GTSAM用于机器人定位

常用定位因子——BetweenFactor

在SLAM中，常用**BetweenFactor**定义帧间里程计的约束（LIO-SAM）：

```
noiseModel::Diagonal::shared_ptr odometryNoise = noiseModel::Diagonal::Variances
gtsam::Pose3 poseFrom = pclPointTogtsamPose3(cloudKeyPoses6D->points.back());
gtsam::Pose3 poseTo   = trans2gtsamPose(transformTobeMapped);
gtSAMgraph.add(BetweenFactor<Pose3>(cloudKeyPoses3D->size()-1, cloudKeyPoses3D->
initialEstimate.insert(cloudKeyPoses3D->size(), poseTo);
```

BetweenFactor的定义：

```
template<class VALUE>
  class BetweenFactor: public NoiseModelFactorN<VALUE, VALUE> {

    // Check that VALUE type is a testable Lie group
    BOOST_CONCEPT_ASSERT((IsTestable<VALUE>));
    BOOST_CONCEPT_ASSERT((IsLieGroup<VALUE>));

  public:

    typedef VALUE T;

  private:

    typedef BetweenFactor<VALUE> This;
    typedef NoiseModelFactorN<VALUE, VALUE> Base;

    VALUE measured_; /** The measurement */

  public:

    // Provide access to the Matrix& version of evaluateError:
    using Base::evaluateError;

    // shorthand for a smart pointer to a factor
    typedef typename std::shared_ptr<BetweenFactor> shared_ptr;

```

```
  /// @name Standard Constructors
  /// @{

  /** default constructor - only use for serialization */
  BetweenFactor() {}

  /** Constructor */
  BetweenFactor(Key key1, Key key2, const VALUE& measured,
      const SharedNoiseModel& model = nullptr) :
    Base(model, key1, key2), measured_(measured) {
  }
```

GTSAM自定义因子

在GTSAM中，可以通过内置类**NoiseModelFactorN<T>**派生一个新类，来创建自定义的N元因子。

在**LocalizationExample.cpp**示例中，创建了一个二元因子，它实现了类似于二元GPS因子的效果：

```
class UnaryFactor: public NoiseModelFactorN<Pose2> {
  // The factor will hold a measurement consisting of an (X,Y) location
  // We could this with a Point2 but here we just use two doubles
  double mx_, my_;

 public:

  // Provide access to Matrix& version of evaluateError:
  using NoiseModelFactor1<Pose2>::evaluateError;

  /// shorthand for a smart pointer to a factor
  typedef std::shared_ptr<UnaryFactor> shared_ptr;

  // The constructor requires the variable key, the (X, Y) measurement value, ar
  UnaryFactor(Key j, double x, double y, const SharedNoiseModel& model):
    NoiseModelFactorN<Pose2>(model, j), mx_(x), my_(y) {}

  ~UnaryFactor() override {}

  // Using the NoiseModelFactorN base class there are two functions that must be
  // The first is the 'evaluateError' function. This function implements the des
  // function, returning a vector of errors when evaluated at the provided varia
  // must also calculate the Jacobians for this measurement function, if request
  Vector evaluateError(const Pose2& q, OptionalMatrixType H) const override {
    // The measurement function for a GPS-like measurement h(q) which predicts t
```

```
    // The error is then simply calculated as E(q) = h(q) - m:
    // error_x = q.x - mx
    // error_y = q.y - my
    // Node's orientation reflects in the Jacobian, in tangent space this is equ
    // H =  [ cos(q.theta)  -sin(q.theta) 0 ]
    //      [ sin(q.theta)   cos(q.theta) 0 ]
    const Rot2& R = q.rotation();
    if (H) (*H) = (gtsam::Matrix(2, 3) << R.c(), -R.s(), 0.0, R.s(), R.c(), 0.0)
    return (Vector(2) << q.x() - mx_, q.y() - my_).finished();
  }

  // The second is a 'clone' function that allows the factor to be copied. Under
  // circumstances, the following code that employs the default copy constructor
  // work fine.
  gtsam::NonlinearFactor::shared_ptr clone() const override {
    return std::static_pointer_cast<gtsam::NonlinearFactor>(
        gtsam::NonlinearFactor::shared_ptr(new UnaryFactor(*this))); }

  // Additionally, we encourage you the use of unit testing your custom factors,
  // (as all GTSAM factors are), in which you would need an equals and print, tc
  // GTSAM_CONCEPT_TESTABLE_INST(T) defined in Testable.h, but these are not nee
};  // UnaryFactor
```

