

多传感器融合定位

第7章 基于图优化的定位方法

主讲人 任乾

北京理工大学本硕

自动驾驶从业者





目录

-  1. 基于图优化的定位简介
-  2. 边缘化原理及应用
-  3. 基于kitti的实现原理
-  4. ceres 基础知识
-  5. lio-mapping 介绍



目录

-  1. 基于图优化的定位简介
-  2. 边缘化原理及应用
-  3. 基于kitti的实现原理
-  4. ceres 基础知识
-  5. lio-mapping介绍

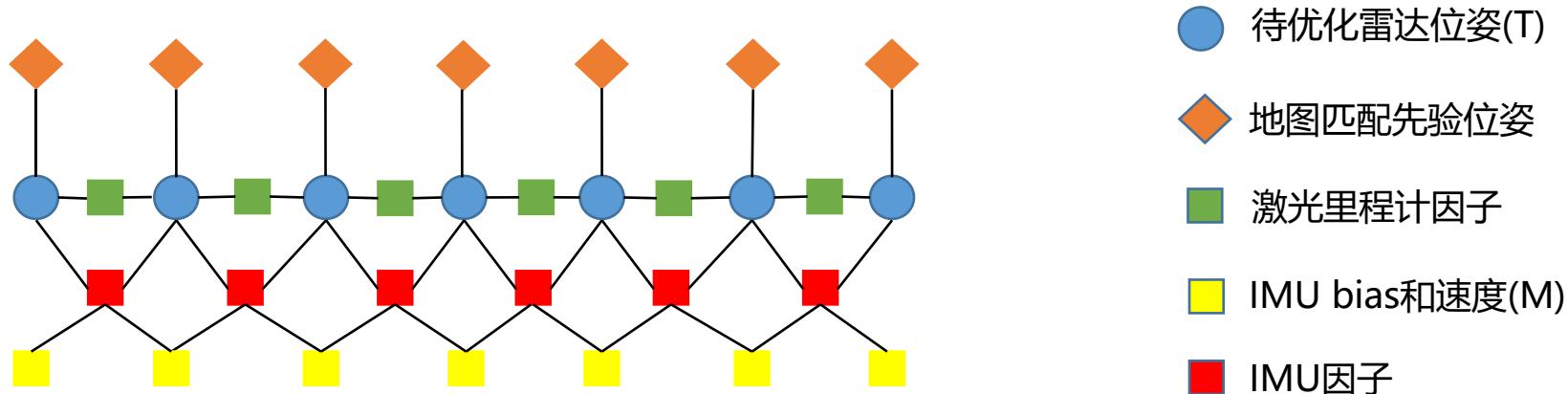


基于图优化的定位流程

1. 核心思路

核心思路是把融合方法从滤波换成图优化，其元素不再是简单的惯性解算，而是预积分。

一个暴力的模型可以设计为：



缺陷：随着时间的进行，图模型会越来越大，导致无法达到实时性。

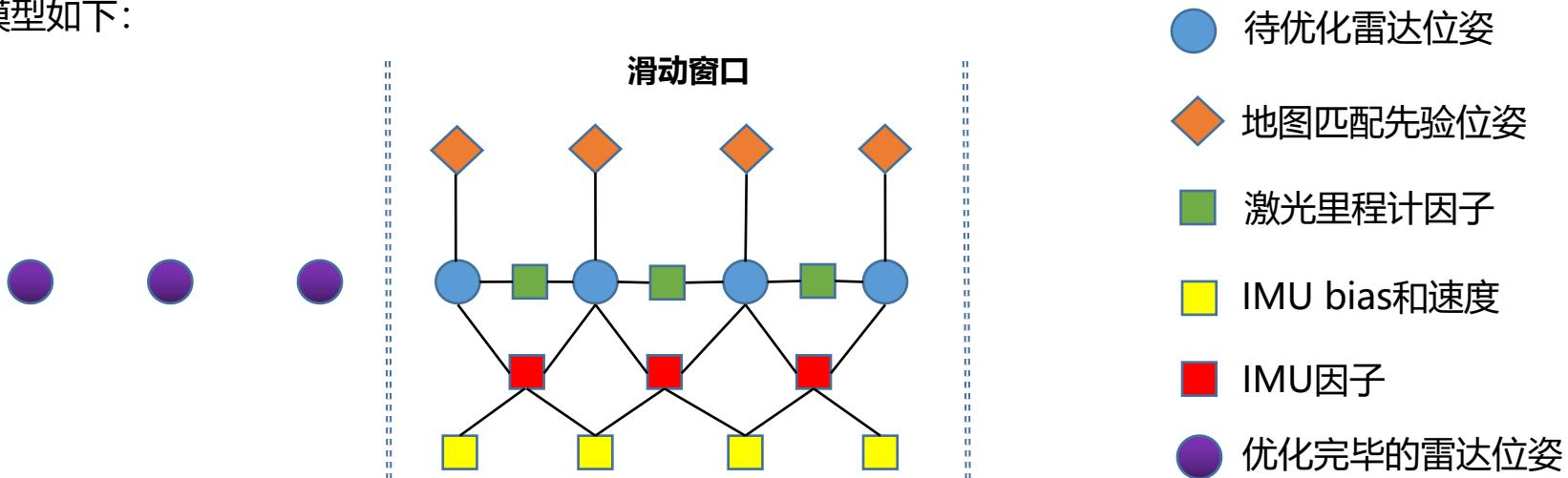


基于图优化的定位流程

1. 核心思路

解决方法：不断删除旧的帧，只优化最新的几帧，即维持一个滑动窗口。

模型如下：



问题：直接从模型中删除，等于损失了信息。

解法：通过模型把旧帧的约束传递下来，即边缘化(后面讲具体细节)。



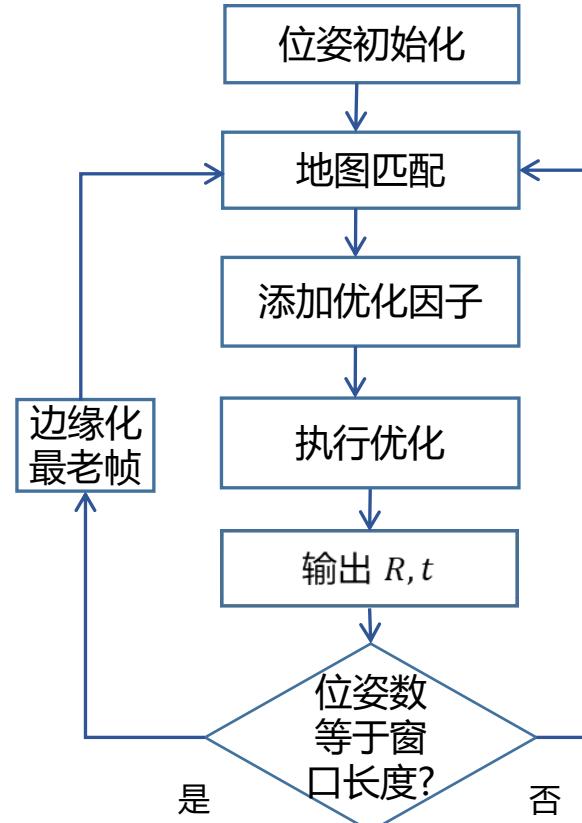
基于图优化的定位流程

2. 定位流程

整个流程：不断往滑窗里添加新信息，并边缘化旧信息。

需要注意的是：

- 1) 正常行驶时，不必像建图那样，提取稀疏的关键帧；
- 2) 停车时，需要按一定策略提取关键帧，但删除的是次新帧，因此不需要边缘化。



基于图优化的定位流程图



目录

-  1. 基于图优化的定位简介
-  2. 边缘化原理及应用
-  3. 基于kitti的实现原理
-  4. ceres 基础知识
-  5. lio-mapping介绍



边缘化原理及应用

1. 边缘化原理

优化问题具有如下通用形式：

$$HX = b$$

并可拆解成如下形式：

$$\begin{bmatrix} H_{mm} & H_{mr} \\ H_{rm} & H_{rr} \end{bmatrix} \begin{bmatrix} X_m \\ X_r \end{bmatrix} = \begin{bmatrix} b_m \\ b_r \end{bmatrix}$$

拆解的目的是通过一系列操作，把 X_m 从状态量里删除掉，并把它的约束保留下。

在滑窗模式里，这个 X_m 即为要边缘化掉的量。

回顾舒尔补：

给定矩阵

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

它可以通过如下变换，变成上三角矩阵，即

$$\begin{bmatrix} I & 0 \\ -CA^{-1} & I \end{bmatrix} \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} A & B \\ 0 & \Delta_A \end{bmatrix}$$

其中， $\Delta_A = D - CA^{-1}B$ 称为 A 关于 M 的舒尔补。



边缘化原理及应用

1. 边缘化原理

拆解后的优化问题，可通过舒尔补对矩阵三角化，
即

$$= \begin{bmatrix} I & 0 \\ -H_{rm}H_{mm}^{-1} & I \end{bmatrix} \begin{bmatrix} H_{mm} & H_{mr} \\ H_{rm} & H_{rr} \end{bmatrix} \begin{bmatrix} X_m \\ X_r \end{bmatrix}$$
$$= \begin{bmatrix} I & 0 \\ -H_{rm}H_{mm}^{-1} & I \end{bmatrix} \begin{bmatrix} b_m \\ b_r \end{bmatrix}$$

进一步化简得，

$$= \begin{bmatrix} H_{mm} & H_{mr} \\ 0 & H_{rr} - H_{rm}H_{mm}^{-1}H_{mr} \end{bmatrix} \begin{bmatrix} X_m \\ X_r \end{bmatrix}$$
$$= \begin{bmatrix} b_m \\ b_r - H_{rm}H_{mm}^{-1}b_m \end{bmatrix}$$

