

从零开始一起学习SLAM | 理解图优化，一步步带你看懂g2o代码

小白：师兄师兄，最近我在看SLAM的优化算法，有种方法叫“图优化”，以前学习算法的时候还有一个优化方法叫“凸优化”，这两个不是一个东西吧？

师兄：哈哈，这个问题有意思，虽然它们中文发音一样，但是意思差别大着呢！我们来看看英文表达吧，图优化的英文是 graph optimization 或者 graph-based optimization，你看，它的“图”其实是数据结构中的graph。而凸优化的英文是 convex optimization，这里的“凸”其实是凸函数的意思，所以单从英文就能区分开它们。

小白：原来是这样，我看SLAM中图优化用的很多啊，我看了一下高博的书，还是迷迷糊糊的，求科普啊师兄

师兄：图优化真的蛮重要的，概念其实不负责，主要是编程稍微有点复杂。。

小白：不能同意更多。。，那个代码看的我一脸懵逼

图优化有什么优势？

师兄：按照惯例，我还是先说说图优化的背景吧。SLAM的后端一般分为两种处理方法，一种是以扩展卡尔曼滤波（EKF）为代表的滤波方法，一种是以图优化为代表的非线性优化方法。不过，目前SLAM研究的主流热点几乎都是基于图优化的。

小白：据我所知，滤波方法很早就有了，前人的研究也很深。为什么现在图优化变成了主流了？

师兄：你说的没错。滤波方法尤其是EKF方法，在SLAM发展很长的一段历史中一直占据主导地位，早期的大神们研究了各种各样的滤波器来改善滤波效果，那会入门SLAM，EKF是必须要掌握的。顺便总结下滤波方法的优缺点：

优点：在当时计算资源受限、待估计量比较简单的情况下，EKF为代表的滤波方法比较有效，经常用在激光SLAM中。

缺点：它的一个大缺点就是存储量和状态量是平方增长关系，因为存储的是协方差矩阵，因此不适合大型场景。而现在基于视觉的SLAM方案，路标点（特征点）数据很大，滤波方法根本吃不消，所以此时滤波的方法效率非常低。

小白：原来如此。那图优化在视觉SLAM中效率很高吗？

师兄：这个其实说来话长了。很久很久以前，其实就是不到十年前吧(感觉好像很久)，大家还都是用滤波方法，因为在图优化里，Bundle Adjustment（后面简称BA）起到了核心作用。但是那会SLAM的研究者们发现包含大量特征点和相机位姿的BA计算量其实很大，根本没办法实时。

小白：啊？后来发生了什么？（认真听故事ing）

师兄：后来SLAM研究者们发现了其实在视觉SLAM中，虽然包含大量特征点和相机位姿，但其实BA是稀疏的，稀疏的就好办了，就可以加速了啊！比较代表性的就是2009年，几个大神发表了自己的研究成果《SBA: A software package for generic sparse bundle adjustment》，而且计算机硬件发展也很快，因此基于图优化的视觉SLAM也可以实时了！

