SVO 算法及代码分析整理

Icey Chiu

2020.11.16

目录

1. Introduction

SVO 全称 Fast Semi-Direct Monocular Visual Odometry. 作者 Forster 等提出了一种半直接单目视觉里程计算法，于 2014 年 ICRA 会议上发表，随后在 github 开源：https://github.com/uzh-rpg/rpg_svo. 半直接方法消除了用于运动估计的昂贵的特征提取和鲁棒匹配技术的需求。直接对像素值进行操作，从而在高帧速率下生成亚像素精度。一种显式建模 outliers 测量的概率建图方法用于估计 3D 点，这将导致较少的 outliers 和更可靠的点。精确且高帧速率的运动估计可在小，重复和高频纹理的场景中提高鲁棒性。SVO 的整体框架如下图 1 所示，主要分为跟踪和构图两个线程，该算法使用两个并行线程，一个用于运动估计，另一个用于在环境探索时进行建图。 这种分割允许在一个线程中进行快速且持续的跟踪，同时第二个线程扩展地图，从而与实时这个硬约束解耦。

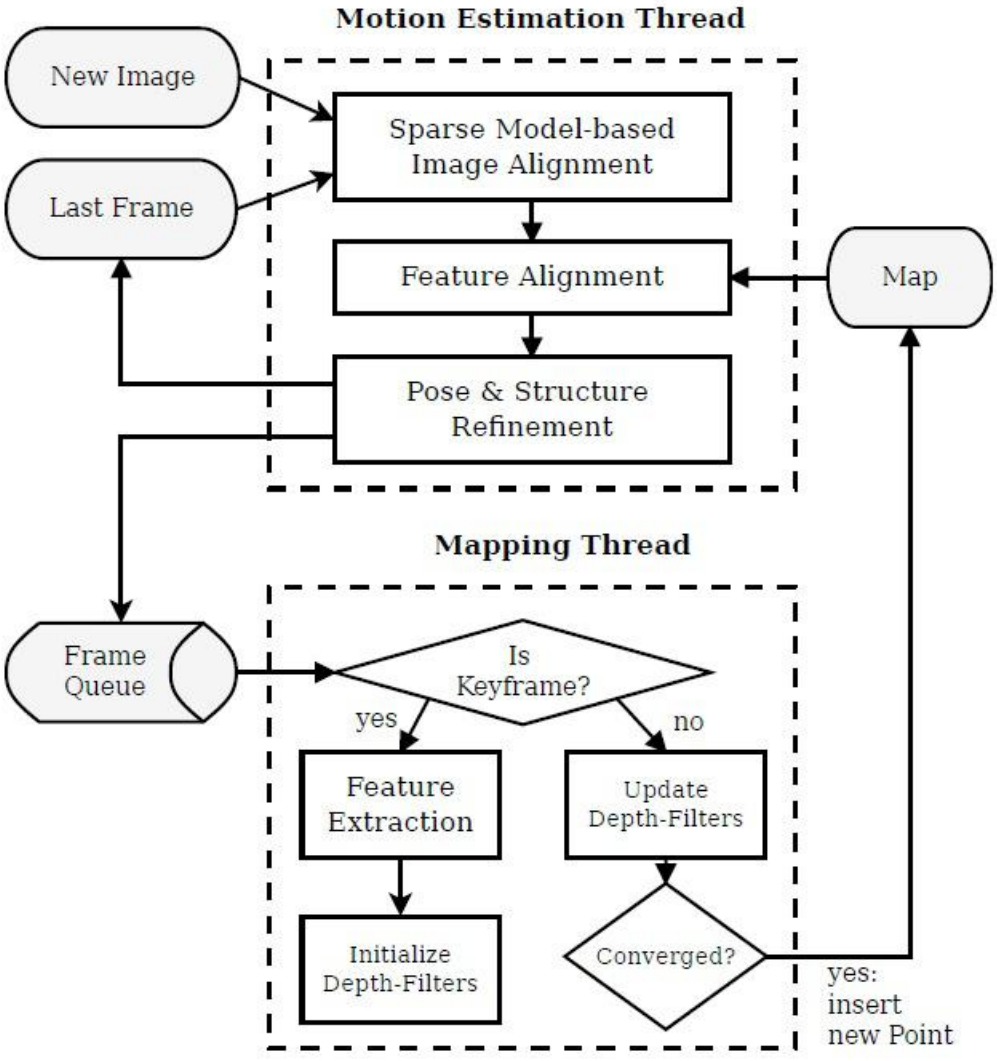


Fig. 1: Tracking and mapping pipeline

图 1 SVO 的跟踪和构图线程

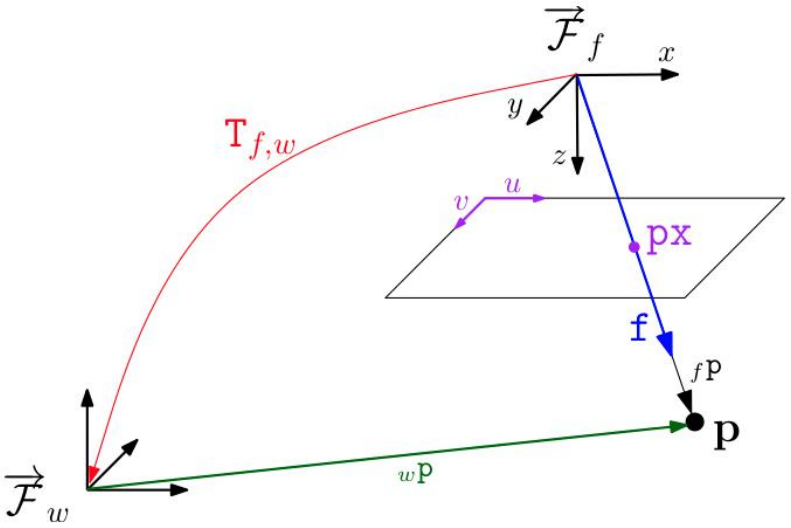
2. 初始化部分

主要涉及 initialization.cpp, feature_detection.cpp, frame_handler_mono.cpp.

Notation

Christian Forster edited this page on Jun 11, 2014 · 1 revision

The figure below illustrates the notation that is used in SVO.



Legend

`px` - Pixel coordinate (u,v)

`f` - Bearing vector of unit length (x,y,z)

`T_f_w` - Rigid body transformation from world frame `w` to camera frame `f`. This transformation transforms a point in world coordinates `p_w` to a point in frame coordinates `p_f` as follows: `p_f = T_f_w * p_w`. The camera position in world coordinates must be obtained by inversion: `pos = T_f_w.inverse().translation()`

图 2 一些基本符号的解释以及相机位姿的获取

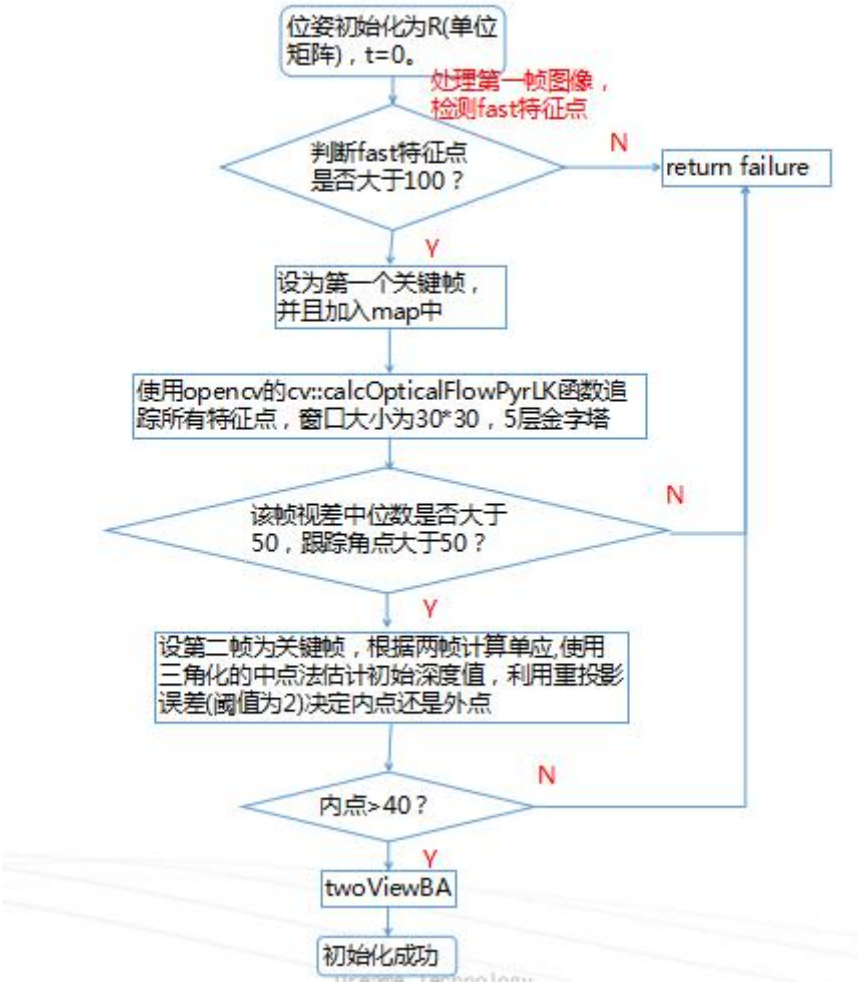


图 3 初始化流程图

代码分析：

处理第一帧用 FrameHandlerMono::processFirstFrame()函数，第一帧位姿 R 为单位阵，t 设为 0。然后第一帧加入初始化队列，这里调用了 class initialization 的 addFirstFrame 函数，对于第一帧，只需要检测特征点。

```
FrameHandlerMono::UpdateResult FrameHandlerMono::processFirstFrame()
{
  new_frame_->T_f_w_ = SE3(Matrix3d::Identity(), Vector3d::Zero());
  if(klt_homography_init_.addFirstFrame(new_frame_) == initialization::FAILURE)
    return RESULT_NO_KEYFRAME;
  new_frame_->setKeyframe();
  map_.addKeyframe(new_frame_);
  stage_ = STAGE_SECOND_FRAME;
  SVO_INFO_STREAM("Init: Selected first frame.");
  return RESULT_IS_KEYFRAME;
}
```

处理第一张图像，addFirstFrame()。先检测 FAST 特征点，如果图像的特征点数量超过 100 个，就把这张图像作为第一个关键帧，并加入 map_中。

```
InitResult KltHomographyInit::addFirstFrame(FramePtr frame_ref)
{
  reset();
  // [***step 1***]把第一帧作为参考帧，检测（跟踪）特征点px_ref_,得到它的归一化平面上的向量f_ref_
  detectFeatures(frame_ref, px_ref_, f_ref_);
  // 特征少则失败
  if(px_ref_.size() < 100)
  {
    SVO_WARN_STREAM_THROTTLE(2.0, "First image has less than 100 features. Retry in more textured environment.");
    return FAILURE;
  }
  // [***step 2***]特征足够，做为参考帧，并且插入其特征点到当前跟踪的特征点
  // px_ref_参考帧上要跟踪的点
  // px_cur_当前帧已经跟踪上的点
  frame_ref_ = frame_ref;
  px_cur_.insert(px_cur_.begin(), px_ref_.begin(), px_ref_.end());   // 在指定位置px_cur_.begin()前，插入从px_ref_.begin()到end()所有元素
  return SUCCESS;
}
```

addFirstFrame 中 detectFeatures()用于特征点检测，主要实现的 FastDetector::detect()在 feature_detection.cpp，选取的方法为：在最新的图像帧划分 30*30 的网格，在图像金字塔的每一层(3 层,scale 是 0.5)都检测 Fast 角点(fast10)，计算 fast 角点的得分，并在 3*3 的区域内进行非极大值抑制(可以解决检测到的角点相连的问题)，把不同金字塔图像上的特征点转换到第 0 层，并计算属于哪个网格，对每个特征点计算 shiTomasiScore，每个网格只选取一个 score 最大的点，如果特征点的 score 大于检测阈值(5),这个特征点作为 feature 加入 fts，否则删掉。

```
void detectFeatures(
    FramePtr frame,
    vector<cv::Point2f>& px_vec,
    vector<Vector3d>& f_vec)
{
  Features new_features;
  feature_detection::FastDetector detector(
      frame->img().cols, frame->img().rows, Config::gridSize(), Config::nPyrLevels());
  detector.detect(frame.get(), frame->img_pyr_, Config::triangMinCornerScore(), new_features);

  // - f_vec是特征点经过相机光心反投影cam2world()
  // - (X, Y, Z) = ((u - cx)/fx, (v - cy)/fy, 1.0)
  // now for all maximum corners, initialize a new seed
  px_vec.clear(); px_vec.reserve(new_features.size());
  f_vec.clear(); f_vec.reserve(new_features.size());
  std::for_each(new_features.begin(), new_features.end(), [&](Feature* ftr){
    px_vec.push_back(cv::Point2f(ftr->px[0], ftr->px[1]));
    f_vec.push_back(ftr->f);
    delete ftr;
  });
}
```

```
void FastDetector::detect(
    Frame* frame,
    const ImgPyr& img_pyr,
    const double detection_threshold,
    Features& fts)
{
    // 每个grid里一个特征点
  Corners corners(grid_n_cols_*grid_n_rows_, Corner(0,0,detection_threshold,0,0.0f));
 // 对每一层金字塔进行循环
  for(int L=0; L<n_pyr_levels_; ++L)
  {
    const int scale = (1<<L);
    vector<fast::fast_xy> fast_corners;
    // 检测fast角点
#if __SSE2__
      fast::fast_corner_detect_10_sse2(
          (fast::fast_byte*) img_pyr[L].data, img_pyr[L].cols,
          img_pyr[L].rows, img_pyr[L].cols, 20, fast_corners);
#elif HAVE_FAST_NEON
      fast::fast_corner_detect_9_neon(
          (fast::fast_byte*) img_pyr[L].data, img_pyr[L].cols,
          img_pyr[L].rows, img_pyr[L].cols, 20, fast_corners);
#else
      fast::fast_corner_detect_10(
          (fast::fast_byte*) img_pyr[L].data, img_pyr[L].cols,
          img_pyr[L].rows, img_pyr[L].cols, 20, fast_corners);
#endif
    vector<int> scores, nm_corners;
    // 计算fast角点的得分
    fast::fast_corner_score_10((fast::fast_byte*) img_pyr[L].data, img_pyr[L].cols, fast_corners, 20, scores);
    // 在3*3区域内取最大值
    fast::fast_nonmax_3x3(fast_corners, scores, nm_corners);

    for(auto it=nm_corners.begin(), ite=nm_corners.end(); it!=ite; ++it)
    {
      fast::fast_xy& xy = fast_corners.at(*it);  // 取最大值的点
      const int k = static_cast<int>((xy.y*scale)/cell_size_)*grid_n_cols_
                  + static_cast<int>((xy.x*scale)/cell_size_);  // 计算在第几个网格内
      // 深度滤波中通过特征对齐的方式得到的特征点，则会占据
      if(grid_occupancy_[k])  // 如果该网格已经有特征点则跳过
        continue;
      // 计算shiTomasiScore得分，取最大的放在grid里
      // 这个grid里的特征点可能来自不同的金字塔层里
      const float score = vk::shiTomasiScore(img_pyr[L], xy.x, xy.y);
      if(score > corners.at(k).score)
        corners.at(k) = Corner(xy.x*scale, xy.y*scale, score, L, 0.0f);
    }
  }

  // Create feature for every corner that has high enough corner score
  // 每个grid里面都只有一个特征点
```

```
  // Create feature for every corner that has high enough corner score
  // 每个grid里面都只有一个特征点
  std::for_each(corners.begin(), corners.end(), [&](Corner& c) {
    if(c.score > detection_threshold)
    // 大于阈值则作为feature加入
      fts.push_back(new Feature(frame, Vector2d(c.x, c.y), c.level));
  });

  resetGrid();
}

} // namespace feature_detection
} // namespace svo
```