此时，可以利用等式第2行直接得到：

$$(H_{rr} - H_{rm}H_{mm}^{-1}H_{mr})X_r = b_r - H_{rm}H_{mm}^{-1}b_m$$

其含义为：此时可以不依赖 X_m 求解出 X_r ，
若我们只关心 X_r 的值，则可以把 X_m 从模型里
删除。



边缘化原理及应用

2. 从滤波角度理解边缘化

kalman滤波是此前已经熟悉的，从边缘化的角度重新看一遍滤波器的推导，更有利于深入理解。

运动模型与观测模型分别为：

$$\mathbf{x}_k = \mathbf{A}_{k-1} \mathbf{x}_{k-1} + \mathbf{v}_k + \mathbf{w}_k$$

$$\mathbf{y}_k = \mathbf{C}_k \mathbf{x}_k + \mathbf{n}_k$$

其中 $k = 1 \dots K$

状态量的求解，可以等效为如下模型

$$\hat{\mathbf{x}} = \arg \min_{\mathbf{x}} J(\mathbf{x})$$

其中

$$J(\mathbf{x}) = \sum_{k=0}^K (J_{v,k}(\mathbf{x}) + J_{y,k}(\mathbf{x}))$$

$$J_{v,k}(\mathbf{x}) = \begin{cases} \frac{1}{2} (\mathbf{x}_0 - \check{\mathbf{x}}_0)^T \check{\mathbf{P}}_0^{-1} (\mathbf{x}_0 - \check{\mathbf{x}}_0), & k = 0 \\ \frac{1}{2} (\mathbf{x}_k - \mathbf{A}_{k-1} \mathbf{x}_{k-1} - \mathbf{v}_k)^T \\ \times \mathbf{Q}_k^{-1} (\mathbf{x}_k - \mathbf{A}_{k-1} \mathbf{x}_{k-1} - \mathbf{v}_k), & k = 1 \dots K \end{cases}$$

$$J_{y,k}(\mathbf{x}) = \frac{1}{2} (\mathbf{y}_k - \mathbf{C}_k \mathbf{x}_k)^T \mathbf{R}_k^{-1} (\mathbf{y}_k - \mathbf{C}_k \mathbf{x}_k), \quad k = 0 \dots K$$

注：此处直接给出结果，具体化简过程可参考《机器人中的状态估计》3.1.2节。



边缘化原理及应用

2. 从滤波角度理解边缘化

将上述模型整理为更简洁的形式，令

$$\mathbf{z} = \begin{bmatrix} \check{\mathbf{x}}_0 \\ \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_K \\ \mathbf{y}_0 \\ \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_K \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \mathbf{x}_0 \\ \vdots \\ \mathbf{x}_K \end{bmatrix}$$

$$\mathbf{H} = \left[\begin{array}{cccccc} 1 & & & & & \\ -\mathbf{A}_0 & 1 & & & & \\ & \ddots & \ddots & & & \\ \hline \mathbf{C}_0 & & & -\mathbf{A}_{K-1} & 1 & \\ & \mathbf{C}_1 & & & & \\ & & \ddots & & & \\ & & & & & \mathbf{C}_K \end{array} \right]$$

$$\mathbf{W} = \left[\begin{array}{ccc|c} \check{\mathbf{P}}_0 & & & \\ & \mathbf{Q}_1 & & \\ & & \ddots & \\ \hline & & & \mathbf{Q}_K \\ & & & \hline & & & \mathbf{R}_0 \\ & & & & \mathbf{R}_1 \\ & & & & & \mathbf{R}_K \end{array} \right]$$



边缘化原理及应用

2. 从滤波角度理解边缘化

此时，目标函数可以重新表示为

$$J(\mathbf{x}) = \frac{1}{2}(\mathbf{z} - \mathbf{Hx})^T \mathbf{W}^{-1} (\mathbf{z} - \mathbf{Hx})$$

求解其最小值，即另其一阶导为零

$$\left. \frac{\partial J(\mathbf{x})}{\partial \mathbf{x}^T} \right|_{\hat{\mathbf{x}}} = -\mathbf{H}^T \mathbf{W}^{-1} (\mathbf{z} - \mathbf{H}\hat{\mathbf{x}}) = \mathbf{0}$$

即

$$(\mathbf{H}^T \mathbf{W}^{-1} \mathbf{H}) \hat{\mathbf{x}} = \mathbf{H}^T \mathbf{W}^{-1} \mathbf{z}$$

然而，这是批量求解模型，当只关心当前时刻(k时刻)状态时，应改为滤波模型。

假设上一时刻后验为

$$\left\{ \hat{\mathbf{x}}_{k-1}, \hat{\mathbf{P}}_{k-1} \right\}$$

目标是得到当前时刻后验

$$\left\{ \hat{\mathbf{x}}_{k-1}, \hat{\mathbf{P}}_{k-1}, \mathbf{v}_k, \mathbf{y}_k \right\} \mapsto \left\{ \hat{\mathbf{x}}_k, \hat{\mathbf{P}}_k \right\}$$



边缘化原理及应用

2. 从滤波角度理解边缘化

由于马尔可夫性，仅与前一时刻有关，因此令

$$\mathbf{z}_k = \begin{bmatrix} \hat{\mathbf{x}}_{k-1} \\ \mathbf{v}_k \\ \mathbf{y}_k \end{bmatrix}$$

$$\mathbf{H}_k = \begin{bmatrix} 1 \\ -\mathbf{A}_{k-1} & 1 \\ & \mathbf{C}_k \end{bmatrix}$$

$$\mathbf{W}_k = \begin{bmatrix} \hat{\mathbf{P}}_{k-1} & & \\ & \mathbf{Q}_k & \\ & & \mathbf{R}_k \end{bmatrix}$$

则模型的解为

$$(\mathbf{H}_k^T \mathbf{W}_k^{-1} \mathbf{H}_k) \hat{\mathbf{x}} = \mathbf{H}_k^T \mathbf{W}_k^{-1} \mathbf{z}_k$$

其中

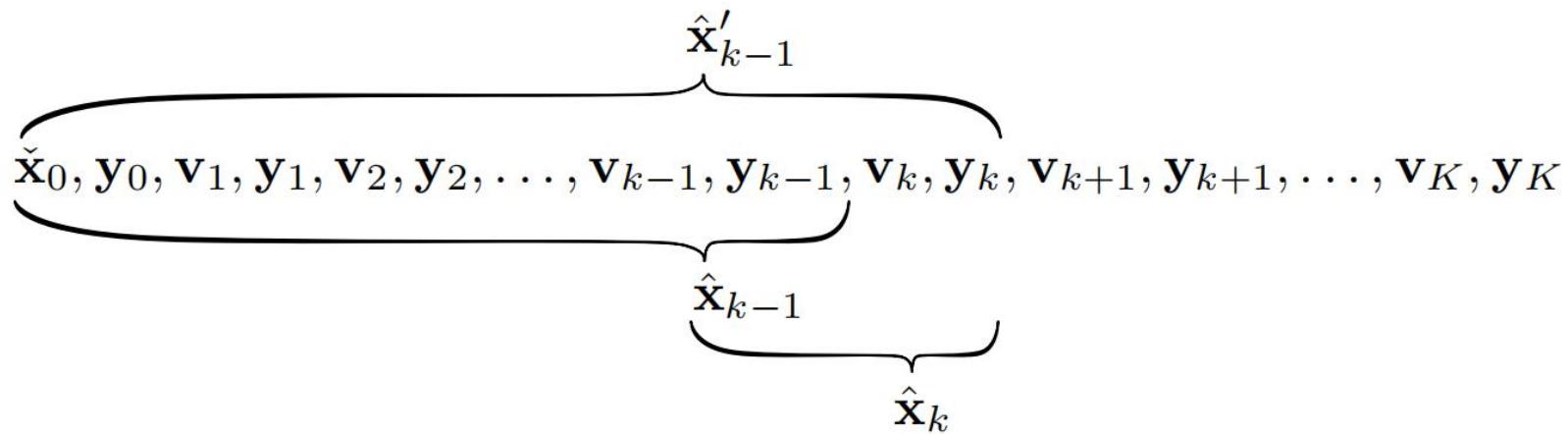
$$\hat{\mathbf{x}} = \begin{bmatrix} \hat{\mathbf{x}}'_{k-1} \\ \hat{\mathbf{x}}_k \end{bmatrix}$$



边缘化原理及应用

2. 从滤波角度理解边缘化

$\hat{\mathbf{x}}_{k-1}$ 和 $\hat{\mathbf{x}}'_{k-1}$ 有本质区别，下图可以明确展示





边缘化原理及应用

2. 从滤波角度理解边缘化

在此基础上，求解模型可以展开为

$$\begin{bmatrix} \hat{\mathbf{P}}_{k-1}^{-1} + \mathbf{A}_{k-1}^T \mathbf{Q}_k^{-1} \mathbf{A}_{k-1} & -\mathbf{A}_{k-1}^T \mathbf{Q}_k^{-1} \\ -\mathbf{Q}_k^{-1} \mathbf{A}_{k-1} & \mathbf{Q}_k^{-1} + \mathbf{C}_k^T \mathbf{R}_k^{-1} \mathbf{C}_k \end{bmatrix} \begin{bmatrix} \hat{\mathbf{x}}'_{k-1} \\ \hat{\mathbf{x}}_k \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{P}}_{k-1}^{-1} \hat{\mathbf{x}}_{k-1} - \mathbf{A}_{k-1}^T \mathbf{Q}_k^{-1} \mathbf{v}_k \\ \mathbf{Q}_k^{-1} \mathbf{v}_k + \mathbf{C}_k^T \mathbf{R}_k^{-1} \mathbf{y}_k \end{bmatrix}$$

