HanLP Handbook

version 1.0

hankcs

April 28, 2018

Contents

vision workspace

Contents:

1.初始化Initializer.h

构建Initializer的类 该类主要完成的功能是： 初始化SLAM的R,t,及点云，计算Fundamental,Homography,以及分解Fundamental 和Homography,存储当前帧与参考帧的关键点以及特征匹配。三角化方法等等。

- 1 .用reference frame来初始化，这个reference frame就是SLAM正式开始的第一帧
- 2.用current frame,也就是用SLAM逻辑上的第二帧来初始化整个SLAM，得到最开始两帧之间的R t,以及点云
- 3. FindHomography
  假设场景为平面情况下通过前两帧求取Homography矩阵(current frame 2 到 reference frame 1),并得到该模型的评分
- 4. FindFundamental
  假设场景为非平面情况下通过前两帧求取Fundamental矩阵(current frame 2 到 reference frame 1),并得到该模型的评分
- 5. ComputeH21
  被FindHomography函数调用具体来算Homography矩阵
- 6. ComputeF21
  被FindFundamental函数调用具体来算Fundamental矩阵
- 7. CheckHomography
  被FindHomography函数调用，具体来算假设使用Homography模型的得分
- 8.CheckFundamental
  被FindFundamental函数调用，具体来算假设使用Fundamental模型的得分
- 9.ReconstructF
  分解F矩阵，并从分解后的多个解中找出合适的R，t
- 10.ReconstructH
  分解H矩阵，并从分解后的多个解中找出合适的R，t
- 11.Triangulate
  通过三角化方法，利用反投影矩阵将特征点恢复为3D点
- 12.Normalize
  归一化三维空间点和帧间位移t
- 13.CheckRT
  ReconstructF调用该函数进行cheirality check，从而进一步找出F分解后最合适的解
- 14.DecomposeE
  F矩阵通过结合内参可以得到Essential矩阵，该函数用于分解E矩阵，将得到4组解
- 15. 除了以上函数外，还有一些变量，主要用来存储参考帧和当前帧的特征点,以及记录匹配的点，相机内参，以及计算Fundamental 和Homography 矩阵时RANSAC迭代次数

  vector<cv::KeyPoint> mvKeys1; ///< 存储Reference Frame中的特征点

  vector<cv::KeyPoint> mvKeys2; ///< 存储Current Frame中的特征点

  vector<Match> mvMatches12; ///< Match的数据结构是pair,mvMatches12只记录Reference到Current匹配上的特征点对 vector<bool> mvbMatched1; ///< 记录Reference Frame的每个特征点在Current Frame是否有匹配的特征点

  cv::Mat mK; ///< 相机内参

  // Standard Deviation and Variance float mSigma, mSigma2; ///< 测量误差

  // Ransac max iterations int mMaxIterations; ///< 算Fundamental和Homography矩阵时RANSAC迭代次数

  // Ransac sets vector<vector<size_t> > mvSets; ///< 二维容器，外层容器的大小为迭代次数，内层容器大小为每次迭代算H或F矩阵需要的点

```
class Initializer
{
    typedef pair<int,int> Match;

public:

    // Fix the reference frame
    // 用reference frame来初始化，这个reference frame就是SLAM正式开始的第一帧
    Initializer(const Frame &ReferenceFrame, float sigma = 1.0, int iterations = 200);

    // Computes in parallel a fundamental matrix and a homography
    // Selects a model and tries to recover the motion and the structure from motion
    // 用current frame,也就是用SLAM逻辑上的第二帧来初始化整个SLAM，得到最开始两帧之间的R t,以及点云
    bool Initialize(const Frame &CurrentFrame, const vector<int> &vMatches12,
                    cv::Mat &R21, cv::Mat &t21, vector<cv::Point3f> &vP3D, vector<bool> &vbTriangulated);
```

主要函数成员

```
private:

    // 假设场景为平面情况下通过前两帧求取Homography矩阵(current frame 2 到 reference frame 1),并得到该模型的评分
    void FindHomography(vector<bool> &vbMatchesInliers, float &score, cv::Mat &H21);
    // 假设场景为非平面情况下通过前两帧求取Fundamental矩阵(current frame 2 到 reference frame 1),并得到该模型的评分
    void FindFundamental(vector<bool> &vbInliers, float &score, cv::Mat &F21);

    // 被FindHomography函数调用具体来算Homography矩阵
    cv::Mat ComputeH21(const vector<cv::Point2f> &vP1, const vector<cv::Point2f> &vP2);
    // 被FindFundamental函数调用具体来算Fundamental矩阵
    cv::Mat ComputeF21(const vector<cv::Point2f> &vP1, const vector<cv::Point2f> &vP2);

    // 被FindHomography函数调用，具体来算假设使用Homography模型的得分
    float CheckHomography(const cv::Mat &H21, const cv::Mat &H12, vector<bool> &vbMatchesInliers, float sigma);
    // 被FindFundamental函数调用，具体来算假设使用Fundamental模型的得分
    float CheckFundamental(const cv::Mat &F21, vector<bool> &vbMatchesInliers, float sigma);

    // 分解F矩阵，并从分解后的多个解中找出合适的R，t
    bool ReconstructF(vector<bool> &vbMatchesInliers, cv::Mat &F21, cv::Mat &K,
                      cv::Mat &R21, cv::Mat &t21, vector<cv::Point3f> &vP3D, vector<bool> &vbTriangulated, float minParallax, int minTriangulated);

    // 分解H矩阵，并从分解后的多个解中找出合适的R，t
    bool ReconstructH(vector<bool> &vbMatchesInliers, cv::Mat &H21, cv::Mat &K,
                      cv::Mat &R21, cv::Mat &t21, vector<cv::Point3f> &vP3D, vector<bool> &vbTriangulated, float minParallax, int minTriangulated);

    // 通过三角化方法，利用反投影矩阵将特征点恢复为3D点
    void Triangulate(const cv::KeyPoint &kp1, const cv::KeyPoint &kp2, const cv::Mat &P1, const cv::Mat &P2, cv::Mat &x3D);

    // 归一化三维空间点和帧间位移t
    void Normalize(const vector<cv::KeyPoint> &vKeys, vector<cv::Point2f> &vNormalizedPoints, cv::Mat &T);

    // ReconstructF调用该函数进行cheirality check，从而进一步找出F分解后最合适的解
    int CheckRT(const cv::Mat &R, const cv::Mat &t, const vector<cv::KeyPoint> &vKeys1, const vector<cv::KeyPoint> &vKeys2,
                       const vector<Match> &vMatches12, vector<bool> &vbInliers,
                       const cv::Mat &K, vector<cv::Point3f> &vP3D, float th2, vector<bool> &vbGood, float &parallax);

    // F矩阵通过结合内参可以得到Essential矩阵，该函数用于分解E矩阵，将得到4组解
    void DecomposeE(const cv::Mat &E, cv::Mat &R1, cv::Mat &R2, cv::Mat &t);


    // Keypoints from Reference Frame (Frame 1)
    vector<cv::KeyPoint> mvKeys1; ///< 存储Reference Frame中的特征点

    // Keypoints from Current Frame (Frame 2)
    vector<cv::KeyPoint> mvKeys2; ///< 存储Current Frame中的特征点

    // Current Matches from Reference to Current
    // Reference Frame: 1, Current Frame: 2
    vector<Match> mvMatches12; ///< Match的数据结构是pair,mvMatches12只记录Reference到Current匹配上的特征点对
    vector<bool> mvbMatched1; ///< 记录Reference Frame的每个特征点在Current Frame是否有匹配的特征点

    // Calibration
    cv::Mat mK; ///< 相机内参

    // Standard Deviation and Variance
    float mSigma, mSigma2; ///< 测量误差

    // Ransac max iterations
    int mMaxIterations; ///< 算Fundamental和Homography矩阵时RANSAC迭代次数

    // Ransac sets
    vector<vector<size_t> > mvSets; ///< 二维容器，外层容器的大小为迭代次数，内层容器大小为每次迭代算H或F矩阵需要的点

};
```