然后处理第一张之后的连续图像，FrameHandlerMono::processSecondFrame()，用于跟第一张进行三角初始化 klt_homography_init_.addSecondFrame。从第一张图像开始，就用 trackKlt 持续跟踪所有的特征点，主要使用 opencv 的 cv::calcOpticalFlowPyrLK 函数，窗口大小为 30*30，5 层金字塔，计算追踪成功的特征点的 disparity 并把特征点转换成在相机坐标系下的深度归一化的点。取视差的中位数，如果小于最小阈值(50)，则不是 keyframe。(个人认为这个步骤只考虑了跟踪角点太少或者视差太小的情况，没有考虑剧烈运动视差大的情况，所以必须在持续跟踪下才可以。)若成功则根据两帧计算单应矩阵

computeHomography 函数，通过 H 矩阵分解得到 SE3,使用三角化的中点法估计初始特征点的深度值，利用重投影误差(阈值为 2)判断特征点是否有效，决定内点还是外点。如果内点少(最小阈值是 40)则不将该帧作为关键帧。若成功将获得的 3D 点的深度 Z 放入 depth_vec，取中位数，并将 scale 进行调整，使得深度的中位数为 1(一般 vslam 初始化都有这个步骤)。将内点尺度变换后的世界 3D 坐标对应到 cur 和 ref feature，特征点加入到对应帧，每个世界坐标系下的 3D 点都会被几个帧观察到,因此给 3D 点增加参考帧(注意这里的 3D 点 point 和 feature 不是一回事，feature 函数包括特征点的 3D 坐标，像素坐标，归一化坐标，对应的帧，像素点在金字塔图像的第几层被找到等信息)。

```
InitResult KltHomographyInit::addSecondFrame(FramePtr frame_cur)
{
  // [***step 1***]使用klt在参考帧与当前帧之间进行特征点跟踪
  trackKlt(frame_ref_, frame_cur, px_ref_, px_cur_, f_ref_, f_cur_, disparities_);
  SVO_INFO_STREAM("Init: KLT tracked "<< disparities_.size() <<" features");

  // 跟踪的角点太少
  if(disparities_.size() < Config::initMinTracked())
    return FAILURE;
  // 特征点（光流场）移动的距离太小，则不是关键帧
  double disparity = vk::getMedian(disparities_);  // 运动方向中位数，方向由当前帧特征点指向参考帧特征点
  SVO_INFO_STREAM("Init: KLT "<<disparity<<"px average disparity.");
  if(disparity < Config::initMinDisparity())    // 确保足够大的视差
    return NO_KEYFRAME;

  // [***step 2***]根据两帧计算单应矩阵，恢复T_cur_from_ref_,以及三角化的点xyz_in_cur_
  computeHomography(
      f_ref_, f_cur_,
      frame_ref_->cam_->errorMultiplier2(), Config::poseOptimThresh(),
      inliers_, xyz_in_cur_, T_cur_from_ref_);
  SVO_INFO_STREAM("Init: Homography RANSAC "<<inliers_.size()<<" inliers.");
  // 内点数量少则初始化失败
  if(inliers_.size() < Config::initMinInliers())
  {
    SVO_WARN_STREAM("Init WARNING: "<<Config::initMinInliers()<<" inliers minimum required.");
    return FAILURE;
  }

  // [***step 3***]获得深度，并将深度中值归一化为尺度，然后计算frame_cur平移
  // Rescale the map such that the mean scene depth is equal to the specified scale
  // 把深度提取出来
  vector<double> depth_vec;
  for(size_t i=0; i<xyz_in_cur_.size(); ++i)
    depth_vec.push_back((xyz_in_cur_[i]).z());

  double scene_depth_median = vk::getMedian(depth_vec);  // 深度的中值(以计算H矩阵的特征点在归一化平面上为前提的深度)
  double scale = Config::mapScale()/scene_depth_median;  // 将深度的中值进行归一化
  frame_cur->T_f_w_ = T_cur_from_ref_ * frame_ref_->T_f_w_;  // 求得当前帧的变换矩阵，这里的ref位姿如何得到？
  // 求得当前帧的平移变量
  frame_cur->T_f_w_.translation() =
      -frame_cur->T_f_w_.rotation_matrix()*(frame_ref_->pos() + scale*(frame_cur->pos() - frame_ref_->pos()));

  // [***step 4***]将内点尺度变换后的世界3D坐标加入到cur和ref feature，ftr加入到两帧上
  // For each inlier create 3D point and add feature in both frames
  SE3 T_world_cur = frame_cur->T_f_w_.inverse();  // 获得cur到world的变换
  for(vector<int>::iterator it=inliers_.begin(); it!=inliers_.end(); ++it)
  {
    // 提取出内点
```

```
    // 提取出内点
    Vector2d px_cur(px_cur_[*it].x, px_cur_[*it].y);
    Vector2d px_ref(px_ref_[*it].x, px_ref_[*it].y);
    // isInFrame判断是否在边缘10个像素之内，默认是0层金字塔，深度为正
    if(frame_ref_->cam_->isInFrame(px_cur.cast<int>(), 10) && frame_ref_->cam_->isInFrame(px_ref.cast<int>(), 10) && xyz_in_cur_[*it].z() > 0)
    {
      Vector3d pos = T_world_cur * (xyz_in_cur_[*it]*scale);  // 乘以尺度，转到world下坐标
      Point* new_point = new Point(pos);
      // frame_cur.get()得到指针
      // 创建特征点，特征点加入到对应帧(cur&ref)
      Feature* ftr_cur(new Feature(frame_cur.get(), new_point, px_cur, f_cur_[*it], 0));
      frame_cur->addFeature(ftr_cur);
      // 每个点都会被几个帧观测到，因此给点增加参考帧，点是世界坐标下的
      // 这里的特征点和点的意义是不同的
      new_point->addFrameRef(ftr_cur);

      Feature* ftr_ref(new Feature(frame_ref_.get(), new_point, px_ref, f_ref_[*it], 0));
      frame_ref_->addFeature(ftr_ref);
      new_point->addFrameRef(ftr_ref);
    }
  }
  return SUCCESS;
}
```

当得到前两关键帧后，用 ba::twoViewBA(初始化后的优化)函数做一个前两帧的局部 BA(阈值为 2)，优化关键帧的位置以及 3D 点的位置，从地图中删掉重投影误差(2)大的点。之后

把第二帧设为关键帧，用所有得到的 3D 点在世界坐标系下的 Z 的中位数作为场景深度的初始值(因为 SVO 是面向无人机下视，因此场景深度基本是单一平面，使用平均深度可以作为较好的初值。如果之后我们应用于地面的话，可以像 svo_edgelet 用周围点的深度来初始化)，当线程未被占用时，使用 depth_filter.cpp 中的 initializeSeeds 函数，传入 frame 这个参数初始化种子点(此步骤不使用多线程)，将已经有特征点的网格设置为占据，在没有特征点的网格用 triangMinCornerScore(阈值为 20)提取新的特征点，暂停更新种子点，上线程锁，增加种子点到 seeds 中，种子点都是新提取的点，初始化后继续更新种子点。将关键帧加入地图中，至此第二个关键帧选择完毕，三角初始化地图结束，stage_状态进入 STAGE_DEFAULT_FRAME。

```
FrameHandlerBase::UpdateResult FrameHandlerMono::processSecondFrame()
{
  initialization::InitResult res = klt_homography_init_.addSecondFrame(new_frame_);
  if(res == initialization::FAILURE)
    return RESULT_FAILURE;
  else if(res == initialization::NO_KEYFRAME)
    return RESULT_NO_KEYFRAME;

  // two-frame bundle adjustment
#ifdef USE_BUNDLE_ADJUSTMENT
  ba::twoViewBA(new_frame_.get(), map_.lastKeyframe().get(), Config::lobaThresh(), &map_);
#endif

  new_frame_->setKeyframe();
  double depth_mean, depth_min;
  frame_utils::getSceneDepth(*new_frame_, depth_mean, depth_min);
  depth_filter_->addKeyframe(new_frame_, depth_mean, 0.5*depth_min);

  // add frame to map
  map_.addKeyframe(new_frame_);
  stage_ = STAGE_DEFAULT_FRAME;
  klt_homography_init_.reset();
  SVO_INFO_STREAM("Init: Selected second frame, triangulated initial map.");
  return RESULT_IS_KEYFRAME;
}
```

3. Motion Estimation Thread

完成前两帧的初始化工作之后就进入了 FrameHandlerMono::processFrame()，对于新来的图像帧，首先基于 Tracking 的三个步骤跟踪相机的位姿：依次为 sparse model-based image alignment(基于 inverse compositional，代码在 sparse_img_align.cpp 中)、feature alignment(feature_alignment.cpp 中)、pose&structure refinement。

3.1 Sparse Image Align

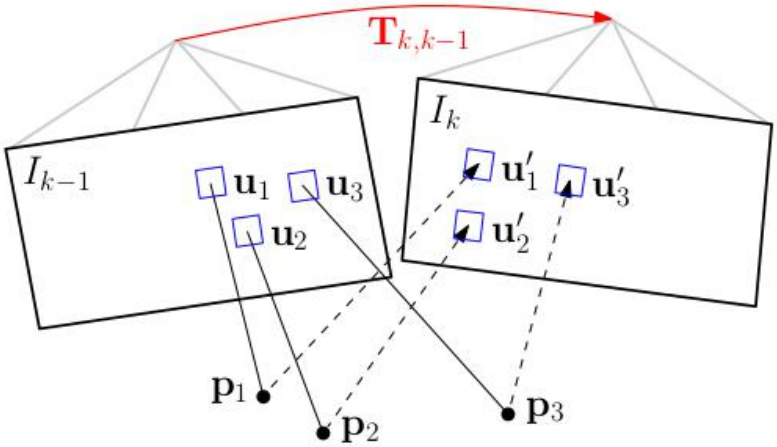


Fig. 2: Changing the relative pose $\mathbf{T}_{k,k-1}$ between the current and the previous frame implicitly moves the position of the reprojected points in the new image $\mathbf{u}'_i$. Sparse image alignment seeks to find $\mathbf{T}_{k,k-1}$ that minimizes the photometric difference between image patches corresponding to the same 3D point (blue squares). Note, in all figures, the parameters to optimize are drawn in red and the optimization cost is highlighted in blue.

该模块优化了相邻两帧之间的位姿变换(这一步忽略 patch 的变形，不做 warping，因为相邻帧之间的形变很小)，先用上一帧的位姿初始化本帧位姿，这是没有先验的情况。优化目标函数如下，使用了 inverse compositional 算法提高了计算效率，基于金字塔实现。

$$\mathbf{T}_{k,k-1} = \arg\min_{\mathbf{T}} \iint_{\bar{\mathcal{R}}} \rho\left[\delta I(\mathbf{T},\mathbf{u})\right] d\mathbf{u}. \qquad (4)$$

代码分析：

```
// Set initial pose TODO use prior
new_frame_->T_f_w_ = last_frame_->T_f_w_;

// sparse image align
SVO_START_TIMER("sparse_img_align");
SparseImgAlign img_align(Config::kltMaxLevel(), Config::kltMinLevel(),
                         30, SparseImgAlign::GaussNewton, false, false);
size_t img_align_n_tracked = img_align.run(last_frame_, new_frame_);
SVO_STOP_TIMER("sparse_img_align");
SVO_LOG(img_align_n_tracked);
SVO_DEBUG_STREAM("Img Align:\t Tracked = " << img_align_n_tracked);
```

run 函数完成了主要工作。初始化 cache：存储参考帧每个特征点的 patch，大小为 feature_size*patch_area(4*4)到 ref_patch_cache；存储每个像素所有 patch 的雅克比，大小为 6*ref_patch_cache_.size 到 jacobian_cache；存储可见的特征点，大小为 feature_size，默认都为 false。用当前帧从世界坐标系下到当前相机坐标系下的位姿转化(cur_frame_->T_f_w_)*参考帧从世界坐标系到参考帧相机坐标系下的位姿转化的逆(ref_frame_->T_f_w_.inverse)得到从参考帧相机坐标系到当前帧相机坐标系的位姿转化 SE3(T_cur_from_ref)，这是一个粗略的 T，之后在不同的金字塔层对 T_cur_from_ref 进行稀疏图像对齐优化，从最高层(第 4 层)到第 2 层，使用继承自 vk::NLLSSolver 的函数 optimize，使用直接法，计算流程来自于 nlls_sover_impl.hpp。

```cpp
size_t SparseImgAlign::run(FramePtr ref_frame, FramePtr cur_frame)
{
    // [***step 1***]复位NLLSSolver的变量值
  reset();

  if(ref_frame->fts_.empty())
  {
    SVO_WARN_STREAM("SparseImgAlign: no features to track!");
    return 0;
  }

  ref_frame_ = ref_frame;
  cur_frame_ = cur_frame;

  //[***step 2***] 初始化cache
  // 存储每个特征点的patch,大小(feature_size*patch_area(4*4))
  ref_patch_cache_ = cv::Mat(ref_frame_->fts_.size(), patch_area_, CV_32F);
  // 存储每个像素所有patch的雅克比，大小(6*ref_patch_cache_.size)
  jacobian_cache_.resize(Eigen::NoChange, ref_patch_cache_.rows*patch_area_);
  // 存储可见的特征点，类型vector<bool>大小feature_size,默认都为false
  visible_fts_.resize(ref_patch_cache_.rows, false); // TODO: should it be reset at each level?


  // [***step 3***]获得从参考帧到当前帧之间的变化
  // T_cur_from_world = T_cur_from_ref * T_ref_from_world
  // T_[to]_[from] T_A_C = T_A_B * T_B_C
  SE3 T_cur_from_ref(cur_frame_->T_f_w_ * ref_frame_->T_f_w_.inverse());

  // [***step 4***]在不同的金字塔层对T_c_r进行稀疏图像对齐优化，由粗到精，具有更好的初值
  // 在4level到2level之间
  for(level_=max_level_; level_>=min_level_; --level_)
  {
    mu_ = 0.1;
    jacobian_cache_.setZero();
    have_ref_patch_cache_ = false;
    if(verbose_)
      printf("\nPYRAMID LEVEL %i\n---------------\n", level_);
    optimize(T_cur_from_ref);
  }

  // [***step 5***]利用求得的T_c_r求得T_c_w
  cur_frame_->T_f_w_ = T_cur_from_ref * ref_frame_->T_f_w_;

  // n_meas_表示前一帧所有特征点块(feature patch)像素投影后在cur_frame中的像素个数
  // n_meas_/patch_area_表示特征点数
  return n_meas_/patch_area_;
}
```

