深蓝学院
shenlanxueyuan.com

多传感器融合定位

第10讲 基于优化的定位方法

主讲人 任 乾

北京理工大学本硕
自动驾驶从业者

目录

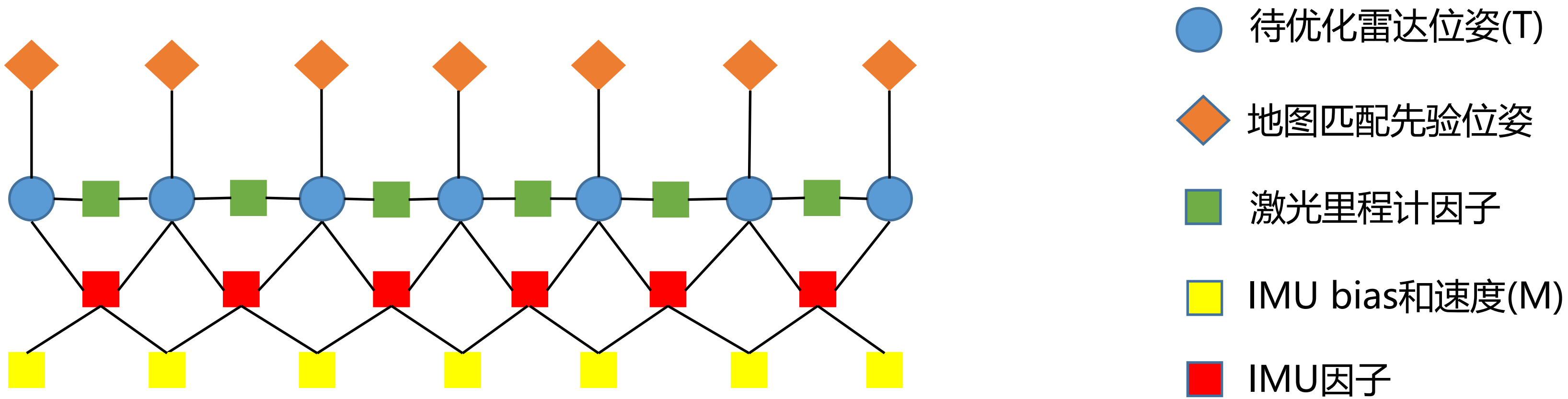
目录

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：



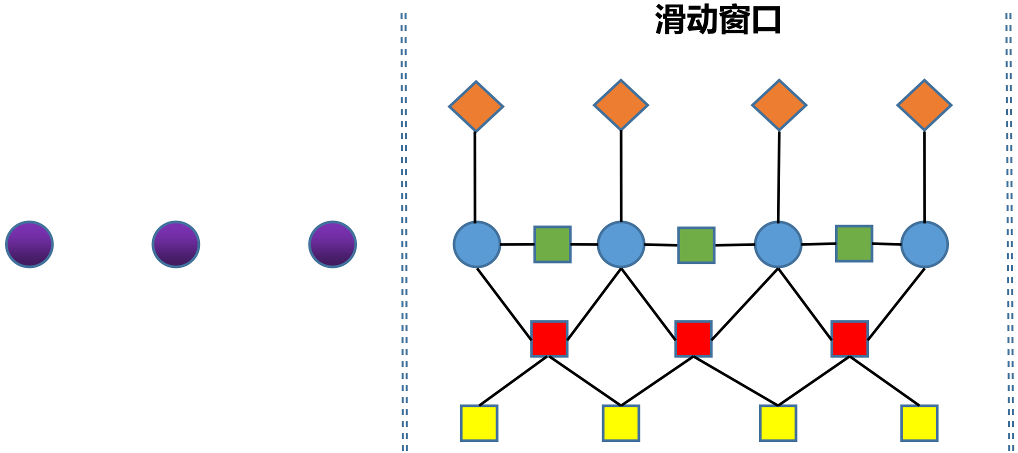
缺陷：随着时间的进行，图模型会越来越大，导致无法达到实时性。

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

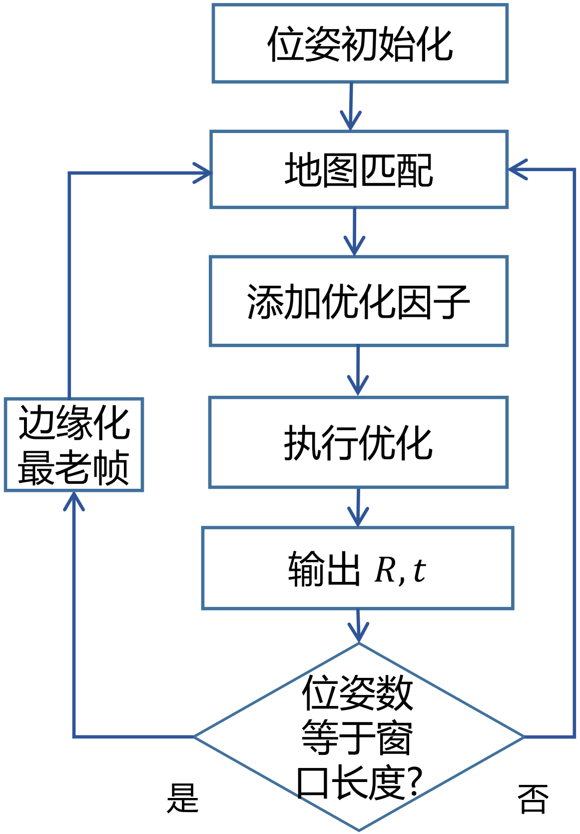
2. 定位流程

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

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来。

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

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

它可以通过如下变换，变成上三角矩阵，即

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

其中，$\Delta\mathbf{A} = \mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}$ 称为 $\mathbf{A}$ 关于 $\mathbf{M}$ 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。

