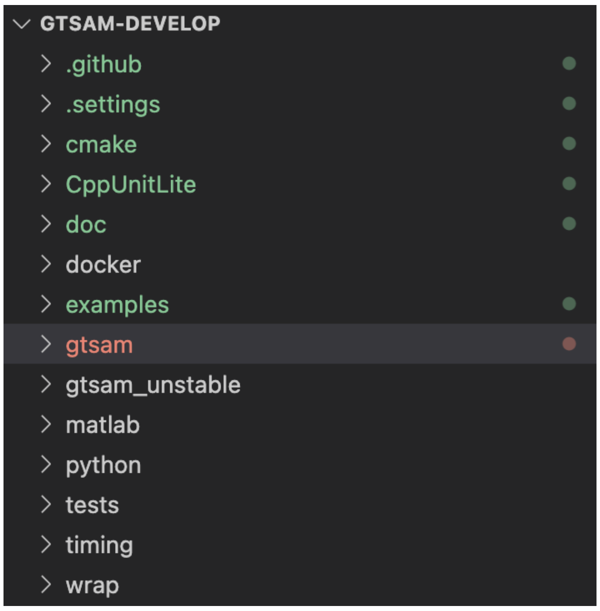
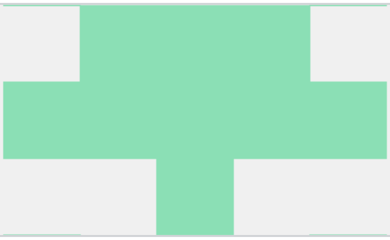
GTSAM模块

GTSAM的介绍、安装大家参考前面同学做的文档，已经非常详细了，另外GTSAM的doc目录下作者也非常贴心放了不少技术文档，例如：gtsam.pdf、math.pdf、LieGroup.pdf等等，非常值得大家查阅。examples目录提供了很多实操的例子，大部分例子后面的几位同学会详细介绍，这次分享主要讲解GTSAM的核心模块。GTSAM的核心代码就在gtsam文件夹目录下，其他的代码例如gtsam的python和Matlab接口以及C++版本的gtsam代码转换为Matlab代码就不介绍了，感兴趣的同学自己查阅。



```
GTSAM-DEVELOP
  > .github
  > .settings
  > cmake
  > CppUnitLite
  > doc
  > docker
  > examples
  > gtsam
  > gtsam_unstable
  > matlab
  > python
  > tests
  > timing
  > wrap
```

这篇文档主要介绍GTSAM的核心代码模块，Github地址：



https://github.com/borglab/gtsam

GitHub - borglab/gtsam: GTSAM is a library of C++ classes that implement smoothing and mapping (SAM)

GTSAM is a library of C++ classes that implement smoothing and mapping (SAM…

gtsam目录下的文件夹是gtsam的核心模块，主要有：

```
gtsam  --|----- base        ： 定义基础的自定义模版变量
         |----- basic       ： 基函数及Chebyshev多项式分解
         |----- geometry    ： 数学运算/相机模型及内参标定
         |----- inference   ： gtsam数据结构及因子图
         |----- linear      ： 线性因子图模型
         |----- hybrid      ： 混合因子图模型
         |----- discrete    ： 离散因子图模型
```

|----- symbolic　：象征因子图模型

|----- sam　：RangeFactor和BearingFactor

|----- slam　：前端里程计因子

|----- sfm　：三维重建

|----- nonLinear　：非线性优化器

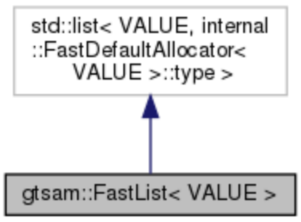
每一个模块的目录下都有单元测试，在对应的tests目录下，对该模块不是很明白怎么应用的同学，可以在单元测试目录下找找有没有你想要的测试例子。

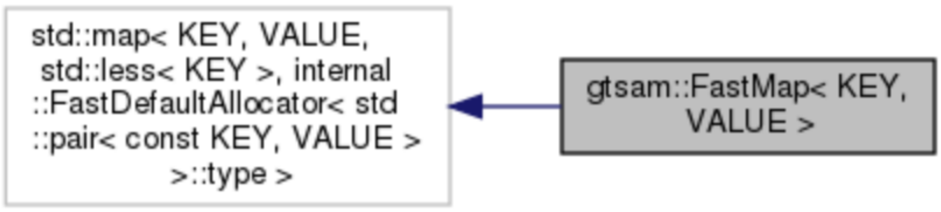
1. base

base里面定义了很多GTSAM自定义的模版变量，这些变量都会被其他模块调用，但我们在用GTSAM的时候，很少会直接使用这些变量，都是直接调用一些模块的接口。下面介绍一些常用的模版变量。

封装C++标准库容器

GTSAM没有直接使用C++标准库的默认的 `std::List`、`std::map`、`std::set` 和 `std::vector` 容器，它在 `FastDefaultAllocator.h` 中自己写了一个容器内存分配器，然后封装成 `gtsam::FastList`、`gtsam::FastMap`、`gtsam::FastSet` 和 `gtsam::FastVector`，作者说封装后的容器执行效率有提升几个百分点，`FastList` 和 `FastMap` 的继承关系可以参考下面的图片，其他的封装的容器大同小异：


std::list< VALUE, internal ::FastDefaultAllocator< VALUE >::type >
gtsam::FastList< VALUE >


std::map< KEY, VALUE, std::less< KEY >, internal ::FastDefaultAllocator< std ::pair< const KEY, VALUE > >::type >
gtsam::FastMap< KEY, VALUE >

图片来源：

https://gtsam-jlblanco-docs.readthedocs.io/en/latest/_static/doxygen/html/classgtsam_1_1FastList.html

GTSAM: gtsam::FastList Class Template Reference

GTSAM 4.0.2 C++ library for smoothing and mapping (SAM) gtsam FastList Public Types | Public Member Functions | Friends | List of all members gtsam::FastList< VALUE > Class Template Reference In

对应的头文件：

```
#include <gtsam/base/FastDefaultAllocator.h>
#include <gtsam/base/FastList.h>
#include <gtsam/base/FastMap.h>
#include <gtsam/base/FastSet.h>
#include <gtsam/base/FastVector.h>
```

此外，GTSAM还封装了 `boost::make_shared` 变为 `gtsam::make_shared` ，作者说是解决了 `boost::make_shared` 在不遵守内存自定义对齐，导致在运行时出现 SEGFAULT，难以调试的问题：

```
#include <gtsam/base/make_shared.h>
```

封装基础数学运算

GTSAM定义了群和李代数以及流形的模版函数，这些模版函数会在geometry模块中的李群李代数运算 `gtsam::SO3` `gtsam::SO4` `gtsam::SOn` 被调用：

```
#include <gtsam/base/Group.h>
#include <gtsam/base/Lie.h>
#include <gtsam/base/Manifold.h>
#include <gtsam/base/ProductLieGroup.h>
```

