GTSAM与卡尔曼滤波

1. 卡尔曼滤波

线性模型

$$\begin{cases} x_t &= Fx_{t-1} + & Bu & + w \\ z_t &= Hx_{t-1} & + & v_t \end{cases}$$

F状态转移矩阵，B为控制输入矩阵，w为过程噪声Q

H为状态观测矩阵，v为观测噪声R

五大公式

预测，根据 x_{t-1} 预测 x_t^-

预测

$$x_t^- \ = Fx_{t-1} + \ Bu \ + w$$

计算卡尔曼增益

$$K_t \ = \frac{P_t^- H^T}{HP_t^- H^T + R}$$

校正，根据观测值 z_t 对预测值 x_t^- 进行修改

$$x_t \ = \ x_t^- \ + \ K_t(z_t - Hx_t^-)$$

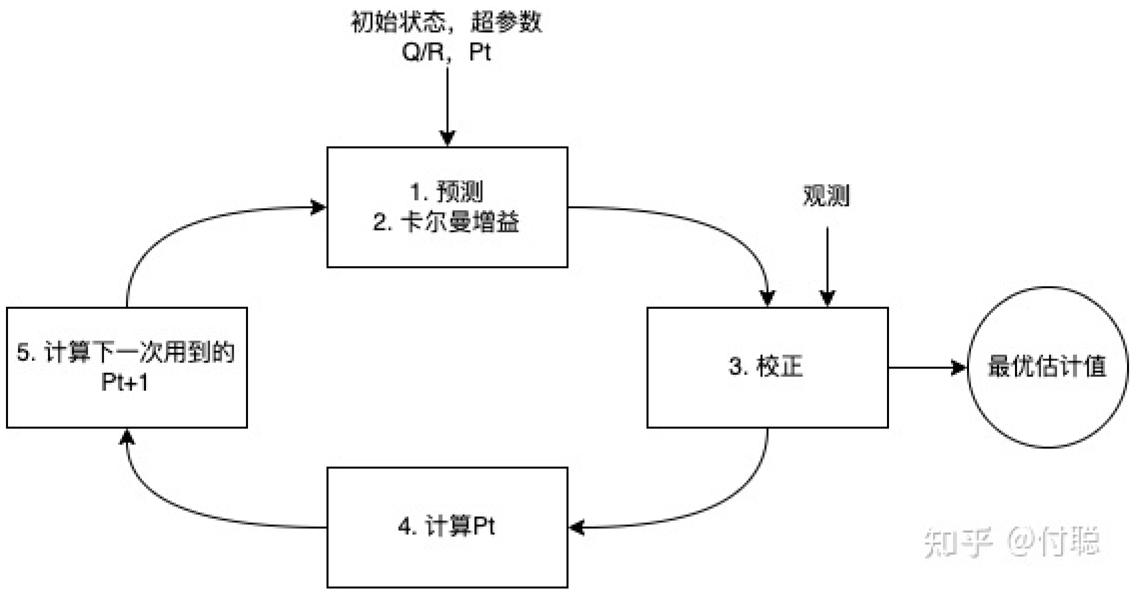
修正协方差

$$P_t \ = \ (I - K_t H)P_t^-$$

预测下一帧的协方差

$$P_{t+1}^- \ = FP_tF^T \ + \ Q_{t+1}$$

计算流程示意图



2. 代码梳理

举例了一个二维运动模型，使用因子图对扩展卡尔曼滤波器求解

分别使用了模板类ExtendedKalmanFilter和自定义模板实现

模板类实现

定义初始状态

初始状态是初始的位姿和协方差(置信度)，并根据其初始化扩展滤波器类ExtendedKalmanFilter

```
// Create the Kalman Filter initialization point
Point2 x_initial(0.0, 0.0);
SharedDiagonal P_initial = noiseModel::Diagonal::Sigmas(Vector2(0.1, 0.1));

// Create Key for initial pose
Symbol x0('x',0);

// Create an ExtendedKalmanFilter object
ExtendedKalmanFilter<Point2> ekf(x0, x_initial, P_initial);
```

预测

定义了第0帧的初始状态后，再预测第1帧的状态，在因子图中相当于求解 *argmax_{x1} P(x1) = P(x1|x0) P(x0)*

在卡尔曼滤波器中，这里是已知t时刻状态，预测t+1时刻的状态，也就是求*x_{t+1|t} 和 P_{t+1|t}*

这里假设了一个线性运动模型，一个小车以1m/s的速度向右行驶

根据 $x_t^- = Fx_{t-1} + Bu + w$

可得 $x_t^- = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} x_{t-1} + \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} + w$