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 $k = 1 \ldots K$

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

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

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

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

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。

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

$$\mathbf{z} = \left[\begin{array}{c} \check{\mathbf{x}}_0 \\ \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_K \\ \hline \mathbf{y}_0 \\ \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_K \end{array}\right], \quad \mathbf{x} = \begin{bmatrix} \mathbf{x}_0 \\ \vdots \\ \mathbf{x}_K \end{bmatrix}$$

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

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

2. 从滤波角度理解边缘化

此时，目标函数可以重新表示为

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

求解其最小值，即令其一阶导为零

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

即

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

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

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

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

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

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

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

则模型的解为

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

其中

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

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

$$\overbrace{\underbrace{\underbrace{\check{\mathbf{x}}_0, \mathbf{y}_0, \mathbf{v}_1, \mathbf{y}_1, \mathbf{v}_2, \mathbf{y}_2, \ldots, \mathbf{v}_{k-1}, \mathbf{y}_{k-1}}_{\hat{\mathbf{x}}_{k-1}}, \mathbf{v}_k, \mathbf{y}_k}_{\hat{\mathbf{x}}_k}}^{\hat{\mathbf{x}}'_{k-1}}, \mathbf{v}_{k+1}, \mathbf{y}_{k+1}, \ldots, \mathbf{v}_K, \mathbf{y}_K$$

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 $\hat{\mathbf{x}}'_{k-1}$

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

边缘化原理及应用

2. 从滤波角度理解边缘化

可得：

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

其中

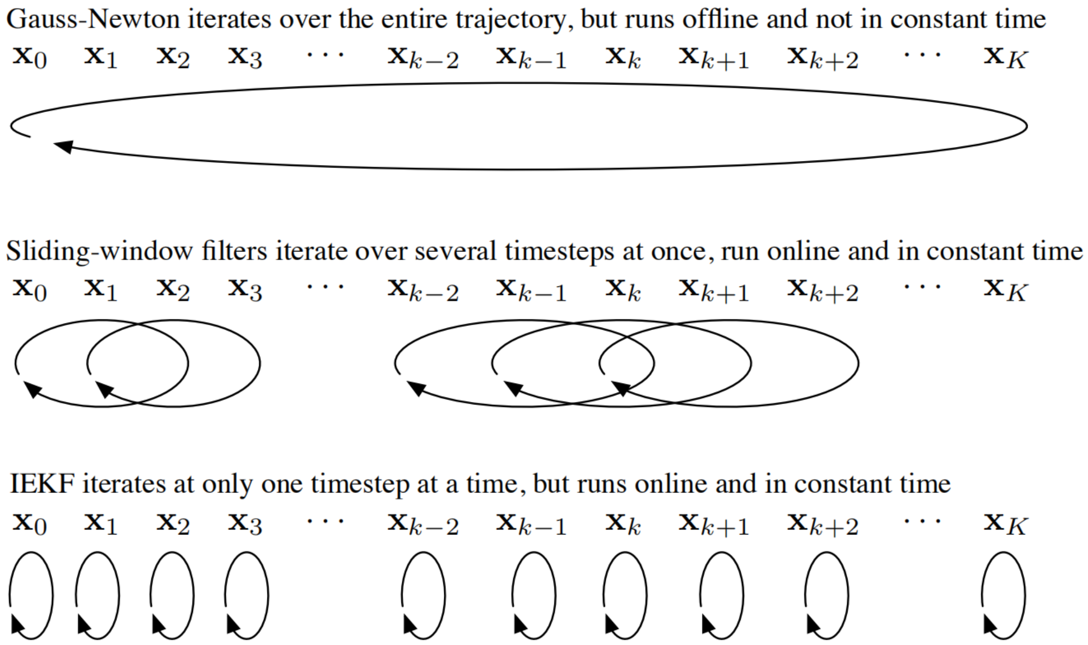
$$\check{\mathbf{P}}_k = \mathbf{Q}_k + \mathbf{A}_{k-1}\hat{\mathbf{P}}_{k-1}\mathbf{A}_{k-1}^T$$

