1. orb-slam特征匹配：

通常估计两帧之间的运动, 我们需要分别计算两帧中的所有特征点, 然后计算特征点之间的匹配关系. 再通过对应特征点的移动情况来估计两帧之间的摄像机运动.

这中间会涉及好很多问题, 至今未能得到有效解决:

提取特征点的类型: 众所周知, 对于自然场景图像, SIFT和SURF特征点具有非常好的特性, 充分考虑了尺度, 光照, 旋转等因素的影响. 然而SIFT, SURF特征的计算非常耗时, 很难满足实时应用的需要. 于是我们只能转而使用质量较差但计算较为简单的图像特征点, 如FAST, ORB等.

特征点匹配的问题: 在确定特征点位置后, 我们可以计算这个特征点的descriptor. 然后在两帧之间两两对比descriptor, 当descriptor之间的distance低于某个阈值时, 便认为这两个特征是同一个点. 然而这里存在两个问题:

(1) 到底distance阈值应该设为多少, 大多凭实验经验获得, 并没有什么理论依据.

(2) 特征点之间两两比对, 计算量非常大. 而且其实有很多比对是不需要的. 目前的解决方案主要是假设两帧之间的相对运动不大, 于是可以在第一帧的特征点位置附近来搜索对应特征即可. 但是即使如此, 计算量还是难以满足实时需要. 所以在ORB-SLAM中, 还有很多其它的工程上的优化被考虑进去.

刚开始接触ORB的时候, 误解了作者对bow模块的使用, 还以为bow只是简单地用于loop closure detection, 后面仔细查看代码后发现并没有那么简单.

在作者使用的Bag of Word词典中, 词典是一个事先训练好的分类树, 而BOW特征有两种:

1. BowVector: 即是分类树中leaf的数值与权重

2. FeatureVector: 是分类树中leaf的id值与对应输入ORB特征列表的特征序号.

ORB SLAM中, 在利用帧间所有特征点比对初始化地图点以后, 后面的帧间比对都采用Feature vector.进行, 而不再利用所有特征点的descriptor两两比对. 这样做的好处当然是加快了处理速度, 但是信息再次被压缩抽象化, 不可避免会造成性能降低. 然而根据作者在之前的文章[1]及github上的描述, 对一幅图片的BOW特征抽取可以在5ms以内完成, 而在19000张图片构成的database中, 图片搜索可以在10ms内完成, 且保证False Positive为0. 具体的实验我没有进行验证, Whatever, ORB-SLAM证明了这样处理是有效的, 至少在数据集上, 及速度较慢的应用上, 可以实现令人满意的精度.

其实处理非常简单, 在已经获得图像特征点集合的基础上, 再根据词典, 对每个特征做一次分类. 再对第二幅图像提取特征, 然后也根据词典, 也对这幅图像的所有特征进行分类. 用分类后的特征类别代替原本的特征descriptor, 即用一个数字代替一个向量进行比对, 显然速度可以大大提升.

这一方案在ORB SLAM整个project中被大量应用, 除了初始化时, 因为需要较准确的地图点及初始位姿估计, 而采用descriptor进行暴力比对. 后面的在SearchForTriangulation, SearchByBoW的过程中, 都是直接使用Feature vector作为特征的值的. 在实际应用中发现效果十分不错.

然而这一种方法有一个最恼人的问题就是, 每次在初始化系统的时候必须载入一个100多m的词典库, 在我的android orb下测试, 这个载入的过程长达2分钟以上.

当前使用的词典是使用K-Nearest-Neighbor算法生成的分类树, 学过机器学习的童鞋都知道, 这是一种instance based的聚类算法, 分类速度快, 但是所需存储空间大. 所以如果使用其它的聚类算法生成分类器, 不知是否还能达到real time的需求.

另外, 是否可以通过一些online learning的方法进行词典学习, 或者把特征descriptor转为二进制表达, 加速运算. 这些都是待考究的问题

————————————————

版权声明：本文为CSDN博主「Qinrui\_Yan」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/qinruiyan/article/details/51042341>

ORBextractor.cc 79行计算特征点方向代码没懂