SparseImgAlign 中实现每个函数具体的计算形式，使用了 inverse compositional 算法，具体理论理解可以参照论文“Lucas-Kanade 20 Years On: A Unifying Framework”。首先是预计算阶段，用 precomputeReferencePatches 函数。获取 level_层金字塔图像，将原图像特征像素点对应到该金字塔层，这里变化后坐标用 floorf 向下取整，判断此特征点 patch 是否在变换后的图像内且对应的 3D 点可见，满足条件则说此特征点 visiblity，用特征点的 3D 坐标减去 ref 的相机坐标得到深度 Z(最原始的求法)，将归一化平面上特征点的坐标 f 乘以深度得到相机坐标系下的 3D 坐标，计算 jacobian_xyz2uv，此处公式用的是十四讲第一版第八章 8.15 去掉内参加个负号，对参考帧金字塔图像每个 patch 中像素进行双边插值，计算图像对像素的导数，图像导数与对 SE3 的导数乘积求 jacobian。(注意预计算阶段的操作都是在参考帧金字塔图像上)

```
void SparseImgAlign::precomputeReferencePatches()
{
  const int border = patch_halfsize_+1;
  // [***step 1***]获取level_层图像金字塔的图像
  const cv::Mat& ref_img = ref_frame_->img_pyr_.at(level_);
  const int stride = ref_img.cols;  // 当前图像的列数
  const float scale = 1.0f/(1<<level_);  // 金字塔的尺度，每层2倍
  const Vector3d ref_pos = ref_frame_->pos();  // 当前帧位置
  const double focal_length = ref_frame_->cam_->errorMultiplier2();  // 返回fx_
  size_t feature_counter = 0;
  std::vector<bool>::iterator visiblity_it = visible_fts_.begin();
  for(auto it=ref_frame_->fts_.begin(), ite=ref_frame_->fts_.end();
      it!=ite; ++it, ++feature_counter, ++visiblity_it)
  {
    // check if reference with patch size is within image
    const float u_ref = (*it)->px[0]*scale;  // 原图像的像素坐标变换到对应金字塔层
    const float v_ref = (*it)->px[1]*scale;

    // [***step 2***]将特征点变换到金字塔图像上，并判断feature的patch是否在其内(不包括边缘)
    const int u_ref_i = floorf(u_ref);  // 变换后坐标下取整
    const int v_ref_i = floorf(v_ref);
    // 特征点patch是否在变换后的图像内(不包括边缘)，并且具有三维坐标则可见
    if((*it)->point == NULL || u_ref_i-border < 0 || v_ref_i-border < 0 || u_ref_i+border >= ref_img.cols || v_ref_i+border >= ref_img.rows)
      continue;
    *visiblity_it = true;

    // [***step 3***] 得到相机坐标系下的特征点，并求得对变换矩阵的雅克比矩阵
    // cannot just take the 3d points coordinate because of the reprojection errors in the reference image!!!
    const double depth(((*it)->point->pos_ - ref_pos).norm()); //点的三维坐标减去ref的相机坐标得到深度z
    const Vector3d xyz_ref((*it)->f*depth);  // 为单位平面上的值(x/z,y/z,1)*z得到camera坐标
    // evaluate projection jacobian
    Matrix<double,2,6> frame_jac;
    Frame::jacobian_xyz2uv(xyz_ref, frame_jac);  // 雅克比du/dp_c * dp_c/dzeta(se3)

    // [***step 4***] 利用该金字塔图像对每个patch中像素进行插值，计算得到图像对像素的导数
    // compute bilateral interpolation weights for reference image
    const float subpix_u_ref = u_ref-u_ref_i;
    const float subpix_v_ref = v_ref-v_ref_i;
    const float w_ref_tl = (1.0-subpix_u_ref) * (1.0-subpix_v_ref);
    const float w_ref_tr = subpix_u_ref * (1.0-subpix_v_ref);
    const float w_ref_bl = (1.0-subpix_u_ref) * subpix_v_ref;
    const float w_ref_br = subpix_u_ref * subpix_v_ref;
    size_t pixel_counter = 0;
    // 首指针+特征点的索引*patch大小，即遍历访问ref_patch_cache，cache内每个特征点patch的访问指针
    float* cache_ptr = reinterpret_cast<float*>(ref_patch_cache_.data) + patch_area_*feature_counter;
    for(int y=0; y<patch_size_; ++y)
    {
      // 计算出该层金字塔图像这个特征的patch的第一个像素指针
      uint8_t* ref_img_ptr = (uint8_t*) ref_img.data + (v_ref_i+y-patch_halfsize_)*stride + (u_ref_i-patch_halfsize_);
      for(int x=0; x<patch_size_; ++x, ++ref_img_ptr, ++cache_ptr, ++pixel_counter)
      {
```

```
    // 首指针+特征点的索引*patch大小，即遍历访问ref_patch_cache，cache内每个特征点patch的访问指针
    float* cache_ptr = reinterpret_cast<float*>(ref_patch_cache_.data) + patch_area_*feature_counter;
    for(int y=0; y<patch_size_; ++y)
    {
      // 计算出该层金字塔图像这个特征的patch的第一个像素指针
      uint8_t* ref_img_ptr = (uint8_t*) ref_img.data + (v_ref_i+y-patch_halfsize_)*stride + (u_ref_i-patch_halfsize_);
      for(int x=0; x<patch_size_; ++x, ++ref_img_ptr, ++cache_ptr, ++pixel_counter)
      {
        // precompute interpolated reference patch color
        *cache_ptr = w_ref_tl*ref_img_ptr[0] + w_ref_tr*ref_img_ptr[1] + w_ref_bl*ref_img_ptr[stride] + w_ref_br*ref_img_ptr[stride+1];

        // we use the inverse compositional: thereby we can take the gradient always at the same position
        // get gradient of warped image (~gradient at warped position)
        // 求导方法：利用四个方向像素的差除以2
        float dx = 0.5f * ((w_ref_tl*ref_img_ptr[1] + w_ref_tr*ref_img_ptr[2] + w_ref_bl*ref_img_ptr[stride+1] + w_ref_br*ref_img_ptr[stride+2])
                           -(w_ref_tl*ref_img_ptr[-1] + w_ref_tr*ref_img_ptr[0] + w_ref_bl*ref_img_ptr[stride-1] + w_ref_br*ref_img_ptr[stride]));
        float dy = 0.5f * ((w_ref_tl*ref_img_ptr[stride] + w_ref_tr*ref_img_ptr[1+stride] + w_ref_bl*ref_img_ptr[stride*2] + w_ref_br*ref_img_ptr[stride*2+1])
                           -(w_ref_tl*ref_img_ptr[-stride] + w_ref_tr*ref_img_ptr[1-stride] + w_ref_bl*ref_img_ptr[0] + w_ref_br*ref_img_ptr[1]));

        // [***step 5***]图像导数与对se(3)的导数乘积求Jacobian
        // 每个像素一行Jacobian (1*2) * (2*6) ，乘上对应的内参，为什么要缩小倍数？
        // cache the jacobian
        jacobian_cache_.col(feature_counter*patch_area_ + pixel_counter) =
            (dx*frame_jac.row(0) + dy*frame_jac.row(1))*(focal_length / (1<<level_));
      }
    }
  }
  have_ref_patch_cache_ = true;
}
```

之后是 computeResiduals，用计算出的双边插值系数对每个 cur_image 上的 feature 周围的 patch 进行插值，计算与 ref_image 上的 patch 的 res，用 res.norm()/scale 计算权重(因为在不同的金字塔层上)，用 res 平方乘权重计算出卡方 chi2，利用 jacobian_cache(是在预计算阶段在 ref frame 上提前求得不随更新变化的，只用计算一次，这就是 inverse compositional 算法高效的原因)求 J，H 是 J 的平方乘权重，求解最小二乘问题 H_*x_ = Jres_，不停迭代 30 次或卡方不再下降则更新停止(在参考帧上更新 T(ξ)，注意这里没有为了减少计算时间而采用图像块 warp 的更新方式，因为此方式对于帧对帧的小运动和小尺寸的图像块运动更有效)T_cur_from_ref 得到一个较精确的值后利用 T_cur_from_ref 反过去求得当前相机坐标系下的位姿转化 cur_frame_->T_f_w_，将前一帧所有特征点块投影到当前帧中的像素个数除特征点块面积得到此步骤追踪到的特征点数 img_align_n_tracked，输出 LOG。

```cpp
double SparseImgAlign::computeResiduals(
    const SE3& T_cur_from_ref,
    bool linearize_system,
    bool compute_weight_scale)
{
  // [***step 1***]得到当前金字塔层的cur_image图像
  // Warp the (cur)rent image such that it aligns with the (ref)erence image
  const cv::Mat& cur_img = cur_frame_->img_pyr_.at(level_);

  // 用来显示res图像
  if(linearize_system && display_)
    resimg_ = cv::Mat(cur_img.size(), CV_32F, cv::Scalar(0));
  // 是否预计算了需要的导数jacobian_cache_等信息，否则计算
  if(have_ref_patch_cache_ == false)
    precomputeReferencePatches();

  // compute the weights on the first iteration
  std::vector<float> errors;  // 存放res的
  // 如果计算compute_weight_scale则分配capacity
  if(compute_weight_scale)
    errors.reserve(visible_fts_.size());
  // 定义并初始化一些信息
  const int stride = cur_img.cols;  // 当前图像列数
  const int border = patch_halfsize_+1;
  const float scale = 1.0f/(1<<level_);
  const Vector3d ref_pos(ref_frame_->pos());  // 参考图像帧位置
  float chi2 = 0.0;  // X^2检验用
  size_t feature_counter = 0; // is used to compute the index of the cached jacobian
  std::vector<bool>::iterator visiblity_it = visible_fts_.begin();
  // 对每个ref图像上的特征点进行循环
  for(auto it=ref_frame_->fts_.begin(); it!=ref_frame_->fts_.end();
      ++it, ++feature_counter, ++visiblity_it)
  {
    // check if feature is within image
    // 不在图像中则忽略
    if(!*visiblity_it)
      continue;
    // [***step 2***]利用T_cur_from_ref将ref_image上的特征点对应的三维点投影到当前帧并转换到图像金字塔上
    // compute pixel location in cur img
    const double depth = ((*it)->point->pos_ - ref_pos).norm();
    const Vector3d xyz_ref((*it)->f*depth);
    const Vector3d xyz_cur(T_cur_from_ref * xyz_ref);
    const Vector2f uv_cur_pyr(cur_frame_->cam_->world2cam(xyz_cur).cast<float>() * scale);
    const float u_cur = uv_cur_pyr[0];
    const float v_cur = uv_cur_pyr[1];
    const int u_cur_i = floorf(u_cur);
    const int v_cur_i = floorf(v_cur);

    // 判断转换后在不在当前层的cur_image上
    // check if projection is within the image
    if(u_cur_i < 0 || v_cur_i < 0 || u_cur_i-border < 0 || v_cur_i-border < 0 || u_cur_i+border >= cur_img.cols || v_cur_i+border >= cur_img.rows)
```

```cpp
    // check if projection is within the image
    if(u_cur_i < 0 || v_cur_i < 0 || u_cur_i-border < 0 || v_cur_i-border < 0 || u_cur_i+border >= cur_img.cols || v_cur_i+border >= cur_img.rows)
      continue;

    // 插值系数
    // compute bilateral interpolation weights for the current image
    const float subpix_u_cur = u_cur-u_cur_i;
    const float subpix_v_cur = v_cur-v_cur_i;
    const float w_cur_tl = (1.0-subpix_u_cur) * (1.0-subpix_v_cur);
    const float w_cur_tr = subpix_u_cur * (1.0-subpix_v_cur);
    const float w_cur_bl = (1.0-subpix_u_cur) * subpix_v_cur;
    const float w_cur_br = subpix_u_cur * subpix_v_cur;
    // ref_image上patch的mat型
    float* ref_patch_cache_ptr = reinterpret_cast<float*>(ref_patch_cache_.data) + patch_area_*feature_counter;
    size_t pixel_counter = 0; // is used to compute the index of the cached jacobian

    // [***step 3***]对每个cur_image上的feature周围的patch进行插值，计算与ref_image上的patch的res
    for(int y=0; y<patch_size_; ++y)
    {
      uint8_t* cur_img_ptr = (uint8_t*) cur_img.data + (v_cur_i+y-patch_halfsize_)*stride + (u_cur_i-patch_halfsize_);

      for(int x=0; x<patch_size_; ++x, ++pixel_counter, ++cur_img_ptr, ++ref_patch_cache_ptr)
      {
        // compute residual
        const float intensity_cur = w_cur_tl*cur_img_ptr[0] + w_cur_tr*cur_img_ptr[1] + w_cur_bl*cur_img_ptr[stride] + w_cur_br*cur_img_ptr[stride+1];
        const float res = intensity_cur - (*ref_patch_cache_ptr);

        // used to compute scale for robust cost
        if(compute_weight_scale)
          errors.push_back(fabsf(res));

        // robustification
        float weight = 1.0;
        if(use_weights_) {
          weight = weight_function_->value(res/scale_);
        }

        chi2 += res*res*weight;  // x^2
        n_meas_++;  // 计算所有patch中的像素数量
        // [***step 4***]利用jacobiansp_cache求H,J
        if(linearize_system)
        {
          // compute Jacobian, weighted Hessian and weighted "steepest descend images" (times error)
          const Vector6d J(jacobian_cache_.col(feature_counter*patch_area_ + pixel_counter));  // 6*1
          H_.noalias() += J*J.transpose()*weight;  // 6*6
          Jres_.noalias() -= J*res*weight;
          if(display_)
            resimg_.at<float>((int) v_cur+y-patch_halfsize_, (int) u_cur+x-patch_halfsize_) = res/255.0;
        }
      }
    }
  }
```

使用直接法优化位姿存在以下问题：1. 此方法基于灰度不变假设，实际上相机会自动调整曝光参数(可以把像素值换为相对于整张图像平均像素值的值)，图像容易模糊，不同物体的材质有高光阴影的区别。2. 这个优化过程最终需要收敛，但图像是一个非凸的，只有当初始的相对位姿估计比较准确时，目标函数的非凸性才不会很明显。

3.2 Feature Alignment

通过上一步 sparse model-based image alignment 我们已经能够估计位姿了，但是这个位姿肯定不是完美的(因为 3D 点位置和相机位姿不准确)，导致重投影预测的特征点在 I_k 中的位置并不和真正的吻合，也就是还会有残差的存在。如下图所示，图中灰色的特征块为真实位置，蓝色特征块为预测位置。

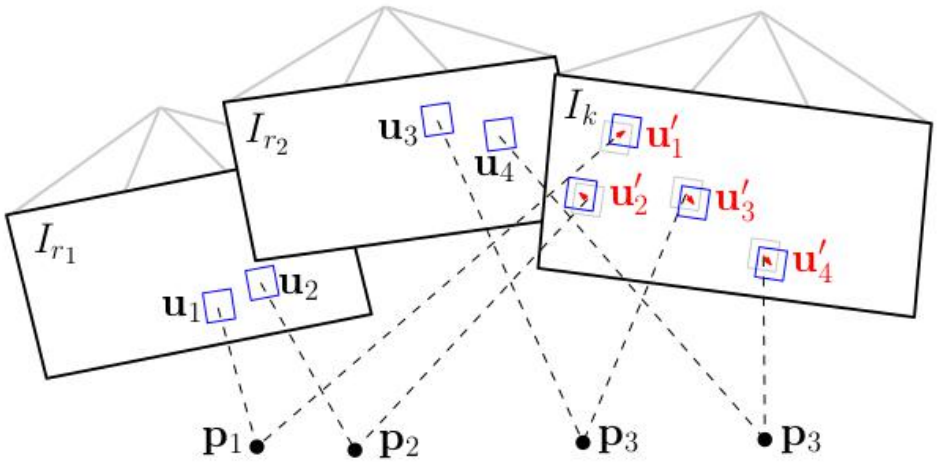


Fig. 3: Due to inaccuracies in the 3D point and camera pose estimation, the photometric error between corresponding patches (blue squares) in the current frame and previous keyframes r_i can further be minimised by optimising the 2D position of each patch individually.