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

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。

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。



目录

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：

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

其中

r 是残差；

J 是残差关于状态量的雅可比；

$\sum$ 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。

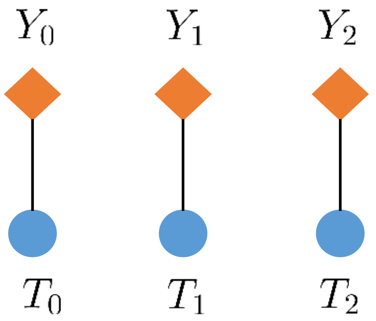
基于kitti的实现原理

1. 基于地图定位的滑动窗口模型

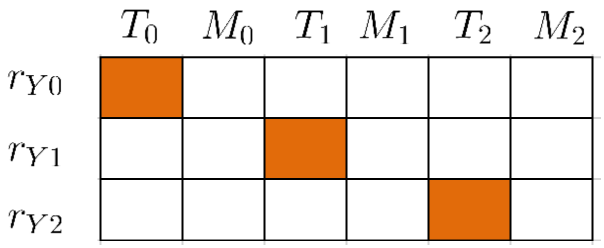
2）地图匹配位姿和优化变量的残差

该残差对应的因子为地图先验因子。

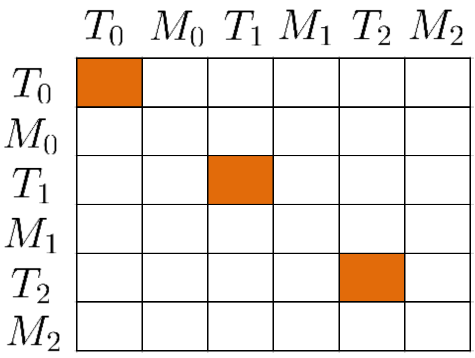
一个因子仅约束一个位姿，其模型如下：



残差关于优化变量的雅可比，可视化如下：



因此，对应的Hessian矩阵的可视化为：



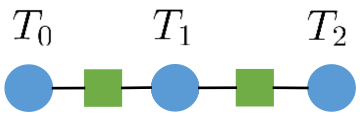
基于kitti的实现原理

1. 基于地图定位的滑动窗口模型

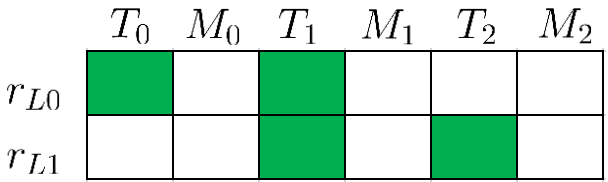
3）激光里程计相对位姿和优化变量的残差

该残差对应的因子为激光里程计因子。

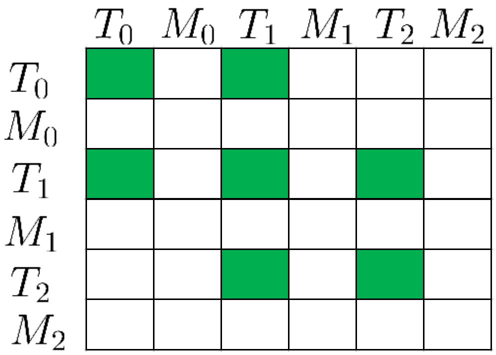
一个因子约束两个位姿，其模型如下：



残差关于优化变量的雅可比，可视化如下：



因此，对应的Hessian矩阵可视化为：



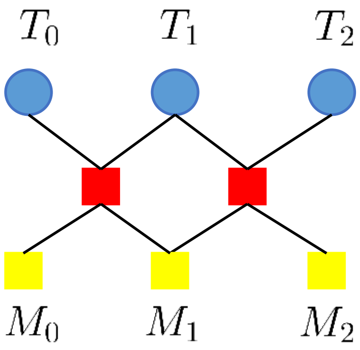
基于kitti的实现原理

1. 基于地图定位的滑动窗口模型

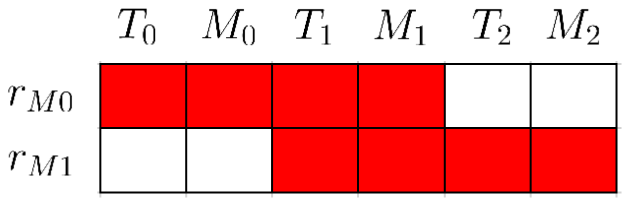
4）IMU预积分和优化变量的残差

该残差对应的因子为IMU因子。

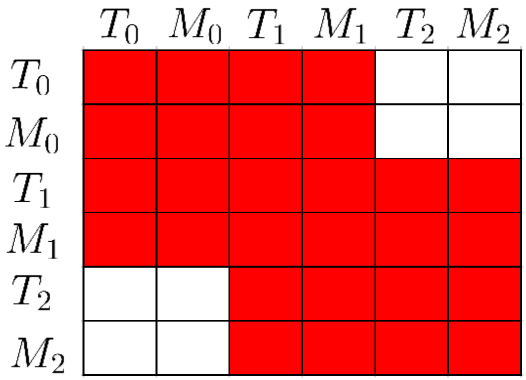
一个因子约束两个位姿，并约束两个时刻 IMU 的速度与 bias。



残差关于优化变量的雅可比，可视化如下：



因此，对应的Hessian矩阵可视化为：

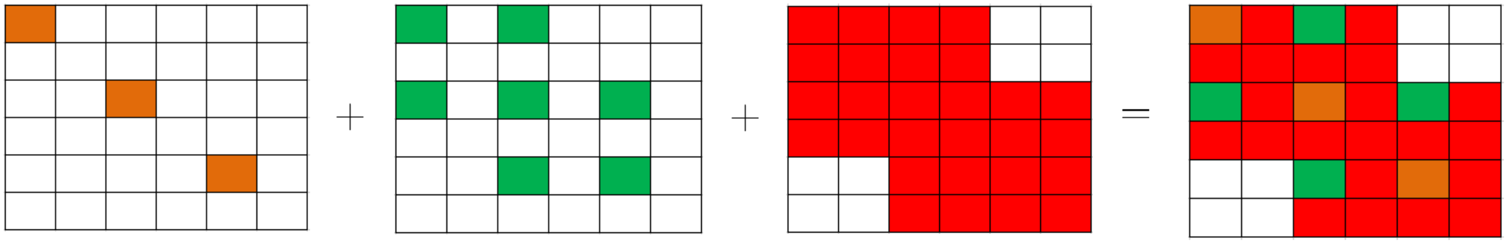


基于kitti的实现原理

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：

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

其中

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

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

$$\mathbf{J}^\top = \begin{bmatrix} \mathbf{J}_1^\top & \mathbf{J}_2^\top & \mathbf{J}_3^\top & \mathbf{J}_4^\top & \mathbf{J}_5^\top & \mathbf{J}_6^\top & \mathbf{J}_7^\top \end{bmatrix}$$