基于 `Eigen::Matrix` `Eigen::Vector` 定义了 `gtsam::Matrix` 和 `gtsam::Vector` 以及相关的运算，熟悉Eigen库看这下面这些头文件都没啥压力：

```
#include <gtsam/base/Matrix.h>
#include <gtsam/base/Vector.h>
```

其他小工具

base模块还封装了一些工具，例如 `debug.h` `timing.h` `utilities.h` `types.h` ，`types.h` 里面有两个比较重要的变量 `Key` 和 `FactorIndex` ，都是64位的整型，在优化的时候会经常看到：

```
  /// Integer nonlinear key type
  typedef std::uint64_t Key;
  /// Integer nonlinear factor index type
  typedef std::uint64_t FactorIndex; // 因子变量的索引
```

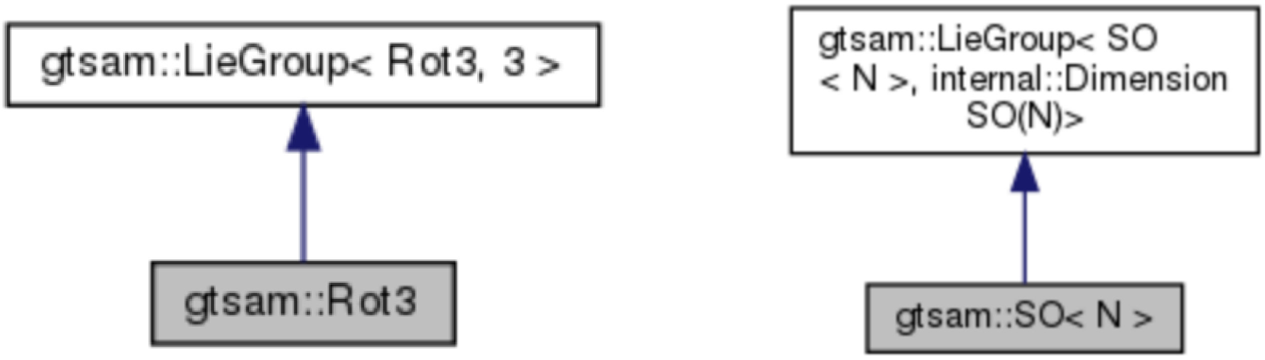
2. basic

基函数及Chebyshev多项式分解

3. geometry

李群李代数数学运算

这个模块封装了很多李群李代数的数学运算，这些运算大都是继承于base模块的基础数学运算：


gtsam::LieGroup< Rot3, 3 >
gtsam::Rot3
gtsam::LieGroup< SO < N >, internal::Dimension SO(N)>
gtsam::SO< N >

图片来源：

https://gtsam-jlblanco-docs.readthedocs.io/en/latest/_static/doxygen/html/classgtsam_1_1SO.html

GTSAM: gtsam::SO Class Template Reference

GTSAM 4.0.2 C++ library for smoothing and mapping (SAM) gtsam SO Classes | Public Types | Public Member Functions | Static Public Member Functions | Protected Types | Protected Attributes | List of al

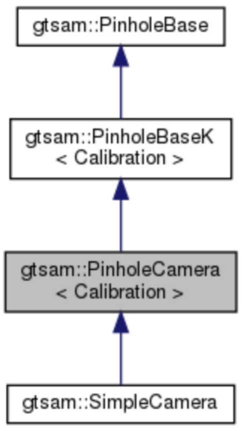
```
// 旋转Rot(Quaternion)、平移Point和位姿Pose
#include <gtsam/geometry/Point2.h> // 2D Point 等价于Vector2
#include <gtsam/geometry/Point3.h> // 3D Point 等价于Vector3
#include <gtsam/geometry/Rot2.h> // 2D rotation
#include <gtsam/geometry/Rot3.h> // 3D rotation represented as a rotation matrix
#include <gtsam/geometry/Pose2.h> // A 2D pose (Point2,Rot2)
#include <gtsam/geometry/Pose3.h> // A 3D pose (R,t) : (Rot3,Point3)

#include <gtsam/geometry/Quaternion.h> // Lie Group wrapper for Eigen Quaternion

#include <gtsam/geometry/Rot3Q.cpp> // Rotation (internal: quaternion representa
#include <gtsam/geometry/Rot3M.cpp> // Rotation (internal: 3*3 matrix representa
// 李群李代数SOn
#include <gtsam/geometry/SOn.h> // N*N matrix representation of SO(N). N can be
#include <gtsam/geometry/SO3.h> // 3*3 matrix representation of SO(3)
#include <gtsam/geometry/SO4.h> // 4*4 matrix representation of SO(4)
// 2D/3D位姿(点云)对齐
// 参考: http://www5.informatik.uni-erlangen.de/Forschung/Publikationen/2005/Zins
#include <gtsam/geometry/Similarity2.h> // Implementation of Similarity2 transfo
#include <gtsam/geometry/Similarity3.h> // Implementation of Similarity3 transfo

```

相机模型及内参标定

模块里面包含了单目、双目以及球形相机的模型及其各自的内参标定，另外有两个比较重要的头文件，分别是本质矩阵和三角化，都是视觉slam非常重要的概念：

```
/**
 * An essential matrix is like a Pose3, except with translation up to scale
 * It is named after the 3*3 matrix aEb = [aTb]x aRb from computer vision,
 * but here we choose instead to parameterize it as a (Rot3,Unit3) pair.
 * We can then non-linearly optimize immediately on this 5-dimensional manifold.
 */
#include <gtsam/geometry/EssentialMatrix.h> // 本质矩阵 5自由度
#include <gtsam/geometry/triangulation.h> // 三角化
```

单目相机模型和内参标定：



```
// A simple camera class with a Cal3_S2 calibration
#include <gtsam/geometry/SimpleCamera.h>

// Pinhole camera with known calibration
#include <gtsam/geometry/PinholePose.h>

// Base class for all pinhole cameras
#include <gtsam/geometry/PinholeCamera.h>

// 不同的单目相机有不同的标定方法：
/**
  // The most common 5DOF 3D->2D calibration
  gtsam::PinholeCamera<gtsam::Cal3_S2>;

  // Calibration used by Bundler
  gtsam::PinholeCamera<gtsam::Cal3Bundler>;

  // Calibration of a camera with radial distortion, calcu
  gtsam::PinholeCamera<gtsam::Cal3DS2>;

  // Unified Calibration Model, see Mei07icra for details
  gtsam::PinholeCamera<gtsam::Cal3Unified>;

  // Calibration of a fisheye camera
  gtsam::PinholeCamera<gtsam::Cal3Fisheye>;
*/
```

双目相机模型及内参标定：

```
#include <gtsam/geometry/StereoCamera.h> // A Rectified Stereo Camera
// The most common 5DOF 3D->2D calibration + Stereo baseline
#include <gtsam/geometry/Cal3_S2Stereo.h>
```

球形相机模型及内参标定：

```
// Calibrated camera with spherical projection
#include <gtsam/geometry/SphericalCamera.h>
```

4. inference

inference模块主要讲了gtsam的数据结构以及因子图的实现，后面的linear、discrete、hybrid和symbolic模块大多都是继承inference模块。

