第十章思路讲解

大家好，很荣幸在这里和大家分享第十课的作业。本章节作业相比较前几章节需要填空的地方更多了，因此也更加具有难度了。考虑到本章节代码较多，这里就不再做一些复杂的说明。作业需要我们完成三项任务，第一，补充代码，第二，与EKF做对比，最后，分析滑窗长度对性能的影响，并做一些分析。

由于代码对于初次接触这个领域的人可能会有点困难，因此首先和大家推荐下本章节的补充材料。由于本章节代码框架与`VINS`框架近似，因此大家可以参考下`vins-mon`o滑动窗口，因子图这部分。另外，大家可以看到课程ppt上采用的是四元数推导，这样可能相对比较简单，但是代码实现却是使用的李代数版本，邱博的代码从李代数角度重新对这块进行了推导，提供另外一个角度去思考。

值得一提，邱博里面有些推导可能有些争议，我将在下一章节进行论述，为了统一表述，一些公式推导都是以机器人状态估计为主。

代码补全

深蓝学院 shenlanxueyuan.com

参考资料：

- VINS系列相关代码：滑动窗口部分
- 邱笑晨博士的《预积分总结与公式推导》（发表于泡泡机器人公众号）
- 机器人状态估计：理论推导部分
- 深蓝学院《从零手写VIO》课程：滑窗和边缘化理论讲解部分

课程代码中提供了`JacobianRInv`这个函数，用来表示雅各比`BCH`的右乘，这里的公式和邱笑晨博士的《预积分总结与公式推导》表述是一致的：见左图；这也是我一开始使用的方法，但是后来发现《机器人状态估计》关于该公式表示如右图所示。通过实验验证，这种表示的代价函数相较前者小一个数量级，从个位数代价会降低到`10e-2`级别，定位精度也有一定提升。因此我个人更倾向于使用后者。

代码补全

深蓝学院 shenlanxueyuan.com

争议问题：右乘BCH及其求逆公式

- 课程代码中提供了JacobianRInv这个函数，用来表示雅各比BCH的右乘，这里的公式和邱笑晨博士的《预积分总结与公式推导》表述是一致的：见左图

$$\mathbf{J}_r(\vec{\phi}) = \mathbf{I} - \frac{1-\cos(\|\vec{\phi}\|)}{\|\vec{\phi}\|^2}\vec{\phi}^\wedge + \frac{\|\vec{\phi}\| - \sin(\|\vec{\phi}\|)}{\|\vec{\phi}\|^3}(\vec{\phi}^\wedge)^2$$

$$\mathbf{J}_r^{-1}(\vec{\phi}) = \mathbf{I} + \frac{1}{2}\vec{\phi}^\wedge + \left(\frac{1}{\|\vec{\phi}\|^2} - \frac{1+\cos(\|\vec{\phi}\|)}{2\cdot\|\vec{\phi}\|\cdot\sin(\|\vec{\phi}\|)}\right)(\vec{\phi}^\wedge)^2$$

- 但是《机器人状态估计》关于该公式表示如右图所示。通过实验验证，这种表示的代价函数相较前者小一个数量级，精度也有一定提升。

$$J_r(\phi)^{-1} = \sum_{n=0}^{\infty}\frac{B_n}{n!}(-\phi^\wedge)^n = \frac{\phi}{2}\cot\frac{\phi}{2}\mathbf{1} + \left(1-\frac{\phi}{2}\cot\frac{\phi}{2}\right)aa^T + \frac{\phi}{2}a^\wedge$$

$$J_l(\phi)^{-1} = \sum_{n=0}^{\infty}\frac{B_n}{n!}(\phi^\wedge)^n = \frac{\phi}{2}\cot\frac{\phi}{2}\mathbf{1} + \left(1-\frac{\phi}{2}\cot\frac{\phi}{2}\right)aa^T - \frac{\phi}{2}a^\wedge$$

代码实现如下展示，

争议问题：右乘BCH及其求逆公式

```cpp
static Eigen::Matrix3d JacobianRInv(const Eigen::Vector3d &w)
{
  Eigen::Matrix3d J_r_inv = Eigen::Matrix3d::Identity();

  double theta = w.norm();

  if (theta > 1e-5)
  {
    Eigen::Vector3d a = w.normalized();
    Eigen::Matrix3d a_hat = Sophus::SO3d::hat(a);
    double theta_half = 0.5 * theta;
    double cot_theta = 1.0 / tan(theta_half);

    J_r_inv = theta_half * cot_theta * J_r_inv + (1.0 -
    theta_half * cot_theta) * a * a.transpose() +
    theta_half * a_hat;
  }

  return J_r_inv;
}
```

```cpp
static Eigen::Matrix3d JacobianR(const Eigen::Vector3d &w)
{
  Eigen::Matrix3d J_r = Eigen::Matrix3d::Identity();

  double theta = w.norm();

  if (theta > 1e-5)
  {
    Eigen::Vector3d a = w.normalized();
    Eigen::Matrix3d a_hat = Sophus::SO3d::hat(a);

    J_r = sin(theta) / theta * Eigen::Matrix3d::Identity()
    + (1.0 - sin(theta) / theta) * a * a.transpose() - (1.0
    - cos(theta)) / theta * a_hat;
  }

  return J_r;
}
```