用上一步得到的位姿作为可观察到的 3D 点在新一帧的初始假设，为了减少漂移，相机姿势应相对于地图对齐，而不是与前一帧对齐。地图上所有可见的 3D 点从估计的相机姿态投影到图像中，从而得出相应 2D 特征位置的估计值 u'i。对于每个重新投影的点，关键帧 r 确定以最近的观察角度观察该点的位置。然后，通过对齐关键帧 r 中的参考图像块和当前帧最小化光度误差，分别优化新图像中的所有 2D 特征位置 u i。

$$\mathbf{u}'_i = \arg\min_{\mathbf{u}'_i} \frac{1}{2} \| \mathbf{I}_k(\mathbf{u}'_i) - \mathbf{A}_i \cdot \mathbf{I}_r(\mathbf{u}_i) \|^2, \quad \forall\ i. \tag{13}$$

使用 inverse compositional Lucas-Kanade 算法解决这种对齐问题。 与上一步不同，因为使用了更大图像块（8×8），并且最接近的关键帧通常比 sparse image alignment 部分用来对比的上一帧更远，我们将仿射 warp A_i 应用于参考图像块。此部分可以理解为放松步骤，该步骤打破了极线约束以实现特征块之间的更高相关性。

代码分析：

整体的代码在 FrameHandlerMono::processFrame：

```
// map reprojection & feature alignment
SVO_START_TIMER("reproject");
reprojector_.reprojectMap(new_frame_, overlap_kfs_);
SVO_STOP_TIMER("reproject");
const size_t repr_n_new_references = reprojector_.n_matches_;
const size_t repr_n_mps = reprojector_.n_trials_;
SVO_LOG2(repr_n_mps, repr_n_new_references);
SVO_DEBUG_STREAM("Reprojection:\t nPoints = "<<repr_n_mps<<"\t \t nMatches = "<<repr_n_new_references);
if(repr_n_new_references < Config::qualityMinFts())
{
  SVO_WARN_STREAM_THROTTLE(1.0, "Not enough matched features.");
  new_frame_->T_f_w_ = last_frame_->T_f_w_; // reset to avoid crazy pose jumps
  tracking_quality_ = TRACKING_INSUFFICIENT;
  return RESULT_FAILURE;
}
```

具体实现的 reprojectMap 函数，在 reprojector.cpp 中。

```
void Reprojector::reprojectMap(
    FramePtr frame,
    std::vector< std::pair<FramePtr,std::size_t> >& overlap_kfs)
{
    // [***step 1***]重置
  resetGrid();

  // Identify those Keyframes which share a common field of view.
  SVO_START_TIMER("reproject_kfs");
  // [***step 2***]找到有重叠的关键帧，返回共享指针和距离
  list< pair<FramePtr,double> > close_kfs;
  map_.getCloseKeyframes(frame, close_kfs);
  // [***step 3***]根据距离排序(这在getclosekeyframe函数不是排过)
  // Sort KFs with overlap according to their closeness
  close_kfs.sort(boost::bind(&std::pair<FramePtr, double>::second, _1) <
                 boost::bind(&std::pair<FramePtr, double>::second, _2));

  // Reproject all mappoints of the closest N kfs with overlap. We only store
  // in which grid cell the points fall.
  size_t n = 0;
  // 预留空间，和resize有差别
  // resize会创建对象
  overlap_kfs.reserve(options_.max_n_kfs);
  // 投影的帧数是有上限的
  for(auto it_frame=close_kfs.begin(), ite_frame=close_kfs.end();
      it_frame!=ite_frame && n<options_.max_n_kfs; ++it_frame, ++n)
  {
  // [***step 4***]获得close_kefs里的每一帧，并创建overlap_kfs对象
    FramePtr ref_frame = it_frame->first;
    // reserve需要pushback,先压入的是近的
    overlap_kfs.push_back(pair<FramePtr,size_t>(ref_frame,0));

    // Try to reproject each mappoint that the other KF observes
    for(auto it_ftr=ref_frame->fts_.begin(), ite_ftr=ref_frame->fts_.end();
        it_ftr!=ite_ftr; ++it_ftr)
    {
      // check if the feature has a mappoint assigned
      if((*it_ftr)->point == NULL)
        continue;

      // 确保每个点向上一帧只投影一次
      // make sure we project a point only once
      if((*it_ftr)->point->last_projected_kf_id_ == frame->id_)
        continue;
      // [***step 5***]把投影帧id赋值给投影id,并进行投影到图像上，放入网格中
      (*it_ftr)->point->last_projected_kf_id_ = frame->id_;
      if(reprojectPoint(frame, (*it_ftr)->point))
        overlap_kfs.back().second++;  // 投影成功的个数，重叠程度
    }
  }
```

```
  SVO_STOP_TIMER("reproject_kfs");
// 地图候选点和之前的closekeyframe不重复吗？
// 不重复，候选点是未分配的收敛点，关键帧上是已经收敛的地图点
// 怎么选的候选点？
// 候选点是深度滤波得到的收敛点
//
  // Now project all point candidates
  SVO_START_TIMER("reproject_candidates");
  {
    boost::unique_lock<boost::mutex> lock(map_.point_candidates_.mut_);  // 多线程上锁
    auto it=map_.point_candidates_.candidates_.begin();
    while(it!=map_.point_candidates_.candidates_.end())
    {
        // [***step 6***]投影候选点
      if(!reprojectPoint(frame, it->first))
      {
          // 失败加3次，增加权重
        it->first->n_failed_reproj_ += 3;
        // 失败10次(顿)则删除这个点
        if(it->first->n_failed_reproj_ > 30)
        {
          map_.point_candidates_.deleteCandidate(*it);
          it = map_.point_candidates_.candidates_.erase(it);
          continue;
        }
      }
      ++it;
    }
  } // unlock the mutex when out of scope
  SVO_STOP_TIMER("reproject_candidates");

  // Now we go through each grid cell and select one point to match.
  // At the end, we should have at maximum one reprojected point per cell.
  SVO_START_TIMER("feature_align");
  for(size_t i=0; i<grid_.cells.size(); ++i)
  {
    // [***step 7***]随机选择网格进行对齐，网格中只要有一个特征点匹配成功即可，超过一点数量则匹配成功
    // 优先投影good点(优化后的)，unknow的点作候选
    // we prefer good quality points over unkown quality (more likely to match)
    // and unknown quality over candidates (position not optimized)
    // 随机产生的顺序有什么意义？
    if(reprojectCell(*grid_.cells.at(grid_.cell_order[i]), frame))
      ++n_matches_;  // 成功匹配的grid cell数目
    // 匹配成功超过一定数目就退出
    if(n_matches_ > (size_t) Config::maxFts())
      break;
  }
  SVO_STOP_TIMER("feature_align");
}
```

首先 getCloseKeyframes 函数找出有共视的关键帧和当前帧的距离。使用创建关键帧时生成的 5 个 keyPoints(这 5 个点的选择为所有特征点的最左上左下右上右下以及最靠近中间点)投影进行寻找，当前帧与 Map 中存储的关键帧有共视的关键帧并按照距离排序，取前 10 个共视最好(投影成功的个数)的关键帧。之后重投影候选点，如果候选点可以投影在当前帧边界 8 个像素之内，则计算属于第几个网格，并把候选点和像素点放进对应的网格中，此候选点就算投影成功，如果该候选点投影失败，n_failed_reproj_增加 3，如果该候选点失败 10 次，则从地图中删除。然后进行特征匹配，随机选择网格进行对齐，网格中只要有一个好的特征点匹配成功即可，超过 120 个则匹配成功。具体实现在 reprojectCell 函数，在网格里按照点的质量排序，优先使用优质点(点分为四种类型，排序为 delete < candidate < unknow < good)，如果是 delete 则从 cell 中删掉，然后用 matcher.cpp 中的 findMatchDirect 匹配，具体操作为：找到与 point 对应的离当前帧最近的关键帧上的特征 ref_ftr，验证该特征的 patch(+2)，是否超过该层图像的大小(因为特征点是在某一层金字塔上提取的)，根据 ref_ftr_周围的 8*8patch 求得 ref 到 cur 之间的 1D 仿射矩阵，找到 cur_frame 最适合的搜索的金字塔层，利用 A_cur_ref 将 ref 变换到 patch_with_border 上，得到的是 search_level 层上的 patch，去掉 patch_with_border 的边界后使用 inverse compositional 图像对齐，得到优化后的对应层数的 px(feature 是 edgelet 类型用 align1D，corner 用 align2D)，之

后在扩展像素点到当前帧的第 0 层。align1D 函数在 feature_alignment.cpp 中，这里的 jacobian 算的是沿着搜索方向(极线方向或者梯度方向)的导数，加入了亮度均值变化的考虑,收敛条件是更新平方小于 0.03*0.03 或者达到最大迭代次数 10。Align2D 函数和 align1D 操作相似收敛判断一样，只不过 align1D 是沿着极限方向变化图像块，align2D 是沿着 x，y 方向变化图像块。

```
/****************
 * @ function: 使用逆向组合法，进行一维特征对齐
 *
 * @ param:    const cv::Mat& cur_img    cur_image在searchlevel的金字塔图像
 *             const Vector2f& dir       patch搜索移动的方向：极线方向or梯度方向
 *             uint8_t* ref_patch_with_border  从ref变换到cur上的参考patch_border(不准确，带一个像素大小的边界！)
 *             uint8_t* ref_patch        从ref变换到cur上的参考patch(不准确的)
 *             const int n_iter          对齐算法最大迭代次数
 *             Vector2d& cur_px_estimate 当前的cur上特征点的位置估计值（待优化量）
 *             double& h_inv             沿着极线的（搜索方向）的hessian
 *
 * @ note:     优化的方程： I(x+u) = T(x) + i
 *             逆向组合： min Sum_x[T(x+delta_u) + delta_i - I(x+u) + i]^2
 *             优化的变量为: l ---- 像素位置增量在搜索方向上的步长 u=x0+v*l(x0 初始偏差； v 搜索方向； u 像素位置增量
 *                           i --- 亮度偏差)
 ****************/
#define SUBPIX_VERBOSE 0

bool align1D(
    const cv::Mat& cur_img,
    const Vector2f& dir,                 // direction in which the patch is allowed to move
    uint8_t* ref_patch_with_border,
    uint8_t* ref_patch,
    const int n_iter,
    Vector2d& cur_px_estimate,
    double& h_inv)
{
  const int halfpatch_size_ = 4;
  const int patch_size = 8;
  const int patch_area = 64;
  bool converged=false;

  // compute derivative of template and prepare inverse compositional
  float __attribute__((__aligned__(16))) ref_patch_dv[patch_area]; // 存ref_patch的导数，对应模板T(16位对齐)
  Matrix2f H; H.setZero(); // Hessian矩阵

  // compute gradient and hessian
  const int ref_step = patch_size+2;  // 图像求导需要带有border的patch
  float* it_dv = ref_patch_dv;  // 图像导数的指针
  // [***step 1***] 计算ref图像的导数和hessian矩阵
  for(int y=0; y<patch_size; ++y)  // 行循环
  {
    uint8_t* it = ref_patch_with_border + (y+1)*ref_step + 1;  // 除去border的patch指针
    for(int x=0; x<patch_size; ++x, ++it, ++it_dv)  // 列循环
    {
      Vector2f J;  // 导数
      // ! dT'/dW*dW/du*du/dl = delta_T * v = (dx dy) * v
      // * 这里的T(ref)是已经变换到I(cur)坐标系下了
      J[0] = 0.5*(dir[0]*(it[1] - it[-1]) + dir[1]*(it[ref_step] - it[-ref_step]));  // 沿着dir方向的导数
      // ! dT'/dl = l
```

```
      J[0] = 0.5*(dir[0]*(it[1] - it[-1]) + dir[1]*(it[ref_step] - it[-ref_step]));  // 沿着dir方向的导数
      // ! dT'/di = 1
      J[1] = 1;  // 亮度增量导数
      *it_dv = J[0];  // 每个像素的导数
      H += J*J.transpose();  // hessian矩阵，（这里用的列向量）
    }
  }
  // * 图像导数平方和取平均的倒数
  h_inv = 1.0/H(0,0)*patch_size*patch_size;   // ?
  Matrix2f Hinv = H.inverse();
  float mean_diff = 0;  // 亮度的平均偏差

  // Compute pixel location in new image:
  float u = cur_px_estimate.x();  // cur上特征位置
  float v = cur_px_estimate.y();

  // termination condition

  const float min_update_squared = 0.03*0.03;  // 收敛条件
  const int cur_step = cur_img.step.p[0];  // 就是step[0]
  float chi2 = 0;
  Vector2f update; update.setZero();  // 状态增量
  for(int iter = 0; iter<n_iter; ++iter)
  {
    int u_r = floor(u);
    int v_r = floor(v);
    // * 在边缘则跳过
    if(u_r < halfpatch_size_ || v_r < halfpatch_size_ || u_r >= cur_img.cols-halfpatch_size_ || v_r >= cur_img.rows-halfpatch_size_)
      break;
    // 不是数了？
    if(isnan(u) || isnan(v)) // TODO very rarely this can happen, maybe H is singular? should not be at corner.. check
      return false;

    // compute interpolation weights
    // * 双线性插值
    float subpix_x = u-u_r;
    float subpix_y = v-v_r;
    float wTL = (1.0-subpix_x)*(1.0-subpix_y);
    float wTR = subpix_x * (1.0-subpix_y);
    float wBL = (1.0-subpix_x)*subpix_y;
    float wBR = subpix_x * subpix_y;

    // loop through search_patch, interpolate
    uint8_t* it_ref = ref_patch;  // ref patch指针
    float* it_ref_dv = ref_patch_dv;  // ref patch导数指针
    float new_chi2 = 0.0;
    Vector2f Jres; Jres.setZero();
    for(int y=0; y<patch_size; ++y)  // cur上patch的行循环
    {
      // [***step 2***]计算残差和J*res
      uint8_t* it = (uint8_t*) cur_img.data + (v_r+y-halfpatch_size_)*cur_step + u_r-halfpatch_size_;  // cur的patch像素指针
```

```
      // [***step 2***]计算残差和J*res
      uint8_t* it = (uint8_t*) cur_img.data + (v_r+y-halfpatch_size_)*cur_step + u_r-halfpatch_size_;  // cur的patch像素指针
      for(int x=0; x<patch_size; ++x, ++it, ++it_ref, ++it_ref_dv)
      {
        float search_pixel = wTL*it[0] + wTR*it[1] + wBL*it[cur_step] + wBR*it[cur_step+1];  // cur上插值得到
        // ！计算残差res=T(x)-I(x+x0+v*l)-i
        float res = search_pixel - *it_ref + mean_diff;  // 像素的差
        // ！Sum_x[delta_T*v_l]*res
        Jres[0] -= res*(*it_ref_dv);  // -J*(I-T)
        Jres[1] -= res;
        new_chi2 += res*res;  // 卡方
      }
    }

    if(iter > 0 && new_chi2 > chi2)
    {
#if SUBPIX_VERBOSE
      cout << "error increased." << endl;
#endif
      // * 残差增加就减下去
      u -= update[0];
      v -= update[1];
      break;
    }

    chi2 = new_chi2;
    // [***step 3***] 计算状态增量
    // ！求得增量delta_p = H^-1 * Jres
    update = Hinv * Jres;
    // !更新p <== p - delta_p
    // [***step 4***]对位置在搜索方向进行更新，对亮度误差进行更新
    // *求Jres时是负的，所以这里还是加法
    u += update[0]*dir[0];
    v += update[0]*dir[1];
    mean_diff += update[1];

#if SUBPIX_VERBOSE
    cout << "Iter " << iter << ":"
         << "\t u=" << u << ", v=" << v
         << "\t update = " << update[0] << ", " << update[1]
         << "\t new chi2 = " << new_chi2 << endl;
#endif

    if(update[0]*update[0]+update[1]*update[1] < min_update_squared)
    {
#if SUBPIX_VERBOSE
      cout << "converged." << endl;
#endif
      converged=true;
      break;
```

findMatchDirect 结束后，如果没找到则重投影失败次数加一，如果 point 是 unknown 重投影失败次数大于 15 则删除该点，如果是 candidate 且失败次数大于 30 也删除；如果成功找到成功投影次数加一，point 是 unknown 且成功投影次数大于 10 则该点晋升为 good，在 frame 上找到匹配的点加入帧中。
重投影部分结束，LOG 输出重投影成功点数以及 match 数，如果 match 数小于 qualityMinFts(50)，则追踪不足，返回失败。

3.3 Pose and Structure Refinement

在此最后一步中，我们再次优化相机姿态 $T_{k,w}$ 以最小化重投影残差:

$$\mathbf{T}_{k,w} = \arg\min_{\mathbf{T}_{k,w}} \frac{1}{2} \sum_i \| \mathbf{u}_i - \pi(\mathbf{T}_{k,w} \ {}_w\mathbf{p}_i) \|^2 . \qquad (14)$$

这是 motion-only BA，可以使用迭代非线性最小二乘最小化算法（例如高斯牛顿）有效地解决。随后，我们通过最小化重投影误差（structure-only BA）来优化观察到的 3D 点的位置。 最后，可以应用局部 BA，在该局部 BA 中，所有接近关键帧的姿态以及所观察到的 3D 点均被一起优化。

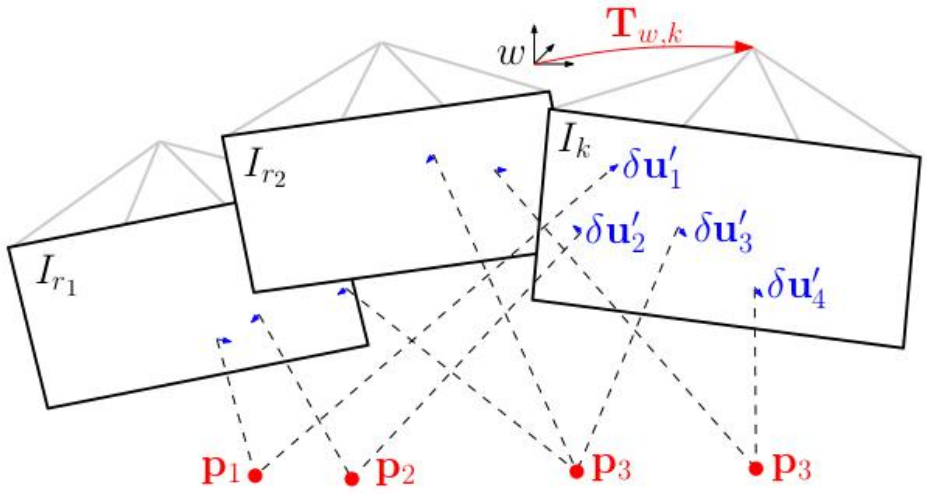


Fig. 4: In the last motion estimation step, the camera pose and the structure (3D points) are optimized to minimize the reprojection error that has been established during the previous feature-alignment step.