数据结构

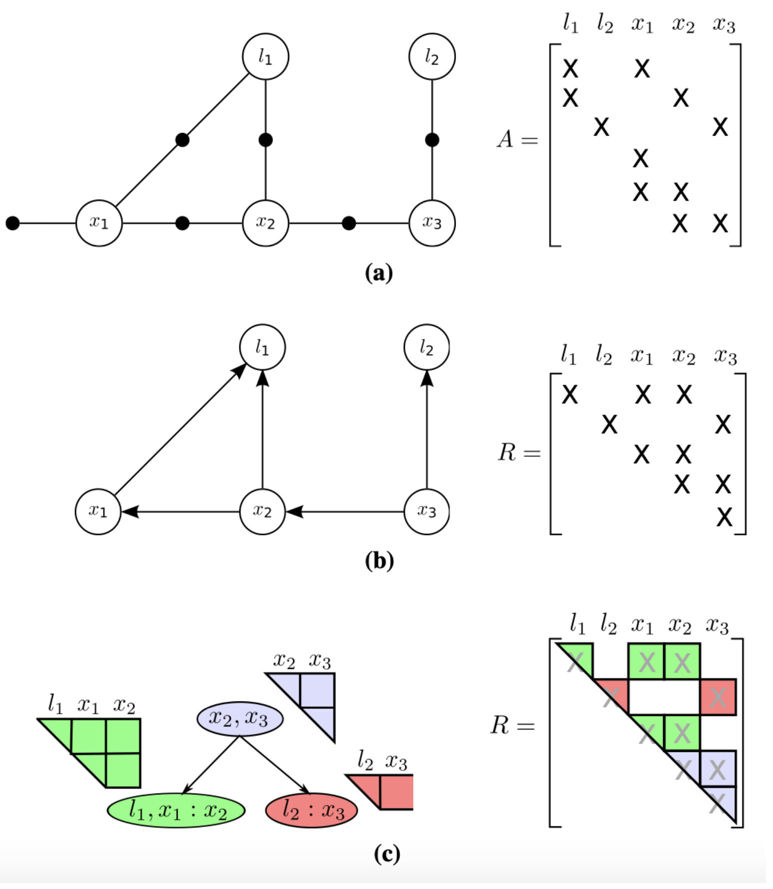
贝叶斯网络 BayesNet

A BayesNet is a tree of conditionals, stored in elimination order.

```
#include <gtsam/inference/BayesNet.h>
```

贝叶斯树 Bayes Tree

这是论文ISAM2里面用到的数据结构，大大提高了优化的效率，感兴趣的同学可以参考论文：



图片来源：https://www.cs.cmu.edu/~kaess/pub/Kaess12ijrr.pdf

```
#include <gtsam/inference/BayesTree.h>
```

聚类树 cluster-tree

A cluster-tree is associated with a factor graph and is defined as in Koller-Friedman

```
#include <gtsam/inference/ClusterTree.h>
```

消除树 elimination tree

An elimination tree is a data structure used intermediately during elimination.

```
#include <gtsam/inference/EliminationTree.h>
```

连接树 JunctionTree

A JunctionTree is a cluster tree, a set of variable clusters with factors, arranged in a tree, with the additional property that it represents the clique tree associated with a Bayes Net.

The difference with the BayesTree is that a JunctionTree stores factors, whereas a BayesTree stores conditionals, that are the product of eliminating the factors in the corresponding JunctionTree cliques.

```
#include <gtsam/inference/JunctionTree.h>
```

ISAM数据结构

Incremental update functionality (iSAM) for BayesTree.

A Bayes tree with updated methods that implements the iSAM algorithm. Given a set of new factors, it re-eliminates the invalidated part of the tree.

```
#include <gtsam/inference/ISAM.h>
```

因子图

从概率图 -> 贝叶斯网络 -> 因子图的过程，大家可以参考上一次分享的内容，写得非常详细，这里不再赘述。

因子图基类

因子图所在的头文件以及被调用最多的函数：

`DiscreteFactorGraph` `HybridFactorGraph` `GaussianFactorGraph` `SymbolicFactorGraph` `ExpressionFactorGraph` `NonlinearFactorGraph`

```
#include <gtsam/inference/FactorGraph.h>


实际过程中我们用的比较多的函数是push_back或者add，表示添加因子：
  /**
   * Add a factor by value, will be copy-constructed (use push_back with a
   * shared_ptr to avoid the copy).
   */
  template <class DERIVEDFACTOR>
  IsDerived<DERIVEDFACTOR> push_back(const DERIVEDFACTOR& factor) {
    factors_.push_back(std::allocate_shared<DERIVEDFACTOR>(
        Eigen::aligned_allocator<DERIVEDFACTOR>(), factor));
  }
```

```
/// `add` is a synonym for push_back.
template <class DERIVEDFACTOR>
IsDerived<DERIVEDFACTOR> add(std::shared_ptr<DERIVEDFACTOR> factor) {
  push_back(factor);
}
```

因子基类

因子的实现在头文件 `Factor.h` ，我们可以通过继承该类，定义自己的因子，下面多种不同的因子继承于 `Factor.h` ：

```
#include <gtsam/inference/Factor.h>

   * There are five broad classes of factors that derive from Factor:

   * - Nonlinear factors, such as class NonlinearFactor and class NoiseModelFact
       represent a nonlinear likelihood function over a set of variables.
   * - Gaussian factors, such as class JacobianFactor and class HessianFactor, w
       represent a Gaussian likelihood over a set of variables.
   * - Discrete factors, such as class DiscreteFactor and class DecisionTreeFact
       represent a discrete distribution over a set of variables.
   * - Hybrid factors, such as class HybridFactor, which represent a mixture of
       Gaussian and discrete distributions over a set of variables.
   * - Symbolic factors, used to represent a graph structure, such as
       class SymbolicFactor, only used for symbolic elimination etc.
```

