# 一. 补全 BA 求解器代码

完成单目 Bundle Adjustment 求解器 problem.cc 中的部分代码

## 1. 完成 Problem::MakeHessian() 中信息矩阵 H 的计算

答:

完成代码中待完成部分，。主要工作就是将所有节点相关的 $H = J^\top J$ 子块加入到整体的 Hessian 矩阵中，通过 index_i 和 index_j 调整加入的字块的位置。具体的原理如图 1 所示:

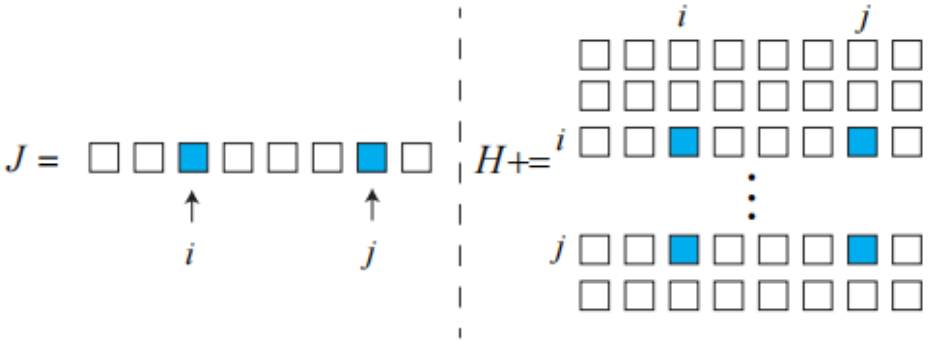


图 1: MakeHessian

代码修改结果如下，通过两层 for 循环，从 Hessian 矩阵左上角 make 到右下角。其中如果是非对角线元素，则在添加了 $H_{ij}$ 之后，还要添加 $H_{ji}$(即最后一个 if 里面的语句作用)

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

    auto jacobian_i = 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 = verticies[j];
```

```
13
14          if (v_j->IsFixed()) continue;
15
16          auto jacobian_j = jacobians[j];
17          ulong index_j = v_j->OrderingId();
18          ulong dim_j = v_j->LocalDimension();
19
20          assert(v_j->OrderingId() != -1);
21          MatXX hessian = JtW * jacobian_j;
22          // 所有的信息矩阵叠加起来
23          // TODO:: home work. 完成 H index 的填写.
24          H.block(index_i, index_j, dim_i, dim_j).noalias() += hessian;
25          if (j != i) {
26          // 对称的下三角
27          // TODO:: home work. 完成 H index 的填写.
28              H.block(index_j, index_i, dim_j, dim_i).noalias() += hessian.transpose();
29          }
30      }
31      b.segment(index_i, dim_i).noalias() -= JtW * edge.second->Residual();
```

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

答:

根据 Schur 补的公式进行实现即可，代码实现如下:

```
1  /*
2   * Solve Hx = b, we can use PCG iterative method or use sparse Cholesky
3   */
4  void Problem::SolveLinearSystem() {
5      if (problemType_ == ProblemType::GENERIC_PROBLEM) {
6          // 非 SLAM 问题直接求解
7          // PCG solver
```

```
8           MatXX H = Hessian_;
9           for (ulong i = 0; i < Hessian_.cols(); ++i) {
10              H(i, i) += currentLambda_;
11          }
12          // delta_x_ = PCGSolver(H, b_, H.rows() * 2);
13          delta_x_ = Hessian_.inverse() * b_;
14      } else {
15          // SLAM 问题采用舒尔补的计算方式
16          // step1: schur marginalization --> Hpp, bpp
17          int reserve_size = ordering_poses_;
18          int marg_size = ordering_landmarks_;
19          // TODO:: home work. 完成矩阵块取值，Hmm，Hpm，Hmp，bpp，bmm
20          MatXX Hmm = Hessian_.block(reserve_size, reserve_size, marg_size, marg_size);
21          MatXX Hpm = Hessian_.block(0, reserve_size, reserve_size , marg_size);
22          MatXX Hmp = Hessian_.block(reserve_size, 0, marg_size, reserve_size );
23          VecX bpp = b_.segment(0,reserve_size);
24          VecX bmm = b_.segment(reserve_size,marg_size);
25          // Hmm 是对角线矩阵，它的求逆可以直接为对角线块分别求逆
26          // 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
27          MatXX Hmm_inv(MatXX::Zero(marg_size, marg_size));
28          for (auto landmarkVertex : idx_landmark_vertices_) {
29              int idx = landmarkVertex.second->OrderingId() - reserve_size;
30              int size = landmarkVertex.second->LocalDimension();
31              Hmm_inv.block(idx,idx, size , size ) = Hmm.block(idx,idx, size , size ).inverse();
32          }
33          // TODO:: home work. 完成舒尔补 Hpp, bpp 代码
34          MatXX tempH = Hpm * Hmm_inv;
35          H_pp_schur_ = Hessian_.block(0, 0, reserve_size , reserve_size ) - tempH * Hmp;
36          b_pp_schur_ = bpp - tempH * bmm;
37          // step2: solve Hpp * delta_x = bpp
38          VecX delta_x_pp(VecX::Zero(reserve_size));
39          // PCG Solver
```

