ORB-SLAM2 Notes

Charlie Li

2019.05.05

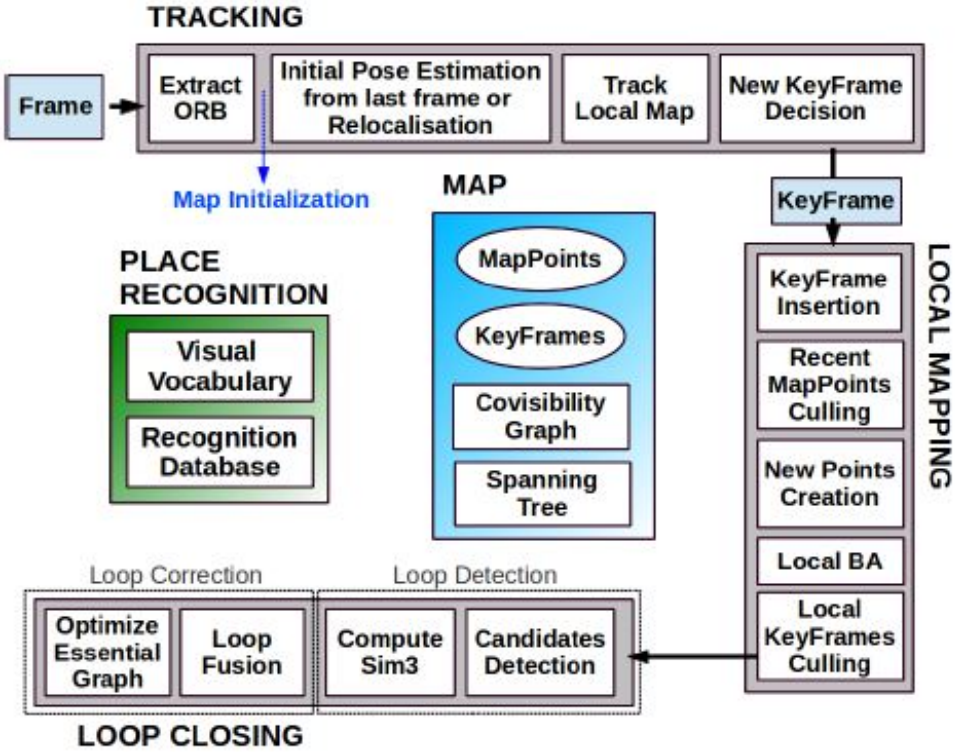
Table of Contents

I. Overview

This section gives the overview of ORB-SLAM2 system (Mur-Artal, Montiel, and Tardós 2015 (link)) (currently monocular only) and describes the general procedures inside each part of the system.

1. System Structure

· Monocular ORB-SLAM:



· 3 parallel threads

(1) Tracking thread

· Feature selection: ORB (Rublee et al. 2011 (link))

· Feature extraction: grid-based for consistent feature extraction within any image grid

· Map initialization

(2) Local Mapping thread

(3) Loop Closing thread

2. General Procedures

2.1 System Initialization

(1) Parse offline configurations

· Camera intrinsics

· Feature extraction parameters

· Viewer parameters

(2) Load bag-of-words vocabulary (Gálvez-López and Tardós 2012 (link)) trained offline

(3) Create threads

· Tracking thread (main thread)

· Local Mapping thread

- Loop Closing thread
- Viewer thread (optional)

2.2 Tracking Thread

For each input frame,

(1) Convert input image frame to grayscale
(2) Construct necessary info for current frame
    a) Extract ORB features
    b) Undistort keypoint coordinates given camera intrinsics
    c) Assign ORB features to each image grid
    d) Assign other necessary info
(3) Do map initialization if it is not initialized
    a) If there's no initial frame, set current frame as initial frame and proceed to next frame
    b) Match ORB features between initial frame and current frame
    c) If there're not enough matched keypoints, discard initial & current frame info and proceed to next frame
    d) Set up RANSAC framework, and for each set of keypoints, compute homography **H** and fundamental matrix **F** simultaneously
    e) Select the best **H** and the best **F** based on minimal symmetric transfer errors of the matched keypoints between the 2 frames
    f) Compute a score according to the best **H** and **F**, and use either **H** or **F** to reconstruct 3D points and camera pose
    g) Create initial map based on 3D points and camera pose
(4) Track current frame and compute camera pose if map is initialized
    a) Do pose estimation if tracking is successful for last frame
    b) Do global relocalization if tracking is failed for last frame
    c) Track local map: find more map point correspondences in a local map if operation in a) or b) is successful, and optimize pose based on local map correspondences
    d) Update velocity data if tracking is successful for current frame after operations in a)/b) and c)
    e) Check whether set current frame as a new keyframe if tracking is successful
    f) Discard keypoint outliers of current frame if tracking is successful
    g) If tracking is failed soon after map initialization (e.g., inserted keyframe is not larger than 5), reset all info and restart map initialization
(5) Return current camera pose **Tcw** ($[\mathbf{R}|\mathbf{t}]_{3\times4}$)

2.3 Local Mapping Thread

The local mapping thread will always check whether there're new keyframes to be processed, and add new keyframes and new map points to the covisibility graph and map, respectively, as long as the ORB-SLAM2 system is not finished running.

The following is the main procedure for each loop in the thread:

(1) Do not accept new keyframes;
(2) Check whether there're new keyframes to be processed;
(3) If new keyframes do not exist, go to the next step, otherwise, for the oldest new keyframe in the list:
    a) Compute bag-of-words for current new keyframe;
    b) Associate map points to the new keyframe;
    c) Update links in the covisibility graph;
    d) Insert the new keyframe to the map;
    e) Cull map points;
    f) Triangulate new map points;
    g) Search neighbour keyframes of current new keyframe if there're no new keyframes to be processed for more keypoint matches, and update map points and connections in the covisibility graph;
    h) Perform *local* bundle adjustment if there're no new keyframes to be processed, and then cull redundant local keyframes;
    i) Add current new keyframe to loop closing thread.
(4) Stop processing if user requests, and exit the loop if the system is finished;
(5) Reset the thread if user requests;
(6) Exit the loop if the system is finished;
(7) Continue accepting new keyframes.

2.4 Loop Closing Thread

The loop closing thread will always check whether there're new keyframes to be processed, and detect whether the map contains loop. If a loop is detected, it will perform loop fusion and pose graph optimization to eliminate the loop, as long as the ORB-SLAM2 system is not finished running.

The following is the main procedure for each loop in the thread:

(1) Check whether there're new keyframes to be processed;
(2) If new keyframes do not exist, proceed to next step. Otherwise, for the oldest new keyframe in the keyframe queue:
    a) Detect whether there's a loop in the map based on the current new keyframe;
    b) If a loop is detected, compute similarity transformations ($Sim(3)$) from current keyframe to each loop candidate keyframe, and check whether there are enough keypoint matches to accept a loop;
    c) If a loop is confirmed, it will be corrected.
(3) Reset the thread if user requests;
(4) Exit the loop if the system is finished.

II. Tracking Thread

This Section gives detail on some specific procedures executed on the tracking thread.

1. Map Initialization (Monocular)

· Called by function `Tracking::MonocularInitialization()` if map is not initialized

1.1 ORB Feature Matching

· For map initialization, the matching scheme is implemented by function `ORBmatcher::SearchForInitialization()`

```
// 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);
```

· For each keypoint in initial frame:

a) Get all keypoints within its neighborhood in current frame

```
// F1: initial frame; F2: current frame
for(size_t i1=0, iend1=F1.mvKeysUn.size(); i1<iend1; i1++) {
    cv::KeyPoint kp1 = F1.mvKeysUn[i1];
    int level1 = kp1.octave;
    if(level1>0) continue;
    // neighborhood size is determined by windowSize
    vector<size_t> vIndices2 = F2.GetFeaturesInArea(
        vbPrevMatched[i1].x,vbPrevMatched[i1].y, windowSize,level1,level1);
}
```

b) Find best matching keypoint with minimum Hamming distance

```
for(size_t i1=0, iend1=F1.mvKeysUn.size(); i1<iend1; i1++) {
    // i1-th descriptor from initial frame
    cv::Mat d1 = F1.mDescriptors.row(i1);
    for(vector<size_t>::iterator vit=vIndices2.begin();
        vit!=vIndices2.end(); vit++) {
        size_t i2 = *vit;
        // i2-th descriptor from current frame
        cv::Mat d2 = F2.mDescriptors.row(i2);
        // compute Hamming distance between 2 descriptors
        int dist = DescriptorDistance(d1,d2);
    }
}
```

c) Screen matched keypoint using distance threshold and feature orientation info

```
// compute 3 most significant orientations for all matched keypoints
ComputeThreeMaxima(rotHist,HISTO_LENGTH,ind1,ind2,ind3);
for(int i=0; i<HISTO_LENGTH; i++) {
    // only keep matched keypoints within 3 most significant orientations
    if(i==ind1 || i==ind2 || i==ind3) continue;
    for(size_t j=0, jend=rotHist[i].size(); j<jend; j++) {
        int idx1 = rotHist[i][j];
        if(vnMatches12[idx1]>=0) {
            vnMatches12[idx1]=-1;
            nmatches--; // remove matched keypoints from other orientations
        }
    }
}
```

1.2 Parallel Computation of 2 Models

- Motivation for computing 2 models: compute pose for different scenes
  - Homography **H**: describe a planar scene better
  - Fundamental matrix **F**: describe a non-planar scene better
- Implemented inside function `Initializer::Initialize()`, which is called by function `Tracking::MonocularInitialization()`

```
// Launch threads to compute in parallel a fundamental matrix and a homography
vector<bool> vbMatchesInliersH, vbMatchesInliersF;
float SH, SF; // model score for H and F
cv::Mat H, F; // homography & fundamental matrix
thread threadH(&Initializer::FindHomography,this,ref(vbMatchesInliersH),
               ref(SH), ref(H));
thread threadF(&Initializer::FindFundamental,this,ref(vbMatchesInliersF),
               ref(SF), ref(F));
// Wait until both threads have finished
threadH.join();
threadF.join();
```

- Homography **H** computation by normalized DLT (Hartley and Zisserman 2004)

Objective

Given $n \geq 4$ 2D to 2D point correspondences $\{\mathbf{x}_i \leftrightarrow \mathbf{x}'_i\}$, determine the 2D homography matrix $\mathtt{H}$ such that $\mathbf{x}'_i = \mathtt{H}\mathbf{x}_i$.

Algorithm

(i) **Normalization of $\mathbf{x}$:** Compute a similarity transformation $\mathtt{T}$, consisting of a translation and scaling, that takes points $\mathbf{x}_i$ to a new set of points $\tilde{\mathbf{x}}_i$ such that the centroid of the points $\tilde{\mathbf{x}}_i$ is the coordinate origin $(0,0)^\mathsf{T}$, and their average distance from the origin is $\sqrt{2}$.

(ii) **Normalization of $\mathbf{x}'$:** Compute a similar transformation $\mathtt{T}'$ for the points in the second image, transforming points $\mathbf{x}'_i$ to $\tilde{\mathbf{x}}'_i$.

(iii) **DLT:** Apply algorithm 4.1(p91) to the correspondences $\tilde{\mathbf{x}}_i \leftrightarrow \tilde{\mathbf{x}}'_i$ to obtain a homography $\tilde{\mathtt{H}}$.

(iv) **Denormalization:** Set $\mathtt{H} = \mathtt{T}'^{-1}\tilde{\mathtt{H}}\mathtt{T}$.

· Implemented in function `Initializer::FindHomography()`

· Fundamental matrix **F** computation by normalized 8-point algorithm

Objective

Given $n \geq 8$ image point correspondences $\{\mathbf{x}_i \leftrightarrow \mathbf{x}'_i\}$, determine the fundamental matrix $\mathtt{F}$ such that $\mathbf{x}'^\mathsf{T}_i \mathtt{F} \mathbf{x}_i = 0$.

Algorithm

(i) **Normalization:** Transform the image coordinates according to $\hat{\mathbf{x}}_i = \mathtt{T}\mathbf{x}_i$ and $\hat{\mathbf{x}}'_i = \mathtt{T}'\mathbf{x}'_i$, where $\mathtt{T}$ and $\mathtt{T}'$ are normalizing transformations consisting of a translation and scaling.

(ii) Find the fundamental matrix $\hat{\mathtt{F}}'$ corresponding to the matches $\hat{\mathbf{x}}_i \leftrightarrow \hat{\mathbf{x}}'_i$ by

  (a) **Linear solution:** Determine $\hat{\mathtt{F}}$ from the singular vector corresponding to the smallest singular value of $\hat{\mathtt{A}}$, where $\hat{\mathtt{A}}$ is composed from the matches $\hat{\mathbf{x}}_i \leftrightarrow \hat{\mathbf{x}}'_i$ as defined in (11.3).

  (b) **Constraint enforcement:** Replace $\hat{\mathtt{F}}$ by $\hat{\mathtt{F}}'$ such that $\det \hat{\mathtt{F}}' = 0$ using the SVD (see section 11.1.1).

(iii) **Denormalization:** Set $\mathtt{F} = \mathtt{T}'^\mathsf{T}\hat{\mathtt{F}}'\mathtt{T}$. Matrix $\mathtt{F}$ is the fundamental matrix corresponding to the original data $\mathbf{x}_i \leftrightarrow \mathbf{x}'_i$.