小白：厉害厉害！向大牛们致敬！

图优化是什么？

小白：图优化既然是主流，那我可以跳过滤波方法直接学习图优化吧，反正滤波方法也看不懂。。

师兄：额，图优化确实是主流，以后有需要你可以再去看滤波方法，那我们今天就只讲图优化好啦

小白：好滴，那问题来了，究竟什么是图优化啊？

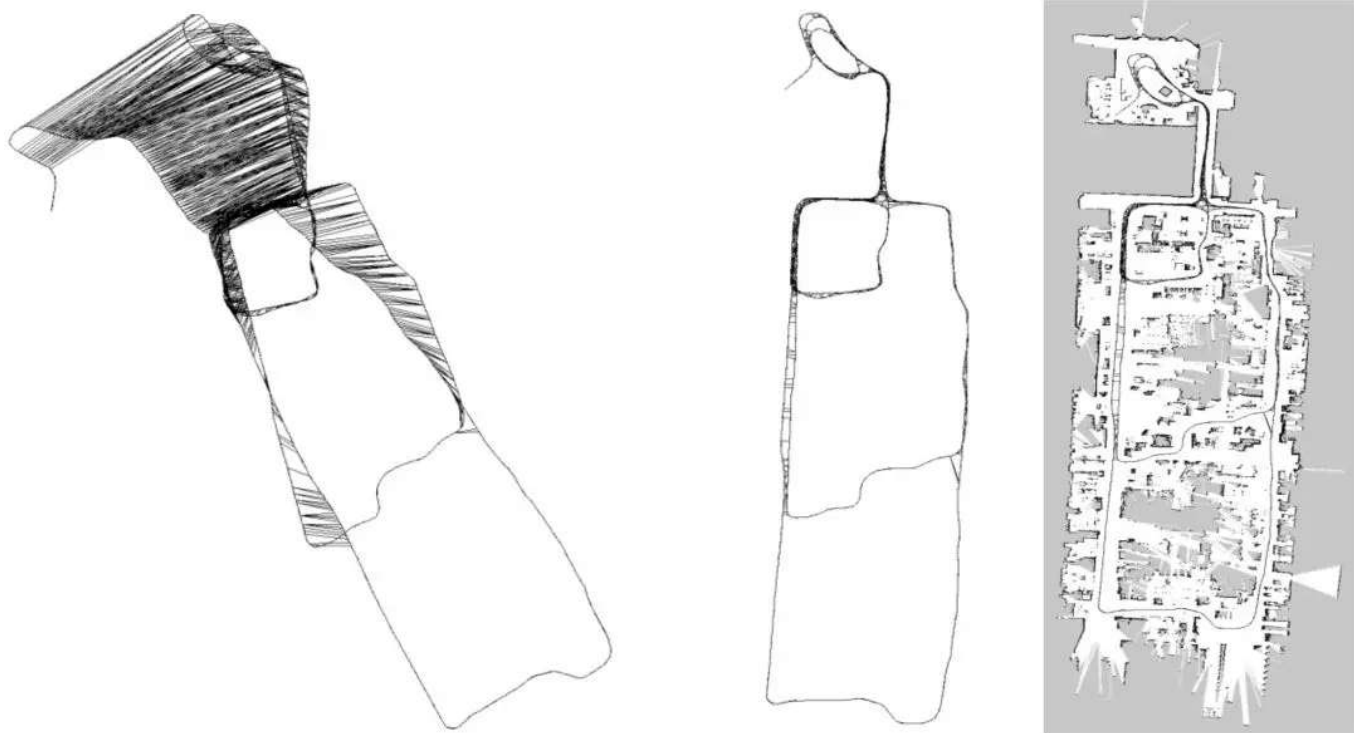
师兄：图优化里的图就是数据结构里的图，一个图由若干个顶点（vertex），以及连接这些顶点的边（edge）组成，给你举个例子

比如一个机器人在房屋里移动，它在某个时刻 t 的位姿（pose）就是一个顶点，这个也是待优化的变量。而位姿之间的关系就构成了一个边，比如时刻 t 和时刻 $t+1$ 之间的相对位姿变换矩阵就是边，边通常表示误差项。

在SLAM里，图优化一般分解为两个任务：

- 1、构建图。机器人位姿作为顶点，位姿间关系作为边。
- 2、优化图。调整机器人的位姿（顶点）来尽量满足边的约束，使得误差最小。

下面就是一个直观的例子。我们根据机器人位姿来作为图的顶点，这个位姿可以来自机器人的编码器，也可以是ICP匹配得到的，图的边就是位姿之间的关系。由于误差的存在，实际上机器人建立的地图是不准的，如下图左。我们通过设置边的约束，使得图优化向着满足边约束的方向优化，最后得到了一个优化后的地图（如下图中所示），它和真正的地图（下图右）非常接近。