```
40          for (ulong i = 0; i < ordering_poses_; ++i) {
41              H_pp_schur_(i, i) += currentLambda_;
42          }
43          int n = H_pp_schur_.rows() * 2;                         // 迭代次数
44          delta_x_pp = PCGSolver(H_pp_schur_, b_pp_schur_, n);
45          // 哈哈，小规模问题，搞 pcg 花里胡哨
46          delta_x_.head( reserve_size ) = delta_x_pp;
47          //          std :: cout << delta_x_pp.transpose() << std::endl;
48          // TODO:: home work. step3: solve landmark
49          VecX delta_x_ll(marg_size);
50          delta_x_ll = Hmm_inv * (bmm - Hmp * delta_x_pp);
51          delta_x_. tail (marg_size) = delta_x_ll ;
52      }
53  }
```

## 二. 完成滑动窗口算法测试函数

题目：完成 problem.cc 文件中的 Problem::TestMarginalize() 中的代码，并通过测试。

答：代码内容在注释中有，大概意思就是将需要 marg 的 vertex 所在行列先移动到最后一行最后一列，然后根据图 2 所示完成 marg 操作，获得先验信息矩阵。(只不过图 2 中 marg 的是第一行第一列的元素，而作用代码中是第二行第二列的元素，而且信息矩阵维度也不一样，即 vertex 数量也不同)

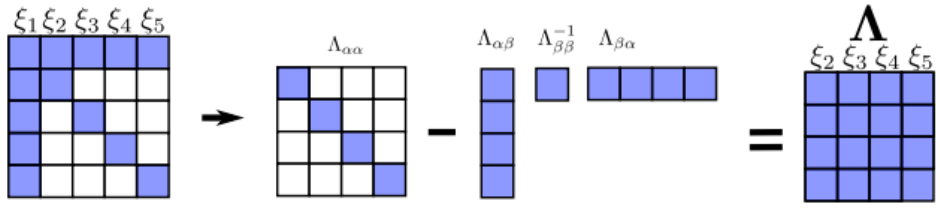


图 2: Marg

```
1  void Problem:: TestMarginalize () {
2
```

```
3    // Add marg test
4    int idx = 1;              // marg 中间那个变量
5    int dim = 1;              // marg 变量的维度
6    int reserve_size = 3;     // 总共变量的维度
7    double delta1 = 0.1 * 0.1;
8    double delta2 = 0.2 * 0.2;
9    double delta3 = 0.3 * 0.3;
10   int cols = 3;
11   MatXX H_marg(MatXX::Zero(cols, cols));
12   H_marg << 1./delta1, -1./delta1, 0,
13           -1./delta1, 1./delta1 + 1./delta2 + 1./delta3, -1./delta3,
14           0., -1./delta3, 1/delta3;
15   std::cout << "---------- TEST Marg: before marg------------"<< std::endl;
16   std::cout << H_marg << std::endl;
17   // TODO:: home work. 将变量移动到右下角
18   /// 准备工作: move the marg pose to the Hmm bottown right
19   // 将 row i 移动矩阵最下面
20   /// @details 先移动的目的是，不论被边缘化的vertex在哪里
21   // 后面的边缘化都只对最后几行进行操作就可以
22   Eigen::MatrixXd temp_rows = H_marg.block(idx, 0, dim, reserve_size );
23   Eigen::MatrixXd temp_botRows =
24       H_marg.block(idx + dim, 0, reserve_size - idx - dim, reserve_size );
25   H_marg.block(idx, 0, reserve_size - dim - idx, reserve_size ) = temp_botRows;
26   H_marg.block(reserve_size - dim, 0, dim, reserve_size ) = temp_rows;
27   // 将 col i 移动矩阵最右边
28   Eigen::MatrixXd temp_cols = H_marg.block(0, idx, reserve_size , dim);
29   Eigen::MatrixXd temp_rightCols =
30       H_marg.block(0, idx + dim, reserve_size , reserve_size - idx - dim);
31   H_marg.block(0, idx, reserve_size , reserve_size - idx - dim) = temp_rightCols;
32   H_marg.block(0, reserve_size - dim, reserve_size , dim) = temp_cols;
33   std::cout << "------- TEST Marg: 将变量移动到右下角---------"<< std::endl;
34   std::cout<< H_marg <<std::endl;
```

