Jindong_Shi
2022-01-16

# 第 14 节课习题

## 1. 完成单目 Bundle Adjustment (BA)求解器部分代码

### 1) Problem::MakeHessian()中完成信息矩阵 H 的计算

在源文件 TestMonoBA.cpp 中创建了虚拟的 poseNums 个相机位姿和 featureNums 个路标点，我们假设每个相机都可以观测到所有路标点，那么我们总共就得到了 poseNums * featureNums 次观测。

我们将 6 维的相机姿态和 1 维的路标点逆深度作为待优化的状态量，构成顶点，这里我们将模拟的真实值设为这些顶点进行优化的初始值。

我们将每次观测作为一条边，其中每条边都连接顶点 Edge::verticies_，这里是三个顶点，分别是相机的初始位姿 vertexCams_vec [ 0 ]，相机第 j 个位姿 vertexCams_vec [ j ] 和 观测到的路标点(世界坐标系下)的逆深度值 verterxPoint。那么每条边对于其三个顶点状态量的雅可比，分别 1x6 维, 1x6 维 和 1x1 维。

为了构建信息矩阵 H，我们需要分别统计所有顶点，所有代表 Pose 的顶点，所有代表路标点逆深度的顶点的个数。我们根据所有顶点累计的参数维度创建信息矩阵 H。然后将所有顶点进行排序，来规定残差对它们的雅可比具体在 H 矩阵的哪个位置。简单来说，就是将代表 Pose 的顶点的雅可比放到 H 矩阵左上角，具体每个 Pose 顶点的先后按照创建该 Pose 顶点时的 Id 进行排序。然后将代表路标点逆深度的顶点的雅可比放到 H 矩阵的右下角，具体每个路标点逆深度顶点的先后也是按照创建逆深度顶点顶点时的 Id 排序。由于代表逆深度的顶点位于 Pose 顶点之后，所以需要将代表逆深度的顶点排序后的 Id 加上所有 Pose 顶点累计的参数维度。 (see function „void Problem::AddOrderingSLAM(…)“ and function „void Problem::SetOrdering()“)

接下来我们就可以根据每条边连接的顶点的 Id，以及顶点包含参数的维度，将每条边对其连接的顶点的参数的雅可比计算得到的小信息矩阵块 hessian 放到大的信息矩阵 H 内。

这里我们先计算了 H 矩阵上三角和对角内包含的小信息矩阵块。然后由于 H 矩阵的对称性，我们再补充下三角对应位置的小信息矩阵块为 hessian.transpose()，如下图。

```
    MatXX hessian = JtW * jacobian_j;
    // 所有的信息矩阵叠加起来
    // TODO:: home work. 完成 H index 的填写.
    // noalias()表示在矩阵计算的时候，不需要构造临时变量
    H.block( startRow: index_i, startCol: index_j,  blockRows: dim_i,  blockCols: dim_j).noalias() += hessian;
    if (j != i) {
        // 对称的下三角(只添加非对角矩阵块)
        // TODO:: home work. 完成 H index 的填写.
        H.block( startRow: index_j, startCol: index_i,  blockRows: dim_j,  blockCols: dim_i).noalias() += hessian.transpose();
    }
}
// 注意这样计算b的话，成员变量b_就是是带负号的: H * delta_x = b_
b.segment( start: index_i,  n: dim_i).noalias() -= JtW * edge.second->Residual();   // Jt: 维度6*1  W: 维度1*1  r: 维度1*1
```

需要注意的是，我们在函数 void Problem::MakeHessian()和函数 void Problem::UpdateStates()内都更新了先验残差，这里使用一阶泰勒展开来近似:

```
// 注意: 随着迭代推进，变量被不断优化，先验残差需要跟随变化，否则，求解系统可能崩溃
if (err_prior_.rows() > 0) {
    b_prior_ -= H_prior_ * delta_x_.head(ordering_poses_);   // update the error_prior  第5讲公式(7)
}
```

**如何更新先验残差？**

- 目的：虽然先验信息矩阵固定不变，但随着迭代的推进，变量被不断优化，先验残差需要跟随变化。否则，求解系统可能奔溃。
- 方法：先验残差的变化可以使用一阶泰勒近似。

$$
\begin{aligned}
\mathbf{b}_{\mathrm{p}}^{\prime} &= \mathbf{b}_{\mathrm{p}} + \frac{\partial \mathbf{b}_{\mathrm{p}}}{\partial \mathbf{x}_{\mathrm{p}}} \delta \mathbf{x}_{\mathrm{p}} \\
&= \mathbf{b}_{\mathrm{p}} + \frac{\partial \left(-\mathbf{J}^{\top} \boldsymbol{\Sigma}^{-1} \mathbf{r}\right)}{\partial \mathbf{x}_{\mathrm{p}}} \delta \mathbf{x}_{\mathrm{p}} \\
&= \mathbf{b}_{\mathrm{p}} - \boldsymbol{\Lambda}_{\mathbf{p}} \delta \mathbf{x}_{\mathrm{p}}
\end{aligned} \tag{7}
$$

### 2) Problem::SolveLinearSystem()中完成 SLAM 问题的求解

在函数 void Problem::SolveLinearSystem()内，我们规定非 SLAM 问题采用直接求逆的方式来解线性方程，对于 SLAM 问题，由于其信息矩阵维度高，具有稀疏性，我们采用舒尔补的方式将路标点给舒尔补掉，从而将高维度稀疏的线性方程转化为低维度的稠密线性矩阵进行求解，来减小运算量。课件中已经给出了具体舒尔补的计算方式，如下:

$$\begin{bmatrix} \mathbf{H}_{\mathrm{pp}} & \mathbf{H}_{\mathrm{pl}} \\ \mathbf{H}_{\mathrm{lp}} & \mathbf{H}_{\mathrm{ll}} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_{\mathrm{p}}^{*} \\ \Delta \mathbf{x}_{\mathrm{l}}^{*} \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_{\mathrm{p}} \\ -\mathbf{b}_{\mathrm{l}} \end{bmatrix} \tag{4}$$

