GTSAM 库用于 2D &3D PoseSLAM

1. PoseSLAM 简介

位姿 SLAM（PoseSLAM）是同时定位与地图构建（SLAM）的变体，只优化机器人的**位姿**，而并不显式地对外部环境构建地图。SLAM 的目标是，在给定传感器观测数据的同时，对机器人进行定位，并且对环境的地图进行构建。除了轮式里程计，对于在平面上移动的机器人最常用的传感器是平面激光雷达。它既能在连续姿态之间提供里程计约束，也能在机器人重新访问之前探索过的环境部分时提供回环约束。由于激光雷达的数据量比较庞大，对一个稠密的三维环境地图显式地进行优化并不现实。而如果能够重建机器人位姿的轨迹，通过简单地将所有的激光雷达测量值重投影到与第一个位姿对齐的坐标系中，就可以得到一个稠密的三维地图。

1.1 位姿表示

二维旋转与二维刚体变换

二维旋转通常有三种表示形式：角度、复数、（2x2）旋转矩阵。 下式(1)

$$\theta \to \cos\theta + \mathrm{i}\sin\theta \leftrightarrow \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

式中第一个箭头指示一个（我们不希望的）多对一的映射。

$SO(2)$：表示机器人的二维旋转。

$SE(2)$：表示机器人的二维位姿，**相对位姿**也是 $SE(2)$中的元素。

三维旋转与三维刚体变换

三维旋转对应有四种表现形式：轴-角、单位四元数、（3x3）旋转矩阵、欧拉角。下式(2)

$$(\bar{\omega},\theta) \leftrightarrow \cos\frac{\theta}{2} + (\bar{\omega}_x\mathrm{i} + \bar{\omega}_y\mathrm{j} + \bar{\omega}_z\mathrm{k})\sin\frac{\theta}{2} \rightrightarrows \boldsymbol{R} \leftarrow \phi,\theta,\psi$$

式中，轴-角到$SO(3)$因为标量角度 θ，存在多对一映射；单位四元数存在 q和$-q$代表同一个旋转，到$SO(3)$存在二对一的映射；欧拉角因为奇异性，到旋转矩阵存在多对一的映射。

1.2 局部位姿坐标

解决了二维和三维的位姿表示问题后，接下来就是如何在二维或三维的位姿上进行优化。与处理旋转矩阵的过程一样，优化解变成了一个局部参数化的过程，使用指数映射将每次迭代的增量更新转换到位姿流形上。在的情况下，通常会考虑变化率，以及通过乘以一个有限的时间量 $\Delta\tau$得到的一个增量 ξ。特别地，定义**角速度** w，**平移速度** v，并将 ξ定义为，下式(3)

$$\xi \stackrel{\text{def}}{=\!=} \begin{bmatrix} \omega \\ v \end{bmatrix} \Delta\tau$$

对于二维或者三维的情况，都可以再次使用指数映射来讲局部位姿坐标 ξ映射到一个初始位姿估计 $x0$附近的值。下式(4)

$$x_0 \oplus \xi \stackrel{\text{def}}{=\!=} x_0 \cdot \exp \hat{\xi}$$

1.3 位姿优化

首先考虑一个在单独的位姿 $x \in SE(2)$或$x \in SE(3)$上简单的最小化问题。下式(5)

$$x^* = \arg\min_x \|h(x) - z\|_\Sigma^2$$

而为了最小化这个目标函数，需要表示非线性测量方程是如何在基位姿附近运作。下式(6)

$$h(x_0 \oplus \xi) \approx h(x_0) + H_0 \xi$$

当有了近似公式后，就可以最小化关于局部坐标的目标函数：下式(7)

$$\xi^* = \operatorname{argmin}_\xi \|h(x_0) + H_0\xi - z\|_\Sigma^2$$

1.4 位姿 SLAM

在位姿 SLAM 中，当然不希望只能不恢复一个位姿，而是希望恢复整个轨迹中的所有位姿。一般来讲，位姿 SLAM 中会涉及两种类型的因子：一元因子，如位姿先验或（例如从 GPS 中获得的）绝对位姿测量；以及二元因子（例如从激光雷达得到的相对位姿约束）。