类中定义了变量**mx_和my_**，以及构造函数。在自定义因子中，有两个函数必须被override。一个是**evaluateError**函数，它定义了误差量的计算方式，以及雅可比。另一个是**clone**函数，它用于因子的复制，通常情况下只需要使用默认的复制构造函数即可。

在本例中，**evaluateError**函数计算的是误差量：

$$E(q) \triangleq h(q) - m$$

其中

$$h(q) = \begin{bmatrix} q_x \\ q_y \end{bmatrix}, H = \begin{bmatrix} \cos(q_\theta) & -\sin(q_\theta) & 0 \\ \sin(q_\theta) & \cos(q_\theta) & 0 \end{bmatrix}$$

在GTSAM中，对于非向量空间的变量(通常是旋转量)，这样定义雅可比矩阵H：

$$h(q \exp \hat{\xi}) \approx h(q) + H\xi$$

其中 $\xi = (\delta x, \delta y, \delta\theta)$ 是小量。在本例中， $q \in SE(2)$ ，则：

$$\exp \hat{\xi} \approx \begin{bmatrix} 1 & -\delta\theta & \delta x \\ \delta\theta & 1 & \delta y \\ 0 & 0 & 1 \end{bmatrix}$$

使用3X3矩阵表示2D姿态，则：

$$h\left(qe^{\hat{\xi}}\right) \approx h\left(\left[\begin{array}{ccc} \cos\left(q_\theta\right) & -\sin\left(q_\theta\right) & q_x \\ \sin\left(q_\theta\right) & \cos\left(q_\theta\right) & q_y \\ 0 & 0 & 1 \end{array}\right]\left[\begin{array}{ccc} 1 & -\delta\theta & \delta x \\ \delta\theta & 1 & \delta y \\ 0 & 0 & 1 \end{array}\right]\right) =$$
$$\left[\begin{array}{c} q_x + \cos\left(q_\theta\right)\delta x - \sin\left(q_\theta\right)\delta y \\ q_y + \sin\left(q_\theta\right)\delta x + \cos\left(q_\theta\right)\delta y \end{array}\right]$$

上式对 $\xi = (\delta x, \delta y, \delta\theta)$ 求导就是矩阵H的形式。

自定义因子的使用方式和其他因子相同，初始化因子后，直接add到graph中即可：

```
NonlinearFactorGraph graph;
 auto odometryNoise = noiseModel::Diagonal::Sigmas(Vector3(0.2, 0.2, 0.1));
 // Create odometry (Between) factors between consecutive poses
 graph.emplace_shared<BetweenFactor<Pose2> >(1, 2, Pose2(2.0, 0.0, 0.0), odomet
 graph.emplace_shared<BetweenFactor<Pose2> >(2, 3, Pose2(2.0, 0.0, 0.0), odomet

 auto unaryNoise =
     noiseModel::Diagonal::Sigmas(Vector2(0.1, 0.1));  // 10cm std on x,y
 graph.emplace_shared<UnaryFactor>(1, 0.0, 0.0, unaryNoise);
 graph.emplace_shared<UnaryFactor>(2, 2.0, 0.0, unaryNoise);
 graph.emplace_shared<UnaryFactor>(3, 4.0, 0.0, unaryNoise);
 graph.print("\nFactor Graph:\n");  // print

 // 3. Create the data structure to hold the initialEstimate estimate to the sc
 // For illustrative purposes, these have been deliberately set to incorrect va
 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, 0.1));
 initialEstimate.print("\nInitial Estimate:\n");  // print

 // 4. Optimize using Levenberg-Marquardt optimization. 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.
 // Here we will use the default set of parameters.  See the
 // documentation for the full set of parameters.
 LevenbergMarquardtOptimizer optimizer(graph, initialEstimate);
 Values result = optimizer.optimize();
 result.print("Final Result:\n");

 // 5. Calculate and print marginal covariances for all variables
 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;
```