```
35      /// 开始 marg： schur
36      double eps = 1e-8;
37      int m2 = dim;        // 为了后面的代码更加简单
38      int n2 = reserve_size - dim;   // 剩余变量的维度
39      Eigen::MatrixXd Amm = 0.5 * (H_marg.block(n2, n2, m2, m2) +
40          H_marg.block(n2, n2, m2, m2).transpose());
41      Eigen:: SelfAdjointEigenSolver <Eigen::MatrixXd> saes(Amm);
42      Eigen::MatrixXd Amm_inv = saes.eigenvectors() * Eigen::VectorXd
43          ((saes.eigenvalues().array() > eps).select
44          (saes.eigenvalues().array().inverse(), 0)).asDiagonal() *
45                                  saes.eigenvectors().transpose();
46      // TODO:: home work. 完成舒尔补操作
47      Eigen::MatrixXd Arm = H_marg.block(0, n2, n2,m2);
48      Eigen::MatrixXd Amr = H_marg.block(n2, 0, m2,n2);
49      Eigen::MatrixXd Arr = H_marg.block(0, 0, n2, n2);
50      Eigen::MatrixXd tempB = Arm * Amm_inv;
51      Eigen::MatrixXd H_prior = Arr - tempB * Amr;
52      std::cout << "---------- TEST Marg: after marg------------"<< std::endl;
53      std::cout << H_prior << std::endl;
54  }
```

完成之后编译运行./testMonoBA，程序中通过仿真产生了 3 个 cam pos，20 个 landmark，只不过 Problem::TestMarginalize() 在这里好像没有用到 problem 对象中的成员变量？程序运行结果包括了前面提到的 solve() 函数求解结果等，具体如下所示：