代码部分：
frame_handler_mono.cpp 里的流程，“pose optimization”部分主要用了 optimizeGaussNewton 函数，使用高斯牛顿法优化重投影误差，具体实现在 pose_optimizer.cpp。在单位平面上计算 frame 上每个特征点位置与 3D 点投影位置的误差并转化到相应金字塔层，把估计的标准差作为 scale 的值，进入 10 次迭代，第 6 次改变 scale 为 0.85/fx，用 jacobian_xyz2uv 得到 J，计算当前特征对应的归一化平面上的像素点与 3D

点投影到归一化平面上的残差，求解最小二乘问题得到 dT，如果卡方误差变大或者 dT 不是数，则 T_f_w 更新失败仍为旧值，否则更新图像位姿，当 dT<EPS(0.0000000001)收敛。计算图像位姿的协方差 cov=variance*$(A*fx^2)^{-1}$，计算优化后的误差，如果大于阈值(2/fx)，则把该点删除，从观测 num_obs 中减去删除的点。

```
// pose optimization
SVO_START_TIMER("pose_optimizer");
size_t sfba_n_edges_final;
double sfba_thresh, sfba_error_init, sfba_error_final;
pose_optimizer::optimizeGaussNewton(
    Config::poseOptimThresh(), Config::poseOptimNumIter(), false,
    new_frame_, sfba_thresh, sfba_error_init, sfba_error_final, sfba_n_edges_final);
SVO_STOP_TIMER("pose_optimizer");
SVO_LOG4(sfba_thresh, sfba_error_init, sfba_error_final, sfba_n_edges_final);
SVO_DEBUG_STREAM("PoseOptimizer:\t ErrInit = "<<sfba_error_init<<"px\t thresh = "<<sfba_thresh);
SVO_DEBUG_STREAM("PoseOptimizer:\t ErrFin. = "<<sfba_error_final<<"px\t nObsFin. = "<<sfba_n_edges_final);
if(sfba_n_edges_final < 20)
  return RESULT_FAILURE;
```

```
void optimizeGaussNewton(
    const double reproj_thresh,
    const size_t n_iter,
    const bool verbose,
    FramePtr& frame,
    double& estimated_scale,
    double& error_init,
    double& error_final,
    size_t& num_obs)
{
  // init
  double chi2(0.0);
  vector<double> chi2_vec_init, chi2_vec_final;
  vk::robust_cost::TukeyWeightFunction weight_function;  // Tukey权重函数(1-x^2/b^2)^2, 0
  SE3 T_old(frame->T_f_w_);  // 原位姿(SE3)!
  Matrix6d A;
  Vector6d b;

  // compute the scale of the error for robust estimation
  std::vector<float> errors; errors.reserve(frame->fts_.size());  // 特征点的位置误差
  // [***step 1***]计算frame上每个特征点位置与3D点投影位置的误差(单位平面上)
  for(auto it=frame->fts_.begin(); it!=frame->fts_.end(); ++it)
  {
    if((*it)->point == NULL)  //？为什么会出现这种情况
      continue;
    // *特征位置和3D点投影的位置误差(在单位平面上！)
    Vector2d e = vk::project2d((*it)->f)
               - vk::project2d(frame->T_f_w_ * (*it)->point->pos_);
    // 转换到相应的金字塔层，层数越高的占比越小，高层的误差噪声大，则缩小
    e *= 1.0 / (1<<(*it)->level);
    errors.push_back(e.norm());  // xy平方和，像素距离，误差大小
  }
  if(errors.empty())
    return;
  // [***step 2***]根据总errors计算中位数绝对误差，误差的尺度，确定误差整体是比较大还是比较小
  vk::robust_cost::MADScaleEstimator scale_estimator;  // 中位数绝对偏差估计
  estimated_scale = scale_estimator.compute(errors);   // 返回估计的标准差

  num_obs = errors.size();
  chi2_vec_init.reserve(num_obs);  // 初始卡方误差
  chi2_vec_final.reserve(num_obs);  // 最终卡方误差
  double scale = estimated_scale;
  //* 迭代优化位姿
  for(size_t iter=0; iter<n_iter; iter++)
  {
    // overwrite scale
    // 第五次迭代会重新改变一次scale
    if(iter == 5)
    // *之前求得J里面不包括fx,因此阈值也要除掉fx
    // *越往后由于迭代，误差的标准差会越小(这个是估计值吧)
    //尺度有什么作用？越往后误差可算性越高？
```

```
    // 尺度有什么作用？越往后误差可靠性越高？
      scale = 0.85/frame->cam_->errorMultiplier2();  // fabs(fx_)

    b.setZero();
    A.setZero();
    double new_chi2(0.0);

    // compute residual
    // [***step 3.1***]计算对T的雅可比矩阵J，先平移后旋转
    for(auto it=frame->fts_.begin(); it!=frame->fts_.end(); ++it)
    {
      if((*it)->point == NULL)
        continue;
      Matrix26d J;
      Vector3d xyz_f(frame->T_f_w_ * (*it)->point->pos_);
      Frame::jacobian_xyz2uv(xyz_f, J);
      //* step3.2 计算残差
      Vector2d e = vk::project2d((*it)->f) - vk::project2d(xyz_f);
      //* step3.3 计算信息矩阵？sigma^(1/2)
      double sqrt_inv_cov = 1.0 / (1<<(*it)->level);
      //* step3.4求和计算整个最小二乘A和b
      e *= sqrt_inv_cov;
      if(iter == 0)
          // e的平方和
        chi2_vec_init.push_back(e.squaredNorm()); // just for debug
      J *= sqrt_inv_cov;
      //* robust处理
      //！ x_square <= b_square ---> const float tmp = 1.0f - x_square / b_square; return tmp * tmp;
      //! x_square >= b_square ---> return 0;
      //* 权重可以使得大的误差系数是0。小的误差权重大一点，大的误差权重小一点，去除外点更加鲁棒
      double weight = weight_function.value(e.norm()/scale);  // 权重函数
      //！ A=J^T*sigma^(-1)*J
      A.noalias() += J.transpose()*J*weight;
      //! b=J^T*sigma^(-1)*e
      b.noalias() -= J.transpose()*e*weight;
      new_chi2 += e.squaredNorm()*weight;
    }
    //[***step 4***]求解最小二乘问题得到dT
    // solve linear system
    const Vector6d dT(A.ldlt().solve(b));

    // check if error increased
    if((iter > 0 && new_chi2 > chi2) || (bool) std::isnan((double)dT[0]))
    {
      if(verbose)
        std::cout << "it " << iter
                  << "\t FAILURE \t new_chi2 = " << new_chi2 << std::endl;
      frame->T_f_w_ = T_old; // roll-back
      break;
    }
//[***step 5***]更新图像的位姿
```

```
//[***step 5***]更新图像的位姿
    // update the model
    SE3 T_new = SE3::exp(dT)*frame->T_f_w_;
    T_old = frame->T_f_w_;
    frame->T_f_w_ = T_new;
    chi2 = new_chi2;
    if(verbose)
      std::cout << "it " << iter
                << "\t Success \t new_chi2 = " << new_chi2
                << "\t norm(dT) = " << vk::norm_max(dT) << std::endl;

    // stop when converged
    if(vk::norm_max(dT) <= EPS)
      break;
  }

  // Set covariance as inverse information matrix. Optimistic estimator!
  // [***step 6***]计算图像位姿的协方差，cov=(J^T*sigma^(-1)*J)^+,伪逆
  const double pixel_variance=1.0;
  //* 由于之前求的J里面没有fx,这里A要乘上fx^2
  frame->Cov_ = pixel_variance*(A*std::pow(frame->cam_->errorMultiplier2(),2)).inverse();

  // Remove Measurements with too large reprojection error
  //* 同理阈值要除掉fx
  double reproj_thresh_scaled = reproj_thresh / frame->cam_->errorMultiplier2();
  size_t n_deleted_refs = 0;
  //[***step 7***]计算优化后的误差，如果大于阈值，则把该点删除
  for(Features::iterator it=frame->fts_.begin(); it!=frame->fts_.end(); ++it)
  {
    if((*it)->point == NULL)
      continue;
    Vector2d e = vk::project2d((*it)->f) - vk::project2d(frame->T_f_w_ * (*it)->point->pos_);
    double sqrt_inv_cov = 1.0 / (1<<(*it)->level);
    e *= sqrt_inv_cov;
    chi2_vec_final.push_back(e.squaredNorm());
    if(e.norm() > reproj_thresh_scaled)
    {
      // we don't need to delete a reference in the point since it was not created yet
      (*it)->point = NULL;
      ++n_deleted_refs;
    }
  }
// 计算优化前后误差的中位数
  error_init=0.0;
  error_final=0.0;
  if(!chi2_vec_init.empty())
    error_init = sqrt(vk::getMedian(chi2_vec_init))*frame->cam_->errorMultiplier2();
  if(!chi2_vec_final.empty())
    error_final = sqrt(vk::getMedian(chi2_vec_final))*frame->cam_->errorMultiplier2();
// 误差的尺度把，反映误差大小
  estimated_scale *= frame->cam_->errorMultiplier2();
```

```
// 误差的尺度把，反映误差大小
  estimated_scale *= frame->cam_->errorMultiplier2();
  if(verbose)
    std::cout << "n deleted obs = " << n_deleted_refs
              << "\t scale = " << estimated_scale
              << "\t error init = " << error_init
              << "\t error end = " << error_final << std::endl;
  num_obs -= n_deleted_refs;  // 从观测中减去删除的点
}

} // namespace pose_optimizer
} // namespace svo
```

“structure optimization”部分主要用了optimizeStructure函数，在frame_handler_base.cpp中实现。

$$\{P_i\}=\text{argmin}\frac{1}{2}\sum_i \| u_i-\pi(T_{k,w}p_i)\|^2$$

```
// structure optimization
SVO_START_TIMER("point_optimizer");
optimizeStructure(new_frame_, Config::structureOptimMaxPts(), Config::structureOptimNumIter());
SVO_STOP_TIMER("point_optimizer");
```

```
/*************************
 * @ function: 优化frame中的关键点
 *
 * @ param:    frame    待优化的关键帧
 *             max_n_pts 最大优化的点数
 *             max_iter  最大优化迭代次数
 *
 * @ note: 对比较旧的点进行优化
 *************************/
void FrameHandlerBase::optimizeStructure(
    FramePtr frame,
    size_t max_n_pts,
    int max_iter)
{
  deque<Point*> pts;
  for(Features::iterator it=frame->fts_.begin(); it!=frame->fts_.end(); ++it)
  {
    if((*it)->point != NULL)
      pts.push_back((*it)->point);
  }
  max_n_pts = min(max_n_pts, pts.size());  // 获得优化的个数
  // pts.begin() + max_n_pts 进行排序，小的在左边，大的在右边
  nth_element(pts.begin(), pts.begin() + max_n_pts, pts.end(), ptLastOptimComparator);
  // 对比较老的point进行更新
  for(deque<Point*>::iterator it=pts.begin(); it!=pts.begin()+max_n_pts; ++it)
  {
    // 重投影优化地图点
    (*it)->optimize(max_iter);
    // 把当前帧的frame_id作为时间戳
    (*it)->last_structure_optim_ = frame->id_;
  }
}
```

“bundle adjustment”部分具体实现在 bundle_adjustment.cpp 中的 localBA 函数。

$$T_{k,w},\{p_i\} = \arg\min \frac{1}{2}\sum_i \| u_i - \pi(T_{k,w}p_i) \|^2$$

setCoreKfs 找出最近的三个关键帧，localBA 用 g2o 优化位姿及 3D 点。

```
  // optional bundle adjustment
#ifdef USE_BUNDLE_ADJUSTMENT
  if(Config::lobaNumIter() > 0)
  {
    SVO_START_TIMER("local_ba");
    setCoreKfs(Config::coreNKfs());
    size_t loba_n_erredges_init, loba_n_erredges_fin;
    double loba_err_init, loba_err_fin;
    ba::localBA(new_frame_.get(), &core_kfs_, &map_,
                loba_n_erredges_init, loba_n_erredges_fin,
                loba_err_init, loba_err_fin);
    SVO_STOP_TIMER("local_ba");
    SVO_LOG4(loba_n_erredges_init, loba_n_erredges_fin, loba_err_init, loba_err_fin);
    SVO_DEBUG_STREAM("Local BA:\t RemovedEdges {"<<loba_n_erredges_init<<", "<<loba_n_erredges_fin<<"} \t "
                     "Error {"<<loba_err_init<<", "<<loba_err_fin<<"}");
  }
#endif
```

```
void localBA(
    Frame* center_kf,
    set<FramePtr>* core_kfs,
    Map* map,
    size_t& n_incorrect_edges_1,
    size_t& n_incorrect_edges_2,
    double& init_error,
    double& final_error)
{

  // init g2o
  g2o::SparseOptimizer optimizer;
  setupG2o(&optimizer);

  list<EdgeContainerSE3> edges;
  set<Point*> mps;    // 要优化的地图点
  list<Frame*> neib_kfs;   // 和core_kfs具有共视关系的帧
  size_t v_id = 0;    // 顶点的id号
  size_t n_mps = 0;   // 加入的地图点的个数
  size_t n_fix_kfs = 0;   // 固定帧的个数
  size_t n_var_kfs = 1;   // 作为G2O变量顶点的关键帧数目
  size_t n_edges = 0;   // 边的个数
  n_incorrect_edges_1 = 0;
  n_incorrect_edges_2 = 0;

  // Add all core keyframes
  // [***step 1***]把core_kfs加入到优化器的顶点，把其观察到的地图点加入mps
  for(set<FramePtr>::iterator it_kf = core_kfs->begin(); it_kf != core_kfs->end(); ++it_kf)
  {
    g2oFrameSE3* v_kf = createG2oFrameSE3(it_kf->get(), v_id++, false);
    (*it_kf)->v_kf_ = v_kf;
    ++n_var_kfs;
    assert(optimizer.addVertex(v_kf));

    // all points that the core keyframes observe are also optimized:
    // 循环每一关键帧的特征点
    for(Features::iterator it_pt=(*it_kf)->fts_.begin(); it_pt!=(*it_kf)->fts_.end(); ++it_pt)
      if((*it_pt)->point != NULL)
        mps.insert((*it_pt)->point);
  }

  // Now go throug all the points and add a measurement. Add a fixed neighbour
  // Keyframe if it is not in the set of core kfs
  // 不懂为什么误差这么定义
  double reproj_thresh_2 = Config::lobaThresh() / center_kf->cam_->errorMultiplier2();  // 创建边时的误差
  double reproj_thresh_1 = Config::poseOptimThresh() / center_kf->cam_->errorMultiplier2();  // 位姿误差阈值
  double reproj_thresh_1_squared = reproj_thresh_1*reproj_thresh_1;
  // [***step 2***]把mps里面的点加入到优化器顶点
  for(set<Point*>::iterator it_pt = mps.begin(); it_pt!=mps.end(); ++it_pt)
  {
    // Create point vertex
```

