

滚动字幕的检测与拼接

刘哲泽 大三 南开大学

一、实验要求

1. 在一段视频中，对于一个给定的字幕区域，判断其是否为滚动字幕。
2. 对于滚动字幕，将其拼接为完整的字幕条。
3. 对于固定字幕，直接将其输出。

我借助了 opencv 完成了实验（opencv 4.9.0，Visual Studio 2022，windows 11）。

二、实验内容

（一）判断是否为滚动字幕

1. 确定字幕区域

根据图片，人工确定[x,y,width,height]，以确定字幕区域。

2. 滚动字幕判断

我使用了一个启发式的判断方法，来区分滚动字幕和固定字幕。该步骤在第（二）节第 3 小节得到 good_matches 之后。

首先定义一个 map，它的 key 值为两点间的 L1 距离，value 值为出现次数。

```
1. map<double, int> count;//L1 距离 出现次数
```

然后，遍历 good_matches，计算 match 的两个点间的 L1 距离，并加入 map 中：

```
2. for (int i = 0; i < good_matches.size(); i++) {
3.     .....
4.     int x1 = kps1[good_matches[i].queryIdx].pt.x;
5.     int y1 = kps1[good_matches[i].queryIdx].pt.y;
6.
7.     int x2 = kps2[good_matches[i].trainIdx].pt.x;
8.     int y2 = kps2[good_matches[i].trainIdx].pt.y;
9.
10.    int L1 = abs(x1 - x2) + abs(y1 - y2);
11.
12.    if (count.find(L1) == count.end()) {
13.        //key 不存在
14.        count.insert(pair<double, int>(L1, 1));
15.    }
16.    else {
17.        map<double, int>::iterator it = count.find(L1);
```

```

18.     it->second = it->second + 1;
19. }
20. }

```

接着，根据 value 值，对 map 从大到小排序：

```

21.     vector< pair<double, int> > vec;
22.     for (map<double, int>::iterator it = count.begin(); it != count.end(); it++) {
23.         vec.push_back(pair<double, int>(it->first, it->second));
24.     }
25.     sort(vec.begin(), vec.end(), [](pair<double, int>a, pair<double, int>b) {return a.second > b.second; });

```

最后，判断出现次数最大的 L1 距离，是否小于 60（经验值），如果小于，判断为固定字幕，退出；否则，认定为滚动字幕，继续接下来的步骤。vec.size() == 0 是为了避免 vec 为空而出现异常。

```

/*=====判断是否为滚动字幕=====*/
26.     if (vec.size() == 0 || vec[0].second < 60) {
27.         // 根据相似点匹配数量进行判断
28.         cout << endl;
29.         cout << "===== " << endl;
30.         cout << "             不是滚动字幕" << endl;
31.         cout << "===== " << endl;
32.         Mat res;
33.         vconcat(image2, image1, res);
34.         imwrite(output_fix, res);
35.         return;
36.     }
37.     else {
38.         cout << endl;
39.         cout << "===== " << endl;
40.         cout << "             是滚动字幕" << endl;
41.         cout << "===== " << endl;
42.     }

```

我选择这种启发式判断方法的依据是，滚动字幕匹配到的特征点，具有平移关系，所以具有相同 L1 距离的滚动字幕的匹配特征点的数量应该比固定字幕的多。

（二）滚动字幕的拼接

我自定义一个函数用于滚动字幕的拼接，其中有四个参数：

- string left: 要拼接的左边图的地址
- string right: 要拼接的右边图的地址

- Rect roi: 字幕区域
- bool processed: 右图是否已经根据 roi 裁剪过（这是为了能够连续拼接多张图）

```
1. void mystitch(string left, string right, Rect roi, bool processed = false)
```