因子索引

变量的索引Symbol 是一个char 型数和一个（64-8）位的数组成的，其中高8位的数表示变量的统一类型，比如我们可以用 `L` 表示landmark， `X` 表示pos， `V` 表示速度， `B` 表示bias，变量的索引更加方便，每种变量取值范围是 $0 - 2^{64-8}$

```
#include <gtsam/inference/Symbol.h>
```

```
namespace symbol_shorthand {
inline Key A(std::uint64_t j) { return Symbol('a', j); }
inline Key B(std::uint64_t j) { return Symbol('b', j); }
inline Key C(std::uint64_t j) { return Symbol('c', j); }
inline Key D(std::uint64_t j) { return Symbol('d', j); }
inline Key E(std::uint64_t j) { return Symbol('e', j); }
inline Key F(std::uint64_t j) { return Symbol('f', j); }
inline Key G(std::uint64_t j) { return Symbol('g', j); }
inline Key H(std::uint64_t j) { return Symbol('h', j); }
inline Key I(std::uint64_t j) { return Symbol('i', j); }
inline Key J(std::uint64_t j) { return Symbol('j', j); }
inline Key K(std::uint64_t j) { return Symbol('k', j); }
inline Key L(std::uint64_t j) { return Symbol('l', j); }
inline Key M(std::uint64_t j) { return Symbol('m', j); }
inline Key N(std::uint64_t j) { return Symbol('n', j); }
inline Key O(std::uint64_t j) { return Symbol('o', j); }
inline Key P(std::uint64_t j) { return Symbol('p', j); }
inline Key Q(std::uint64_t j) { return Symbol('q', j); }
inline Key R(std::uint64_t j) { return Symbol('r', j); }
inline Key S(std::uint64_t j) { return Symbol('s', j); }
inline Key T(std::uint64_t j) { return Symbol('t', j); }
inline Key U(std::uint64_t j) { return Symbol('u', j); }
inline Key V(std::uint64_t j) { return Symbol('v', j); }
inline Key W(std::uint64_t j) { return Symbol('w', j); }
inline Key X(std::uint64_t j) { return Symbol('x', j); }
inline Key Y(std::uint64_t j) { return Symbol('y', j); }
inline Key Z(std::uint64_t j) { return Symbol('z', j); }
}
```

```
/** Create a symbol key from a character and index, i.e. x5. */
inline Key symbol(unsigned char c, std::uint64_t j) { return (Key)Symbol(c,j); }

Key Symbol::key() const {
  if (j_ > indexMask) {
    boost::format msg("Symbol index is too large, j=%d, indexMask=%d");
    msg % j_ % indexMask;
    throw std::invalid_argument(msg.str());
  }
  Key key = (Key(c_) << indexBits) | j_;
  return key;
}
```

例如LIOSAM中的变量符号：

```
using gtsam::symbol_shorthand::X; // Pose3 (x,y,z,r,p,y)
using gtsam::symbol_shorthand::V; // Vel   (xdot,ydot,zdot)
using gtsam::symbol_shorthand::B; // Bias  (ax,ay,az,gx,gy,gz)

X(key); V(key); B(key); // key为新变量的序号
```

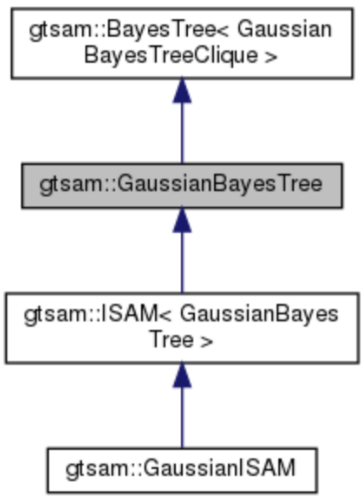
因子图稀疏性COLAMD

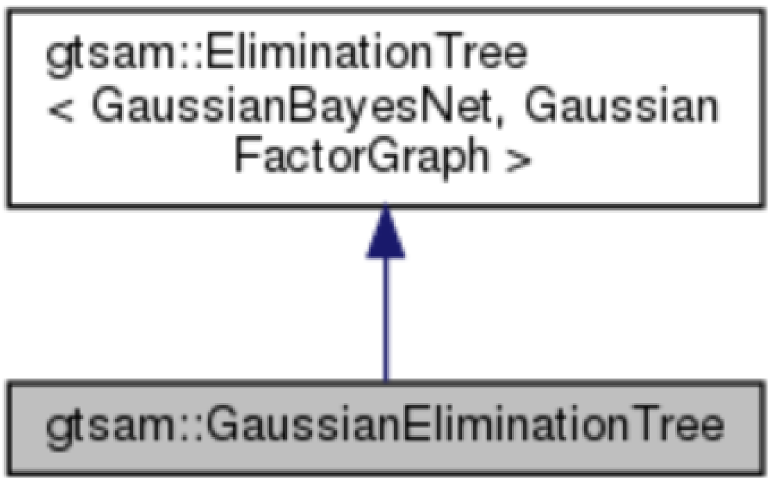
在第一次分享的时候，就提到GTSAM会使用COLAMD，近似查找最优的因子排序，让 $J^T J$ 保持很好的稀疏性，具体的代码是在这个头文件实现：

```
#include <gtsam/inference/Ordering.h>
```

5. linear & discrete & hybrid & symbolic

linear模块主要是线性高斯因子图模型，discrete是离散因子图模型，hybrid是混合因子图模型，状态有离散的也有连续的；symbolic是象征因子图模型(名称翻译有待商榷)，它们都继承于inference模块，只是应用的场景不一样，具体的头文件就不再赘述：


gtsam::BayesTree< Gaussian BayesTreeClique >
gtsam::GaussianBayesTree
gtsam::ISAM< GaussianBayes Tree >
gtsam::GaussianISAM


gtsam::EliminationTree < GaussianBayesNet, Gaussian FactorGraph >
gtsam::GaussianEliminationTree

这里重点讲一下linear模块的噪声模型 `NoiseModel.h`，它包含了下面6种噪声模型：

```
#include <gtsam/inference/NoiseModel.h>

// Forward declaration
class Gaussian; // 高斯噪声
class Diagonal; // 对角线噪声
class Constrained; // 约束噪声，对角线噪声的特殊版本，对角线上的噪声可以为0
class Isotropic; // 各向同性噪声，用于缩放对角线协方差
class Unit; // 单位方差噪声，所有维度上都是1.0
class RobustModel; // 鲁棒模型噪声，
```

LIOSAM的噪声模型为对角噪声：

```
    gtsam::noiseModel::Diagonal::shared_ptr priorPoseNoise; // 先验位置噪声
    gtsam::noiseModel::Diagonal::shared_ptr priorVelNoise;  // 先验速度噪声
    gtsam::noiseModel::Diagonal::shared_ptr priorBiasNoise; // 先验偏置噪声
    gtsam::noiseModel::Diagonal::shared_ptr correctionNoise;
    gtsam::noiseModel::Diagonal::shared_ptr correctionNoise2;
```

6. sam

这里就介绍了三种因子：

RangeFactor

```
// Serializable factor induced by a range measurement
#include <gtsam/sam/RangeFactor.h>

// between points:
typedef gtsam::RangeFactor<gtsam::Point2, gtsam::Point2> RangeFactor2;
typedef gtsam::RangeFactor<gtsam::Point3, gtsam::Point3> RangeFactor3;

// between pose and point:
typedef gtsam::RangeFactor<gtsam::Pose2, gtsam::Point2> RangeFactor2D;
typedef gtsam::RangeFactor<gtsam::Pose2, gtsam::Pose2> RangeFactorPose2;

// between poses:
typedef gtsam::RangeFactor<gtsam::Pose3, gtsam::Point3> RangeFactor3D;
typedef gtsam::RangeFactor<gtsam::Pose3, gtsam::Pose3> RangeFactorPose3;
```

BearingFactor

```
// Serializable factor induced by a bearing measurement
#include <gtsam/sam/BearingFactor.h>

typedef gtsam::BearingFactor<gtsam::Pose2, gtsam::Point2, gtsam::Rot2>
    BearingFactor2D;
typedef gtsam::BearingFactor<gtsam::Pose3, gtsam::Point3, gtsam::Unit3>
    BearingFactor3D;
typedef gtsam::BearingFactor<gtsam::Pose2, gtsam::Pose2, gtsam::Rot2>
    BearingFactorPose2;
```