矩阵乘法写成累加形式为：

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

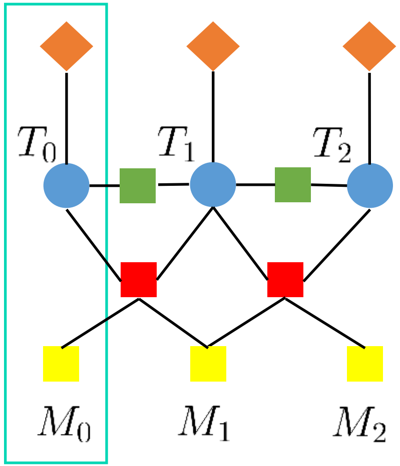
此累加过程，即对应前面可视化时各矩阵叠加。

基于kitti的实现原理

2. 边缘化过程

1）移除老的帧

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



用前述公式，可以表示为

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

但是在实际代码中，会把它拆成两步实现。

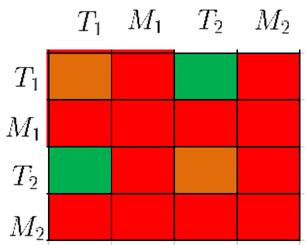
基于kitti的实现原理

2. 边缘化过程

1）移除老的帧

第一步：使用和要边缘化掉的量无关的因子，构建剩余变量对应的Hessian矩阵。

$$H_{rr}^{a}$$



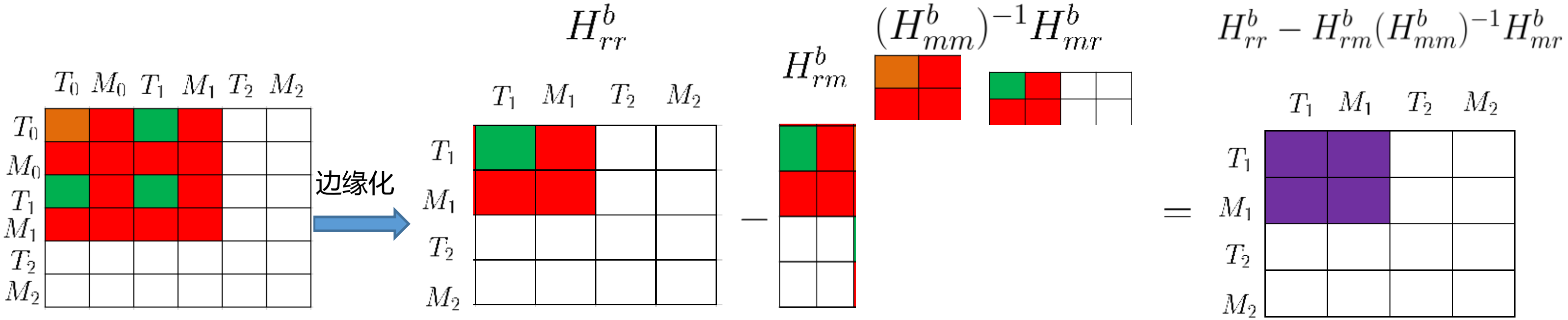
上标 a 代表是第一步中的变量，包含Hessian矩阵的完整表达式为

$$H_{rr}^{a}\delta x_r = b_r^{a}$$

2. 边缘化过程

1）移除老的帧

第二步：挑出和要边缘化掉的量相关的因子，构建待边缘化的Hessian矩阵，并使用舒尔补做边缘化。



上标 b 代表是第二步中的变量，包含Hessian矩阵的完整表达式为

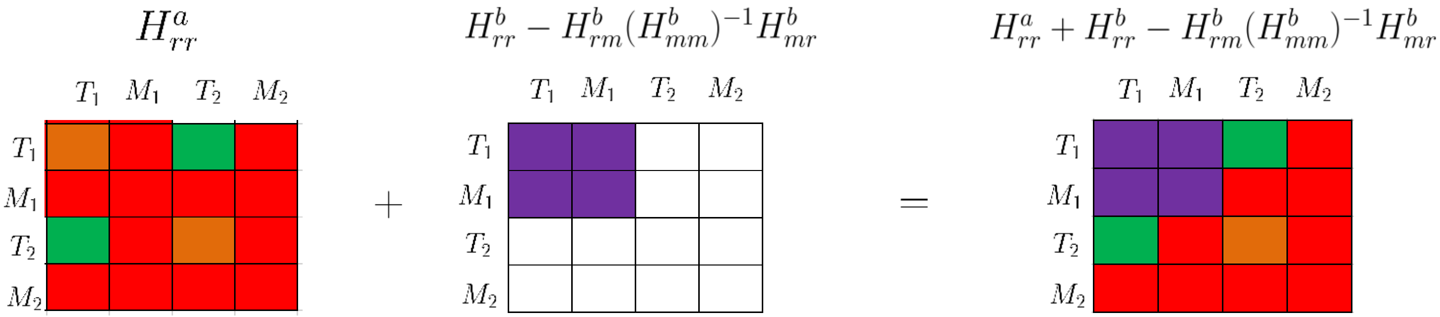
$$\left[H^b_{rr} - H^b_{rm}(H^b_{mm})^{-1}H^b_{mr}\right]\delta x_r = b^b_r - H^b_{rm}(H^b_{mm})^{-1}b^b_m$$

这一步形成的约束(上式)就叫先验因子，它包含了边缘化掉的量对剩余变量的约束关系。

2. 边缘化过程

1）移除老的帧

最终使用的是两步的叠加，Hessian矩阵叠加的可视化如下



对应的完整公式为

$$H_{rr}\delta x_r = b_r$$

其中

$$H_{rr} = H^a_{rr} + H^b_{rr} - H^b_{rm}(H^b_{mm})^{-1}H^b_{mr}$$

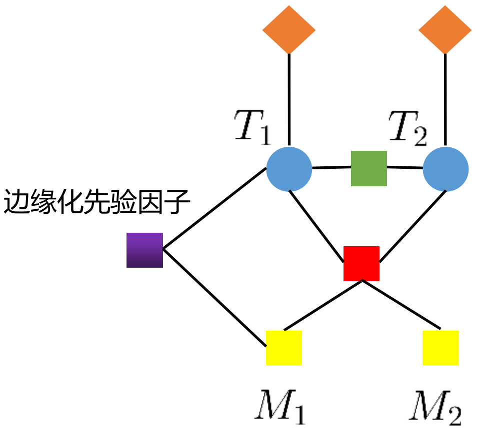
$$b_r = b^a_r + b^b_r - H^b_{rm}(H^b_{mm})^{-1}b^b_m$$