```
0 pose order: 0
1 pose order: 6
2 pose 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: 1.5668 ms
   makeHessian cost: 0.891005 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 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
---------- 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
```

# 三. Paper Reading

题目：阅读论文 [1]，总结关于优化过程中处理 H 自由度的不同操作方式，总结文章内容，需要包括：具体处理方式，实验效果，结论等。

答：

## 1. 摘要

单目 VI(visual-inertial) 系统有 4 个自由度不可观 (unobservable)(本文中称为 Gauge Freedom，不太清楚这个 gauge 怎么翻译，好像是物理相关的，gauge theory 就是规范场论？)，分别是 X、Y、Z 的绝对位置以及 yaw 角。因此其 Hessian 矩阵也是是奇异矩阵，其零空间的维度为 4，因此 $H\Delta x = b$ 这个增量方程 or 正规方程 (normal equation) 并没有唯一解。而在实际的优化过程中，我们需要连贯的唯一解，不然定位就定得不是很位。为了求得唯一解 (unique solution)，研究人员们采用了许多不同的方式 (主要是 3 种)，这篇文章对这些方式的优缺点进行了总结，并对其结果进行了对比。

## 2. 实际处理方法

实际中主要的处理方法有中，本文的主要目的也就是分析这三种方法的优缺点，以及有没有明显的差别。这三种方法及其主要思路是这样的：

- gauge fixation: 将不可观测的状态固定到某些给定值，在优化过程中固定第一个相机的 pos 和 yaw 角。在更小的参数空间中进行优化，在这个空间中系统没有不客观维度，Heissian 矩阵是可逆的。这是一种比较符合直觉的想法 (natural way)，将第一帧的坐标认为是世界坐标的原点？但是添加了更多约束。
- gauge prior：给不可观的状态加上先验，通过添加一个惩罚项，使得 Hessian 矩阵可逆 (非奇异)，相当于在目标函数中添加一个由先验组成的惩罚项。
- free gauge：允许不可观状态量在优化中随意变化，使用奇异的 Hessian 的伪逆矩阵隐性地提供额外的约束 (使用最小范数来更新参数)，以获得唯一解。

前两种方法需要 (the first two strategies require VI problem-specific knowledge(which state to constrain) 这句话中间那个 “VI” 没看懂什么意思)。三种方法得到的状态空间如表 1 所示：

| | 参数维度 vec. | Hessian(Noral eqs) | 举例 |
|---|---|---|---|
| fixed gauge | n - 4 | inverse, (n-4)×(n-4) | 固定第一帧 |
| gauge prior | n | inverse, n×$n$ | 对 1st pose's H 添加单位阵 I |
| free gauge | n | pseudoinverse, n×$n$ | LM 法 |

表 1: 不同 gauge freedom 处理方法

## 3. 评估方法

本文分别从精度、计算效率以及协方差三个维度对上述三种方法进行了评估，得出一下结论：

- 精度：三种方法具有相同的进度。(精度评估准则: RMSE-位置和速度的欧式距离、旋转的相对角度误差)
- 计算效率 gauge prior 方法需要选择合适的先验权值以避免计算量的增加。在适当的权重值下，gauge prior 与 gauge fixed 计算性能几乎相同
- gauge free 略快于其他方法，只需要较少的迭代就能收敛。

最后贴一个图，这就是 VI 的 NULL SPACE 吗，然后 gauge fixation 在指定的 gauge $C$ 上移动，因为它的零空间变成 1 维了。

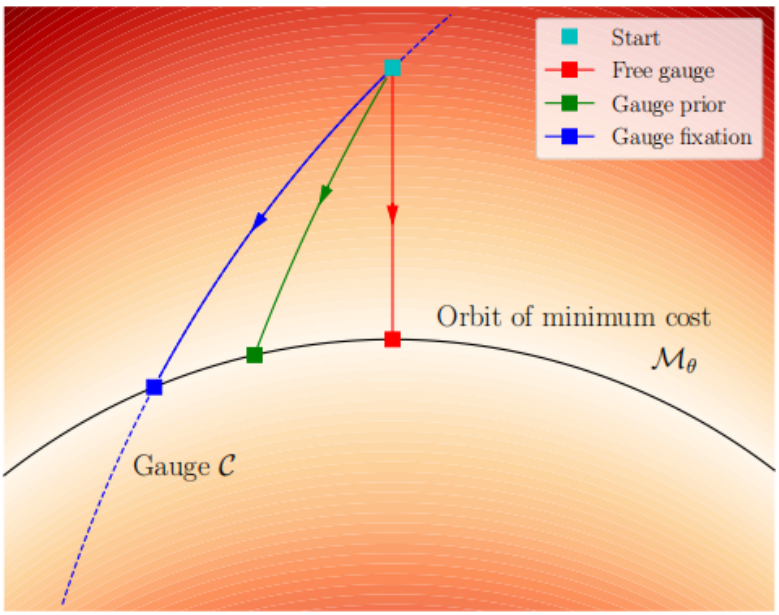


Fig. 2: Illustration of the optimization paths taken by different gauge handling approaches. The gauge fixation approach always moves on the gauge $\mathcal{C}$, thus satisfying the gauge constraints. The free gauge approach uses the pseudoinverse to select parameter steps of minimal size for a given cost decrease, and therefore, moves perpendicular to the isocontours of the cost (1). The gauge prior approach follows a path in between the gauge fixation and free gauge approaches. It minimizes a cost augmented by (11), so it may not exactly end up on the orbit of minimum visual-inertial cost (1).

# 四. 添加 prior 约束

题目：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 求解收敛精度和速度。

其实我没怎么搞懂，看程序的意思是不是就是给前两个 cam pos 的节点再添加一条边的约束，而这条边的约束是先验信息的约束。抄的代码是在 testMonoBA.cpp 中，将所有 cam pos 加入节点序列之后，对前两个节点加入 prior_edge 这个边，具体代码如下：