· Implemented in function `Initializer::FindFundamental()`

1.3 Reconstruction of 3D Points and Camera Pose

· Inside function `Initializer::Initialize()`, which is called by function `Tracking::MonocularInitialization()`

· Using either homography **H** or fundamental matrix **F** for the reconstruction based on their model score **SH** and **SF**:

```
// Compute ratio of scores
float RH = SH/(SH+SF);
// Try to reconstruct from homography or fundamental depending on the
// ratio (0.40-0.45)
if(RH>0.40)
    return ReconstructH(vbMatchesInliersH,H,mK,R21,t21,vP3D,
                        vbTriangulated,1.0,50);
```

```
else //if(pF_HF>0.6)
    return ReconstructF(vbMatchesInliersF,F,mK,R21,t21,vP3D,
                        vbTriangulated,1.0,50);
```

1.3.1 Reconstruction Using Homography H

- Compute 8 possible pose solutions from $\mathbf{H}$ (implemented the method proposed in (Faugeras and Lustman 1988))
  - Let $\mathbf{A} = \mathbf{K}^{-1}\mathbf{H}\mathbf{K}$ where $\mathbf{K}$ is the camera calibration matrix
    - $\mathbf{H}$: homography between 2 2D pixel coordinates
    - $\mathbf{A}$: similar with $\mathbf{H}$ (share same eigenvalues), and denote homography between 2 3D camera coordinates
  - Assume $\mathbf{A} = d\mathbf{R} + \mathbf{t}\mathbf{n}^T$ where
    - Pose $\mathbf{P} = [\mathbf{R}|\mathbf{t}]$ (rotation & translation)
    - $\mathbf{n}$ is the normal to the image plane of reference frame
    - $d = \mathbf{n} \cdot \mathbf{X}_1$ where $\mathbf{X}_1$ is a 3D point on the image plane of reference frame, and d denotes the distance between the plane and the origin
  - Do SVD on $\mathbf{A}$, and extract 8 possible solutions to $\{\mathbf{R}, \mathbf{t}\}$ based on the 3 singular values
- Select the best pose from the 8 hypotheses
  - Compute number of "good" keypoints for each pose (under function `Initializer::CheckRT()`)
    - Triangulate all matched keypoints to get their 3D position w.r.t. corresponding camera origin (via SVD)
      - $\mathbf{x} \times \mathbf{P}\mathbf{X} = \mathbf{0} \rightarrow \mathbf{A}\mathbf{X} = \mathbf{0}$
      - Implemented in function `Initializer::Triangulate()`
        - $\mathbf{A}$ is constructed by 1 pair of matched keypoints and their corresponding camera poses (4 equations)
    - Get number of "good" keypoints for each pose
      - Criteria:
        - 3D space: low parallax and have positive depth for triangulated 3D points
        - 2D space: low reprojection error for triangulated 3D points
  - Select the best pose with most number of "good" points or discard all poses if there's no clear winner

1.3.2 Reconstruction Using Fundamental Matrix F

- Compute essential matrix $\mathbf{E}$: $\mathbf{E} = \mathbf{K}^T\mathbf{F}\mathbf{K}$ where $\mathbf{K}$ is camera calibration matrix
- Compute 4 motion (pose) hypotheses: (by calling function `Initializer::DecomposeE()`)
  - Assume pose for initial frame (from 1st frame to initial frame) is $\mathbf{P}_{ini|1} = [\mathbf{I}_{3\times3}|\mathbf{0}_{3\times1}]$
  - Assume SVD of essential matrix $\mathbf{E} = \mathbf{U}diag(1,1,0)\mathbf{V}^T = \begin{bmatrix}\mathbf{u}_1 & \mathbf{u}_2 & \mathbf{u}_3\end{bmatrix} diag(1,1,0)\mathbf{V}^T$
  - Pose of current frame can be:
    $\mathbf{P}_{cur|1} = [\mathbf{U}\mathbf{W}\mathbf{V}^T| + \mathbf{u}_3]$ or $[\mathbf{U}\mathbf{W}\mathbf{V}^T| - \mathbf{u}_3]$ or $[\mathbf{U}\mathbf{W}^T\mathbf{V}^T| + \mathbf{u}_3]$ or $[\mathbf{U}\mathbf{W}^T\mathbf{V}^T| - \mathbf{u}_3]$
    where $\mathbf{W} = \begin{bmatrix}0 & -1 & 0\\ 1 & 0 & 0\\ 0 & 0 & 1\end{bmatrix}$

- Select the best pose from the 4 hypotheses: same as the selection procedure from previous section

1.4 Initial Map Creation

- Set both initial and current frame as keyframes
- Compute bag-of-words for feature descriptors of the 2 keyframes
- Insert these 2 keyframes into the map
- Assign computed 3D keypoints as map points
  - Normal of a map point is computed in `MapPoint::UpdateNormalAndDepth()`: a 3x1 vector with the "average" direction from corresponding camera centers (the cameras that observed the current map point) to current 3D world position

$$\mathbf{n} = \frac{1}{k}\sum_{i=1}^{k}\frac{\mathbf{X}_w - \mathbf{O}_{c,i}}{\|\mathbf{X}_w - \mathbf{O}_{c,i}\|}$$

    where $\mathbf{n}$ is the normal vector, $\mathbf{X}_w$ is the current 3D robot world position, and $\mathbf{O}_{c,i}$ is the ith camera center in the world that observed the current map point
  - Distance range of a map point: also computed in `MapPoint::UpdateNormalAndDepth()`, the distance range is $[\|\mathbf{X}_w - \mathbf{O}_c\| \cdot s^{o-o_{max}}, \|\mathbf{X}_w - \mathbf{O}_c\| \cdot s^{o}]$ where $\mathbf{X}_w$ is the 3D world position of the map point, $\mathbf{O}_c$ is the camera center in the world coordinate system; s is the scale factor of ORB image pyramid, o is the octave level where the keypoint corresponding to the map point is observed, and o_{max} is the maximum octave level with minimal resolution in the pyramid
  - Distinctive descriptor for the map point: implemented in function `MapPoint::ComputeDistinctiveDescriptors()`
    Assume the map point is observed by k keyframes, there will be k ORB descriptors available. Compute Hamming distance between any 2 of them, so there're $k \times k$ distances. Regard the distances as a $k \times k$ matrix. For each row, sort the k distances in ascending order, and then find the median distance of each row. Find the minimum median distance among the k rows, and assume it is the ith row. Then, the distinctive descriptor for the map point is assigned with the ORB descriptor associated with the ith keyframe
- Add observations of these 2 keyframes
- Update connections in covisibility graph for these 2 keyframes
- Execute *full/global* bundle adjustment (BA) on current map
- Compute scene median depth and check whether depth info is valid
- If depth info is invalid or there're not enough map points for current frame, reset all info (nothing being initialized) and proceed to next frame
- Set miscellaneous info on related threads and set map status as initialized

2. Tracking

2.1 Pose Estimation

2.1.1 Pose Estimation Based on Constant Velocity Motion Model

- Procedures

(1) Directly compute new pose with velocity data
(2) Search for map point correspondences
(3) Optimize pose via *motion-only BA* (implemented in function `Optimizer::PoseOptimization()`)
(4) Discard map point outliers

· Condition:
(1) Tracking for last frame is successful
(2) Velocity is already computed

· Tracking is successful if enough matched keypoints are found

2.1.1.1 New Pose Computation

(1) Update info for last frame
· Set pose for last frame $\mathbf{P}_{k-1|1} = \mathbf{P}_{k-1|x}\mathbf{P}_{x|1}$ where the last keyframe is at frame x
· Create *visual odometry* map points if
· Last frame is not a keyframe, or
· The sensor is a non-monocular one, or
· ORB-SLAM2 system works in tracking-only mode

(2) Set pose for current frame $\mathbf{P}_{k|1} = \mathbf{v}\mathbf{P}_{k-1|1} = \mathbf{P}_{k-1|k-2}\mathbf{P}_{k-1|1} = \mathbf{P}_{k|k-1}\mathbf{P}_{k-1|1}$
· Constant velocity assumption: $\mathbf{v} = \mathbf{P}_{k-1|k-2} = \mathbf{P}_{k|k-1}$

2.1.1.2 Map Correspondences Searching: ORB Feature Matching

Implemented in function `ORBmatcher::SearchByProjection()`.

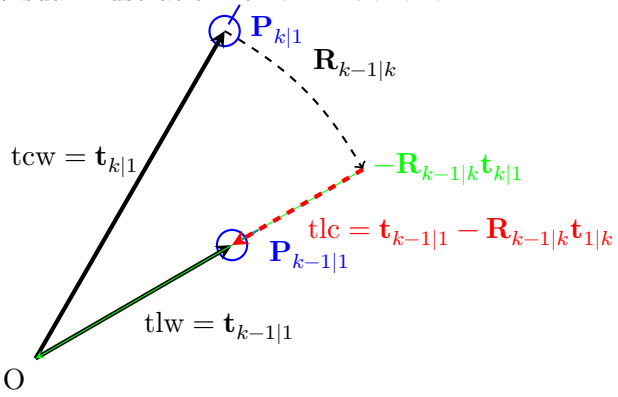
```
// Project MapPoints tracked in last frame into the current frame and
// search matches. Used to track from previous frame (Tracking).
int SearchByProjection(Frame &CurrentFrame, const Frame &LastFrame,
                       const float th, const bool bMono);
```

The Searching is done once or twice: if number of matched keypoints is not enough, there will be a 2nd searching with 2x window size. If still get less matched keypoints than expected, this tracking scheme is failed.

· Some pose-related variables:

```
// R_{k|1} & t_{k|1}
const cv::Mat Rcw = CurrentFrame.mTcw.rowRange(0,3).colRange(0,3);
const cv::Mat tcw = CurrentFrame.mTcw.rowRange(0,3).col(3);
// t in P_{1|k} = [R | t]: - R_{1|k} t_{k|1}
const cv::Mat twc = -Rcw.t()*tcw;
// R_{k-1|1} & t_{k-1|1}
const cv::Mat Rlw = LastFrame.mTcw.rowRange(0,3).colRange(0,3);
const cv::Mat tlw = LastFrame.mTcw.rowRange(0,3).col(3);
// tlc = R_{k-1|1} t_{1|k} + t_{k-1|1} = t_{k-1|1} - R_{k-1|k} t_{k|1}
// (see illustration below)
const cv::Mat tlc = Rlw*twc+tlw;
```

· Visual illustration for `cv::Mat tlc`:



· For each keypoint in previous frame,

a) Get candidate keypoints in current frame

```
// Project 3D map point in world coordinate of frame k-1
// to camera coordinate using pose of current frame k
cv::Mat x3Dw = pMP->GetWorldPos();
cv::Mat x3Dc = Rcw*x3Dw+tcw; // world->cam coordinate conversion

const float xc = x3Dc.at<float>(0);
const float yc = x3Dc.at<float>(1);
const float invzc = 1.0/x3Dc.at<float>(2);

// depth must be positive (can be seen by camera)
if(invzc<0)
    continue;

// 3D cam coord -> 2D pixel coord projection
float u = CurrentFrame.fx*xc*invzc+CurrentFrame.cx;
float v = CurrentFrame.fy*yc*invzc+CurrentFrame.cy;

// (u,v) must inside image border
if(u<CurrentFrame.mnMinX || u>CurrentFrame.mnMaxX)
    continue;
if(v<CurrentFrame.mnMinY || v>CurrentFrame.mnMaxY)
    continue;

// octave of the ORB keypoint
int nLastOctave = LastFrame.mvKeys[i].octave;

// Search in a window. Size depends on scale & pre-defined threshold
// (scale = base_scale ^ octave_num, num start from 0,
// base_scale = 1.2 by default)
float radius = th * CurrentFrame.mvScaleFactors[nLastOctave];
```

```
// candidate keypoints: within the grid cells covering
// [u-radius,u+radius]*[v-radius,v+radius] region
// for octaves [nLastOctave-1,nLastOctave+1]
vIndices2 = CurrentFrame.GetFeaturesInArea(u,v, radius, nLastOctave-1,
                                           nLastOctave+1);
```

- Bag-of-words data stored in keyframe only, so here the candidate keypoints are searched in a window
- Constant velocity: assume the keypoint position is determined by relative pose (velocity)

b) For each candidate keypoint in current frame,

- Find best matching keypoint with minimum Hamming distance
- Screen matched keypoint using distance threshold and feature orientation info

2.1.2 Pose Estimation Based on Reference Keyframe

- Procedures
  (1) Do ORB feature matching to find map point correspondences
    a) Compute bag-of-words of ORB descriptors for current frame
    b) For each keypoint in reference keyframe, find matching keypoints in current frame (implemented in function `ORBmatcher::SearchByBoW()`)

```
// Search matches between MapPoints in a KeyFrame and ORB in a Frame.
// Brute force constrained to ORB that belong to the same vocabulary
// node (at a certain level). Used in Relocalisation and Loop Detection
int SearchByBoW(KeyFrame *pKF, Frame &F,
                std::vector<MapPoint*> &vpMapPointMatches);
```

    - Use bag-of-words to get candidate keypoints
      - Only consider candidates of the **same node ID** for both keyframe and current frame
      - Advantage: more efficient than brute-force searching
    - Find best matching candidate keypoint with minimum Hamming distance
    - Screen matched keypoint using the same criteria described in ORB feature matching for map initialization part

  (2) Set pose of current frame the same as the last frame
  (3) Optimize pose via *motion-only BA*
- Conditions:
  (1) No velocity is assigned
  (2) Robot is just relocalized after tracking is lost
  (3) Tracking is failed for last frame under constant velocity motion model
- Tracking is successful if enough matched keypoints are found