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：



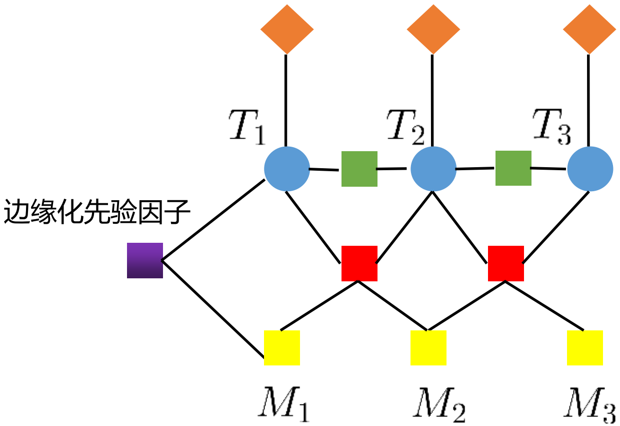
注意：边缘化先验因子只有在第一次边缘化之前是不存在的，完成第一次边缘化之后就一直存在，并且随着后续新的边缘化进行，其内容不断更新。

基于kitti的实现原理

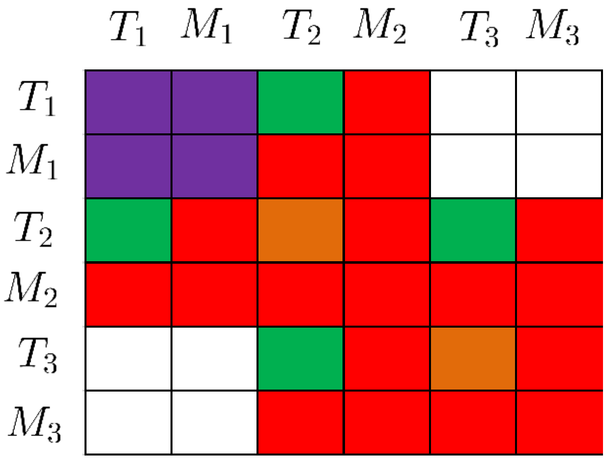
2. 边缘化过程

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：



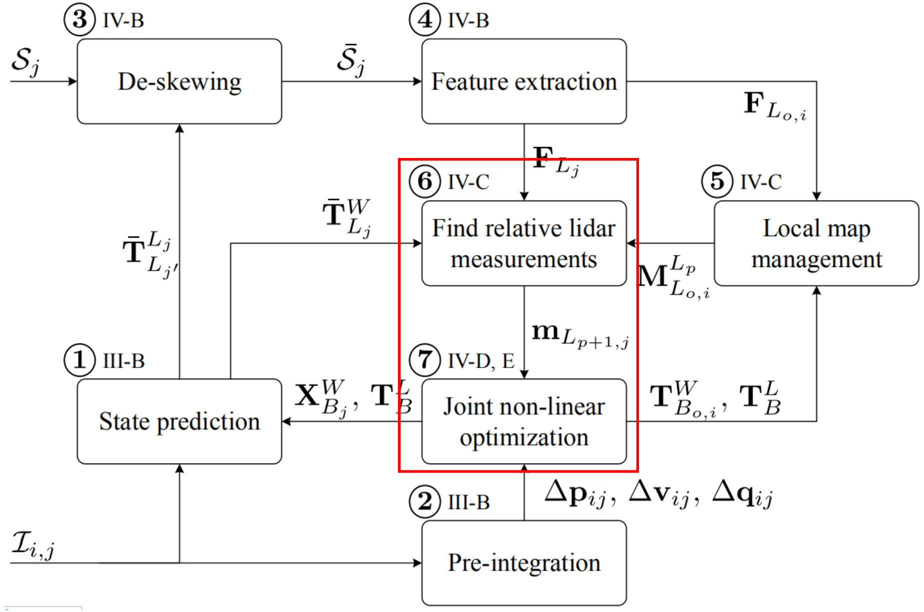
此后，随着定位过程的进行，便不断循环“边缘化老帧->添加新帧”的过程，从而维持窗口长度不变。

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

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

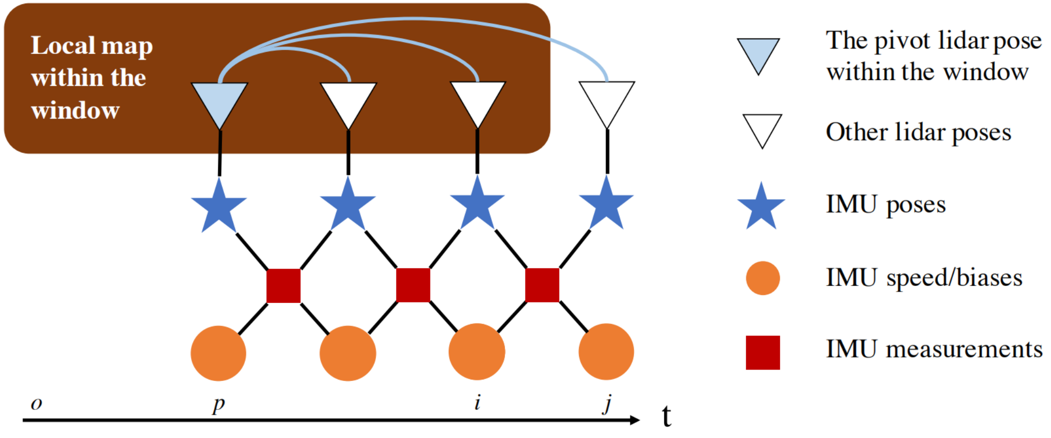
1. 核心思想

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($p < k <= j$)。

lio-mapping 介绍

1. 核心思想

其优化模型为

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

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；

$\mathbf{r}_{\mathcal{B}}\left(z^{\beta}_{\beta+1},\mathbf{X}\right)$ 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) 定义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 {…