利用舒尔补，等式两边左乘如下矩阵，便可以直接求解出 $\hat{\mathbf{x}}_k$ ，且不需求解 $\hat{\mathbf{x}}'_{k-1}$

$$\begin{bmatrix} 1 & 0 \\ \mathbf{Q}_k^{-1} \mathbf{A}_{k-1} \left(\hat{\mathbf{P}}_{k-1}^{-1} + \mathbf{A}_{k-1}^T \mathbf{Q}_k^{-1} \mathbf{A}_{k-1} \right)^{-1} & 1 \end{bmatrix}$$



边缘化原理及应用

2. 从滤波角度理解边缘化

可得：

$$\hat{\mathbf{P}}_k^{-1} \hat{\mathbf{x}}_k = \check{\mathbf{P}}_k^{-1} (\mathbf{A}_{k-1} \hat{\mathbf{x}}_{k-1} + \mathbf{v}_k) + \mathbf{C}_k^T \mathbf{R}_k^{-1} \mathbf{y}_k$$

其中

$$\check{\mathbf{P}}_k = \mathbf{Q}_k + \mathbf{A}_{k-1} \hat{\mathbf{P}}_{k-1} \mathbf{A}_{k-1}^T$$

$$\hat{\mathbf{P}}_k = (\check{\mathbf{P}}_k^{-1} + \mathbf{C}_k^T \mathbf{R}_k^{-1} \mathbf{C}_k)^{-1}$$

注：此处直接给出结果，具体化简过程可参考《机器人中的状态估计》3.3.2节。



边缘化原理及应用

2. 从滤波角度理解边缘化

以上过程，核心即为边缘化，因此滤波(IEKF)可以看做长度为1的滑动窗口。

Gauss-Newton iterates over the entire trajectory, but runs offline and not in constant time

$\mathbf{x}_0 \quad \mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_3 \quad \cdots \quad \mathbf{x}_{k-2} \quad \mathbf{x}_{k-1} \quad \mathbf{x}_k \quad \mathbf{x}_{k+1} \quad \mathbf{x}_{k+2} \quad \cdots \quad \mathbf{x}_K$



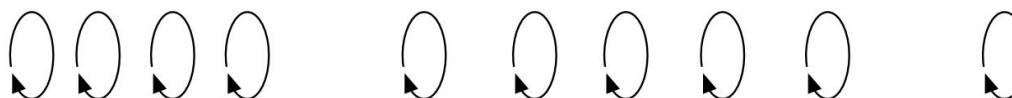
Sliding-window filters iterate over several timesteps at once, run online and in constant time

$\mathbf{x}_0 \quad \mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_3 \quad \cdots \quad \mathbf{x}_{k-2} \quad \mathbf{x}_{k-1} \quad \mathbf{x}_k \quad \mathbf{x}_{k+1} \quad \mathbf{x}_{k+2} \quad \cdots \quad \mathbf{x}_K$



IEKF iterates at only one timestep at a time, but runs online and in constant time

$\mathbf{x}_0 \quad \mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_3 \quad \cdots \quad \mathbf{x}_{k-2} \quad \mathbf{x}_{k-1} \quad \mathbf{x}_k \quad \mathbf{x}_{k+1} \quad \mathbf{x}_{k+2} \quad \cdots \quad \mathbf{x}_K$





目录

-  1. 基于图优化的定位简介
-  2. 边缘化原理及应用
-  3. 基于kitti的实现原理
-  4. ceres 基础知识
-  5. lio-mapping介绍



基于kitti的实现原理

1. 基于地图定位的滑动窗口模型

1) 窗口优化模型构成

在图优化模型中，优化模型也可写成如下形式：

$$\mathbf{J}^\top \Sigma \mathbf{J} \delta \mathbf{x} = -\mathbf{J}^\top \Sigma \mathbf{r}$$

其中

\mathbf{r} 是残差；

\mathbf{J} 是残差关于状态量的雅可比；

Σ 是信息矩阵。

在kitti工程中，基于地图定位的滑动窗口，其残差包括：

- 地图匹配位姿和优化变量的残差
- 激光里程计相对位姿和优化变量的残差
- IMU预积分和优化变量的残差
- 边缘化形成的先验因子对应的残差

此处先介绍前3项，第4项待边缘化后介绍。



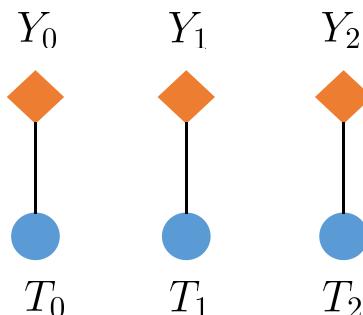
基于kitti的实现原理

1. 基于地图定位的滑动窗口模型

2) 地图匹配位姿和优化变量的残差

该残差对应的因子为地图先验因子。

一个因子仅约束一个位姿，其模型如下：



残差关于优化变量的雅可比，可视化如下：

	T_0	M_0	T_1	M_1	T_2	M_2
r_{Y0}	orange					
r_{Y1}			orange			
r_{Y2}					orange	

因此，对应的Hessian矩阵的可视化为：

	T_0	M_0	T_1	M_1	T_2	M_2
T_0	orange					
M_0						
T_1			orange			
M_1						
T_2					orange	
M_2						



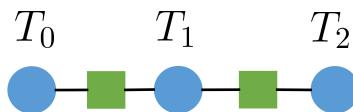
基于kitti的实现原理

1. 基于地图定位的滑动窗口模型

3) 激光里程计相对位姿和优化变量的残差

该残差对应的因子为激光里程计因子。

一个因子约束两个位姿，其模型如下：



残差关于优化变量的雅可比，可视化如下：

	T_0	M_0	T_1	M_1	T_2	M_2
r_{L0}	■		■			
r_{L1}			■		■	

因此，对应的Hessian矩阵可视化为：

	T_0	M_0	T_1	M_1	T_2	M_2
T_0	■		■			
M_0						
T_1			■		■	
M_1						
T_2				■		■
M_2						



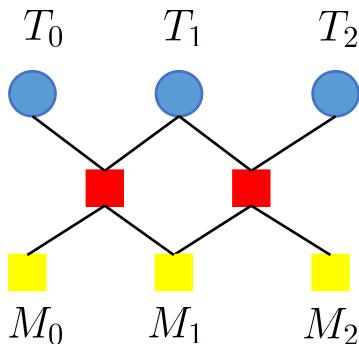
基于kitti的实现原理

1. 基于地图定位的滑动窗口模型

4) IMU预积分和优化变量的残差

该残差对应的因子为IMU因子。

一个因子约束两个位姿，并约束两个时刻IMU的速度与bias。



残差关于优化变量的雅可比，可视化如下：

	T_0	M_0	T_1	M_1	T_2	M_2
r_{M0}	red	red	red	red	white	white
r_{M1}	white	white	red	red	red	red

因此，对应的Hessian矩阵可视化为：

	T_0	M_0	T_1	M_1	T_2	M_2
T_0	red	red	red	red	white	white
M_0	red	red	red	red	white	white
T_1	red	red	red	red	white	red
M_1	red	red	red	red	red	red
T_2	white	white	red	red	red	red
M_2	white	white	red	red	red	red



基于kitti的实现原理

1. 基于地图定位的滑动窗口模型

5) 完整模型

完整Hessian矩阵，即为以上各因子对应矩阵的累加。



基于kitti的实现原理

1. 基于地图定位的滑动窗口模型

5) 完整模型

上述过程用公式可表示为：

$$\underbrace{\mathbf{J}^\top \Sigma \mathbf{J}}_{\mathbf{H}} \delta \mathbf{x} = -\underbrace{\mathbf{J}^\top \Sigma \mathbf{r}}_{\mathbf{b}}$$

其中

$$\mathbf{r} = \begin{bmatrix} \mathbf{r}_{Y0} \\ \mathbf{r}_{Y1} \\ \mathbf{r}_{Y2} \\ \mathbf{r}_{L0} \\ \mathbf{r}_{L1} \\ \mathbf{r}_{M0} \\ \mathbf{r}_{M1} \end{bmatrix}$$

$$\mathbf{J} = \frac{\partial \mathbf{r}}{\partial \delta \mathbf{x}} = \begin{bmatrix} \frac{\partial \mathbf{r}_{Y0}}{\partial \delta \mathbf{x}} \\ \frac{\partial \mathbf{r}_{Y1}}{\partial \delta \mathbf{x}} \\ \frac{\partial \mathbf{r}_{Y2}}{\partial \delta \mathbf{x}} \\ \frac{\partial \mathbf{r}_{L0}}{\partial \delta \mathbf{x}} \\ \frac{\partial \mathbf{r}_{L1}}{\partial \delta \mathbf{x}} \\ \frac{\partial \mathbf{r}_{M0}}{\partial \delta \mathbf{x}} \\ \frac{\partial \mathbf{r}_{M1}}{\partial \delta \mathbf{x}} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_1 \\ \mathbf{J}_2 \\ \mathbf{J}_3 \\ \mathbf{J}_4 \\ \mathbf{J}_5 \\ \mathbf{J}_6 \\ \mathbf{J}_7 \end{bmatrix}$$
$$\mathbf{J}^\top = [\mathbf{J}_1^\top \quad \mathbf{J}_2^\top \quad \mathbf{J}_3^\top \quad \mathbf{J}_4^\top \quad \mathbf{J}_5^\top \quad \mathbf{J}_6^\top \quad \mathbf{J}_7^\top]$$