最终优化将-在每一次迭代中-通过对下面的线性化观测因子的和进行最小化，来求解所有位姿的局部坐标。

2. 2D PoseSLAMexample

2.1 带回环

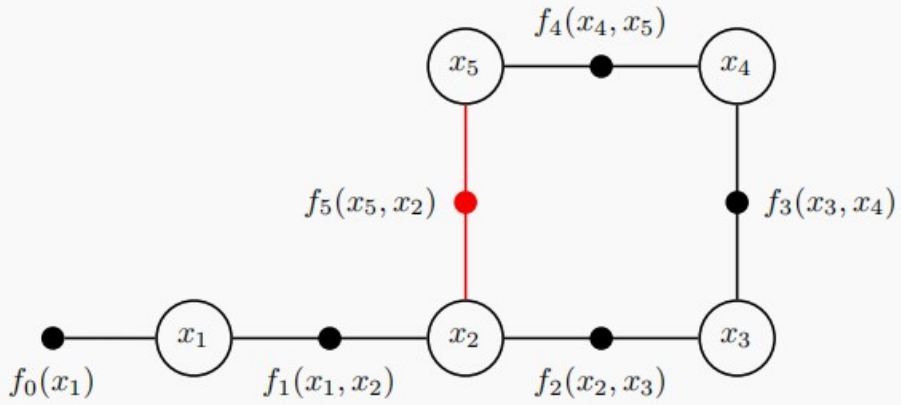

Figure 6: Factor graph for PoseSLAM.

案例说明：机器人由原点 **x1**（0，0）→ **x2**（2，0）→ **x3**（4，0）→ **x4**（4，2）→ **x5**（2，2）→ **x2**（2，0）形成一个闭环轨迹，给定一组与之对应且有误差的观测值，通过因子图优化求解出估计值并与真值作比较。

	(PriorFactor、BetweenFactor)	(ExpressionFactor)	高斯牛顿法	LM法	time（ms）	error
Pose2SLAMExample.cpp	✔		✔		2.361532	e-20
Pose2SLAMExample_graphviz.cpp	✔			✔	4.2087	e-13
Pose2SLAMwSPCG.cpp	✔			✔		
Pose2SLAMExampleExpression.cpp		✔	✔		1.62113	e-22

A. 构建因子图（Graph）

① 非线性因子图（NonlinearFactorGraph）

```
C++
// 1. Create a factor graph container and add factors to it
  NonlinearFactorGraph graph;
```

② 表达式因子图（ExpressionFactorGraph）

自动微分表达式因子图框架：GTSAM 库提供的一种简单的方法来描述状态、噪声和因子。使用自动微分表达式，可以更轻松地实现导数的计算，从而使优化更有效。

```
C++
// 1. Create a factor graph container and add factors
to it
  ExpressionFactorGraph graph;
```

B. 添加因子（Factor）

①添加一元因子（PriorFactor）和二元因子（BetweenFactor）

```
C++
// 2a. Add a prior on the first pose, setting it to
the origin
  // A prior factor consists of a mean and a noise
model (covariance matrix)
  noiseModel::Diagonal::shared_ptr priorNoise =
noiseModel::Diagonal::Sigmas(Vector3(0.3, 0.3, 0.1));
  graph.emplace_shared<PriorFactor<Pose2> >(1,
Pose2(0, 0, 0), priorNoise);

  // For simplicity, we will use the same noise model
for odometry and loop closures
  noiseModel::Diagonal::shared_ptr model =
noiseModel::Diagonal::Sigmas(Vector3(0.2, 0.2, 0.1));

  // 2b. Add odometry factors
  // Create odometry (Between) factors between
consecutive poses
  graph.emplace_shared<BetweenFactor<Pose2> >(1, 2,
Pose2(2, 0, 0     ), model);
  graph.emplace_shared<BetweenFactor<Pose2> >(2, 3,
Pose2(2, 0, M_PI_2), model);
  graph.emplace_shared<BetweenFactor<Pose2> >(3, 4,
Pose2(2, 0, M_PI_2), model);
  graph.emplace_shared<BetweenFactor<Pose2> >(4, 5,
Pose2(2, 0, M_PI_2), model);

  // 2c. Add the loop closure constraint
  // This factor encodes the fact that we have
returned to the same pose. In real systems,
  // these constraints may be identified in many
ways, such as appearance-based techniques
  // with camera images. We will use another Between
```

```
Factor to enforce this constraint:
  graph.emplace_shared<BetweenFactor<Pose2> >(5, 2,
Pose2(2, 0, M_PI_2), model);
  graph.print("\nFactor Graph:\n"); // print
```