2.Frame帧类

Frame类中包含了MapPoint类和KeyFrame类

```
#include "MapPoint.h"
#include "Thirdparty/DBoW2/DBoW2/BowVector.h"
#include "Thirdparty/DBoW2/DBoW2/FeatureVector.h"
#include "ORBVocabulary.h"
#include "KeyFrame.h"
#include "ORBextractor.h"

class MapPoint;
class KeyFrame;
```

分别为双目摄像头，深度摄像头，单目摄像头三类构建帧类的复制构造函数

```
// Constructor for stereo cameras.
Frame(const cv::Mat &imLeft, const cv::Mat &imRight, const double &timeStamp, ORBextractor* extractorLeft, ORBextractor* extractorRight, ORBVocabulary* voc

// Constructor for RGB-D cameras.
Frame(const cv::Mat &imGray, const cv::Mat &imDepth, const double &timeStamp, ORBextractor* extractor,ORBVocabulary* voc, cv::Mat &K, cv::Mat &distCoef, co

// Constructor for Monocular cameras.
Frame(const cv::Mat &imGray, const double &timeStamp, ORBextractor* extractor,ORBVocabulary* voc, cv::Mat &K, cv::Mat &distCoef, const float &bf, const flo
```

抽取ORB特征

```
// Extract ORB on the image. 0 for left image and 1 for right image.
// 提取的关键点存放在mvKeys和mDescriptors中
// ORB是直接调orbExtractor提取的
void ExtractORB(int flag, const cv::Mat &im);
```

计算词袋BoW

```
// Compute Bag of Words representation.
// 存放在mBowVec中
void ComputeBoW();
```

设置相机位姿

```
// Set the camera pose.
// 用Tcw更新mTcw
void SetPose(cv::Mat Tcw);
```

从相机姿态中计算旋转，平移和相机中心矩阵

```
// Computes rotation, translation and camera center matrices from the camera pose.
void UpdatePoseMatrices();
```

得到相机中心点

```
// Returns the camera center.
inline cv::Mat GetCameraCenter()
{
    return mOw.clone();
}
// Returns inverse of rotation
```

得到旋转矩阵的逆矩阵

```
inline cv::Mat GetRotationInverse()
{
    return mRwc.clone();
}
```

判断路标点是否在视野中

```
// Check if a MapPoint is in the frustum of the camera
// and fill variables of the MapPoint to be used by the tracking
// 判断路标点是否在视野中
bool isInFrustum(MapPoint* pMP, float viewingCosLimit);
```

判断关键点是否在grid中

```
// Compute the cell of a keypoint (return false if outside the grid)
bool PosInGrid(const cv::KeyPoint &kp, int &posX, int &posY);

vector<size_t> GetFeaturesInArea(const float &x, const float  &y, const float  &r, const int minLevel=-1, const int maxLevel=-1) const;
```

判断左右图关键点是否match,如果match,计算深度信息并将左右关键点坐标存储

```
// Search a match for each keypoint in the left image to a keypoint in the right image.
// If there is a match, depth is computed and the right coordinate associated to the left keypoint is stored.
void ComputeStereoMatches();

// Associate a "right" coordinate to a keypoint if there is valid depth in the depthmap.
void ComputeStereoFromRGBD(const cv::Mat &imDepth);
```

将一个关键点从映射到3D世界坐标

```
// Backprojects a keypoint (if stereo/depth info available) into 3D world coordinates.
cv::Mat UnprojectStereo(const int &i);
```

3.keyframe 关键帧

3.keyframe 关键帧

这里有线程锁的概念，还不是很清楚这块

关键帧，和普通的Frame不一样，但是可以由Frame来构造 许多数据会被三个线程同时访问，所以用锁的地方很普遍

关键帧包含了地图，路标点，帧，关键帧数据库等类

```
class Map;
class MapPoint;
class Frame;
class KeyFrameDatabase;
```

设置Pose,得到Pose,Pose的逆矩阵，Get相机中心，Get双目相机中心，Get旋转矩阵，Get平移,计算BoW

```
// Pose functions
// 这里的set,get需要用到锁
void SetPose(const cv::Mat &Tcw);
cv::Mat GetPose();
cv::Mat GetPoseInverse();
cv::Mat GetCameraCenter();
cv::Mat GetStereoCenter();
cv::Mat GetRotation();
cv::Mat GetTranslation();

// Bag of Words Representation
void ComputeBoW();
```

图优化相关的一些函数

Covisibility graph是不同关键帧之间共享的可见点。

添加连接connection，删除连接，更新连接，更新最好的共享可见点.

添加子节点child，删除子节点，得到子节点

添加路标点MapPoint，删除路标点，得到路标点，

LoopEdge，

关键点 keypoint

```
// Covisibility graph functions
void AddConnection(KeyFrame* pKF, const int &weight);
void EraseConnection(KeyFrame* pKF);
void UpdateConnections();
void UpdateBestCovisibles();
std::set<KeyFrame *> GetConnectedKeyFrames();
std::vector<KeyFrame* > GetVectorCovisibleKeyFrames();
std::vector<KeyFrame*> GetBestCovisibilityKeyFrames(const int &N);
std::vector<KeyFrame*> GetCovisiblesByWeight(const int &w);
int GetWeight(KeyFrame* pKF);

// Spanning tree functions
void AddChild(KeyFrame* pKF);
void EraseChild(KeyFrame* pKF);
void ChangeParent(KeyFrame* pKF);
std::set<KeyFrame*> GetChilds();
KeyFrame* GetParent();
bool hasChild(KeyFrame* pKF);

// Loop Edges
void AddLoopEdge(KeyFrame* pKF);
std::set<KeyFrame*> GetLoopEdges();

// MapPoint observation functions
void AddMapPoint(MapPoint* pMP, const size_t &idx);
void EraseMapPointMatch(const size_t &idx);
void EraseMapPointMatch(MapPoint* pMP);
void ReplaceMapPointMatch(const size_t &idx, MapPoint* pMP);
std::set<MapPoint*> GetMapPoints();
std::vector<MapPoint*> GetMapPointMatches();
int TrackedMapPoints(const int &minObs);
MapPoint* GetMapPoint(const size_t &idx);

// KeyPoint functions
std::vector<size_t> GetFeaturesInArea(const float &x, const float  &y, const float  &r) const;
cv::Mat UnprojectStereo(int i);

// Image
bool IsInImage(const float &x, const float &y) const;

// Enable/Disable bad flag changes
void SetNotErase();
void SetErase();

// Set/check bad flag
void SetBadFlag();
```

```
bool isBad();

// Compute Scene Depth (q=2 median). Used in monocular.
float ComputeSceneMedianDepth(const int q);

static bool weightComp( int a, int b)
{
    return a>b;
}

static bool lId(KeyFrame* pKF1, KeyFrame* pKF2)
{
    return pKF1->mnId<pKF2->mnId;
}
```