2.2 Global Relocalization

- Procedures
  (1) Compute bag-of-words of ORB descriptors for current frame
  (2) Find keyframe candidates from keyframe database for relocalization
    - Criteria for candidates: keyframe that shares enough words with current frame under bag-of-words representation
  (3) For each candidate keyframe,
    a) Search for keypoint correspondence using ORB bag-of-words as described here (implemented by function `ORBmatcher::SearchByBoW()`)
    b) Set up a PnP solver for the (keyframe, current frame) pair if enough correspondences are found
  (4) For each PnP solver, find pose of current frame ($\mathbf{P}_{k|1}$):
    a) Run a few iterations based on sets of randomly-chosen matched keypoints (RANSAC framework), and for each set of keypoints:
      - Compute the camera pose using EPnP (Lepetit, Moreno-Noguer, and Fua 2008) algorithm
      - Find the keypoints with sufficiently low reprojection error based on computed pose, and regard them as new inliers
      - If there are enough inliers, refine the pose using EPnP based on all available inliers
      - Update inlier info (number of inliers) based on refined pose
      - If there are enough inliers, return the refined pose
      - If there are not enough inliers, go to the next iteration
    b) After all the iterations, if there are enough inliers, return the current computed pose
    c) If there are not enough inliers, return an empty pose
  (5) For each candidate keyframe, optimize the computed pose from step (4) using *motion-only BA* if the pose is not empty
    a) If there are too few keypoint correspondence inliers after pose optimization, continue processing next candidate keyframe
    b) If there are not too few and still not enough keypoint correspondence inliers after pose optimization, search for additional correspondences and optimize current pose using *motion-only BA* for at most 2 iterations
    c) If there are now enough keypoint correspondence inliers, record current camera pose and stop further processing as the relocalization is already successful
  (6) Relocalization is failed if no qualified pose is computed

2.2.1 Additional Keypoint Correspondence Searching

- Implemented in an overloaded version of function `ORBmatcher::SearchByProjection()`) by projecting map points seen in keyframe into current frame and matching between 2D image coordinates of keyframe and current frame

```
// Project MapPoints seen in KeyFrame into the Frame and search matches.
// Used in relocalisation (Tracking)
int SearchByProjection(Frame &CurrentFrame, KeyFrame* pKF,
```

```
                        const std::set<MapPoint*> &sAlreadyFound,
                        const float th, const int ORBdist);
```

- Differences between this scheme and that described above
  - Different search window

```
const float maxDistance = pMP->GetMaxDistanceInvariance();
const float minDistance = pMP->GetMinDistanceInvariance();
// Depth must be inside the scale pyramid of the image
if(dist3D<minDistance || dist3D>maxDistance) continue;
// have to predict the scale of the keypoint in the pyramid
int nPredictedLevel = pMP->PredictScale(dist3D,&CurrentFrame);
// Search in a window
const float radius = th*CurrentFrame.mvScaleFactors[nPredictedLevel];
// candidate keypoints: within the grid cells covering
// [u-radius,u+radius]*[v-radius,v+radius] region
// for octaves [nPredictedLevel-1,nPredictedLevel+1]
const vector<size_t> vIndices2 =
    CurrentFrame.GetFeaturesInArea(u, v, radius, nPredictedLevel-1,
                                   nPredictedLevel+1);
```

2.3 Local Map Tracking

- Procedures
  (1) Update local map for current frame
    a) Update keyframe info of local map
    b) Update map point info of local map
  (2) Search for more keypoint correspondences in updated local map
  (3) Optimize pose $\mathbf{P}_{k|1}$ of current frame via *motion-only BA*
  (4) Check if tracking is successful based on the newly optimized pose

2.3.1 Keyframe Info Update of Local Map

- Include keyframes by which at least one map point in current frame is observed
- Include keyframes that are neighbors of previously included keyframes: for each previously included keyframe,
  (1) Include neighbors in the covisibility graph that are closest to current keyframe
    - 10 such neighbors for actual implementation
  (2) Include extra neighbors that are the childs of current keyframe in the spanning tree generated from the covisibility graph
  (3) Include extra neighbors that are the parents of current keyframe in the spanning tree generated from the covisibility graph
  (4) Stop including more keyframes if the number of them exceeds an upper limit (80 in actual implementation)

- Set keyframe having most shared keypoints with current frame as reference keyframe

2.3.2 Keypoint Correspondence Searching in Local Map

- Implemented in another overloaded version of function `ORBmatcher::SearchByProjection()`

```
// Search matches between Frame keypoints and projected MapPoints.
// Returns number of matches. Used to track the local map (Tracking).
int SearchByProjection(Frame &F, const std::vector<MapPoint*> &vpMapPoints,
                       const float th=3);
```

- Differences between this scheme and the previous described schemes (here and here)
  - No need to compute projected 2D image coordinates given input map points in the local map (projected coordinates already computed by function `Frame::isInFrustum()`)
  - Different search window

```
// get predicted scale level
const int &nPredictedLevel = pMP->mnTrackScaleLevel;
// The size of the window will depend on the viewing direction
float r = RadiusByViewingCos(pMP->mTrackViewCos);
// multiply threshold if it is not 1.0
if(bFactor) r*=th;
// candidate keypoints: within the grid cells covering
// [u-radius,u+radius]*[v-radius,v+radius] region
// for octaves [nPredictedLevel-1,nPredictedLevel]
// where u = pMP->mTrackProjX, v = pMP->mTrackProjY,
// radius = r*F.mvScaleFactors[nPredictedLevel]
const vector<size_t> vIndices =
    F.GetFeaturesInArea(pMP->mTrackProjX,pMP->mTrackProjY,
    r*F.mvScaleFactors[nPredictedLevel],nPredictedLevel-1,nPredictedLevel);
```

2.4 Velocity Data Update

Update velocity data of current frame using the poses of current and last frame:

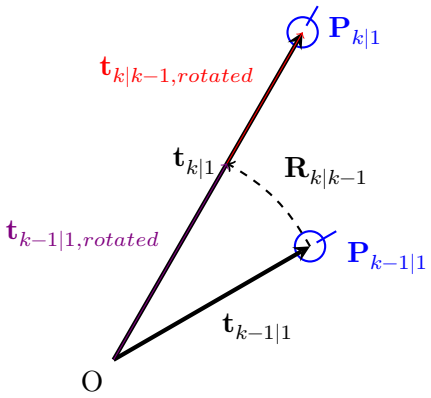
- Pose of last ($(k-1)$th) frame (from frame 1 to frame $(k-1)$) $\mathbf{P}_{k-1|1} = \begin{bmatrix} \mathbf{R}_{k-1|1} & \mathbf{t}_{k-1|1} \\ 0 & 1 \end{bmatrix}$
  - Rotation matrix $\mathbf{R}_{k-1|1}$ & translation vector $\mathbf{t}_{k-1|1}$ are accumulated from 1st frame to $(k-1)$th frame
  - Its inverse $\mathbf{P}_{k-1|1}^{-1} = \mathbf{P}_{1|k-1} = \begin{bmatrix} \mathbf{R}_{k-1|1}^T & -\mathbf{R}_{k-1|1}^T \mathbf{t}_{k-1|1} \\ 0 & 1 \end{bmatrix}$
- Pose of current kth frame from frame 1 $\mathbf{P}_{k|1} = \begin{bmatrix} \mathbf{R}_{k|1} & \mathbf{t}_{k|1} \\ 0 & 1 \end{bmatrix}$

· Velocity:

$$
\begin{aligned}
\mathbf{V} &= \mathbf{P}_{k|1}\mathbf{P}_{k-1|1}^{-1} \\
&= \begin{bmatrix} \mathbf{R}_{k|1}\mathbf{R}_{k-1|1}^{T} & -\mathbf{R}_{k|1}\mathbf{R}_{k-1|1}^{T}\mathbf{t}_{k-1|1} + \mathbf{t}_{k|1} \\ 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{R}_{k|1}\mathbf{R}_{1|k-1} & -\mathbf{R}_{k|1}\mathbf{R}_{1|k-1}\mathbf{t}_{k-1|1} + \mathbf{t}_{k|1} \\ 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{R}_{k|k-1} & \mathbf{t}_{k|1} - \mathbf{R}_{k|k-1}\mathbf{t}_{k-1|1} \\ 0 & 1 \end{bmatrix} \\
&= \mathbf{P}_{k|1}\mathbf{P}_{1|k-1} \\
&= \mathbf{P}_{k|k-1}
\end{aligned}
$$

· Relative pose from frame $(k-1)$ to frame k

· Illustration:



where

- $\mathbf{t}_{k-1|1,rotated} = \mathbf{R}_{k|k-1}\mathbf{t}_{k-1|1}$
- $\mathbf{t}_{k|k-1,rotated} = \mathbf{t}_{k|1} - \mathbf{t}_{k-1|1,rotated}$

· Implementation:

```
// inverse pose for last frame (camera to world)
// [R, t; 0, 1] -> [R^T, -R^T*t; 0, 1]
cv::Mat LastTwc = cv::Mat::eye(4,4,CV_32F);
mLastFrame.GetRotationInverse().copyTo(LastTwc.rowRange(0,3).colRange(0,3));
mLastFrame.GetCameraCenter().copyTo(LastTwc.rowRange(0,3).col(3));
// v = [R_{k|0}, t{k|0}; 0, 1]
//     * [R_{k-1|0}^T, -R_{k-1|0}^T * t_{k-1|0}; 0, 1]
//   = [R_{k|k-1}, t_{k|0} - R_{k|k-1} * t_{k-1|0}; 0, 1]
mVelocity = mCurrentFrame.mTcw*LastTwc;
```

2.5 New Keyframe Insertion

· Procedures

(1) Check whether to add current frame into keyframe set

(2) If current frame is needed to be added into the keyframe set, create a new keyframe using info of current frame

  · Set this new keyframe as reference keyframe to current frame

2.5.1 Keyframe Insertion Check

· Related conditions (monocular only):

(1) Local mapping is not freezed by a loop closure

(2) Enough frames (fps in actual implementation) have passed from latest relocalization

(3) Enough frames (fps in actual implementation) have passed from latest keyframe insertion

(4) Enough frames (0 in actual implementation) have passed and local mapping is idle

(5) Number of tracked keypoints is large enough (15 in actual implementation) but less than a portion (90% in actual implementation) of reference keyframe

· Procedures

  · Reject insertion if any of the conditions (1), (2), (5) is not met, or (3) and (4) are not met simultaneously

  · If keyframe insertion is currently not rejected, check if local mapping is idle

    · If local mapping is idle, insert a new keyframe

    · Otherwise, interrupt *local BA* running in local mapping thread for inserting future qualified keyframe as soon as possible, and then reject keyframe insertion

III. Local Mapping Thread

This section gives detail on specific procedures executed on this thread.

1. New Keyframe Processing

The new keyframe processing consists of the following procedure:

(1) Compute bag-of-words representation of the new keyframe;
(2) Associate map points and the new keyframe;
(3) Update covisibility graph;
(4) Add new keyframe into the global map.

1.1 Association Between Map Points and the New Keyframe

Procedure for each map point in the new keyframe that is observed by other keyframes in the map:

(1) Add observation of current new keyframe to the map point;
(2) Update *normal* and depth info of current map point (refer to the same procedure in initial map creation for details);
(3) Compute distinctive descriptor for the map point (refer to the same procedure in initial map creation for details).

1.2 Covisibility Graph Update

Covisibility graph is an undirected weighted graph with the following features:

- Vertex/Node: keyframe;
- Edge between 2 keyframes indicates that they share enough observations of map points (15 for actual implementation);
- Edge weight: number of common map points.

The update procedure is triggered by the newly processed keyframe, and is implemented in function `KeyFrame::UpdateConnections()`:

(1) For each map point in the keyframe, find all the other keyframes that observe it, and count the number of common map points between the new keyframe and other keyframes;
(2) Add edges between new keyframe and other keyframes that share enough map points (at least 15) with edge weight assigned;
(3) If no new edge is added, add an edge between new keyframe and the keyframe that shares the most map points with edge weight assigned.
(4) Expand the spanning tree for the keyframes in the map:
    a) Sort the edge weights of the connected keyframes in descending order;
    b) Set the keyframe with the most connections as the parent node in the spanning tree, and add current keyframe as its child.
        - Each keyframe can only be connected to one parent, thus the spanning tree expansion procedure is executed only once for each keyframe.

2. Redundant Map Point Removal

For each newly added keyframe, a list of recently added new map points are checked, and redundant map points in the list will be removed by the following criteria:

(1) The ratio of the number of a map point being an inlier of each keyframe observed it, to the number of the map point being visible from each keyframe, is less than a threshold (0.25 in actual implementation);

- The numerator of the ratio is increased in function `Tracking::TrackLocalMap()`, and the increase is triggered by the following code:

```
for (int i=0; i<mCurrentFrame.N; i++) // traverse all keypoint in the frame
    if(mCurrentFrame.mvpMapPoints[i]) // check if it is a map point
        if(!mCurrentFrame.mvbOutlier[i]) // check if it is an inlier
            mCurrentFrame.mvpMapPoints[i]->IncreaseFound();
```

- The denominator of the ratio is increased in function `Tracking::SearchLocalPoints()` within the function `Tracking::TrackLocalMap()`, and the increase is triggered by the following code:

```
// search for map points in current frame
for (vector<MapPoint*>::iterator vit=mCurrentFrame.mvpMapPoints.begin(),
     vend=mCurrentFrame.mvpMapPoints.end(); vit!=vend; vit++) {
    MapPoint* pMP = *vit;
    if(pMP && !pMp->isBad())
        pMP->IncreaseVisible();
}
// project local map points in frame and check its visibility
for (vector<MapPoint*>::iterator vit=mvpLocalMapPoints.begin(),
     vend=mvpLocalMapPoints.end(); vit!=vend; vit++) {
    MapPoint* pMP = *vit;
    // skip map points in current frame
    if(pMP->mnLastFrameSeen == mCurrentFrame.mnId) continue;
    // skip bad point
    if(pMP->isBad()) continue;
    // Project (this fills MapPoint variables for matching)
    if(mCurrentFrame.isInFrustum(pMP,0.5))
        pMP->IncreaseVisible(); // increase the visible count
}
```

(2) The current keyframe is at least 2 frames ahead of the keyframe where a map point is first observed, and the map point can only be observed by no more than 2 frames (in monocular configuration).