矩阵乘法写成累加形式为：

$$\sum_{i=1}^7 \mathbf{J}_i^\top \Sigma_i \mathbf{J}_i \delta \mathbf{x} = - \sum_{i=1}^7 \mathbf{J}_i^\top \Sigma_i \mathbf{r}_i$$

此累加过程，即对应前面可视化时各矩阵叠加。

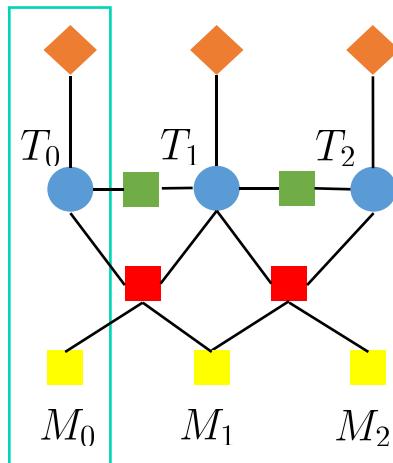


基于kitti的实现原理

2. 边缘化过程

1) 移除旧的帧

假设窗口长度为3，在加入新的帧之前，需要先边缘化掉旧的帧，即下图方框中的变量。



用前述公式，可以表示为

$$(H_{rr} - H_{rm}H_{mm}^{-1}H_{mr})\delta x_r = b_r - H_{rm}H_{mm}^{-1}b_m$$

在实际使用中，会把它拆成两部分

$$\begin{cases} H_{rr}\delta x_r = b_r \\ (-H_{rm}H_{mm}^{-1}H_{mr})\delta x_r = -H_{rm}H_{mm}^{-1}b_m \end{cases}$$

其中，第1行表示剩余变量对应的各类因子的组合，而第2行作为单独的因子，可称之为边缘化先验因子。



基于kitti的实现原理

2. 边缘化过程

1) 移除老的帧

上述过程，通过可视化可以表示为

$$\begin{array}{c} T_0 \ M_0 \ T_1 \ M_1 \ T_2 \ M_2 \\ \begin{matrix} M_0 \\ M_1 \\ M_2 \end{matrix} \end{array} \xrightarrow{\text{边缘化}} \begin{array}{c} H_{rr} \\ T_1 \ M_1 \ T_2 \ M_2 \\ \begin{matrix} M_1 \\ M_2 \end{matrix} \end{array} - \begin{array}{c} H_{rm} \quad H_{mm}^{-1} \quad H_{mr} \\ \begin{matrix} M_0 \\ M_1 \end{matrix} \quad \begin{matrix} M_0 \\ M_1 \end{matrix} \quad \begin{matrix} M_1 \\ M_2 \end{matrix} \\ - \quad = \quad = \end{array} \begin{array}{c} H_{rr} - H_{rm}H_{mm}^{-1}H_{mr} \\ \begin{matrix} M_0 \\ M_1 \end{matrix} \quad \begin{matrix} M_0 \\ M_1 \end{matrix} \quad \begin{matrix} M_1 \\ M_2 \end{matrix} \end{array}$$

The diagram illustrates the marginalization process. On the left, a 6x6 matrix is shown with columns labeled $T_0, M_0, T_1, M_1, T_2, M_2$ and rows labeled M_0, M_1, M_2 . A blue arrow labeled "边缘化" points to the right, where the matrix is transformed. The transformed matrix has columns T_1, M_1, T_2, M_2 and rows M_1, M_2 . To the right of the transformation, the matrices H_{rm} , H_{mm}^{-1} , and H_{mr} are shown. Below these, the resulting matrix after subtraction is shown, with its columns labeled M_0, M_1, M_2 .

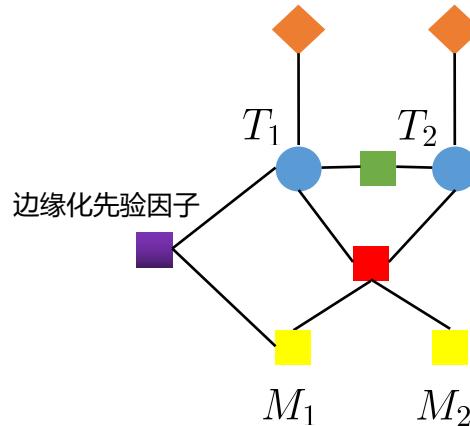


基于kitti的实现原理

2. 边缘化过程

1) 移除老的帧

边缘化之后，模型如下：



注意：边缘化先验因子只有在第一次边缘化之前是不存在的，完成第一次边缘化之后就一直存在，并且随着后续新的边缘化进行，其内容不断更新。

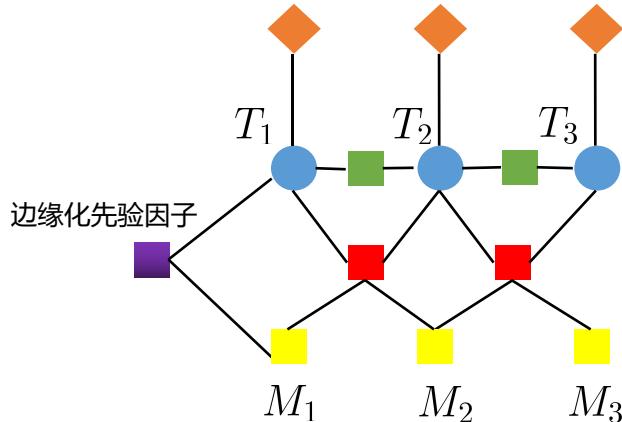


基于kitti的实现原理

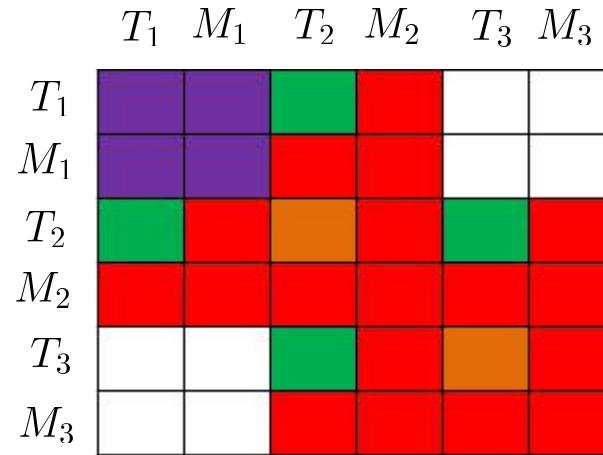
2. 边缘化过程

2) 添加新的帧

添加新的帧之后，模型如下：



此处直接给出新的Hessian矩阵可视化结果：



此后，随着定位过程的进行，便不断循环“边缘化老帧->添加新帧”的过程，从而维持窗口长度不变。

该过程的代码实现可参考后面lio-mapping的实现，理解后者便很容易实现前者。



目录

-  1. 基于图优化的定位简介
-  2. 边缘化原理及应用
-  3. 基于kitti的实现原理
-  4. **ceres** 基础知识
-  5. lio-mapping介绍



ceres 基础知识

1. 基本概念

优化任务一般可以表示成如下形式：

$$\min_{\mathbf{x}} \frac{1}{2} \sum_i \rho_i \left(\|f_i(x_{i_1}, \dots, x_{i_k})\|^2 \right)$$

$$\text{s.t. } l_j \leq x_j \leq u_j$$

其中

- 1) $\rho_i \left(\|f_i(x_{i_1}, \dots, x_{i_k})\|^2 \right)$ 称为残差块，即 ResidualBlock；
- 2) $f_i(\cdot)$ 称为代价函数，即 CostFunction；
- 3) $[x_{i_1}, \dots, x_{i_k}]$ 这一系列参数称为参数块，即 ParameterBlock；
- 4) $\rho_i(\cdot)$ 称为损失函数，即 LossFunction。



ceres 基础知识

2. 自动求导与解析求导

1) 自动求导

以一个简单的例子来说明该问题，假设代价函数为 $f(x) = 10 - x$

则首先编写 CostFunctor 的代码如下：

```
struct CostFunctor {  
    template <typename T>  
    bool operator()(const T* const x, T* residual) const {  
        residual[0] = T(10.0) - x[0];  
        return true;  
    }  
};
```



ceres 基础知识

2. 自动求导与解析求导

随后可直接构建 ceres 优化问题

```
int main(int argc, char** argv) {
    google::InitGoogleLogging(argv[0]);

    // The variable to solve for with its initial value.
    double initial_x = 5.0;
    double x = initial_x;

    // Build the problem.
    Problem problem;

    // Set up the only cost function (also known as residual). This uses
    // auto-differentiation to obtain the derivative (jacobian).
    CostFunction* cost_function =
        new AutoDiffCostFunction<CostFunctor, 1, 1>(new CostFunctor);
    problem.AddResidualBlock(cost_function, NULL, &x);
```

此处的 AutoDiffCostFunction 即代表当前模式为自动求导，它使用 CostFunctor 中的残差公式自动求解出导数，而不需要手动给出导数形式



ceres 基础知识

2. 自动求导与解析求导

2) 解析求导

相应地，解析求导的含义就是直接给出导数的解析形式，而不是ceres去推导

```
class QuadraticCostFunction : public ceres::SizedCostFunction<1, 1> {  
    // 定义一个CostFunction或SizedCostFunction(如果参数和残差在编译时就已知了)的子类。  
public:  
    virtual ~QuadraticCostFunction() {}  
    virtual bool Evaluate(double const* const* parameters,  
        //输入参数数组  
        double* residuals,  
        //输出残差数组  
        double** jacobians) const {  
        //输出雅可比行列式  
  
    const double x = parameters[0][0];  
    residuals[0] = 10 - x;  
  
    // Compute the Jacobian if asked for.  
    if (jacobians != NULL && jacobians[0] != NULL) {  
        jacobians[0][0] = -1;  
    }  
    return true;  
};
```