下面的变量只可以单线程访问

包含了keyframe的ID号，时间戳，Grid,local mapping的一些变量,回环的一些变量

相机补偿的参数，等等

```
// The following variables are accesed from only 1 thread or never change (no mutex needed).
public:

    // nNextID名字改为nLastID更合适，表示上一个KeyFrame的ID号
    static long unsigned int nNextId;
    // 在nNextID的基础上加1就得到了mnID，为当前KeyFrame的ID号
    long unsigned int mnId;
    // 每个KeyFrame基本属性是它是一个Frame，KeyFrame初始化的时候需要Frame，
    // mnFrameId记录了该KeyFrame是由哪个Frame初始化的
    const long unsigned int mnFrameId;

    const double mTimeStamp;

    // Grid (to speed up feature matching)
    // 和Frame类中的定义相同
    const int mnGridCols;
    const int mnGridRows;
    const float mfGridElementWidthInv;
    const float mfGridElementHeightInv;

    // Variables used by the tracking
    long unsigned int mnTrackReferenceForFrame;
    long unsigned int mnFuseTargetForKF;

    // Variables used by the local mapping
    long unsigned int mnBALocalForKF;
    long unsigned int mnBAFixedForKF;

    // Variables used by the keyframe database
    long unsigned int mnLoopQuery;
    int mnLoopWords;
    float mLoopScore;
    long unsigned int mnRelocQuery;
    int mnRelocWords;
    float mRelocScore;

    // Variables used by loop closing
    cv::Mat mTcwGBA;
    cv::Mat mTcwBefGBA;
    long unsigned int mnBAGlobalForKF;

    // Calibration parameters
    const float fx, fy, cx, cy, invfx, invfy, mbf, mb, mThDepth;

    // Number of KeyPoints
    const int N;

    // KeyPoints, stereo coordinate and descriptors (all associated by an index)
    // 和Frame类中的定义相同
    const std::vector<cv::KeyPoint> mvKeys;
    const std::vector<cv::KeyPoint> mvKeysUn;
    const std::vector<float> mvuRight; // negative value for monocular points
    const std::vector<float> mvDepth; // negative value for monocular points
    const cv::Mat mDescriptors;

    //BoW
    DBoW2::BowVector mBowVec; ///< Vector of words to represent images
    DBoW2::FeatureVector mFeatVec; ///< Vector of nodes with indexes of local features

    // Pose relative to parent (this is computed when bad flag is activated)
    cv::Mat mTcp;
```

```
    // Scale
    const int mnScaleLevels;
    const float mfScaleFactor;
    const float mfLogScaleFactor;
    const std::vector<float> mvScaleFactors;// 尺度因子，scale^n，scale=1.2，n为层数
    const std::vector<float> mvLevelSigma2;// 尺度因子的平方
    const std::vector<float> mvInvLevelSigma2;

    // Image bounds and calibration
    const int mnMinX;
    const int mnMinY;
    const int mnMaxX;
    const int mnMaxY;
    const cv::Mat mK;
```

4.KeyframeDatabase

该类会用到KeyFrame和Frame两个类

```
class KeyFrame;
class Frame;
```

类定义

包含了添加，删除，清除，回环检测，重定位等函数

包含了ORB词典，索引文件等

```
class KeyFrameDatabase
{
public:

    KeyFrameDatabase(const ORBVocabulary &voc);

   void add(KeyFrame* pKF);

   void erase(KeyFrame* pKF);

   void clear();

   // Loop Detection
   std::vector<KeyFrame *> DetectLoopCandidates(KeyFrame* pKF, float minScore);

   // Relocalization
   std::vector<KeyFrame*> DetectRelocalizationCandidates(Frame* F);

protected:

  // Associated vocabulary
  const ORBVocabulary* mpVoc; ///< 预先训练好的词典

  // Inverted file
  std::vector<list<KeyFrame*> > mvInvertedFile; ///< 倒排索引，mvInvertedFile[i]表示包含了第i个word id的所有关键帧

  // Mutex
  std::mutex mMutex;
};
```

5.MapPoint 路标点,地图点

设置世界坐标,得到世界坐标

```
void SetWorldPos(const cv::Mat &Pos);
cv::Mat GetWorldPos();
```

归一化

```
cv::Mat GetNormal();
```

得到参考的关键帧

```
KeyFrame* GetReferenceKeyFrame();
```

观测点

```
std::map<KeyFrame*,size_t> GetObservations();
int Observations();
void AddObservation(KeyFrame* pKF,size_t idx);
void EraseObservation(KeyFrame* pKF);
```

关键帧的index

```
int GetIndexInKeyFrame(KeyFrame* pKF);
bool IsInKeyFrame(KeyFrame* pKF);

void SetBadFlag();
bool isBad();

void Replace(MapPoint* pMP);
MapPoint* GetReplaced();

void IncreaseVisible(int n=1);
void IncreaseFound(int n=1);
float GetFoundRatio();
inline int GetFound(){
    return mnFound;
}
```

计算描述子

```
void ComputeDistinctiveDescriptors();

cv::Mat GetDescriptor();

void UpdateNormalAndDepth();
```

计算最大，最小距离方差

```
float GetMinDistanceInvariance();
float GetMaxDistanceInvariance();
int PredictScale(const float &currentDist, KeyFrame*pKF);
int PredictScale(const float &currentDist, Frame* pF);
```

Tracking

TrackLocalMap - SearchByProjection中决定是否对该点进行投影的变量

mbTrackInView==false的点有几种:

- 已经和当前帧经过匹配（TrackReferenceKeyFrame，TrackWithMotionModel）但在优化过程中认为是外点
- 已经和当前帧经过匹配且为内点，这类点也不需要再进行投影
- 不在当前相机视野中的点（即未通过isInFrustum判断）

3D Descriptor

每个3D点也有一个descriptor

如果MapPoint与很多帧图像特征点对应（由keyframe来构造时），那么距离其它描述子的平均距离最小的描述子是最佳描述子

MapPoint只与一帧的图像特征点对应（由frame来构造时），那么这个特征点的描述子就是该3D点的描述子

```
public:
    long unsigned int mnId; ///< Global ID for MapPoint
    static long unsigned int nNextId;
    const long int mnFirstKFid; ///< 创建该MapPoint的关键帧ID
    const long int mnFirstFrame; ///< 创建该MapPoint的帧ID（即每一关键帧有一个帧ID）
    int nObs;

    // Variables used by the tracking
    float mTrackProjX;
    float mTrackProjY;
    float mTrackProjXR;
    int mnTrackScaleLevel;
    float mTrackViewCos;

    // TrackLocalMap - SearchByProjection中决定是否对该点进行投影的变量
    // mbTrackInView==false的点有几种:
    // a 已经和当前帧经过匹配（TrackReferenceKeyFrame，TrackWithMotionModel）但在优化过程中认为是外点
    // b 已经和当前帧经过匹配且为内点，这类点也不需要再进行投影
    // c 不在当前相机视野中的点（即未通过isInFrustum判断）
    bool mbTrackInView;
    // TrackLocalMap - UpdateLocalPoints中防止将MapPoints重复添加至mvpLocalMapPoints的标记
    long unsigned int mnTrackReferenceForFrame;
    // TrackLocalMap - SearchLocalPoints中决定是否进行isInFrustum判断的变量
    // mnLastFrameSeen==mCurrentFrame.mnId的点有几种:
    // a 已经和当前帧经过匹配（TrackReferenceKeyFrame，TrackWithMotionModel）但在优化过程中认为是外点
    // b 已经和当前帧经过匹配且为内点，这类点也不需要再进行投影
    long unsigned int mnLastFrameSeen;

    // Variables used by local mapping
    long unsigned int mnBALocalForKF;
    long unsigned int mnFuseCandidateForKF;

    // Variables used by loop closing
    long unsigned int mnLoopPointForKF;
```

```
    long unsigned int mnCorrectedByKF;
    long unsigned int mnCorrectedReference;
    cv::Mat mPosGBA;
    long unsigned int mnBAGlobalForKF;


    static std::mutex mGlobalMutex;

protected:

    // Position in absolute coordinates
    cv::Mat mWorldPos; ///< MapPoint在世界坐标系下的坐标

    // Keyframes observing the point and associated index in keyframe
    std::map<KeyFrame*,size_t> mObservations; ///< 观测到该MapPoint的KF和该MapPoint在KF中的索引

    // Mean viewing direction
    // 该MapPoint平均观测方向
    cv::Mat mNormalVector;

    // Best descriptor to fast matching
    // 每个3D点也有一个descriptor
    // 如果MapPoint与很多帧图像特征点对应（由keyframe来构造时），那么距离其它描述子的平均距离最小的描述子是最佳描述子
    // MapPoint只与一帧的图像特征点对应（由frame来构造时），那么这个特征点的描述子就是该3D点的描述子
    cv::Mat mDescriptor; ///< 通过 ComputeDistinctiveDescriptors() 得到的最优描述子

    // Reference KeyFrame
    KeyFrame* mpRefKF;

    // Tracking counters
    int mnVisible;
    int mnFound;

    // Bad flag (we do not currently erase MapPoint from memory)
    bool mbBad;
    MapPoint* mpReplaced;

    // Scale invariance distances
    float mfMinDistance;
    float mfMaxDistance;

    Map* mpMap;

    std::mutex mMutexPos;
    std::mutex mMutexFeatures;
};
```