The removal procedure is implemented in function `LocalMapping::MapPointCulling()` as follows: for each map point in the list of recently added map points,

(1) Remove the map point from the list if it is a bad point;

(2) If the criteria above is met, set the map point as a bad point and remove it from the map, and then remove the map point from the list as well;
(3) If the map point does not meet the criteria above, and the current frame is at least 3 frames ahead of the keyframe that first observes the map point, remove the map point from the list.

3. New Map Point Creation

Implemented in function `LocalMapping::CreateNewMapPoints()`, the procedure for each newly added keyframe is as follows:

(1) Retrieve k (20 in actual implementation) best neighbor keyframes from the covisibility graph;
   · The *best* k keyframes: sort the keyframes that are connected with the current keyframe by edge weight in descending order, and select the first k of them.
(2) For each neighbor keyframe, compute the fundamental matrix between current keyframe and the neighbor keyframe;
(3) Search keypoint matches that fulfill epipolar restriction;
(4) Triangulate qualified keypoint matches to 3D map point;
(5) If the triangulated map point meet various restrictions, add it to the map.
   · Update observation info of the map point on current keyframe and each neighbor keyframe that successfully triangulated it if necessary;
   · For each neighbor keyframe that successfully triangulated the map point, add the map point info to it;
   · Compute distinctive descriptor of the map point (check here for details);
   · Update *normal* and depth of the map point (check here for details);
   · Add the map point to the list of recently added map points.

After selecting k best neighbor keyframes and before computing fundamental matrix between the current keyframe and each neighbor keyframe, the baseline between the 2 keyframes is checked. If it is too small, skip processing current neighbor keyframe.

3.1 Fundamental Matrix Computation

Given the current keyframe KF_1 with 3D world to 2D image transformation matrix $\mathbf{P}_1 = \mathbf{K}_1\mathbf{T}_1 = \mathbf{K}_1[\mathbf{R}_1|\mathbf{t}_1]$, and the neighbor keyframe KF_2 with the corresponding matrix $\mathbf{P}_2 = \mathbf{K}_2\mathbf{T}_2 = \mathbf{K}_2[\mathbf{R}_2|\mathbf{t}_2]$, the fundamental matrix from KF_2 to KF_1, noted as $\mathbf{F}_{12}$, is computed as follows (see section 9.2 of (Hartley and Zisserman 2004) for more details):

$$\mathbf{F}_{12} = \mathbf{K}_1^{-T}[\mathbf{t}_{12}]_\times \mathbf{R}_{12}\mathbf{K}_2^{-1}$$

where $[\mathbf{t}_{12}]_\times = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}_\times = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix}$ is a skew-symmetric matrix, and $\mathbf{R}_{12}$ and $\mathbf{t}_{12}$ is computed

from $\mathbf{T}_{12}$, the 3D camera to 3D camera coordinate mapping matrix (from KF_2 to KF_1):

$$\mathbf{T}_{12} = \mathbf{T}_1\mathbf{T}_2^T = \begin{bmatrix} \mathbf{R}_1 & \mathbf{t}_1 \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}_2^T & -\mathbf{R}_2^T\mathbf{t}_2 \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_1\mathbf{R}_2^T & -\mathbf{R}_1\mathbf{R}_2^T\mathbf{t}_2 + \mathbf{t}_1 \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_{12} & \mathbf{t}_{12} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

3.2 Keypoint Matches Search

Keypoint matches are searched between the current keyframe KF_1 and a neighbor keyframe KF_2, with a preciously computed fundamental matrix $\mathbf{F}_{12}$ (KF_2 to KF_1) and corresponding camera parameters $\mathbf{K}_1\mathbf{T}_1 = \mathbf{K}_1[\mathbf{R}_1|\mathbf{t}_1]$ and $\mathbf{K}_2\mathbf{T}_2 = \mathbf{K}_2[\mathbf{R}_2|\mathbf{t}_2]$. Implemented in function `ORBmatcher::SearchForTriangulation()`, the procedure is as follows:

(1) Compute epipole $\mathbf{e}_2$ in KF_2: $\mathbf{e}_2 = \mathbf{K}_2[\mathbf{R}_2|\mathbf{t}_2]\mathbf{X}_{cam1}$ where $\mathbf{X}_{cam1} = \mathbf{T}_1^{-1}\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}^T$ is the camera center of KF_1;
(2) For each keypoint in KF_1 that is not a map point yet,
  a) Find matched keypoints in KF_2 with the same node ID within bag-of-words model for feature matching acceleration;
  b) Find the best match in KF_2 from a) with the following criteria:
    · The keypoint is not a map point;
    · The keypoint is not already matched;
    · The Hamming distance between the descriptors of the 2 keypoints is below a preset threshold (50 in actual implementation);
    · The following equation is met: $\|\mathbf{e}_2 - \mathbf{x}_2\|^2 \geq 100s^o$ where $\mathbf{x}_2$ is the keypoint coordinate in image, s is ORB scale factor for its image pyramid, and o is the octave of the keypoint start from 0;
      · I think the equation means that the keypoint should not be near the epipole, and when this is the case, the actual 3D world point may be too near to the baseline of the 2 cameras, whose depth is too small;
      · PS: maybe the correct equation is $\|\mathbf{e}_2 - \mathbf{x}_2\| \geq 10s^o$.
    · The distance from the keypoint to the corresponding epipolar line in the image of KF_2 should be smaller than a threshold: $\frac{\mathbf{l}_2^T\mathbf{x}_2}{\sqrt{a^2+b^2}} < 1.8s^o$ where $\mathbf{l}_2 = \begin{bmatrix}a & b & c\end{bmatrix}^T$;
    · The orientation of the keypoint should be within the 3 most significant orientations (see here for more details).

3.3 Triangulation of New Map Points

New map points are triangulated from the foregoing searched keypoint matches between 2 views of current and a neighbor keyframe.

The procedure is as follows: for each keypoint match searched from the foregoing section,

(1) Check parallax between rays;
(2) If the parallax is appropriate, use direct linear transformation (DLT) and then SVD to triangulate a 3D point;
(3) Check if the triangulated point is in front of both cameras;
(4) Check if the reprojection error is appropriate for the triangulated point in both cameras;

(5) If the triangulate point meets all the above restrictions, add it to the map.

3.3.1 Parallax Checking

The goal is to make sure that the parallax for observing the world point, triangulated from the keypoint match between 2 keyframes, is not too low. That is, the angle between 2 rays that is emitted from the camera center to the world point is not near 0 degree.

First, compute the 3D coordinate of the keypoint with its origin at the camera center in both cameras corresponding to the 2 keyframes KF_1 and KF_2, assuming its depth is 1 ($Z_{c1} = Z_{c2} = 1$):

$$\mathbf{x}_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \frac{1}{Z_{c1}} \mathbf{K}_1 \mathbf{X}_{c1} = \begin{bmatrix} f_{x1} & 0 & c_{x1} \\ 0 & f_{y1} & c_{y1} \end{bmatrix} \begin{bmatrix} X_{c1} \\ Y_{c1} \\ 1 \end{bmatrix}$$

Therefore,

$$X_{c1} = \frac{x_1 - c_{x1}}{f_{x1}}, \quad Y_{c1} = \frac{y_1 - c_{x1}}{f_{y1}}.$$

The same is applied to the keypoint $\mathbf{x}_2$ in KF_2:

$$X_{c2} = \frac{x_2 - c_{x2}}{f_{x2}}, \quad Y_{c2} = \frac{y_2 - c_{x2}}{f_{y2}}.$$

The ray for the view in KF_1 is computed as follows:

$$\begin{aligned} \mathbf{r}_1 &= \mathbf{R}_1^T \mathbf{X}_{c1} \\ &= \mathbf{R}_1^T (\mathbf{R}_1 \mathbf{X}_{w1} + \mathbf{t}_1) \\ &= \mathbf{X}_{w1} - (-\mathbf{R}_1^T \mathbf{t}_1) \\ &= \mathbf{X}_{w1} - \mathbf{O}_{c1} \end{aligned}$$

where $\mathbf{X}_{w1}$ is the 3D world coordinate of the keypoint based on the camera coordinate with a depth of 1, $\mathbf{O}_{c1}$ is the camera center in the world coordinate system.

The same applied to the ray for the view in KF_2:

$$\mathbf{r}_2 = \mathbf{X}_{w2} - \mathbf{O}_{c2}$$

Then, we check the angle between the 2 rays as the angle of the parallax between the 2 views:

$$\theta = \arccos \frac{\mathbf{r}_1 \cdot \mathbf{r}_2}{\|\mathbf{r}_1\| \|\mathbf{r}_2\|}$$

The keypoint match can be used to triangulate new map point if $\theta \in (0, 90)$ degrees and $\cos\theta < 0.9998$ for actual implementation. That is, the angle must not be too small to be near 0 degree.

3.3.2 Triangulation

Given the 3D camera coordinates $\mathbf{X}_{c1} = (X_{c1}, Y_{c1}, 1)^T$ and $\mathbf{X}_{c2} = (X_{c2}, Y_{c2}, 1)^T$ with depths to be 1, based on each matched keypoint pair, and the corresponding 3×4 world to camera transformation matrices $\mathbf{T}_1$ and $\mathbf{T}_2$, the triangulation process is based on the equations below:

$$\mathbf{X}_{c1} = \mathbf{T}_1 \mathbf{X}_w, \quad \mathbf{X}_{c2} = \mathbf{T}_2 \mathbf{X}_w$$

where $\mathbf{X}_w$ is the triangulated 3D world point.

From the equation for camera 1 from KF_1, we can use cross-product to rewrite the equation as follows:

$$\mathbf{X}_{c1} \times \mathbf{T}_1 \mathbf{X}_w = \mathbf{0}$$

$$[\mathbf{X}_{c1}]_\times \begin{bmatrix} \text{——} \mathbf{t}_{11}^T \text{——} \\ \text{——} \mathbf{t}_{12}^T \text{——} \\ \text{——} \mathbf{t}_{13}^T \text{——} \end{bmatrix} \mathbf{X}_w = \mathbf{0}_{3\times 1}$$

$$\begin{bmatrix} 0 & -1 & Y_{c1} \\ 1 & 0 & -X_{c1} \\ -Y_{c1} & X_{c1} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{t}_{11}^T \mathbf{X}_w \\ \mathbf{t}_{12}^T \mathbf{X}_w \\ \mathbf{t}_{13}^T \mathbf{X}_w \end{bmatrix} = \mathbf{0}_{3\times 1}$$

$$\begin{bmatrix} -\mathbf{t}_{12}^T \mathbf{X}_w + Y_{c1} \mathbf{t}_{13}^T \mathbf{X}_w \\ \mathbf{t}_{11}^T \mathbf{X}_w - X_{c1} \mathbf{t}_{13}^T \mathbf{X}_w \\ * \end{bmatrix} = \mathbf{0}_{3\times 1}$$

$$\begin{bmatrix} -\mathbf{t}_{12}^T + Y_{c1} \mathbf{t}_{13}^T \\ \mathbf{t}_{11}^T - X_{c1} \mathbf{t}_{13}^T \\ * \end{bmatrix} \mathbf{X}_w = \mathbf{0}_{3\times 1}$$

As the 3rd row of the left-hand-side matrix is just a linear combination of the 1st and 2nd row, we discard the row, and then stack the similar 2 equations from camera 2, and change the row order. Then we get the following:

$$\mathbf{A}\mathbf{X}_w = \begin{bmatrix} X_{c1} \mathbf{t}_{13}^T - \mathbf{t}_{11}^T \\ Y_{c1} \mathbf{t}_{13}^T - \mathbf{t}_{12}^T \\ X_{c2} \mathbf{t}_{23}^T - \mathbf{t}_{21}^T \\ Y_{c2} \mathbf{t}_{23}^T - \mathbf{t}_{22}^T \end{bmatrix} \mathbf{X}_w = \mathbf{0}_{4\times 1}$$

where $\mathbf{t}_{ij}^T$ is the jth row of $\mathbf{T}_i$.

We then perform SVD decomposition on $\mathbf{A}$:

$$\begin{aligned} \mathbf{A}_{4\times 4} &= \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \\ &= \mathbf{U} \begin{bmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 \\ 0 & 0 & 0 & \sigma_4 \end{bmatrix} \begin{bmatrix} \text{——} \mathbf{v}_1^T \text{——} \\ \text{——} \mathbf{v}_2^T \text{——} \\ \text{——} \mathbf{v}_3^T \text{——} \\ \text{——} \mathbf{v}_4^T \text{——} \end{bmatrix} \end{aligned}$$

where $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \sigma_4$.

