Jindong_Shi 2022.04.01

激光 SLAM 第二次作业

1. 补充直接线性方法的里程计标定模块代码;

代码如下：

首先是函数 Eigen::Vector3d cal_delta_distence(Eigen::Vector3d odom_pose);

形参 odom_pose 存储当前时刻里程计的全局位姿，我们希望从中计算出前后两帧里程计数据之间的位姿差。

我们将上一个时间戳的里程计坐标系写作坐标系 B，将当前时间戳的里程计坐标系写作坐标系 A，利用上一章坐标变换的技巧，可以求出当前里程计位姿在上一时刻里程计坐标系下的坐标（增量）表示。

CMakeLists.txt | Odom_Calib.hpp | Odom_Calib.cpp | main.cpp

```
//TODO:
//求解得到两帧数据之间的位姿差
//即求解当前位姿 在 上一时刻 坐标系中的坐标
Eigen::Vector3d  cal_delta_distence(Eigen::Vector3d odom_pose)
{
    Eigen::Vector3d d_pos;  //return value
    now_pos = odom_pose;
    //TODO:
    // We assume, B as the coordinate system of last time stamp, A as the coordinate system of time stamp now
    Eigen::Matrix3d TOB;
    TOB << cos( x: last_pos( index: 2)), -sin( x: last_pos( index: 2)), last_pos( index: 0),
           sin( x: last_pos( index: 2)),  cos( x: last_pos( index: 2)), last_pos( index: 1),
                 0,                        0,                          1;
    Eigen::Matrix3d TBO = TOB.inverse();

    Eigen::Matrix3d TOA;
    TOA << cos( x: now_pos( index: 2)), -sin( x: now_pos( index: 2)), now_pos( index: 0),
           sin( x: now_pos( index: 2)),  cos( x: now_pos( index: 2)), now_pos( index: 1),
            0,                          0,                           1;

    Eigen::Matrix3d TBA = TBO * TOA;
    d_pos = Eigen::Vector3d( x: TBA( row: 0,  col: 2),  y: TBA( row: 1,  col: 2),  z: std::atan2( y: TBA( row: 1,  col: 0),  x: TBA( row: 0,  col: 0)));
    //end of TODO:
    return d_pos;
}
```

然后是函数 bool OdomCalib::Add_Data(Eigen::Vector3d Odom, Eigen::Vector3d scan);

形参 Odom 和 scan 是里程计和激光雷达位姿的增量表示，即通过里程计和激光分别计算得到的前后两帧数据之间的旋转 & 平移。

通过 Eigen::Matrix::block()和 Eigen::Vector::segment()可以将对应的数据补充到矩阵 A 和向量 b 中，构建超定方程。

CMakeLists.txt | Odom_Calib.hpp | Odom_Calib.cpp | main.cpp

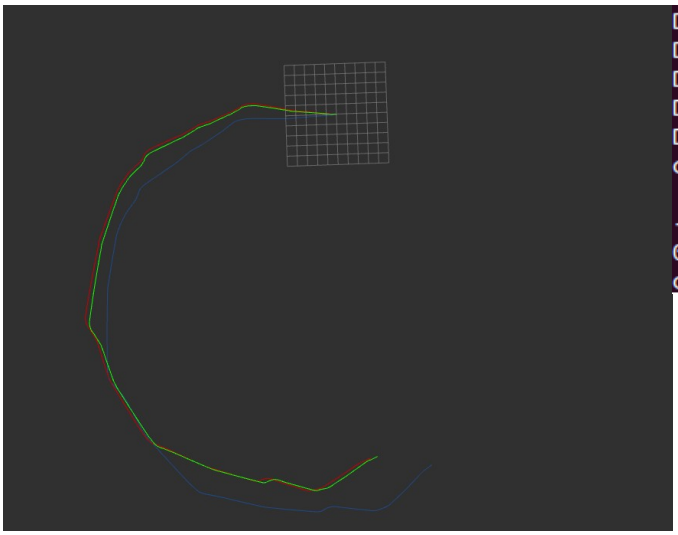
```cpp
/*
输入:里程计和激光数据
TODO:
构建最小二乘需要的超定方程组
Ax = b
*/
bool OdomCalib::Add_Data(Eigen::Vector3d Odom,Eigen::Vector3d scan)
{

    if(now_len<INT_MAX)
    {
        //TODO: 构建超定方程组

        // matrix.block(i,j,p,q); is represented for Block of size (p,q), starting at (i,j)   (alternative: matrix.block<p,q>(i,j);)
        A.block( startRow: now_len * 3,  startCol: 0,  blockRows: 1,  blockCols: 3) = Odom.transpose();
        A.block( startRow: now_len * 3 + 1,  startCol: 3,  blockRows: 1,  blockCols: 3) = Odom.transpose();
        A.block( startRow: now_len * 3 + 2,  startCol: 6,  blockRows: 1,  blockCols: 3) = Odom.transpose();

        // vector.segment(i,n); is represented for Block containing n elements, starting at position i
        b.segment( start: now_len * 3,  n: 3) = scan;

        //end of TODO
        now_len++;
        return true;
    }
    else
```