6.Map地图

地图负责管理关键帧，路标点的功能

在地图中添加关键帧，添加路标点，删除路标点，删除关键帧，设置参考路标点，获得所有关键帧，过得参考的地图点

```
class Map
{
public:
    Map();

    void AddKeyFrame(KeyFrame* pKF);
    void AddMapPoint(MapPoint* pMP);
    void EraseMapPoint(MapPoint* pMP);
    void EraseKeyFrame(KeyFrame* pKF);
    void SetReferenceMapPoints(const std::vector<MapPoint*> &vpMPs);

    std::vector<KeyFrame*> GetAllKeyFrames();
    std::vector<MapPoint*> GetAllMapPoints();
    std::vector<MapPoint*> GetReferenceMapPoints();

    long unsigned int MapPointsInMap();
    long unsigned  KeyFramesInMap();

    long unsigned int GetMaxKFid();

    void clear();

    vector<KeyFrame*> mvpKeyFrameOrigins;

    std::mutex mMutexMapUpdate;

    // This avoid that two points are created simultaneously in separate threads (id conflict)
    std::mutex mMutexPointCreation;
```

```
protected:
    std::set<MapPoint*> mspMapPoints; ///< MapPoints
    std::set<KeyFrame*> mspKeyFrames; ///< Keyframs

    std::vector<MapPoint*> mvpReferenceMapPoints;

    long unsigned int mnMaxKFid;

    std::mutex mMutexMap;
};
```

7.ORBExtractor ORB特征提取

请详细阅读ORB特征提取的论文，搞懂它的内部算法。

ExtractorNode

```
class ExtractorNode
{
public:
    ExtractorNode():bNoMore(false){}

    void DivideNode(ExtractorNode &n1, ExtractorNode &n2, ExtractorNode &n3, ExtractorNode &n4);

    std::vector<cv::KeyPoint> vKeys;
    cv::Point2i UL, UR, BL, BR;
    std::list<ExtractorNode>::iterator lit;
    bool bNoMore;
};
```

ORBextractor

```
class ORBextractor
{
public:

    enum {HARRIS_SCORE=0, FAST_SCORE=1 };

    ORBextractor(int nfeatures, float scaleFactor, int nlevels,
                 int iniThFAST, int minThFAST);

    ~ORBextractor(){}

    // Compute the ORB features and descriptors on an image.
    // ORB are dispersed on the image using an octree.
    // Mask is ignored in the current implementation.
    void operator()( cv::InputArray image, cv::InputArray mask,
      std::vector<cv::KeyPoint>& keypoints,
      cv::OutputArray descriptors);

    int inline GetLevels(){
        return nlevels;}

    float inline GetScaleFactor(){
        return scaleFactor;}

    std::vector<float> inline GetScaleFactors(){
        return mvScaleFactor;
    }

    std::vector<float> inline GetInverseScaleFactors(){
        return mvInvScaleFactor;
    }

    std::vector<float> inline GetScaleSigmaSquares(){
        return mvLevelSigma2;
    }

    std::vector<float> inline GetInverseScaleSigmaSquares(){
        return mvInvLevelSigma2;
    }

    std::vector<cv::Mat> mvImagePyramid;

protected:

    void ComputePyramid(cv::Mat image);
    void ComputeKeyPointsOctTree(std::vector<std::vector<cv::KeyPoint> >& allKeypoints);
    std::vector<cv::KeyPoint> DistributeOctTree(const std::vector<cv::KeyPoint>& vToDistributeKeys, const int &minX,
                                           const int &maxX, const int &minY, const int &maxY, const int &nFeatures, const int &level);

    void ComputeKeyPointsOld(std::vector<std::vector<cv::KeyPoint> >& allKeypoints);
```

```
    std::vector<cv::Point> pattern;

    int nfeatures;
    double scaleFactor;
    int nlevels;
    int iniThFAST;
    int minThFAST;

    std::vector<int> mnFeaturesPerLevel;

    std::vector<int> umax;

    std::vector<float> mvScaleFactor;
    std::vector<float> mvInvScaleFactor;
    std::vector<float> mvLevelSigma2;
    std::vector<float> mvInvLevelSigma2;
};
```

8.ORBmatcher ORB 特征匹配

特征匹配

计算ORB描述子的汉明距离

```
// Computes the Hamming distance between two ORB descriptors
static int DescriptorDistance(const cv::Mat &a, const cv::Mat &b);
```

Search matches between Frame keypoints and projected MapPoints. Returns number of matches

- 搜索关键点和投影地图点的匹配。返回匹配的个数
- 经常被用来跟踪局部地图
- @brief 通过投影，对Local MapPoint进行跟踪
- 将Local MapPoint投影到当前帧中，由此增加当前帧的MapPoints n
- 在SearchLocalPoints()中已经将Local MapPoints重投影（isInFrustum()）到当前帧 n
- 并标记了这些点是否在当前帧的视野中，即mbTrackInView n
- 对这些MapPoints，在其投影点附近根据描述子距离选取匹配，以及最终的方向投票机制进行剔除
- @param F 当前帧
- @param vpMapPoints Local MapPoints
- @param th 阈值
- @return 成功匹配的数量
- @see SearchLocalPoints() isInFrustum()

Project MapPoints seen in KeyFrame into the Frame and search matches.

```
int SearchByProjection(Frame &F, const std::vector<MapPoint*> &vpMapPoints, const float th=3);
```

Used in relocalisation (Tracking)

```
int SearchByProjection(Frame &CurrentFrame, KeyFrame* pKF, const std::set<MapPoint*> &sAlreadyFound, const float th, const int ORBdist);
```

Project MapPoints using a Similarity Transformation and search matches. Used in loop detection (Loop Closing)

::

```
    int SearchByProjection(KeyFrame* pKF, cv::Mat Scw, const std::vector<MapPoint*> &vpPoints, std::vector<MapPoint*> &vpMatched, int th);
```

Use for Loop Detection

```
rute force constrained to ORB that belong to the same vocabulary node (at a certain level)
Used in Relocalisation and Loop Detection
@brief 通过词包，对关键帧的特征点进行跟踪

KeyFrame中包含了MapPoints，对这些MapPoints进行tracking \n
由于每一个MapPoint对应有描述子，因此可以通过描述子距离进行跟踪 \n
为了加速匹配过程，将关键帧和当前帧的描述子划分到特定层的nodes中 \n
对属于同一node的描述子计算距离进行匹配 \n
通过距离阈值、比例阈值和角度投票进行剔除误匹配
@param  pKF               KeyFrame
@param  F                 Current Frame
@param  vpMapPointMatches F中MapPoints对应的匹配，NULL表示未匹配
@return                   成功匹配的数量

int SearchByBoW(KeyFrame *pKF, Frame &F, std::vector<MapPoint*> &vpMapPointMatches);
int SearchByBoW(KeyFrame *pKF1, KeyFrame* pKF2, std::vector<MapPoint*> &vpMatches12);
```

Matching for the Map Initialization (only used in the monocular case)

```
int SearchForInitialization(Frame &F1, Frame &F2, std::vector<cv::Point2f> &vbPrevMatched, std::vector<int> &vnMatches12, int windowSize=10);
```

Matching to triangulate new MapPoints. Check Epipolar Constraint.

```
int SearchForTriangulation(KeyFrame *pKF1, KeyFrame* pKF2, cv::Mat F12,
                           std::vector<pair<size_t, size_t> > &vMatchedPairs, const bool bOnlyStereo);
```