  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

2) 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

3) 定义点云面特征因子

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，残差与第三讲一致，雅可比采用李代数推导。

lio-mapping 介绍

2. 具体流程

4) 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

5) 定义边缘化先验因子

```
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。

其详细内容待讲解边缘化实现时再展开。

lio-mapping 介绍

2. 具体流程

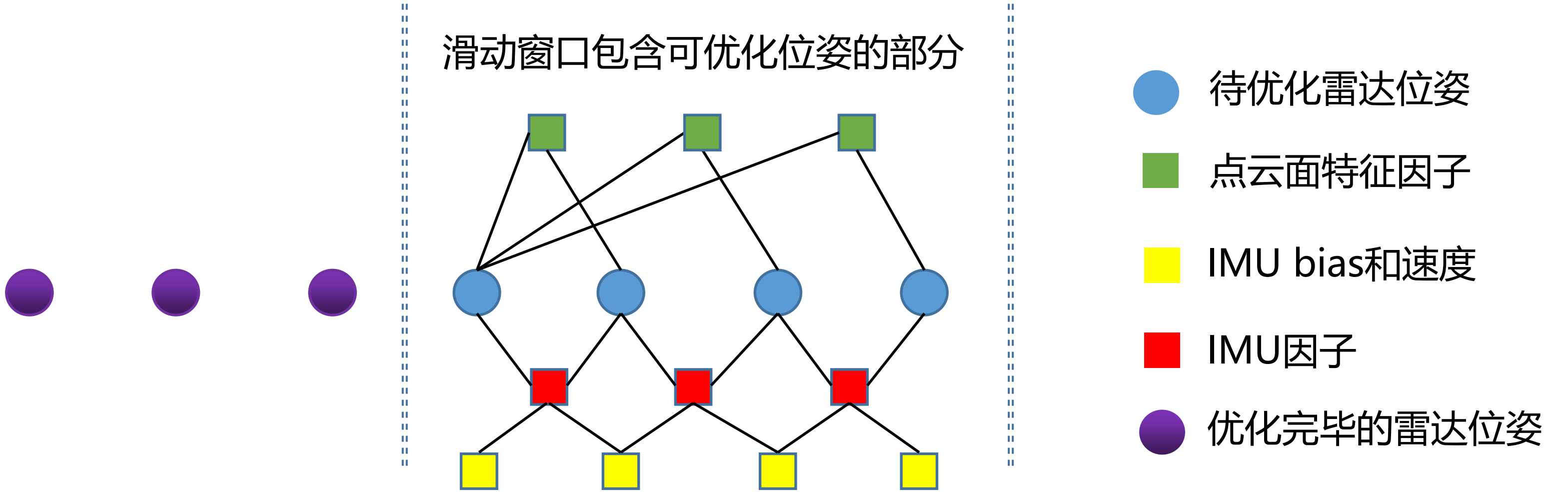
6) 添加边缘化先验因子

```
// region 边缘化
if (estimator_config_.marginalization_factor) {
  if (last_marginalization_info) {
    // construct new marginlization_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. 具体流程