BearingRangeFactor

```cpp
// a single factor contains both the bearing and the range to prevent
// handle to pair bearing and range factors
#include <gtsam/sam/BearingRangeFactor.h>

typedef gtsam::BearingRangeFactor<gtsam::Pose2, gtsam::Point2, gtsam::Rot2, doub
    BearingRangeFactor2D;
typedef gtsam::BearingRangeFactor<gtsam::Pose2, gtsam::Pose2, gtsam::Rot2, doubl
    BearingRangeFactorPose2;
typedef gtsam::BearingRangeFactor<gtsam::Pose3, gtsam::Point3, gtsam::Unit3, dou
    BearingRangeFactor3D;
typedef gtsam::BearingRangeFactor<gtsam::Pose3, gtsam::Pose3, gtsam::Unit3, doub
    BearingRangeFactorPose3;
```

7. Navigation

GPS因子：

```cpp
#include <gtsam/navigation/GPSFactor.h>

  /**
   * @brief Constructor from a measurement in a Cartesian frame.
   * Use GeographicLib to convert from geographic (latitude and longitude) coorc
   * @param key of the Pose3 variable that will be constrained
   * @param gpsIn measurement already in correct coordinates
   * @param model Gaussian noise model
   */
  GPSFactor(Key key, const Point3& gpsIn, const SharedNoiseModel& model) :
      Base(model, key), nT_(gpsIn) {
  }
```

LIOSAM中添加GPS因子(mapOptmization.cpp)

```cpp
gtsam::noiseModel::Diagonal::shared_ptr gps_noise = gtsam::noiseModel::Diagonal:
gtsam::GPSFactor gps_factor(cloudKeyPoses3D->size(), gtsam::Point3(gps_x, gps_y,
gtSAMgraph.add(gps_factor);
```

IMU预积分因子：

```
#include <gtsam/navigation/ImuFactor.h> // IMU预积分因子
#include <gtsam/navigation/ImuBias.h> // imu bias 模型
#include <gtsam/navigation/CombinedImuFactor.h> // IMUFactor + IMUBias
```

```
  /**
   * Add a single IMU measurement to the preintegration.
   * Both accelerometer and gyroscope measurements are taken to be in the sensor
   * frame and conversion to the body frame is handled by `body_P_sensor` in
   * `PreintegrationParams`.
   *
   * @param measuredAcc Measured acceleration (as given by the sensor)
   * @param measuredOmega Measured angular velocity (as given by the sensor)
   * @param dt Time interval between this and the last IMU measurement
   */
  void integrateMeasurement(const Vector3& measuredAcc,
      const Vector3& measuredOmega, const double dt) override;

  /// Add multiple measurements, in matrix columns
  void integrateMeasurements(const Matrix& measuredAccs, const Matrix& measuredO
                             const Matrix& dts);
```

LIOSAM中添加IMU因子和bias (imuPreintegration.cpp):

```
gtsam::PreintegratedImuMeasurements *imuIntegratorImu_;

imuIntegratorOpt_->integrateMeasurement(
    gtsam::Vector3(thisImu->linear_acceleration.x, thisImu->linear_acceleration.
    gtsam::Vector3(thisImu->angular_velocity.x, thisImu->angular_velocity.y, thi


// add imu factor to graph
// 将imu因子添加到因子图中
const gtsam::PreintegratedImuMeasurements &preint_imu = dynamic_cast<const gtsam
gtsam::ImuFactor imu_factor(X(key - 1), V(key - 1), X(key), V(key), B(key - 1),
graphFactors.add(imu_factor);
// add imu bias between factor
// 将imu偏置因子添加到因子图中
graphFactors.add(gtsam::BetweenFactor<gtsam::imuBias::ConstantBias>(B(key - 1),
                 gtsam::noiseModel::Diagonal::Sigmas(sqrt(imuIntegratorOpt_->delt
```

8. slam

slam模块有很多关于里程计的因子，下面罗列几种经常用到的：

`BetweenFactor`

```
// A class for a measurement predicted by "between(config[key1],config[key2])"
#include <gtsam/slam/BetweenFactor.h>

// 模版类
template<class VALUE>
class BetweenFactor: public NoiseModelFactorN<VALUE, VALUE>;

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

重投影因子 `ProjectionFactor`

```
// Reprojection of a LANDMARK to a 2D point.
#include <gtsam/slam/ProjectionFactor.h>
```

双目测量因子 `StereoFactor`

```
#include <gtsam/slam/StereoFactor.h>
```

三角化因子 `triangulationFactor`

```
#include <gtsam/slam/triangulationFactor.h>
```

本质矩阵因子 `EssentialMatrixFactor`

```
#include <gtsam/slam/EssentialMatrixFactor.h>
```

9. sfm

三维重建

10. nonLinear

先验因子 `PriorFactor`

```
#include <gtsam/nonlinear/PriorFactor.h>
```

LIOSAM添加先验因子(imuPreintegration.cpp):

```
// add pose
gtsam::PriorFactor<gtsam::Pose3> priorPose(X(0), prevPose_, updatedPoseNoise);
graphFactors.add(priorPose);
```

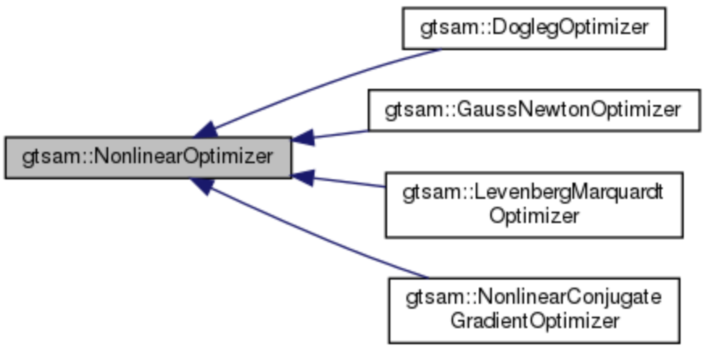
GraphValues

Values structure is a map from keys to values. It is used to specify the value of a bunch of variables in a factor graph.

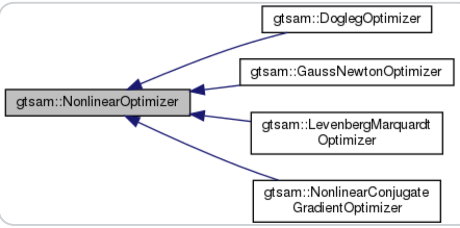
```
#include <gtsam/nonlinear/Values.h>

// 插入factor的值
/** Add a variable with the given j, throws KeyAlreadyExists<J> if j is already
void insert(Key j, const Value& val);