Search matches between MapPoints seen in KF1 and KF2 transforming by a Sim3 [s12*R12|t12]

```
// In the stereo and RGB-D case, s12=1
int SearchBySim3(KeyFrame* pKF1, KeyFrame* pKF2, std::vector<MapPoint *> &vpMatches12, const float &s12, const cv::Mat &R12, const cv::Mat &t12, const floa
```

Project MapPoints into KeyFrame and search for duplicated MapPoints.

```
int Fuse(KeyFrame* pKF, const vector<MapPoint *> &vpMapPoints, const float th=3.0);
```

Project MapPoints into KeyFrame using a given Sim3 and search for duplicated MapPoints.

```
int Fuse(KeyFrame* pKF, cv::Mat Scw, const std::vector<MapPoint*> &vpPoints, float th, vector<MapPoint *> &vpReplacePoint);
```

others functions

```
public:

    static const int TH_LOW;
    static const int TH_HIGH;
    static const int HISTO_LENGTH;


protected:

    bool CheckDistEpipolarLine(const cv::KeyPoint &kp1, const cv::KeyPoint &kp2, const cv::Mat &F12, const KeyFrame *pKF);

    float RadiusByViewingCos(const float &viewCos);

    void ComputeThreeMaxima(std::vector<int>* histo, const int L, int &ind1, int &ind2, int &ind3);

    float mfNNratio;
    bool mbCheckOrientation;
};
```

9.FrameDrawer

包含更新frame,画frame函数

```
class FrameDrawer
{
public:
    FrameDrawer(Map* pMap);

    // Update info from the last processed frame.
    void Update(Tracking *pTracker);

    // Draw last processed frame.
    cv::Mat DrawFrame();

protected:

    void DrawTextInfo(cv::Mat &im, int nState, cv::Mat &imText);

    // Info of the frame to be drawn
    cv::Mat mIm;
    int N;
    vector<cv::KeyPoint> mvCurrentKeys;
    vector<bool> mvbMap, mvbVO;
    bool mbOnlyTracking;
    int mnTracked, mnTrackedVO;
    vector<cv::KeyPoint> mvIniKeys;
    vector<int> mvIniMatches;
    int mState;

    Map* mpMap;

    std::mutex mMutex;
};
```

10. Tracking 跟踪

获取图像并进行跟踪

9.FrameDrawer

GramImage

```
cv::Mat GrabImageStereo(const cv::Mat &imRectLeft,const cv::Mat &imRectRight, const double &timestamp);
cv::Mat GrabImageRGBD(const cv::Mat &imRGB,const cv::Mat &imD, const double &timestamp);
cv::Mat GrabImageMonocular(const cv::Mat &im, const double &timestamp);

void SetLocalMapper(LocalMapping* pLocalMapper);
void SetLoopClosing(LoopClosing* pLoopClosing);
void SetViewer(Viewer* pViewer);
```

更改补偿信息

```
void ChangeCalibration(const string &strSettingPath);
```

跟踪状态

- 系统未准备
- 还没有图像
- 没有初始化
- 跟踪OK
- 跟踪丢失

```
// Tracking states
enum eTrackingState{
    SYSTEM_NOT_READY=-1,
    NO_IMAGES_YET=0,
    NOT_INITIALIZED=1,
    OK=2,
    LOST=3
};

eTrackingState mState;
eTrackingState mLastProcessedState;
```

输入传感器,当前帧

```
// Input sensor:MONOCULAR, STEREO, RGBD
int mSensor;

// Current Frame
Frame mCurrentFrame;
cv::Mat mImGray;
```

Initialization Variables (Monocular) 初始化时前两帧相关变量

```
std::vector<int> mvIniLastMatches;
std::vector<int> mvIniMatches;// 跟踪初始化时前两帧之间的匹配
std::vector<cv::Point2f> mvbPrevMatched;
std::vector<cv::Point3f> mvIniP3D;
Frame mInitialFrame;
```

Lists used to recover the full camera trajectory at the end of the execution. Basically we store the reference keyframe for each frame and its relative transformation

```
list<cv::Mat> mlRelativeFramePoses;
list<KeyFrame*> mlpReferences;
list<double> mlFrameTimes;
list<bool> mlbLost;
```

其他函数

```
// Main tracking function. It is independent of the input sensor.
void Track();

// Map initialization for stereo and RGB-D
void StereoInitialization();

// Map initialization for monocular
void MonocularInitialization();
void CreateInitialMapMonocular();

void CheckReplacedInLastFrame();
bool TrackReferenceKeyFrame();
void UpdateLastFrame();
bool TrackWithMotionModel();

bool Relocalization();
```

```
void UpdateLocalMap();
void UpdateLocalPoints();
void UpdateLocalKeyFrames();

bool TrackLocalMap();
void SearchLocalPoints();

bool NeedNewKeyFrame();
void CreateNewKeyFrame();

// In case of performing only localization, this flag is true when there are no matches to
// points in the map. Still tracking will continue if there are enough matches with temporal points.
// In that case we are doing visual odometry. The system will try to do relocalization to recover
// "zero-drift" localization to the map.
bool mbVO;

//Other Thread Pointers
LocalMapping* mpLocalMapper;
LoopClosing* mpLoopClosing;

//ORB
// orb特征提取器，不管单目还是双目，mpORBextractorLeft都要用到
// 如果是双目，则要用到mpORBextractorRight
// 如果是单目，在初始化的时候使用mpIniORBextractor而不是mpORBextractorLeft，
// mpIniORBextractor属性中提取的特征点个数是mpORBextractorLeft的两倍
ORBextractor* mpORBextractorLeft, *mpORBextractorRight;
ORBextractor* mpIniORBextractor;

//BoW
ORBVocabulary* mpORBVocabulary;
KeyFrameDatabase* mpKeyFrameDB;

// Initalization (only for monocular)
// 单目初始器
Initializer* mpInitializer;

//Local Map
KeyFrame* mpReferenceKF;// 当前关键帧就是参考帧
std::vector<KeyFrame*> mvpLocalKeyFrames;
std::vector<MapPoint*> mvpLocalMapPoints;

// System
System* mpSystem;

//Drawers
Viewer* mpViewer;
FrameDrawer* mpFrameDrawer;
MapDrawer* mpMapDrawer;

//Map
Map* mpMap;

//Calibration matrix
cv::Mat mK;
cv::Mat mDistCoef;
float mbf;

//New KeyFrame rules (according to fps)
int mMinFrames;
int mMaxFrames;

// Threshold close/far points
// Points seen as close by the stereo/RGBD sensor are considered reliable
// and inserted from just one frame. Far points requiere a match in two keyframes.
float mThDepth;

// For RGB-D inputs only. For some datasets (e.g. TUM) the depthmap values are scaled.
float mDepthMapFactor;

//Current matches in frame
int mnMatchesInliers;

//Last Frame, KeyFrame and Relocalisation Info
KeyFrame* mpLastKeyFrame;
Frame mLastFrame;
unsigned int mnLastKeyFrameId;
unsigned int mnLastRelocFrameId;

//Motion Model
cv::Mat mVelocity;

//Color order (true RGB, false BGR, ignored if grayscale)
bool mbRGB;
```

```
list<MapPoint*> mlpTemporalPoints;
```

