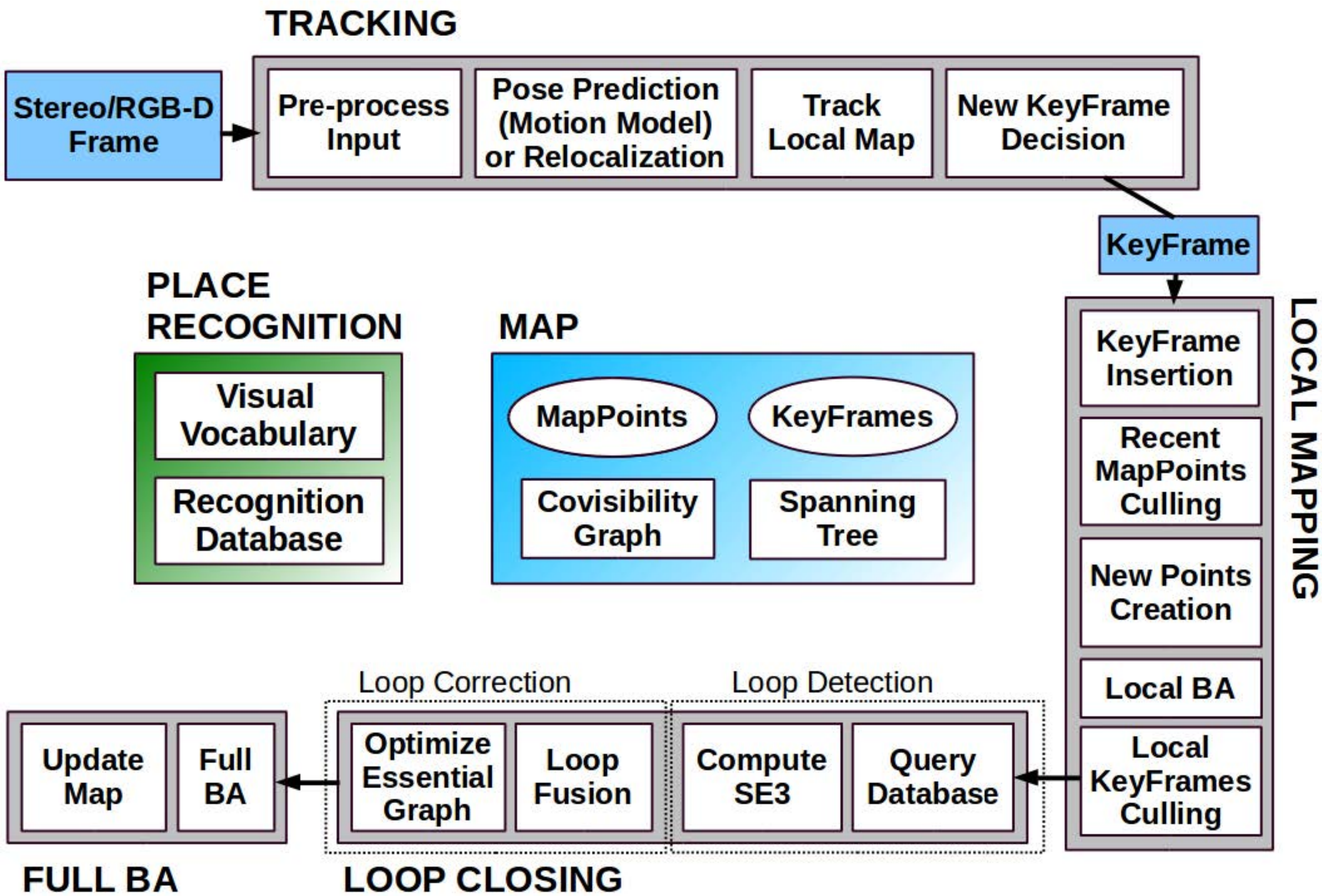
ORB-SLAM2代码解读

1. 总述

ORB-SLAM2主要由3个模块组成，分别为Tracking, Mapping和Loop closing. 如下图所示：



每个模块对应一个线程，另外还有viewer线程用于显示，执行完Loop closing后还会开启一个全局BA线程。

类	文件	描述
Tracking	`Tracking.cc`	追踪线程，运动优化
LocalMapping	`LocalMapping.cc`	局部地图管理
LoopClosing	`LoopClosing.cc`	回环检测
System	`System.cc`	系统入口，管理所有线程