// 例子:
/*
gtsam::Pose3 prevPose_;  // 上一时刻估计imu的位姿信息
gtsam::Vector3 prevVel_; // 上一时刻估计imu的速度信息
gtsam::imuBias::ConstantBias prevBias_; // imu bias

graphValues.insert(X(0), prevPose_);
graphValues.insert(V(0), prevVel_);
graphValues.insert(B(0), prevBias_);
*/
```

非线性优化器

gtsam的非线性优化器有狗腿法 `DoglegOptimizer` 、 高斯牛顿法 `GaussNewtonOptimizer` 、 LM法 `LevenbergMarquardtOptimizer` 以及非线性共轭梯度法 `NonlinearConjugateGradientOptimizer` ，它们都继承于 `gtsam::NonlinearOptimizer` ，如下图所示：


gtsam::DoglegOptimizer
gtsam::GaussNewtonOptimizer
gtsam::NonlinearOptimizer
gtsam::LevenbergMarquardt Optimizer
gtsam::NonlinearConjugate GradientOptimizer

图片来源：


gtsam::DoglegOptimizer
gtsam::GaussNewtonOptimizer
gtsam::NonlinearOptimizer
gtsam::LevenbergMarquardt Optimizer
gtsam::NonlinearConjugate GradientOptimizer

https://gtsam-jlblanco-docs.readthedocs.io/en/latest/_static/doxygen/html/cla…

GTSAM: gtsam::NonlinearOptimizer Class Reference

GTSAM 4.0.2 C++ library for smoothing and mapping (SAM) gtsam NonlinearOptimizer Public Types | Protected Member Functions | Protected…

对应的头文件：

```
#include <gtsam/nonlinear/NonlinearOptimizer.h>

#include <gtsam/nonlinear/DoglegOptimizer.h>
#include <gtsam/nonlinear/GaussNewtonOptimizer.h>
#include <gtsam/nonlinear/LevenbergMarquardtOptimizer.h>
#include <gtsam/nonlinear/NonlinearConjugateGradientOptimizer.h>
```

下面例子来自：

https://blog.csdn.net/weixin_41394379/article/details/87967446

gtsam代码阅读_Lemonade__的博客-CSDN博客_gtsam+设置输出信息

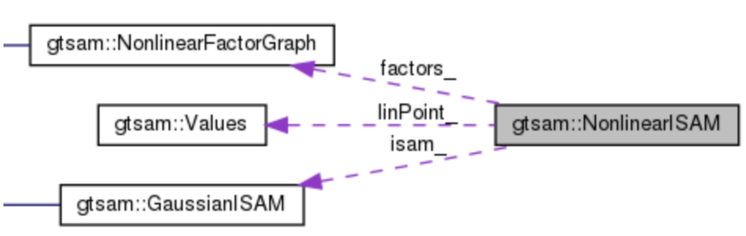
使用位姿图gtsam优化步骤:1.构建问题(因子图)建立因子图模型gtsam::NonlinearFactorGraph::shared_ptr graph(new gtsam::NonlinearFactorGraph);初始化值gtsam::Values::shared_ptr initial (new gtsam::Values);向initial中加入顶点(...

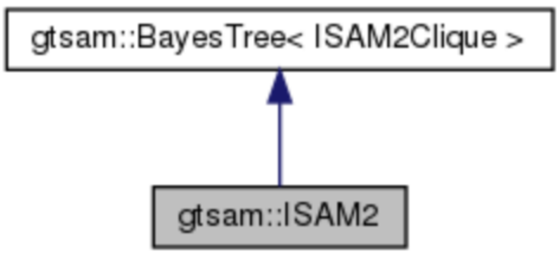
```
// 使用 LM 优化
gtsam::LevenbergMarquardtParams params_lm;//构建LM算法参数类(相当于g2o options)
params_lm.setVerbosity("ERROR");//设置输出信息
params_lm.setMaxIterations(20);//最大迭代次数
params_lm.setLinearSolverType("MULTIFRONTAL_QR");//分解算法
```

```
//构建下降算法(图,初值,参数)
gtsam::LevenbergMarquardtOptimizer optimizer_LM( graphWithPrior, *initial, param

//  GN优化
// gtsam::GaussNewtonParams params_gn;
// params_gn.setVerbosity("ERROR");
// params_gn.setMaxIterations(20);
// params_gn.setLinearSolverType("MULTIFRONTAL_QR");
// gtsam::GaussNewtonOptimizer optimizer ( graphWithPrior, *initial, params_gn )

gtsam::Values result = optimizer_LM.optimize();//开始优化
```

iSAM1和iSAM2

与上面非线性优化器不同的是，iSAM1和iSAM2不继承于 `gtsam::NonlinearOptimizer` ，下面是它们各自的继承关系：





图片来源：

https://gtsam-jlblanco-docs.readthedocs.io/en/latest/_static/doxygen/html/classgtsam_1_1NonlinearISAM.html

GTSAM: gtsam::NonlinearISAM Class Reference

GTSAM 4.0.2 C++ library for smoothing and mapping (SAM) gtsam NonlinearISAM Protected Attributes | List of all members gtsam::NonlinearISAM Class Reference #include < NonlinearISAM.h > Collaboration d

iSAM1的头文件：

```
#include <gtsam/nonlinear/NonlinearISAM.h>

// 构造函数
  /**
   * Periodically reorder and relinearize
   * @param reorderInterval is the number of updates between reorderings,
   *   0 never reorders (and is dangerous for memory consumption)
   *  1 (default) reorders every time, in worse case is batch every update
   *  typical values are 50 or 100
   */
  NonlinearISAM(int reorderInterval = 1,
    const GaussianFactorGraph::Eliminate& eliminationFunction = GaussianFactorGr
  reorderInterval_(reorderInterval), reorderCounter_(0), eliminationFunction_(el

// 更新状态
  /** Add new factors along with their initial linearization points */
  void update(const NonlinearFactorGraph& newFactors, const Values& initialValue

// 获取优化后的结果
  /** Return the current solution estimate */
  Values estimate() const;
```

参考例子：tests/testNonlinearISAM.cpp

iSAM2的头文件以及初始化的例子：

```
#include <gtsam/nonlinear/ISAM2.h>

// iSAM2初始化例子:
/*
gtsam::ISAM2Params optParameters;
optParameters.relinearizeThreshold = 0.1;
optParameters.relinearizeSkip = 1;
optimizer = gtsam::ISAM2(optParameters);

// 将因子图更新到isam2优化器中
optimizer.update(graphFactors, graphValues); // update有多个重载函数
// 获取优化后的结果
gtsam::Values result = optimizer.calculateEstimate();
*/
```

11. 总结

- 建立因子**图**模型并初始化(gtsam::NonlinearFactorGraph)
- 初始化值(gtsam::Values)
- 添加边(对应到因子图中的因子):构造因子,向图中添加因子(gtsam::NonlinearFactorGraph::add(..))
  构造因子所需参数:
    连接顶点类型;
    连接顶点的序号;
    测量值;
    高斯噪声模型,使用gtsam定义的信息矩阵形式构造(gtsam::noiseModel::Gaussian)

- 选择优化器(线性/非线性)(gtsam::ISAM2)
- 获取结果

LIOSAM中IMU预积分代码注释