GTSAM加入landmark因子

通过**BearingRangeFactor**类来实现：

```
class BearingRangeFactor
    : public ExpressionFactorN<BearingRange<A1, A2>, A1, A2> {
 private:
  typedef BearingRange<A1, A2> T;
  typedef ExpressionFactorN<T, A1, A2> Base;
  typedef BearingRangeFactor<A1, A2> This;

 public:
  typedef std::shared_ptr<This> shared_ptr;

  /// Default constructor
  BearingRangeFactor() {}

  /// Construct from BearingRange instance
  BearingRangeFactor(Key key1, Key key2, const T &bearingRange,
                     const SharedNoiseModel &model)
      : Base({{key1, key2}}, model, T(bearingRange)) {
    this->initialize(expression({{key1, key2}}));
  }

  /// Construct from separate bearing and range
  BearingRangeFactor(Key key1, Key key2, const B &measuredBearing,
                     const R &measuredRange, const SharedNoiseModel &model)
      : Base({{key1, key2}}, model, T(measuredBearing, measuredRange)) {
    this->initialize(expression({{key1, key2}}));
  }
```

其中，构造函数要求输入两个Key，两个key之间的相对旋转量以及平移量。

在示例**PlanarSLAMExample.cpp**中，首先定义了x1,x2,x3之间的里程计测量（通过BetweenFactor）：

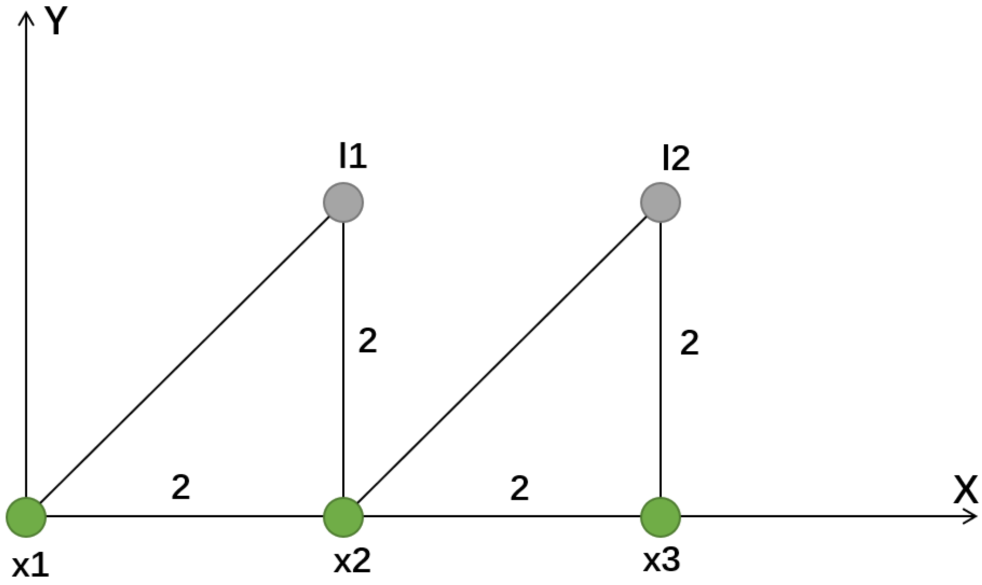
```
NonlinearFactorGraph graph;

  // Create the keys we need for this simple example
```

```cpp
static Symbol x1('x', 1), x2('x', 2), x3('x', 3);
static Symbol l1('l', 1), l2('l', 2);

// Add a prior on pose x1 at the origin. A prior factor consists of a mean and
// a noise model (covariance matrix)
Pose2 prior(0.0, 0.0, 0.0);  // prior mean is at origin
auto priorNoise = noiseModel::Diagonal::Sigmas(
    Vector3(0.3, 0.3, 0.1));            // 30cm std on x,y, 0.1 rad on theta
graph.addPrior(x1, prior, priorNoise);  // add directly to graph

// Add two odometry factors
Pose2 odometry(2.0, 0.0, 0.0);
// create a measurement for both factors (the same in this case)
auto odometryNoise = noiseModel::Diagonal::Sigmas(
    Vector3(0.2, 0.2, 0.1));  // 20cm std on x,y, 0.1 rad on theta
graph.emplace_shared<BetweenFactor<Pose2> >(x1, x2, odometry, odometryNoise);
graph.emplace_shared<BetweenFactor<Pose2> >(x2, x3, odometry, odometryNoise);
```