类	文件	描述
Map	`Map.cc`	
MapPoint	`MapPoint.cc`	
Frame	`Frame.cc`	
KeyFrame	`KeyFrame.cc`	
viewer	`viewer.cc`	
FrameDrawer	`FrameDrawer.cc`	
MapDrawer	`MapDrawer.cc`	
ORBextractor	`ORBextractor.cc`	提取ORB特征点

2. stereo代码解读

1. `stereo_kitti.cc` ：程序入口

```
LoadImages(); // 加载左右图像的路径及其时间戳
ORB_SLAM2::System SLAM(); // 初始化SLAM系统对象(2.1)
for(int ni=0; ni<nImages; ni++){ // 循环读取图像
    SLAM.TrackStereo(imLeft,imRight,tframe); // 把图像传递给SLAM系统对象(2.2)
}
SLAM.Shutdown(); // 停止所有线程
SLAM.SaveTrajectoryKITTI(); // 保存轨迹
```

2.1 `System.cc` ：SLAM系统的初始化

```
mpVocabulary = new ORBVocabulary();
mpVocabulary->loadFromTextFile(strVocFile); //加载ORB词典
mpKeyFrameDatabase = new KeyFrameDatabase(*mpVocabulary); // 创建关键帧数据库
mpMap = new Map(); // 创建地图
mpFrameDrawer = new FrameDrawer(mpMap);
mpMapDrawer = new MapDrawer(mpMap, strSettingsFile); // 创建可视化对象

mpTracker = new Tracking(); // 创建Tracking对象(3.1)

mpLocalMapper = new LocalMapping();
mptLocalMapping = new thread(); // 创建LocalMapping对象并启动线程

mpLoopCloser = new LoopClosing();
mptLoopClosing = new thread(); // 创建LoopClosing对象并启动线程

mpViewer = new Viewer();
mptViewer = new thread(); // 创建Viewer对象并启动线程

// 设置线程之间的指针
```

2.2 System::TrackStereo()：Tracking线程（主线程）

```
// 判断是否更改模式（定位，定位与建图）
// 判断是否重置
cv::Mat Tcw = mpTracker->GrabImageStereo(imLeft,imRight,timestamp); // 输入图像，获得变换位姿(3.2
```

3.1 Tracking::Tracking()：Tracking初始化

```
// 加载相机参数（内参、去畸变参数、基线、帧率、ORB参数等）
mpORBextractorLeft = new ORBextractor();
mpORBextractorRight = new ORBextractor(); // 初始化左右图像的ORB特征提取器(5.1)
mThDepth = mbf*(float)fSettings["ThDepth"]/fx; // 设置远近点阈值
```

3.2 Tracking::GrabImageStereo()：输入图像处理

```
cvtColor(mImGray,mImGray,CV_RGB2GRAY);
cvtColor(imGrayRight,imGrayRight,CV_RGB2GRAY);  //图像转换为灰度图
mCurrentFrame = Frame(); //构造Frame(4.1)

Track(); // (3.3)
```

3.3 Tracking::Track()：Tracking线程（主线程）

```
StereoInitialization(); // 第一帧初始化
mpFrameDrawer->Update(this); // 更新FrameDrawer

if(!mbOnlyTracking){ // 定位与建图模式
    if(mState==OK){
        CheckReplacedInLastFrame();

        if(mVelocity.empty() || mCurrentFrame.mnId<mnLastRelocFrameId+2){
            bOK = TrackReferenceKeyFrame();
        }else{
            bOK = TrackWithMotionModel();
            if(!bOK)
                bOK = TrackReferenceKeyFrame();
        }
    }else{ // 丢失，重新定位
        bOK = Relocalization();
    }
}else{ // 定位模式
    if(mState==LOST){
        bOK = Relocalization(); // 追踪丢失，重新定位
    }else{
        if(!mbVO){
            if(!mVelocity.empty()){
                bOK = TrackWithMotionModel();
            }else{
                bOK = TrackReferenceKeyFrame();
            }
        }else{
            // In last frame we tracked mainly "visual odometry" points.

            // We compute two camera poses, one from motion model and one doing relocalization.
            // If relocalization is sucessfull we choose that solution, otherwise we retain
            // the "visual odometry" solution.

            bool bOKMM = false;
            bool bOKReloc = false;
            vector<MapPoint*> vpMPsMM;
            vector<bool> vbOutMM;
            cv::Mat TcwMM;
            if(!mVelocity.empty()){
                bOKMM = TrackWithMotionModel();
                vpMPsMM = mCurrentFrame.mvpMapPoints;
                vbOutMM = mCurrentFrame.mvbOutlier;
                TcwMM = mCurrentFrame.mTcw.clone();
            }
            bOKReloc = Relocalization();

            if(bOKMM && !bOKReloc){
                mCurrentFrame.SetPose(TcwMM);
                mCurrentFrame.mvpMapPoints = vpMPsMM;
                mCurrentFrame.mvbOutlier = vbOutMM;
```

```
                if(mbVO){
                    for(int i =0; i<mCurrentFrame.N; i++){
                        if(mCurrentFrame.mvpMapPoints[i] && !mCurrentFrame.mvbOutlier[i]){
                            mCurrentFrame.mvpMapPoints[i]->IncreaseFound();
                        }
                    }
                }
            }else if(bOKReloc){
                mbVO = false;
            }

            bOK = bOKReloc || bOKMM;
        }
    }
}
```