可以利用舒尔补操作，使上式中信息矩阵变成下三角，从而得到：

$$\left(\mathbf{H}_{\mathrm{pp}} - \mathbf{H}_{\mathrm{pl}} \mathbf{H}_{\mathrm{ll}}^{-1} \mathbf{H}_{\mathrm{pl}}^{\top}\right) \Delta \mathbf{x}_{\mathrm{p}}^{*} = -\mathbf{b}_{\mathrm{p}} + \mathbf{H}_{\mathrm{pl}} \mathbf{H}_{\mathrm{ll}}^{-1} \mathbf{b}_{\mathrm{l}} \tag{5}$$

求得 $\Delta \mathbf{x}_{\mathrm{p}}^{*}$ 后，再计算 $\Delta \mathbf{x}_{\mathrm{l}}^{*}$：

$$\mathbf{H}_{\mathrm{ll}} \Delta \mathbf{x}_{\mathrm{l}}^{*} = -\mathbf{b}_{\mathrm{l}} - \mathbf{H}_{\mathrm{pl}}^{\top} \Delta \mathbf{x}_{\mathrm{p}}^{*} \tag{6}$$

由于之前在 Problem::MakeHessian()函数中我们已经对信息矩阵内对应的顶点状态量进行过排序，Pose 顶点在前，逆深度代表的路标点顶点在后，所以现在我们只需要根据 Pose 顶点维度和逆深度顶点维度，从 H 矩阵取出 Hpp, Hpm, Hmp, Hmm。从而根据上图中的式(5)计算该小维度正规方程的信息矩阵 H_pp_schur_。由于我们采用的 LM 算法，所以这里需要将更新的阻尼因子加到 H_pp_schur_上，来解这个小规模的正规方程 H_pp_schur_ * delta_x_pp = b_pp_schur_。这里采用了 PCG 进行迭代求解，当然也可以用更一般的方法来解这个线性方程。

第二步是根据解出的 delta_x_pp 解出 delta_x_ll。

(需要注意的是这里的向量 b_, bpp, bmm 是带负号的)

```
// SLAM 问题采用舒尔补的计算方式
// step1: schur marginalization --> Hpp, bpp
int reserve_size = ordering_poses_;
int marg_size = ordering_landmarks_;

// TODO:: home work. 完成矩阵块取值，Hmm，Hpm，Hmp，bpp，bmm
MatXX Hmm = Hessian_.block( startRow: reserve_size, startCol: reserve_size,  blockRows: marg_size,  blockCols: marg_size);
MatXX Hpm = Hessian_.block( startRow: 0, startCol: reserve_size,  blockRows: reserve_size,  blockCols: marg_size);
MatXX Hmp = Hessian_.block( startRow: reserve_size, startCol: 0,  blockRows: marg_size,  blockCols: reserve_size);
// 注意这里b_是带负号的，所以bpp和bmm也是带负号的
VecX bpp = b_.segment( start: 0, n: reserve_size);
VecX bmm = b_.segment( start: reserve_size, n: marg_size);

// Hmm 是对角线矩阵，它的求逆可以直接为对角线块分别求逆，如果是逆深度，对角线块为1维的，则直接为对角线的倒数，这里可以加速
MatXX Hmm_inv( x: MatXX::Zero( rows: marg_size,  cols: marg_size));
for (auto landmarkVertex :pair<…>  : idx_landmark_vertices_) {
    int idx = landmarkVertex.second->OrderingId() - reserve_size;
    int size = landmarkVertex.second->LocalDimension();
    Hmm_inv.block( startRow: idx,  startCol: idx,  blockRows: size,  blockCols: size) = Hmm.block( startRow: idx,  startCol: idx,  blockRows: size,  blockCols: size).inverse()
}

// TODO:: home work. 完成舒尔补 Hpp, bpp 代码
MatXX tempH = Hpm * Hmm_inv;
H_pp_schur_ = Hessian_.block( startRow: 0, startCol: 0, blockRows: reserve_size, blockCols: reserve_size) - tempH * Hmp;
b_pp_schur_ = bpp - tempH * bmm;
```

```
// step2: solve Hpp * delta_x = bpp
VecX delta_x_pp( x: VecX::Zero( size: reserve_size));
// PCG Solver
for (ulong i = 0; i < ordering_poses_; ++i) {
    // 只对舒尔补之后得到的矩阵H_pp_schur_加上Lambda，因为我们只解这一个关于delta_x_pp的正规方程，不包括delta_x_ll
    H_pp_schur_( row: i,  col: i) += currentLambda_;
}

int n = H_pp_schur_.rows() * 2;                        // 迭代次数
delta_x_pp = PCGSolver( A: H_pp_schur_,  b: b_pp_schur_,  maxiter: n);  // 哈哈，小规模问题，搞 pcg 花里胡哨
delta_x_.head( n: reserve_size) = delta_x_pp;
//        std::cout << delta_x_pp.transpose() << std::endl;

// TODO:: home work. step3: solve landmark
VecX delta_x_ll( x: marg_size);
delta_x_ll = Hmm_inv * (bmm - Hmp * delta_x_pp);
delta_x_.tail( n: marg_size) = delta_x_ll;
```