11.PnPsolver

直接看cpp实现部分

12_PoseEstimate 2d-2d

主程序

- 读取两张图片
- 寻找特征匹配点
- pose_estimation_2d2d (pts1, pts2, R, t)

```
int main ( int argc, char** argv )
{
    if ( argc != 3 )
    {
        cout<<"usage: pose_estimation_2d2d img1 img2"<<endl;
        return 1;
    }
    //-- 读取图像
    Mat img_1 = imread ( argv[1], CV_LOAD_IMAGE_COLOR );
    Mat img_2 = imread ( argv[2], CV_LOAD_IMAGE_COLOR );

    vector<KeyPoint> keypoints_1, keypoints_2;
    vector<DMatch> matches;
    find_feature_matches ( img_1, img_2, keypoints_1, keypoints_2, matches );
    cout<<"一共找到了"<<matches.size() <<"组匹配点"<<endl;

    //-- 估计两张图像间运动
    Mat R,t;
    pose_estimation_2d2d ( keypoints_1, keypoints_2, matches, R, t );

    //-- 验证E=t^R*scale
    Mat t_x = ( Mat_<double> ( 3,3 ) <<
                0,                      -t.at<double> ( 2,0 ),     t.at<double> ( 1,0 ),
                t.at<double> ( 2,0 ),      0,                      -t.at<double> ( 0,0 ),
                -t.at<double> ( 1.0 ),     t.at<double> ( 0,0 ),      0 );

    cout<<"t^R="<<endl<<t_x*R<<endl;

    //-- 验证对极约束
    Mat K = ( Mat_<double> ( 3,3 ) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1 );
    for ( DMatch m: matches )
    {
        Point2d pt1 = pixel2cam ( keypoints_1[ m.queryIdx ].pt, K );
        Mat y1 = ( Mat_<double> ( 3,1 ) << pt1.x, pt1.y, 1 );
        Point2d pt2 = pixel2cam ( keypoints_2[ m.trainIdx ].pt, K );
        Mat y2 = ( Mat_<double> ( 3,1 ) << pt2.x, pt2.y, 1 );
        Mat d = y2.t() * t_x * R * y1;
        cout << "epipolar constraint = " << d << endl;
    }
    return 0;
}
```

pose_estimation_2d2d/2d-2d如何获得R,T

1. 将匹配的特征点放入findFundamentalMat()函数，获得fundamental_matrix
2. 根据fundamental_matrix以及相机光心和焦距获得essential_matrix
3. 根据essential_matrix通过recoverPose函数获得相机位姿

```
void pose_estimation_2d2d ( std::vector<KeyPoint> keypoints_1,
                            std::vector<KeyPoint> keypoints_2,
                            std::vector< DMatch > matches,
                            Mat& R, Mat& t )
{
    // 相机内参,TUM Freiburg2
    Mat K = ( Mat_<double> ( 3,3 ) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1 );

    //-- 把匹配点转换为vector<Point2f>的形式
    vector<Point2f> points1;
    vector<Point2f> points2;

    for ( int i = 0; i < ( int ) matches.size(); i++ )
    {
        points1.push_back ( keypoints_1[matches[i].queryIdx].pt );
        points2.push_back ( keypoints_2[matches[i].trainIdx].pt );
```

```
    }

    //-- 计算基础矩阵
    Mat fundamental_matrix;
    fundamental_matrix = findFundamentalMat ( points1, points2, CV_FM_8POINT );
    cout<<"fundamental_matrix is "<<endl<< fundamental_matrix<<endl;

    //-- 计算本质矩阵
    Point2d principal_point ( 325.1, 249.7 ); //相机光心, TUM dataset标定值
    double focal_length = 521;                      //相机焦距, TUM dataset标定值
    Mat essential_matrix;
    essential_matrix = findEssentialMat ( points1, points2, focal_length, principal_point );
    cout<<"essential_matrix is "<<endl<< essential_matrix<<endl;

    //-- 计算单应矩阵
    Mat homography_matrix;
    homography_matrix = findHomography ( points1, points2, RANSAC, 3 );
    cout<<"homography_matrix is "<<endl<<homography_matrix<<endl;

    //-- 从本质矩阵中恢复旋转和平移信息.
    recoverPose ( essential_matrix, points1, points2, R, t, focal_length, principal_point );
    cout<<"R is "<<endl<<R<<endl;
    cout<<"t is "<<endl<<t<<endl;

}
```

13_PoseEstimate slambook_3d2d

先看出程序代码:

1. 读取两幅图像 imread
2. find_feature_matches 找出匹配的特征点 keypoints, match 看能找到多少对匹配的特征点
3. 建立3D点. 并利用solvePnP计算R,T 从深度图像1中读取第一图像对应的深度图像.

   获得匹配点像素坐标对应的深度信息d

   pt1_camera -> pix2camera(keypoints_1,相机内参)

   pt1_3d = (pt1_camera.x * d, pt1_camera.y*d , d) pt2_2d = keypoints_2

   如此获得了pt1的3d坐标和对应的pt2的像素坐标

   即获得了keypoints_1对应的3d坐标和keypoints_2的像素坐标

   将这两个坐标放入solvePnP函数中求解R,T

   solvePnP (pts_3d, pts_2d, K, Mat(), r, t, false); // 调用OpenCV 的 PnP 求解, 可选择EPNP, DLS等方法

   获得r,t,并利用Rodrigues公式将r转化为矩阵形式R

   cv::Rodrigues (r, R); // r为旋转向量形式, 用Rodrigues公式转换为矩阵

   如此获得R,T

4. bundleAdjustment 如何进行bundleAdjustment

主程序

```
int main ( int argc, char** argv )
{
    if ( argc != 5 )
    {
        cout<<"usage: pose_estimation_3d2d img1 img2 depth1 depth2"<<endl;
        return 1;
    }
    //-- 读取图像
    Mat img_1 = imread ( argv[1], CV_LOAD_IMAGE_COLOR );
    Mat img_2 = imread ( argv[2], CV_LOAD_IMAGE_COLOR );

    vector<KeyPoint> keypoints_1, keypoints_2;
    vector<DMatch> matches;
    find_feature_matches ( img_1, img_2, keypoints_1, keypoints_2, matches );
    cout<<"一共找到了"<<matches.size() <<"组匹配点"<<endl;

    // 建立3D点
    Mat d1 = imread ( argv[3], CV_LOAD_IMAGE_UNCHANGED );       // 深度图为16位无符号数, 单通道图像
    Mat K = ( Mat_<double> ( 3,3 ) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1 );
    vector<Point3f> pts_3d;
    vector<Point2f> pts_2d;
    for ( DMatch m:matches )
    {
        ushort d = d1.ptr<unsigned short> (int ( keypoints_1[m.queryIdx].pt.y )) [ int ( keypoints_1[m.queryIdx].pt.x ) ];
        if ( d == 0 )   // bad depth
            continue;
        float dd = d/5000.0;
        Point2d p1 = pixel2cam ( keypoints_1[m.queryIdx].pt, K );
```

```
        pts_3d.push_back ( Point3f ( p1.x*dd, p1.y*dd, dd ) );
        pts_2d.push_back ( keypoints_2[m.trainIdx].pt );
    }

    cout<<"3d-2d pairs: "<<pts_3d.size() <<endl;

    Mat r, t;
    solvePnP ( pts_3d, pts_2d, K, Mat(), r, t, false ); // 调用OpenCV 的 PnP 求解，可选择EPNP，DLS等方法
    Mat R;
    cv::Rodrigues ( r, R ); // r为旋转向量形式，用Rodrigues公式转换为矩阵

    cout<<"R="<<endl<<R<<endl;
    cout<<"t="<<endl<<t<<endl;

    cout<<"calling bundle adjustment"<<endl;

    bundleAdjustment ( pts_3d, pts_2d, K, R, t );
}
```