接下来在函数 Eigen::Matrix3d OdomCalib::Solve()中，解最小二乘方程: $A^TAx = A^Tb$。
我们可以直接求 A^TA 的逆来解该方程，但由于病态特性可能产生数值计算误差。所以一般我们使用 QR 分解来解该方程，这里使用 Eigen 提供的函数 colPivHouseholderQr().solve(), 该方法不要求矩阵一定要是正定或半正定的。这里 correct_vector 是所求变换关系的 9x1 向量表示形式，返回的 correct_matrix 是 3x3 矩阵表示形式。

CMakeLists.txt | main.cpp | Odom_Calib.cpp

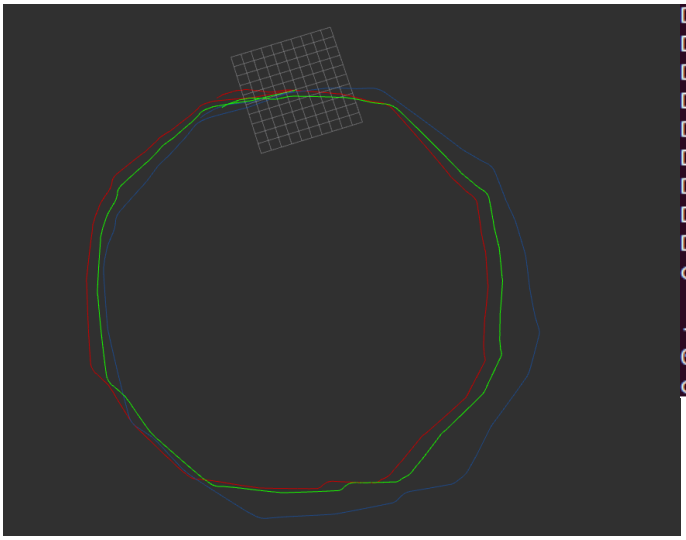
```cpp
/*
 * TODO:
 * 求解线性最小二乘Ax=b
 * 返回得到的矫正矩阵
*/
Eigen::Matrix3d OdomCalib::Solve()
{
    Eigen::Matrix3d correct_matrix;
    //TODO: 求解线性最小二乘

    // 1. 根据最小二乘公式，直接求逆
    // Eigen::Matrix<double, 9, 1> correct_vector = (A.transpose() * A).inverse() * A.transpose() * b;

    // 2. 改进的Cholesky分解法 ldlt() 要求矩阵半正定: 这里无法使用！
    // Eigen::Matrix<double, 9, 1> correct_vector = (A.transpose() * A).ldlt().solve(A.transpose() * b);

    // 3. QR分解法 colPivHouseholderQr()对矩阵无要求
    Eigen::Matrix<double, 9, 1> correct_vector = (A.transpose() * A).colPivHouseholderQr().solve( b: A.transpose() * b);

    correct_matrix << correct_vector( row: 0,  col: 0), correct_vector( row: 1,  col: 0), correct_vector( row: 2,  col: 0),
                      correct_vector( row: 3,  col: 0), correct_vector( row: 4,  col: 0), correct_vector( row: 5,  col: 0),
                      correct_vector( row: 6,  col: 0), correct_vector( row: 7,  col: 0), correct_vector( row: 8,  col: 0);
    //end of TODO
    return correct_matrix;
}
```