我们设定 featureNums = 20，poseNums = 3，程序运行结果如下:

```
/home/jindong/VIO/Exercise/L5/hw_course5_new/cmake-build-debug/app/testMonoBA
0 order: 0
1 order: 6
2 order: 12

 ordered_landmark_vertices_ size : 20
iter: 0 , chi= 5.35099 , Lambda= 0.00597396
iter: 1 , chi= 0.0289048 , Lambda= 0.00199132
iter: 2 , chi= 0.000109162 , Lambda= 0.000663774
problem solve cost: 24.4394 ms
   makeHessian cost: 20.222 ms
Compare MonoBA results after opt...
after opt, point 0 : gt 0.220938 ,noise 0.227057 ,opt 0.220992
after opt, point 1 : gt 0.234336 ,noise 0.314411 ,opt 0.234854
after opt, point 2 : gt 0.142336 ,noise 0.129703 ,opt 0.142666
after opt, point 3 : gt 0.214315 ,noise 0.278486 ,opt 0.214502
after opt, point 4 : gt 0.130629 ,noise 0.130064 ,opt 0.130562
after opt, point 5 : gt 0.191377 ,noise 0.167501 ,opt 0.191892
after opt, point 6 : gt 0.166836 ,noise 0.165906 ,opt 0.167247
after opt, point 7 : gt 0.201627 ,noise 0.225581 ,opt 0.202172
after opt, point 8 : gt 0.167953 ,noise 0.155846 ,opt 0.168029
after opt, point 9 : gt 0.21891 ,noise 0.209697 ,opt 0.219314
after opt, point 10 : gt 0.205719 ,noise 0.14315 ,opt 0.205995
after opt, point 11 : gt 0.127916 ,noise 0.122109 ,opt 0.127908
after opt, point 12 : gt 0.167904 ,noise 0.143334 ,opt 0.168228
after opt, point 13 : gt 0.216712 ,noise 0.18526 ,opt 0.216866
after opt, point 14 : gt 0.180009 ,noise 0.184249 ,opt 0.180036
after opt, point 15 : gt 0.226935 ,noise 0.245716 ,opt 0.227491
after opt, point 16 : gt 0.157432 ,noise 0.176529 ,opt 0.157589
after opt, point 17 : gt 0.182452 ,noise 0.14729 ,opt 0.182444
after opt, point 18 : gt 0.155701 ,noise 0.182258 ,opt 0.155769
after opt, point 19 : gt 0.14646 ,noise 0.240649 ,opt 0.14677
------------ pose translation ----------------
translation after opt: 0 :-0.00047801  0.00115904 0.000366507 || gt: 0 0 0
translation after opt: 1 :-1.06959  4.00018 0.863877 || gt:  -1.0718        4 0.866025
translation after opt: 2 :-4.00232  6.92678 0.867244 || gt:       -4   6.9282 0.866025
```

我们发现由于该单目 BA 问题中信息矩阵存在自由度，导致优化问题解不唯一，所以这里优化出来的第一个位姿与实际位姿有区别，我们发现第一个位姿的位置向量与实际位置向量[0; 0; 0]并不重合。我们通过将前两个相机位姿顶点对应的雅可比置为 0, 将前两个位姿 fix 住，使得前两个位姿与实际位姿对齐。

- 设定对应雅克比矩阵为 $0$，意味着残差等于 $0$. 求解方程为 $(\mathbf{0}+\lambda \mathbf{I})\,\Delta \mathbf{x}=\mathbf{0}$，只能 $\Delta x=0$。

```
// 所有 Pose
vector<shared_ptr<VertexPose> > vertexCams_vec;
for (size_t i = 0; i < cameras.size(); ++i) {
    shared_ptr<VertexPose> vertexCam( p: new VertexPose());
    Eigen::VectorXd pose( x: 7);
    pose << cameras[i].twc, cameras[i].qwc.x(), cameras[i].qwc.y(), cameras[i].qwc.z(), cameras[i].qwc.w();
    // 这里设定待优化相机位姿的初始值为真实值
    vertexCam->SetParameters( params: pose);

    if(i < 2)
        vertexCam->SetFixed();

    problem.AddVertex( vertex: vertexCam);
    vertexCams_vec.push_back(vertexCam);
}
```

```
for (size_t i = 0; i < verticies.size(); ++i) {
    auto v_i :shared_ptr<Vertex>  = verticies[i];
    if (v_i->IsFixed()) continue;    // Hessian 里不需要添加它的信息，也就是它的雅克比为 0

    auto jacobian_i :Matrix<...>  = jacobians[i];
    ulong index_i = v_i->OrderingId();
    ulong dim_i = v_i->LocalDimension();

    MatXX JtW = jacobian_i.transpose() * edge.second->Information();
    for (size_t j = i; j < verticies.size(); ++j) {
        auto v_j :shared_ptr<Vertex>  = verticies[j];

        if (v_j->IsFixed()) continue;

        auto jacobian_j :Matrix<...>  = jacobians[j];
        ulong index_j = v_j->OrderingId();
        ulong dim_j = v_j->LocalDimension();
```

运行结果如下，前两个相机位姿与实际位姿对齐:

```
/home/jindong/VIO/Exercise/L5/hw_course5_new/cmake-build-debug/app/testMonoBA
0 order: 0
1 order: 6
2 order: 12

 ordered_landmark_vertices_ size : 20
iter: 0 , chi= 5.35099 , Lambda= 0.00597396
iter: 1 , chi= 0.0282599 , Lambda= 0.00199132
iter: 2 , chi= 0.000117497 , Lambda= 0.000663774
problem solve cost: 18.9966 ms
   makeHessian cost: 15.8805 ms
Compare MonoBA results after opt...
after opt, point 0 : gt 0.220938 ,noise 0.227057 ,opt 0.220909
after opt, point 1 : gt 0.234336 ,noise 0.314411 ,opt 0.234374
after opt, point 2 : gt 0.142336 ,noise 0.129703 ,opt 0.142353
after opt, point 3 : gt 0.214315 ,noise 0.278486 ,opt 0.214501
after opt, point 4 : gt 0.130629 ,noise 0.130064 ,opt 0.130511
after opt, point 5 : gt 0.191377 ,noise 0.167501 ,opt 0.191539
after opt, point 6 : gt 0.166836 ,noise 0.165906 ,opt 0.166965
after opt, point 7 : gt 0.201627 ,noise 0.225581 ,opt 0.201859
after opt, point 8 : gt 0.167953 ,noise 0.155846 ,opt 0.167965
after opt, point 9 : gt 0.21891 ,noise 0.209697 ,opt 0.218834
after opt, point 10 : gt 0.205719 ,noise 0.14315 ,opt 0.205683
after opt, point 11 : gt 0.127916 ,noise 0.122109 ,opt 0.127751
after opt, point 12 : gt 0.167904 ,noise 0.143334 ,opt 0.167924
after opt, point 13 : gt 0.216712 ,noise 0.18526 ,opt 0.216885
after opt, point 14 : gt 0.180009 ,noise 0.184249 ,opt 0.179961
after opt, point 15 : gt 0.226935 ,noise 0.245716 ,opt 0.227114
after opt, point 16 : gt 0.157432 ,noise 0.176529 ,opt 0.157529
after opt, point 17 : gt 0.182452 ,noise 0.14729 ,opt 0.1823
after opt, point 18 : gt 0.155701 ,noise 0.182258 ,opt 0.155627
after opt, point 19 : gt 0.14646 ,noise 0.240649 ,opt 0.146533
------------ pose translation ----------------
translation after opt: 0 :0 0 0 || gt: 0 0 0
translation after opt: 1 : -1.0718        4 0.866025 || gt:  -1.0718        4 0.866025
translation after opt: 2 :-3.99917  6.92852 0.859878 || gt:       -4   6.9282 0.866025
```