The triangulated 3D world point in a homogeneous coordinate is then assigned as $\mathbf{v}_4$, the 4th column vector of the right-singular matrix corresponding with the smallest singular value σ_4. Ideally, as $\mathbf{X}_w$ is mapped to the nullspace of $\mathbf{A}$, the smallest singular value σ_4 should be 0:

$$\mathbf{A}\mathbf{v}_4 = \sigma_4\mathbf{v}_4 = \mathbf{0}.$$

After the triangulated result is computed, check if the 4th dimension of $\mathbf{v}_4 = (v_1, v_2, v_3, v_4)^T$ is 0. Discard the triangulated result if $v_4 = 0$.

Finally, we normalize the triangulated result to have a scale factor of 1:

$$\mathbf{X}_w = (\frac{v_1}{v_4}, \frac{v_2}{v_4}, \frac{v_3}{v_4}, 1)^T$$

In actual implementation, the inhomogeneous coordinate of map points are stored.

3.3.3 Post-check on Triangulated Points

The triangulated points should meet the following restrictions:

(1) They should be in front of both cameras: $Z_c > 0$ where Z_c is 3rd dimension of $\mathbf{X}_c = \mathbf{R}\mathbf{X}_w + \mathbf{t}$;
(2) Their reprojection errors in both camera should be below some thresholds;

$$\|\mathbf{x} - \frac{1}{Z_c}\mathbf{K}(\mathbf{R}\mathbf{X}_w + \mathbf{t})\| \leq \chi\sigma$$

where

- $\mathbf{x}$ is the 2D image coordinate of the keypoint;
- Z_c is the Z-coordinate of the 3D camera coordinate;
- $\mathbf{K}$ is the camera intrinsics;
- $\mathbf{R}$ and $\mathbf{t}$ are the camera extrinsics;
- $\mathbf{X}_w$ is the triangulated point;
- $\chi^2 = 5.991$ is an outlier threshold based on χ^2 test assuming the standard deviation of measurement error is 1 pixel for homography;
- $\sigma = s^o$ where s is the scale factor of the ORB image pyramid, o is the octave where the keypoint $\mathbf{x}$ is observed, starting from 0;
- The above restriction is applied to both cameras in both keyframes.

(3) They should have scale consistency under the 2 cameras.

- The scale consistency check is implemented as follows:

```
// the ray emits from camera center of KF1/KF2 to the triangulated point
cv::Mat normal1 = x3D-Ow1;
float dist1 = cv::norm(normal1); // ||d||
cv::Mat normal2 = x3D-Ow2;
float dist2 = cv::norm(normal2);

if(dist1==0 || dist2==0) continue;
```

```
const float ratioDist = dist2/dist1;
const float ratioOctave = mpCurrentKeyFrame->mvScaleFactors[kp1.octave]
                          / pKF2->mvScaleFactors[kp2.octave];

// scale consistency check (ratioFactor = 1.5 * s^o)
if(ratioDist*ratioFactor<ratioOctave || ratioDist>ratioOctave*ratioFactor)
    continue;
```

- The equivalent restriction as an equation:

$$\frac{1}{1.5s^{o1}} \leq \frac{s^{o_2} \cdot \|\mathbf{X}_w - \mathbf{O}_{c2}\|}{s^{o_1} \cdot \|\mathbf{X}_w - \mathbf{O}_{c1}\|} = \frac{\|\mathbf{X}_w - \mathbf{O}_{c2}\|_{max}}{\|\mathbf{X}_w - \mathbf{O}_{c1}\|_{max}} \leq 1.5s^{o1}$$

  where s is the ORB scale factor of the image pyramid, $o1$ and $o2$ are the octave where the keypoint is observed in camera 1 and 2 of KF_1 and KF_2; $\mathbf{X}_w$ is the triangulated 3D world point; $\mathbf{O}_{c1}$ and $\mathbf{O}_{c2}$ are the camera center of camera 1 and camera 2 in world coordinate system. $\|\cdot\|_{max}$ is the max distance that is described before.
- Scale consistency: the distance from the triangulated point to the camera center should be consistent with the scale of the observed keypoint. The higher octave a keypoint is observed, the larger the range of the distance is.

4. Map Point Searching and Fusing in Neighbor Keyframes

If currently no more new keyframes are needed to be processed, find best neighbor keyframes of current keyframe, and search for more map points linked to each neighbor keyframe. Fuse the duplicate map points for each neighbor keyframe.

The procedure is as follows: for the current keyframe,

(1) Get the best neighbor keyframes (20 in actual implementation) from the covisibility graph, and for each neighbor keyframe, get its best neighbor keyframes (5 in actual implementation);
(2) Remove redundant and bad keyframes from (1), and the rest keyframes are the target keyframes for new map point searching and fusing scheme;
(3) Get all map points created from the current keyframe;
(4) Search for new keypoint matches and fuse duplicate ones for each target keyframe, based on map point info of current keyframe;
(5) For each target keyframes, get all map points created from the keyframe, and set the non-duplicated and non-bad ones as the candidate map points to be fused into the current keyframe;
(6) Search for new keypoint matches and fuse duplicate ones for current keyframe, based on the map point info of the candidate map points from all target keyframes;
(7) Update map point info (distinctive descriptor, normal, depth, …) in the current keyframe;
(8) Update connections in the covisibility graph based on the current keyframe.

4.1 Map Point Data Update and Fusion

Given an input keyframe and an input list of map points to be updated, the map point data in the input keyframe is updated, with fused/replaced better map points, and newly added map points. Also, the input list of map points may also be updated.

The data above is updated via a feature matching scheme by projecting input map points into image coordinate using camera instrinsics $\mathbf{K}$ and extrinsics $\mathbf{T} = [\mathbf{R}|\mathbf{t}]$ corresponding to the input keyframe.

The procedure, implemented in `ORBmatcher::Fuse()`, is as follows: for each map point in the input list of map points,

(1) Project the map point from 3D world coordinate to 2D image coordinate by $\mathbf{x} = \frac{1}{Z_c}\mathbf{K}(\mathbf{R}\mathbf{X}_w + \mathbf{t})$ where Z_c is the z-axis of camera coordinate $\mathbf{X}_c = \mathbf{R}\mathbf{X}_w + \mathbf{t}$. Discard the whole procedure and continue processing the next map point in the list if the following conditions are met:
   a) The camera coordinate $\mathbf{X}_c$ has a negative depth, i.e., $Z_c < 0$;
   b) The projected $\mathbf{x}$ is out of the image border;
   c) The distance d_x from the camera center $\mathbf{O}_c$ to the map point $\mathbf{X}_w$ is out of the distance range described here;
   d) The angle between the *normal* of the map point (which is described here) and the ray $\mathbf{r} = \mathbf{X}_w - \mathbf{O}_c$ is at least 60 degrees.

(2) Search candidate keypoints around $\mathbf{x}$:
   a) Predict an appropriate octave o_x as if it were a keypoint observed by the camera corresponding to the input keyframe: given the scale s of ORB image pyramid, the octave o of the keypoint which generates the map point, and the distance d from the center of the camera of the input keyframe, to the world position of the map point, the predicted octave is computed as follows:
   $$o_x = \lceil \log_s^{\frac{d \cdot s^o}{d_x}} \rceil = \lceil o + \log_s^{\frac{d}{d_x}} \rceil$$
   and $o_x \in [0, o_{max})$ where the ORB image pyramid has a maximum of o_{max} octaves;
   b) Compute search radius $r = th \cdot s^{o_x}$, where th is a parameter for controlling the radius. $th = 3$ in actual implementation;
   c) Search and get all candidate keypoints around $\mathbf{x}$ with the radius of r. If no candidate keypoint is found, discard the whole procedure and continue processing the next map point in the list.

(3) Find a best matching keypoint based on the projected $\mathbf{x}$ among the candidate keypoints:
   a) Only find a best match among candidates whose octave $o_m \in [o_x - 1, o_x + 1]$ ("m" in the subscript means "matched");
   b) Compute the distance between $\mathbf{x}$ and the candidate $\mathbf{x}_m$, and discard current candidate if the distance is too large:
   $$\frac{\|\mathbf{x} - \mathbf{x}_m\|}{s^{o_m}} > \chi$$
   where χ is described here;
   c) Compute the Hamming distance between the ORB descriptors corresponding to $\mathbf{x}$ and $\mathbf{x}_m$;
   d) A best match is the one with minimum Hamming distance.

(4) Update map point data of the map point itself and that in the input keyframe if the Hamming

distance between ORB descriptor of $\mathbf{x}$ and that of the best match $\mathbf{x}_m$ is lower than a threshold (50 in actual implementation):

a) Get the corresponding map point $\mathbf{X}_m$ of the best matched keypoint $\mathbf{x}_m$;

b) If a corresponding map point is found, compare the number of observations of $\mathbf{X}_w$ and $\mathbf{X}_m$, and replace the map point data of the map point having less observations with that of another map point;

· The number of observations of a map point is the number of cameras that observe the keypoint corresponding to the map point, or the number of keyframes that include the above keypoint data;

c) If no map point is found, add an observation of the map point by the input keyframe, and add the map point data into the keyframe.

5. Local Redundant Keyframe Removal

After *local* bundle adjustment is performed, redundant local keyframes are removed from the map, in order to make map information as compact as possible.

The procedure, implemented in `LocalMapping::KeyFrameCulling()`, is as follows: for current keyframe,

(1) Get all keyframes connected with the current keyframe in the covisibility graph;

(2) For each connected keyframe,

a) Count the total number of non-bad map points in the keyframe, denoted as n_t;

b) Count the number of distinct map points that are observed by at least 3 other keyframes at the same or finer scale (lower octave) in the ORB image pyramid, denoted as n;

c) Remove the keyframe from the map if $\frac{n}{n_t} > 0.9$.

Note: keyframes can only be removed by this procedure, and all the other procedures in loop closing thread that try to remove them can just trigger the removal if the criteria above is met.

5.1 Keyframe Removal

The removal of a keyframe from the map is implemented in function `KeyFrame::SetBadFlag()`.

The keyframe will not be immediately removed from the map. Only when loop edges are empty, and the keyframe is flagged as "to be erased", the keyframe will then be removed.

The procedure to remove a keyframe is as follows:

(1) Remove connections from all keyframes that are connected to the current keyframe;

(2) Remove connections from all map points linked to the current keyframe, that is, remove all observations from the current keyframe;

(3) Update the spanning tree of the keyframes without the node of the current keyframe;

(4) Compute the relative pose from the parent of this keyframe to this keyframe: $\mathbf{T}_{cp} = \mathbf{T}_{cw}\mathbf{T}_{pw}^{-1}$ where $\mathbf{T}_{cw}$ and $\mathbf{T}_{pw}$ are the poses of current and its parent keyframe, respectively ("c" stands for current keyframe, "p" for parent of current frame, and "w" for world);

(5) Set bad flag on this keyframe, and remove this keyframe from the map and the keyframe database.

5.1.1 Spanning Tree Update

If the current keyframe is to be removed, the children of the current keyframe in the spanning tree have to be assigned a different parent node.

The procedure is as follows:

(1) Add the parent of current keyframe into the set of parent candidates;
(2) While the set of children nodes of the current keyframe is not empty, for each iteration,
    a) Traverse each child keyframe, get its connected keyframes in the covisibility graph, and for each its connected keyframe that is also in the parent candidates set, check the edge weight between the child and the connected keyframe, and find the connected keyframe with the largest weight;
    b) The child with the largest edge weight between it and the connected keyframe, which is in the parent candidates set, is assigned this connected keyframe as its parent;
    c) The child assigned a new parent in this iteration is removed from the children set of the current keyframe, and is added to the set of parent candidates;
    d) If there're children that have no link with any keyframes in the parent candidates set, meanwhile all other children are assigned a new parent, proceed to the next step.
(3) If the set of children nodes of the current keyframe is still not empty, assign each child in the set the parent of the current keyframe as their parent.

IV. Loop Closing Thread

This section gives detail on specific procedures executed on this thread.

1. Loop Detection

The procedure for detecting loop in the map is as follows: for the first keyframe (oldest new keyframe) in the keyframe queue of the loop closing thread,

(1) Add the keyframe into the keyframe database if there're less than k keyframes in the map, or if less than k keyframes are added in the map after a latest loop is detected ($k = 10$ in actual implementation), and continue processing the next incoming keyframe. Otherwise, go to the next step;
(2) Compute similarity scores between the bag-of-words vector of current keyframe and that of each keyframe connected with the current keyframe in the covisibility graph, and maintain the lowest score among them as a baseline for loop detection;
(3) Find keyframes which are loop candidates based on the baseline score;
(4) If no loop candidates are found, try to erase the current keyframe, and add current keyframe to the keyframe database for subsequent loop detection. Also, clear the maintained list of loop candidate groups. Then continue processing the next incoming keyframe;
    · All operations that try to erase/remove the keyframe in this thread is implemented in function `KeyFrame::SetErase()`;
    · The keyframe removal operation is done only when a bad flag is set on the keyframe, and the flag will only be set by the local mapping thread when it tries to remove redundant keyframes (check here for details);
    · Only the redundant keyframes detected by the local mapping thread will be removed.
(5) If loop candidates are found, for each loop candidate keyframe, check the consistency between current and previously maintained loop candidates that are found based on previous incoming keyframes. If there're enough consistency, add the current loop candidate into a list for loop candidates with enough consistency;
(6) Add the current keyframe into the keyframe database;
(7) If the loop candidate list with enough consistency is not empty, a loop is detected; otherwise no loop is detected.