```
class IMUPreintegration : public ParamServer
{
public:
    std::mutex mtx;

    ros::Subscriber subImu;        // imu信息订阅器
    ros::Subscriber subOdometry;   // 最终优化后的里程计增量信息(用来矫正imu的偏置)
    ros::Publisher pubImuOdometry; // 估计的imu里程计信息发布器(其实是通过imu估计的雷

    bool systemInitialized = false;

    gtsam::noiseModel::Diagonal::shared_ptr priorPoseNoise; // 先验位置噪声
    gtsam::noiseModel::Diagonal::shared_ptr priorVelNoise;  // 先验速度噪声
    gtsam::noiseModel::Diagonal::shared_ptr priorBiasNoise; // 先验偏置噪声
    gtsam::noiseModel::Diagonal::shared_ptr correctionNoise;
    gtsam::noiseModel::Diagonal::shared_ptr correctionNoise2;
    gtsam::Vector noiseModelBetweenBias;

    gtsam::PreintegratedImuMeasurements *imuIntegratorOpt_;
    gtsam::PreintegratedImuMeasurements *imuIntegratorImu_;

    std::deque<sensor_msgs::Imu> imuQueOpt;
    std::deque<sensor_msgs::Imu> imuQueImu;

    gtsam::Pose3 prevPose_;  // 上一时刻估计imu的位姿信息
    gtsam::Vector3 prevVel_; // 上一时刻估计imu的速度信息
    gtsam::NavState prevState_;
    gtsam::imuBias::ConstantBias prevBias_;

    gtsam::NavState prevStateOdom;
    gtsam::imuBias::ConstantBias prevBiasOdom;

    bool doneFirstOpt = false;
    double lastImuT_imu = -1;
    double lastImuT_opt = -1;

    gtsam::ISAM2 optimizer;
    gtsam::NonlinearFactorGraph graphFactors;
    gtsam::Values graphValues;

    const double delta_t = 0;

    int key = 1;

    gtsam::Pose3 imu2Lidar = gtsam::Pose3(gtsam::Rot3(1, 0, 0, 0), gtsam::Point3
    gtsam::Pose3 lidar2Imu = gtsam::Pose3(gtsam::Rot3(1, 0, 0, 0), gtsam::Point3

    IMUPreintegration()
    {
        subImu = nh.subscribe<sensor_msgs::Imu>(imuTopic, 2000, &IMUPreintegrati
        subOdometry = nh.subscribe<nav_msgs::Odometry>("lio_sam/mapping/odometry

        pubImuOdometry = nh.advertise<nav_msgs::Odometry>(odomTopic + "_incremen

        // 定义进行imu积分的imu传感器信息
        boost::shared_ptr<gtsam::PreintegrationParams> p = gtsam::Preintegration
        p->accelerometerCovariance = gtsam::Matrix33::Identity(3, 3) * pow(imuAc
        p->gyroscopeCovariance = gtsam::Matrix33::Identity(3, 3) * pow(imuGyrNoi
        p->integrationCovariance = gtsam::Matrix33::Identity(3, 3) * pow(1e-4, 2
        gtsam::imuBias::ConstantBias prior_imu_bias((gtsam::Vector(6) << 0, 0, 0

        priorPoseNoise = gtsam::noiseModel::Diagonal::Sigmas((gtsam::Vector(6) <
        priorVelNoise = gtsam::noiseModel::Isotropic::Sigma(3, 1e4);
        priorBiasNoise = gtsam::noiseModel::Isotropic::Sigma(6, 1e-3);
        correctionNoise = gtsam::noiseModel::Diagonal::Sigmas((gtsam::Vector(6)
        correctionNoise2 = gtsam::noiseModel::Diagonal::Sigmas((gtsam::Vector(6)
        noiseModelBetweenBias = (gtsam::Vector(6) << imuAccBiasN, imuAccBiasN, i

        // 根据上面的参数，定义两个imu预积分器，一个用于imu信息处理线程，一个用于优化线程
        imuIntegratorImu_ = new gtsam::PreintegratedImuMeasurements(p, prior_imu
        imuIntegratorOpt_ = new gtsam::PreintegratedImuMeasurements(p, prior_imu
    }

    void resetOptimization()
    {
        // 重置isam2优化器
        gtsam::ISAM2Params optParameters;
        optParameters.relinearizeThreshold = 0.1;
        optParameters.relinearizeSkip = 1;
        optimizer = gtsam::ISAM2(optParameters);

        // 重置初始化非线性因子图
        gtsam::NonlinearFactorGraph newGraphFactors;
        graphFactors = newGraphFactors;

        gtsam::Values NewGraphValues;
        graphValues = NewGraphValues;
    }

    void resetParams()
    {
        lastImuT_imu = -1;
        doneFirstOpt = false;
        systemInitialized = false;
    }

    void odometryHandler(const nav_msgs::Odometry::ConstPtr &odomMsg)
    {
        std::lock_guard<std::mutex> lock(mtx);

        double currentCorrectionTime = ROS_TIME(odomMsg);

        // make sure we have imu data to integrate
        // 确保我们已经进行过imu数据积分了
        if (imuQueOpt.empty())
            return;

        // 转换消息数据为gtsam的3d位姿信息
        float p_x = odomMsg->pose.pose.position.x;
        float p_y = odomMsg->pose.pose.position.y;
        float p_z = odomMsg->pose.pose.position.z;
        float r_x = odomMsg->pose.pose.orientation.x;
        float r_y = odomMsg->pose.pose.orientation.y;
        float r_z = odomMsg->pose.pose.orientation.z;
        float r_w = odomMsg->pose.pose.orientation.w;
        bool degenerate = (int)odomMsg->pose.covariance[0] == 1 ? true : false;
        gtsam::Pose3 lidarPose = gtsam::Pose3(gtsam::Rot3::Quaternion(r_w, r_x,

        // 0. initialize system
        // 矫正过程的初始化
        if (systemInitialized == false)
        {
            resetOptimization(); // 初始化isam2优化器及非线性因子图

            // pop old IMU message
            // 丢弃老的imu信息
            while (!imuQueOpt.empty())
            {
                if (ROS_TIME(&imuQueOpt.front()) < currentCorrectionTime - delta
                {
                    lastImuT_opt = ROS_TIME(&imuQueOpt.front());
                    imuQueOpt.pop_front();
                }
                else
                    break;
            }
            // initial pose
            // 通过最终优化过的雷达位姿初始化先验的位姿信息并添加到因子图中
            prevPose_ = lidarPose.compose(lidar2Imu);
            gtsam::PriorFactor<gtsam::Pose3> priorPose(X(0), prevPose_, priorPos
            graphFactors.add(priorPose);
            // initial velocity
            // 初始化先验速度信息为0并添加到因子图中
            prevVel_ = gtsam::Vector3(0, 0, 0);
            gtsam::PriorFactor<gtsam::Vector3> priorVel(V(0), prevVel_, priorVel
            graphFactors.add(priorVel);
            // initial bias
            // 初始化先验偏置信息为0并添加到因子图中
            prevBias_ = gtsam::imuBias::ConstantBias();
            gtsam::PriorFactor<gtsam::imuBias::ConstantBias> priorBias(B(0), pre
            graphFactors.add(priorBias);
            // add values
            // 设置变量的初始估计值
            graphValues.insert(X(0), prevPose_);
            graphValues.insert(V(0), prevVel_);
            graphValues.insert(B(0), prevBias_);
            // optimize once
            // 将因子图更新到isam2优化器中
            optimizer.update(graphFactors, graphValues);
            graphFactors.resize(0);
            graphValues.clear();