**3) Problem::TestMarginalize()中完成滑动窗口算法测试函数**

我们通过模拟第 4 讲中房间温度的例子，练习如何将某个状态量边缘化掉，构建先验矩阵 H_prior。

```
// Add marg test
int idx = 1;            // marg 中间那个变量
int dim = 1;            // marg 变量的维度
int reserve_size = 3;   // 总共变量的维度
double delta1 = 0.1 * 0.1;
double delta2 = 0.2 * 0.2;
double delta3 = 0.3 * 0.3;

int cols = 3;
MatXX H_marg(MatXX::Zero(cols, cols));
H_marg << 1./delta1, -1./delta1, 0,
       -1./delta1, 1./delta1 + 1./delta2 + 1./delta3, -1./delta3,
       0.,  -1./delta3, 1/delta3;
std::cout << "---------- TEST Marg: before marg------------"<< std::endl;
std::cout << H_marg << std::endl;
```

首先我们需要将原有待 marg 的信息矩阵 H_marg 进行移动，将需要保留的状态量对应的信息矩阵块放到 H_marg 的左上角，将需要边缘化的状态量对应的信息矩阵块放到 H_marg 右下角，如下:

```
// TODO:: home work. 将变量移动到右下角
/// 准备工作: move the marg pose to the Hmm bottown right
// 将 row i 移动矩阵最下面
Eigen::MatrixXd temp_rows = H_marg.block( startRow: idx,  startCol: 0,  blockRows: dim,  blockCols: reserve_size);
Eigen::MatrixXd temp_botRows = H_marg.block( startRow: idx + dim,  startCol: 0,  blockRows: reserve_size - idx - dim,  blockCols: reserve_size);
H_marg.block( startRow: idx, startCol: 0, blockRows: reserve_size - idx - dim, blockCols: reserve_size) = temp_botRows;
H_marg.block( startRow: reserve_size - dim, startCol: 0, blockRows: dim, blockCols: reserve_size) = temp_rows;

// 将 col i 移动矩阵最右边
Eigen::MatrixXd temp_cols = H_marg.block( startRow: 0,  startCol: idx,  blockRows: reserve_size,  blockCols: dim);
Eigen::MatrixXd temp_rightCols = H_marg.block( startRow: 0,  startCol: idx + dim,  blockRows: reserve_size,  blockCols: reserve_size - idx - dim);
H_marg.block( startRow: 0,  startCol: idx,  blockRows: reserve_size,  blockCols: reserve_size - idx - dim) = temp_rightCols;
H_marg.block( startRow: 0,  startCol: reserve_size - dim,  blockRows: reserve_size,  blockCols: dim) = temp_cols;

std::cout << "---------- TEST Marg: 将变量移动到右下角------------"<< std::endl;
std::cout<< H_marg <<std::endl;
```

然后根据提取出来的信息矩阵块 Arr, Arm, Amr, Amm 通过舒尔补计算将某个状态量边缘化之后的信息矩阵 H_prior。

```
// TODO:: home work. 完成舒尔补操作
Eigen::MatrixXd Arm = H_marg.block( startRow: 0, startCol: n2, blockRows: n2, blockCols: m2);
Eigen::MatrixXd Amr = H_marg.block( startRow: n2, startCol: 0, blockRows: m2, blockCols: n2);
Eigen::MatrixXd Arr = H_marg.block( startRow: 0, startCol: 0, blockRows: n2, blockCols: n2);

Eigen::MatrixXd tempB = Arm * Amm_inv;
Eigen::MatrixXd H_prior = Arr - tempB * Amr;
```

结果如下:

```
---------- TEST Marg: before marg------------
     100     -100        0
    -100  136.111 -11.1111
       0 -11.1111  11.1111
---------- TEST Marg: 将变量移动到右下角------------
     100        0     -100
       0  11.1111 -11.1111
    -100 -11.1111  136.111
---------- TEST Marg: after marg------------
 26.5306 -8.16327
-8.16327  10.2041

Process finished with exit code 0
```

## 2. 总结论文 [1] 中优化过程中处理 H 自由度的不同操作方式

**1) 引入:**

单目 VI 系统中的状态估计一般包含 4 自由度不可观，分别是 3 自由度位置和 yaw 角，论文[1]中将这些不可观的状态量称为“gauge freedom“。 通过视觉测量和 IMU 测量，可以构建 VI 系统的非线性最小二乘问题(NLLS)。其中我们最小化一个由视觉重投影误差和 IMU 预积分误差构成的目标函数:

$$J(\boldsymbol{\theta}) \doteq \underbrace{\|\mathbf{r}^V(\boldsymbol{\theta})\|^2_{\Sigma_V}}_{\text{Visual}} + \underbrace{\|\mathbf{r}^I(\boldsymbol{\theta})\|^2_{\Sigma_I}}_{\text{Inertial}},$$

VI 系统中我们需要估计的参数有位置 p，姿态 R，速度 v 和路标点坐标 X:

$$\boldsymbol{\theta} \doteq \{\mathbf{p}_i, \mathbf{R}_i, \mathbf{v}_i, \mathbf{X}_j\}$$