4.1 `Frame.cc` ：Frame初始化（构造Frame）

```
mnId=nNextId++; // 设置frame ID

thread threadLeft(&Frame::ExtractORB,this,0,imLeft);
thread threadRight(&Frame::ExtractORB,this,1,imRight); // 提取特征点
UndistortKeyPoints();   // 关键点去畸变， 只适用于RGBD图像
ComputeStereoMatches(); // 计算左右图像的匹配点
ComputeImageBounds();   // 对于第一帧，计算图像边界
AssignFeaturesToGrid(); // 将特征点分配到网格中，加速特征点匹配
```

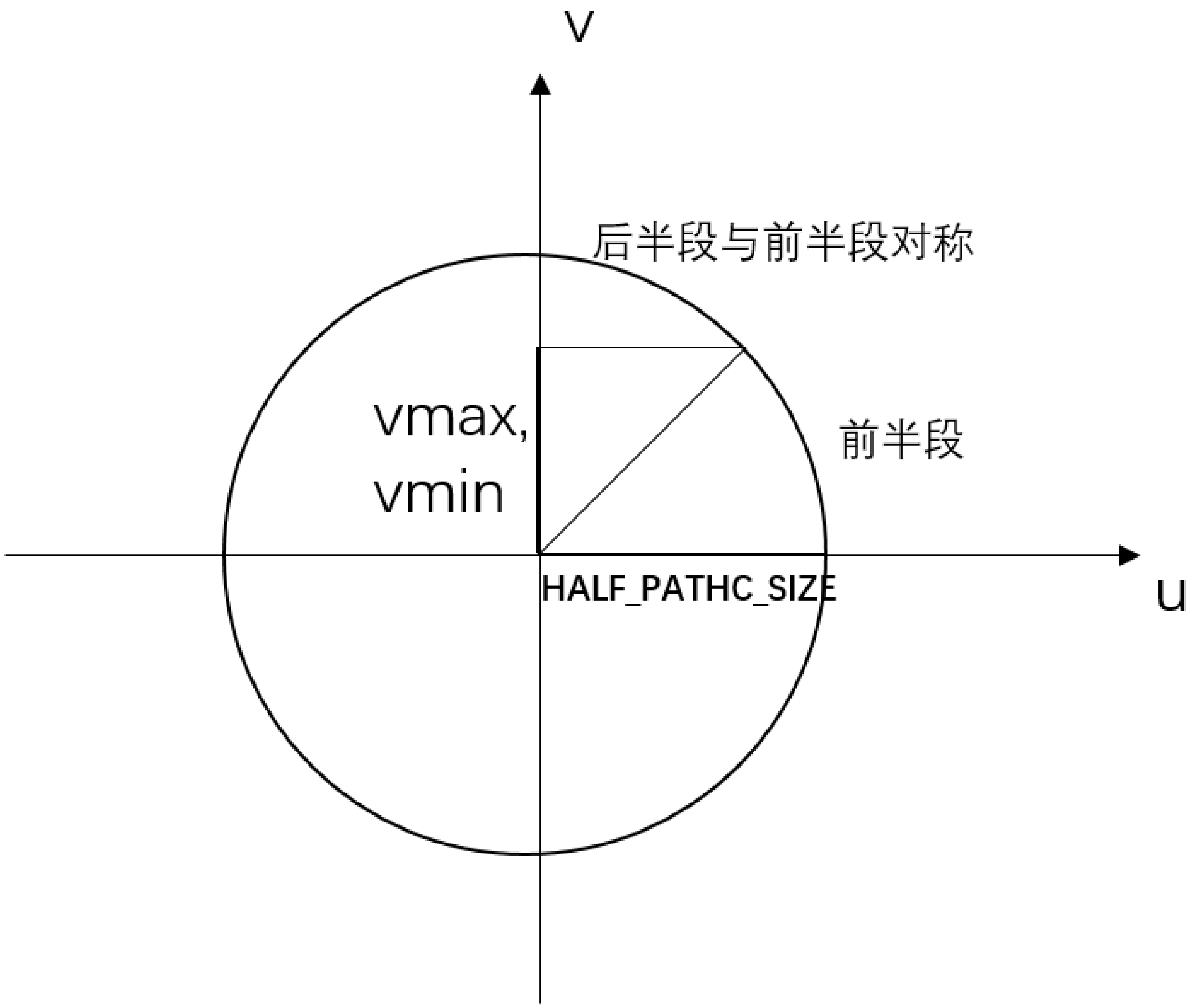
5.1 `ORBextractor.cc` ：ORBextractor初始化

```
mvScaleFactor[i]=mvScaleFactor[i-1]*scaleFactor; // 每层的缩放因子
mvLevelSigma2[i]=mvScaleFactor[i]*mvScaleFactor[i]; // 每层的缩放因子平方
mvInvScaleFactor[i]=1.0f/mvScaleFactor[i];
mvInvLevelSigma2[i]=1.0f/mvLevelSigma2[i];

mnFeaturesPerLevel[level] = cvRound(nDesiredFeaturesPerScale); // 每层预期的特征点个数

// 计算特征点方向准备
umax.resize(HALF_PATCH_SIZE + 1);
// 计算每个v坐标对应的最大u坐标
int v, v0, vmax = cvFloor(HALF_PATCH_SIZE * sqrt(2.f) / 2 + 1); // 11
int vmin = cvCeil(HALF_PATCH_SIZE * sqrt(2.f) / 2); // 11
const double hp2 = HALF_PATCH_SIZE*HALF_PATCH_SIZE;
// 计算前半段
for (v = 0; v <= vmax; ++v)
    umax[v] = cvRound(sqrt(hp2 - v * v));

// 后半段与前半段对称，确保能组成一个圆
for (v = HALF_PATCH_SIZE, v0 = 0; v >= vmin; --v){