过程噪声w是均值为0 的高斯白噪声Q

```
Vector u = Vector2(1.0, 0.0);
SharedDiagonal Q = noiseModel::Diagonal::Sigmas(Vector2(0.1, 0.1), true);

// This simple motion can be modeled with a BetweenFactor
// Create Key for next pose
Symbol x1('x',1);
// Predict delta based on controls
Point2 difference(1,0);
// Create Factor
BetweenFactor<Point2> factor1(x0, x1, difference, Q);

// Predict the new value with the EKF class
Point2 x1_predict = ekf.predict(factor1);
traits<Point2>::Print(x1_predict, "X1 Predict");
ekf.Density();
```

更新

然后，系统会收到一个对小车位置的观测值z1，卡尔曼滤波器就会进入更新校正阶段

这也等价于因子图中的求解*P(x1|z1) ~ P(z1|x1)*P(x1)*

对卡尔曼滤波器来说，这里是 $z_t = Hx_{t-1} + v_t$

在当前例子的卡尔曼滤波器中，我们假设观测值类似GPS信号，返回当前位置

于是设H = [1 0 ; 0 1]，R = [0.25 0 ; 0 0.25]

```
SharedDiagonal R = noiseModel::Diagonal::Sigmas(Vector2(0.25, 0.25), true);

// This simple measurement can be modeled with a PriorFactor
Point2 z1(1.0, 0.0);
PriorFactor<Point2> factor2(x1, z1, R);

// Update the Kalman Filter with the measurement
Point2 x1_update = ekf.update(factor2);
traits<Point2>::Print(x1_update, "X1 Update");
```

循环

接下来再重复以上 预测-更新 步骤，就可以持续地预测位姿了

```cpp
// Do the same thing two more times...
// Predict
Symbol x2('x',2);
difference = Point2(1,0);
BetweenFactor<Point2> factor3(x1, x2, difference, Q);
Point2 x2_predict = ekf.predict(factor1);  // bug?
traits<Point2>::Print(x2_predict, "X2 Predict");

// Update
Point2 z2(2.0, 0.0);
PriorFactor<Point2> factor4(x2, z2, R);
Point2 x2_update = ekf.update(factor4);
traits<Point2>::Print(x2_update, "X2 Update");



// Do the same thing one more time...
// Predict
Symbol x3('x',3);
difference = Point2(1,0);
BetweenFactor<Point2> factor5(x2, x3, difference, Q);
Point2 x3_predict = ekf.predict(factor5);
traits<Point2>::Print(x3_predict, "X3 Predict");

// Update
Point2 z3(3.0, 0.0);
PriorFactor<Point2> factor6(x3, z3, R);
Point2 x3_update = ekf.update(factor6);
traits<Point2>::Print(x3_update, "X3 Update");
```

模板类

```cpp
public:
 typedef boost::shared_ptr<ExtendedKalmanFilter<VALUE> > shared_ptr;
 typedef VALUE T;

 //@deprecated: any NoiseModelFactor will do, as long as they have the right key
 typedef NoiseModelFactor2<VALUE, VALUE> MotionFactor;
 typedef NoiseModelFactor1<VALUE> MeasurementFactor;

protected:
 T x_;                                     // linearization point
 JacobianFactor::shared_ptr priorFactor_;  // Gaussian density on x_
                                           // 高斯概率密度
```

```

  static T solve_(const GaussianFactorGraph& linearFactorGraph, const Values& lin
                   Key x, JacobianFactor::shared_ptr* newPrior);

public:
  /// @name Standard Constructors
  /// @{

  ExtendedKalmanFilter(Key key_initial, T x_initial, noiseModel::Gaussian::shared

  /// @}
  /// @name Testable
  /// @{

  /// print
  void print(const std::string& s = "") const {
    std::cout << s << "\n";
    x_.print(s + "x");
    priorFactor_->print(s + "density");
  }

  /// @}
  /// @name Interface
  /// @{

  /**
   * Calculate predictive density P(x_) ~ \int  P(x_min) P(x_min, x_)
   * The motion model should be given as a factor with key1 for x_min and key2_ f
   */
  T predict(const NoiseModelFactor& motionFactor);

  /**
   * Calculate posterior density P(x_) ~ L(z|x) P(x)
   * The likelihood L(z|x) should be given as a unary factor on x
   */
  T update(const NoiseModelFactor& measurementFactor);

  /// Return current predictive (if called after predict)/posterior (if called af
  const JacobianFactor::shared_ptr Density() const {
    return priorFactor_;
  }