接下来对该函数的实现进行介绍：

1. 读取图片

设置保存地址，如果为滚动字幕，存储在 output_roll 中；反正存储在 output_fix 中；接着进行对图片的预处理操作，首先使用 imread 读取图片，然后对右图进行裁剪得到字幕区域；根据传入的 processed，判断左图是否需要裁剪，并根据判断结果进行不同操作。注意，最终 image1 存储的是右图的字幕区域。image2 存储的是左图的字幕区域。

```
1. string output_roll = "result/roll.jpg";
2. string output_fix = "result/fix.jpg";
3.
4. // [预处理]
5. Mat src1 = imread(left);
6. Mat src2 = imread(right);
7.
8. /*
9.  * 提取 ROI 区域
10.  * x y width height
11.  */
12. // 获取 ROI 区域，这是要拼接的滚动字幕区域
13. Mat image1 = src2(roi); // 右图
14. Mat image2; // 左图
15. if (processed) {
16.     image2 = src1;
17. }
18. else {
19.     // 如果传入的是未处理的原始图像，直接赋值
20.     // （为了能够多次拼接）
21.     image2 = src1(roi);
22. }
23.
24. imwrite("result/right.jpg", image1);
25. imwrite("result/left.jpg", image2);
```

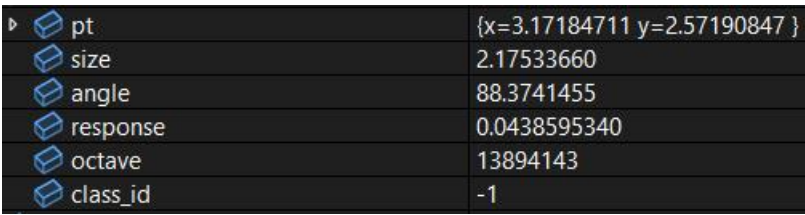
2. 计算特征点和描述子¹

我使用了 SIFT 算法进行图片特征点的检测，得到图片 image1 和 image2 的特征点集合 kps1 和 kps2。SIFT 检测器得到的 KeyPoint 的内容如下：

➤ 关键点的坐标位置 (x, y)

¹ 该小节和第 3 小节内容的原理来自 D.Lowe 的论文 Distinctive Image Features from Scale-Invariant Keypoints，本人未阅读，在此仅作为说明。

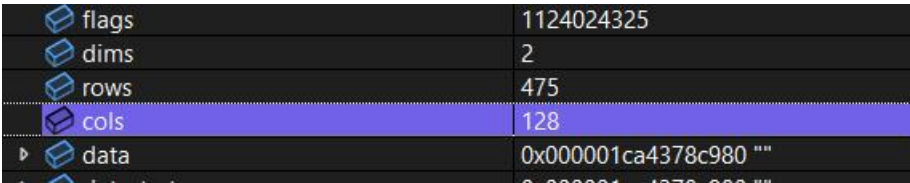
- 关键点的尺度大小
- 关键点的方向角度（为每个特征点指定方向，以实现图像旋转的不变性）
- 关键点的响应值（表示关键点的显著性），SIFT 默认以 0.03 作为阈值
- 其他可能的属性或描述符



pt	{x=3.17184711 y=2.57190847 }
size	2.17533660
angle	88.3741455
response	0.0438595340
octave	13894143
class_id	-1

图 1 KeyPoint 的值

得到特征点后，就可以进行特征描述子的计算。在特征点（KeyPoint）周围选取 16x16 的邻域。将它为 16 个 4x4 大小的子块。对于每个子块，创建包含 8 个 bin²的方向直方图。因此总共有 128 个 bin 值可用（如下图中的 cols = 128）。由这 128 个形成的向量构成了特征点描述子。如下所示，特征描述子记录了特征点的一些信息，比如“rows = 475”表示共有 475 个特征点。



flags	1124024325
dims	2
rows	475
cols	128
data	0x000001ca4378c980 ""

图 2 特征描述子部分内容

```
1. // [SIFT]
2. // 检测特征点
3. Ptr<SIFT> detector = SIFT::create();
4. vector<KeyPoint> kps1, kps2;
5. detector->detect(image1, kps1);
6. detector->detect(image2, kps2);
7.
8. // 计算特征描述符
9. Mat descriptors1;
10. Mat descriptors2;
11. detector->compute(image1, kps1, descriptors1);
12. detector->compute(image2, kps2, descriptors2);
13. // 绘制特征点
14. //draw_feature(image1, kps1, descriptors1);
15. //draw_feature(image2, kps2, descriptors2);
16. // [SIFT]
```