find_feature_matches 如何进行特征匹配

```
void find_feature_matches ( const Mat& img_1, const Mat& img_2,
                            std::vector<KeyPoint>& keypoints_1,
                            std::vector<KeyPoint>& keypoints_2,
                            std::vector< DMatch >& matches )
{
    //-- 初始化
    Mat descriptors_1, descriptors_2;
    // used in OpenCV3
    Ptr<FeatureDetector> detector = ORB::create();
    Ptr<DescriptorExtractor> descriptor = ORB::create();
    // use this if you are in OpenCV2
    // Ptr<FeatureDetector> detector = FeatureDetector::create ( "ORB" );
    // Ptr<DescriptorExtractor> descriptor = DescriptorExtractor::create ( "ORB" );
    Ptr<DescriptorMatcher> matcher  = DescriptorMatcher::create ( "BruteForce-Hamming" );
    //-- 第一步:检测 Oriented FAST 角点位置
    detector->detect ( img_1,keypoints_1 );
    detector->detect ( img_2,keypoints_2 );

    //-- 第二步:根据角点位置计算 BRIEF 描述子
    descriptor->compute ( img_1, keypoints_1, descriptors_1 );
    descriptor->compute ( img_2, keypoints_2, descriptors_2 );

    //-- 第三步:对两幅图像中的BRIEF描述子进行匹配，使用 Hamming 距离
    vector<DMatch> match;
    // BFMatcher matcher ( NORM_HAMMING );
    matcher->match ( descriptors_1, descriptors_2, match );

    //-- 第四步:匹配点对筛选
    double min_dist=10000, max_dist=0;

    //找出所有匹配之间的最小距离和最大距离，即是最相似的和最不相似的两组点之间的距离
    for ( int i = 0; i < descriptors_1.rows; i++ )
    {
        double dist = match[i].distance;
        if ( dist < min_dist ) min_dist = dist;
        if ( dist > max_dist ) max_dist = dist;
    }

    printf ( "-- Max dist : %f \n", max_dist );
    printf ( "-- Min dist : %f \n", min_dist );

    //当描述子之间的距离大于两倍的最小距离时,即认为匹配有误.但有时候最小距离会非常小,设置一个经验值30作为下限.
    for ( int i = 0; i < descriptors_1.rows; i++ )
    {
        if ( match[i].distance <= max ( 2*min_dist, 30.0 ) )
        {
            matches.push_back ( match[i] );
        }
    }
}
```

pixel2camera

```
Point2d pixel2cam ( const Point2d& p, const Mat& K )
{
    return Point2d
           (
               ( p.x - K.at<double> ( 0,2 ) ) / K.at<double> ( 0,0 ),
               ( p.y - K.at<double> ( 1,2 ) ) / K.at<double> ( 1,1 )
```

```
        );
}
```

bundleAdjustment

```
void bundleAdjustment (
    const vector< Point3f > points_3d,
    const vector< Point2f > points_2d,
    const Mat& K,
    Mat& R, Mat& t )
{
    // 初始化g2o
    typedef g2o::BlockSolver< g2o::BlockSolverTraits<6,3> > Block;  // pose 维度为 6, landmark 维度为 3
    Block::LinearSolverType* linearSolver = new g2o::LinearSolverCSparse<Block::PoseMatrixType>(); // 线性方程求解器
    Block* solver_ptr = new Block ( linearSolver );     // 矩阵块求解器
    g2o::OptimizationAlgorithmLevenberg* solver = new g2o::OptimizationAlgorithmLevenberg ( solver_ptr );
    g2o::SparseOptimizer optimizer;
    optimizer.setAlgorithm ( solver );

    // vertex
    g2o::VertexSE3Expmap* pose = new g2o::VertexSE3Expmap(); // camera pose
    Eigen::Matrix3d R_mat;
    R_mat <<
          R.at<double> ( 0,0 ), R.at<double> ( 0,1 ), R.at<double> ( 0,2 ),
               R.at<double> ( 1,0 ), R.at<double> ( 1,1 ), R.at<double> ( 1,2 ),
               R.at<double> ( 2,0 ), R.at<double> ( 2,1 ), R.at<double> ( 2,2 );
    pose->setId ( 0 );
    pose->setEstimate ( g2o::SE3Quat (
                            R_mat,
                            Eigen::Vector3d ( t.at<double> ( 0,0 ), t.at<double> ( 1,0 ), t.at<double> ( 2,0 ) )
                        ) );
    optimizer.addVertex ( pose );

    int index = 1;
    for ( const Point3f p:points_3d )   // landmarks
    {
        g2o::VertexSBAPointXYZ* point = new g2o::VertexSBAPointXYZ();
        point->setId ( index++ );
        point->setEstimate ( Eigen::Vector3d ( p.x, p.y, p.z ) );
        point->setMarginalized ( true ); // g2o 中必须设置 marg 参见第十讲内容
        optimizer.addVertex ( point );
    }

    // parameter: camera intrinsics
    g2o::CameraParameters* camera = new g2o::CameraParameters (
        K.at<double> ( 0,0 ), Eigen::Vector2d ( K.at<double> ( 0,2 ), K.at<double> ( 1,2 ) ), 0
    );
    camera->setId ( 0 );
    optimizer.addParameter ( camera );

    // edges
    index = 1;
    for ( const Point2f p:points_2d )
    {
        g2o::EdgeProjectXYZ2UV* edge = new g2o::EdgeProjectXYZ2UV();
        edge->setId ( index );
        edge->setVertex ( 0, dynamic_cast<g2o::VertexSBAPointXYZ*> ( optimizer.vertex ( index ) ) );
        edge->setVertex ( 1, pose );
        edge->setMeasurement ( Eigen::Vector2d ( p.x, p.y ) );
        edge->setParameterId ( 0,0 );
        edge->setInformation ( Eigen::Matrix2d::Identity() );
        optimizer.addEdge ( edge );
        index++;
    }

    chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
    optimizer.setVerbose ( true );
    optimizer.initializeOptimization();
    optimizer.optimize ( 100 );
    chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
    chrono::duration<double> time_used = chrono::duration_cast<chrono::duration<double>> ( t2-t1 );
    cout<<"optimization costs time: "<<time_used.count() <<" seconds."<<endl;

    cout<<endl<<"after optimization:"<<endl;
    cout<<"T="<<endl<<Eigen::Isometry3d ( pose->estimate() ).matrix() <<endl;
}
```

14_PoseEstimation 3d3d

主程序流程

- 读取两张图片
- 寻找特征匹配点
- 读取两张深度图像
- 匹配特征点对应的深度信息 d1 d2
- 通过内参获得pix2camera 获得两张图对应特征点的相机坐标
- 相机坐标*d1，d2获得3d世界坐标
- 通过3d坐标获得R,T
- pose_estimation_3d3d (pts1, pts2, R, t)
- bundleAdjustment

主程序

```
int main ( int argc, char** argv )
{
    if ( argc != 5 )
    {
        cout<<"usage: pose_estimation_3d3d img1 img2 depth1 depth2"<<endl;
        return 1;
    }
    //-- 读取图像
    Mat img_1 = imread ( argv[1], CV_LOAD_IMAGE_COLOR );
    Mat img_2 = imread ( argv[2], CV_LOAD_IMAGE_COLOR );

    vector<KeyPoint> keypoints_1, keypoints_2;
    vector<DMatch> matches;
    find_feature_matches ( img_1, img_2, keypoints_1, keypoints_2, matches );
    cout<<"一共找到了"<<matches.size() <<"组匹配点"<<endl;

    // 建立3D点
    Mat depth1 = imread ( argv[3], CV_LOAD_IMAGE_UNCHANGED );       // 深度图为16位无符号数，单通道图像
    Mat depth2 = imread ( argv[4], CV_LOAD_IMAGE_UNCHANGED );       // 深度图为16位无符号数，单通道图像
    Mat K = ( Mat_<double> ( 3,3 ) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1 );
    vector<Point3f> pts1, pts2;

    for ( DMatch m:matches )
    {
        ushort d1 = depth1.ptr<unsigned short> ( int ( keypoints_1[m.queryIdx].pt.y ) ) [ int ( keypoints_1[m.queryIdx].pt.x ) ];
        ushort d2 = depth2.ptr<unsigned short> ( int ( keypoints_2[m.trainIdx].pt.y ) ) [ int ( keypoints_2[m.trainIdx].pt.x ) ];
        if ( d1==0 || d2==0 )   // bad depth
            continue;
        Point2d p1 = pixel2cam ( keypoints_1[m.queryIdx].pt, K );
        Point2d p2 = pixel2cam ( keypoints_2[m.trainIdx].pt, K );
        float dd1 = float ( d1 ) /5000.0;
        float dd2 = float ( d2 ) /5000.0;
        pts1.push_back ( Point3f ( p1.x*dd1, p1.y*dd1, dd1 ) );
        pts2.push_back ( Point3f ( p2.x*dd2, p2.y*dd2, dd2 ) );
    }

    cout<<"3d-3d pairs: "<<pts1.size() <<endl;
    Mat R, t;
    pose_estimation_3d3d ( pts1, pts2, R, t );
    cout<<"ICP via SVD results: "<<endl;
    cout<<"R = "<<R<<endl;
    cout<<"t = "<<t<<endl;
    cout<<"R_inv = "<<R.t() <<endl;
    cout<<"t_inv = "<<-R.t() *t<<endl;

    cout<<"calling bundle adjustment"<<endl;

    bundleAdjustment( pts1, pts2, R, t );

    // verify p1 = R*p2 + t
    for ( int i=0; i<5; i++ )
    {
        cout<<"p1 = "<<pts1[i]<<endl;
        cout<<"p2 = "<<pts2[i]<<endl;
        cout<<"(R*p2+t) = "<<
            R * (Mat_<double>(3,1)<<pts2[i].x, pts2[i].y, pts2[i].z) + t
            <<endl;
        cout<<endl;
    }
}
```