```
//**************************添加先验项************************//
double weight = 1e5;
for(size_t k=0;k<2;k++){
    shared_ptr<EdgeSE3Prior> edge_prior
                    (new EdgeSE3Prior(cameras[k].twc, cameras[k].qwc));
    std::vector<std::shared_ptr<Vertex>> edge_prior_vertex;
    edge_prior_vertex.push_back(vertexCams_vec[k]);
    edge_prior->SetVertex(edge_prior_vertex);
    edge_prior->SetInformation(edge_prior->Information()*weight);
    problem.AddEdge(edge_prior);
}
//**************************添加先验项************************//
```

加入先验信息之后的对比结果如下：

| 权重 | 迭代次数 | 求解时间(ms) | RMSE Translation | RMSE Rotation |
|---|---|---|---|---|
| 0 | 3 | 1.08324 | 0.00261831 | 0.00301854 |
| 3 | 3 | 0.645179 | 0.00678568 | 0.000992244 |
| 10 | 3 | 0.638841 | 0.00315454 | 0.000519869 |
| 100 | 3 | 0.630379 | 0.00375711 | 0.000575194 |
| 500 | 3 | 0.928295 | 0.00422012 | 0.000623944 |
| 5000 | 4 | 0.776276 | 0.00435992 | 0.000657869 |
| 1.00E+04 | 4 | 0.799114 | 0.00400621 | 0.000631276 |
| 1.00E+05 | 5 | 1.05863 | 0.0031826 | 0.000582881 |
| 1.00E+06 | 3 | 0.596518 | 0.000188488 | 0.000256018 |
| 1.00E+07 | 3 | 0.611311 | 1.05E-05 | 2.30E-05 |
| 1.00E+08 | 3 | 0.616097 | 2.51836E-07 | 5.75E-07 |
| 1.00E+09 | 3 | 0.62331 | 2.93E-09 | 6.74E-09 |
| 1.00E+10 | 3 | 0.627971 | 2.98E-11 | 6.86E-11 |

但是在 edge_prior.cc 文件里面有一个 void EdgeSE3Prior::ComputeJacobians() 这个函数里面的雅克比感觉跟有一个作业讲解的不一样吗，一个是添加在左上角 + 右下角，但是代码里面好像是添加在右上角 + 左下角？想问一下这个公式出自哪里啊？谢谢大哥。

$$r_{prior} = \begin{bmatrix} r_R \\ r_p \end{bmatrix}_{prior} = \begin{bmatrix} \ln(\tilde{R}_{wc}^{-1} R_{wc}) \\ p_{wc} - \tilde{p}_{wc} \end{bmatrix} \qquad \tilde{R}$$

$$J = \frac{\partial r_{prior}}{\partial \begin{bmatrix} R_{wc} \\ p_{wc} \end{bmatrix}} = \begin{bmatrix} \frac{\partial r_R}{\partial R_{wc}} & 0 \\ 0 & \frac{\partial r_p}{\partial p_{wc}} \end{bmatrix}$$

$$\begin{aligned} \frac{\partial r_R}{\partial R_{wc}} &= \frac{\partial \ln(\tilde{R}_{wc}^{-1} R_{wc})}{\partial R_{wc}} \\ &= \lim_{\delta\theta \to 0} \frac{\ln(\tilde{R}_{wc}^{-1} R_{wc} \exp(\delta\theta)) - \ln(\tilde{R}_{wc}^{-1} R_{wc})}{\delta\theta} \\ &= \lim_{\delta\theta \to 0} \frac{r_R + J_r^{-1} \delta\theta - r_R}{\delta\theta} \\ &= J_r^{-1}(r_R) \end{aligned}$$

$$\frac{\partial r_p}{\partial p_{wc}} = \frac{\partial (p_{wc} - \tilde{p}_{wc})}{\partial p_{wc}} = I$$

图 3: 先验约束?

下面是函数的实现

```
1  void EdgeSE3Prior::ComputeJacobians() {
2      VecX param_i = verticies_[0]->Parameters();
3      Qd Qi(param_i[6], param_i[3], param_i[4], param_i[5]);
4      // w.r.t. pose i
5      Eigen::Matrix<double, 6, 6> jacobian_pose_i = Eigen::Matrix<double, 6, 6>::Zero();
6  #ifdef USE_SO3_JACOBIAN
7      Sophus::SO3d ri(Qi);
8      Sophus::SO3d rp(Qp_);
9      Sophus::SO3d res_r = rp.inverse() * ri;
10     // http://rpg.ifi.uzh.ch/docs/RSS15_Forster.pdf 公式A.32
```

```
11      // 下面这一行应该是把3x3的JR加在jacobian_pose_i的右上角的意思哦--是吧
12      // 我就是感觉这里好像和公式不太一样
13      jacobian_pose_i.block<3,3>(0,3) = Sophus::SO3d::JacobianRInv(res_r.log()); // JR
14  #else
15      jacobian_pose_i.block<3,3>(0,3) = Qleft(Qp_.inverse() * Qi).bottomRightCorner<3, 3>();
16  #endif
17      jacobian_pose_i.block<3,3>(3,0) = Mat33::Identity();     // Jp
18      jacobians_[0] = jacobian_pose_i;
19  //      std::cout << jacobian_pose_i << std::endl;
20  }
```

# 参考文献

[1] Zhang, Z., Gallego, G., Scaramuzza, D. (2018). On the comparison of gauge freedom handling in optimization-based visual-inertial state estimation. IEEE Robotics and Automation Letters, 3(3), 2710-2717.

[2] Bloesch, M., Burri, M., Omari, S., Hutter, M., Siegwart, R. (2017). Iterated extended Kalman filter based visual-inertial odometry using direct photometric feedback. The International Journal of Robotics Research, 36(10), 1053-1072.