- 第一个参数为ResidualBlock维数
- 第二个参数为第一个ParameterBlock维数
- 当有多个ParameterBlock时，此处参数就不只两个

- 给出雅可比时，ceres会直接使用该雅可比
- 不给出雅可比时，ceres就会自动去求导



ceres 基础知识

2. 自动求导与解析求导

随后可构建优化问题

```
int main(int argc, char** argv) {
    google::InitGoogleLogging(argv[0]);

    // 初始化待优化变量
    double x = 0.5;
    const double initial_x = x;

    // 构建问题
    Problem problem;

    // 设置残差函数
    CostFunction* cost_function = new QuadraticCostFunction;
    problem.AddResidualBlock(cost_function, NULL, &x);
```

这种使用方式，就是vio/lio中使用ceres构建优化问题的方式



ceres 基础知识

2. 自动求导与解析求导

3) 总结

- 自动求导实现方便，但效率会比解析求导低 (比较 A-LOAM 和 F-LOAM)；
- 实际使用中，能够自动求导且效率没有形成障碍的，优先使用自动求导；
- 除这两种方法外，还有数值求导 (SLAM问题中不常见，不过多介绍) 。



ceres 基础知识

3. LocalParameterization

1) 产生原因

姿态使用的是四元数，它是4维，但只有3个自由度，且不支持加法。

优化时，每更新一次都要对其进行归一化，否则就不再是单位四元数。

这种方式显然比较复杂，因此要寻找3个参数的运算方法。

2) 解决方法

常见解法如下，即定义对应的3维localparameter，并实现关于它的广义加法和雅可比。

```
class CERES_EXPORT QuaternionParameterization : public LocalParameterization {  
public:  
    virtual ~QuaternionParameterization() {}  
    virtual bool Plus(const double* x,  
                      const double* delta,  
                      double* x_plus_delta) const;  
    virtual bool ComputeJacobian(const double* x,  
                                double* jacobian) const;  
    virtual int GlobalSize() const { return 4; }  
    virtual int LocalSize() const { return 3; }  
};
```



ceres 基础知识

3. LocalParameterization

在vins和lio-mapping中，把平移和旋转放在一个参数块里，因此是直接定义该7维参数对应的6维localparameter

```
class PoseLocalParameterization : public ceres::LocalParameterization {  
  
    virtual bool Plus(const double *x, const double *delta, double *x_plus_delta) const;  
    virtual bool ComputeJacobian(const double *x, double *jacobian) const;  
    virtual int GlobalSize() const { return 7; };  
    virtual int LocalSize() const { return 6; };  
  
};
```



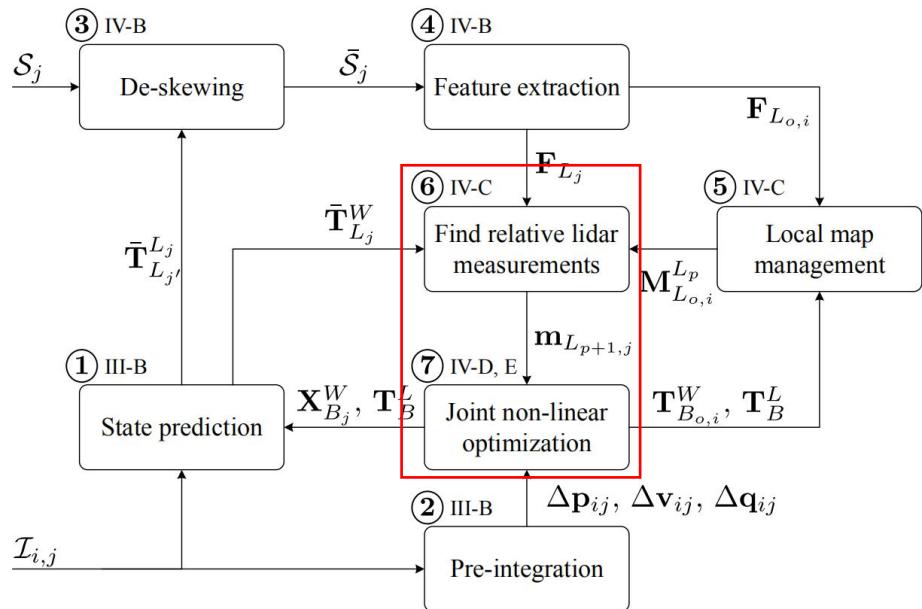
目录

-  1. 基于图优化的定位简介
-  2. 边缘化原理及应用
-  3. 基于kitti的实现原理
-  4. ceres 基础知识
-  5. lio-mapping介绍



lio-mapping 介绍

1. 核心思想



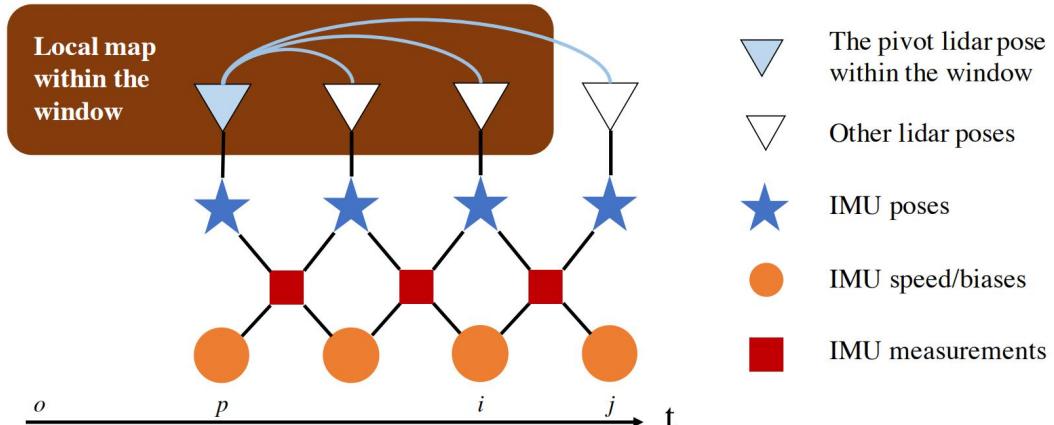
基于滑动窗口方法，把雷达线/面特征、IMU预积分等的约束放在一起进行优化。



lio-mapping 介绍

1. 核心思想

- 1) o到i是滑窗；
- 2) 只有p到i的位姿在滑窗中优化；
- 3) o到p是为了构建局部地图，防止地图过于稀疏；
- 4) 局部地图都投影到p的位姿处；
- 5) 滑窗中点云约束是当前优化帧和局部地图特征匹配，因此特征对应的因子约束的是p帧和k帧($p < k \leq j$)。





lio-mapping 介绍

1. 核心思想

其优化模型为

$$\begin{aligned} \min_{\mathbf{X}} \frac{1}{2} \left\{ & \| \mathbf{r}_{\mathcal{P}}(\mathbf{X}) \|^2 + \sum_{\substack{m \in \mathbf{m}_{L_\alpha} \\ \alpha \in \{p+1, \dots, j\}}} \| \mathbf{r}_{\mathcal{L}}(m, \mathbf{X}) \|^2_{\mathbf{C}_{L_\alpha}^m} \right. \\ & \left. + \sum_{\beta \in \{p, \dots, j-1\}} \left\| \mathbf{r}_{\mathcal{B}}(z_{\beta+1}^\beta, \mathbf{X}) \right\|_{\mathbf{C}_{B_{\beta+1}}^{B_\beta}}^2 \right\} \end{aligned}$$

其中

$\mathbf{r}_{\mathcal{P}}(\mathbf{X})$ 是边缘化产生的先验因子对应的残差；

$\mathbf{r}_{\mathcal{L}}(m, \mathbf{X})$ 是点云特征匹配对应的残差；

$\mathbf{r}_{\mathcal{B}}(z_{\beta+1}^\beta, \mathbf{X})$ 是IMU约束对应的残差。



lio-mapping 介绍

2. 具体流程

流程讲解思路：

- 以前述kitti中实现原理为基础，此处只是多了点云特征的约束；
- 只介绍可借鉴的内容，因此不介绍bias、外参初始化和外参优化等内容。



lio-mapping 介绍

2. 具体流程

2.1 各类因子

1) IMU 因子

- 1) 残差为 15 维, 分别是 P(3)、Q(3)、V(3)、Ba(3)、Bg(3)
- 2) 第一个参数块 7 维, 包含 k 时刻 P、Q
- 3) 第二个参数块 9 维, 包含 k 时刻 V、Ba、Bg
- 4) 第三个参数块 7 维, 包含 k+1 时刻 P、Q
- 5) 第四个参数块 9 维, 包含 k+1 时刻 V、Ba、Bg

```
class ImuFactor : public ceres::SizedCostFunction<15, 7, 9, 7, 9> {

public:
    ImuFactor() = delete;
    ImuFactor(std::shared_ptr<IntegrationBase> pre_integration) : pre_integration_{
        pre_integration} {
        // NOTE: g_vec_ is the gravity in laser's original frame
        g_vec_ = pre_integration_->g_vec_;
    }
    virtual bool Evaluate(double const *const *parameters, double *residuals, double **jacobians) const { ... }

private:
    std::shared_ptr<IntegrationBase> pre_integration_;
    Eigen::Vector3d g_vec_;

    const double eps_ = 10e-8;
};
```