还有一个问题我看到有人在群里问，所以在这里解释下。由于每次迭代需要使用奇异值分解，从边缘化后的信息矩阵中恢复出来雅克比矩阵`linearized_jacobians`和残差`linearized_residuals`，这两者会作为先验残差带入到下一轮的先验残差的雅克比和残差的计算当中去。这一部分和vins是差不多的，如果看不懂的可以去`vins`那边参考参考。其中推导部分相对比较简单，如图片所示

代码补全

深蓝学院 shenlanxueyuan.com

雅各比矩阵和残差获取

每次迭代需要使用奇异值分解，从H和b中获取相应的雅各比矩阵和残差便于更新：

$$H = J^T J = QSQ^T$$

其中，S为奇异值，Q为对应的右奇异向量，那么：

$$J = \sqrt{S}\, Q^T$$

根据：

$$b = J^T e$$

有：

$$e = (J^T)^{-1} Q^T b$$

代码逻辑也是类似的，我们首先使用Eigen进行SVD分解，然后按照公式推导一步步走下来就可以了

代码补全

深蓝学院 shenlanxueyuan.com

雅各比矩阵和残差获取

```
// H_tmp 是对称矩阵，所以是hermitian矩阵，
// 代码中使用Eigen::SelfAdjointEigenSolver完成对herrmitian的SVD分解。得到：
Eigen::SelfAdjointEigenSolver<Eigen::MatrixXd> saes(H_tmp);
// 这部分参考vins代码：https://zhuanlan.zhihu.com/p/51330624;（思路是差不多的）

// (R.array() > s).select(P,Q)  -> (R > s ? P : Q)
Eigen::VectorXd S = Eigen::VectorXd((saes.eigenvalues().array() > 1.0e-5).select(saes.eigenvalues().array(), 0));
Eigen::VectorXd S_inv = Eigen::VectorXd((saes.eigenvalues().array() > 1.0e-5).select(saes.eigenvalues().array().
inverse(), 0));

// cwiseSqrt : 对每个元素做sqrt()处理
Eigen::VectorXd S_sqrt = S.cwiseSqrt();
Eigen::VectorXd S_inv_sqrt = S_inv.cwiseSqrt(); // S^{-1/2}
// 从边缘化后的信息矩阵中恢复出来雅克比矩阵linearized_jacobians和残差linearized_residuals，
// 这两者会作为先验残差带入到下一轮的先验残差的雅克比和残差的计算当中去
// J = S^{1/2} * V.t()
J_ = S_sqrt.asDiagonal() * saes.eigenvectors().transpose();
// e_0 = (V * S^{-1/2}).t() * b
e_ = S_inv_sqrt.asDiagonal() * saes.eigenvectors().transpose() * b_tmp;
```

代码里面还有一个`Cholesky`分解，这里的整体框架如下所示。

代码补全

深蓝学院 shenlanxueyuan.com

Cholesky 分解

```
// TODO: get square root of information matrix:
// Cholesky 分解 : http://eigen.tuxfamily.org/dox/classEigen_1_1LLT.html
Eigen::LLT<Eigen::Matrix<double, 15, 15>> LowerI(I_);
// sqrt_info 为上三角阵
Eigen::Matrix<double, 15, 15> sqrt_info = LowerI.matrixL().transpose();

  ......

// TODO: compute jacobians:
if (jacobians)
{
    // compute shared intermediate results:
    Eigen::Map<Eigen::Matrix<double, 15, 15, Eigen::RowMajor>> jacobian_i(jacobians[0]);
    Eigen::Map<Eigen::Matrix<double, 15, 15, Eigen::RowMajor>> jacobian_j(jacobians[1]);

    if (jacobians[0])
    {
        ......
    }

    if (jacobians[1])
    {
        ......
    }

    jacobian_i = sqrt_info * jacobian_i;
    jacobian_j = sqrt_info * jacobian_j;
}
```