小白：哇塞，这个图优化效果这么明显啊！刚开始误差那么大，最后都校正过来了

师兄：是啊，所以图优化在SLAM中举足轻重啊，一定得掌握！

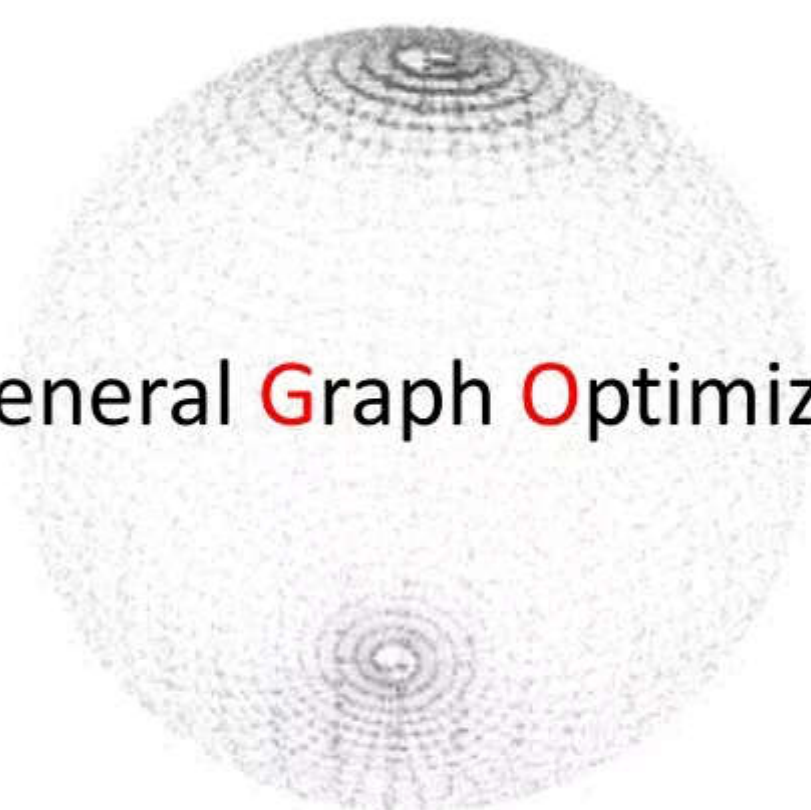
小白：好，有学习的动力了！我们开启编程模式吧！

先了解g2o 框架

师兄：前面我们简单介绍了图优化，你也看到了它的神通广大，那如何编程实现呢？

小白：对啊，有没有现成的库啊，我还只是个“调包侠”。。

师兄：这个必须有啊！在SLAM领域，基于图优化的一个用的非常广泛的库就是g2o，它是 General Graphic Optimization 的简称，是一个用来优化非线性误差函数的c++框架。这个库可以满足你调包侠的梦想~