Evaluate函数内部为计算残差和雅可比, 与之前预积分公式推导一致, 不再展开讲解。



lio-mapping 介绍

2. 具体流程

添加imu因子

```
if (estimator_config_.imu_factor) {
    for (int i = 0; i < estimator_config_.opt_window_size; ++i) {
        int j = i + 1;
        int opt_i = int(estimator_config_.window_size -
                        estimator_config_.opt_window_size + i);
        int opt_j = opt_i + 1;
        if (pre_integrations_[opt_j]->sum_dt_ > 10.0) {
            continue;
        }

        auto *f = new ImuFactor(pre_integrations_[opt_j]);

        // TODO: is it better to use g_vec_ as global parameter?
        ceres::internal::ResidualBlock *res_id = problem.AddResidualBlock(
            f, nullptr, para_pose_[i], para_speed_bias_[i], para_pose_[j],
            para_speed_bias_[j]);
        res_ids_pim.push_back(res_id);
    }
}
```

从p帧处开始取，o和p之间的不优化

约束相邻两时刻各自的P、Q、V、Ba、Bg



lio-mapping 介绍

2. 具体流程

2) 点云面特征因子

- 1) 残差为 1 维，即点到面的距离
- 2) 第一个参数块 7 维，包含 p 时刻 P、Q
- 3) 第二个参数块 7 维，包含 k 时刻 P、Q
- 4) 第三个参数块 7 维，为外参的 P、Q

```
class PivotPointPlaneFactor : public ceres::SizedCostFunction<1, 7, 7, 7> {

public:
    PivotPointPlaneFactor(const Eigen::Vector3d &point,
                          const Eigen::Vector4d &coeff);
    virtual bool Evaluate(double const *const *parameters, double *residuals, double **jacobians) const;
    void Check(double **parameters);

    Eigen::Vector3d point_;
    Eigen::Vector4d coeff_;

    // TODO: necessary?
    // static Eigen::Matrix3d sqrt_info;
    static double sum_t;

    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
};

};
```

Evaluate函数内部为计算残差和雅可比，残差与第一章一致，雅可比采用李代数推导。



lio-mapping 介绍

2. 具体流程

添加点云面特征

```
for (auto &feature : features) {  
    PointPlaneFeature feature_j;  
    feature->GetFeature(&feature_j);  
  
    const double &s = feature_j.score;  
  
    const Eigen::Vector3d &p_eigen = feature_j.point;  
    const Eigen::Vector4d &coeff_eigen = feature_j.coeffs;  
    // 第一个, 也就是p帧  
    if (i == 0) {  
    } else {  
        auto *f = new PivotPointPlaneFactor(p_eigen, coeff_eigen);  
        ceres::internal::ResidualBlock *res_id = problem.AddResidualBlock(  
            f, loss_function,  
            // NULL,  
            para_pose_[0], para_pose_[i], para_ex_pose_);  
        res_ids_proj.push_back(res_id);  
    }  
}
```

每个特征对应一个因子

两个时刻的位姿以及外参



lio-mapping 介绍

2. 具体流程

3) 边缘化先验因子

```
class MarginalizationFactor : public ceres::CostFunction {
public:
    MarginalizationFactor(MarginalizationInfo* _marginalization_info);
    virtual bool Evaluate(double const *const *parameters, double *residuals, double **jacobians) const;

    MarginalizationInfo* marginalization_info;
};
```

由于不确定边缘化后会和哪些量产生关联，因此没有固定size。
其详细内容待讲到边缘化实现时再展开。



lio-mapping 介绍

2. 具体流程

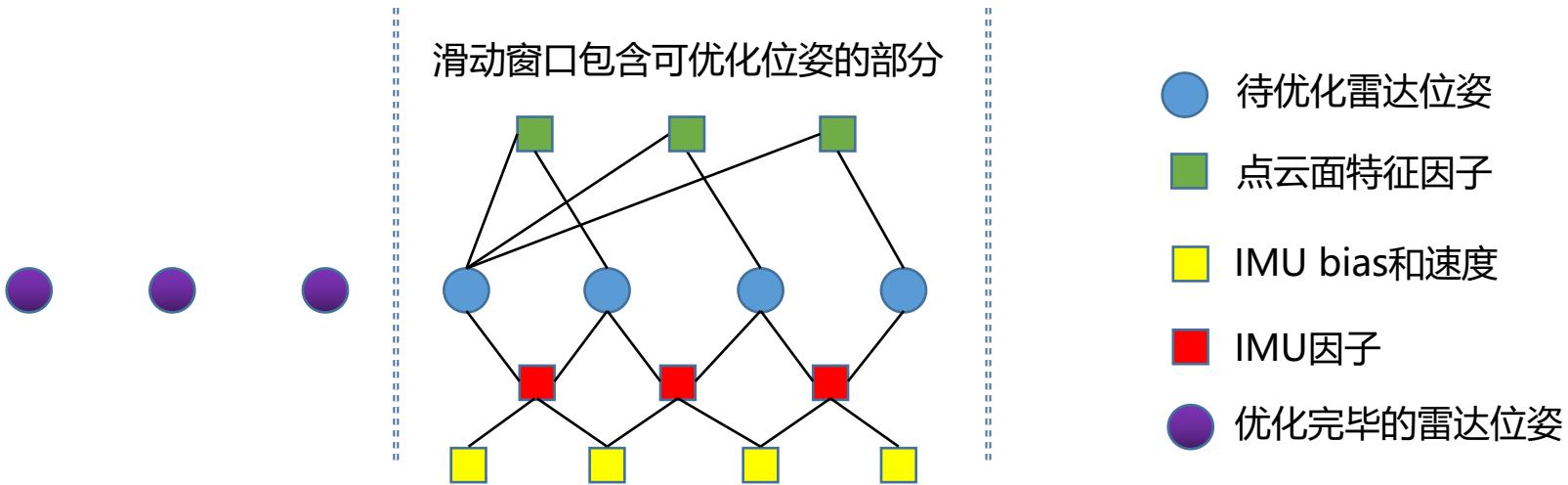
添加边缘化先验因子

```
// region 边缘化
if (estimator_config_.marginalization_factor) {
    if (last_marginalization_info) {
        // construct new marginalization_factor
        auto *marginalization_factor =
            new MarginalizationFactor(last_marginalization_info);
        //向问题中添加误差项
        res_id_marg =
            problem.AddResidualBlock(marginalization_factor, nullptr,
                                     last_marginalization_parameter_blocks);
        res_ids_marg.push_back(res_id_marg);
    }
}
```



lio-mapping 介绍

2.2. 滑窗模型



- 帧与帧之间通过特征约束，因此没有了激光里程计因子。
- 当前模型中没有使用点云的线特征。

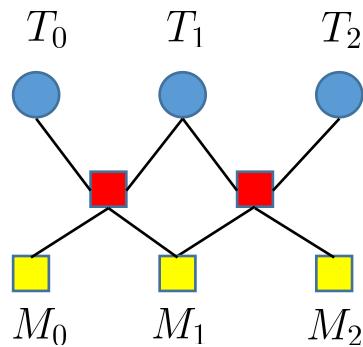


lio-mapping 介绍

2.2. 滑窗模型

1) IMU预积分和优化变量的残差

一个因子约束两个位姿，并约束两个
时刻 IMU 的速度与 bias



残差关于优化变量的雅可比可视化如下

	T_0	M_0	T_1	M_1	T_2	M_2
r_{M0}	red	red	red	red	white	white
r_{M1}	white	white	red	red	red	red

因此对应的Hessian矩阵的可视化为

	T_0	M_0	T_1	M_1	T_2	M_2
T_0	red	red	red	red	white	white
M_0	red	red	red	red	white	white
T_1	red	red	red	red	white	red
M_1	red	red	red	red	white	red
T_2	white	white	red	red	red	red
M_2	white	white	red	red	red	red

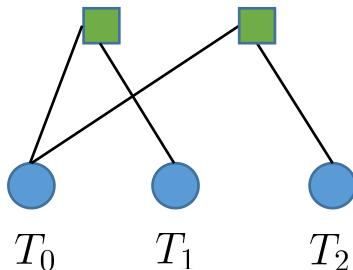


lio-mapping 介绍

2.2. 滑窗模型

2) 点云面特征对应的残差

一个因子约束两个位姿



残差关于优化变量的雅可比可视化如下

	T_0	M_0	T_1	M_1	T_2	M_2
r_{F0}	■		■			
r_{F1}	■				■	

因此对应的Hessian矩阵的可视化为

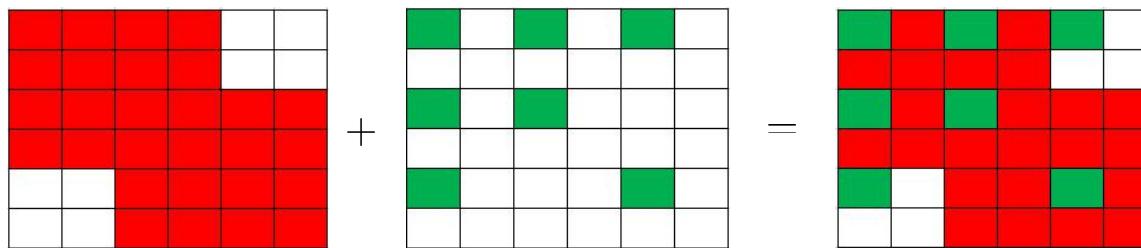
	T_0	M_0	T_1	M_1	T_2	M_2
T_0	■			■		
M_0						
T_1				■		
M_1						
T_2	■				■	
M_2						