    // 前半段的u相同对应为后半段的v相同，而一个v只能对应一个u
    while (umax[v0] == umax[v0 + 1])
        ++v0;
    umax[v] = v0;
    ++v0;
}
```



5.2 ORBextractor::operator()()：检测关键点及描述子

```
ComputePyramid(image);  // 构建图像金字塔(5.3)
ComputeKeyPointsOctTree(); // 关键点计算(以八叉树表示)(5.4)
```

5.3 ORBextractor::ComputePyramid(cv::Mat image)：构建图像金字塔

```
// 计算每一层图像
for (int level = 0; level < nlevels; ++level)
    mvImagePyramid[level] = temp(Rect(EDGE_THRESHOLD, EDGE_THRESHOLD, sz.width, sz.height));

    // Compute the resized image
    if( level != 0 ){
        resize(mvImagePyramid[level-1], mvImagePyramid[level], sz, 0, 0, INTER_LINEAR);
    }else{
        copyMakeBorder(image, temp, EDGE_THRESHOLD, EDGE_THRESHOLD,
        EDGE_THRESHOLD, EDGE_THRESHOLD, BORDER_REFLECT_101);
    }
```

5.4 ORBextractor::ComputeKeyPointsOctTree()：关键点计算(以四叉树表示)

```
for (int level = 0; level < nlevels; ++level)
    // 计算FAST特征点
    FAST(mvImagePyramid[level].rowRange(iniY,maxY).colRange(iniX,maxX),
                        vKeysCell,iniThFAST,true);
    DistributeOctTree(); // 将关键点以四叉数表示(5.5)
```

5.5 ORBextractor::DistributeOctTree()：以四叉数的形式表示关键点，按照坐标将关键点划分到不同的节点中，每个节点只包含一个响应最大的关键点