```

自定义实现(手动isam2)

手动通过创建因子图实现卡尔曼滤波器，对位姿进行预测更新

初始化

```
// Create a factor graph to perform the inference
// 创建因子图用来推理
GaussianFactorGraph::shared_ptr linearFactorGraph(new GaussianFactorGraph);

// Create the desired ordering
Ordering::shared_ptr ordering(new Ordering);

// Create a structure to hold the linearization points
Values linearizationPoints;

// Create new state variable
Symbol x0('x',0);
ordering->insert(x0, 0);

// Initialize state x0 (2D point) at origin by adding a prior factor, i.e., Baye
// This is equivalent to x_0 and P_0
Point2 x_initial(0,0);
SharedDiagonal P_initial = noiseModel::Diagonal::Sigmas((Vec(2) << 0.1, 0.1));
PriorFactor<Point2> factor1(x0, x_initial, P_initial);
// Linearize the factor and add it to the linear factor graph
linearizationPoints.insert(x0, x_initial);
linearFactorGraph->push_back(factor1.linearize(linearizationPoints, *ordering));
```

预测

根据 因子图介绍及应用 中对因子误差模型的介绍，每个因子一般定义为指数函数误差函数 $||f_i(X_i)||^2_{\Sigma_i} = f_i(X_i)^T \Sigma^{-1} f_i(X_i)$

这里应使用BetweenFactor因子：$f(x_i, x_{i+z}) = (x_{i+1} - x_i) - z$

又 $x_{i+1} = Fx_i + Bu + w$

```
// In the case of factor graphs, the factor related to the motion model would be
// f2 = (f(x_{t}) - x_{t+1}) * Q^-1 * (f(x_{t}) - x_{t+1})^T
// Conveniently, there is a factor type, called a BetweenFactor, that can genera
// given the expected difference, f(x_{t}) - x_{t+1}, and Q.
// so, difference = x_{t+1} - x_{t} = F*x_{t} + B*u_{t} - I*x_{t}
//                                  = (F - I)*x_{t} + B*u_{t}
//                                  = B*u_{t} (for our example)
Symbol x1('x',1);
ordering->insert(x1, 1);

Point2 difference(1,0);
SharedDiagonal Q = noiseModel::Diagonal::Sigmas((Vec(2) << 0.1, 0.1));
BetweenFactor<Point2> factor2(x0, x1, difference, Q);
// Linearize the factor and add it to the linear factor graph
linearizationPoints.insert(x1, x_initial);
linearFactorGraph->push_back(factor2.linearize(linearizationPoints, *ordering));
```

按照X0，X1的顺序消元，就可得到贝叶斯网络，对贝叶斯网络进行优化就可得到当前的最佳估计(P36)

滤波器这里只需要P(X1)的先验值，所以我们只需要保留贝叶斯网络的根即可

实际上后文中也提到了变量消元就是矩阵分解，QR分解后得到的上三角矩阵就是贝叶斯网络(iSAM2)

3.2 消元算法

给定任意一个（最好是稀疏的）因子图，存在一个通用算法，可以计算出未知变量 $\boldsymbol{X}$ 的后验概率密度 $p(\boldsymbol{X}|\boldsymbol{Z})$，因此可以很容易地得到求解问题的最大后验概率解。正如我们所看到的那样，因子图将未归一化的后验概率 $\phi(\boldsymbol{X}) \propto P(\boldsymbol{X}|\boldsymbol{Z})$ 表示为一系列因子的乘积。在 SLAM 问题中，因子图通常直接由观测量生成。消元算法是一种将因子图转换回贝叶斯网络的方法，但是它现在仅仅转换未知量 $\boldsymbol{X}$。这使得最大后验概率推断、采样（之前提到过），以及边缘化（marginalization）变得很容易。

特别地，**变量消元**（variable elimination）算法是一种将因子图

$$\phi(\boldsymbol{X}) = \phi(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n) \tag{3.5}$$