1.1 Similarity Score Computation

The similarity score between 2 bag-of-words vector $\mathbf{a} = (a_1, \dots, a_n)$ and $\mathbf{b} = (b_1, \dots, b_n)$ is related to the l_1-norm, and is computed as follows:

$$s_{l_1} = -\frac{1}{2} \sum_{i=1}^{n} \left(|a_i - b_i| - |a_i| - |b_i| \right)$$

The larger the score, the more similar the 2 vectors are. $s_{l_1} = \sum_i |a_i| = \|\mathbf{a}\|_1$ if $\mathbf{a} = \mathbf{b}$, and $s_{l_1} = 0$ if $\mathbf{a} = -\mathbf{b}$.

The score computation scheme is determined by the constructor of the bag-of-words class

`ORB_SLAM2::ORBVocabulary`, which has the actual type of `DBoW2::TemplatedVocabulary<DBoW2::FORB::TDescriptor, DBoW2::FORB>`. The default constructor of this class is defined as follows:

```
DBoW2::TemplatedVocabulary(int k = 10, int L = 5,
    WeightingType weighting = TF_IDF, ScoringType scoring = L1_NORM);
```

1.2 Loop Candidates Detection

Loop candidates detection scheme based on an input keyframe is implemented in function `KeyFrameDatabase::DetectLoopCandidates()`.

The procedure is as follows:

(1) Search all keyframes that share a word with the input keyframe based on their bag-of-words representations, and only include the ones that is not connected with the input keyframe in the covisibility graph into a list of keyframe candidates;
(2) If no keyframe candidate is found in step 1, terminate the loop candidate detection scheme, and no loop candidate is found;
(3) If there're keyframe candidates found in step 1, get the maximum number of shared words from the list of keyframe candidates, and denote it as n_{max};
(4) Compute similarity score between the input keyframe and keyframe candidates in the list whose number of shared words with the input keyframe is larger than $0.8 \cdot n_{max}$, and keep the candidates whose score is not less than the baseline score;
(5) If no keyframe candidates left after step 4, terminate the detection scheme, and no loop candidate is found;
(6) If there are still keyframe candidates left, for each keyframe candidate,
    a) Find their 10 best neighbor keyframes (connected keyframes with 10 largest edge weights);
    b) For each neighbor keyframe, if they have their similarity score computed, that is, if they are currently in the list of keyframe candidates, accumulate the similarity score of this neighbor keyframe, and find the neighbor keyframe with the highest similarity score. Then the accumulated similarity score is bound to the neighbor keyframe with the highest score, which is one of the keyframe candidates.
(7) Record the best accumulative similarity score among the keyframe candidates in the last step, and denote is as s_{cb};
(8) For the keyframe candidates in step 6, assign the keyframes whose bound accumulated similarity score $s_c \geq 0.75 \cdot s_{cb}$ as loop candidates.

1.3 Consistency Check for Loop Candidate

For each loop candidate keyframe, the consistency check is as follows:

(1) Get connected keyframes in the covisibility graph and group them and the loop candidate into a loop candidate group (set);
(2) The loop candidate group is compared with previously maintained list of loop candidate groups, and if the 2 groups have at least one common keyframe, the consistency index of the current

loop candidate group will be added by 1.

(3) If there're k consecutive input keyframes processed by loop closing thread that have loop candidates found, and at least one of their corresponding loop candidate groups have common keyframe with a certain one of the previously maintained list of loop candidate groups, the consistency index for the latest loop candidate group will be k, and if $k \geq th$, which is a consistency threshold (3 in actual implementation), the current loop candidate keyframe will be added into a list of loop candidate with enough consistency.

- If only less than k consecutive input keyframes meet the conditions above, the consistency index will be reset to 0 because the maintained list of loop candidate groups will be cleared.

2. Similarity Transformation Computation

After a loop is detected, in this step, a similarity transformation $\mathbf{S}_{cl} = \begin{bmatrix} s_{cl}\mathbf{R}_{cl} & \mathbf{t}_{cl} \\ \mathbf{0}^T & 1 \end{bmatrix} \in Sim(3)$ from each loop candidate keyframe to the current keyframe found above is computed. Its inverse $\mathbf{S}_{cl}^{-1} = \mathbf{S}_{lc}$ is also computed. They are noted as $\mathbf{S}$ if not specifically distinguished.

The procedure is as follows:

(1) Do ORB feature matching between current keyframe and each loop candidate keyframe by function `ORBmatcher::SearchByBoW()` (similar to the overloaded version described here);

```
// Search matches between MapPoints in a KeyFrame and ORB in a Frame.
// Brute force constrained to ORB that belong to the same vocabulary node
// (at a certain level), Used in Relocalisation and Loop Detection.
int SearchByBoW(KeyFrame *pKF1, KeyFrame* pKF2,
                std::vector<MapPoint*> &vpMatches12);
```

(2) For each loop candidate, if the number of matches from step 1 is not less than 20 (for actual implementation), set up a $Sim(3)$ solver based on data from current keyframe and the loop candidate keyframe, in order to compute the similarity transformation $\mathbf{S}$. Otherwise continue processing the next loop candidate;

(3) For each $Sim(3)$ solver based on a (current keyframe, loop candidate keyframe) pair, try to find a similarity transformation $\mathbf{S}$:

a) Run a few iterations based on sets of randomly-chosen map point correspondences (RANSAC framework), and for each set of map point correspondence:

- Compute similarity transformation $\mathbf{S}$ based on 3 pairs of randomly-chosen map point correspondences;
- Check if the pairs of map point are inliers after projected onto image plane using computed $\mathbf{S}$ and camera intrinsics $\mathbf{K}$:

$$\mathbf{x}_{c,c} = \frac{1}{Z_{c,c}}\mathbf{K}_c\mathbf{S}_{cl}\mathbf{X}_{w,l}$$

where $\mathbf{x}_{c,c}$ is the 2D image coordinate from camera of current keyframe, $\mathbf{X}_{w,l}$ is the 3D world coordinate of the loop candidate keyframe; $\mathbf{K}_c$ is the camera intrinsics of current

keyframe, and $Z_{c,c}$ is the Z coordinate of the camera coordinate in current keyframe mapped by the similarity transformation $\mathbf{S}_{cl}$ from loop candidate keyframe to current keyframe;

- Record the best $\mathbf{S}$ with most inliers for each iteration;
- If accumulated number of inliers exceeds a threshold (20 in actual implementation), return the best $\mathbf{S}$ computed; otherwise, no $\mathbf{S}$ result is computed.

b) After all the iterations, if no $\mathbf{S}$ is computed, continue running next $Sim(3)$ solver;

c) If $\mathbf{S}$ can be computed, find map point correspondences between current and the corresponding loop candidate;

d) Optimize computed $Sim(3)$ transformation $\mathbf{S}$ by previously found map point correspondences;

e) If inliers are no less than a threshold (20 in actual implementation), record current $\mathbf{S}$ info and stop further processing as the $Sim(3)$ transformation computation is successful. The corresponding loop candidate keyframe becomes a loop keyframe.

(4) If $\mathbf{S}$ with enough map point inliers is not computed, remove all the loop candidates and the current keyframe from the map, and continue processing the next input keyframe in the loop closing thread;

(5) If $\mathbf{S}$ is computed successfully in step 3, retrieve all distinct map points observed by the corresponding loop keyframe and all its neighbors in the covisibility graph;

(6) Find map point matches between the current keyframe and the loop keyframe, and if there are enough correspondences (40 in actual implementation), accept the detected loop and the corresponding loop keyframe. All the other loop candidate keyframes are removed from the map.

2.1 $Sim(3)$ Computation based on 3 Pairs of Map Point Correspondences

In ORB-SLAM2 system, $\mathbf{S}$ is computed based on 3 pairs of map point correspondences in current keyframe and in a loop candidate keyframe, using the (Horn 1987) (paper) method (see section 4.C for a summary of the method).

The computed $\mathbf{S}$ is a mapping from 3D world coordinate of one view to the camera coordinate of another view. That is,

$$\mathbf{X}_{c,c} = \mathbf{S}_{cl}\mathbf{X}_{w,l}, \quad \mathbf{X}_{c,l} = \mathbf{S}_{lc}\mathbf{X}_{w,c}$$

where $\mathbf{X}_{w,c}$ and $\mathbf{X}_{w,l}$ are the world coordinates in current and loop candidate keyframe, respectively; $\mathbf{X}_{c,c}$ and $\mathbf{X}_{c,l}$ are the camera coordinates in current and loop candidate keyframe, respectively.

The actual implementation is divided into the following steps:

(1) Compute the centroid and the point coordinates relative to the centroid (see section 2.C for details);

Given 3 map points $\{\mathbf{X}_{c,1}, \mathbf{X}_{c,2}, \mathbf{X}_{c,3}\}$ in the current keyframe, and 3 corresponding map points $\{\mathbf{X}_{l,1}, \mathbf{X}_{l,2}, \mathbf{X}_{l,3}\}$ in the loop candidate keyframe, compute their centroid $\mathbf{C}_*$ as follows:

$$\mathbf{C}_c = \frac{1}{3}\sum_{i=1}^{3}\mathbf{X}_{c,i}, \quad \mathbf{C}_l = \frac{1}{3}\sum_{i=1}^{3}\mathbf{X}_{l,i}$$

Let $\mathbf{P}_c = \begin{bmatrix}\mathbf{X}_{c,1} & \mathbf{X}_{c,2} & \mathbf{X}_{c,3}\end{bmatrix}$, and $\mathbf{P}_l = \begin{bmatrix}\mathbf{X}_{l,1} & \mathbf{X}_{l,2} & \mathbf{X}_{l,3}\end{bmatrix}$, the relative coordinate in the above matrix form is computed as follows:

$$\mathbf{P}'_c = \mathbf{P}_c - \mathbf{C}_c, \quad \mathbf{P}'_l = \mathbf{P}_l - \mathbf{C}_l$$

(2) Compute a 3×3 matrix $\mathbf{M}$ (see section 4.A for details);

$$\mathbf{M} = \mathbf{P}'_l \mathbf{P}'^T_c$$

(3) Compute a 4×4 matrix $\mathbf{N}$ which is the product of 2 4×4 matrices that are expanded from 2 unit quaternions (see section 4 and 4.A for details);

Let

$$\mathbf{M} = \begin{bmatrix} S_{xx} & S_{xy} & S_{xz} \\ S_{yx} & S_{yy} & S_{yz} \\ S_{zx} & S_{zy} & S_{zz} \end{bmatrix}$$

Then

$$\mathbf{N} = \mathbf{N}^T = \begin{bmatrix} S_{xx}+S_{yy}+S_{zz} & S_{yz}-S_{zy} & S_{zx}-S_{xz} & S_{xy}-S_{yx} \\ * & S_{xx}-S_{yy}-S_{zz} & S_{xy}+S_{yz} & S_{zx}+S_{xz} \\ * & * & -S_{xx}+S_{yy}-S_{zz} & S_{yz}+S_{zy} \\ * & * & * & -S_{xx}-S_{yy}+S_{zz} \end{bmatrix}$$

(4) Compute the rotation element $\mathbf{R}_{cl}$ in $\mathbf{S}_{cl}$ (see section 4.B for details);

Find the eigenvector $\mathbf{e}_{max}$ of $\mathbf{N}$ with the highest eigenvalue λ_{max}, which is the unit quaternion representation of the rotation element $\mathbf{R}_{cl}$ in the similarity transformation $\mathbf{S}_{cl} \in Sim(3)$ from current to loop candidate keyframe. The conversion from the unit quaternion $\mathbf{e}_{max}$ to $\mathbf{R}_{cl}$ is trivial.

(5) Compute the scale element s_{cl} in $\mathbf{S}_{cl}$ (see section 2.D and 2.E for details);

The actual implementation in ORB-SLAM2 system uses the 2nd formula in section 2.E of the paper for the asymmetry case where the map point measurements in one of the views are known with much better precision than those in the other view. The better view is assumed to be the view measured by the loop candidate keyframe.

$$s_{cl} = \frac{\sum_{i=1}^{3} \mathbf{X}'_{c,i} \cdot (\mathbf{R}_{cl}\mathbf{X}'_{l,i})}{\sum_{i=1}^{3} \|\mathbf{R}_{cl}\mathbf{X}'_{l,i}\|^2} = \frac{\sum_{i=1}^{3} \mathbf{X}'_{c,i} \cdot (\mathbf{R}_{cl}\mathbf{X}'_{l,i})}{\sum_{i=1}^{3} \|\mathbf{X}'_{l,i}\|^2}$$

(6) Compute the translation element $\mathbf{t}_{cl}$ in $\mathbf{S}_{cl}$ (see section 2.B and 2.C for details);

$$\mathbf{t}_{cl} = \mathbf{C}_c - s_{cl}\mathbf{R}_{cl}\mathbf{C}_l$$

(7) Combine the elements in $\mathbf{S}_{cl}$ together, and compute its inverse $\mathbf{S}_{cl}^{-1} = \mathbf{S}_{lc}$.

$$\mathbf{S}_{cl}^{-1} = \mathbf{S}_{lc} = \begin{bmatrix} s_{cl}^{-1}\mathbf{R}_{cl}^T & -s_{cl}^{-1}\mathbf{R}_{cl}^T\mathbf{t}_{cl} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

3. Loop Correction

V. Bundle Adjustment

1. Overview

Bundle adjustment (BA) in ORB-SLAM2 is implemented using g2o (Kümmerle et al. 2011). The operations on Lie group SE(3) and SO(3) is implemented by extending basic classes defined in g2o.