注：笔者所有代码参数用的是 USE_ROS 版本

```
  // [***step 2***]把mps里面的点加入到优化器顶点
  for(set<Point*>::iterator it_pt = mps.begin(); it_pt!=mps.end(); ++it_pt)
  {
    // Create point vertex
    g2oPoint* v_pt = createG2oPoint((*it_pt)->pos_, v_id++, false);
    (*it_pt)->v_pt_ = v_pt;
    assert(optimizer.addVertex(v_pt));
    ++n_mps;

    // Add edges
    // [***step 3***]将3D点被观测的帧也加入到优化器，属于固定的帧
    list<Feature*>::iterator it_obs=(*it_pt)->obs_.begin();
    while(it_obs!=(*it_pt)->obs_.end())
    {
      Vector2d error = vk::project2d((*it_obs)->f) - vk::project2d((*it_obs)->frame->w2f((*it_pt)->pos_));
      //* 把未加入的加入
      if((*it_obs)->frame->v_kf_ == NULL)
      {
        // frame does not have a vertex yet -> it belongs to the neib kfs and
        // is fixed. create one:
        g2oFrameSE3* v_kf = createG2oFrameSE3((*it_obs)->frame, v_id++, true);
        (*it_obs)->frame->v_kf_ = v_kf;
        ++n_fix_kfs;
        assert(optimizer.addVertex(v_kf));
        neib_kfs.push_back((*it_obs)->frame);
      }
      // [***step 4***]将所有的这些点构成边，加入到优化中
      // create edge
      g2oEdgeSE3* e = createG2oEdgeSE3((*it_obs)->frame->v_kf_, v_pt,
                                       vk::project2d((*it_obs)->f),
                                       true,
                                       reproj_thresh_2*Config::lobaRobustHuberWidth(),
                                       1.0 / (1<<(*it_obs)->level));  // 金字塔层做权重
      assert(optimizer.addEdge(e));
      edges.push_back(EdgeContainerSE3(e, (*it_obs)->frame, *it_obs));
      ++n_edges;
      ++it_obs;
    }
  }
  // [***step 5***]先单独对点进行优化，再对点和位姿进行联合优化
  // structure only
  g2o::StructureOnlySolver<3> structure_only_ba;  // 点的维度是3
  g2o::OptimizableGraph::VertexContainer points;
  for (g2o::OptimizableGraph::VertexIDMap::const_iterator it = optimizer.vertices().begin(); it != optimizer.vertices().end(); ++it)
  {
      //* mVertexIDMap是map<ID, vertex>
    g2o::OptimizableGraph::Vertex* v = static_cast<g2o::OptimizableGraph::Vertex*>(it->second);
    //* 是3D地图点的图优化顶点并且所连边大于2，则放入点容器points
    if (v->dimension() == 3 && v->edges().size() >= 2)
        points.push_back(v);
  }
```

```
  g2o::OptimizableGraph::VertexContainer points;
  for (g2o::OptimizableGraph::VertexIDMap::const_iterator it = optimizer.vertices().begin(); it != optimizer.vertices().end(); ++it)
  {
      //* mVertexIDMap是map<ID, vertex>
    g2o::OptimizableGraph::Vertex* v = static_cast<g2o::OptimizableGraph::Vertex*>(it->second);
    //* 是3D地图点的图优化顶点并且所连边大于2，则放入点容器points
    if (v->dimension() == 3 && v->edges().size() >= 2)
        points.push_back(v);
  }
  //* 只优化这些地图点，固定帧位姿，用在联合优化之前或之后
  structure_only_ba.calc(points, 10);

  // Optimization
  if(Config::lobaNumIter() > 0)
    runSparseBAOptimizer(&optimizer, Config::lobaNumIter(), init_error, final_error);

  // [***step 6***]对优化的帧和点进行更新，共视帧不更新位姿
  // Update Keyframes
  for(set<FramePtr>::iterator it = core_kfs->begin(); it != core_kfs->end(); ++it)
  {
    (*it)->T_f_w_ = SE3( (*it)->v_kf_->estimate().rotation(),
                         (*it)->v_kf_->estimate().translation());
    (*it)->v_kf_ = NULL;
  }

  for(list<Frame*>::iterator it = neib_kfs.begin(); it != neib_kfs.end(); ++it)
    (*it)->v_kf_ = NULL;

  // Update Mappoints
  for(set<Point*>::iterator it = mps.begin(); it != mps.end(); ++it)
  {
    (*it)->pos_ = (*it)->v_pt_->estimate();
    (*it)->v_pt_ = NULL;
  }
  // [***step 7***]如果重投影误差过大，则将point和feature之间的约束切断
  // Remove Measurements with too large reprojection error
  double reproj_thresh_2_squared = reproj_thresh_2*reproj_thresh_2;
  for(list<EdgeContainerSE3>::iterator it = edges.begin(); it != edges.end(); ++it)
  {
    if(it->edge->chi2() > reproj_thresh_2_squared) //*(1<<it->feature_->level))
    {
      map->removePtFrameRef(it->frame, it->feature);
      ++n_incorrect_edges_2;
    }
  }

  // TODO: delete points and edges!
  // 乘以焦距f转化为像素误差
  init_error = sqrt(init_error)*center_kf->cam_->errorMultiplier2();
  final_error = sqrt(final_error)*center_kf->cam_->errorMultiplier2();
}
```

4. Depth Filter

特征点的深度估计被建模为一个 probability distribution，随后的每个观测值$\{I_k, T_{k,w}\}$用于更新贝叶斯框架中的分布，如下图所示，当 distribution 的方差足够小时，深度估计转化为一个 3D 点，这些点被插入 map 立刻被用于运动估计阶段。每个深度滤波器都和一个参考关键帧链接，滤波器初始化时深度非常不确定且均值被设为参考帧 r 的平均场景深度。随后的每个观测值$\{I_k, T_{k,w}\}$，我们在新来的帧I_k的极线上搜索一个与参考图像块最高联系度的图像块。极线可以根据帧$T_{r,k}$与穿过u_i的光线之间的相对姿势来计算。最高关联性的$u_i^{'}$的点对应的深度可以用三角化算出。深度的测量被建模成一个高斯均匀混合分布模型，outlier 在$[d_i^{\min}, d_i^{\max}]$中均匀分布。

$$p(\tilde{d}_i^k | d_i, \rho_i) = \rho_i \mathcal{N}(\tilde{d}_i^k | d_i, \tau_i^2) + (1-\rho_i)\mathcal{U}(\tilde{d}_i^k | d_i^{\min}, d_i^{\max})$$

ρ_i是 inlier 的概率，τ_i^2是可以通过假设图像平面中一个像素的光度视差方差来进行几何计算的良好测量的方差。我们在迭代贝叶斯更新步骤中用逆深度来处理大场景深度。当仅在对极线上当前深度估计附近搜索范围很小时，估计的深度估计非常有效。实践中，该范围对应于当前深度估算值标准偏差的两倍。图 6 展示了需要多小的运动才能显著减小深度不确定性。 与从两个角度对点进行三角化的标准方法相比，此方法的主要优势在于，由于每个滤波都要经过多次测量直到收敛，因此我们观察到的异常值要少得多。此外，对错误的测量进行了显式建模，即使在高度相似的环境中，也可以使深度收敛。

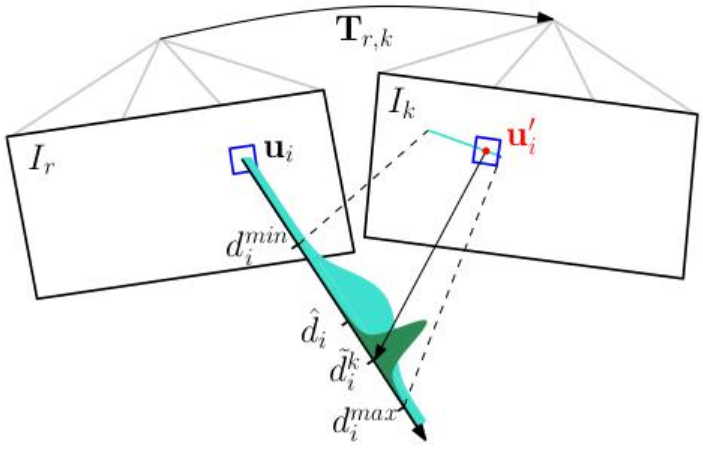


Fig. 5: Probabilistic depth estimate $\hat{d}_i$ for feature i in the reference frame r. The point at the true depth projects to similar image regions in both images (blue squares). Thus, the depth estimate is updated with the triangulated depth $\tilde{d}_i^k$ computed from the point $\mathbf{u}_i'$ of highest correlation with the reference patch. The point of highest correlation lies always on the epipolar line in the new image.

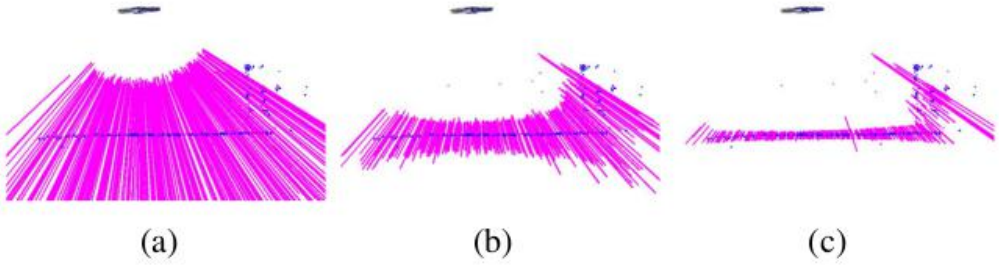


Fig. 6: Very little motion is required by the MAV (seen from the side at the top) for the uncertainty of the depth-filters (shown as mangenta lines) to converge.

此算法思路如下(参考论文“Video-based, Real-Time Multi View Stereo”)：

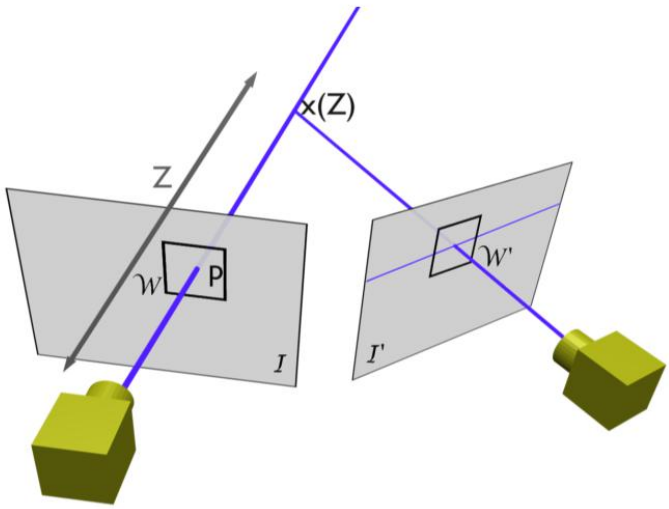


Figure 1: **Searching for a match along an optic ray.** For a given pixel p we wish to find the depth Z along the optic ray through p such that the 3d point $\mathbf{x}(Z)$ projects to similar image regions in images $\mathcal{I}$ and $\mathcal{I}'$. We can measure this similarity by computing a matching score between the two image patches $\mathcal{W}$ and $\mathcal{W}'$.

图 5 沿着光线方向搜索匹配点示意图

如图 5 所示，给定第一帧图像 I 以及图像上的一个像素点 p 和其周围区域 w，然后反投影回去，在深度为 Z 的一个点，将其投影到附近的相机图像 I’中，投影的像素周围区域为 w’，那么可以计算 w 和 w’的 NCC。

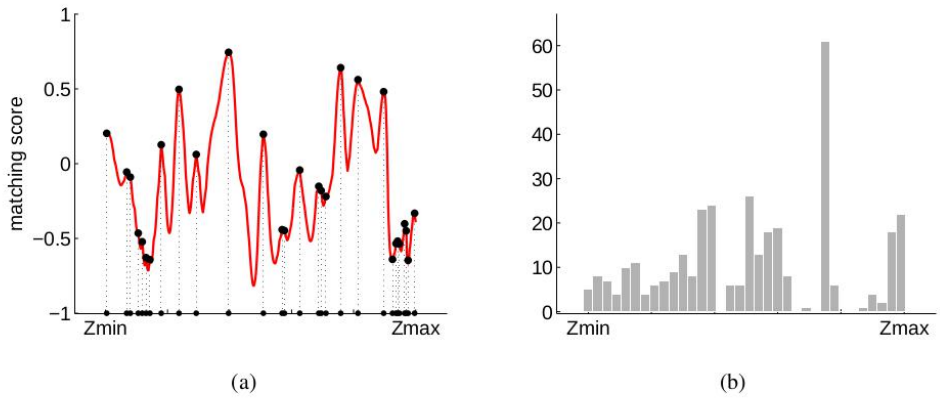


Figure 2: **Depth estimation with NCC maxima.** (a) NCC score across depth along optic ray. The black dots correspond to local maxima. (b) Histogram of local maxima for 60 neighboring images. Local maxima are either generated in the vicinity of the true depth or are uniformly generated across the depth range.

图 6 基于 NCC 极大值估计深度。(a)沿着光线方向搜索的 NCC score，黑色的点对应局部极大值。(b)周围 60 帧图像进行 NCC 匹配局部极大值的直方图，局部极大值在真实深度附近生成或者在深度范围内均匀生成。

观察图 6 右边可以发现这个直方图在在某一个值附近为高斯分布，在其他地方近似均匀分布(可能是由于遮挡、图像的变换以及重复性的纹理等)。将最大后验概率近似成高斯*Beta 分布。

$$q(z,\pi \mid a,b,\mu,\sigma) = Beta(\pi \mid a,b) * N(z \mid \mu,\sigma)$$

深度滤波器公式推导：

在得到了同一个种子点的多次测量 x1，x2，.....，xn 后，可以简单的使用最大似然估计得到模型的参数，但是极大似然的方法不能够得到深度估计的置信度以及判断估计是否已经收敛还是失败。因此，作者采用了贝叶斯的方法，给出了深度和内点数目的先验值，然后使用测量来计算后验的分布。深度滤波器融合所有深度测量的方法是最大后验概率估计，假设同一个种子点的所有测量为 x1，x2，.....，xn，需要估计种子点模型中的参数 Z，π，等价于求解下面的问题：

$$\arg\max_{z,\pi} p(z,\pi \mid x_1,x_2,...,x_n)$$

根据条件概率公式，有

$$p(z,\pi \mid x_1,x_2,...,x_n) = \frac{p(z,\pi,x_1,x_2,...,x_n)}{p(x_1,x_2,...,x_n)} = \frac{p(z,\pi)p(x_1,x_2,...,x_n \mid z,\pi)}{p(x_1,x_2,...,x_n)}$$

$$\sim p(z,\pi)p(x_1,x_2,...,x_n \mid z,\pi)$$

由于各个变量测量时是独立同分布的，因此有

$$p(z,\pi)p(x_1,x_2,...,x_n \mid z,\pi) = p(z,\pi)\prod_{i=1}^{n} p(x_i \mid z,\pi)$$

$$p(z,\pi)\prod_{i=1}^{n} p(x_i \mid z,\pi) = p(x_n \mid z,\pi)(p(z,\pi)\prod_{i=1}^{n} p(x_i \mid z,\pi))$$

作者证明上式左侧可以用高斯*Beta 分布来近似，这个分布和真实的后验分布有着最小的 KL 散度，即：

$$p(z,\pi)\prod_{i=1}^{n} p(x_i \mid z,\pi) \approx Beta(\pi \mid a_n,b_n) * N(z \mid \mu_n,\sigma_n) = q(z,\pi \mid a_n,b_n,\mu_n,\sigma_n)$$

通过上面的近似，需要估计的参数 Z，π，就可以通过迭代求解 a_n,b_n,μ_n,σ_n，然后求解最大值获得参数估计 $z=\mu_n,\pi=\frac{a_n-1}{a_n+b_n-2}$。公式中的 a_n,b_n 表示测量中内点和外点的数目，这个值在深度滤波器的迭代过程中会更新。

由迭代公式和近似公式，可以得到

$$q(z,\pi \mid a_n,b_n,\mu_n,\sigma_n) = p(x_n \mid z,\pi)q(z,\pi \mid a_{n-1},b_{n-1},\mu_{n-1},\sigma_{n-1})$$

由于上式右侧并不满足高斯*Beta 分布的形式，因此作者尝试用另一个高斯*Beta 分布来近似，使得其对于 Z，π 的一阶矩和二阶矩相同。

得到了近似分布的迭代公式，我们可以迭代的估计参数：

$$q(z,\pi \mid a,b,\mu,\sigma) = Beta(\pi \mid a,b)N(z \mid \mu,\sigma)$$

其中 $N(z \mid \mu,\sigma)$ 是以μ为均值，σ为方差的均匀分布，而 Beta(π|a,b)为

$$Beta(\pi \mid a,b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\pi^{a-1}(1-\pi)^{b-1}$$

将 $p(x \mid z,\pi) = \pi N(x \mid z,\tau^2) + (1-\pi)U(x)$ 带入，有：

$$p(x \mid z,\pi)q(z,\pi \mid a,b,\mu,\sigma) = (\pi N(x \mid z,\tau^2) + (1-\pi)U(x))q(z,\pi \mid a,b,\mu,\sigma)$$