其中需要估计的状态量的维度 n = 9N + 3K, 其中 N 为相机位姿个数，K 为路标点个数。那么可观的状态量的维度为 n-4.
我们知道视觉重投影误差为路标点重投影的值减去路标的观测值，而路标点重投影的值由路标点世界系下的坐标先后乘以外参数矩阵和内参数矩阵得到。其中造成 4 自由度不可观的原因，就是因为外参数矩阵

$$g \doteq \begin{pmatrix} \mathbf{R}_z & \mathbf{t} \\ 0 & 1 \end{pmatrix}$$

具有 4 个自由度，即平移向量 t 和 yaw 角不可观，意味着我们在世界坐标系下的 p，R，v 和 X 可经由任意的平移向量 t 和 yaw 角变换到相机坐标系下某个固定值，

$$J(\boldsymbol{\theta}) = J(g(\boldsymbol{\theta}))$$

而不会对我们的优化问题造成任何影响，这就导致了优化问题的解不唯一。
另一方面，该优化问题的 Hessian 矩阵由于 gauge freedom 的存在而产生奇异性，导致单纯的高斯牛顿法无法对优化问题进行求解。为了解该奇异矩阵构成的线性方程，我们可以使用伪逆或者带阻尼的优化方法，比如 LM 算法。我们也可以在优化问题中加入额外的约束，即指定不可观量使得方程有唯一解。

**2) 三种处理信息矩阵 H 自由度的方法**

针对信息矩阵 H 存在自由度的问题，该论文中总结了三种处理办法, 分别是加入固定点法(gauge fixation approach), 加入超强先验法(gauge prior approach) 和 加阻尼因子或使用伪逆法(free gauge approach)。

A. 加入固定点法(gauge fixation approach)

我们既要固定位置 p, 又要固定 yaw 角。通常的做法是固定住第一个相机的位置$\mathbf{p}_0$为初始位置$\mathbf{p}_0^0$，并且让将 yaw 角的更新置为 0，从而固定住 yaw 角。

$$\mathbf{p}_0 = \mathbf{p}_0^0, \qquad \Delta\boldsymbol{\phi}_{0z} \doteq \mathbf{e}_z^\top \Delta\boldsymbol{\phi}_0 = 0, \qquad \mathbf{R}_0 = \mathrm{Exp}(\Delta\boldsymbol{\phi}_0)\mathbf{R}_0^0$$

我们也可以选择将对应的雅可比置为 0，从而固定位置 $\mathbf{p}_0$ 和 $\Delta\boldsymbol{\phi}_0$

$$\mathbf{J}_{\mathbf{p}_0} = \mathbf{0},\ \mathbf{J}_{\Delta\boldsymbol{\phi}_{0z}} = \mathbf{0}$$

B. 加入超强先验法(gauge prior approach)

我们可以将第一个相机的位置$\mathbf{p}_0$设为初始位置$\mathbf{p}_0^0$，将$\Delta\boldsymbol{\phi}_0$置为 0，通过额外引入一个超强权重 ($\omega^P \to \infty$)的先验位姿残差:

$$\|\mathbf{r}_0^P\|_{\Sigma_0^P}^2, \quad \text{where} \quad \mathbf{r}_0^P(\boldsymbol{\theta}) \doteq (\mathbf{p}_0 - \mathbf{p}_0^0,\ \Delta\boldsymbol{\phi}_{0z})$$

这里先验权重 $\omega^P$ 即先验协方差 $\Sigma_0^P$ 的倒数。

这样一来先验位姿残差项$r_0^P$总是被优先优化为 0，使得优化得到的相机位置$\mathbf{p}_0$总是接近初始位置$\mathbf{p}_0^0$，$\Delta\boldsymbol{\phi}_0$总是为 0。

论文中比较了不同先验权重对优化结果的影响，如下图:

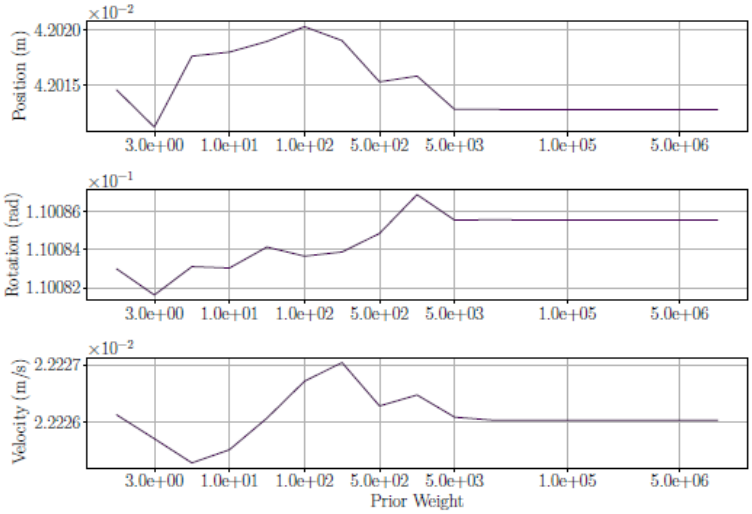



Fig. 4: RMSE in position, orientation and velocity for different prior weights

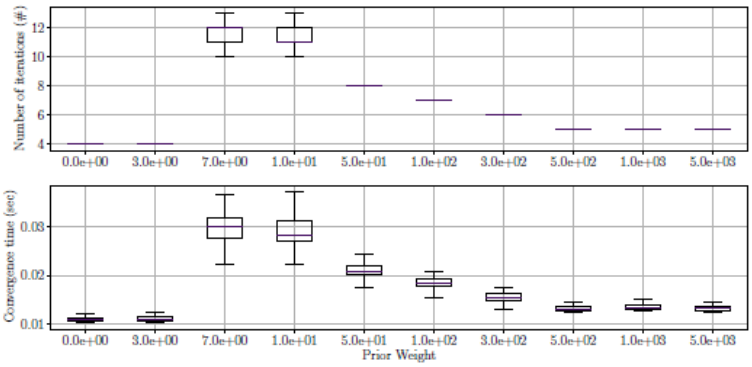



Fig. 5: Number of iterations and computing time for different prior weights.