编译运行节点后，启动 rviz, 播放 bag 数据，结果如下:
当我们只标定前 1473 个数据时，标定结果看起来更为准确:



```
Data Cnt:1469
Data Cnt:1470
Data Cnt:1471
Data Cnt:1472
Data Cnt:1473
correct_matrix:
  0.936829    1.21574 -0.0279553
-0.0131336    10.3754  0.0702394
0.00101879    32.5964   0.171031
calibration over!!!!
```

当我们标定前 2541 个数据时，标定结果看起来更为反而更不准确了:



```
Data Cnt:2533
Data Cnt:2534
Data Cnt:2535
Data Cnt:2536
Data Cnt:2537
Data Cnt:2538
Data Cnt:2539
Data Cnt:2540
Data Cnt:2541
correct_matrix:
  0.936025     2.1271 -0.0493379
-0.0129118    6.77184   0.154067
0.00129784    22.6873   0.400548
calibration over!!!!
```

2. 补充基于模型方法的里程计标定模块代码;

代码如下:

先是解第一个最小二乘:

CMakeLists.txt | odom_calib.cpp | odom.txt | scan_match.txt

```
++id_r;      // 对于同一个 id_s 可递增多次 id_r, 直到 r_t >= s_t, 才递增一次 id_s
// 在2帧激光匹配时间内进行里程计角度积分
if (r_t < s_t)
{
    // 对于一开始的数据： 对里程计数据进行积分，直到离激光数据的时间戳最近的前一个里程计数据的时间戳
    double dt = r_t - last_rt;
    w_Lt += w_L * dt;
    w_Rt += w_R * dt;
    last_rt = r_t;
}
else
{
    // 对于后续的数据： 将上一个里程计数据的时间戳被设定为上一个激光数据的时间戳，从而计算积分值
    double dt = s_t - last_rt;
    w_Lt += w_L * dt;
    w_Rt += w_R * dt;
    last_rt = s_t;
    // 填充A, b矩阵
    //TODO: (3~5 lines)
    A.row( i: id_s) = Eigen::Vector2d( x: w_Lt,  y: w_Rt).transpose();
    b( index: id_s) = s_th;
    //end of TODO
    w_Lt = 0;
    w_Rt = 0;
    ++id_s;      // 只在该else条件内 递增激光数据的时间戳
}
```

CMakeLists.txt | odom_calib.cpp | odom.txt | scan_match.txt

```
// 进行最小二乘求解
Eigen::Vector2d J21J22;

//TODO: (1~2 lines)
// 直接求逆
// J21J22 = (A.transpose() * A).inverse() * A.transpose() * b;
// 使用RQ分解求解
J21J22 = (A.transpose() * A).colPivHouseholderQr().solve( b: A.transpose() * b);
//end of TODO

const double &J21 = J21J22( index: 0);
const double &J22 = J21J22( index: 1);
cout << "J21: " << J21 << endl;
cout << "J22: " << J22 << endl;
```

再解第二个最小二乘:

CMakeLists.txt | odom_calib.cpp | odom.txt | scan_match.txt

```
138         // 在2帧激光匹配时间内进行里程计位置积分
139         if (r_t < s_t)
140         {
141             double dt = r_t - last_rt;
142             cx += 0.5 * (-J21 * w_L * dt + J22 * w_R * dt) * cos( x: th);
143             cy += 0.5 * (-J21 * w_L * dt + J22 * w_R * dt) * sin( x: th);
144             th += (J21 * w_L + J22 * w_R) * dt;
145             last_rt = r_t;
146         }
147         else
148         {
149             double dt = s_t - last_rt;
150             cx += 0.5 * (-J21 * w_L * dt + J22 * w_R * dt) * cos( x: th);
151             cy += 0.5 * (-J21 * w_L * dt + J22 * w_R * dt) * sin( x: th);
152             th += (J21 * w_L + J22 * w_R) * dt;
153             last_rt = s_t;
154             // 填充C, S矩阵
155             //TODO: (4~5 lines)
156             C.segment( start: id_s * 2,  n: 2) = Eigen::Vector2d( x: cx,  y: cy).transpose();
157             S.segment( start: id_s * 2,  n: 2) = Eigen::Vector2d( x: s_x,  y: s_y).transpose();
158             //end of TODO
159             cx = 0;
160             cy = 0;
161             th = 0;
162             ++id_s;
163         }
```