g2o:General Graph Optimization

小白：哈哈，太好了，否则打死我也写不出来啊！那这个g2o怎么用呢？

师兄：我先说安装吧，其实g2o安装很简单，参考GitHub上官网：

<https://github.com/RainerKuemmerle/g2o>

按照步骤来安装就行了。需要注意的是安装之前确保电脑上已经安装好了第三方依赖。

小白：好的，这个看起来很好装。不过问题是，我看相关的代码，感觉很复杂啊，不知如何下手啊

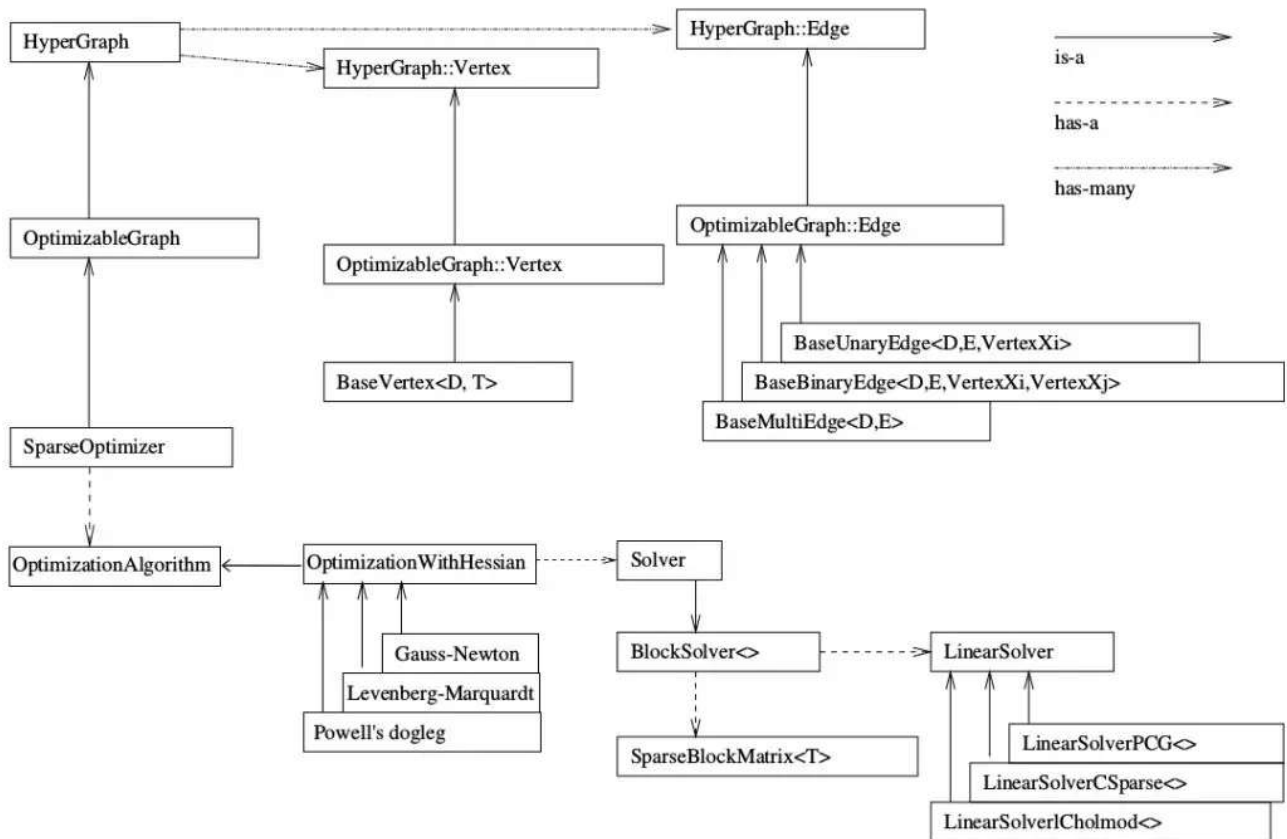
师兄：别急，第一次接触g2o，确实有这种感觉，而且官网文档写的也比较“不通俗不易懂”，不过如果你能捋顺了它的框架，再去看代码，应该很快能够入手了

小白：是的，先对框架了然于胸才行，不然即使能凑合看懂别人代码，自己也不会写啊！

师兄：嗯嗯，其实g2o帮助我们实现了很多内部的算法，只是在进行构造的时候，需要遵循一些规则，在我看来这是可以接受的，毕竟一个程序不可能满足所有的要求，因此在以后g2o的使用中还是应该多看多记，这样才能更好的使用这个库。

小白：记住了。养成记笔记的好习惯，还要多练习。

师兄：好，那我们首先看一下下面这个图，是g2o的基本框架结构。如果你查资料的话，你会在很多地方都能看到。看图的时候要注意箭头类型



1、图的核心

小白：师兄，这个图该从哪里开始看？感觉好多东西。。

师兄：如果你想要知道这个图中哪个最重要，就去看看箭头源头在哪里

小白：我看看。。。好像是最左侧的SparseOptimizer？

师兄：对的，SparseOptimizer是整个图的核心，我们注意右上角的 is-a 实心箭头，这个SparseOptimizer它是一个Optimizable Graph，从而也是一个超图（HyperGraph）。

小白：我去，师兄，怎么突然冒出来这么多奇怪的术语，都啥意思啊？

师兄：这个你不需要一个个弄懂，不然可能黄花菜都凉了。你先暂时只需要了解一下它们的名字，有些以后用不到，有些以后用到了再回看。目前如果遇到重要的我会具体解释。

小白：好。那下一步看哪里？

2、顶点和边

师兄：我们先来看上面的结构吧。注意看 has-many 箭头，你看这个超图包含了许多顶点（HyperGraph::Vertex）和边（HyperGraph::Edge）。而这些顶点继承自 Base Vertex，也就是OptimizableGraph::Vertex，而边可以继承自 BaseUnaryEdge（单边），BaseBinaryEdge（双边）或BaseMultiEdge（多边），它们都叫做OptimizableGraph::Edge

小白：头有点晕了，师兄

师兄：哈哈，不用一个个记，现阶段了解这些就行。顶点和边在编程中很重要的，关于顶点和边的定义我们以后会详细说的。下面我们来看底部的结构。

小白：嗯嗯，知道啦！

3、配置SparseOptimizer的优化算法和求解器

师兄：你看下面，整个图的核心SparseOptimizer 包含一个优化算法（OptimizationAlgorithm）的对象。OptimizationAlgorithm是通过OptimizationWithHessian 来实现的。其中迭代策略可以从 Gauss-Newton（高斯牛顿法，简称GN），Levenberg-Marquardt（简称LM法），Powell's dogleg 三者中间选择一个（我们常用的是GN和LM）