②添加 ExpressionFactor 因子

```
C++
// Create Expressions for unknowns
  Pose2_ x1(1), x2(2), x3(3), x4(4), x5(5);

  // 2a. Add a prior on the first pose, setting it to
the origin
  noiseModel::Diagonal::shared_ptr priorNoise =
noiseModel::Diagonal::Sigmas(Vector3(0.3, 0.3, 0.1));
  graph.addExpressionFactor(x1, Pose2(0, 0, 0),
priorNoise);

  // For simplicity, we will use the same noise model
for odometry and loop closures
  noiseModel::Diagonal::shared_ptr model =
noiseModel::Diagonal::Sigmas(Vector3(0.2, 0.2, 0.1));

  // 2b. Add odometry factors
  graph.addExpressionFactor(between(x1,x2), Pose2(2,
0, 0     ), model);
  graph.addExpressionFactor(between(x2,x3), Pose2(2,
0, M_PI_2), model);
  graph.addExpressionFactor(between(x3,x4), Pose2(2,
0, M_PI_2), model);
  graph.addExpressionFactor(between(x4,x5), Pose2(2,
0, M_PI_2), model);

  // 2c. Add the loop closure constraint
  graph.addExpressionFactor(between(x5,x2), Pose2(2,
0, M_PI_2), model);
  graph.print("\nFactor Graph:\n"); // print
```

C. 添加初始估计值（Value）

```
C++
// 3. Create the data structure to hold the
initialEstimate estimate to the solution
  // For illustrative purposes, these have been
deliberately set to incorrect values
  Values initialEstimate;
```

```
  initialEstimate.insert(1, Pose2(0.5, 0.0,
0.2   ));
  initialEstimate.insert(2, Pose2(2.3, 0.1, -
0.2   ));
  initialEstimate.insert(3, Pose2(4.1, 0.1,
M_PI_2));
  initialEstimate.insert(4, Pose2(4.0, 2.0,
M_PI  ));
  initialEstimate.insert(5, Pose2(2.1, 2.1, -
M_PI_2));
  initialEstimate.print("\nInitial Estimate:\n"); //
print
```

D. 初始化优化求解器并求解（Optimizer）

① 高斯牛顿法

```
C++
// 4. Optimize the initial values using a Gauss-
Newton nonlinear optimizer
  // The optimizer accepts an optional set of
configuration parameters,
  // controlling things like convergence criteria,
the type of linear
  // system solver to use, and the amount of
information displayed during
  // optimization. We will set a few parameters as a
demonstration.
  GaussNewtonParams parameters;
  // Stop iterating once the change in error between
steps is less than this value
  parameters.relativeErrorTol = 1e-5;
  // Do not perform more than N iteration steps
  parameters.maxIterations = 100;
  // Create the optimizer ...
  GaussNewtonOptimizer optimizer(graph,
initialEstimate, parameters);
  // ... and optimize
  Values result = optimizer.optimize();
  result.print("Final Result:\n");
```

② LM 法

```
C++
// 4. Single Step Optimization using Levenberg-
Marquardt
```

```
  LevenbergMarquardtParams parameters;
  parameters.verbosity =
NonlinearOptimizerParams::ERROR;
  parameters.verbosityLM =
LevenbergMarquardtParams::LAMBDA;

  // LM is still the outer optimization loop, but by
specifying "Iterative" below
  // We indicate that an iterative linear solver
should be used.
  // In addition, the *type* of the iterativeParams
decides on the type of
  // iterative solver, in this case the SPCG
(subgraph PCG)
  parameters.linearSolverType =
NonlinearOptimizerParams::Iterative;
  parameters.iterativeParams =
boost::make_shared<SubgraphSolverParameters>();

  LevenbergMarquardtOptimizer optimizer(graph,
initialEstimate, parameters);
  Values result = optimizer.optimize();
  result.print("Final Result:\n");
```