            imuIntegratorImu_->resetIntegrationAndSetBias(prevBias_);
            imuIntegratorOpt_->resetIntegrationAndSetBias(prevBias_);

            key = 1;
            systemInitialized = true;
            return;
        }

        // reset graph for speed
        // 当isam2规模太大时，进行边缘化，重置优化器和因子图； 在建图的时候没重置优化器
        if (key == 100)
        {
            // get updated noise before reset
            // 获取最新关键帧的协方差
            gtsam::noiseModel::Gaussian::shared_ptr updatedPoseNoise = gtsam::no
            gtsam::noiseModel::Gaussian::shared_ptr updatedVelNoise = gtsam::noi
            gtsam::noiseModel::Gaussian::shared_ptr updatedBiasNoise = gtsam::no
            // reset graph
            // 重置isam2优化器和因子图
            resetOptimization();
            // 按最新关键帧的协方差将位姿、速度、偏置因子添加到因子图中
            // add pose
            gtsam::PriorFactor<gtsam::Pose3> priorPose(X(0), prevPose_, updatedP
            graphFactors.add(priorPose);
            // add velocity
            gtsam::PriorFactor<gtsam::Vector3> priorVel(V(0), prevVel_, updatedV
            graphFactors.add(priorVel);
            // add bias
            gtsam::PriorFactor<gtsam::imuBias::ConstantBias> priorBias(B(0), pre
            graphFactors.add(priorBias);
            // add values
            // 并用最新关键帧的位姿、速度、偏置初始化对应的因子
            graphValues.insert(X(0), prevPose_);
            graphValues.insert(V(0), prevVel_);
            graphValues.insert(B(0), prevBias_);
            // optimize once
            // 并将最新初始化的因子图更新到重置的isam2优化器中
            optimizer.update(graphFactors, graphValues);
            graphFactors.resize(0);
            graphValues.clear();

            key = 1; // 重置关键帧数量
        }

        // 1. integrate imu data and optimize
        // 1. 预积分imu数据并进行优化
        // 当存在imu数据时
        while (!imuQueOpt.empty())
        {
            // pop and integrate imu data that is between two optimizations
            // 对相邻两次优化之间的imu帧进行积分，并移除
            sensor_msgs::Imu *thisImu = &imuQueOpt.front();
            double imuTime = ROS_TIME(thisImu);
            if (imuTime < currentCorrectionTime - delta_t)
            {
                double dt = (lastImuT_opt < 0) ? (1.0 / 500.0) : (imuTime - last
                imuIntegratorOpt_->integrateMeasurement(
                    gtsam::Vector3(thisImu->linear_acceleration.x, thisImu->line
                    gtsam::Vector3(thisImu->angular_velocity.x, thisImu->angular

                lastImuT_opt = imuTime;
                imuQueOpt.pop_front();
            }
            else
                break;
        }
        // add imu factor to graph
        // 将imu因子添加到因子图中
        const gtsam::PreintegratedImuMeasurements &preint_imu = dynamic_cast<con
        gtsam::ImuFactor imu_factor(X(key - 1), V(key - 1), X(key), V(key), B(ke
        graphFactors.add(imu_factor);
        // add imu bias between factor
        // 将imu偏置因子添加到因子图中
        graphFactors.add(gtsam::BetweenFactor<gtsam::imuBias::ConstantBias>(B(ke
                                                                             gtsa
        // add pose factor
        // 添加当前关键帧位姿因子
        gtsam::Pose3 curPose = lidarPose.compose(lidar2Imu);
        gtsam::PriorFactor<gtsam::Pose3> pose_factor(X(key), curPose, degenerate
        graphFactors.add(pose_factor);
        // insert predicted values
        // 设置当前关键帧位姿因子、速度因子和偏置因子的初始值
        gtsam::NavState propState_ = imuIntegratorOpt_->predict(prevState_, prev
        graphValues.insert(X(key), propState_.pose());
        graphValues.insert(V(key), propState_.v());
        graphValues.insert(B(key), prevBias_);
        // optimize
        // 将最新关键帧相关的因子图更新到isam2优化器中，并进行优化
        optimizer.update(graphFactors, graphValues);
        optimizer.update();
        graphFactors.resize(0);
        graphValues.clear();
        // Overwrite the beginning of the preintegration for the next step.
        // 获取当前关键帧的优化结果，并将结果置为先前值
        gtsam::Values result = optimizer.calculateEstimate();
        prevPose_ = result.at<gtsam::Pose3>(X(key));
        prevVel_ = result.at<gtsam::Vector3>(V(key));
        prevState_ = gtsam::NavState(prevPose_, prevVel_);
        prevBias_ = result.at<gtsam::imuBias::ConstantBias>(B(key));
        // Reset the optimization preintegration object.
        // 利用优化后的imu偏置信息重置imu预积分对象
        imuIntegratorOpt_->resetIntegrationAndSetBias(prevBias_);
        // check optimization
        // 对优化结果进行失败检测: 当速度和偏置太大时，则认为优化失败
        if (failureDetection(prevVel_, prevBias_))
        {
            resetParams();
            return;
        }

        // 2. after optiization, re-propagate imu odometry preintegration
        // 2. 优化后，重新对imu里程计进行预积分
        // 利用优化结果更新prev状态
        prevStateOdom = prevState_;
        prevBiasOdom = prevBias_;
        // first pop imu message older than current correction data
        // 丢弃早于矫正时间的imu帧
        double lastImuQT = -1;
        while (!imuQueImu.empty() && ROS_TIME(&imuQueImu.front()) < currentCorre
        {
            lastImuQT = ROS_TIME(&imuQueImu.front());
            imuQueImu.pop_front();
        }
        // repropogate
        // 重新进行预积分，从矫正时间开始
        if (!imuQueImu.empty())
        {
            // reset bias use the newly optimized bias
            // 将优化后的imu偏置信息更新到预积分器内
            imuIntegratorImu_->resetIntegrationAndSetBias(prevBiasOdom);
            // integrate imu message from the beginning of this optimization
            // 从矫正时间开始，对imu数据重新进行预积分
            for (int i = 0; i < (int)imuQueImu.size(); ++i)
            {
                sensor_msgs::Imu *thisImu = &imuQueImu[i];
                double imuTime = ROS_TIME(thisImu);
                double dt = (lastImuQT < 0) ? (1.0 / 500.0) : (imuTime - lastImu

                imuIntegratorImu_->integrateMeasurement(gtsam::Vector3(thisImu->
                                                        gtsam::Vector3(thisImu->
                lastImuQT = imuTime;
            }
        }

        ++key;
        doneFirstOpt = true;
    }
```