The followings are related materials:

- g2o: paper, documentation
- Lie group and related operations
  - Lie Groups for 2D and 3D Transformations by Ethan Eade
    - Related formulas are implemented in ORB-SLAM2
  - Chapter 7 on (Barfoot 2017) (link)

1.1 Nonlinear Graph Optimization Using Least-Squares

- Optimization scheme:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$$

  where

$$f(\mathbf{x}) = \sum_{i,j} \underbrace{\mathbf{e}_{ij}^T \mathbf{\Omega}_{ij} \mathbf{e}_{ij}}_{f_{ij}}$$

  - $f(\cdot) = \sum_{i,j} f_{ij}$: 1×1 scalar cost function
  - $\mathbf{e}$: error function with dimension $nk \times 1$ (assuming there are n terms in $f(\cdot)$)
  - Each $\mathbf{e}_{ij}$ term has the same $nk \times 1$ dimension
    - $\mathbf{e}_{ij} = \mathbf{e}(\mathbf{x}_i, \mathbf{x}_j, \mathbf{z}_{ij}) = \mathbf{e}(\mathbf{x})$
      - Vector error function that measures how well the parameter blocks (vertices in the g2o graph) $\mathbf{x}_i$ and $\mathbf{x}_j$ satisfy the constraint $\mathbf{z}_{ij}$ (edges in the g2o graph)
      - $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_n)^T$: the vector to be optimized with dimension $nk \times 1$
        - Dimension of $\mathbf{x}_i$: $k \times 1$
  - $\mathbf{\Omega}$: information matrix (inverse of covariance matrix $\mathbf{\Sigma}$) with dimension $nk \times nk$
- Iteration step: $\mathbf{x} \leftarrow \mathbf{x} \oplus \Delta\mathbf{x}$
  - The operator "$\oplus$" is the "addition" operation in a smooth manifold as the vector $\mathbf{x}$ may not be in an Euclidean space where the normal addition operator "$+$" is defined
    - For example, camera poses are in a $SE(3)$ Lie group, where the equivalent "addition" operation is the multiplication operation
- Find the best $\Delta\mathbf{x}$ for each iteration:
  - 1st-order Taylor expansion around current $\mathbf{x}_{ij}$ on error function $\mathbf{e}_{ij}$:

$$\mathbf{e}_{ij}(\mathbf{x}_i \oplus \Delta\mathbf{x}_i, \mathbf{x}_j \oplus \Delta\mathbf{x}_j) \approx \mathbf{e}_{ij}(\mathbf{x}) + \mathbf{J}_{ij}\Delta\mathbf{x} = \mathbf{e}_{ij} + \mathbf{J}_{ij}\Delta\mathbf{x}$$

    where

    - $\mathbf{e}_{ij} = (\mathbf{0}, \dots, \mathbf{e}_i, \dots, \mathbf{e}_j, \dots, \mathbf{0})^T$ is a $nk \times 1$ vector with each $\mathbf{e}_i$ a $k \times 1$ vector
    - $\mathbf{x}$ in $\mathbf{e}_{ij}$ is a *sparse* vector: $\mathbf{x} = (\mathbf{0}, \dots, \mathbf{x}_i, \dots, \mathbf{x}_j, \dots, \mathbf{0})^T$
    - $\mathbf{J}_{ij} = \frac{\partial \mathbf{e}_{ij}}{\partial \mathbf{x}}$ is the Jacobian of error function term $\mathbf{e}_{ij}$ w.r.t. parameter vector $\mathbf{x}$

· Expand $f_{ij}(\mathbf{x} \oplus \Delta\mathbf{x})$ using the above approximation (Gauss-Newton method):

$$\begin{aligned} f_{ij}(\mathbf{x} \oplus \Delta\mathbf{x}) &= \mathbf{e}_{ij}(\mathbf{x} \oplus \Delta\mathbf{x})^T \mathbf{\Omega}_{ij} \mathbf{e}_{ij}(\mathbf{x} \oplus \Delta\mathbf{x}) \\ &= (\mathbf{e}_{ij} + \mathbf{J}_{ij}\Delta\mathbf{x}_{ij})^T \mathbf{\Omega}_{ij} (\mathbf{e}_{ij} + \mathbf{J}_{ij}\Delta\mathbf{x}_{ij}) \\ &= \underbrace{\mathbf{e}_{ij}^T \mathbf{\Omega}_{ij} \mathbf{e}_{ij}}_{c_{ij}} + 2\underbrace{\mathbf{e}_{ij}^T \mathbf{\Omega}_{ij} \mathbf{J}_{ij}}_{\mathbf{b}_{ij}^T} \Delta\mathbf{x}_{ij} + \Delta\mathbf{x}_{ij}^T \underbrace{\mathbf{J}_{ij}^T \mathbf{\Omega}_{ij} \mathbf{J}_{ij}}_{\mathbf{H}_{ij}} \Delta\mathbf{x}_{ij} \end{aligned}$$

where

- c_{ij} is a 1×1 scalar data
- $\mathbf{b}_{ij}$ is a $nk \times 1$ vector
- $\mathbf{H}_{ij}$ is the approximated $nk \times nk$ Hessian matrix of f_{ij}

· Add each f_{ij} together:

$$\begin{aligned} f(\mathbf{x} \oplus \Delta\mathbf{x}) &\approx \sum_{i,j} c_{ij} + 2\mathbf{b}_{ij}^T \Delta\mathbf{x} + \Delta\mathbf{x}^T \mathbf{H}_{ij} \Delta\mathbf{x} \\ &= c + 2\mathbf{b}^T \Delta\mathbf{x} + \Delta\mathbf{x}^T \mathbf{H} \Delta\mathbf{x} \end{aligned}$$

where

- c is a scalar
- $\mathbf{b}$ is a $nk \times 1$ vector that is possibly dense (with each $\mathbf{b}_{ij}$ stacked together)
- $\mathbf{H}$ is the approximated $nk \times nk$ Hessian matrix of f

· To obtain a best $\Delta\mathbf{x}$ for each iteration:

- Gauss-Newton (GN) method: let $\partial f(\mathbf{x} \oplus \Delta\mathbf{x}) / \partial \Delta\mathbf{x} = 0$

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{b}$$

- Levenberg-Marquadt (LM) method: add a damping term based on GN method

$$(\mathbf{H} + \lambda\mathbf{I})\Delta\mathbf{x} = -\mathbf{b}$$

  where λ is a damping factor: as λ becomes higher, the direction of $\Delta\mathbf{x}$ will become nearer that of the negative gradient vector. Therefore it is a mixture of Gauss-Newton and gradient descent method.

· Solve the above linear system of form $\mathbf{A}\mathbf{x} = \mathbf{b}$, and then update $\mathbf{x}$ for current iteration ($\mathbf{x} \leftarrow \mathbf{x} \oplus \Delta\mathbf{x}$) until the convergence of the cost function f

2. Motion-only BA

Motion-only BA is implemented in `Optimizer::PoseOptimization()` function. In this optimization scheme, only the current camera pose ($\mathbf{T}_{cw}$) measurement is unknown and is to be optimized with all the 2D keypoint observations in image coordinates, meanwhile all the 3D map point measurements in world coordinates are fixed during the optimization.

· Procedures:

(1) Add the vertex and edges into g2o solver (Levenberg-Marquardt)

- Vertex: camera pose $\mathbf{T}_{cw} \in SE(3)$ (only for current frame)

        - Pose data is encapsulated in class `g2o::SE3Quat`
        - Vertex is defined in class `g2o::VertexSE3Expmap`
    - Edge: unary edge with only one vertex of the current pose
        - Information matrix: inverse of covariance matrix, which is a diagonal matrix, with its diagonal elements being observation variances (σ^2) determined by scale factor of the ORB image pyramid ($\sigma^2 = (\text{orb scale factor}^{\text{lvl}})^2$)

(2) Perform motion-only BA (10 iterations for actual implementation) several times (4 times for actual implementation)

- For each optimization, find edge outliers with χ^2 of error larger than a preset threshold ($\chi^2 = 5.991$ for actual implementation)
    - χ^2: an outlier threshold based on χ^2 test assuming the standard deviation of measurement error is 1 pixel for homograhy
- Exclude the outlier edges for the next optimization, and include the once-outlier edges if they are inliers again
- Count the current number of outliers

(3) Update optimized pose, and number of map correspondences between 2D keypoint observations and 3D map point measurements (the subtraction of map correspondences before optimization and the number of outliers of last optimization)

2.1 Error Function and Jacobian Computation

- Error function (implemented in function `g2o::EdgeSE3ProjectXYZOnlyPose::computeError()`):

$$\begin{aligned}\mathbf{e}_{ij}(\mathbf{x}) &= \mathbf{e}(\boldsymbol{\xi}) \\ &= \mathbf{x}_o - \frac{1}{Z_c}\mathbf{K}_{2\times4}\mathbf{T}_{cw}\mathbf{X}_w \\ &= \mathbf{x}_o - \frac{1}{Z_c}\mathbf{K}_{2\times4}\mathbf{X}_c \\ &= \mathbf{x}_o - \frac{1}{Z_c}\mathbf{K}_{2\times4}\exp(\boldsymbol{\xi}^\wedge)\mathbf{X}_w\end{aligned}$$

where

- $\mathbf{x}_o$ is the observation (2D undistorted keypoint coordination)
- $\mathbf{K}$ is the camera intrinsic matrix
    - $\mathbf{K}_{2\times3} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \end{bmatrix}$
    - $\mathbf{K}_{2\times4} = \begin{bmatrix} \mathbf{K}_{2\times3} & \mathbf{0} \end{bmatrix}$
    - f_x and f_y: focal length on x-axis and y-axis
    - c_x and c_y: (x, y) coordination for camera center
- $\mathbf{T}_{cw} = \begin{bmatrix} \mathbf{R}_{cw} & \mathbf{t}_{cw} \\ \mathbf{0}^T & 1 \end{bmatrix} = \exp(\boldsymbol{\xi}^\wedge)$ is the camera pose (world (3D) to camera (3D) transformation matrix), where $\mathbf{R}_{cw}$ is the rotation matrix and $\mathbf{t}_{cw}$ is the translation vector
- $\boldsymbol{\xi}^\wedge_{6\times1} = \begin{bmatrix} \boldsymbol{\phi}_{3\times1} \\ \boldsymbol{\rho}_{3\times1} \end{bmatrix}^\wedge = \left((\phi_1, \phi_2, \phi_3, \rho_1, \rho_2, \rho_3)^T\right)^\wedge = \begin{bmatrix} \boldsymbol{\phi}^\wedge & \boldsymbol{\rho} \\ \mathbf{0}^T & 1 \end{bmatrix}_{4\times4} \in \mathfrak{se}(3)$

where $\boldsymbol{\phi}^\wedge = \begin{bmatrix} 0 & -\phi_3 & \phi_2 \\ \phi_3 & 0 & -\phi_1 \\ -\phi_2 & \phi_1 & 0 \end{bmatrix}$

- $\mathbf{X}_w = (\bar{\mathbf{X}}_w, 1)^T = (X_w, Y_w, Z_w, 1)^T$ is the 3D robot position in the world in homogeneous coordinate
- $\mathbf{X}_c = (\bar{\mathbf{X}}_c, 1)^T = (X_c, Y_c, Z_c, 1)^T$ is the 3D robot position relative to camera origin

· Jacobian of the above error function w.r.t. $\boldsymbol{\xi}$ using purturbation method & chain rule (implemented in function `g2o::EdgeSE3ProjectXYZOnlyPose::linearizeOplus()`):

$$\begin{aligned}
\mathbf{J}_{2\times 6} = \frac{\partial \mathbf{e}_{2\times 1}}{\partial \boldsymbol{\xi}_{6\times 1}} &= \left(\frac{\partial \mathbf{e}}{\partial (\mathbf{T}_{cw}\mathbf{X}_w)_{4\times 1}}\right)_{2\times 4} \cdot \left(\frac{\partial (\mathbf{T}_{cw}\mathbf{X}_w)}{\partial \boldsymbol{\xi}}\right)_{4\times 6} \\
&\approx \frac{\partial \mathbf{e}}{\partial (\mathbf{T}_{cw}\mathbf{X}_w)} \cdot \frac{\partial \left(\exp(\delta\boldsymbol{\xi}^\wedge)\mathbf{T}_{cw}\mathbf{X}_w\right)}{\partial \delta\boldsymbol{\xi}} \\
&= \frac{\partial \mathbf{e}}{\partial \mathbf{X}_c} \cdot \frac{\partial \left(\exp(\delta\boldsymbol{\xi}^\wedge)\mathbf{X}_c\right)}{\partial \delta\boldsymbol{\xi}}
\end{aligned}$$

where

$$\begin{aligned}
\frac{\partial \mathbf{e}}{\partial \mathbf{X}_c} &= \frac{\partial}{\partial \mathbf{X}_c}\left(\mathbf{x}_o - \frac{1}{Z_c}\mathbf{K}_{2\times 4}\mathbf{X}_c\right) \\
&= \frac{\partial}{\partial \begin{bmatrix} X_c & Y_c & Z_c & 1 \end{bmatrix}^T}\left(\begin{bmatrix} x_o \\ y_o \end{bmatrix} - \frac{1}{Z_c}\begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \end{bmatrix}\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}\right) \\
&= \frac{\partial}{\partial \begin{bmatrix} X_c & Y_c & Z_c & 1 \end{bmatrix}^T}\left(\begin{bmatrix} x_o - f_x\frac{X_c}{Z_c} - c_x \\ y_o - f_y\frac{Y_c}{Z_c} - c_y \end{bmatrix}\right) \\
&= \begin{bmatrix} -f_x\frac{1}{Z_c} & 0 & f_x\frac{X_c}{Z_c^2} & 0 \\ 0 & -f_y\frac{1}{Z_c} & f_y\frac{Y_c}{Z_c^2} & 0 \end{bmatrix},
\end{aligned} \tag{5-1}$$