3. 匹配特征

² bin 值指的是方向直方图中每个柱子的值，用来表示该子块内的梯度方向信息。

使用 `opencv` 的 `BruteForce` 匹配器进行特征匹配。其中的 `knnMatch` 函数，对于每个查询特征描述子，它会计算与其最接近的 `k` 个训练特征描述子，并返回这 `k` 个最近邻的距离。这样可以得到更多的匹配结果，但也可能包含一些不太准确的匹配。（使用 `knnMatch` 函数时，设置 `k=2`，然后可以计算最近距离与第二最近距离的比率。如果比率大于 `0.8`，则忽略它们。根据脚注 1，这样做消除了大约 `90%` 的错误匹配，而同时只去除了 `5%` 的正确匹配）

```
1.  // [BruteForce Matcher]
2.  // 创建特征点匹配器
3.  Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("BruteForce");
4.  // 进行特征匹配
5.  /*
6.  class DMatch{
7.  ...
8.  int queryIdx; // 特征点在第一幅图像中的索引
9.  int trainIdx; // 特征点在第二幅图像中的索引
10. float distance; // 两个特征点之间的距离
11. };
12. */
13. vector<vector<DMatch>> matches;
14. vector<DMatch> good_matches;
15. //matcher->match(descriptors1, descriptors2, matches); //vector<DMatch> matches
16. matcher->knnMatch(descriptors1, descriptors2, matches, 2); //k=2
17.
18. // Lowe's algorithm, 获取优秀匹配点
19. for (int i = 0; i < matches.size(); i++)
20. {
21.     if (matches[i][0].distance > 0.8 * matches[i][1].distance)
22.     {
23.         //最近距离与第二最近距离的比率，大于 0.8，则忽略它们
24.         //消除了大约 90% 的错误匹配，而同时只去除了 5% 的正确匹配
25.     }
26.     else {
27.         good_matches.push_back(matches[i][0]);
28.     }
29. }
```

4. 根据匹配到的特征，计算单应性矩阵

首先，得到源点和目的点：

```
1.  vector<Point2f> srcPoints, dstPoints;
2.  for (int i = 0; i < good_matches.size(); i++) {
3.      srcPoints.push_back(kps1[good_matches[i].queryIdx].pt);
4.      dstPoints.push_back(kps2[good_matches[i].trainIdx].pt);
```

```
5. }
```

然后使用 `opencv` 的 `RANSAC` 算法计算单应性矩阵，用于透视校正或图像对齐：

```
1. Mat H = findHomography(srcPoints, dstPoints, RANSAC);
```

5. 对图片进行透视变换

首先，使用 `opencv` 的 `warpPerspective` 方法对右侧图像（`image1`）进行透视变换，设置变换后的图片宽度为原始图像的两倍，高度相同：

```
1. //对图像进行透视变换
2. Mat imageTransform1, imageTransform2;
3. //warpPerspective(image1, imageTransform1, H, Size(image1.cols + image2.cols, image2.
   rows));
4. warpPerspective(image1, imageTransform1, H, Size(image1.cols + image2.cols, image2.ro
   ws));
```

得到的某个右图输出如下所示：



图 3 透视变换 1

然后，将左侧图像拷贝到透视变换的图像中：

```
1. //图像拷贝
2. //创建拼接后的图,需提前计算图的大小
3. int dst_width = imageTransform1.cols; //取最右点的长度为拼接图的长度
4. int dst_height = image2.rows;
5.
6. Mat dst(dst_height, dst_width, CV_8UC3);
7. dst.setTo(0);
8.
9. imageTransform1.copyTo(dst(Rect(0, 0, imageTransform1.cols, imageTransform1.rows)));
10. image2.copyTo(dst(Rect(0, 0, image2.cols, image2.rows)));//复制到左边
```

得到的图像如下所示：



图 4 透视变换 2

注意到右侧有黑边，为了美观也为了能够拼接多个滚动字幕，我进行了消除黑色区域的操作。

6. 其他操作

(1) 去除右侧黑色区域

定义了一个函数 `delete_black`，用于消除右侧的黑色区域，如下所示，先将 RGB 图像转为灰度图像，然后再转为二值图像，接着我利用 `opencv` 自带的函数 `boundingRect` 进行分割，得到一个矩形框，该矩形块包含的是非黑色区域内容，如图 5 所示。接着只保留矩形框区域即可，最终得到的图如图 6 所示。