lio-mapping 介绍

2.2. 滑窗模型

4) 完整模型



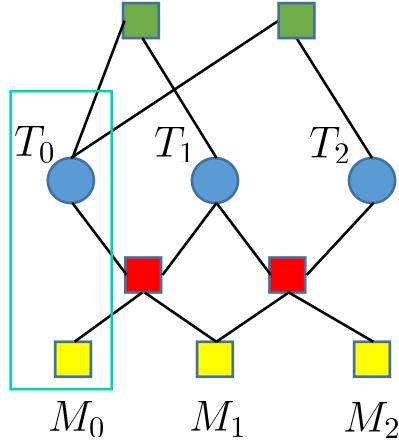
以上工程，就是前述代码中添加各类因子到模型的过程



lio-mapping 介绍

2.3. 边缘化

1) 边缘化模型



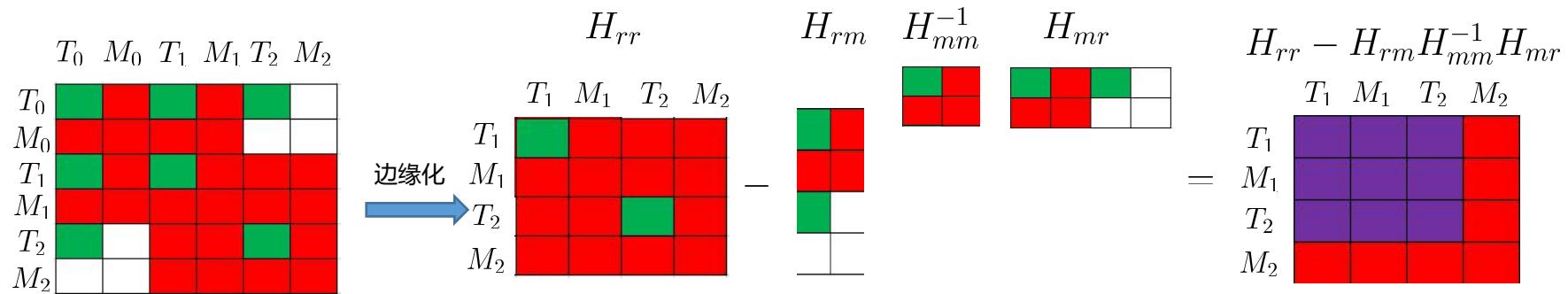
需要边缘化掉的为方框中的变量



lio-mapping 介绍

2.3. 边缘化

2) 边缘化可视化





lio-mapping 介绍

2.3. 边缘化

3) 边缘化实现

核心思路是把要边缘化掉的变量，以及跟这些变量被同一个因子约束的变量，汇总在一起。

三个函数：

- void addResidualBlockInfo()
- void preMarginalize()
- void marginalize()

五个变量：

- parameter_block_size：每个变量的维度
- parameter_block_data：每个变量的数据
- parameter_block_idx：每个变量在H矩阵中的索引
- m：需要marg掉的变量的总维度
- n：需要保留的变量的总维度

```
class MarginalizationInfo {  
public:  
    ~MarginalizationInfo();  
    int LocalSize(int size) const;  
    int GlobalSize(int size) const;  
    void AddResidualBlockInfo(ResidualBlockInfo *residual_block_info);  
    void PreMarginalize();  
    void Marginalize();  
    std::vector<double *> GetParameterBlocks(std::unordered_map<long, double *> &addr_shift);  
  
    std::vector<ResidualBlockInfo *> factors;  
    int m, n;  
    std::unordered_map<long, int> parameter_block_size; //global size  
    int sum_block_size;  
    std::unordered_map<long, int> parameter_block_idx; //local size  
    std::unordered_map<long, double *> parameter_block_data;  
  
    std::vector<int> keep_block_size; //global size  
    std::vector<int> keep_block_idx; //local size  
    std::vector<double *> keep_block_data;  
  
    Eigen::MatrixXd linearized_jacobians;  
    Eigen::VectorXd linearized_residuals;  
    const double eps = 1e-8;  
};
```

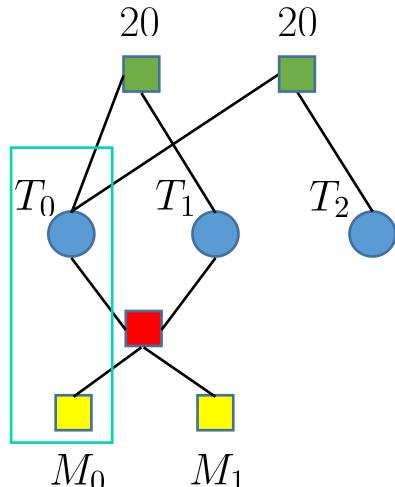


lio-mapping 介绍

2.3. 边缘化

3) 边缘化实现

在该例中，把和模型无关的量去除，
剩余部分如下(设每帧有20个面特征)



因此，放入MarginalizationInfo中的信息包括：

5个变量： $T_0 \quad M_0 \quad T_1 \quad M_1 \quad T_2$

41个因子：40个面特征因子、1个IMU因子

(此处假设的是第一次进行边缘化，若不是第一次，因子中还应该有边缘化先验因子)



lio-mapping 介绍

2.3. 边缘化

3) 边缘化实现

找出所有变量后，需要知道哪些是应该边缘化的，哪些是应该保留的

右图代码中，形参里_parameter_blocks 包含所有相关参数，而_drop_set即为这些参数中要边缘化掉的参数的id

```
struct ResidualBlockInfo {
    ResidualBlockInfo(ceres::CostFunction *_cost_function,
                      ceres::LossFunction *_loss_function,
                      std::vector<double *> _parameter_blocks,
                      std::vector<int> _drop_set)
        : cost_function(_cost_function),
          loss_function(_loss_function),
          parameter_blocks(_parameter_blocks),
          drop_set(_drop_set) {}

    void Evaluate();

    ceres::CostFunction *cost_function;
    ceres::LossFunction *loss_function;
    std::vector<double *> parameter_blocks;
    std::vector<int> drop_set;

    double **raw_jacobians;
    std::vector<Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>> jacobians;
    Eigen::VectorXd residuals;

    int localSize(int size) {
        return size == 7 ? 6 : size;
    }
};
```



lio-mapping 介绍

2.3. 边缘化

3) 边缘化实现

添加和IMU因子相关的ResidualBlockInfo

```
if (estimator_config_.imu_factor) {
    int pivot_idx =
        estimator_config_.window_size - estimator_config_.opt_window_size;
    if (pre_integrations_[pivot_idx + 1]->sum_dt_ < 10.0) {
        auto *imu_factor = new ImuFactor(pre_integrations_[pivot_idx + 1]);
        auto *residual_block_info = new ResidualBlockInfo(
            imu_factor, NULL,
            vector<double *>{para_pose_[0], para_speed_bias_[0], para_pose_[1],
                               para_speed_bias_[1]},
            vector<int>{0, 1});
        marginalization_info->AddResidualBlockInfo(residual_block_info);
    }
}
```

drop_set取0和1，就代表要边缘化掉 T_0 和 M_0



lio-mapping 介绍

2.3. 边缘化

3) 边缘化实现

添加和面特征因子相关的ResidualBlockInfo

```
for (auto &feature : features) {
    PointPlaneFeature feature_j;
    feature->GetFeature(&feature_j);

    const double &s = feature_j.score;

    const Eigen::Vector3d &p_eigen = feature_j.point;
    const Eigen::Vector4d &coeff_eigen = feature_j.coeffs;

    auto *pivot_point_plane_factor =
        new PivotPointPlaneFactor(p_eigen, coeff_eigen);

    auto *residual_block_info = new ResidualBlockInfo(
        pivot_point_plane_factor, loss_function,
        vector<double *>{para_pose_[0], para_pose_[i], para_ex_pose_},
        vector<int>{0});
    marginalization_info->AddResidualBlockInfo(residual_block_info);
}
```

drop_set取0, 代表只
边缘化第p帧



lio-mapping 介绍

2.3. 边缘化

3) 边缘化实现

在添加以上ResidualBlockInfo的同时，核心变量parameter_block_size 就被赋值

```
void MarginalizationInfo::AddResidualBlockInfo(ResidualBlockInfo *residual_block_info) {
    factors.emplace_back(residual_block_info);

    std::vector<double *> &parameter_blocks = residual_block_info->parameter_blocks;
    std::vector<int> parameter_block_sizes = residual_block_info->cost_function->parameter_block_sizes();

    for (int i = 0; i < static_cast<int>(residual_block_info->parameter_blocks.size()); i++) {
        double *addr = parameter_blocks[i];
        int size = parameter_block_sizes[i];
        parameter_block_size[reinterpret_cast<long>(addr)] = size;
    }

    for (int i = 0; i < static_cast<int>(residual_block_info->drop_set.size()); i++) {
        double *addr = parameter_blocks[residual_block_info->drop_set[i]];
        parameter_block_idx[reinterpret_cast<long>(addr)] = 0;
    }
}
```



lio-mapping 介绍

2.3. 边缘化

3) 边缘化实现

第二个核心函数的作用是计算每个因子对应的变量 (parameter_blocks)、误差项 (residuals)、雅可比矩阵 (jacobians)，并把变量数值放到parameter_block_data中

```
void MarginalizationInfo::PreMarginalize() {
    for (auto it : factors) {
        it->Evaluate();

        std::vector<int> block_sizes = it->cost_function->parameter_block_sizes();
        for (int i = 0; i < static_cast<int>(block_sizes.size()); i++) {
            long addr = reinterpret_cast<long>(it->parameter_blocks[i]);
            int size = block_sizes[i];
            if (parameter_block_data.find(addr) == parameter_block_data.end()) {
                double *data = new double[size];
                memcpy(data, it->parameter_blocks[i], sizeof(double) * size);
                parameter_block_data[addr] = data;
            }
        }
    }
}
```



lio-mapping 介绍

2.3. 边缘化

3) 边缘化实现

第三个核心函数的作用构建
Hessian矩阵，Schur掉需要
marg的变量，得到对剩余变
量的约束，即为边缘化约束
(先验约束)