另外一个核心部分就是对残差和雅各比推导，理论部分在ppt和邱博中展示，这里仅仅展示部分代码

残差和雅各比推导：以imu预积分为例

```
const Eigen::Matrix3d oriRT_i = ori_i.inverse().matrix();
Eigen::Map<Eigen::Matrix<double, 15, 1>> resid(residuals);
resid.block<3, 1>(INDEX_P, 0) = oriRT_i * (pos_j - pos_i - vel_i * T_ + 0.5 * g_ * T_ * T_) - alpha_ij;
resid.block<3, 1>(INDEX_R, 0) = (ori_ij.inverse() * (ori_i.inverse() * ori_j)).log();
resid.block<3, 1>(INDEX_V, 0) = oriRT_i * (vel_j - vel_i + g_ * T_) - beta_ij;
resid.block<3, 1>(INDEX_A, 0) = b_a_j - b_a_i;
resid.block<3, 1>(INDEX_G, 0) = b_g_j - b_g_i;
```

```
jacobian_j.block<3, 3>(INDEX_P, INDEX_P) = oriRT_i;
// b. residual, orientation:
jacobian_j.block<3, 3>(INDEX_R, INDEX_R) = J_r_inv;
// c. residual, velocity:
jacobian_j.block<3, 3>(INDEX_V, INDEX_V) = oriRT_i;
// d. residual, bias accel:
jacobian_j.block<3, 3>(INDEX_A, INDEX_A) = Eigen::Matrix3d::Identity();
jacobian_j.block<3, 3>(INDEX_G, INDEX_G) = Eigen::Matrix3d::Identity();
```

代码补全

深蓝学院 shenlanxueyuan.com

残差和雅各比推导：以imu预积分为例

```
jacobian_i.block<3, 3>(INDEX_P, INDEX_P) = -oriRT_i;
jacobian_i.block<3, 3>(INDEX_P, INDEX_R) = Sophus::SO3d::hat(ori_i.inverse() * (pos_j - pos_i - vel_i * T_ + 0.50 *
g_ * T_ * T_)).matrix();
jacobian_i.block<3, 3>(INDEX_P, INDEX_V) = -oriRT_i * T_;
jacobian_i.block<3, 3>(INDEX_P, INDEX_A) = -J_.block<3, 3>(INDEX_P, INDEX_A);
jacobian_i.block<3, 3>(INDEX_P, INDEX_G) = -J_.block<3, 3>(INDEX_P, INDEX_G);

// b. residual, orientation:
jacobian_i.block<3, 3>(INDEX_R, INDEX_R) = -J_r_inv * ((ori_j.inverse() * ori_i).matrix());
jacobian_i.block<3, 3>(INDEX_R, INDEX_G) = -J_r_inv
                                           * Sophus::SO3d::exp(resid.block<3, 1>(INDEX_R, 0)).matrix().inverse() *
                                           JacobianR(J_.block<3, 3>(INDEX_R, INDEX_G) * (b_g_i - m_.block<3, 1>
                                           (INDEX_G, 0))) * J_.block<3, 3>(INDEX_R, INDEX_G);

// c. residual, velocity:
jacobian_i.block<3, 3>(INDEX_V, INDEX_R) = Sophus::SO3d::hat(ori_i.inverse() * (vel_j - vel_i + g_ * T_));
jacobian_i.block<3, 3>(INDEX_V, INDEX_V) = -oriRT_i;
jacobian_i.block<3, 3>(INDEX_V, INDEX_A) = -J_.block<3, 3>(INDEX_V, INDEX_A);
jacobian_i.block<3, 3>(INDEX_V, INDEX_G) = -J_.block<3, 3>(INDEX_V, INDEX_G);

// d. residual, bias accel:
jacobian_i.block<3, 3>(INDEX_A, INDEX_A) = -Eigen::Matrix3d::Identity();
jacobian_i.block<3, 3>(INDEX_G, INDEX_G) = -Eigen::Matrix3d::Identity();
```