小白：GN和LM就是我们以前讲过的非线性优化方法中常用的两种吧

师兄：是的，如果不了解的话具体看《从零开始学习「张氏相机标定法」（四）优化算法前传》《从零开始学习「张氏相机标定法」（五）优化算法正传》这两篇文章。

4、如何求解

师兄：那么如何求解呢？OptimizationWithHessian 内部包含一个求解器（Solver），这个Solver实际是由一个BlockSolver组成的。这个BlockSolver有两个部分，一个是SparseBlockMatrix，用于计算稀疏的雅可比和Hessian矩阵；一个是线性方程的求解器（LinearSolver），它用于计算迭代过程中最关键的一步 $H\Delta x = -b$ ，LinearSolver有几种方法可以选择：PCG, CSparse, Choldmod，具体定义后面会介绍

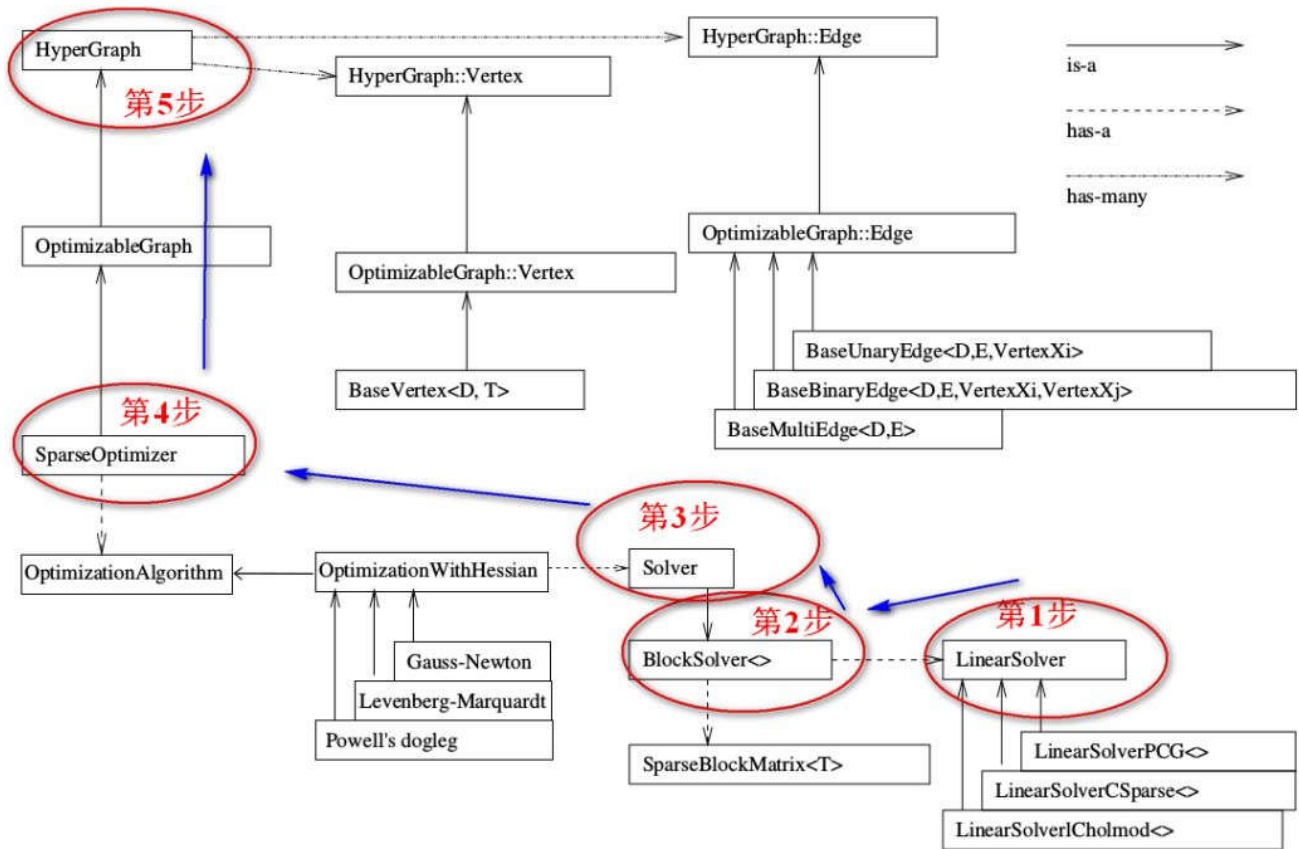
到此，就是上面图的一个简单理解。

一步步带你看懂g2o编程流程

小白：师兄，看完了我也不知道编程时具体怎么编呢！

师兄：我正好要说这个。首先这里需要说一下，我们梳理是从顶层到底层，但是编程实现时需要反过来，像建房子一样，从底层开始搭建框架一直到顶层。g2o的整个框架就是按照下图中我标的

这个顺序来写的。



高博在十四讲中g2o求解曲线参数的例子来说明，源代码地址

https://github.com/gaoxiang12/slambook/edit/master/ch6/g2o_curve_fitting/main.cpp

为了方便理解，我重新加了注释。如下所示，

```

typedef g2o::BlockSolver< g2o::BlockSolverTraits<3,1> > Block; // 每个误差项优化变量维度

// 第1步：创建一个线性求解器LinearSolver
Block::LinearSolverType* linearSolver = new g2o::LinearSolverDense<Block::PoseMatrixType>

// 第2步：创建BlockSolver。并用上面定义的线性求解器初始化