$$
\begin{aligned}
\frac{\partial\left(\exp(\delta\boldsymbol{\xi}^{\wedge})\mathbf{X}_c\right)}{\partial\delta\boldsymbol{\xi}} &= \frac{\partial}{\partial\delta\boldsymbol{\xi}}\left(\begin{bmatrix}\delta\boldsymbol{\phi}^{\wedge} & \delta\boldsymbol{\rho}\\ \mathbf{0}^T & 1\end{bmatrix}\mathbf{X}_c\right)\\
&= \frac{\partial}{\partial\begin{bmatrix}\delta\phi_1 & \delta\phi_2 & \delta\phi_3 & \delta\rho_1 & \delta\rho_2 & \delta\rho_3\end{bmatrix}^T}\begin{bmatrix}0 & -\delta\phi_3 & \delta\phi_2 & \delta\rho_1\\ \delta\phi_3 & 0 & -\delta\phi_1 & \delta\rho_2\\ -\delta\phi_2 & \delta\phi_1 & 0 & \delta\rho_3\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}X_c\\ Y_c\\ Z_c\\ 1\end{bmatrix}\\
&= \frac{\partial}{\partial\begin{bmatrix}\delta\phi_1 & \delta\phi_2 & \delta\phi_3 & \delta\rho_1 & \delta\rho_2 & \delta\rho_3\end{bmatrix}^T}\begin{bmatrix}-Y_c\delta\phi_3 + Z_c\delta\phi_2 + \delta\rho_1\\ X_c\delta\phi_3 - Z_c\delta\phi_1 + \delta\rho_2\\ -X_c\delta\phi_2 + Y_c\delta\phi_1 + \delta\rho_3\\ 1\end{bmatrix}\\
&= \begin{bmatrix}0 & Z_c & -Y_c & 1 & 0 & 0\\ -Z_c & 0 & X_c & 0 & 1 & 0\\ Y_c & -X_c & 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}.
\end{aligned}
$$

Therefore,

$$
\mathbf{J}_{2\times 6} = \begin{bmatrix}f_x\frac{X_cY_c}{Z_c^2} & -f_x(1+\frac{X_c^2}{Z_c^2}) & f_x\frac{Y_c}{Z_c} & -f_x\frac{1}{Z_c} & 0 & f_x\frac{X_c}{Z_c^2}\\ f_y(1+\frac{Y_c^2}{Z_c^2}) & -f_y\frac{X_cY_c}{Z_c^2} & -f_y\frac{X_c}{Z_c} & 0 & -f_y\frac{1}{Z_c} & f_y\frac{Y_c}{Z_c^2}\end{bmatrix} \tag{5-2}
$$

3. Full/Global BA

Full/Global BA is implemented in function `Optimizer::BundleAdjustment()`, which is called by function `Optimizer::GlobalBundleAdjustment()`. It is performed in the tracking thread during initial map creation, and in the loop closing thread during the correction of a detected loop.

In this optimization scheme, all the 3D map points $\mathbf{X}_w$ triangulated and their associated camera poses $\mathbf{T}_{cw}$ are to be optimized with 2D keypoint observations in image coordinates. An exception during the optimization is that the pose of 1st keyframe is fixed, and only the other poses are to be optimized.

- Procedures:
  (1) Add the vertices and edges into g2o solver (Levenberg-Marquardt)
    - Vertex:
      - Camera pose $\mathbf{T}_{cw} \in SE(3)$ for each keyframe
        - Pose data is encapsulated in class `g2o::SE3Quat`
        - Vertex is defined in class `g2o::VertexSE3Expmap`
      - All 3D map points $\bar{\mathbf{X}}_w \in \mathbf{R}^3$ (inhomogeneous coordinate) in the current initial map
        - Map point data is encapsulated in class `Eigen::Vector3d`
        - Vertex is defined in class `g2o::VertexSBAPointXYZ`
      - The vertices are sorted: if there're n_1 poses and n_2 map points, the pose vertices have the index range of $[0, n_1-1]$, and the map point vertices have that of $[n_1, n_1+n_2-1]$
    - Edge: binary edge with 2 vertices
      - Vertex 0: map point $\bar{\mathbf{X}}_w$
      - Vertex 1: the associated camera pose from which the associated map point is

triangulated (and the 2D keypoint is observed)

- Information matrix is the same described above

(2) Perform full BA for some iterations

- 20 iterations during initial map creation
- 10 iterations during loop correction

(3) Update map point data in the map and camera pose data in each keyframe

3.1 Error Function and Jacobian Computation

- Error function (implemented in function `g2o::EdgeSE3ProjectXYZ::computeError()`):

$$\begin{aligned}\mathbf{e}_{ij}(\mathbf{x}) &= \mathbf{e}_{ij}(\mathbf{x}_i, \mathbf{x}_j, \mathbf{z}_{ij}) = \mathbf{e}(\bar{\mathbf{X}}_w, \boldsymbol{\xi}) \\ &= \mathbf{x}_o - \frac{1}{Z_c}\mathbf{K}_{2\times 4}\mathbf{T}_{cw}\mathbf{X}_w \\ &= \mathbf{x}_o - \frac{1}{Z_c}\mathbf{K}_{2\times 4}\mathbf{X}_c \\ &= \mathbf{x}_o - \frac{1}{Z_c}\mathbf{K}_{2\times 4}\exp(\boldsymbol{\xi}^\wedge)\begin{bmatrix}\bar{\mathbf{X}}_w \\ 1\end{bmatrix}\end{aligned}$$

where related info on the above parameters are described here

- The error term is robustified using Huber kernel

$$f_{ij} = \rho_H\left(\sqrt{\mathbf{e}_{ij}^T\boldsymbol{\Omega}_{ij}\mathbf{e}_{ij}}\right)$$

where

$$\rho_H(x) = \begin{cases} x^2 & \text{if } |x| < b \\ 2b|x| - b^2 & \text{otherwise}\end{cases}$$

- $b = \chi = \sqrt{5.991}$ for monocular SLAM in actual implementation
- If no robust kernel is used, $\rho(x) = x^2$

- In g2o, original $\mathbf{e}_{ij}$ is first computed, and then replaced by $w_{ij}\mathbf{e}_{ij}$ such that

$$(w_{ij}\mathbf{e}_{ij})^T\boldsymbol{\Omega}_{ij}(w_{ij}\mathbf{e}_{ij}) = \rho_H\left(\sqrt{\mathbf{e}_{ij}^T\boldsymbol{\Omega}_{ij}\mathbf{e}_{ij}}\right)$$

where

$$w_{ij} = \frac{\sqrt{\rho_H(\|\mathbf{e}_{ij}\|_{\boldsymbol{\Omega}_{ij}})}}{\|\mathbf{e}_{ij}\|_{\boldsymbol{\Omega}_{ij}}} \quad \text{with} \quad \|\mathbf{e}_{ij}\|_{\boldsymbol{\Omega}_{ij}} = \sqrt{\mathbf{e}_{ij}^T\boldsymbol{\Omega}_{ij}\mathbf{e}_{ij}}$$

- Jacobian $\partial\mathbf{e}_{2\times 1}/\partial\mathbf{x}_{nk\times 1}$ computation
  - Only need to compute 2 sub Jacobian matrices $\partial\mathbf{e}/\partial\bar{\mathbf{X}}_w$ and $\partial\mathbf{e}/\partial\boldsymbol{\xi}$ to form the full $2\times nk$ Jacobian
  - Jacobian $\partial\mathbf{e}/\partial\boldsymbol{\xi}$: the result has been shown in (5-2)

· Jacobian $\partial \mathbf{e}/\partial \bar{\mathbf{X}}_w$:

$$\begin{aligned}\mathbf{J}_{2\times 3} &= \frac{\partial \mathbf{e}_{2\times 1}}{\partial \bar{\mathbf{X}}_{w,3\times 1}} \\ &= \frac{\partial \mathbf{e}}{\partial \bar{\mathbf{X}}_c}\frac{\partial \bar{\mathbf{X}}_c}{\partial \bar{\mathbf{X}}_w} \\ &= \frac{\partial \mathbf{e}}{\partial \bar{\mathbf{X}}_c}\frac{\partial (\mathbf{R}_{cw}\bar{\mathbf{X}}_w + \mathbf{t}_{cw})}{\partial \bar{\mathbf{X}}_w} \\ &= \frac{\partial \mathbf{e}}{\partial \bar{\mathbf{X}}_c}\mathbf{R}_{cw}\end{aligned}$$

From (5-1),

$$\frac{\partial \mathbf{e}}{\partial \bar{\mathbf{X}}_c} = \begin{bmatrix} -f_x\frac{1}{Z_c} & 0 & f_x\frac{X_c}{Z_c^2} \\ 0 & -f_y\frac{1}{Z_c} & f_y\frac{Y_c}{Z_c^2} \end{bmatrix}$$

4. Local BA

Local BA is performed in the local mapping thread after a new keyframe is added to the map and all the map point data related to this new keyframe is updated. It is implemented in function `Optimizer::LocalBundleAdjustment()`.

In this optimization scheme, the new keyframe with all its connected keyframes are to be optimized, that is, all the corresponding camera poses of the keyframes, and all the map points observed by these keyframes are to be optimized. All other keyframes that can observe the map points above, but are not connected with the new keyframe, are also included in the optimization scheme, but with their camera poses fixed throughout the optimization.

The procedures are as follows:

(1) Add the current new keyframe and its connected keyframes into a list of local keyframes;
(2) Add all the map points that can be observed by the local keyframes into a list of local map points;
(3) Add all the keyframes that can observe the local map points but are not connected to the current new keyframe into a list of local fixed keyframes;
(4) Add vertices and edges into g2o solver (Levenberg-Marquardt):
    · Vertex:
        · Camera pose for each local keyframe, and fixed camera pose for each local fixed keyframe;
            · Pose data is encapsulated in class `g2o::SE3Quat`
            · Vertex is defined in class `g2o::VertexSE3Expmap`
        · All 3D map points in the local map point list
            · Map point data is encapsulated in class `Eigen::Vector3d`
            · Vertex is defined in class `g2o::VertexSBAPointXYZ`
    · Edge: binary edge with 2 vertices
        · Vertex 0: map point
        · Vertex 1: the associated camera pose from which the associated map point is triangulated (and the 2D keypoint is observed)

· Information matrix is the same described above

(5) Perform local BA optimization for some iterations;
   · 5 iterations in actual implementation
(6) After the optimization, exclude outlier edges from the g2o optimization graph whose χ^2 error is larger than the χ^2 threshold 5.991 (see here for the explanation of the threshold);
(7) Perform another round of local BA optimization for some iterations;
   · 10 iterations in actual implementation
(8) For all the edges including the previously outlier ones, find all edges with their χ^2 error larger than the χ^2 threshold, and remove all the map points that are associated to these outlier edges from the associated keyframes. The observation info of the map points on these keyframes are also removed;
(9) Update camera poses with optimized ones;
(10) Update map point coordinates with optimized ones, and then update the normal and depth info of each map point.

4.1 Error Function and Jacobian Computation

Error function and Jacobian computation are the same with those in global BA procedure described above.

References

Barfoot, Timothy D. 2017. *State Estimation for Robotics*. 1st ed. New York, NY, USA: Cambridge University Press.

Faugeras, Olivier D, and Francis Lustman. 1988. "Motion and Structure from Motion in a Piecewise Planar Environment." *International Journal of Pattern Recognition and Artificial Intelligence* 2 (03): 485–508.

Gálvez-López, Dorian, and J. D. Tardós. 2012. "Bags of Binary Words for Fast Place Recognition in Image Sequences." *IEEE Transactions on Robotics* 28 (5): 1188–97. https://doi.org/10.1109/TRO.2012.2197158.

Hartley, R. I., and A. Zisserman. 2004. *Multiple View Geometry in Computer Vision*. Second. Cambridge University Press, ISBN: 0521540518.

Horn, Berthold KP. 1987. "Closed-Form Solution of Absolute Orientation Using Unit Quaternions." *Josa a* 4 (4): 629–42.

Kümmerle, Rainer, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. 2011. "G 2 O: A General Framework for Graph Optimization." In *2011 Ieee International Conference on Robotics and Automation*, 3607–13. IEEE.

Lepetit, Vincent, Francesc Moreno-Noguer, and Pascal Fua. 2008. "EPnP: An Accurate O(n) Solution to the PnP Problem." *International Journal of Computer Vision* 81 (2): 155. https://doi.org/10.1007/s11263-008-0152-6.

Mur-Artal, R., J. M. M. Montiel, and J. D. Tardós. 2015. "ORB-SLAM: A Versatile and Accurate Monocular SLAM System." *IEEE Transactions on Robotics* 31 (5): 1147–63. https://doi.org/10.1109/TRO.2015.2463671.

Rublee, Ethan, Vincent Rabaud, Kurt Konolige, and Gary R Bradski. 2011. "ORB: An Efficient Alternative to SIFT or SURF." In *ICCV*, 11:2. 1. Citeseer.