论文作者发现估计误差 RMSE 一开始随着先验权重增加而减小(e.g. 50 - 500), 随后随着先验权重增加而增加，最后 RMSE 固定在一个稳定值。不同先验权重下的计算时间和迭代步数也有相似的变化。由于先验权重的选择对估计精度不产生太大影响，而且太大的先验权重会造成优化的不稳定，所以论文中暂时将先验权重设为$10^5$。

C. 加阻尼因子或使用伪逆法(free gauge approach)

为了解决信息矩阵 H 的奇异性，我们可以在解正规方程的时候使用伪逆来解方程，或者在信息矩阵上面加入一些阻尼(比如 LM 算法)，来保证 H 的正定性。当然课上也提到，往信息矩阵上加入阻尼因子虽然解决了奇异性问题，但是由于零空间发生变化，会使得估计出来的轨迹发生整体的漂移，可参考 VINS-MONO 的做法，将第一个相机的位姿点强行移动到初始位置，后面的位姿点也会跟着移过去。

**3) 实验效果**

论文中模拟了三种轨迹方式(分别形如三角函数，反三角函数和矩形)，以及两种不同的路标点位置设置(在轨迹附近随机出现或者在某些平面上随机出现)。通过 B 样条插值法计算得到加速度和角速度后加上噪声，作为 IMU 的测量值。对于每一种模拟设置情况选取多个关键帧，将路标点投影到相机模型上，并加入噪声，来得到相机测量值。通过使用 Ceres Solver 来解这个优化问题，并将解得的优化状态量和实际真实状态量做比较，计算出每种处理 H 自由度方法在每一种模拟设置上状态估计的 RMSE。最后论文比较了三种方法的精度和计算时间, 得到下图结果，(其中 Gauge prior 和 Gauge fixation 计算精度十分接近，故未展示):

TABLE II: RMSE on different trajectories and 3D points configurations. The smallest errors (e.g., p gauge fixation vs. p free gauge) are highlighted.

| Configuration | Gauge fixation | | | Free gauge | | |
|---|---|---|---|---|---|---|
| | p | $\phi$ | v | p | $\phi$ | v |
| *sine plane* | 0.04141 | 0.1084 | **0.02182** | 0.04141 | 0.1084 | 0.02183 |
| *arc plane* | **0.02328** | 0.6987 | 0.01303 | 0.02329 | 0.6987 | 0.01303 |
| *rec plane* | **0.01772** | 0.1668 | 0.01496 | 0.01774 | 0.1668 | **0.01495** |
| *sine random* | 0.03932 | 0.0885 | 0.01902 | **0.03908** | **0.0874** | **0.01886** |
| *arc random* | 0.02680 | 0.6895 | 0.01167 | **0.02678** | 0.6895 | **0.01166** |
| *rec random* | **0.02218** | 0.1330 | 0.009882 | 0.02220 | 0.1330 | **0.009881** |

Position, rotation and velocity RMSE are measured in m, deg and m/s, respectively.

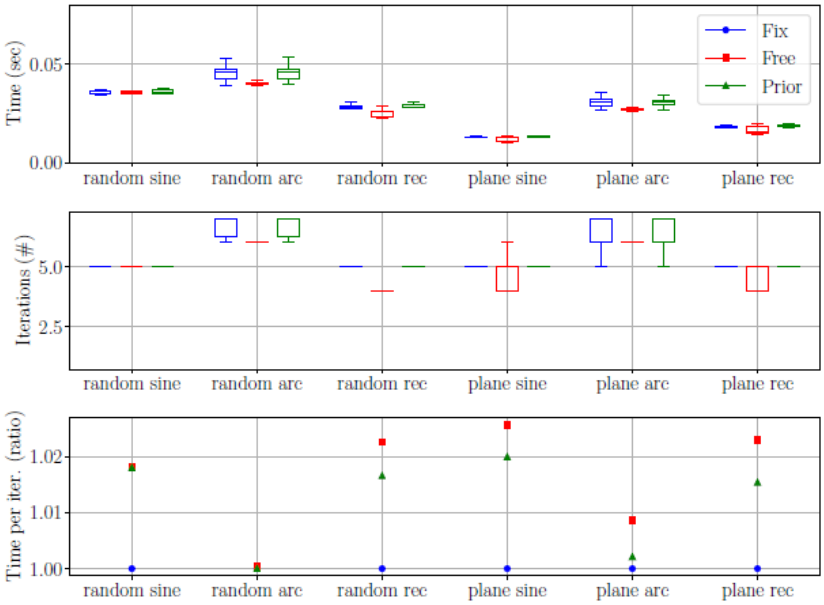



Fig. 7: Number of iterations, total convergence time and time per iteration for all configurations. The time per iteration is the ratio with respect to the gauge fixation approach (in blue), which takes least time per iteration.

从上图我们发现，三种方法对于估计的精度并无太大影响，都比较相似，对于不同状态量的估计精度都有好有坏。计算时间上来看，free gauge 比其他两种方法更快，而 gauge fixation 单步计算最快，因为它解的正规方程中 H 矩阵是 n-4 维的，其他都是 n 维的。

**4) 结论**

a. 三种方法计算精度十分类似。
b. 在 gauge prior 方法中我们需要选择一个最佳的先验权重来获取较优的计算时间。
c. 选择某种先验权重的情况下，Gauge prior 和 Gauge fixation 是两种精度和计算量上十分接近的方法。
d. Free gauge 明显比其他两种方法计算更快，因为它更快收敛。
e. Free gauge 相比于其他两种方法具有更普遍的适用性，不仅仅适用于 VI 系统，它不要求对旋转矩阵做特殊处理。