Block* solver_ptr = new Block( linearSolver );

// 第3步：创建总求解器solver。并从GN, LM, DogLeg 中选一个，再用上述块求解器BlockSolver初始化
g2o::OptimizationAlgorithmLevenberg* solver = new g2o::OptimizationAlgorithmLevenberg( s

// 第4步：创建终极大boss 稀疏优化器 (SparseOptimizer)
g2o::SparseOptimizer optimizer; // 图模型
optimizer.setAlgorithm( solver ); // 设置求解器
optimizer.setVerbose( true ); // 打开调试输出

// 第5步：定义图的顶点和边。并添加到SparseOptimizer中
CurveFittingVertex* v = new CurveFittingVertex(); //往图中增加顶点
v->setEstimate( Eigen::Vector3d(0,0,0) );
v->setId(0);
optimizer.addVertex( v );

```

```

for ( int i=0; i<N; i++ )    // 往图中增加边
{
    CurveFittingEdge* edge = new CurveFittingEdge( x_data[i] );
    edge->setId(i);
    edge->setVertex( 0, v );           // 设置连接的顶点
    edge->setMeasurement( y_data[i] ); // 观测数值
    edge->setInformation( Eigen::Matrix<double,1,1>::Identity()*1/(w_sigma*w_sigma) ); //
    optimizer.addEdge( edge );
}

// 第6步：设置优化参数，开始执行优化
optimizer.initializeOptimization();
optimizer.optimize(100);

```

(左右滑动试试)

