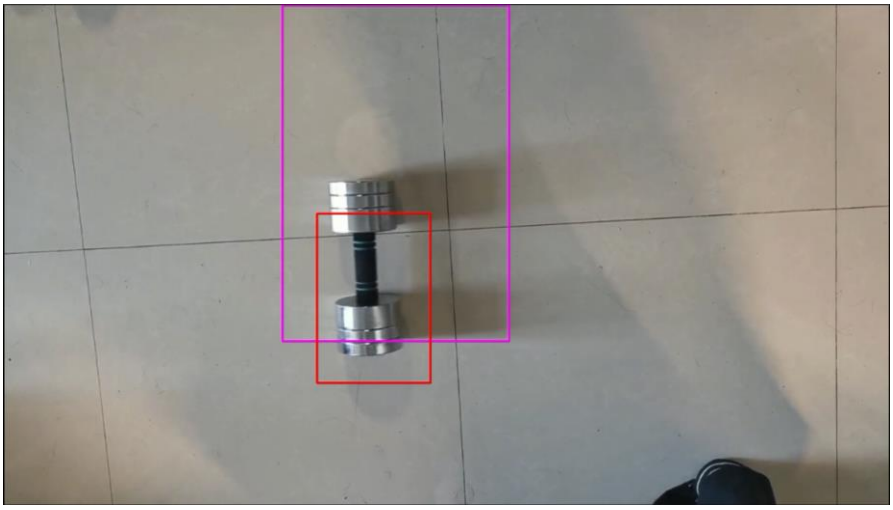
现将 RGB 转换为 HSV 格式，然后计算直方图，也进行归一化操作，然后调用 opencv 提供的 compareHist 函数，采用巴氏距离比较两个直方图的相似性。

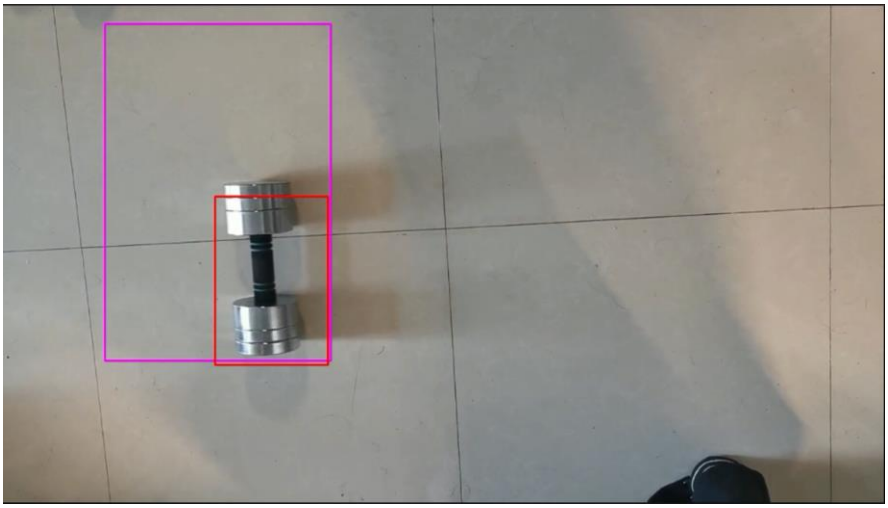
```
double compHist(const MatND srcHist, Mat compareImage)
{
    //在比较直方图时，最佳操作是在HSV空间中操作，所以需要将RGB空间转换为HSV空间
    Mat compareHsvImage;
    cvtColor(compareImage, compareHsvImage, CV_BGR2HSV);
    //采用H-S直方图进行处理
    //首先得配置直方图的参数
    MatND  compHist;
    //进行原图直方图的计算

    //对需要比较的图进行直方图的计算
    calcHist(&compareHsvImage, 1, channels, Mat(), compHist, 2, histSize, ranges, true, false);
    //注意：这里需要对两个直方图进行归一化操作

    normalize(compHist, compHist, 0, 1, NORM_MINMAX);
    //对得到的直方图对比
    double g_dCompareRecult = compareHist(srcHist, compHist, 3);//3表示采用巴氏距离进行两个直方图的比较
    return g_dCompareRecult;
}
```

实验结果：

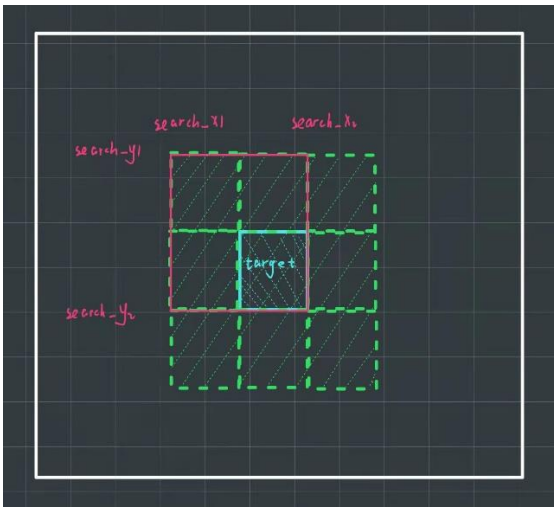








其中红色的矩形是跟踪的物体，紫色的矩形就是搜索范围矩形的左上角的范围，即下面这个图里面的粉色的矩形



这个视频放在了附件中

另一个视频：

结果分析与体会：

在本次实验中，通过实现基于直方图的目标跟踪进行了对直方图的应用。总结一下实验中遇到的几个问题：

1. 搜索的时候不能在整张图像上进行搜索，不然会导致计算过量太大，造成视频卡顿，所以我们智能假设物体不会瞬移，即下一帧中的物体不会在距离当前帧物体距离很远的地方出现。于是我们在搜索的时候就可以只在当前物体位置的周围进行搜索，这个范围大小也算是一个超参，如果搜索范围太大，就会增大计算量，造成视频卡顿；如果搜索范围太小，在目标物体移动太快的时候无法追踪到物体。

2. 在原始图像上刷新矩阵的问题，因为我们只在物体周围一定范围内进行搜索，所以显然这个搜索矩阵一定是要进行移动的，而移动的时机是当搜索到的目标

矩形与目标非常相关时，才更新搜索矩形，这样可以减少噪声的干扰，增加稳定性。但我在实验的过程遇到的问题是，一开始把判定“非常相关”的阈值设置得太苛刻了，所以搜索矩阵始终不移动，等到物体超出搜索矩阵的时候，搜索矩阵还停留在原地，导致“跟丢”物体。于是我把这个阈值设置得大一点后就解决了这个问题。

3. 计算速度的问题：整个计算过程中最消耗时间的就是在搜索过程中的两个 for loop，所以可以考虑减少循环的次数来加快速度。因为视频中一帧图像的大小通常比较大，都是几百大小的，所以个位数的误差可以忽略不计，我们可以利用这一点，再循环的时候不是每次移动一个像素，而是移动多个像素，即增大补偿 step，这样可以明显的减少循环次数，同时对效果影响不大。