关于Pose Graph求导的笔记

原始程序来自视觉SLAM 14讲, 第10.2节, 作者给出了位姿图计算细节, 但是其内容与Local Accuracy and Global Consistency for Efficient Visual SLAM中的内容相左. 事实证明, 两个地方给出的计算都是正确的, 只是因为具体的细节描述不同导致.
这里, 包含两个算法两种计算, 其最终都可以优化sphere.g2o. 下面对两个计算进行描述.

1. 14讲中的求解方法

目前还没有找到这个算法的推导出处. 如后续发现会在这里加上.
误差方程定义为

$$e = log(T_{ij}^{-1} T_i^{-1} T_j)^\vee$$

这里,

- 观测值(measurement)为T_{ij};
- 变量(顶点)T_i表示的是T_{wi}, w代表世界坐标系(world);
- $e = \begin{bmatrix} \tau & \phi \end{bmatrix}^T$, 其中$\tau$为李代数下的平移分量, ϕ为李代数下的旋转分量.

1.1 误差和雅克比推导

为了和**第2节**中的算法进行区分, 将上面误差函数重写成:

$$e = log(T_{ij}^{-1} T_{wi}^{-1} T_{wj})^\vee$$

14讲中使用左扰动模型, 对算法进行求导, 注意对于T_{wi}的左扰动实际上是对世界坐标系的扰动, 这与**第2节**也有出入.
误差对于李代数下的扰动$\delta\xi$求导的结果为:

$$\frac{e}{\delta\xi_i} = -J_r^{-1}(e) Ad(T_{wj}^{-1})$$

$$\frac{e}{\delta\xi_j} = J_r^{-1}(e) Ad(T_{wj}^{-1})$$

J_r为右乘雅克比, 考虑到e为小值, 可以使用其一阶泰勒展开得到:

$$J_r^{-1} \approx I + \frac{1}{2}\begin{bmatrix} \phi^\wedge & \tau^\wedge \\ 0_{3\times 3} & \phi^\wedge \end{bmatrix}$$

$$Ad(T) = \begin{bmatrix} R & t^\wedge \\ 0_{3\times 3} & R \end{bmatrix}$$

1.2 对应代码

```
// 给定误差求J_R^{-1}的近似
Matrix6d JRInv(const SE3d &e) {
    Matrix6d J;
    J.block(0, 0, 3, 3) = SO3d::hat(e.so3().log());
    J.block(0, 3, 3, 3) = SO3d::hat(e.translation());
    J.block(3, 0, 3, 3) = Matrix3d::Zero(3, 3);
    J.block(3, 3, 3, 3) = SO3d::hat(e.so3().log());
    J = J * 0.5 + Matrix6d::Identity();
    return J;
}

// 误差计算与书中推导一致
virtual void computeError() override {
    SE3d v1 = (static_cast<VertexSE3LieAlgebra *> (_vertices[0]))->estimate();
    SE3d v2 = (static_cast<VertexSE3LieAlgebra *> (_vertices[1]))->estimate();
    _error = (_measurement.inverse() * v1.inverse() * v2).log();
}

// 雅可比计算
virtual void linearizeOplus() override {
    SE3d v1 = (static_cast<VertexSE3LieAlgebra *> (_vertices[0]))->estimate();
    SE3d v2 = (static_cast<VertexSE3LieAlgebra *> (_vertices[1]))->estimate();
    Matrix6d J = JRInv(SE3d::exp(_error));

    _jacobianOplusXi = -J * v2.inverse().Adj();
    _jacobianOplusXj = J * v2.inverse().Adj();
}
```

1.3 问题

注意代码 `JRInv` 关于$\tau^\wedge$的代码

```
J.block(0, 0, 3, 3) = SO3d::hat(e.so3().log());
J.block(0, 3, 3, 3) = SO3d::hat(e.translation());
```

对比旋转的反对称阵, 第二行直接使用了平移部分的反对称阵, 而不是**李代数下平移分量**的反对称阵. 这里实测代码部分是正确的, 那么真实实现的J_r^{-1}实际上为:

$$J_r^{-1} \approx I + \frac{1}{2}\begin{bmatrix} \phi^\wedge & t^\wedge \\ 0_{3\times 3} & \phi^\wedge \end{bmatrix}$$

其中ϕ为误差在**李代数**下的旋转分量, t为误差在**李群**下的平移分量.