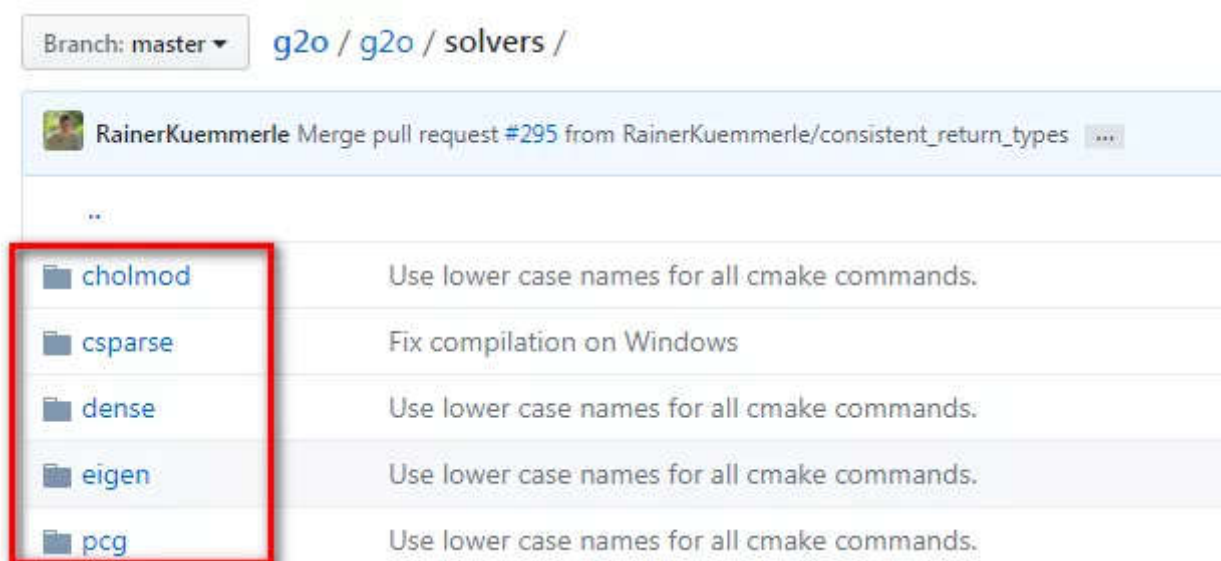
结合上面的流程图和代码。下面一步步解释具体步骤。

1、创建一个线性求解器LinearSolver

我们要求的增量方程的形式是： $H \Delta X = -b$ ，通常情况下想到的方法就是直接求逆，也就是 $\Delta X = -H.\text{inv} * b$ 。看起来好像很简单，但这有个前提，就是H的维度较小，此时只需要矩阵的求逆就能解决问题。但是当H的维度较大时，矩阵求逆变得很困难，求解问题也变得很复杂。

小白：那有什么办法吗？

师兄：办法肯定是有的。此时我们就需要一些特殊的方法对矩阵进行求逆，你看下图是GitHub上g2o相关部分的代码



如果你点进去看，可以分别查看每个方法的解释，如果不想挨个点进去看，看看下面我的总结就行了

LinearSolverCholmod : 使用sparse cholesky分解法。继承自LinearSolverCCS
LinearSolverCSparse: 使用CSparse法。继承自LinearSolverCCS
LinearSolverPCG : 使用preconditioned conjugate gradient 法, 继承自LinearSolver
LinearSolverDense : 使用dense cholesky分解法。继承自LinearSolver
LinearSolverEigen: 依赖项只有eigen, 使用eigen中sparse Cholesky 求解, 因此编译好后可以方便的

2、创建BlockSolver。并用上面定义的线性求解器初始化。

BlockSolver 内部包含 LinearSolver, 用上面我们定义的线性求解器LinearSolver来初始化。它的定义在如下文件夹内:

g2o/g2o/core/block_solver.h

你点进去会发现 BlockSolver有两种定义方式

一种是指定的固定变量的solver, 我们来看一下定义

```
using BlockSolverPL = BlockSolver< BlockSolverTraits<p, l> >;
```

其中p代表pose的维度 (注意一定是流形manifold下的最小表示) , l表示landmark的维度

另一种是可变尺寸的solver, 定义如下

```
using BlockSolverX = BlockSolverPL<Eigen::Dynamic, Eigen::Dynamic>;
```

小白: 为何会有可变尺寸的solver呢?

师兄: 这是因为在某些应用场景, 我们的Pose和Landmark在程序开始时并不能确定, 那么此时这个块状求解器就没办法固定变量, 此时使用这个可变尺寸的solver, 所有的参数都在中间过程中被确定

另外你看block_solver.h的最后, 预定义了比较常用的几种类型, 如下所示:

BlockSolver_6_3 : 表示pose 是6维, 观测点是3维。用于3D SLAM中的BA
BlockSolver_7_3: 在BlockSolver_6_3 的基础上多了一个scale
BlockSolver_3_2: 表示pose 是3维, 观测点是2维