E. 打印优化结果

```
C++
// 5. Calculate and print marginal covariances for
all variables
  cout.precision(3);
  Marginals marginals(graph, result);
  cout << "x1 covariance:\n" <<
marginals.marginalCovariance(1) << endl;
  cout << "x2 covariance:\n" <<
marginals.marginalCovariance(2) << endl;
  cout << "x3 covariance:\n" <<
marginals.marginalCovariance(3) << endl;
  cout << "x4 covariance:\n" <<
marginals.marginalCovariance(4) << endl;
  cout << "x5 covariance:\n" <<
marginals.marginalCovariance(5) << endl;
```

2.2 noisyToyGragh.txt

案例说明：该 noisyToyGragh.txt 为 g2o 格式的数据文件内容如下：

```
C++
VERTEX_SE2 0 0.000000 0.000000 0.000000
VERTEX_SE2 1 0.774115 1.183389 1.576173
VERTEX_SE2 2 -0.262420 2.047059 -3.127594
VERTEX_SE2 3 -1.605649 0.993891 -1.561134
EDGE_SE2 0 1 0.774115 1.183389 1.576173 1.000000 0.000000
0.000000 1.000000 0.000000 1.000000
EDGE_SE2 1 2 0.869231 1.031877 1.579418 1.000000 0.000000
0.000000 1.000000 0.000000 1.000000
EDGE_SE2 2 3 1.357840 1.034262 1.566460 1.000000 0.000000
0.000000 1.000000 0.000000 1.000000
EDGE_SE2 2 0 0.303492 1.865011 -3.113898 1.000000 0.000000
0.000000 1.000000 0.000000 1.000000
EDGE_SE2 0 3 -0.928526 0.993695 -1.563542 1.000000 0.000000
0.000000 1.000000 0.000000 1.000000
```

A. Pose2SLAMExample_g2o.cpp

读取 g2o 文件数据后，使用 NonlinearFactorGraph 和高斯牛顿法求解。

B. Pose2SLAMExample_lago.cpp

读取 g2o 文件数据后，同样构建 NonliearFactorGraph，但是使用线性逼近（Linear Approximation for Graph Optimization）的方法求解。

	time（ms）	error
Pose2SLAMExample_g2o.cpp	2.05807	e-23
Pose2SLAMExample_lago.cpp	1.617291	e-21

2.3 w100.graph

案例说明：读取 100 个 2D 位姿的数据，构建 NonlinearFactorGraph 并使用 LM 法求解。

	time（ms）	error
Pose2SLAMExample_graph.cpp	13.0231	e-15

2.4 N 个关联位姿

案例说明：本 cpp 为压力测试，测试 gtsam 库的运行效率上限?

3. 3D PoseSLAMexample

3.1 pose3example.txt

案例说明：读取 pose3example.txt 中的 5 个位姿数据和协方差矩阵

```
C++
VERTEX_SE3:QUAT 0 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 1.000000
VERTEX_SE3:QUAT 1 1.001367 0.015390 0.004948 0.190253
0.283162 -0.392318 0.854230
VERTEX_SE3:QUAT 2 1.993500 0.023275 0.003793 -0.351729 -
0.597838 0.584174 0.421446
VERTEX_SE3:QUAT 3 2.004291 1.024305 0.018047 0.331798 -
0.200659 0.919323 0.067024
VERTEX_SE3:QUAT 4 0.999908 1.055073 0.020212 -0.035697 -
0.462490 0.445933 0.765488
EDGE_SE3:QUAT 0 1   1.001367 0.015390 0.004948   0.190253
0.283162 -0.392318 0.854230   10000.000000 0.000000 0.000000
0.000000 0.000000 0.000000   10000.000000 0.000000 0.000000
0.000000 0.000000   10000.000000 0.000000 0.000000 0.000000
10000.000000 0.000000 0.000000   10000.000000 0.000000
10000.000000
EDGE_SE3:QUAT 1 2   0.523923 0.776654 0.326659   0.311512
0.656877 -0.678505 0.105373   10000.000000 0.000000 0.000000
0.000000 0.000000 0.000000   10000.000000 0.000000 0.000000
0.000000 0.000000   10000.000000 0.000000 0.000000 0.000000
10000.000000 0.000000 0.000000   10000.000000 0.000000
10000.000000
EDGE_SE3:QUAT 2 3   0.910927 0.055169 -0.411761   0.595795 -
0.561677 0.079353 0.568551   10000.000000 0.000000 0.000000
0.000000 0.000000 0.000000   10000.000000 0.000000 0.000000
0.000000 0.000000   10000.000000 0.000000 0.000000 0.000000
10000.000000 0.000000 0.000000   10000.000000 0.000000
10000.000000
EDGE_SE3:QUAT 3 4   0.775288 0.228798 -0.596923   -0.592077
0.303380 -0.513226 0.542221   10000.000000 0.000000 0.000000
0.000000 0.000000 0.000000   10000.000000 0.000000 0.000000
0.000000 0.000000   10000.000000 0.000000 0.000000 0.000000
10000.000000 0.000000 0.000000   10000.000000 0.000000
10000.000000
EDGE_SE3:QUAT 1 4   -0.577841 0.628016 -0.543592   -0.125250
-0.534379 0.769122 0.327419   10000.000000 0.000000 0.000000
0.000000 0.000000 0.000000   10000.000000 0.000000 0.000000
```

```
0.000000 0.000000   10000.000000 0.000000 0.000000 0.000000
10000.000000 0.000000 0.000000   10000.000000 0.000000
10000.000000
EDGE_SE3:QUAT 3 0   -0.623267 0.086928 0.773222   0.104639
0.627755 0.766795 0.083672   10000.000000 0.000000 0.000000
0.000000 0.000000 0.000000   10000.000000 0.000000 0.000000
0.000000 0.000000   10000.000000 0.000000 0.000000 0.000000
10000.000000 0.000000 0.000000   10000.000000 0.000000
10000.000000
```