分解为如下表示形式的因子化贝叶斯网络概率密度。

$$p(\boldsymbol{X}) = p(\boldsymbol{x}_1|\boldsymbol{S}_1)p(\boldsymbol{x}_2|\boldsymbol{S}_2) \cdots p(\boldsymbol{x}_n) = \prod_j p(\boldsymbol{x}_j|\boldsymbol{S}_j) \tag{3.6}$$

```
// Eliminate this in order x0, x1, to get Bayes net P(x0|x1)P(x1)
// As this is a filter, all we need is the posterior P(x1), so we just keep the
//
// Because of the way GTSAM works internally, we have used nonlinear class even
// system is linear. We first convert the nonlinear factor graph into a linear c
// ordering. Linear factors are simply numbered, and are not accessible via name
// variables. Also, the nonlinear factors are linearized around an initial estin
// system, the initial estimate is not important.

// Solve the linear factor graph, converting it into a linear Bayes Network ( P(
GaussianSequentialSolver solver0(*linearFactorGraph);
GaussianBayesNet::shared_ptr linearBayesNet = solver0.eliminate();// 矩阵分解

// Extract the current estimate of x1,P1 from the Bayes Network
VectorValues result = optimize(*linearBayesNet);
Point2 x1_predict = linearizationPoints.at<Point2>(x1).retract(result[ordering->
x1_predict.print("X1 Predict");

// Update the new linearization point to the new estimate
linearizationPoints.update(x1, x1_predict);
```

重置因子图模型

这里与iSAM2的优化理论相关，将优化后的X1作为先验因子重置因子图模型

对于部分QR分解 $A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$ ，于是对于Ax=b，则有 $\begin{bmatrix} R \\ 0 \end{bmatrix} x = Q^T b = \begin{bmatrix} d \\ e \end{bmatrix}$

> 总结一下贝叶斯树是如何处理iSAM1遗留下来的问题：
>
> 如何减少重排序的次数，如何降低新增状态量，对R矩阵稀疏性的破坏程度
>
> 变量消除法实际上是在对雅可比矩阵求QR分解，只不过不需要全局分解，可以通过逐步的局部操作完成，这样就说明了局部的QR分解不会影响全局的一致性。是后面贝叶斯树可以局部更新的前提。
>
> 贝叶斯树，通过维护树结构，**当新增状态量时，通过树结构快速找出，受影响的其他变量，将该部分树结构删除**，并重新线性化生成贝叶斯网再生成贝叶斯树。而其他未受影响的子树结构得以保留，从而减少了计算。
>
> ————————————————
>
>

将本例中的根条件概率P(X1)转化成下一步的先验因子

这个线性化的点接下来与初始化时相比，将会发生变化

根据一阶泰勒展开公式 $f(x) = f(x_0) + (x - x_0)f'(x)$

在x0处展开，可得 $f'(x_1)dx = f(x_0) - f(x_1) = x_1 \ - \ Fx_0$

这一步相当于更新雅克比矩阵，为什么不能在两点展开，参考FEJ理论

```
// Create a new, empty graph and add the prior from the previous step
linearFactorGraph = GaussianFactorGraph::shared_ptr(new GaussianFactorGraph);

// Eliminate this in order x0, x1, to get Bayes net P(x0|x1)P(x1)

// Convert the root conditional, P(x1) in this case, into a Prior for the next s
// Some care must be done here, as the linearization point in future steps will
// than what was used when the factor was created.
// f = || F*dx1' - (F*x0 - x1) ||^2, originally linearized at x1 = x0
// After this step, the factor needs to be linearized around x1 = x1_predict
// This changes the factor to f = || F*dx1'' - b'' ||^2
//                              = || F*(dx1' - (dx1' - dx1'')) - b'' ||^2
```

```
//                                = || F*dx1' - F*(dx1' - dx1'') - b'' ||^2
//                                = || F*dx1' - (b'' + F(dx1' - dx1'')) ||^2
//                                -> b' = b'' + F(dx1' - dx1'')
//                                -> b'' = b' - F(dx1' - dx1'')
//                                = || F*dx1'' - (b'  - F(dx1' - dx1'')) ||^2
//                                = || F*dx1'' - (b'  - F(x_predict - x_inital)) |
const GaussianConditional::shared_ptr& cg0 = linearBayesNet->back(); // 条件概率
assert(cg0->nrFrontals() == 1);
assert(cg0->nrParents() == 0);
// 雅克比，残差
linearFactorGraph->add(0, cg0->R(), cg0->d() - cg0->R()*result[ordering->at(x1)]
// Create the desired ordering
ordering = Ordering::shared_ptr(new Ordering);
ordering->insert(x1, 0);
```