以后遇到了知道这些数字是什么意思就行了

3、创建总求解器solver。并从GN, LM, DogLeg 中选一个, 再用上述块求解器BlockSolver初始化

我们来看 g2o/g2o/core/ 目录下，发现 Solver 的优化方法有三种：分别是高斯牛顿（GaussNewton）法，LM（Levenberg–Marquardt）法、Dogleg法，如下图所示，也和前面的图相匹配

optimization_algorithm_dogleg.cpp	- added ability to change the floating point precision (float/double)...
optimization_algorithm_dogleg.h	- added ability to change the floating point precision (float/double)...
optimization_algorithm_factory.cpp	Improve listSolvers output
optimization_algorithm_factory.h	some documentation of the factory
optimization_algorithm_gauss_newt...	- added ability to change the floating point precision (float/double)...
optimization_algorithm_gauss_newt...	Simplified Memory Ownership
optimization_algorithm levenberg.c...	- added ability to change the floating point precision (float/double)...
optimization_algorithm levenberg.h	- added ability to change the floating point precision (float/double)...
optimization_algorithm_property.h	change license of most files to BSD
optimization_algorithm_with_hessia...	- added ability to change the floating point precision (float/double)...
optimization_algorithm_with_hessia...	- added ability to change the floating point precision (float/double)...

小白：师兄，上图最后那个OptimizationAlgorithmWithHessian 是干嘛的？

师兄：你点进去 GN、LM、Doglet 算法内部，会发现他们都继承自同一个类：OptimizationWithHessian，如下图所示，这也和我们最前面那个图是相符的

```

namespace g2o {

/**
 * \brief Implementation of the Gauss Newton Algorithm
 */
class G2O_CORE_API OptimizationAlgorithmGaussNewton : public OptimizationAlgorithmWithHessian
{
public:
/**
 * construct the Gauss Newton algorithm, which use the given Solver for solving the
 * linearized system.
 */

namespace g2o {

/**
 * \brief Implementation of the Levenberg Algorithm
 */
class G2O_CORE_API OptimizationAlgorithmLevenberg : public OptimizationAlgorithmWithHessian
{
public:
/**
 * construct the Levenberg algorithm, which will use the given Solver for solving the
 * linearized system.
 */

namespace g2o {

class BlockSolverBase;

/**
 * \brief Implementation of Powell's Dogleg Algorithm
 */
class G2O_CORE_API OptimizationAlgorithmDogleg : public OptimizationAlgorithmWithHessian
{
public:
/** \brief type of the step to take */
enum {

```

然后，我们点进去看 `OptimizationAlgorithmWithHessian`，发现它又继承自 `OptimizationAlgorithm`，这也和前面的相符

```

namespace g2o {

class Solver;

/**
 * \brief Base for solvers operating on the approximated Hessian, e.g., Gauss-Newton, Levenberg
 */
class G2O_CORE_API OptimizationAlgorithmWithHessian : public OptimizationAlgorithm
{
public:
explicit OptimizationAlgorithmWithHessian(Solver& solver);

```

总之，在该阶段，我们可以选则三种方法：

```
g2o::OptimizationAlgorithmGaussNewton
g2o::OptimizationAlgorithmLevenberg
g2o::OptimizationAlgorithmDogleg
```

4、创建终极大boss 稀疏优化器（SparseOptimizer），并用已定义求解器作为求解方法。

创建稀疏优化器

```
g2o::SparseOptimizer optimizer;
```

用前面定义好的求解器作为求解方法：

```
SparseOptimizer::setAlgorithm(OptimizationAlgorithm* algorithm)
```

其中setVerbose是设置优化过程输出信息用的

```
SparseOptimizer::setVerbose(bool verbose)
```

不信我们来看一下它的定义

```
if (verbose()) {
    computeActiveErrors();
    cerr << "iteration= -1\t chi2= " << activeChi2()
        << "\t time= 0.0"
        << "\t cumTime= 0.0"
        << "\t (using initial guess from " << costFunction.name() << ")" << endl;
}
```

5、定义图的顶点和边。并添加到SparseOptimizer中。

这部分比较复杂，我们下一次再介绍。

6、设置优化参数，开始执行优化。

设置SparseOptimizer的初始化、迭代次数、保存结果等。

初始化

```
SparseOptimizer::initializeOptimization(HyperGraph::EdgeSet& eset)
```

设置迭代次数，然后就开始执行图优化了。

```
SparseOptimizer::optimize(int iterations, bool online)
```

小白：终于搞明白g2o流程了！谢谢师兄！必须给你个「好看」啊！

注：以上内容部分参考了如下文章，感谢原作者：

<https://www.jianshu.com/p/e16ffb5b265d>

<https://blog.csdn.net/heyijia0327/article/details/47686523>

讨论

推荐阅读