然后加入了**BearingRangeFactor**：

```cpp
1 auto measurementNoise = noiseModel::Diagonal::Sigmas(
2      Vector2(0.1, 0.2));  // 0.1 rad std on bearing, 20cm on range
3   // create the measurement values - indices are (pose id, landmark id)
4   Rot2 bearing11 = Rot2::fromDegrees(45), bearing21 = Rot2::fromDegrees(90),
5        bearing32 = Rot2::fromDegrees(90);
6   double range11 = std::sqrt(4.0 + 4.0), range21 = 2.0, range32 = 2.0;
7
8   // Add Bearing-Range factors
9   graph.emplace_shared<BearingRangeFactor<Pose2, Point2> >(x1, l1, bearing11, ra
10  graph.emplace_shared<BearingRangeFactor<Pose2, Point2> >(x2, l1, bearing21, ra
11  graph.emplace_shared<BearingRangeFactor<Pose2, Point2> >(x3, l2, bearing32, ra
```

以第9行为例，bearing11表示x1和l1之间的相对旋转，是45度；range11表示了x1和l1之间的相对距离。10行和11行同理。如图所示：



最后赋初值，求解即可：

```
// Create (deliberately inaccurate) initial estimate
  Values initialEstimate;
  initialEstimate.insert(x1, Pose2(0.5, 0.0, 0.2));
  initialEstimate.insert(x2, Pose2(2.3, 0.1, -0.2));
  initialEstimate.insert(x3, Pose2(4.1, 0.1, 0.1));
  initialEstimate.insert(l1, Point2(1.8, 2.1));
  initialEstimate.insert(l2, Point2(4.1, 1.8));

  // Print
  initialEstimate.print("Initial Estimate:\n");
  LevenbergMarquardtOptimizer optimizer(graph, initialEstimate);
  Values result = optimizer.optimize();
  result.print("Final Result:\n");

  // Calculate and print marginal covariances for all variables
  Marginals marginals(graph, result);
  print(marginals.marginalCovariance(x1), "x1 covariance");
  print(marginals.marginalCovariance(x2), "x2 covariance");
  print(marginals.marginalCovariance(x3), "x3 covariance");
  print(marginals.marginalCovariance(l1), "l1 covariance");
  print(marginals.marginalCovariance(l2), "l2 covariance");
```

GTSAM拟合曲线示例

拟合曲线 $f = e^{ax+b}$,求参数a和b的值。首先自定义因子：

```
class CurveFitFactor : public gtsam::NoiseModelFactor1<Vector2>
{
private:
  double _mx, _my;

public:
  CurveFitFactor(Key j, double x, double y, const SharedNoiseModel &model) : Noi
  virtual ~CurveFitFactor() {}

  Vector evaluateError(const Vector2 &q, OptionalMatrixType H) const override
  {
    if (H)
      (*H) = (Matrix(1, 2) << -_mx * exp(q[0] * _mx + q[1]), -exp(q[0] * _mx + q
    return (Vector(1) << _my - exp(q[0] * _mx + q[1])).finished();
    // if (H)
    //   (*H) = (Matrix(2, 2) << -_mx * exp(q[0] * _mx + q[1]), -exp(q[0] * _mx
    // return (Vector(2) << _my - exp(q[0] * _mx + q[1]), 0).finished();
  }
  virtual gtsam::NonlinearFactor::shared_ptr clone() const override
  {
    return std::static_pointer_cast<gtsam::NonlinearFactor>(
        gtsam::NonlinearFactor::shared_ptr(new CurveFitFactor(*this)));
  }
};
```

其中在evaluateError函数中定义了误差量以及雅可比。

```
using symbol_shorthand::X;
  NonlinearFactorGraph graph;

  srand(time(nullptr)); //设置随机数种子

  // f = exp(ax + b)
  double a = 0.03;
  double b = 0.5;

  auto CurveNoise = noiseModel::Diagonal::Sigmas(Vector1(0.1)); // 10cm std on x
  for (size_t i = 0; i < 100; i++)
  {
    double randoxNoise = (rand() % (1000)) * 0.0001;
    graph.emplace_shared<CurveFitFactor>(X(0), 0.1 * i, exp(a * 0.1 * i + b) + r
    // cout << "i: " << i << " exp(a * i + b): " << exp(a * i + b) << " randoxNc
  }

```

```
Values initialEstimate;
initialEstimate.insert(X(0), Vector2(0.0, 0.0));
initialEstimate.print("\nInitial Estimate:\n"); // print

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

生成100个数据，并加上随机噪声，把CurveFitFactor添加到graph当中，最后求解即可。