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。

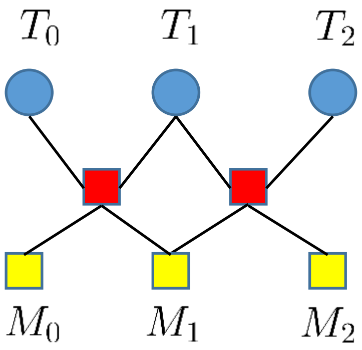
lio-mapping 介绍

2. 具体流程

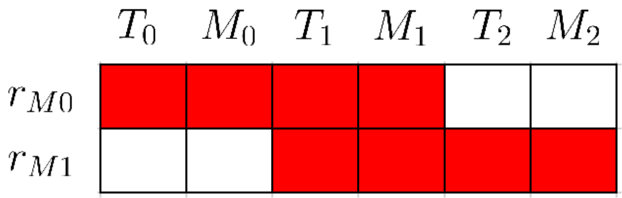
2.2. 滑窗模型

1）IMU预积分和优化变量的残差

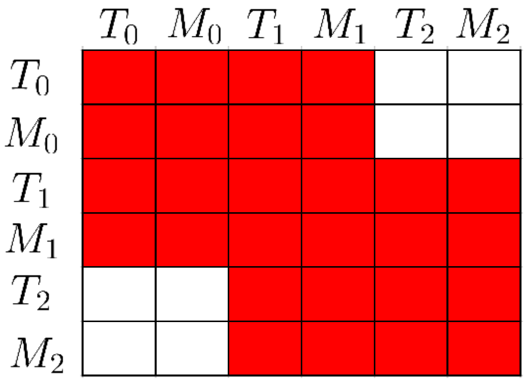
一个因子约束两个位姿，并约束两个时刻 IMU 的速度与 bias



残差关于优化变量的雅可比可视化如下



因此对应的Hessian矩阵的可视化为



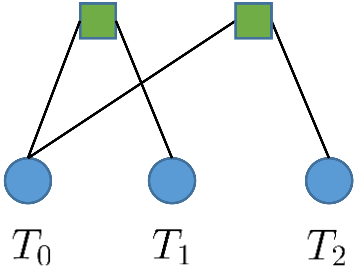
lio-mapping 介绍

2. 具体流程

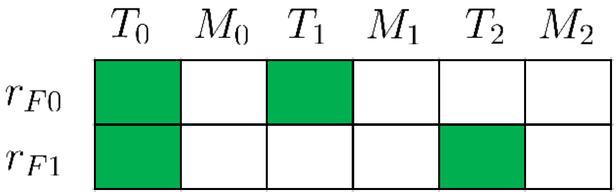
2.2. 滑窗模型

2）点云面特征对应的残差

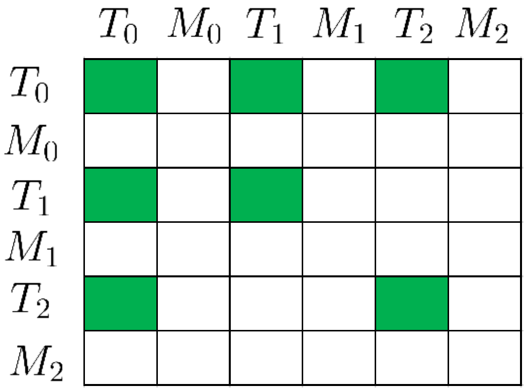
一个因子约束两个位姿



残差关于优化变量的雅可比可视化如下



因此对应的Hessian矩阵的可视化为

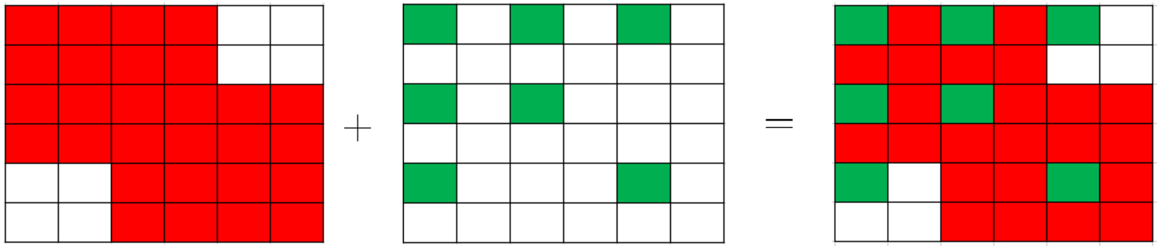


lio-mapping 介绍

2. 具体流程

2.2. 滑窗模型

4）完整模型



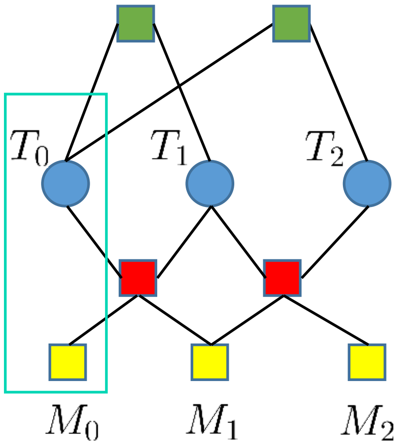
以上工程，就是前述代码中添加各类因子到模型的过程。

lio-mapping 介绍

2. 具体流程

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

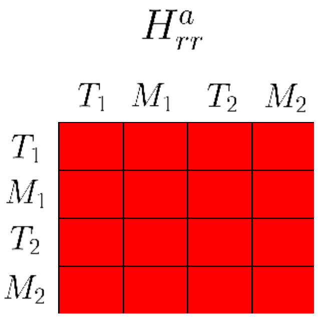
lio-mapping 介绍

2. 具体流程

2.3. 边缘化

2) 边缘化可视化

第一步：使用和要边缘化掉的量无关的因子，构建剩余变量对应的Hessian矩阵。



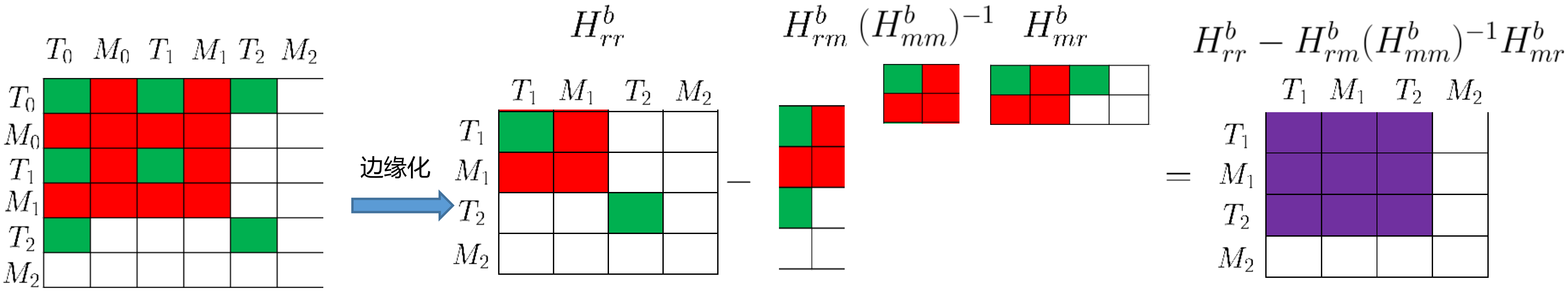
lio-mapping 介绍

2. 具体流程

2.3. 边缘化

2) 边缘化可视化

第二步：挑出和要边缘化掉的量相关的因子，构建待边缘化的Hessian矩阵，并使用舒尔补做边缘化。



lio-mapping 介绍

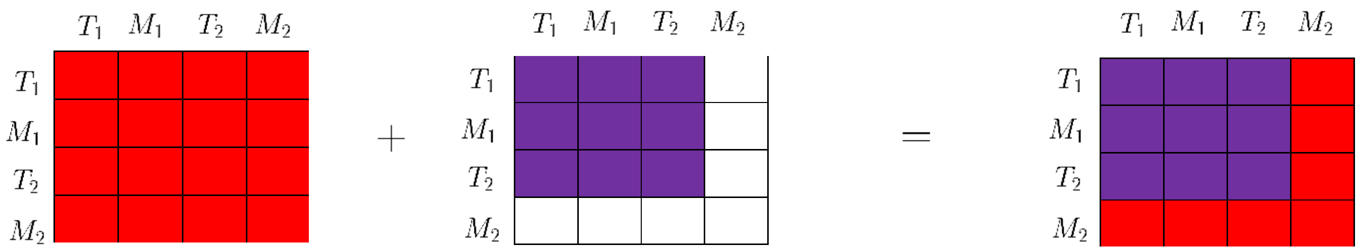
2. 具体流程

2.3. 边缘化

2) 边缘化可视化

对应的完整Hessian矩阵就是他们合在一起的结果。

$$H^a_{rr} \quad + \quad H^b_{rr} - H^b_{rm}(H^b_{mm})^{-1}H^b_{mr} \quad = \quad H^a_{rr} + H^b_{rr} - H^b_{rm}(H^b_{mm})^{-1}H^b_{mr}$$