函数的前半部分，对m、n和
parameter_block_idx这三个
核心变量进行了赋值

```
void MarginalizationInfo::Marginalize() {
    int pos = 0;
    for (auto &it : parameter_block_idx) {
        it.second = pos;
        pos += LocalSize(parameter_block_size[it.first]);
    }

    m = pos;

    for (const auto &it : parameter_block_size) {
        if (parameter_block_idx.find(it.first) == parameter_block_idx.end()) {
            parameter_block_idx[it.first] = pos;
            pos += LocalSize(it.second);
        }
    }

    n = pos - m;
}
```



lio-mapping 介绍

2.3. 边缘化

3) 边缘化实现

函数的中间部分开始构建Hessian矩阵，由于使用多线程，因此要给不同的线程平均分配因子

```
//multi thread
TicToc t_thread_summing;
pthread_t tids[NUM_THREADS];
ThreadsStruct threadsstruct[NUM_THREADS];
int i = 0;
for (auto it : factors) {
    threadsstruct[i].sub_factors.push_back(it);
    i++;
    i = i % NUM_THREADS;
}
for (int i = 0; i < NUM_THREADS; i++) {
    TicToc zero_matrix;
    threadsstruct[i].A = Eigen::MatrixXd::Zero(pos, pos);
    threadsstruct[i].b = Eigen::VectorXd::Zero(pos);
    threadsstruct[i].parameter_block_size = parameter_block_size;
    threadsstruct[i].parameter_block_idx = parameter_block_idx;
    int ret = pthread_create(&tids[i], NULL, ThreadsConstructA, (void *) &(threadsstruct[i]));
    if (ret != 0) {
        ROS_DEBUG("pthread_create error");
        ROS_BREAK();
    }
}
for (int i = NUM_THREADS - 1; i >= 0; i--) {
    pthread_join(tids[i], NULL);
    A += threadsstruct[i].A;
    b += threadsstruct[i].b;
}
```



lio-mapping 介绍

2.3. 边缘化

3) 边缘化实现

函数的最后便是执行边缘化，
得到边缘化先验因子

```
//TODO
Eigen::MatrixXd Amm = 0.5 * (A.block(0, 0, m, m) + A.block(0, 0, m, m).transpose());
Eigen::SelfAdjointEigenSolver<Eigen::MatrixXd> saes(Amm);

//ROS_ASSERT_MSG(saes.eigenvalues().minCoeff() >= -1e-4, "min eigenvalue %f", saes.eigenvalues().minCoeff());

Eigen::MatrixXd Amm_inv = saes.eigenvectors()
    * Eigen::VectorXd((saes.eigenvalues().array() > eps).select(saes.eigenvalues().array().inverse(), 0)).asDiagonal()
    * saes.eigenvectors().transpose();
//printf("error1: %f\n", (Amm * Amm_inv - Eigen::MatrixXd::Identity(m, m)).sum());

Eigen::VectorXd bmm = b.segment(0, m);
Eigen::MatrixXd Amr = A.block(0, m, m, n);
Eigen::MatrixXd Arm = A.block(m, 0, n, m);
Eigen::MatrixXd Arr = A.block(m, m, n, n);
Eigen::VectorXd brr = b.segment(m, n);
A = Arr - Arm * Amm_inv * Amr;
b = brr - Arm * Amm_inv * bmm;

Eigen::SelfAdjointEigenSolver<Eigen::MatrixXd> saes2(A);
Eigen::VectorXd S = Eigen::VectorXd((saes2.eigenvalues().array() > eps).select(saes2.eigenvalues().array(), 0));
Eigen::VectorXd
    S_inv = Eigen::VectorXd((saes2.eigenvalues().array() > eps).select(saes2.eigenvalues().array().inverse(), 0));

Eigen::VectorXd S_sqrt = S.cwiseSqrt();
Eigen::VectorXd S_inv_sqrt = S_inv.cwiseSqrt();

linearized_jacobians = S_sqrt.asDiagonal() * saes2.eigenvectors().transpose();
linearized_residuals = S_inv_sqrt.asDiagonal() * saes2.eigenvectors().transpose() * b;
```



lio-mapping 介绍

2.3. 边缘化

3) 边缘化实现

上述过程是假设第一次执行边缘化，当不是第一次时，步骤上只是多了在AddResidualBlockInfo时把边缘化先验因子也加入进来，剩余过程不变

```
if (last_marginalization_info) {
    vector<int> drop_set;
    for (int i = 0;
        i < static_cast<int>(last_marginalization_parameter_blocks.size());
        i++) {
        if (last_marginalization_parameter_blocks[i] == para_pose_[0] ||
            last_marginalization_parameter_blocks[i] == para_speed_bias_[0])
            drop_set.push_back(i);
    }
    // construct new marginalization_factor
    auto *marginalization_factor =
        new MarginalizationFactor(last_marginalization_info);
    auto *residual_block_info = new ResidualBlockInfo(
        marginalization_factor, NULL, last_marginalization_parameter_blocks,
        drop_set);

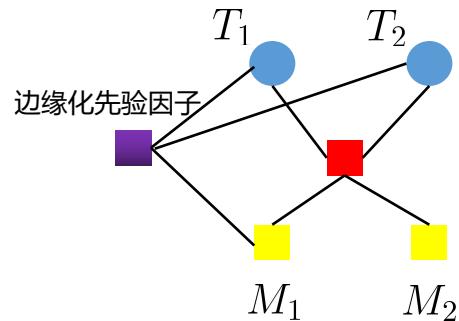
    marginalization_info->AddResidualBlockInfo(residual_block_info);
}
```



lio-mapping 介绍

2.4. 添加新帧

边缘化之后，模型如下

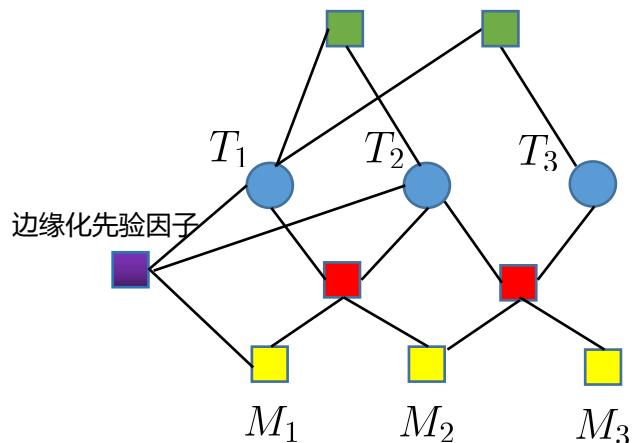


注意：由于面特征因子都是和第一帧(T_0)关联，当它边缘化掉之后，该因子便不存在，因此在加入新的帧的同时，还需要构建所有帧和当前第一帧(T_1)的面特征关联。



lio-mapping 介绍

添加新的帧以后，模型如下



Hessian矩阵可视化如下

	T_1	M_1	T_2	M_2	T_3	M_3	
T_1	green	purple	green	red	green		
M_1	purple		purple		white		
T_2	green		purple	green		red	
M_2	red		red		red		
T_3	green			red	green		
M_3				red		red	

此后不断循环该过程，便可以实现lio功能



lio-mapping 介绍

3. 讨论与思考

- 1) 为何 lio-mapping 耦合方式比 lio-sam 更紧，但效果却并不好？
- 2) 基于特征一致性的方法是否有必要？



作业

1. 根据第一章给出的点到线和点到面误差模型(如下), 推导特征残差对与它相关联的两个位姿的雅可比

点到线残差:

$$d_{\mathcal{E}} = \frac{\left| \left(\tilde{\mathbf{X}}_{(k+1,i)}^L - \bar{\mathbf{X}}_{(k,j)}^L \right) \times \left(\tilde{\mathbf{X}}_{(k+1,i)}^L - \bar{\mathbf{X}}_{(k,l)}^L \right) \right|}{\left| \bar{\mathbf{X}}_{(k,j)}^L - \bar{\mathbf{X}}_{(k,l)}^L \right|}$$

点到面残差:

$$d_{\mathcal{H}} = \frac{\left| \begin{array}{l} (\tilde{\mathbf{X}}_{(k+1,i)}^L - \bar{\mathbf{X}}_{(k,j)}^L) \\ ((\bar{\mathbf{X}}_{(k,j)}^L - \bar{\mathbf{X}}_{(k,l)}^L) \times (\bar{\mathbf{X}}_{(k,j)}^L - \bar{\mathbf{X}}_{(k,m)}^L)) \end{array} \right|}{\left| (\bar{\mathbf{X}}_{(k,j)}^L - \bar{\mathbf{X}}_{(k,l)}^L) \times (\bar{\mathbf{X}}_{(k,j)}^L - \bar{\mathbf{X}}_{(k,m)}^L) \right|}$$

提示: F-LOAM的代码中有推导结果, 它是解析式求导而非自动求导。



作业

2. 大作业（定位部分），以下二选一：

- 1) 简易版：使用大作业建图任务建立的地图，进行基于滤波方法的融合定位(第四章已完成)；
- 2) 进阶版：使用大作业建图任务建立的地图，进行基于滑动窗口的融合定位(整体思路本章第三小节已给出，代码实现可借鉴lio-mapping)。

备注：kitti数据缺点很多，若不愿使用，可替换成下面的数据集：

<https://github.com/weisongwen/UrbanNavDataset#1-hong-kong-dataset>



感谢聆听 !
Thanks for Listening !