(Sample1/016660.jpg Sample1/016717.jpg 165, 520, 550, 50)

```
1. void delete_black(Mat& image) {
2.     // 去除右侧黑边
3.
4.     int rs = image.rows;
5.     int cs = image.cols;
6.
7.     try {
8.         // 转换为灰度图像
9.         Mat grayImage;
10.        cvtColor(image, grayImage, cv::COLOR_BGR2GRAY);
11.
12.        // 阈值处理
13.        Mat binaryImage;
14.        threshold(grayImage, binaryImage, 1, 255, cv::THRESH_BINARY);
15.
16.        // 使用 boundingRect 确定黑边范围
17.        Rect boundingRect = cv::boundingRect(binaryImage);
18.
19.        // // 在原始图像上绘制边界矩形
20.        // rectangle(image, boundingRect, cv::Scalar(0, 255, 0), 2);
21.
22.        image = image(boundingRect);
23.        // show(image);
24.
25.        // cout << rs << " " << cs << endl;
26.        // cout << image.rows << " " << image.cols << endl;
27.    }
28.    catch (Exception e) {
29.        cout << e.err;
30.    }
31.
32. }
```

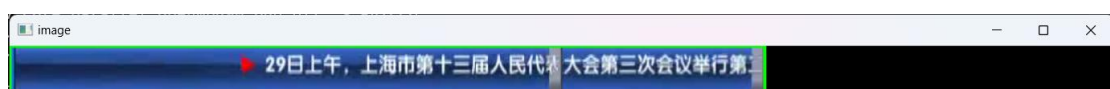


图 5 矩形框



图 6 拼接结果

去除了黑框以后，就能进行以拼接结果作为左图，新的图作为右图，进行连续拼接了。（如图 7 所示）

（图 6 Sample1/016816.jpg 165, 520, 550, 50）

```
1. //连续拼接
2. string left = "result/roll.jpg";
3. string right = "Sample1/016816.jpg";
4.
5. mystitch(left, right, roi[0], true);
```



图 7 多次拼接结果

三、测试

（一）测试 1：滚动字幕



图 8 left



图 9 right



图 10 合并

（二）测试 2：固定字幕

这两个数据是玩自己创建³。



图 11 left



图 12 right

判定为固定字幕，我将两个图片上下拼接，然后保存拼接后的图片。

³<https://tv.cctv.com/2024/05/01/VIDEyGVj8XoEEclFwer6XKy5240501.shtml?spm=C45404.P9Z56wAyBcZ2.E8JIE0NUktSy.2> 中的 2:51 和 3:11 的截图。

多项举措巩固增强经济回升向好态势 三大动力源地区经济较快增长

图 13 固定

(三) 测试 3: 多个滚动字幕拼接



图 14 left



图 15 right1



图 16 right2



图 17 合并

(四) 测试 4: 距离很近的滚动字幕, 判断为 是滚动字幕



图 18 left



图 19 right



图 20 合并

四、存在问题

(一) 测试 5: 距离很远的滚动字幕, 判断为 不是滚动字幕

我的算法, 会将此种情况的字幕, 判断为固定字幕。



图 21 left



图 22 right

判断为固定字幕:



图 23 固定

(二) 测试 6: 两个不同帧但相同的固定字幕

我的算法会将此种情况, 判断为滚动字幕。



图 24 left



图 25 right



图 26 合并

(三) 思考

1. 给定的样本不含有下面情况的固定字幕, 即两帧的固定字幕不同。



2. 我的算法会将距离很长的两帧的滚动字幕判定为固定字幕, 以及会将不同帧但内容相同的固定字幕判定为滚动字幕; 我觉得这两种情况应当不予讨论, 我认为滚动字幕总有一部分文字相同, 而固定字幕应当具有不同文字。另一方面, 我暂时找不出这个问题的解决办法。
3. 我遇到的主要问题, 就是如何区分滚动字幕和固定字幕, 我放到了最后才解决这个问题。我是假设滚动字幕的字体是平移过去的, 因此滚动字幕的匹配特征点应当具有大量相等的距离, 我也是基于此来进行二者的区分的。除此之外, 我还有另外一个想法, 滚动字幕应当有部分连续的字体相同, 因此可以识别出字体, 然后进行判断, 但此种方法未进行实践。
4. 我目前未能很好的理解单应性矩阵以及透视变换的原理, 对它们的理解仅仅停留在单应性矩阵是确定四个点位置, 并在不同的视图角度中均能确定这四个位置; 透视变换是将图片将图片投影到一个新的视平面, 即投影映射。