CMakeLists.txt | odom_calib.cpp | odom.txt | scan_match.txt

```
165     // 进行最小二乘求解，计算b_wheel, r_L, r_R
166     double b_wheel;
167     Eigen::Matrix<double, 1, 1> b_wheel_matrix;
168     double r_L;
169     double r_R;
170     //TODO: (3~5 lines)
171     // 直接求逆
172     // b_wheel = (C.transpose() * C).inverse() * C.transpose() * S;
173     // 使用QR分解
174     b_wheel_matrix = (C.transpose() * C).colPivHouseholderQr().solve( b: C.transpose() * S);
175     b_wheel = b_wheel_matrix( row: 0,  col: 0);
176
177     r_L = -J21 * b_wheel;
178     r_R = J22 * b_wheel;
179     //end of TODO
180
181     cout << "b_wheel: " << b_wheel << endl;
182     cout << "r_L: " << r_L << endl;
183     cout << "r_R: " << r_R << endl;
184
185     cout << "参考答案：轮间距b为0.6m左右，两轮半径为0.1m左右" << endl;
```

计算结果如下，与参考结果一致:

```
odom_calib
/home/jindong/Lidar_SLAM/Exercise/ch2/HW2/odom_calib/odom_calib
J21: -0.163886
J22: 0.170575
b_wheel: 0.59796
r_L: 0.0979974
r_R: 0.101997
参考答案：轮间距b为0.6m左右，两轮半径为0.1m左右
```

3. 通过互联网总结学习线性方程组 Ax=b 的求解方法,回答以下问题:(2 分)

(1)对于该类问题,你都知道哪几种求解方法?

一般可以使用直接求逆法，Cholesky 分解法，QR 分解法, 奇异值分解法等等。

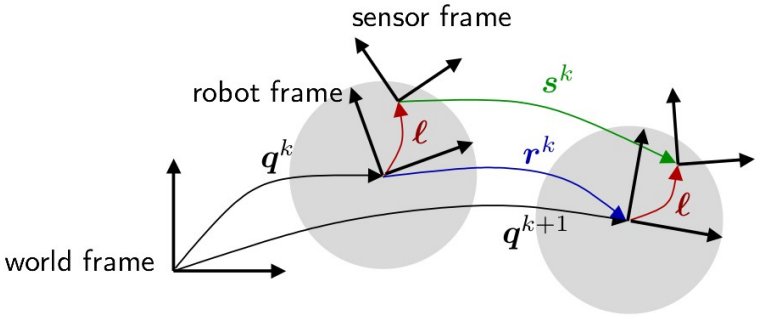
(2)各方法的优缺点有哪些?分别在什么条件下较常被使用?

一般直接求逆要求矩阵可逆，而且计算病态矩阵时容易产生数值计算精度问题，Cholesky 分解一般要求正定或半正定矩阵，QR 分解和奇异值分解适用于所有情况。对应这些方法，Eigen 库提供了以下函数:

Decomposition	Method	Requirements on the matrix	Speed (small-to-medium)	Speed (large)	Accuracy
PartialPivLU	partialPivLu()	Invertible	++	++	+
FullPivLU	fullPivLu()	None	-	--	+++
HouseholderQR	householderQr()	None	++	++	+
ColPivHouseholderQR	colPivHouseholderQr()	None	+	-	+++
FullPivHouseholderQR	fullPivHouseholderQr()	None	-	--	+++
CompleteOrthogonalDecomposition	completeOrthogonalDecomposition()	None	+	-	+++
LLT	llt()	Positive definite	+++	+++	+
LDLT	ldlt()	Positive or negative semidefinite	+++	+	++
BDCSVD	bdcSvd()	None	-	-	+++
JacobiSVD	jacobiSvd()	None	-	---	+++