### 3. 在代码中给第一帧和第二帧添加 prior 约束，并比较为 prior 设定不同权重时，BA 求解的收敛精度和速度

第二题论文中已经介绍了 gauge_prior 的方法，这里我们可以通过为第一帧和第二帧引入两个超强权重 ($\omega^P \to \infty$)的先验位姿残差项，为第一帧和第二帧添加 prior 约束。问题在于这时我们在 problem 中会引入一种新的代表先验位姿残差的边，我们需要构建其对连接的状态量的雅可比，以及信息矩阵，从而进行后续的优化。很遗憾我并不知道如何求其雅可比，但是幸运的是样例代码中已经定义了这种代表先验位姿残差的边 EdgeSE3Prior，并在该类内部定义了如何计算残差和雅可比，如下图:

```
void EdgeSE3Prior::ComputeResidual() {
    VecX param_i = verticies_[0]->Parameters();
    Qd Qi( w: param_i[6],  x: param_i[3],  y: param_i[4],  z: param_i[5]);
    Vec3 Pi = param_i.head<3>();

//    std::cout << Qi.vec().transpose() <<" "<< Qp_.vec().transpose()<<std::endl;
    // rotation error
#ifdef USE_SO3_JACOBIAN
    Sophus::SO3d ri( quat: Qi);
    Sophus::SO3d rp( quat: Qp_);
    Sophus::SO3d res_r = rp.inverse() * ri;
    residual_.block<3,1>( startRow: 0, startCol: 0) = Sophus::SO3d::log( other: res_r);
#else
    residual_.block<3,1>(0,0) = 2 * (Qp_.inverse() * Qi).vec();
#endif
    // translation error
    residual_.block<3,1>( startRow: 3, startCol: 0) = Pi - Pp_;
//    std::cout << residual_.transpose() <<std::endl;
}
```

```
void EdgeSE3Prior::ComputeJacobians() {

    VecX param_i = verticies_[0]->Parameters();
    Qd Qi( w: param_i[6],  x: param_i[3],  y: param_i[4],  z: param_i[5]);

    // w.r.t. pose i
    Eigen::Matrix<double, 6, 6> jacobian_pose_i = Eigen::Matrix<double, 6, 6>::Zero();

#ifdef USE_SO3_JACOBIAN
    Sophus::SO3d ri( quat: Qi);
    Sophus::SO3d rp( quat: Qp_);
    Sophus::SO3d res_r = rp.inverse() * ri;
    // http://rpg.ifi.uzh.ch/docs/RSS15_Forster.pdf  公式A.32
    jacobian_pose_i.block<3,3>( startRow: 0, startCol: 3) = Sophus::SO3d::JacobianRInv( w: res_r.log());
#else
    jacobian_pose_i.block<3,3>(0,3) = Qleft(Qp_.inverse() * Qi).bottomRightCorner<3, 3>();
#endif
    jacobian_pose_i.block<3,3>( startRow: 3, startCol: 0) = Mat33::Identity();

    jacobians_[0] = jacobian_pose_i;
//    std::cout << jacobian_pose_i << std::endl;
}
```

我们使用 EdgeSE3Prior 这种边的定义，在源文件 TestMonoBA.cpp 中，我们可以依靠第一帧和第二帧的实际位姿作为构造函数的实参来构建两个先验位姿残差项的边，并将其加入到 problem 中。如下图:

```
double wp = 1e5;
for (size_t n = 0; n < 2; ++n) {
    // 将先验的位姿残差作为额外的边，放入problem进行优化，residual_dimension = 6
    shared_ptr<EdgeSE3Prior> edge_prior( p: new EdgeSE3Prior( p: cameras[n].twc, q: cameras[n].qwc));
    std::vector<std::shared_ptr<Vertex> > edge_prior_vertex;
    edge_prior_vertex.push_back(vertexCams_vec[n]);
    // edge_prior只连接一个顶点，就是先验位姿
    edge_prior->SetVertex( vertices: edge_prior_vertex);

    // edge_prior->Information()返回inforamtion_,构造边时初始化为单位矩阵，维度为residual_dimension的方阵
    // 增大先验位姿残差的权重，即增大对应的信息矩阵
    edge_prior->SetInformation(edge_prior->Information() * wp);
    problem.AddEdge( edge: edge_prior);
}
```

加入该先验位置残差边进行优化后的结果如下，我们发现第一个和第二个相机位置被固定在了真实的相机位置附近，当然后续的相机位姿也会跟着挪动到真实相机位置附近:

```
RMSE of points: 0.000120879
------------ pose translation ----------------
translation after opt: 0 : 3.7616e-11 1.03705e-10  5.4084e-10 || gt: 0 0 0
translation after opt: 1 : -1.0718        4 0.866025 || gt:  -1.0718        4 0.866025
translation after opt: 2 :-4.00011  6.92815 0.866042 || gt:       -4   6.9282 0.866025
RMSE of pose: 6.92753e-05
```