然后是边缘化的部分代码，由于篇幅限制，仅展示部分内容：

代码补全

深蓝学院 shenlanxueyuan.com

边缘化函数

```
void SetResIMUPreIntegration(
    const ceres::CostFunction *residual,
    const std::vector<double *> &parameter_blocks)
{
    ......
    // 注意顺序
    // a. H_mm:
    const Eigen::MatrixXd H_mm = J_m.transpose() * J_m;
    H_.block<15, 15>(INDEX_M, INDEX_M) += H_mm;
    // b. H_mr:
    const Eigen::MatrixXd H_mr = J_m.transpose() * J_r;
    H_.block<15, 15>(INDEX_M, INDEX_R) += H_mr;
    // c. H_rm:
    const Eigen::MatrixXd H_rm = J_r.transpose() * J_m;
    H_.block<15, 15>(INDEX_R, INDEX_M) += H_rm;
    // d. H_rr:
    const Eigen::MatrixXd H_rr = J_r.transpose() * J_r;
    H_.block<15, 15>(INDEX_R, INDEX_R) += H_rr;

    // a. b_m:
    const Eigen::MatrixXd b_m = J_m.transpose() * residuals;
    b_.block<15, 1>(INDEX_M, 0) += b_m;
    // a. b_r:
    const Eigen::MatrixXd b_r = J_r.transpose() * residuals;
    b_.block<15, 1>(INDEX_R, 0) += b_r;
    ......
}
```

```
virtual bool Evaluate(double const *const *parameters, double
*residuals, double **jacobians) const
{
    // parse parameters:
    Eigen::Map<const Eigen::Matrix<double, 15, 1>> x(parameters[0])
    ;
    Eigen::VectorXd dx = x - x_0_;

    // TODO: compute residual:
    Eigen::Map<Eigen::Matrix<double, 15, 1>> residual(residuals);
    // TODO:
    // 先验残差的更新：可以使用一阶泰勒近似
    residual = e_ + J_ * dx;

    // TODO: compute jacobian:
    if (jacobians)
    {
        if (jacobians[0])
        {
            Eigen::Map<Eigen::Matrix<double, 15, 15,
            Eigen::RowMajor>> jacobian_(jacobians[0]);
            jacobian_ = J_;
        }
    }

    return true;
}
```

代码补全

深蓝学院 shenlanxueyuan.com

边缘化函数

```
// add unary constraints:
// a. map matching / GNSS position:
if (N > 0 && measurement_config_.source.map_matching)
{
    // get prior position measurement:
    Eigen::Matrix4d prior_pose = current_map_matching_pose_.pose.
    cast<double>();

    // TODO: add constraint, GNSS position:
    sliding_window_ptr_->AddPRVAGMapMatchingPoseFactor(
        param_index_j,
        prior_pose, measurement_config_.noise.map_matching);
}
```

```
// add binary constraints:
if (N > 1)
{
    // get param block ID, previous:
    const int param_index_i = N - 2;

    // a. lidar frontend:
    // get relative pose measurement:
    Eigen::Matrix4d relative_pose = (last_key_frame_.pose.inverse
    () * current_key_frame_.pose).cast<double>();
    // TODO: add constraint, lidar frontend / loop closure
    detection:
    sliding_window_ptr_->AddPRVAGRelativePoseFactor(
        param_index_i, param_index_j,
        relative_pose, measurement_config_.noise.lidar_odometry);
    // b. IMU pre-integration:
    if (measurement_config_.source.imu_pre_integration)
    {
        // TODO: add constraint, IMU pre-integraion:
        sliding_window_ptr_->AddPRVAGIMUPreIntegrationFactor(
            param_index_i, param_index_j,
            imu_pre_integration_);
    }
}
```

在这里我们与之前实现的EKF方法进行了比较，我们着重看下优化后的比较（也就是每列第二排，着重看到`rmse`）。乾哥之前在课程里面说过，EKF的滤波方法就相当于滑动窗口为1的情况，可以看出优化方法比`ekf`效果好很多。

与EKF进行对比

深蓝学院 shenlanxueyuan.com

优化方法

```
APE w.r.t. full transformation (unit-less)
(not aligned)

       max      3.317282
      mean      1.545114
    median      1.589696
       min      0.000001
      rmse      1.675192
       sse      12703.979064
       std      0.647218

APE w.r.t. full transformation (unit-less)
(not aligned)

       max      3.317282
      mean      0.159852
    median      0.117935
       min      0.000001
      rmse      0.214325
       sse      207.948226
       std      0.142767
```

EKF方法

```
APE w.r.t. full transformation (unit-less)
(with origin alignment)

       max      3.453187
      mean      1.616198
    median      1.684988
       min      0.000001
      rmse      1.734038
       sse      13200.234901
       std      0.628324

APE w.r.t. full transformation (unit-less)
(with origin alignment)

       max      2.399886
      mean      1.136519
    median      1.116511
       min      0.000002
      rmse      1.216617
       sse      6497.891043
       std      0.434145
```

滑动窗口的长度也是一个关键的因素，过高或者过低的窗口长度会造成精度的降低。

窗口长度比较

深蓝学院 shenlanxueyuan.com

次数	2	5	10	15	20	25
RMES	0.832806	0.940843	**0.277835**	0.327291	0.342759	0.953997

可以看出，选取适当的滑动窗口大小是必要的，能够直接影响最终的性能。