根据如下两个等式：

$$B(\pi \mid a,b) = \frac{1}{\pi}\frac{a}{a+b}B(\pi \mid a+1,b) = \frac{1}{1-\pi}\frac{b}{a+b}B(\pi \mid a,b+1)$$

$$N(x \mid z,\tau^2)N(z \mid \mu,\sigma^2) = N(x \mid \mu,\sigma^2+\tau^2)N(z \mid m,s^2)$$

$$\frac{1}{s^2} = \frac{1}{\sigma^2}+\frac{1}{\tau^2} \quad m = s^2(\frac{\mu}{\sigma^2}+\frac{x}{\tau^2})$$

从而有

$$p(x \mid z,\pi)q(z,\pi \mid a,b,\mu,\sigma) = C_1 N(z \mid m,s^2)Beta(\pi \mid a+1,b) + C_2 N(z \mid \mu,\sigma^2)Beta(\pi \mid a,b+1)$$

其中 $C_1 = \frac{a}{a+b}N(x \mid \mu,\sigma^2+\tau^2), C_2 = \frac{b}{a+b}U(x)$

值得注意的是，上面的概率分布不是严格的概率分布，需要将 $C_1,C_2,C=C_1+C_2$ 进行归一化：

$$C = \int p(x \mid z,\pi)q(z,\pi \mid a,b,\mu,\sigma)dzd\pi = C_1 + C_2$$

有 $C_1^{'}=\frac{C_1}{C},C_2^{'}=\frac{C_2}{C}$

该分布关于 z 和π的一阶矩和二阶矩分别为 $C_1^{'}m+C_2^{'}\mu,C_1^{'}(m^2+s^2)+C_2^{'}(\mu^2+\sigma^2)$

$C_1^{'}\frac{a+1}{a+b+1}+C_2^{'}\frac{a}{a+b+1}$， $C_1^{'}\frac{(a+1)(a+2)}{(a+b+1)(a+b+2)}+C_2^{'}\frac{a(a+1)}{(a+b+1)(a+b+2)}$

下面根据一阶矩和二阶矩相等求解近似的高斯*Beta 分布 $q(z,\pi\mid a_n,b_n,\mu_n,\sigma_n)$：

Z 的一阶矩和二阶矩：

$$\int zq(z,\pi\mid a_n,b_n,\mu_n,\sigma_n)dzd\pi=\int zN(z\mid\mu^{'},\sigma^{'2})Beta(\pi\mid a^{'},b^{'})dzd\pi$$

$$=\int zN(z\mid\mu^{'},\sigma^{'2})dz=\mu^{'}$$

$$\int z^2q(z,\pi\mid a_n,b_n,\mu_n,\sigma_n)dzd\pi=\int z^2N(z\mid\mu^{'},\sigma^{'2})Beta(\pi\mid a^{'},b^{'})dzd\pi$$

$$=\int z^2N(z\mid\mu^{'},\sigma^{'2})dz=\mu^{'2}+\sigma^{'2}$$

π的一阶矩和二阶矩：

$$\int \pi q(z,\pi\mid a_n,b_n,\mu_n,\sigma_n)dzd\pi=\int \pi N(z\mid\mu^{'},\sigma^{'2})Beta(\pi\mid a^{'},b^{'})dzd\pi$$

$$=\int \pi Beta(\pi\mid a^{'},b^{'})dz=\frac{a^{'}}{a^{'}+b^{'}}$$

$$\int \pi^2q(z,\pi\mid a_n,b_n,\mu_n,\sigma_n)dzd\pi=\int \pi^2N(z\mid\mu^{'},\sigma^{'2})Beta(\pi\mid a^{'},b^{'})dzd\pi$$

$$=\int \pi^2Beta(\pi\mid a^{'},b^{'})dz=\frac{a^{'}(a^{'}+1)}{(a^{'}+b^{'})(a^{'}+b^{'}+1)}$$

根据一阶矩和二阶矩相等，有：

$$\mu^{'}=C_1^{'}m+C_2^{'}\mu$$

$$\sigma^{'2}=C_1^{'}(s^2+m^2)+C_2^{'}(\sigma^2+\mu^2)-\mu^{'2}$$

$$\frac{a^{'}}{a^{'}+b^{'}}=C_1^{'}\frac{a+1}{a+b+1}+C_2^{'}\frac{a}{a+b+1}$$

$$\frac{a^{'}(a^{'}+1)}{(a^{'}+b^{'})(a^{'}+b^{'}+1)}=C_1^{'}\frac{(a+1)(a+2)}{(a+b+1)(a+b+2)}+C_2^{'}\frac{a(a+1)}{(a+b+1)(a+b+2)}$$

令 $f=C_1^{'}\frac{a+1}{a+b+1}+C_2^{'}\frac{a}{a+b+1}$

$$e=C_1^{'}\frac{(a+1)(a+2)}{(a+b+1)(a+b+2)}+C_2^{'}\frac{a(a+1)}{(a+b+1)(a+b+2)}$$

则可以得到$a^{'},b^{'}$的迭代公式为：$a^{'}=\frac{\frac{e-f}{f-e}}{f},b^{'}=\frac{1-f}{f}a^{'}$

中间量	$C_1=\frac{a}{a+b}N(x\mid\mu,\sigma^2+\tau^2),C_2=\frac{b}{a+b}U(x)$，$C_1^{'}=\frac{C_1}{C},C_2^{'}=\frac{C_2}{C}$，$\frac{1}{s^2}=\frac{1}{\sigma^2}+\frac{1}{\tau^2}$，$m=s^2(\frac{\mu}{\sigma^2}+\frac{x}{\tau^2})$
均值迭代	$\mu^{'}=C_1^{'}m+C_2^{'}\mu$
方差迭代	$\sigma^{'2}=C_1^{'}(s^2+m^2)+C_2^{'}(\sigma^2+\mu^2)-\mu^{'2}$
Beta 函数的参数迭代	$a^{'}=\frac{\frac{e-f}{f-e}}{f},b^{'}=\frac{1-f}{f}a^{'}$

表 1 深度融合迭代公式

$N(x\mid\mu,\sigma^2+\tau^2)$	高斯分布采样
$U(x\mid 0,d_i^{\max})$	均匀分布采样
μ	上一次融合的深度估计值
x	此次的深度估计值
σ^2	上一次融合的深度不确定度
τ^2	此次深度测量的不确定度

表 2 深度融合公式中参数的含义

深度滤波主要流程图如下所示：

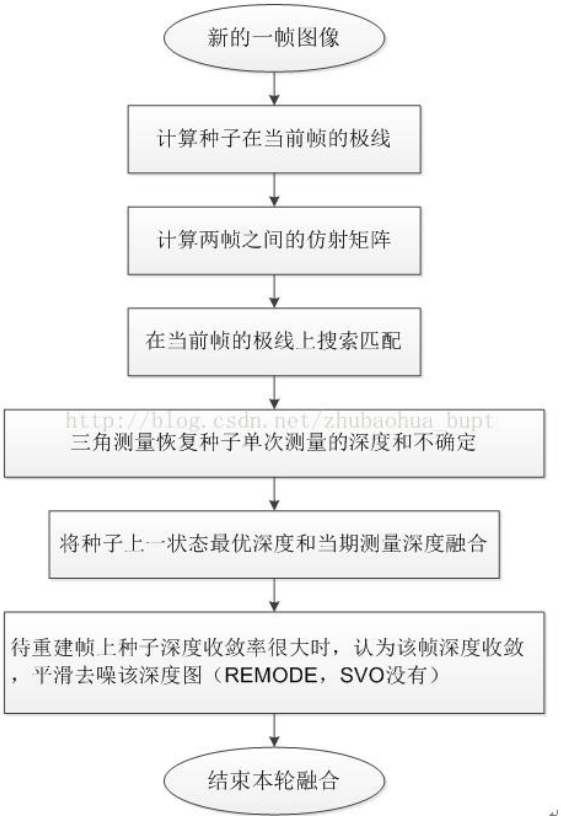


图 7 深度滤波主要流程图

深度滤波分步骤详细介绍：

4.1 种子点初始化

种子点参数	公式
Beta 分布参数	a=10，b=10
逆深度的均值	$d_{mean}=1/z_{mean}$
逆深度的最大值	$d_{\max}=1/z_{\min}$
99%在这个区间的协方差	$\sigma^2=\frac{d_{\max}*d_{\max}}{36}$

表 3 种子点初始化参数

4.2 计算极线

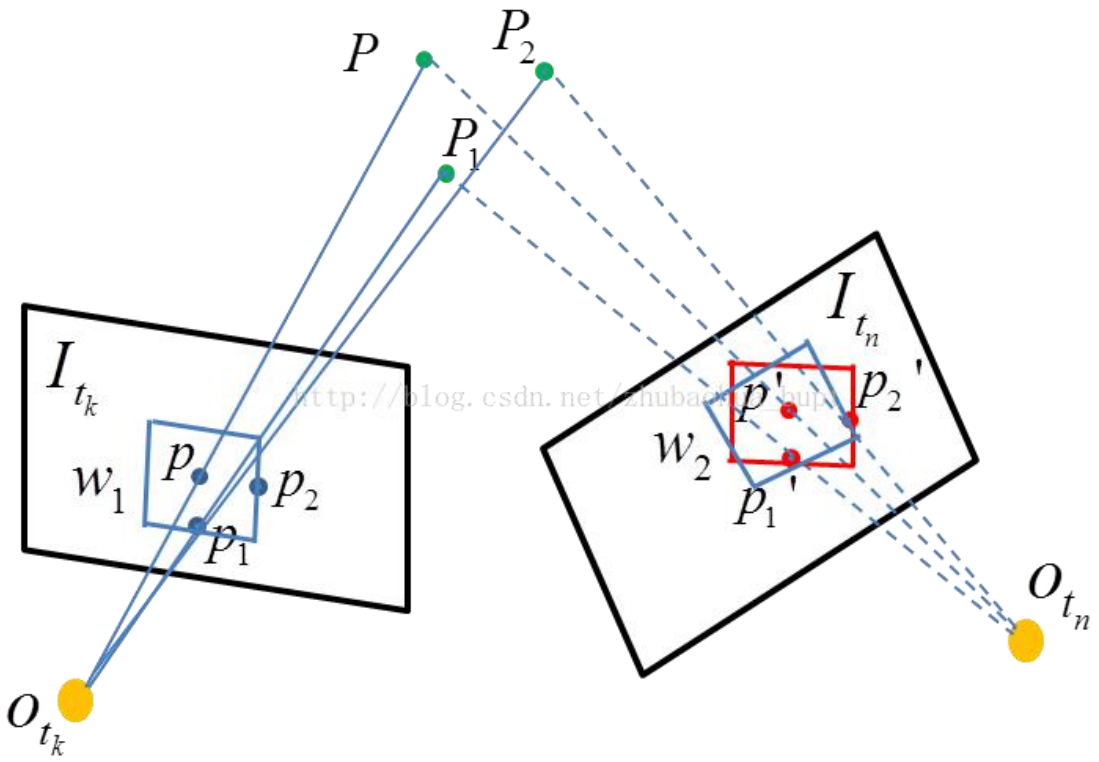
对于每个种子，在当前帧计算极线的条件是：
<1>已知种子所在帧与当前帧的相对位姿
<2>已知种子的初始深度
条件<1>的作用是用来做匹配，由 VO 提供。
条件<2>的作用是缩小找匹配的搜索量。当种子新提取时，这个时候还没有深度值，用场景平均深度初始。

本文用 z 表示种子的深度，sigma2 表示深度方差，sigma 表示深度标准差。
极线的计算方法如下：
step1：
在深度延长线上，构造两个三维点 P1，P2，这两个三维点来源同一个像素，唯一的不同就是深度，分别为 P1(x,y，z- n*sigma)，P2(x,y，z+ n*sigma)，这里 n 可以调节，一般选择 n=1,2,3（3sigma 原则）。
Step2：
将 P1，P2 利用 T_cur_ref，投影至当前帧，投影点为 u1，u2，连接 u1，u2 就是我们所要计算的极线。SVO 工程里，实现在 Matcher.cpp 里的 findEpipolarMatchDirect()函数里。

```
// [***step 1***]得到单位平面上极线的范围
// Compute start and end of epipolar line in old_kf for match search, on unit plane!
Vector2d A = vk::project2d(T_cur_ref * (ref_ftr.f*d_min));  // 单位平面上最小深度对应的极限端点
Vector2d B = vk::project2d(T_cur_ref * (ref_ftr.f*d_max));  // 单位平面上最大深度对应的极限端点
epi_dir_ = A - B;  // 单位平面上的极线段向量
```

4.3 计算仿射矩阵

如下图所示，描述 p 的窗口 w1 在后一帧上投影点为 p’，我们在匹配 p 和 p’时，应该用红色窗口而不是蓝色窗口，计算红色窗口就会用到仿射矩阵。



SVO 里仿射矩阵的计算思路是先利用三点法计算出 tk 和 tn 两时刻图像的仿射变换矩阵，然后再把窗口 w1 里的像素坐标逐一映射到图像 tn 里，这样映射的所有坐标就组成了窗口 w2。

```
/****************
 * @ function: 得到已知位姿的两个图像之间仿射变换
 *
 * @ param:    输入ref的相机参数，像素坐标，归一化坐标，深度，层数
 *             输入cur的相机参数，ref到cur的变换矩阵
 *             返回2*2的仿射矩阵
 *
 * @ note:     这里的金字塔层数有什么意义？
 * ******************/
void getWarpMatrixAffine(
    const vk::AbstractCamera& cam_ref,
    const vk::AbstractCamera& cam_cur,
    const Vector2d& px_ref,  // patch取得是中间点，所以加上half
    const Vector3d& f_ref,
    const double depth_ref,
    const SE3& T_cur_ref,
    const int level_ref,
    Matrix2d& A_cur_ref)
{
  // Compute affine warp matrix A_ref_cur
  const int halfpatch_size = 5;  // 考虑边界8+2的一半
  const Vector3d xyz_ref(f_ref*depth_ref);  // 点在ref下的3D坐标
  // ！为什么层数越大加的数越大
  // *px_ref虽然是在某一层金字塔提取的，但是也都会扩大到相应倍数到0层的坐标上
  // *因为特征是在level_ref上提取的，所以该层的patch对应到0层上要扩大相应倍数
  // 图像上根据金字塔层数对应的patch大小，得到patch的右上角坐标
  Vector3d xyz_du_ref(cam_ref.cam2world(px_ref + Vector2d(halfpatch_size,0)*(1<<level_ref)));
  // 图像上根据金字塔层数对应的patch大小，得到patch的左下角坐标
  Vector3d xyz_dv_ref(cam_ref.cam2world(px_ref + Vector2d(0,halfpatch_size)*(1<<level_ref)));
  // 根据该点的深度得到右上角，左下角的3D坐标（一种近似？）
  xyz_du_ref *= xyz_ref[2]/xyz_du_ref[2];
  xyz_dv_ref *= xyz_ref[2]/xyz_dv_ref[2];
  // 将这三点变换到cur下的图像坐标
  const Vector2d px_cur(cam_cur.world2cam(T_cur_ref*(xyz_ref)));
  const Vector2d px_du(cam_cur.world2cam(T_cur_ref*(xyz_du_ref)));
  const Vector2d px_dv(cam_cur.world2cam(T_cur_ref*(xyz_dv_ref)));
  //* 把原来的当作轴，变换得到对应的轴就是两列(相当于原来的是(1,0)和(0,1))
  //* 这个A_cur_ref是从ref金字塔到cur第0层的变换
  A_cur_ref.col(0) = (px_du - px_cur)/halfpatch_size;
  A_cur_ref.col(1) = (px_dv - px_cur)/halfpatch_size;
}
```