4. 简答题,开放性答案:设计里程计与激光雷达外参标定方法。(2 分)
我们一般把传感器内自身要调节的参数称为内参,比如前面作业中里程计模型的两轮间距与两个轮子的半径。把传感器之间的信息称为外参,比如里程计与激光雷达之间的时间延迟,位姿变换等。请你选用直接线性方法或基于模型的方法,设计一套激光雷达与里程计外参的标定方法,并回答以下问题:

(1)你设计的方法是否存在某些假设?基于这些假设下的标定观测值和预测值分别是什么?
假设激光观测数据是足够精确的，可以用来标定里程计内参以及激光和里程计之间的外参。我们可以将如下论文[1]中坐标轴表示中的 robot frame 看作是激光雷达的坐标轴，将 sensor frame 看作是里程计的坐标轴。



里程计相对于激光雷达的变换关系为:

$$\boldsymbol{\ell} \;=\; (\ell_x, \ell_y, \ell_\theta) \;\in\; \mathrm{SE}(2)$$

观测值是激光雷达的匹配增量值，即激光雷达前后两帧之间的平移和旋转增量，记作 $\hat{\boldsymbol{s}}^k$。
预测值是将内参和外参:

$$r_\mathrm{L}, r_\mathrm{R}, b, \ell_x, \ell_y, \ell_\theta$$

带入预测方程后得到的值，预测方程可以写作:

$$\boldsymbol{s}^k = \ominus\boldsymbol{\ell} \oplus \boldsymbol{r}^k(r_\mathrm{L}, r_\mathrm{R}, b) \oplus \boldsymbol{\ell}$$

其中 r^k 表示里程计前后两帧的增量表示，s^k 表示通过里程计增量位姿和外参 l 预测出的激光增量位姿。其中 SE(2)的运算符号含义如下:

$\oplus, \ominus$ “$\oplus$” is the group operation on $\mathrm{SE}(2)$:

$$\begin{pmatrix} a_x \\ a_y \\ a_\theta \end{pmatrix} \oplus \begin{pmatrix} b_x \\ b_y \\ b_\theta \end{pmatrix} = \begin{pmatrix} a_x + b_x\cos(a_\theta) - b_y\sin(a_\theta) \\ a_y + b_x\sin(a_\theta) + b_y\cos(a_\theta) \\ a_\theta + b_\theta \end{pmatrix}$$

“$\ominus$” is the group inverse:

$$\ominus \begin{pmatrix} a_x \\ a_y \\ a_\theta \end{pmatrix} = \begin{pmatrix} -a_x\cos(a_\theta) - a_y\sin(a_\theta) \\ +a_x\sin(a_\theta) - a_y\cos(a_\theta) \\ -a_\theta \end{pmatrix}$$

(2)如何构建你的最小二乘方程组求解该外参?

可以构建如下最小二乘方程组, 左边为观测值，右边为预测值:

$$\hat{\boldsymbol{s}}^k = \ominus\boldsymbol{\ell} \oplus \boldsymbol{r}^k(r_{\mathrm{L}}, r_{\mathrm{R}}, b) \oplus \boldsymbol{\ell}$$

我们也可以根据论文[1]构建如下损失函数:

$$\mathcal{J} = -\frac{1}{2}\sum_{k=1}^{n} ||\hat{\boldsymbol{s}}^k - \ominus\boldsymbol{\ell} \oplus \boldsymbol{r}^k(r_{\mathrm{L}}, r_{\mathrm{R}}, b) \oplus \boldsymbol{\ell}||^2_{\boldsymbol{\Sigma}_k^{-1}}$$

$\boldsymbol{\Sigma}_k$表示预测值的方差，用来构建带权重的最小二乘。通过闭式求解或迭代优化可以求出内参和外参。

参考文献:

[1] Simultaneous calibration of odometry and sensor parameters for mobile robots

https://github.com/AndreaCensi/calibration