lio-mapping 介绍

2. 具体流程

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

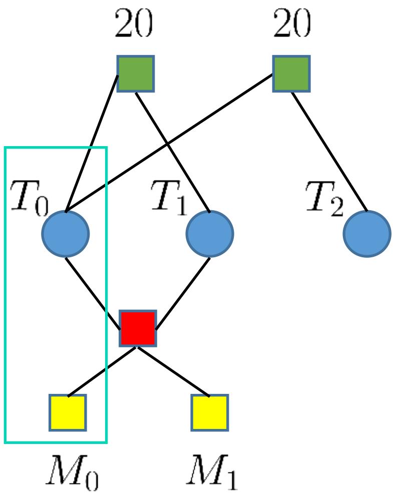
```
class MarginalizationInfo {
 public:
  ~MarginalizationInfo();
  int LocalSize(int size) const;
  int GlobalSize(int size) const;
  void AddResidualBlockInfo(ResidualBlockInfo *residual_block_info);
  void PreMarginalize();
  void Marginalize();
  std::vector<double *> GetParameterBlocks(std::unordered_map<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;
};
```

2. 具体流程

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：$T_0 \quad M_0 \quad T_1 \; M_1 \; T_2$

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. 具体流程

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. 具体流程

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. 具体流程

2.3. 边缘化

3) 边缘化实现　添加和面特征因子相关的ResidualBlockInfo

```cpp
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. 具体流程

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. 具体流程

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. 具体流程

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. 具体流程

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。

```cpp
//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. 具体流程

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. 具体流程

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。

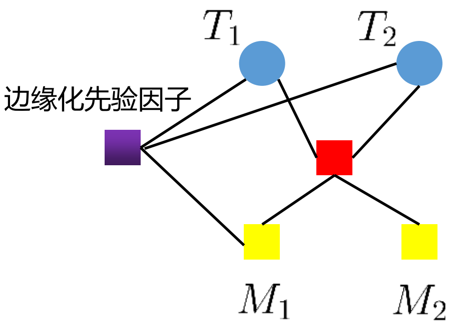
```cpp
if (last_marginalization_info) {
  vector<int> drop_set;
  for (int i = 0;
       i < static_cast<int>(last_marginalization_parameter_blocks.size());
       i++) {
    if (last_marginalization_parameter_blocks[i] == para_pose_[0] ||
        last_marginalization_parameter_blocks[i] == para_speed_bias_[0])
      drop_set.push_back(i);
  }
  // construct new marginlization_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. 具体流程

2.4. 添加新帧

边缘化之后，模型如下

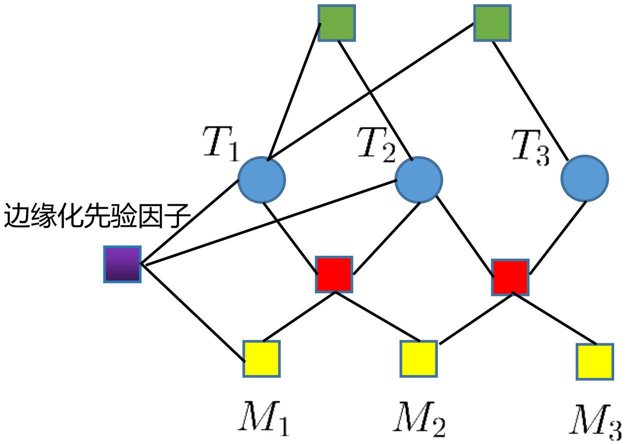


注意：由于面特征因子都是和第一帧(T_0)关联，当它边缘化掉之后，该因子便不存在，因此在加入新的帧的同时，还需要构建所有帧和当前第一帧(T_1)的面特征关联。

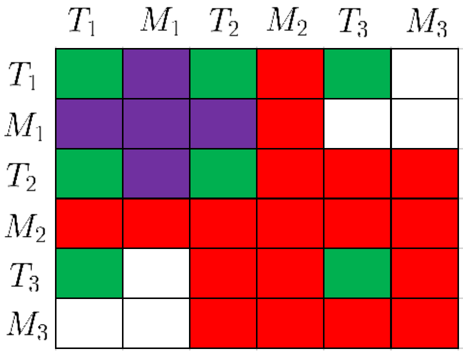
lio-mapping 介绍

2. 具体流程

添加新的帧以后，模型如下



Hessian矩阵可视化如下



此后不断循环该过程，便可以实现lio功能。

作业

按照课程讲述的模型，在提供的工程框架中，补全基于滑动窗口的融合定位方法的实现(整体思路本章第三小节已给出，代码实现可借鉴lio-mapping)，并分别与不加融合、EKF融合的效果做对比。

评价标准：

及格：补全代码，且功能正常。

良好：实现功能的基础上，性能在部分路段比EKF有改善。

优秀：由于基于滑窗的方法中，窗口长度对最终的性能有很大影响，请在良好的基础上，提供不同窗口长度下的融合结果，并对效果及原因做对比分析。

作业

附加题(不参与考核)：

基于地图定位时，滑窗中是否要加入帧间里程计相对位姿约束，这是一个有争议的话题。若各位愿意，可在工程基础上对比两种方案的不同，并分析造成差异的原因。

深蓝学院
shenlanxueyuan.com

感谢聆听!

Thanks for Listening