4.4 搜索匹配

SVO 用 ZMSSD(Zero Mean Sum of Squared Differences Cost)，通过 8*8 矩形 patch 来描述像素，用于计算种子和当前帧搜索点的相似性。当计算的极线小于两个像素时，取均值直接采用像素对齐，因为这个时候深度不确定较小，否则沿极线搜索匹配。代码在 findEpipolarMatchDirect()：

```
// [***step 5***] 若极线长度小于2，则进行特征对齐得到更精确的特征点位置，然后进行三角化计算深度
// 极线长度小于2则，即其附近8个点
if(epi_length_ < 2.0)
{
  px_cur_ = (px_A+px_B)/2.0;  // 取平均值
  Vector2d px_scaled(px_cur_/(1<<search_level_));  // cur点变到相应的层
  bool res;
  if(options_.align_1d)
    res = feature_alignment::align1D(
        cur_frame.img_pyr_[search_level_], (px_A-px_B).cast<float>().normalized(),
        patch_with_border_, patch_, options_.align_max_iter, px_scaled, h_inv_);
  else
    res = feature_alignment::align2D(
        cur_frame.img_pyr_[search_level_], patch_with_border_, patch_,
        options_.align_max_iter, px_scaled);
  if(res)
  {
    px_cur_ = px_scaled*(1<<search_level_);
    if(depthFromTriangulation(T_cur_ref, ref_ftr.f, cur_frame.cam_->cam2world(px_cur_), depth))
      return true;
  }
  return false;
}

size_t n_steps = epi_length_/0.7; //步数，为什么是0.7？ one step per pixel
Vector2d step = epi_dir_/n_steps; //步长

if(n_steps > options_.max_epi_search_steps)
{
    // %zu 输出 size_t
  printf("WARNING: skip epipolar search: %zu evaluations, px_lenght=%f, d_min=%f, d_max=%f.\n",
         n_steps, epi_length_, d_min, d_max);
  return false;
}

// for matching, precompute sum and sum2 of warped reference patch
int pixel_sum = 0;
int pixel_sum_square = 0;
PatchScore patch_score(patch_);  // ?使用cur上的先计算的patch_作为ref_patch?
```

4.5 三角测量恢复深度以及匹配不确定性的计算

经过搜索匹配后，能够得到种子 p 以及种子 p 在当前帧上的匹配像素点 p'，通过三角测量，就可以恢复种子 p 的深度。

```
/*****************
 * @ function: 三角化
 *
 * @ param: R, t, X1, X2
 *
 * @ note: 返回的是ref下的深度
 *         ! d2X2 = Rd1X1 + t ==> [RX1,  X2]*[-d1, d2]^T = t
 *****************/
bool depthFromTriangulation(
    const SE3& T_search_ref,
    const Vector3d& f_ref,
    const Vector3d& f_cur,
    double& depth)
{
  //* A是公式中的[RX1, X2]
  Matrix<double,3,2> A; A << T_search_ref.rotation_matrix() * f_ref, f_cur;
  //* 正规方程A^TA*d=A^T*t
  const Matrix2d AtA = A.transpose()*A;
  //* 判断是否可逆
  if(AtA.determinant() < 0.000001)
    return false;
  //* 求解正规方程
  const Vector2d depth2 = - AtA.inverse()*A.transpose()*T_search_ref.translation();
  //* 得到ref下的深度
  depth = fabs(depth2[0]);
  return true;
}
```

多次三角测量的深度是为了，融合得到种子较准确的深。那么既然有融合，不同测量值肯定有不同权重。不确定性就是用来计算权重的。在 SVO 中，深度的不确定被认为是，在匹配时，误匹配一个像素所带来的最大深度误差。

记三角化后，点的深度估计为 $_r p$，两帧之间的相对平移为 t，参考帧 I_r 中特征点观测所对

应的单位方向向量为$\overline{f}$，那么可以计算出下图的两个角度：

a=$_r p$ -t

α=arccos($\frac{\overline{f} * t}{\| t \|}$)

β=arccos($-\frac{a * t}{\| a \| * \| t \|}$)

记相机的焦距为 f，那么角度 β 加上一个像素的不确定度后的角度(相机中心正负 0.5 度，这是一种近似处理，由于焦距可能和像素点到成像平面中心的距离相当，从而有可能近似误差有点大)为：

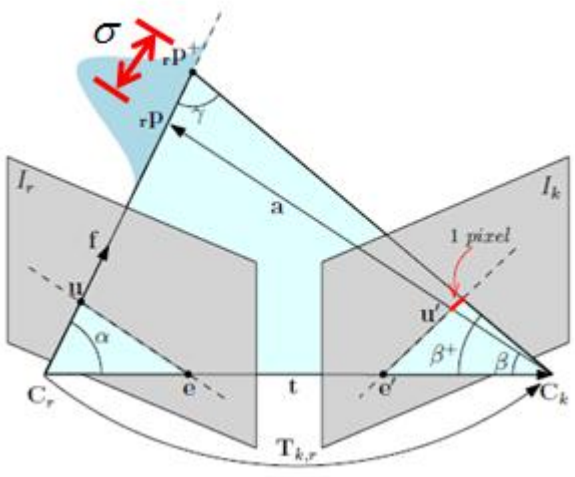
$$\beta^{+} = \beta + 2\tan^{-1}(\frac{1}{2f})$$

$$\gamma = \pi - \alpha - \beta^{+}$$

$$\|_r p^{+}\| = \|t\| \frac{\sin\beta^{+}}{\sin\gamma}$$

从而计算出深度的不确定度为：

$$\tau_k^2 = (\|_r p^{+}\| - \|_r p\|)^2$$



4.6 深度融合

SVO 的融合是不断利用最新时刻深度的观测值，来融合上一时刻深度最优值，直至深度收敛。具体过程如下：

1）符号含义：

Z 表示深度，这个点经过上一次融合后深度均值和方差分别为 Z_{est} 和 σ_{est}^2，Beta 分布参数为 a_{est}、b_{est}。当对一个像素的深度做了一次新的测量，新的深度值和方差的观测数据分别为 Z_{obs} 和 a,b，其中深度的方差 σ_{obs}^2 由深度的不确定性得到。Z_{update}、σ_{update}^2、a_{update}、b_{update} 分别为经过当前融合后的深度均值、不确定性、Beta 分布的两个参数 a,b。

2）深度融合：

$$m = Z_{est}\frac{\sigma_{obs}^2}{\sigma_{est}^2+\sigma_{obs}^2} + Z_{obs}\frac{\sigma_{est}^2}{\sigma_{est}^2+\sigma_{obs}^2}$$

计算权重系数 c_1, c_2：

$$c_1 = \frac{a_{est}}{a_{est}+b_{est}}\frac{1}{\sqrt{2\pi}(\sigma_{obs}+\sigma_{est})}e^{-\theta}$$

$$\theta = \frac{(Z_{obs}-Z_{est})^2}{2(\sigma_{obs}^2+\sigma_{est}^2)}$$

权重系数 c_1 将决定着新的观测深度值和方差对本次融合深度值和方差的加权比重，当观测值和估计值越接近，该系数越大，进而新的观测值所占本次融合的权重越大。与卡尔曼滤波不同的是，由于此系数的存在，很大程度减少了部分方差很小的离群深度观测值对融合的影响。原因如下，假设当前深度观测值 Z_{obs} 为离群噪点，$(Z_{obs}-Z_{obs})^2$ 必然很大，因此权重系数 c_1 会很小，所以会减少离群深度噪点对深度融合的影响。

$$c_2 = \frac{a_{est}}{a_{est}+b_{est}}\frac{1}{Z_{range}}$$

Z_{range} 为场景的平均深度，权重系数 c_2 由上次融合后的 Beta 分布参数决定，a_{est} 为局内点参数，当融合后的深度局内概率越大时，a_{est} 也越大，b_{est} 为局外点参数。c_2 控制由上一次融合后的深度得到的深度估计值对本次深度融合的加权比重，a_{est} 越大，估计值可信度越高，c_2 越大，因此在本次深度融合中估计值的权重就越大。

归一化系数c_1, c_2：

$$c_1 = \frac{c_1}{c_1 + c_2}$$

$$c_2 = \frac{c_2}{c_1 + c_2}$$

计算系数 f，e，用来更新 Beta 分布的参数a, b：

$$f = c_1 \frac{(a_{est} + 1)}{(a_{est} + b_{est} + 1)} + c_2 \frac{a_{est}}{(a_{est} + b_{est} + 1)}$$

$$e = c_1 \frac{(a_{est} + 1)(a_{est} + 2)}{(a_{est} + b_{est} + 1)(a_{est} + b_{est} + 2)} + c_2 \frac{a_{est}(a_{est} + 1)}{(a_{est} + b_{est} + 1)(a_{est} + b_{est} + 2)}$$

更新模型参数：

融合后像素的深度值为$Z_{update} = c_1 * m + c_2 * d_{est}$

融合后像素深度值的方差为$\sigma^2_{update} = c_1(S_2 + m^2) + c_2(\sigma^2_{est} + d^2_{est}) - Z^2_{update}$

其中$S_2 = \frac{\sigma^2_{est}\sigma^2_{obs}}{\sigma^2_{est} + \sigma^2_{obs}}$

融合后的 Beta 分布参数分别为$a_{update} = \frac{e - f}{f - \frac{e}{f}}$，$b_{update} = a_{est}\frac{1 - f}{f}$

如果第一次深度观测值为离群噪点，但之后大部分的深度观测数据为有效深度，那么随着融合，深度数据会逐渐接近真实值。如果若干次的深度观测值数据相差很大，经过融合后，深度值的方差将会很大，而且深度值的局内点概率会很小，因此可以利用若干次融合的结果判别深度发散的像素点。

像素深度更新的收敛条件：直到深度方差收敛或者判断像素点是局外点后就停止更新模型。

融合更新策略如下：

当$\sigma^2 < \sigma^2_{thr}$且$\rho_{inlier} > \rho_{thr_inlier}$时，认为深度收敛，停止对深度估计，当$\rho_{inlier} < \rho_{thr_inlier}$时，认为深度发散，求取失败，停止对深度估计，其中$\rho_{inlier} = \frac{a}{a + b}$，其他情况下继续深度融合。

```cpp
//* 公式更新后验参数
//* 论文： Video-based, Real-Time Multi View Stereo
void DepthFilter::updateSeed(const float x, const float tau2, Seed* seed)
{
  float norm_scale = sqrt(seed->sigma2 + tau2);
  if(std::isnan(norm_scale))
    return;
  //! N(mu, sigma^2 + tau^2)
  boost::math::normal_distribution<float> nd(seed->mu, norm_scale);
  //! 1/s^2 = 1/sigma^2 + 1/tau^2
  float s2 = 1./(1./seed->sigma2 + 1./tau2);
  //! s2 * (mu/sigma^2 + x/tau^2)
  float m = s2*(seed->mu/seed->sigma2 + x/tau2);
  //! a/(a+b) * N (x|mu, sigma^2 + tau^2)
  float C1 = seed->a/(seed->a+seed->b) * boost::math::pdf(nd, x);
  //! b/(a+b) * U(x)
  float C2 = seed->b/(seed->a+seed->b) * 1./seed->z_range;
  //! C = C1 + C2, 归一化
  float normalization_constant = C1 + C2;
  C1 /= normalization_constant;
  C2 /= normalization_constant;
  float f = C1*(seed->a+1.)/(seed->a+seed->b+1.) + C2*seed->a/(seed->a+seed->b+1.);
  float e = C1*(seed->a+1.)*(seed->a+2.)/((seed->a+seed->b+1.)*(seed->a+seed->b+2.))
          + C2*seed->a*(seed->a+1.0f)/((seed->a+seed->b+1.0f)*(seed->a+seed->b+2.0f));

  // update parameters
  float mu_new = C1*m+C2*seed->mu;
  seed->sigma2 = C1*(s2 + m*m) + C2*(seed->sigma2 + seed->mu*seed->mu) - mu_new*mu_new;
  seed->mu = mu_new;
  seed->a = (e-f)/(f-e/f);
  seed->b = seed->a*(1.0f-f)/f;
}
```

```cpp
//* 公式求标准差(\tau)
//* 对应论文: REMODE: Probabilistic, Monocular Dense Reconstruction in Real Time
double DepthFilter::computeTau(
      const SE3& T_ref_cur,
      const Vector3d& f,
      const double z,
      const double px_error_angle)
{
  Vector3d t(T_ref_cur.translation());
  Vector3d a = f*z-t;
  double t_norm = t.norm();
  double a_norm = a.norm();
  double alpha = acos(f.dot(t)/t_norm); // dot product
  double beta = acos(a.dot(-t)/(t_norm*a_norm)); // dot product
  double beta_plus = beta + px_error_angle;
  double gamma_plus = PI-alpha-beta_plus; // triangle angles sum to PI
  double z_plus = t_norm*sin(beta_plus)/sin(gamma_plus); // law of sines
  return (z_plus - z); // tau
}
```

注：笔者所有代码参数用的是 USE_ROS 版本

5. 关键帧选取策略

关键帧选取策略在 frame_handler_mono.cpp 中的 needNewKf 函数，通过判断当前帧附近关键帧到当前帧的位移 t_x/d_m， t_y/d_m， t_z/d_m（d_m 是场景平均深度)是否大于设定的阈值来判断是否需要添加新的关键帧。kfselect_mindist(0.12)。

```
bool FrameHandlerMono::needNewKf(double scene_depth_mean)
{
  for(auto it=overlap_kfs_.begin(), ite=overlap_kfs_.end(); it!=ite; ++it)
  {
    Vector3d relpos = new_frame_->w2f(it->first->pos());
    if(fabs(relpos.x())/scene_depth_mean < Config::kfSelectMinDist() &&
       fabs(relpos.y())/scene_depth_mean < Config::kfSelectMinDist()*0.8 &&
       fabs(relpos.z())/scene_depth_mean < Config::kfSelectMinDist()*1.3)
      return false;
  }
  return true;
}
```

6. SVO 优缺点

优点：

1）速度快。主要原因在于：(a)SVO 只在关键帧上提取特征点，而不是所有帧。(b)Sparse Model-based Image Alignment 使用 inverse compositional 的方法。

2）深度滤波器在模型中加入了外点，3D 点只有在收敛时才会加入到地图中的 candidate，从而使地图中的外点很少。

3）因为使用了网格划分，关键点分布比较均匀。

缺点：

1）关键帧策略比较简单，对于探索性的旋转场景不太友善，因为这个时候关键帧可能因为不满足阈值条件丢失很多。

2）深度滤波器的收敛速度会导致跟踪失败。比如在旋转的场景下，地图点能投影上的数量可能比较少，但是特征点又不会很快收敛，导致此时没什么特征点，跟踪失败。

3）没有闭环，无法消除累积误差。

4）只有简单的与上一关键帧的重定位。

5）追踪部分：SVO 首先将当前帧与上一个追踪的帧比较，以求得粗略的位姿估计。这就要求上一帧是准确的，如果上一帧由于遮挡、模糊等原因丢失，那当前帧也会得到一个错误的结果。

6）追踪部分有直接法所有的缺点：怕模糊(需要全局曝光相机)，怕大运动(图像非凸性)，怕光照变化(灰度不变假设)。

7）depth filter 收敛比较慢，结果比较依赖于准确的位姿估计。收敛的种子点比例不高，很多计算浪费在不收敛的点上。

8）相比于纯高斯的逆深度，SVO 的主要特点是能够通过 a,b 来判断一个种子点是否为 outlier。然而在特征点法中也能通过描述来判断 outlier，所以不具有明显优势。

References

Journal papers

Forster, Christian , M. Pizzoli , and Davide Scaramuzza, "SVO: Fast semi-direct monocular visual odometry." IEEE International Conference on Robotics & Automation IEEE, 2014.

S. Baker and I. Matthews, “Lucas-Kanade 20 Years On: A Unifying Framework: Part 1,” International Journal of Computer Vision, vol. 56, no. 3, pp. 221–255, 2002.

M. Pizzoli, C. Forster, and D. Scaramuzza, “REMODE: Probabilistic, Monocular Dense Reconstruction in Real Time,” in Proc. IEEE Int. Conf. on Robotics and Automation, 2014.

G. Vogiatzis and C. Hernández, “Video-based, Real-Time Multi View Stereo,” Image and Vision Computing, vol. 29, no. 7, 2011.

Books/ Book chapters

Gao, x. (2017) Visual slam 14: from theory to practice, ISBN 978721311048, Beijing, Electronic Industry Press.

Internet resources

SVO 深度解析(三)之深度滤波(建图部分)(2017)
https://blog.csdn.net/zhubaohua_bupt/article/details/74911000?utm_medium=distribute.pc_relevant.none-task-blog-title-2&spm=1001.2101.3001.4242

能否具体解释下 svo 的运动估计与深度估计两方面？(2017)
https://www.zhihu.com/question/39904950

SVO 原理解析(2016)
https://www.cnblogs.com/luyb/p/5773691.html