2 Local Accuracy and Global Consistency for Efficient Visual SLAM中的求解方法

见论文附录B.6, 第195页.

2.1 误差与雅克比推导

误差方程定义为

$$e = log(T_{ji} T_i T_j^{-1})$$

其实这里观测量和变量的定义均**和1.1节相反**:

- 观测值为T_{ji};
- 变量(顶点)T_i表示T_{iw}, w代表世界坐标系(world);
- $e = \begin{bmatrix} \tau & \phi \end{bmatrix}^T$, 其中$\tau$为李代数下的平移分量, ϕ为李代数下的旋转分量.

这里也使用左扰动模型, 这里扰动的是局部坐标系, 这和**第1节中不同**.

$$\begin{aligned} \frac{e}{\delta \xi_i} &= J_r^{-1}(-e) Ad(T_{ji}) \\ &= \left(I + \frac{1}{2}\begin{bmatrix} -\phi^\wedge & -\tau^\wedge \\ 0_{3\times 3} & -\phi^\wedge \end{bmatrix} \right) \cdot \begin{bmatrix} R_{ji} & t_{ji}^\wedge R_{ji} \\ 0_{3\times 3} & R_{ji} \end{bmatrix} \end{aligned}$$

$$\begin{aligned} \frac{e}{\delta \xi_j} &= -J_r^{-1}(e) \\ &= -\left(I + \frac{1}{2}\begin{bmatrix} \phi^\wedge & \tau^\wedge \\ 0_{3\times 3} & \phi^\wedge \end{bmatrix} \right) \end{aligned}$$

2.2 对应代码

```
virtual void linearizeOplus() override {
    SE3d T_wi = (static_cast<VertexSE3LieAlgebra *> (_vertices[0]))->estimate();
    SE3d T_wj = (static_cast<VertexSE3LieAlgebra *> (_vertices[1]))->estimate();

    SE3d T_iw = T_wi.inverse();
    SE3d T_jw = T_wj.inverse();

    SE3d& T_ij = _measurement;
    SE3d T_ji = T_ij.inverse();

    Eigen::Matrix<double, 6, 6> I6 = Eigen::Matrix<double, 6, 6>::Identity();
    SE3d delta_sim3 = T_ji * T_iw * T_wj;

    Eigen::Matrix<double, 6, 1> sim3 = SE3d::log(delta_sim3);
    Eigen::Vector3d tau = sim3.block<3, 1>(0, 0); // translation
    Eigen::Vector3d phi = sim3.block<3, 1>(3, 0); // rotation

    // 计算伴随
    Eigen::Matrix<double, 6, 6> Ad_T_ji = T_ji.Adj();

    Eigen::Matrix<double, 6, 6> J1 = Eigen::Matrix<double, 6, 6>::Zero();
    J1.block<3, 3>(0, 0) = -skew(phi);
    J1.block<3, 3>(3, 3) = -skew(phi);
    J1.block<3, 3>(0, 3) = -skew(tau);
    _jacobianOplusXi = (I6 + 0.5 * J1) * Ad_T_ji;

    Eigen::Matrix<double, 6, 6> J2 = Eigen::Matrix<double, 6, 6>::Zero();
    J2.block<3, 3>(0, 0) = skew(phi);
    J2.block<3, 3>(3, 3) = skew(phi);
    J2.block<3, 3>(0, 3) = skew(tau);
    _jacobianOplusXj = -(I6 + 0.5 * J2);
}
```