校正

在接收到观测值Z1后对X1的预测值进行校正，用因子图的语言描述*P(x1|z1) ~ P(z1|x1)*P(x1) ~ f3(x1)*f4(x1;z1)*

这也印证了前面为什么要重置因子图更新雅克比矩阵，因为接下来要在X1处展开线性化了

```
// For the purposes of this example, let us assume we have something like a GPS
// the current position of the robot. For this simple example, we can use a Pric
// observation as it depends on only a single state variable, x1. To model real
// generally requires the creation of a new factor type. For example, factors fc
// sensors, and camera projections have already been added to GTSAM.

// In the case of factor graphs, the factor related to the measurements would be
// f4 = (h(x_{t}) - z_{t}) * R^-1 * (h(x_{t}) - z_{t})^T
//    = (x_{t} - z_{t}) * R^-1 * (x_{t} - z_{t})^T
// This can be modeled using the PriorFactor, where the mean is z_{t} and the cc
```

```
Point2 z1(1.0, 0.0);
SharedDiagonal R1 = noiseModel::Diagonal::Sigmas((Vec(2) << 0.25, 0.25));
PriorFactor<Point2> factor4(x1, z1, R1);
// Linearize the factor and add it to the linear factor graph
linearFactorGraph->push_back(factor4.linearize(linearizationPoints, *ordering));
```

和前文一样转化成贝叶斯网络求解

```
// We have now made the small factor graph f3-(x1)-f4
// where factor f3 is the prior from previous time ( P(x1) )
// and   factor f4 is from the measurement, z1 ( P(x1|z1) )
// Eliminate this in order x1, to get Bayes net P(x1)
// As this is a filter, all we need is the posterior P(x1), so we just keep the
// We solve as before...

// Solve the linear factor graph, converting it into a linear Bayes Network ( P(
GaussianSequentialSolver solver1(*linearFactorGraph);
linearBayesNet = solver1.eliminate();

// Extract the current estimate of x1 from the Bayes Network
result = optimize(*linearBayesNet);
Point2 x1_update = linearizationPoints.at<Point2>(x1).retract(result[ordering->a
x1_update.print("X1 Update");

// Update the linearization point to the new estimate
linearizationPoints.update(x1, x1_update);
```

再重置

这里没用公式，而是构造了一个

```
// Wash, rinse, repeat for another time step
// Create a new, empty graph and add the prior from the previous step
linearFactorGraph = GaussianFactorGraph::shared_ptr(new GaussianFactorGraph);

// Convert the root conditional, P(x1) in this case, into a Prior for the next s
// The linearization point of this prior must be moved to the new estimate of x,
// the first key in the next iteration
const GaussianConditional::shared_ptr& cg1 = linearBayesNet->back();
assert(cg1->nrFrontals() == 1);
assert(cg1->nrParents() == 0);
JacobianFactor tmpPrior1 = JacobianFactor(*cg1);
linearFactorGraph->add(0, tmpPrior1.getA(tmpPrior1.begin()), tmpPrior1.getb() -

// Create a key for the new state
Symbol x2('x',2);

// Create the desired ordering
ordering = Ordering::shared_ptr(new Ordering);
ordering->insert(x1, 0);
ordering->insert(x2, 1);
```

接下来依次循环往复即可

3. 运行结果

z1(0.9, 0.0)　z1(2.1, 0.0)　z1(3.2, 0.0)

x(0.0, 0.0)　x1(1.0, 0.0)　x1(2.0, 0.0)　x1(3.0, 0.0)

🍰 X1 Predict(1, 0)

X1 Update(0.975758, 0)

X2 Predict(1.97576, 0)

X2 Update(2.01141, 0)

X3 Predict(3.01141, 0)

X3 Update(3.06966, 0)

同时，也可以通过ekf.Density()->print()输出雅克比矩阵与残差

在预测后调用，返回先验值

在更新后调用，返回后验值

🍰 X1 Predict(1, 0)

A[x1] = [7.07107 0;

0 7.0710]

b = [-8.88178e-16 0]

No noise model

X1 Update(0.975758, 0)

A[x1] = [8.12404 0;

0 8.12404]

b = [0 0]

No noise model

X2 Predict(1.97576, 0)

X2 Update(2.01141, 0)

X3 Predict(3.01141, 0)

X3 Update(3.06966, 0)

https://h2nikt4dp2.feishu.cn/minutes/obcn7a6pckxvx9648q982i2i