如何通过3d-3d来估计位姿

pose_estimation_3d3d (pts1, pts2, R, t)

```
void pose_estimation_3d3d (
    const vector<Point3f>& pts1,
    const vector<Point3f>& pts2,
    Mat& R, Mat& t
)
{
    Point3f p1, p2;     // center of mass
    int N = pts1.size();
    for ( int i=0; i<N; i++ )
    {
        p1 += pts1[i];
        p2 += pts2[i];
    }
    p1 = Point3f( Vec3f(p1) /  N);
    p2 = Point3f( Vec3f(p2) / N);
    vector<Point3f>     q1 ( N ), q2 ( N ); // remove the center
    for ( int i=0; i<N; i++ )
    {
        q1[i] = pts1[i] - p1;
        q2[i] = pts2[i] - p2;
    }

    // compute q1*q2^T
    Eigen::Matrix3d W = Eigen::Matrix3d::Zero();
    for ( int i=0; i<N; i++ )
    {
        W += Eigen::Vector3d ( q1[i].x, q1[i].y, q1[i].z ) * Eigen::Vector3d ( q2[i].x, q2[i].y, q2[i].z ).transpose();
    }
    cout<<"W="<<W<<endl;

    // SVD on W
    Eigen::JacobiSVD<Eigen::Matrix3d> svd ( W, Eigen::ComputeFullU|Eigen::ComputeFullV );
    Eigen::Matrix3d U = svd.matrixU();
    Eigen::Matrix3d V = svd.matrixV();
    cout<<"U="<<U<<endl;
    cout<<"V="<<V<<endl;

    Eigen::Matrix3d R_ = U* ( V.transpose() );
    Eigen::Vector3d t_ = Eigen::Vector3d ( p1.x, p1.y, p1.z ) - R_ * Eigen::Vector3d ( p2.x, p2.y, p2.z );

    // convert to cv::Mat
    R = ( Mat_<double> ( 3,3 ) <<
          R_ ( 0,0 ), R_ ( 0,1 ), R_ ( 0,2 ),
          R_ ( 1,0 ), R_ ( 1,1 ), R_ ( 1,2 ),
          R_ ( 2,0 ), R_ ( 2,1 ), R_ ( 2,2 )
        );
    t = ( Mat_<double> ( 3,1 ) << t_ ( 0,0 ), t_ ( 1,0 ), t_ ( 2,0 ) );
}
```

bundleAdjustment

如何进行bundleAdjustment

```
void bundleAdjustment (
    const vector< Point3f >& pts1,
    const vector< Point3f >& pts2,
    Mat& R, Mat& t )
{
    // 初始化g2o
    typedef g2o::BlockSolver< g2o::BlockSolverTraits<6,3> > Block;  // pose维度为 6, landmark 维度为 3
    Block::LinearSolverType* linearSolver = new g2o::LinearSolverEigen<Block::PoseMatrixType>(); // 线性方程求解器
    Block* solver_ptr = new Block( linearSolver );      // 矩阵块求解器
    g2o::OptimizationAlgorithmGaussNewton* solver = new g2o::OptimizationAlgorithmGaussNewton( solver_ptr );
    g2o::SparseOptimizer optimizer;
    optimizer.setAlgorithm( solver );

    // vertex
    g2o::VertexSE3Expmap* pose = new g2o::VertexSE3Expmap(); // camera pose
    pose->setId(0);
    pose->setEstimate( g2o::SE3Quat(
        Eigen::Matrix3d::Identity(),
        Eigen::Vector3d( 0,0,0 )
    ) );
    optimizer.addVertex( pose );

    // edges
    int index = 1;
    vector<EdgeProjectXYZRGBDPoseOnly*> edges;
    for ( size_t i=0; i<pts1.size(); i++ )
    {
        EdgeProjectXYZRGBDPoseOnly* edge = new EdgeProjectXYZRGBDPoseOnly(
            Eigen::Vector3d(pts2[i].x, pts2[i].y, pts2[i].z) );
```

```
        edge->setId( index );
        edge->setVertex( 0, dynamic_cast<g2o::VertexSE3Expmap*> (pose) );
        edge->setMeasurement( Eigen::Vector3d(
            pts1[i].x, pts1[i].y, pts1[i].z) );
        edge->setInformation( Eigen::Matrix3d::Identity()*1e4 );
        optimizer.addEdge(edge);
        index++;
        edges.push_back(edge);
    }

    chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
    optimizer.setVerbose( true );
    optimizer.initializeOptimization();
    optimizer.optimize(10);
    chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
    chrono::duration<double> time_used = chrono::duration_cast<chrono::duration<double>>(t2-t1);
    cout<<"optimization costs time: "<<time_used.count()<<" seconds."<<endl;

    cout<<endl<<"after optimization:"<<endl;
    cout<<"T="<<endl<<Eigen::Isometry3d( pose->estimate() ).matrix()<<endl;

}
```

图优化定义图的边 g2o edge

```
class EdgeProjectXYZRGBDPoseOnly : public g2o::BaseUnaryEdge<3, Eigen::Vector3d, g2o::VertexSE3Expmap>
{
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
    EdgeProjectXYZRGBDPoseOnly( const Eigen::Vector3d& point ) : _point(point) {}

    virtual void computeError()
    {
        const g2o::VertexSE3Expmap* pose = static_cast<const g2o::VertexSE3Expmap*> ( _vertices[0] );
        // measurement is p, point is p'
        _error = _measurement - pose->estimate().map( _point );
    }

    virtual void linearizeOplus()
    {
        g2o::VertexSE3Expmap* pose = static_cast<g2o::VertexSE3Expmap *>(_vertices[0]);
        g2o::SE3Quat T(pose->estimate());
        Eigen::Vector3d xyz_trans = T.map(_point);
        double x = xyz_trans[0];
        double y = xyz_trans[1];
        double z = xyz_trans[2];

        _jacobianOplusXi(0,0) = 0;
        _jacobianOplusXi(0,1) = -z;
        _jacobianOplusXi(0,2) = y;
        _jacobianOplusXi(0,3) = -1;
        _jacobianOplusXi(0,4) = 0;
        _jacobianOplusXi(0,5) = 0;

        _jacobianOplusXi(1,0) = z;
        _jacobianOplusXi(1,1) = 0;
        _jacobianOplusXi(1,2) = -x;
        _jacobianOplusXi(1,3) = 0;
        _jacobianOplusXi(1,4) = -1;
        _jacobianOplusXi(1,5) = 0;

        _jacobianOplusXi(2,0) = -y;
        _jacobianOplusXi(2,1) = x;
        _jacobianOplusXi(2,2) = 0;
        _jacobianOplusXi(2,3) = 0;
        _jacobianOplusXi(2,4) = 0;
        _jacobianOplusXi(2,5) = -1;
    }

    bool read ( istream& in ) {}
    bool write ( ostream& out ) const {}
protected:
    Eigen::Vector3d _point;
};
```