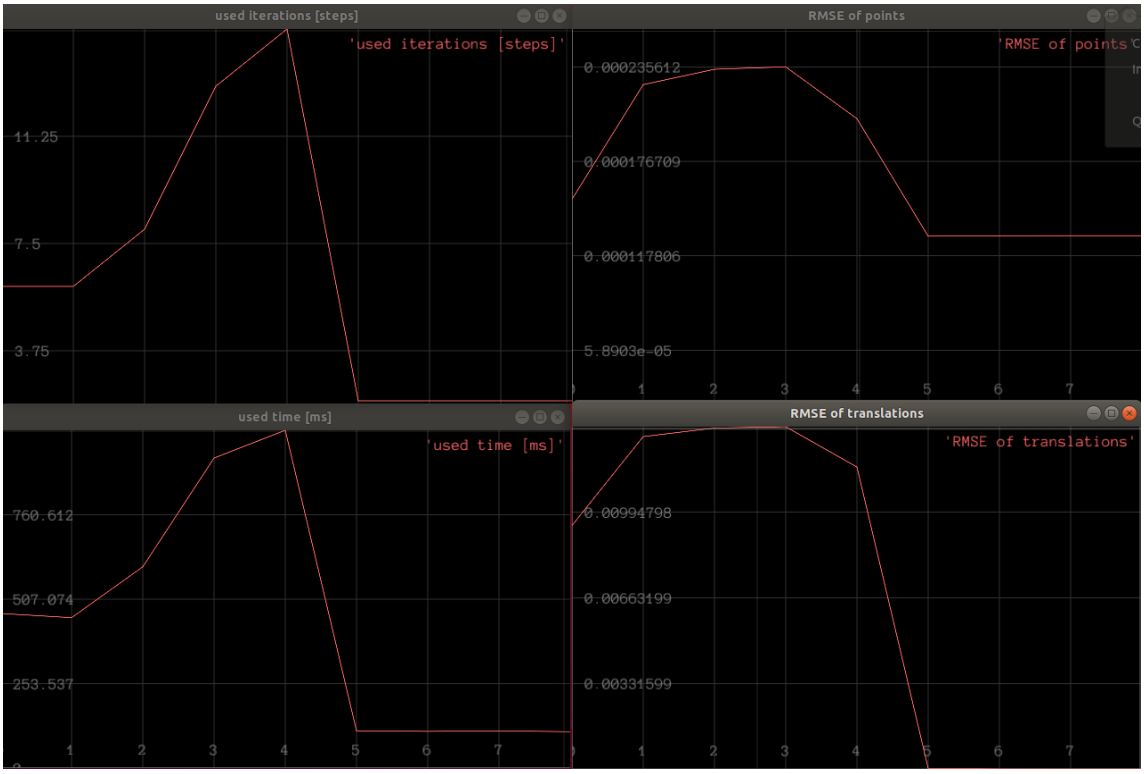
我们唯一需要做的就是改变这两条先验残差边的权重，来比较不同权重下，优化问题的计算精度和速度，为此我们可以计算估计的路标点位置和相机位姿的均方根误差 RMSE，代码如下:

```
double RMSE_points = 0.;
std::cout << "\nCompare MonoBA results after opt..." << std::endl;
for (size_t k = 0; k < allPoints.size(); k+=1) {
    std::cout << "after opt, point " << k << " : gt " << 1. / points[k].z() << " ,noise "
              << noise_invd[k] << " ,opt " << allPoints[k]->Parameters() << std::endl;
    Vec1 e = Vec1( x: 1. / points[k].z()) - allPoints[k]->Parameters();
    RMSE_points += e.transpose() * e;
}
RMSE_points = std::sqrt( x: RMSE_points/allPoints.size());
std::cout << "RMSE of points: " << RMSE_points << std::endl;

std::cout<<"------------ pose translation ----------------"<<std::endl;
double RMSE_pose = 0.;
for (size_t i = 0; i < vertexCams_vec.size(); ++i) {
    std::cout<<"translation after opt: "<< i <<" :"<< vertexCams_vec[i]->Parameters().head( n: 3).transpose()
    << " || gt: "<<cameras[i].twc.transpose()<<std::endl;
    Vec3 e = vertexCams_vec[i]->Parameters().head( n: 3) - cameras[i].twc;
    RMSE_pose += e.transpose() * e;
}
RMSE_pose = std::sqrt( x: RMSE_pose/vertexCams_vec.size());
std::cout << "RMSE of pose: " << RMSE_pose << std::endl;
/// 优化完成后，第一帧相机的 pose 平移 (x,y,z) 不再是原点 0,0,0. 说明向零空间发生了漂移。
/// 解决办法： fix 第一帧和第二帧，固定 7 自由度。 或者加上非常大的先验值。
```

我们设置先验位姿残差的权重$\omega^p$分别为 1, 10, 100, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8，并且为了获得更好的比较结果，我们设置 featureNums = 30，poseNums = 10。我们分别输出优化使用的迭代步骤，优化使用的时间，估计的路标点的 RMSE，以及估计的相机位置(pose translation)的 RMSE，如下:

```
/home/jindong/VIO/Exercise/L5/hw_course5_new/cmake
----------weight of prior wp: 1 ------------
used iterations: 6 steps
problem solve cost: 464.339 ms
RMSE of points: 0.000153057
RMSE of pose: 0.00943609

----------weight of prior wp: 10 ------------
used iterations: 6 steps
problem solve cost: 451.557 ms
RMSE of points: 0.000224517
RMSE of pose: 0.0128811

----------weight of prior wp: 100 ------------
used iterations: 8 steps
problem solve cost: 604.286 ms
RMSE of points: 0.000234095
RMSE of pose: 0.0132113

----------weight of prior wp: 1000 ------------
used iterations: 13 steps
problem solve cost: 930.855 ms
RMSE of points: 0.000235612
RMSE of pose: 0.013264

----------weight of prior wp: 10000 ------------
used iterations: 15 steps
problem solve cost: 1014.15 ms
RMSE of points: 0.000203275
RMSE of pose: 0.0117022

----------weight of prior wp: 100000 ------------
used iterations: 2 steps
problem solve cost: 112.181 ms
RMSE of points: 0.000130029
RMSE of pose: 2.68235e-05

----------weight of prior wp: 1e+06 ------------
used iterations: 2 steps
problem solve cost: 110.255 ms
RMSE of points: 0.000130378
RMSE of pose: 1.52626e-06

----------weight of prior wp: 1e+07 ------------
used iterations: 2 steps
problem solve cost: 111.845 ms
RMSE of points: 0.000130403
RMSE of pose: 3.21817e-08
```

我们使用 Pangolin 将优化使用的迭代步骤，优化使用的时间，估计的路标点的 RMSE，以及估计的相机位置(pose translation)的 RMSE 随着不同先验位姿残差权重$\omega^p$的变化情况绘制出来，如下图:




(这里横坐标表示的是 $log_{10}(\omega^p)$)

我们发现结果与论文[1]中的结果类似。我们发现估计路标点位置和相机位姿的误差 RMSE 一开始都随着先验权重增加而增加 ($log_{10}(\omega^p) = 0 \sim 3$), 随后 RMSE 随着先验权重增加而减小 ($log_{10}(\omega^p) = 3 \sim 5$), 最后 RMSE 固定在一个稳定值 ($log_{10}(\omega^p) \geq 6$)。不同先验权重下的计算时间和迭代步数也有相似的变化。所以我们可以将 $\omega^p$ 设为大于$10^5$，得到最好的优化精度和最快的优化速度。同时论文中提到太大的先验权重会造成优化的不稳定，所以我们可以简单地将 $\omega^p$ 设为$10^5$。

**参考文献:**

**[1] Zichao Zhang, Guillermo Gallego, and Davide Scaramuzza. "On the Comparison of Gauge Freedom Handling in Optimization-based Visual-Inertial State Estimation". In: IEEE ROBOTICS AND AUTOMATION LETTERS. PREPRINT VERSION. ACCEPTED APRIL, 2018.**