A. 高斯牛顿法

Gauss-Newton 优化器比 LM 优化器更快但更不稳定。Gauss-Newton 只在当前估计值是优化目标函数的局部极小值点时才能保证快速收敛。而当当前估计值不在局部极小值点附近时，Gauss-Newton 可能会发散。

相比之下，LM 优化器的性能更加稳定，因为它使用一种自适应的方式来控制步长大小，从而可以在不同位置的优化问题中获得更好的收敛性能。

因此，如果计算速度是最重要的因素，可以选择 Gauss-Newton 优化器；如果稳定性更重要，可以选择 LM 优化器。当然，具体选择哪种优化器还取决于问题的特性和优化的目标。

B. 梯度下降法（黎曼梯度）

黎曼梯度法可以用于处理优化问题，其中因子类型和变量类型可以是几何、李群、李代数或其他非欧几里德类型。在 gtsam 中，可以使用带有 Riemannian（黎曼）后缀的因子类型和变量类型，例如 `Rot3` 和 `Rot3::Jacobian`。这些类型支持黎曼梯度法，并且在 `LevenbergMarquardtOptimizer`、`GaussNewtonOptimizer` 和 `DoglegOptimizer` 中均可用。

C. 弦松弛法

GTSAM 库中提供了弦松弛算法（Chordal Relaxation）来处理大规模稀疏矩阵，该算法基于稀疏图的特殊结构，可以在不牺牲优化精度的情况下大幅降低计算成本。

接下来，可以使用 gtsam 提供的 CholeskyFactor 类和 NonlinearFactorGraph 类定义优化问题的因子图。CholeskyFactor 类可以自动检测稀疏性，根据稀疏性采用不同的算法来计算分解因子。

在优化完成后，可以使用 Marginals 类计算各个变量的边缘协方差矩阵。

需要注意的是，弦松弛算法不适用于所有的稀疏矩阵，只有在矩阵具有一定的结构性时才能取得优秀的计算效果。因此，在使用弦松弛算法之前，需要先评估矩阵结构，以决定是否适用弦松弛算法来优化问题。

	time（ms）	error
Pose3SLAMExample_g2o.cpp	4.085662	e-09
Pose3SLAMExample_initializePose3Gradient.cpp	2.76227	e-03
Pose3SLAMExample_initializePose3Chordal.cpp	2.67711	e-12

3.2 Pose3SLAMExampleExpressions_BearingRangeWithTransform.cpp

案例说明：该 cpp 文件使用 ExpressionFactorGraph 表达式因子图，创建机器人的运动轨迹和一些点的位置，同时指定一些噪声模型，添加包括初始姿态先验、每个位姿的 BearingRange 因子、位姿之间的因子。

	time（ms）	error
Pose3SLAMExampleExpressions_BearingRangeWithTransform.cpp	17.7804	e-25

3.3 Pose3Localization.cpp

暂时没法运行，涉及 values 类的新成员函数，以及一些智能指针的应用。

4. 总结与分析